Master Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of control engineering

Design and implementation of an autonomous chess-playing robot

Bc. David Pařil

Supervisor: Ing. Martin Hlinovský, Ph.D.
August 2023

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Pařil David**	Personal ID number:	**483565**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Control Engineering**		
Study program:	**Cybernetics and Robotics**		

II. Master’s thesis details

Master’s thesis title in English:

Design and implementation of an autonomous chess-playing robot

Master’s thesis title in Czech:

Návrh a realizace autonomního robota hrajícího šachy

Guidelines:

1. Familiarize yourself with computer vision methods for detecting the state of a chessboard from an RGB monocular camera
mounted on a robot.
2. Design and build a robotic manipulator playing chess using LEGO® EV3 for promotional purposes of the faculty.
3. Design and implement a manipulator control system to win over your opponent in a game of chess.
4. Create a graphical user interface to allow setting up the game and displaying its state in real time.

Bibliography / sources:

[1] M. Piškorec, N. Antulov-Fantulin, J. Ćurić, O. Dragoljević, V. Ivanac and L. Karlović, "Computer vision system for the chess
game reconstruction," 2011 Proceedings of the 34th International Convention MIPRO, Opatija, Croatia, 2011, pp. 870-876.
[2] P. Kołosowski, A. Wolniakowski and K. Miatliuk, "Collaborative Robot System for Playing Chess," 2020 International Conference Mechatronic Systems and Materials (MSM), Bialystok, Poland, 2020, pp. 1-6, doi: 10.1109/MSM49833.2020.9202398.
[3] A. T. -Y. Chen and K. I. -K. Wang, "Computer vision based chess playing capabilities for the Baxter humanoid robot," 2016
2nd International Conference on Control, Automation and Robotics (ICCAR), Hong Kong, China, 2016, pp. 11-14, doi: 10.1109/ICCAR.2016.7486689.

Name and workplace of master’s thesis supervisor:

Ing. Martin Hlinovský, Ph.D. Department of Control Engineering FEE

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **30.01.2023** Deadline for master's thesis submission: **14.08.2023**

Assignment valid until: **22.09.2024**

Ing. Martin Hlinovský, Ph.D.	prof. Ing. Michael Šebek, DrSc.	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature	Head of department’s signature	Dean’s signature

III. Assignment receipt

The student acknowledges that the master’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master’s thesis, the author must state the names of consultants and include a list of references.

.
Date of assignment receipt — Student’s signature

Acknowledgements

I would like to thank my master's thesis advisor, Ing. Martin Hlinovský, Ph.D., for his guidance and the time he invested in me. I also wish to thank Ing. Tomáš Fridrichovský, Ph.D., for providing access to the 3D scanners at the Faculty of Mechanical Engineering. Last but not least, I wish to express my deep appreciation to my family for their unwavering support throughout my studies at CTU and in my personal life.

Declaration

Abstract

This thesis introduces an autonomous chess-playing robotic manipulator utilizing computer vision methodologies for chessboard detection. We delve into the design of the robot, capable of playing a game of chess against human opponents, and discuss various implementation approaches. We introduce a robust detection system for the chessboard, the individual pieces and the player's move. This system empowers the robot to operate under various lighting conditions, utilizing a single non-static RGB camera mounted on the robot's frame. Furthermore, we outline the methodologies used to generate synthetic and real-world datasets essential for machine learning processes.

Keywords: autonomous robot, chess, computer vision, chess piece recognition, synthetic data generation

Supervisor: Ing. Martin Hlinovský, Ph.D.
ČVUT v Praze,
Fakulta elektrotechnická,
Katedra řídicí techniky,
Karlovo náměstí 13,
121 35 Praha 2

Abstrakt

Tato diplomová práce představuje autonomní šachový robotický manipulátor využívající metod počítačového vidění k detekci šachovnice. Věnujeme se návrhu robota schopného hrát šachy proti lidským oponentům a diskutujeme různé přístupy implementace. Představujeme spolehlivý detekční systém šachovnice, jednotlivých figurek a tahů hráče. Tento systém umožňuje robotu fungovat za různých světelných podmínek s využitím jedné nestatické RGB kamery umístěné na rámu robota. Dále popisujeme metody použité k vytváření syntetických a reálných datasetů nezbytných pro procesy strojového učení.

Klíčová slova: autonomní robot, šachy, počítačové vidění, rozpoznávání šachových figurek, generování syntetických dat

Překlad názvu: Návrh a realizace autonomního robota hrajícího šachy

Contents

Figures

Tables

Chapter 1

Introduction

Chess, an ancient strategic board game with origins tracing back over a millennium, has always been a testament to human intellect, strategy, and the ability to foresee the opponent's move. Two players face off on an 8x8 grid chessboard, each commanding 16 pieces. The game offers 6 distinct types of pieces, each with unique movement capabilities and strategic value. The ultimate objective of both players is to dismantle the opponent's defences and expose their king to inescapable danger.

Over the centuries, chess evolved from a game cherished by nobility to a universally recognized sport played by the masses. Competitions emerged in which players optimize every move of the vast strategic landscape. The game's complexity made it an object of intensive study and admiration.

With the dawn of the computer age in the mid-20th century, the exploration of chess strategies began transitioning to the digital realm. For computer scientists, the game of chess became a benchmark for the performance of algorithms in reasoning, pattern recognition, and decision-making. Naturally, humanity began to ponder whether a computer algorithm could ever match or even surpass human intellect and defeat a human player in a game of chess.

This groundbreaking moment came to pass in 1997 when IBM's supercomputer, Deep Blue, bested the former world chess champion, Garry Kasparov [1]. This match marked a pivotal moment in human history when a machine crafted by humans triumphed over the human intellect, symbolically defeating its creator. Since then, advances in computing and chess algorithms enabled people to challenge a virtual chess grandmaster on their personal mobile devices.

However, humanity's defeat in the game of chess did not mark the end of our pursuit of knowledge. In the world of computer chess, a new frontier emerged, transitioning from the realm of virtual chessboards and computer screens to tangible reality. The objective now extends beyond crafting an algorithm that generates moves to designing a platform that physically executes these moves, interacting with both its environment and the human opponent. This introduces new challenges from the realms of robotics and computer vision, which are the main focus of this thesis.

1.1 Previous work

Chess has long held humanity's fascination, not only as a game of intellect but also in the domain of automation. In the late 18th century, the world was introduced to the "Mechanical Turk", a supposed automaton created by Wolfgang von Kempelen. This chess-playing machine wowed audiences across Europe, though it concealed a human player secretly directing its moves. A notable step forward came with "El Ajedrecista" in the early 20th century. Designed by Leonardo Torres Quevedo, it was one of the first true automated chess players, capable of managing a specific endgame scenario.

The drive to develop autonomous chess-playing machines intensified with the dawn of the computer and robotics era. Various solutions emerged, each showcasing diverse levels of complexity and success. However, as highlighted in [2], "A number of existing implementations of chess-playing robots exist in the literature. However, there are often a number of significant constraints that simplify the problem greatly.". Any superfluous conditions inevitably limit the system's applicability in real-world settings or increase its deployment cost. The most common simplifications found in various implementations include:

1. Fixed Chessboard and Camera: Many systems utilize a permanently fixed camera and chessboard setup [3, 4, 5], often eliminating the need for chessboard detection. In some implementations, the user must calibrate the system at the start of each game by manually marking the chessboard's corners in the graphical user interface [6]. This setup allows the use of basic image recognition methods like image subtraction, significantly simplifying the task of chess piece detection and hand detection.

2. Top-Down Camera View: This perspective is favoured by many [3, 4, 5] as it simplifies board recognition. Pieces do not occlude the gridlines

nor the edges of the chessboard. The pieces are not distorted and do not overlap, making identification based on attributes like diameter or colour more straightforward.

3. Controlled Environment: Some systems need to operate in a controlled environment to mitigate issues caused by improper lighting conditions [3, 4, 7]. Shadows from spectators, players and chess pieces can cast sharp shadows that further complicate the image recognition task.

4. Modified chessboards: Expensive digitization boards, or boards with integrated sensor arrays, can be used to detect or even classify pieces on board reliably [8]. To ad computer vision recognition, some systems utilize contrasting chessboards with pieces of distinct colours [4, 7].

5. Modified chess pieces: Special piece shapes and pieces with embedded magnets are also employed for easier robotic grasping.

6. Advanced sensors: Depth cameras, stereo cameras or LIDARs have been employed in some systems [9].

Numerous studies have delved into the challenges of autonomous chess-playing robots, but often, they concentrate on particular facets of this complex system. One of the frequent focal points is chessboard state recognition through a camera, which, under unrestricted conditions, is still considered a very challenging computer vision task, and a universal and reliable solution is still yet to be found.

In [2], a humanoid robot capable of playing chess against human opponents is introduced. They utilize Canny edge detection for piece detection, which leads to issues with sharp shadows and doesn't allow for identifying the type of piece. They highlight the limitations of using a collaborative robot, which requires approximately 45-90 seconds to execute a move. Their implementation is one of the few that doesn't require a fixed position of the chessboard relative to the robot.

In studies [10] and [11], the authors focus on creating a synthetic chess dataset. Both papers train neural networks on the generated dataset, achieving excellent chess piece recognition capabilities.

The study [12] introduces a robust corner-based method for detecting partially occluded checkerboard patterns. The presented system can identify the chessboard even in images with low resolution or lens distortion. In study [13], a system coupling line-based and corner-based chessboard detection is introduced, which can reliably detect chessboard even at a lower-angle view.

In conclusion, while there have been significant advancements in the domain of chess-playing robots and chessboard recognition, many systems still rely on various simplifications to achieve their objectives. The challenge remains to develop a system that can operate in real-world settings without compromising on the authenticity of the chess-playing experience.

1.2 Introduction of the implemented system

This thesis introduces an autonomous robotic system developed to play chess against a human opponent. Initially, this project was conceived as a technological demo to represent CTU at public events. For this reason, we set ourselves strict limiting conditions:

- For image detection, we utilize a monocular RGB camera mounted directly to the robot's frame. This setup confronts us with challenges of image distortion, autofocus failures, over-exposure, and high dynamic range (HDR) effects.

- We employ a standard, unmodified, wooden chessboard with low contrast between chess pieces and squares, significantly complicating all computer vision-related tasks. Furthermore, the chessboard lacks well-defined edges between the individual squares.

- To ensure the system's reliability in dynamic environments, we've implemented a robust system for recognizing the opponent's moves. This employs a combination of chessboard, chess piece, and hand detection algorithms. The environment in which the robot operates is very challenging from the computer vision standpoint as it often contains moving shadows and flashing lights.

- From a robot-human interaction perspective, it was desirable that the robot's design did not obstruct the view of the chessboard. Furthermore, we aimed for the robot to autonomously recognize the progression and completion of a player's move, enabling fluid gameplay without the need for explicit user signalization. We've also integrated a graphical user interface (GUI) that displays the game's current status.

The implemented system is segmented into three main modules: the chess-playing robot, the control computer, and the graphical user interface. A robust communication layer connects all modules, ensuring the system's functionality even with unstable connectivity. Our robotic system is illustrated in Figure 1.1.

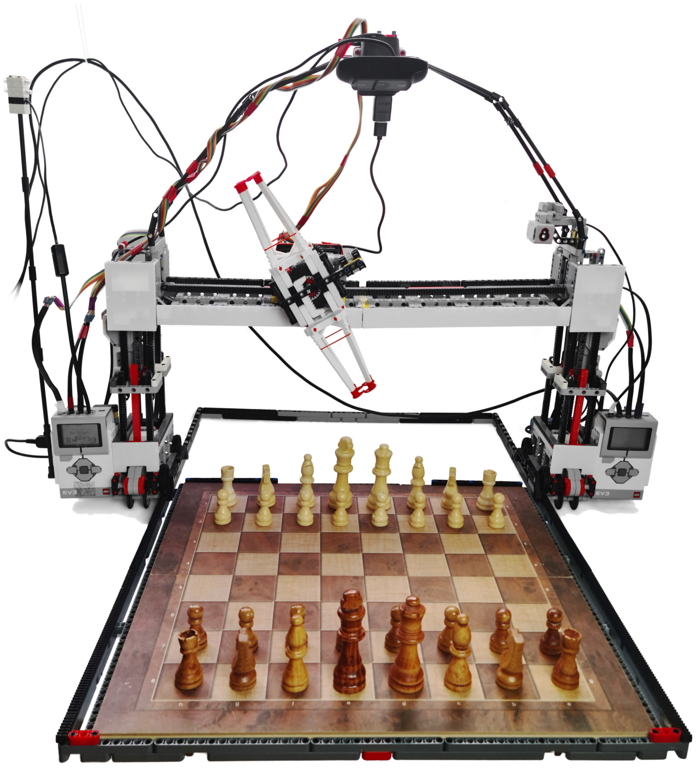


Figure 1.1: The autonomous chess-playing robot introduced in this thesis, along with the used wooden chess set.

1.3 Outline of the work

Chapter 2 delves deeper into the robot's design, elucidating its hardware components, their integration, and the rationale behind each decision. We introduce a unique dual-rotational gripper solution and familiarize the reader with the challenges encountered during the development.

Chapter 3 provides an in-depth discussion of computer vision techniques for detecting the chessboard's state from camera imagery. In Section 3.1, we introduce methods for chessboard detection, while the subsequent Section 3.2 explores dataset generation techniques and piece recognition strategies. Section 3.3, dedicated to player move detection, integrates the gathered insights and presents a system capable of flawlessly tracking an entire chess game. Section 3.4 enhances the system's robustness by implementing a hand-detection feature for end-of-move recognition.

Chapter 4 details our implementation. We introduce the system as a whole, discussing its components and the communication among them. We introduce our implemented collision-free motion planning technique, the algorithms for the control computer, and the algorithm powering the robot itself. This chapter concludes by presenting the graphical user interface we developed.

In Chapter 5, we evaluate the outcomes of this thesis and showcase the robot in action.

Finally, in Chapter 6, we summarize our efforts and set the direction for future research and enhancements.

All the results presented in this work were obtained using a computer with the following specifications (unless stated otherwise): AMD Ryzen 7 6800HS CPU, 32GB DDR5 RAM, NVIDIA GeForce RTX 3070 Ti (120W laptop edition) GPU, Pop!_OS 22.04.

Chapter 2

Design, construction and hardware implementation

The design and construction of the Chess-Playing Robot is a complex task that starts with carefully evaluating different mechanical structures (Figure 2.1). We have to consider the key functional requirements - the robot must be capable of identifying pieces and executing precise movements to pick and place pieces while navigating around obstacles (other pieces) on the chessboard. We have already set several requirements for this thesis in chapter 1.2. Adding to the complexity, the thesis supervisor has outlined two specific constraints: the robot must be constructed primarily from parts included in the LEGO® Mindstorms® Education 45544/45560 sets, and the robot must be capable of holding two chess pieces simultaneously, simulating human-like captures.

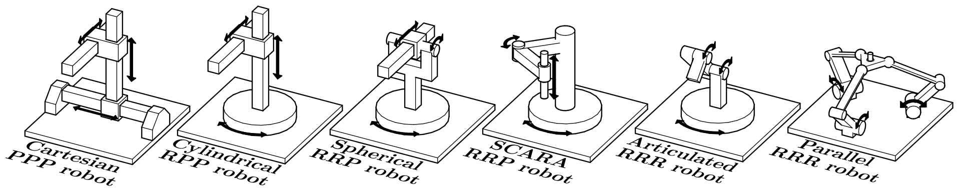


Figure 2.1: Overview of most common mechanical structure types for robotic manipulators. (P = prismatic joint, R = revolute joint)

These constraints impose unique challenges on the design process. The use of ABS plastic, the primary material of the robot's body, introduces a considerable degree of flexibility, which reduces the movement accuracy significantly. The robot's structure is further constrained by the limited power and precision of the available actuators. Due to these reasons, most of the mechanical structures depicted in Figure 2.1 cannot be used.

Ultimately, we have chosen a cartesian coordinate robot with a rotary dual gripper head. With its 4 degrees of freedom (DOF), this design allows us to fulfil all the specified requirements. The cartesian coordinate robot efficiently distributes the load by supporting the robot's frame from two points, ensuring precise movements with straightforward kinematics and optimal work envelope at the expense of increased motor load on the Z-axis. Figure 2.2 shows an overview of the robot's final design.

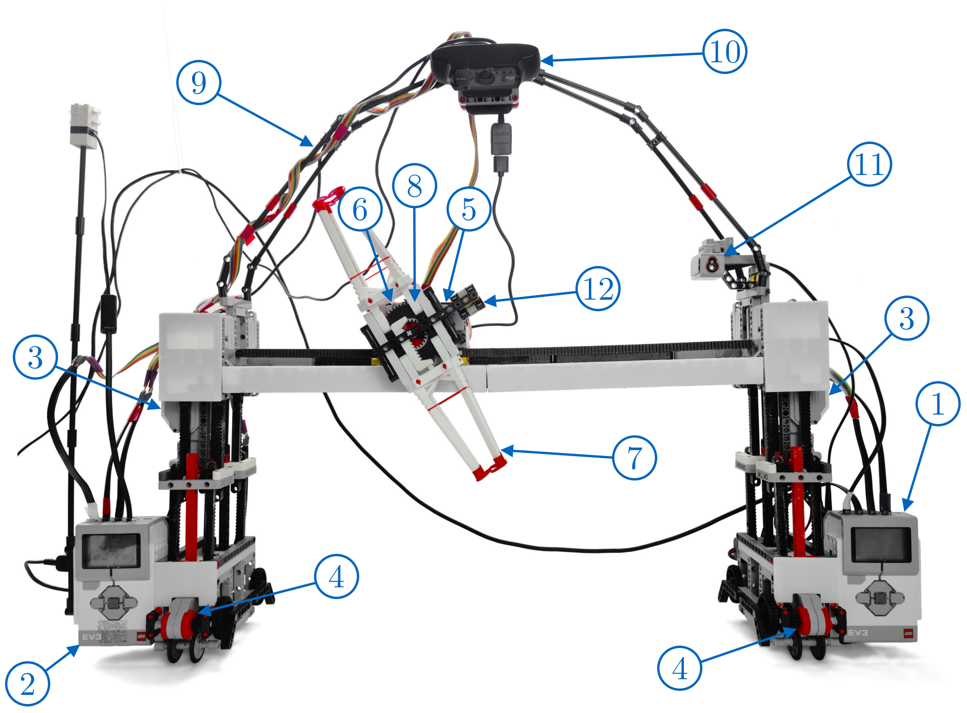


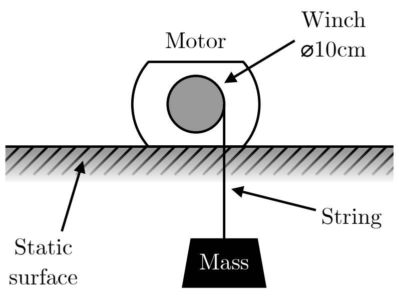
Figure 2.2: The final design of the chess-playing robot.

The robot is equipped with two identical MCUs (Microcontroller Units) featuring ARM9 CPU clocked at 300 MHz, 64/16 MB of RAM/flash, Bluetooth and USB connectivity. The right unit ① functions as the master, while the left one ② serves as the client. In total, the robot is outfitted with seven motors, comprising four large motors (part number 45502) and three medium motors (part number 45503). The large motors, under the control of the master MCU, facilitate movement along the Z ③ and Y ④ axes. The client MCU manages the medium motors responsible for head movements along the X-axis ⑤, rotation ⑥ of the grippers ⑦, and control of both grippers' opening and closing ⑧. An elevated camera bridge ⑨ provides sufficient distance between the camera ⑩ and the chessboard, compensating for camera's limited horizontal field of view (HFOV) of 60°. All of the robot's DOFs are capable of sensorless auto-calibration except for the rotation of grippers, which uses a light sensor ⑪ in combination with a reflective target ⑫ to self-calibrate.

2.1 Actuators and transmission

This thesis uses two types of motors consisting of three fundamental components: a DC motor, a gearbox, and an optical encoder. The gearbox of the large motor comprises a series of 7 plastic gears, resulting in a noticeable backlash, which is often exacerbated by wear and tear of the components over time. In contrast, the medium motor features a two-stage planetary gearbox with minimal backlash, and based on our observations, it is less prone to wear and tear. Additionally, the medium motor is equipped with thermal protection, which shuts down the motor in the event of overheating.

The manufacturer claims that the large motor should achieve a torque of 20 $N \cdot cm$, while the medium motor should have a torque of 8 $N \cdot cm$. Additionally, the large motor is expected to attain a rotational speed of 160-170 rpm (960-1020 deg/s), and the small motor should reach 240-250 rpm (1440-1500 deg/s). We validated the torque and speed values using a simplified measurement setup and the motor's integrated optical encoder - both the measurement setup and the measurement results are shown in Figure 2.3.



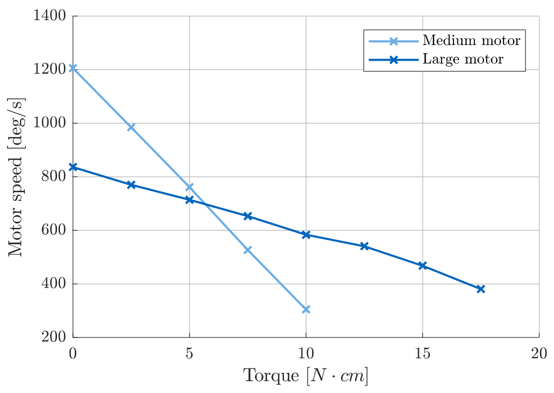


Figure 2.3: The measurement setup used to measure the torque and speed of large/medium motors (left) and the results of the measurement (right).

The measurement results show that neither motor achieves the manufacturer's claimed speed. The medium motor reached a torque of 10 Ncm before stalling, surpassing the specified values. However, the large motor only achieved about 18 Ncm before stalling. It's important to mention that the large motors frequently fail, leading to multiple replacements during the robot's development. On the other hand, the medium motor did not require any replacements but experienced overheating during dataset creation (Chapter 3.2.2).

To convert rotational motion into translational movement along the X, Y, and Z axes, we employ a gear rack and pinion mechanism. The Z-axis utilizes two gear racks around the pinion, forming a highly compact lifting mechanism that can elevate the robot by up to 13 cm (Figure 2.4). The gear ratios on all translational axes were carefully chosen to achieve comparable speeds while optimally utilizing the torque-speed characteristics of the large motors (see Table 2.1).

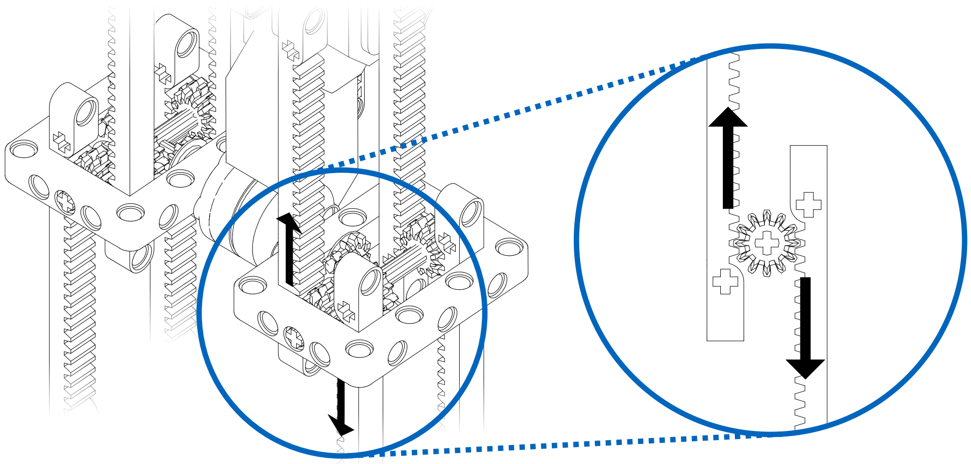


Figure 2.4: The lifting mechanisms of the Z-axis consists of 4 pinions and 8 gear racks powered by the large motor.

Axis	Movement per rotation of motor	Number of motors
X-axis	5.0944 cm/rot	1 Medium
Y-axis	6.8774 cm/rot	2 Large
Z-axis	4.5850 cm/rot	2 Large
R-axis	0.1429 rot/rot	1 Medium
Gripper	200 deg to open/close	1 Medium

Table 2.1: An overview of the motors and gearboxes used on each axis, where axis R represents the 4th degree of freedom of the robot, i.e., the rotation of the gripper pair:

2.2 Rotary head and the grippers

Most robotic manipulators start their capture move by removing the player's piece from the chessboard [3, 2, 4, 8]. This is because these manipulators cannot grasp two pieces simultaneously; thus, they must free up space for the attacking piece first. This motion confuses most players since a human player typically picks up their own piece first and then exchanges it with the opponent's piece, removing it from the playing surface. As our goal is to create a robot for human-robot interaction, we designed a unique rotating double gripper to address this issue (Figure 2.5).

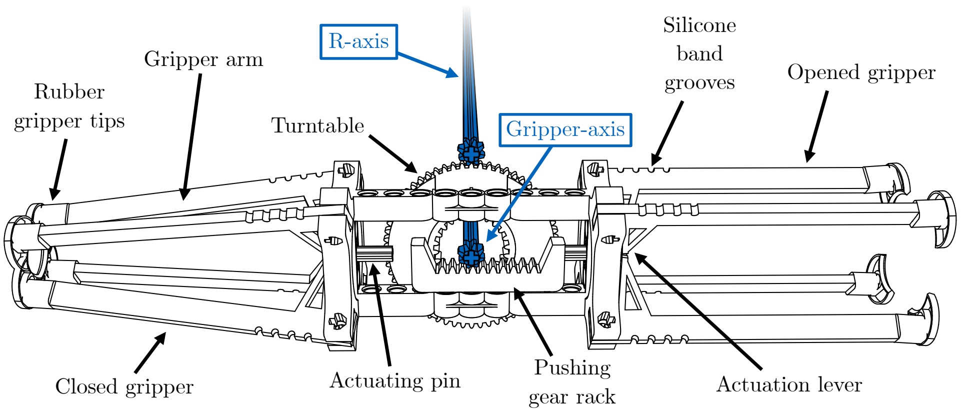


Figure 2.5: A simplified sketch of the rotary dual gripper with descriptions of its individual components.

The majority of the dual gripper was 3D-printed. Two medium motors power the dual gripper. One of the motors controls the R-axis, enabling unlimited rotation of the entire dual gripper around its axis. The second motor moves the pushing gear rack, activating one of the actuating pins that open the gripper arms using the actuation lever. The movement of the gripper motor must be synchronized with the movement of the R-axis. We utilize inverted logic to control both grippers with just one motor, meaning the gripper motor doesn't close the gripper; instead, it opens it. This trick relies on the fact that both grippers never need to be open simultaneously during the game. The silicone band is responsible for closing the gripper and thus gripping the chess pieces. At the end of each gripper arm, there is a rubber gripper tip (see Figure 2.6).

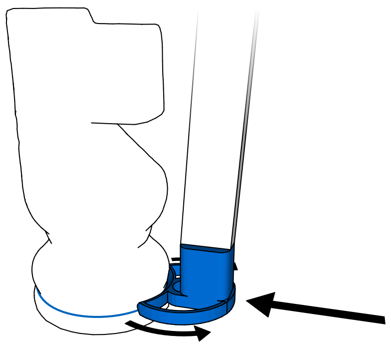


Figure 2.6: Illustration of the movement of flexible gripper tips while gripping a chess piece (knight).

The 3D-printed flexible gripper tips utilize the base collar of the chess pieces (a circular groove at the base of the piece) present on most chess sets, including the standardized Staunton chess set. This base collar allows for a secure attachment of the chess piece. The design of these gripper tips enables adjusting their radius to accommodate chess pieces with different base diameters (the chess pieces in the set we are using vary significantly in size). If a set without the base collar needs to be used, the rubber gripper tips can be easily replaced with foam cushions that surround the chess piece from all sides, ensuring a stable grip.

Chapter 3

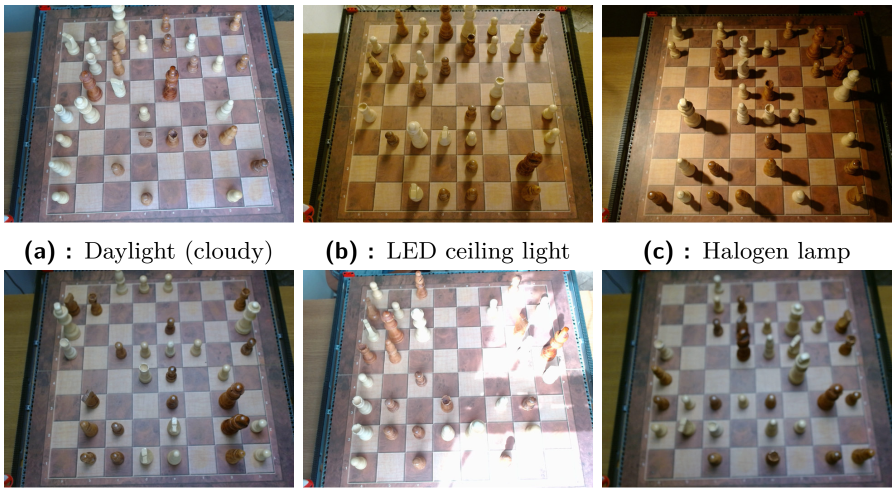
Computer vision

In the realm of robotics, computer vision plays a pivotal role in empowering robots to interact with their environment and make intelligent decisions based on visual information. This chapter delves into the critical aspects of vision-based perception and recognition, enabling our robot to perceive and interpret the chessboard pieces and their positions.

We have limited ourselves to using a monocular camera for chessboard state recognition. This cost-effective solution provides a substantial amount of information at the expense of more challenging data processing. The Logitech C920 HD PRO USB camera was chosen for its compatibility with older drivers in the MCU units. The webcam features automatic exposure and focusing capabilities and provides RGB images of acceptable quality at a resolution of 640x480 pixels. However, the camera lacks HDR support, and during our testing, the images were often overexposed or poorly focused.

In contrast to other research works, which often utilize statically mounted and calibrated top-down cameras above the chessboard, significantly simplifying image processing, our approach involves directly attaching the camera to the robot to avoid obstructing the player's view. Additionally, we do not use special high-contrast chessboards or coloured pieces commonly used in other works. In this thesis, we use a wooden chessboard with pieces that have nearly the same colour as the chessboard itself, making image recognition significantly more challenging. Figure 3.1 shows the output images from the camera under various lighting conditions.

Many previous research works deploy chess robots only in a static labora-



(a) : Daylight (cloudy) **(b) :** LED ceiling light **(c) :** Halogen lamp

(d) : Studio light **(e) :** Daylight (sunny) **(f) :** Bad autofocus

Figure 3.1: Examples of camera output under different lighting conditions.

tory environment. This thesis introduces image recognition that functions effectively under a wide range of lighting conditions, as the robot is intended to operate in exhibitions and public events where coloured/flashing lights and large crowds of people are expected. Shadows, reflections, and uneven lighting are well known for posing a significant challenge in computer vision applications.

In general, the computer vision pipeline for chessboard state recognition consists of three main components: chessboard detector, chess piece recognition, and move detector. In the following text, we will delve into the detailed implementation of these components. Additionally, we implement hand detection, enabling us to detect the end of a player's move without any explicit input from their side. Figure 3.2 shows an overview of the implemented pipeline for move detection from a chessboard image.

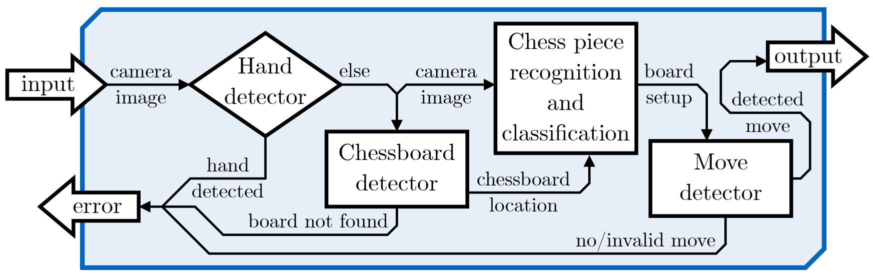


Figure 3.2: Overview of the implemented pipeline for move detection from a chessboard image.

This thesis utilizes the open-source library OpenCV [14] for standard image processing methods and the YOLO (You Only Look Once) [15, 16] neural network for the implementation of the beforementioned pipeline.

3.1 Chessboard detection

This chapter discusses different approaches and the intricacies of developing a robust algorithm for chessboard detection in an image. Successful chessboard detection forms the critical first step in the computer vision pipeline of our robot. At the end of this chapter, we introduce two implemented algorithms for board detection and discuss their positive and negative attributes.

Based on the camera mounting, we can categorize the approaches for chessboard detection as follows:

1. Static calibrated camera: The most straightforward solution; often requires a static laboratory environment and/or a fixed frame around the chessboard. Typically, the camera is placed directly above the chessboard. With proper calibration, the chessboard detection step can be skipped entirely. This setup allows for the utilization of the background subtraction method, which significantly simplifies the steps of chess piece detection and hand detection.

2. Close-range dynamic camera: A significantly more challenging scenario when the chessboard occupies most of the image, but its precise position is unknown. The chessboard must be correctly detected in each frame (typically searching for its four outermost corners). The output of this step (a cropped image of the chessboard) resembles the output from a static top-down camera from point 1. However, due to distortion, methods from point 1 (e.g., background subtraction) cannot be reliably used (see Figure 3.3). There are various approaches to finding the chessboard, which we explore in later parts of this chapter. These methods often leverage the geometric properties of the chessboard in combination with classical image processing techniques; neural networks are rarely used here. The robot addressed in this thesis falls into this category.

3. Distant dynamic camera: The position of the chessboard in the image is unknown, and it occupies a small portion of the image. Image processing approaches from point 2 typically fail here due to numerous distracting objects around the chessboard. Region of interest (ROI) is commonly used to tackle this issue, defining a sub-area of the image where the chessboard is sought. Techniques like probability heat maps of the chessboard occurrence can be employed to locate the ROI. ROI is obtained from the heat map by selecting the tetragonal region with the highest

probability values [17]. Another alternative for finding the ROI is the utilisation of neural networks. Once the ROI is located, algorithms from point 2 can be applied.

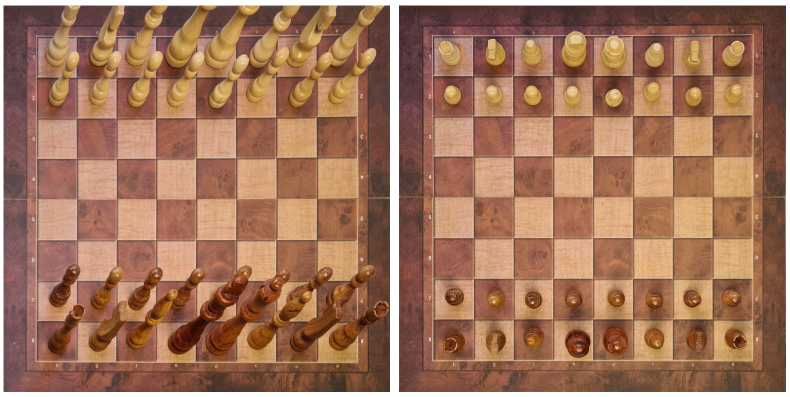


Figure 3.3: The distortion effects in a processed image of an angled camera (left) compared to a cropped image from a static top-down camera (right).

The two most commonly used approaches for detecting the chessboard in the camera image are:

1. Corner-based detector: These algorithms are popular for their speed and straightforward implementation. They have found applications in [13, 7]. They are robust to camera distortion. The algorithm's output is a list of 2D points representing all detected corners in the image. Points corresponding to the chessboard can be filtered based on the geometric properties of the chessboard. In [9], they utilise a template matching with the RANSAC (Random sample consensus) algorithm to search for a 9x9 grid of chessboard points in the 2D point cloud. This algorithm can be sensitive to occlusions, so it is best used with a top-down camera.

2. Line-based detector: These algorithms leverage the most distinctive visual feature of the chessboard - the lines separating individual squares. Thanks to this, these detectors are highly effective, even in cases of partial occlusion. As lines contain more information than points (corner-based detectors), these algorithms are generally more accurate and less sensitive to poor lighting conditions. However, it is still necessary to filter out detected lines that do not belong to the chessboard. Line-based detectors are sensitive to image distortion, high noise and require high-contrast transitions between chessboard squares. The parameters need to be set correctly for the specific configuration. Line-based detectors have found applications in [17, 10, 13, 2, 11].

Many other approaches have been tested with mixed results. In [3], they find edges in the image and then segment it into 65 regions (64 squares + border). This calibration can only be done before the game as no pieces can be on the chessboard - this approach requires a static camera. In [5], they utilise the Ramer-Douglas-Peucker algorithm to detect rectangles in the image. The chessboard is obtained by selecting the largest rectangle with the desired geometric properties. However, the authors note that the algorithm is sensitive to poor lighting conditions. In [18], they employ the flood-fill method - by random sampling, they attempt to find all empty squares of the central 4x8 grid in the initial configuration. The positions of the squares with pieces are extrapolated. A static camera is required.

3.1.1 Algorithm 1: Gradient descent chessboard detector

In this section, we will describe the implemented gradient descent algorithm for chessboard detection and evaluate its performance on various test cases. The proposed approach employs a gradient descent technique to iteratively refine the positioning of a grayscale mask over the chessboard image based on a calculated correspondence score. Figure 3.4 shows a visualisation of the algorithm's functionality.

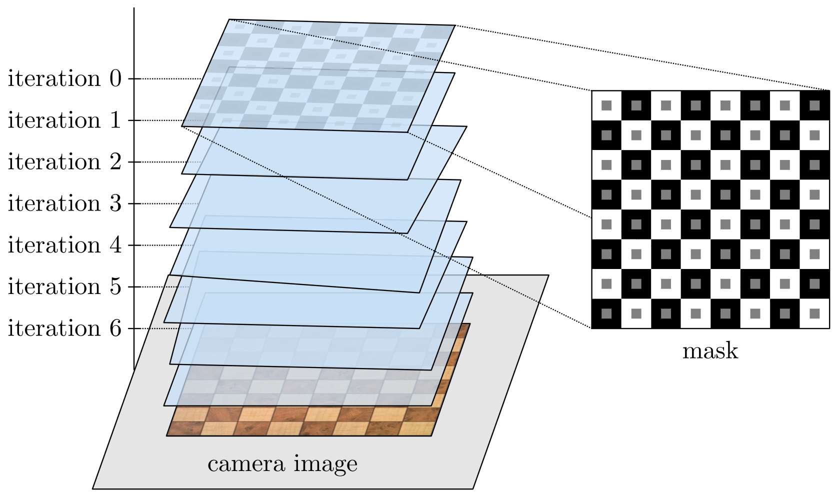


Figure 3.4: A visualisation of the algorithm's functionality.

The algorithm starts with a mask in pre-defined initial corners (their positions can be estimated or provided as input of the function). For faster iterative descent, a variable step length was implemented (in our case, the initial value is $step = 16$ pixels). Each iteration finds the optimal movement direction for each mask corner (the direction returning the highest score). If no corner movement results in a higher score than the current position of the mask, the step size is halved. The algorithm terminates when the current position of the mask yields the highest score, and the step size is already 1. The pseudo-code for this algorithm is shown in Algorithm 1.

Algorithm 1: The gradient descent chessboard detector

```
Data: input image, optionally initial $corners$ values
Result: coordinates of chessboard corners
$img \leftarrow$ Normalize0to1(GrayScale(input.png));
$mask \leftarrow$ Normalize0to1(mask.png);
$corners \leftarrow$ InitCornerValues();
$step \leftarrow 16$;
while $true$ do
    $goodCorners \leftarrow [true, true, true, true]$;
    for $c \in corners$ do
        for $d \in Directions()$ do
            // Get $corners$ with $c$ moved in $d$ direction
            $testCorners \leftarrow$ GetTestCorners($corners, c, d$);
            $score \leftarrow$ GetScore($img, mask, testCorners$);
            if $IsNewBestScore(score)$ then
                $goodCorners[c] \leftarrow false$;
                $corners \leftarrow testCorners$;
        end
    end
    if $All(goodCorners)$ then
        if $step = 1)$ then
            return $corners$
        $step \leftarrow step/2$;
end
```

The mask is an image of an ideal chessboard consisting of an 8x8 matrix of alternating black and white squares. Each square contains a grey area in the centre, which aids the final iterations of the algorithm (precisely aligning with chessboard lines) by giving less weight to the centre of the squares, where pieces are typically located (see Figure 3.4). The correspondence score is calculated using Equation 3.1, where the $transMask$ represents the mask after the perspective transformation (Equations 3.2 and 3.3). The denominator of the Equation 3.1 normalizes the score for differently-sized $transMask$s.

$$\text{score} = \frac{\sum\limits_{x,y \in \text{transMask}} \text{transMask}[x,y] \cdot \text{img}[x,y]}{\sum\limits_{x,y \in \text{transMask}} 1} \tag{3.1}$$

$$C_i \cdot \begin{bmatrix} x_{dst_i} \\ y_{dst_i} \\ 1 \end{bmatrix} = \begin{bmatrix} M_{11} & M_{12} & M_{13} \\ M_{21} & M_{22} & M_{23} \\ M_{31} & M_{32} & M_{33} \end{bmatrix} \begin{bmatrix} x_{src_i} \\ y_{src_i} \\ 1 \end{bmatrix} for\ i \in corners \tag{3.2}$$

$$\text{transMask}(x, y) = \text{mask}\left(\frac{M_{11}x + M_{12}y + M_{13}}{M_{31}x + M_{32}y + M_{33}}, \frac{M_{21}x + M_{22}y + M_{23}}{M_{31}x + M_{32}y + M_{33}}\right) \tag{3.3}$$

This approach utilizes the geometric properties of the chessboard, but unlike corner-based and line-based detectors, it leverages the entire chessboard area as a source of information. This enables it to correctly detect chessboard corners even in cases where other algorithms fail. However, its gradient-based approach is susceptible to local minima of the correspondence score, which can lead to algorithm failure. Unilateral lighting of the chessboard can negatively affect the score computation due to the nonlinearity of the camera's light perception. In Figure 3.5, a series of inputs and outputs of the discussed algorithm can be observed together with the number of iterations and the processing time of each image.

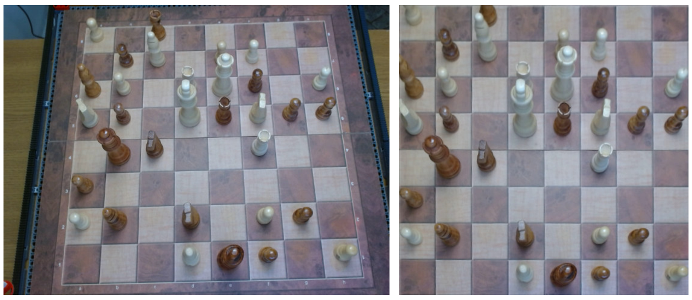
As can be observed in Figures 3.5b and 3.5c, the algorithm can handle challenging situations, such as overexposure and image blur. The main advantage of this approach is its ability to cope with occlusions in the image, as demonstrated in Figure 3.6. On the other hand, Figure 3.5d illustrates the algorithm's failure due to the unilateral lighting of the chessboard.

Despite the aforementioned limitations, we have successfully played several games of chess using this algorithm without any perception errors (laboratory environment). Future work could enhance the algorithm by incorporating methods to mitigate local-minima issues in the gradient descent approach. Additionally, calibrating the camera's light perception could address problems caused by unilateral lighting conditions.

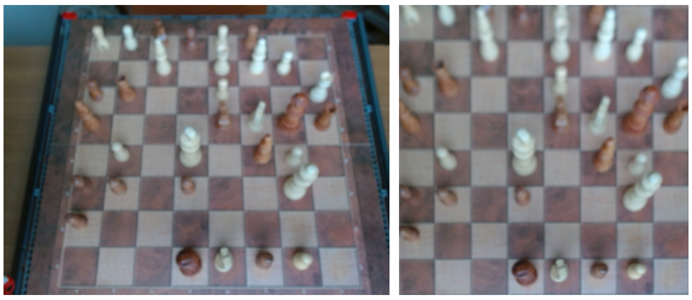
In the next part of this work, we will focus on our line-based algorithm implementation, which was integrated into the final version of the robot's control program due to its high reliability under various lighting conditions. However, we believe that the simplicity and functionality of the gradient-based approach will be appealing to many researchers, and we hope that future versions will bring many improvements.

3.1.2 Algorithm 2: Line-based chessboard detector

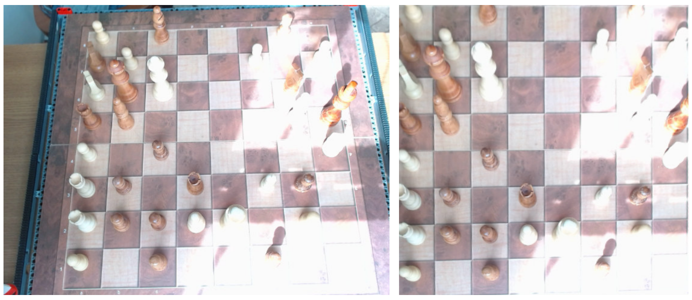
This section introduces the second implemented algorithm, utilizing the popular line detection method. We propose a novel approach utilizing Hough-line detection to recognize a chessboard's vertical and horizontal lines, thereby accurately detecting the entire grid structure. We will present the individual



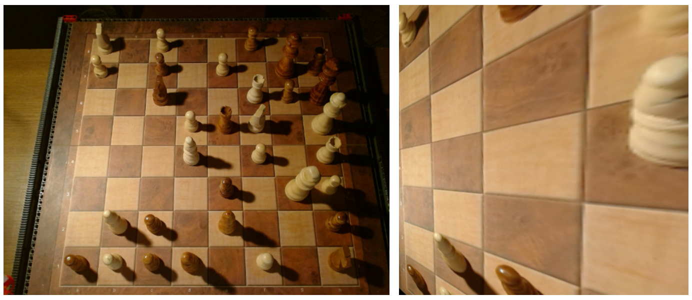
(a) : Standart chessboard image [17 iters, 179 ms].



(b) : Blurred image - camera autofocus failure [16 iters, 207 ms].



(c) : Over-exposed unbalanced lighting [16 iters, 206 ms].



(d) : Algorithm failed due to inconsistent lighting [69 iters, 911 ms].

Figure 3.5: Examples of the algorithm's inputs and outputs with each image's number of iterations and processing time.

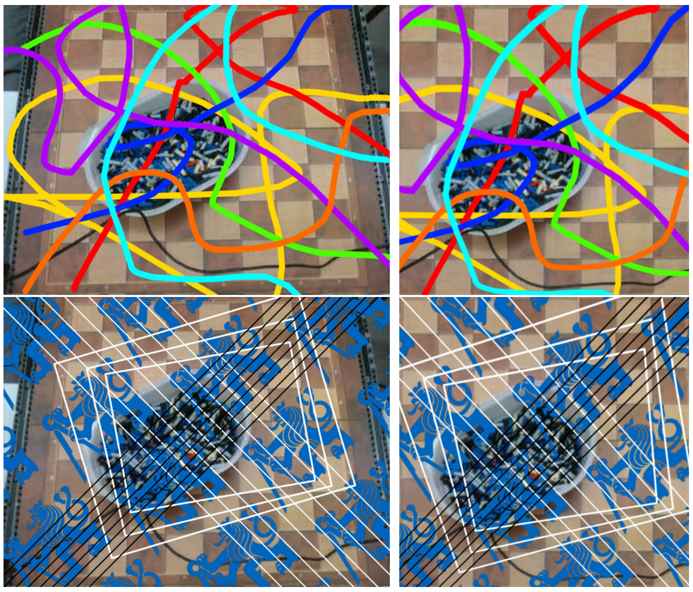


Figure 3.6: Digitally edited images demonstrating the algorithm's ability to cope with many occlusions in the image. The majority of corner-based and line-based detectors would fail on these example images. [17 iters, 154 ms (top); 15 iters, 175 ms (bottom)]

steps the algorithm performs and discuss the results achieved at the end of this section.

Step 1: Line detection

The Hough Transform is a feature extraction technique used in computer vision image processing to identify simple shapes such as lines and circles. The algorithm works by transforming points in the image space into curves in the Hough space (Figure 3.7). To detect straight lines, the Hough Transform maps each point in the image to a line in Hesse normal form (Equation 3.4) in the Hough space. The most dominant intersection points of the Hough space curves correspond to collinear points in the image.

$$r = x\cos\theta + y\sin\theta, \quad \text{where } \begin{array}{l} r \text{ is the line's distance from the origin} \\ \theta \text{ is the angle of the line's normal} \end{array} \tag{3.4}$$

As shown in Figure 3.7a, the input to the Hough Line detector is an image with highlighted edges. We perform this preprocessing step using the Canny edge detector. Thanks to edge detection, the line-detection algorithm is less dependent on the illumination conditions of the chessboard, as edge detection is scarcely affected by the overall image exposure (except for overexposure). Both the Hough line and the Canny edge detectors have

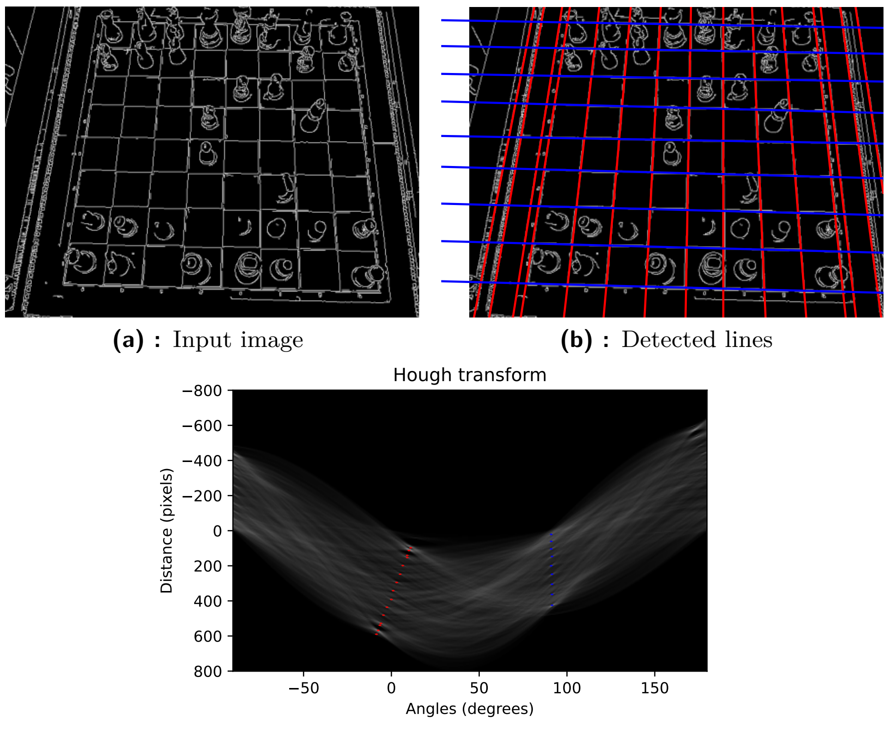


Figure 3.7: An illustration of the Hough transform algorithm applied to a chessboard image (3.7a) after the Canny line detector processing. We can observe continuous rows of dominant points in the Hough space (3.7c, highlighted), corresponding to the detected lines on the chessboard (3.7b, highlighted).

many tunable parameters that must be correctly set for specific purposes (see Figure 3.8). Our algorithm extracts the 100 most dominant points from the Hough space (a typical 8x8 chessboard contains 18 lines), from which it then selects those that most correspond to the characteristics of the chessboard using the RANSAC algorithm (i.e., two groups of collinear points in the Hough space - see Figure 1 below). Due to the low contrast of our chessboard, we cannot rely on the most dominant points in the Hough space corresponding to the lines of the chessboard. The Hough line algorithm is typically applied in the range $\theta = [0, \pi)$ (the direction of the lines is negligible). We perform the Hough line transformation in the range $\theta = [-\pi/2, \pi)$ to avoid situations where the colinear points pass through the $0/\pi$ border; the RANSAC algorithm would fail in these cases. The RANSAC method is the only stochastic part of our algorithm. We tried to replace RANSAC with DETSAC (deterministic RANSAC), which tries all possible combinations, but this solution was ultimately scraped for its slower speed.

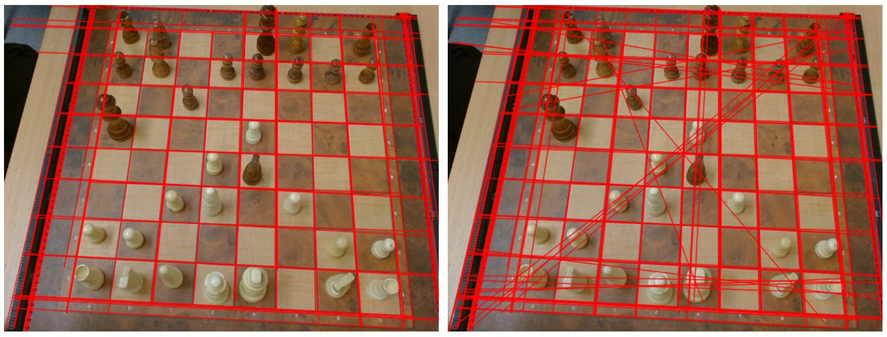


Figure 3.8: A comparison between well-set parameters of the Hough line algorithm (left) and poorly set parameters (right).

Step 2: Line filtering

If the output of Step 1 contains at least five lines in both directions (termed a_lines and b_lines), the algorithm proceeds to the line filtering step. The outcome of Step 1 frequently includes lines unrelated to the chessboard and, conversely, might miss lines that are a part of the chessboard. In this part of the text, we will show the implemented methods by which we were able to filter out the outliers and supplement missing lines.

In the first step, the algorithm detects the arrangement of lines in *a_lines* and *b_lines* and sorts them according to their orientation. Subsequently, it identifies the four outermost corners, arranging them in a clockwise direction from their central point. We calculate a transformation that removes the image distortion using four corners and the equations 3.2 and 3.3 from Chapter 3.1.1. This transformation is applied to all detected lines, resulting in distortion-free *trans_a_lines* and *trans_b_lines*. The output of this step is visualised in Figure 3.9.

The algorithm can then proceed with detecting the chessboard pattern in the lines. This is done separately for both groups of lines (directions). The algorithm can be divided into these three steps:

1. Compute the reference gap size: This calculates the reference gap size between lines, which is later used to help define valid lines. This step uses two essential functions. The $compute\_diffs\_and\_ratios(lines)$ function returns a sorted list of gap sizes between the *trans_lines*. The $mean\_largest\_group(gaps, epsilon)$ returns the mean of the largest contiguous group of numbers within the epsilon range of each other, which in our case represents the reference gap size between valid lines (i.e. the chessboard square size in a given direction).

2. Valid line detection: For each combination of lines, the algorithm attempts to fit the expected number of lines (samples) between them (based on the distance of the pair and the reference gap size). If the

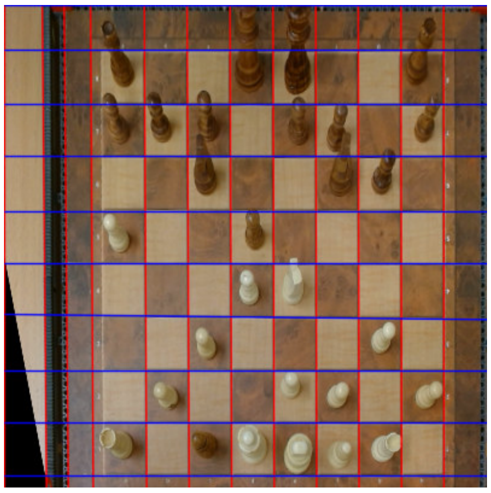


Figure 3.9: A visualisation of the detected lines after removing the camera distortion, i.e. *trans_a_lines* and *trans_b_lines*. The leftmost line of the chessboard was not detected, while many lines in the image do not belong to the chessboard. The image is stretched in the x and y axes because, at this point, no assumptions can be made about the dimensions of the detected lines.

selected pair cannot fit at least MIN_LINES (4 in our case) samples between them, it is automatically rejected. Each combination pair is scored based on the number of generated samples that match the $trans\_lines$ with at least $max\_deviation$ accuracy (in our case $\frac{1}{10}$ of reference gap size). The pair with the highest score is selected, and missing lines between the points of this pair are generated (Figure 3.10).

3. Additional line generation: While the previous step supplemented lines between the detected chessboard lines, it may happen that the outermost lines of the chessboard are missing, as seen in Figure 3.9. In fact, the outermost lines are often not detected, as they are not as distinctive and are often occluded by chess pieces. The algorithm automatically adds $(9 - N_{detected\_lines}) + EXTRA\_LINES$ to each side by extrapolating the inner lines. $EXTRA\_LINES$ (in our case 1) is a variable that we were forced to add, as our chessboard has edges precisely the same size as the squares themselves, thus behaving like an 11x11 grid from the perspective of a line detector. For this reason, we always need to add one more line on each side to ensure that the generated grid contains the actual 9x9 chessboard.

The final output of the line filtering step can be seen in Figure 3.11. We can observe that the chessboard pattern was correctly detected, all outlier lines have been removed, and new missing lines have been generated that extend beyond the borders of the chessboard to enhance the robustness of the algorithm.

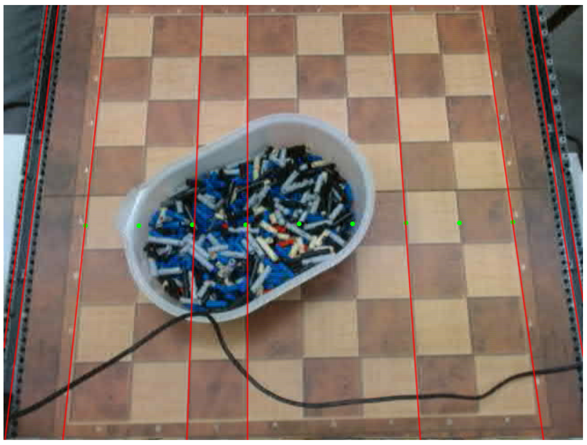


Figure 3.10: The generated samples (green dots) of the winning pair of lines in the worst-case scenario situation (some of the missing lines were deleted manually for testing purposes).

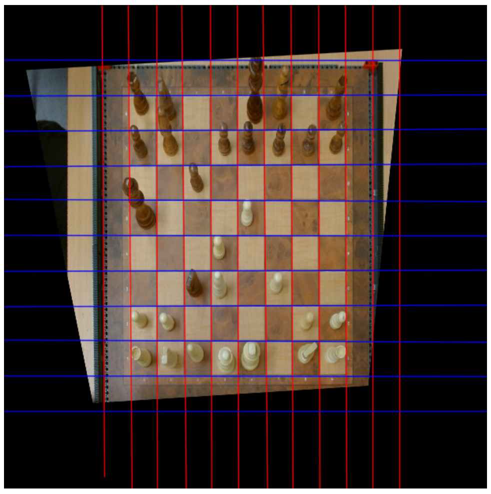


Figure 3.11: Output of the line filtering step. The chessboard pattern was correctly detected. The lines for the chessboard candidates reach beyond the borders of the image.

Step 3: Template matching

In this final step, the algorithm searches for the chessboard in the $m \times n$ grid from Step 2, where $m, n \geq 8$. We use the same mask as in the Gradient Descent approach (see Figure 3.4). Here, however, we do not move the mask over the image; instead, we transform individual candidates (sub-grids of 8x8) over the mask. Similarly to the Gradient Descent algorithm, we compare the image with the mask and calculate a correspondence score (here, we apply a different calculation method). The candidate with the highest score is chosen.

To calculate the correspondence score, we first overlay the mask with the candidate, thereby reducing the size of the candidate to 24x24 pixels (Figure

3.12, left). We then convert the candidate into a black-and-white image using Otsu's Binarization (Figure 3.12, middle). The black-and-white image is normalized so that black corresponds to negative values and white corresponds to positive values of the same magnitude. The correspondence score is then obtained by element-wise multiplication (Figure 3.12, right) of the normalized candidate with the normalized mask. We obtain the correspondence score by summing all the values in the resulting matrix. The output of this algorithm is the four corners of the winning candidate, i.e., the candidate with the highest correspondence score (Figure 3.13).

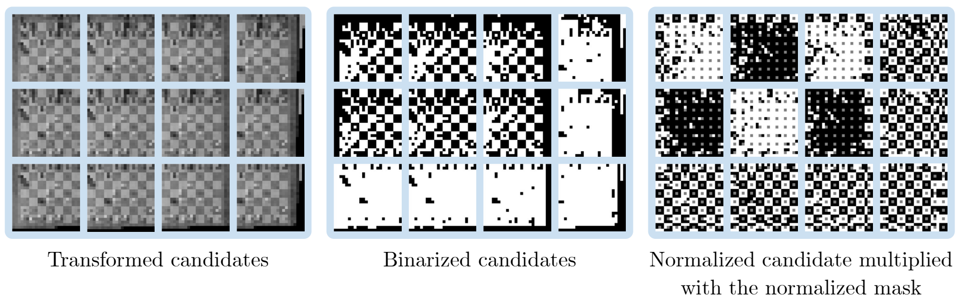


Figure 3.12: Visualization of template matching for all candidates from Figure 3.11. In the right picture, the winning candidate can be distinctly identified as the image with the highest brightness.

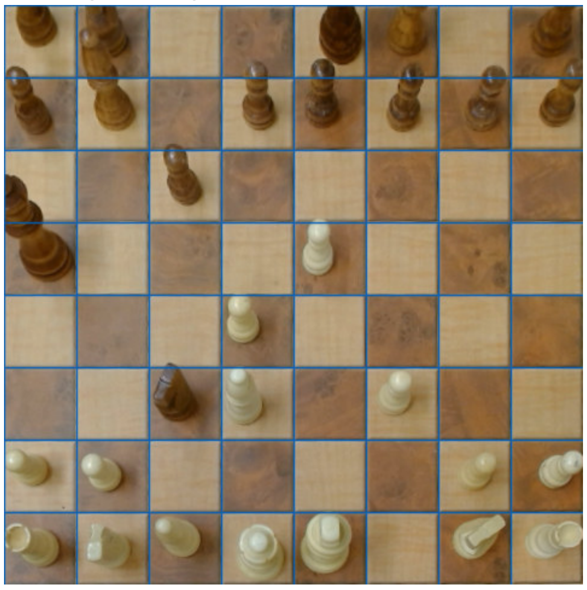


Figure 3.13: The output of the line-based chessboard detector. [processing time: 497 ms]

Performance analysis of the Line-Based Detector

The line-based detector was utilized to generate the real-world image dataset (refer to Chapter 3.2.3). Out of the total number of 5032 captured images, our line-based detector successfully identified the chessboard in 4814 images.

This figure can be slightly misleading as there are numerous poor-quality images (e.g. total darkness, severe overexposure, etc.) among the shots where the algorithm failed. A notable benefit is our ability to identify instances of algorithmic failure. The dataset generator successfully detected and eliminated all images where the chessboard detector failed without a single error. Chapter 3.2.3 shows examples from the real-world dataset that our implemented algorithm successfully detected.

The literature, specifically Paper [10], posits that the parameters of a line-based chessboard detector utilizing the Hough line algorithm must be modified in response to every alteration in lighting conditions. However, our results demonstrate that it is indeed feasible to construct a highly robust chessboard detector based on this principle, negating the necessity for frequent parameter adjustments. In fact, our detector displayed immediate functionality with several other chessboard configurations, as depicted in Figure 3.14.

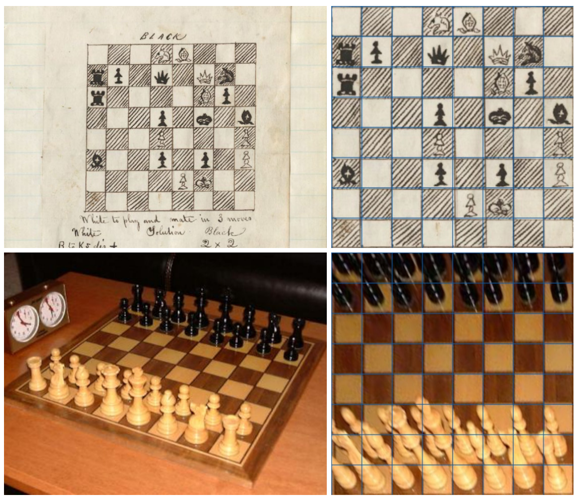
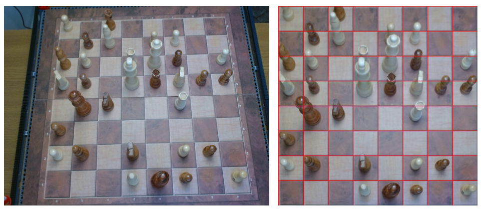
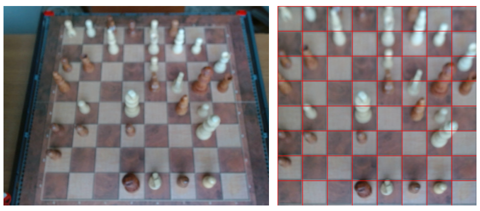
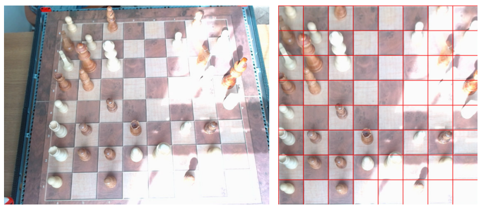


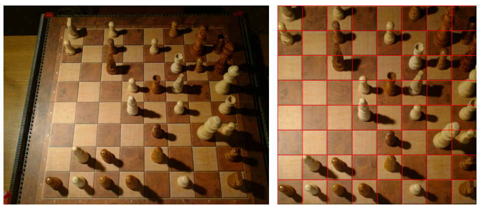
Figure 3.14: Our proposed line-based chessboard detector deployed on never-before-seen chessboards without any parameter tuning or other modifications. [Processing time: 269 ms (top), 336 ms (bottom)]

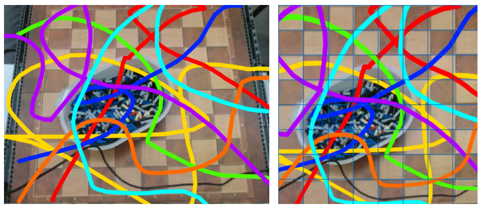
Figure 3.15 compares the results of our line-based detector with the results of the gradient descent detector (Figure 3.5) introduced in Chapter 3.1.1.


(a) : Standart chessboard image [378 ms].


(b) : Camera autofocus failure [488 ms].


(c) : Over-exposed lighting [518 ms].


(d) : Inconsistent lighting [553 ms].


(e) : Challenging image [274 ms].

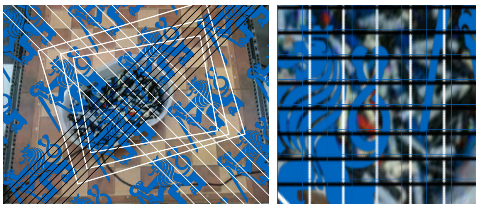

(f) : Algorithm failure [468 ms].

Figure 3.15: Examples of the algorithm's inputs and outputs with each image's processing time. A direct comparison to Figures 3.5 and 3.6.

3.2 Chess piece detection and classification

The detection and recognition of chess pieces is an integral component of a chess-playing robot. Most of the available literature simplifies this exceedingly challenging task by detecting only changes on the chessboard or colours of the pieces. Scientific papers presenting more sophisticated algorithms, typically based on machine learning methodologies, frequently mention the lack of available chess datasets. In this chapter, we will address the issue of automated generation of datasets (both synthetic and real), implement a highly reliable chess piece detector utilising a neural network, and evaluate the achieved results.

3.2.1 Introduction to the problem

There exist numerous techniques for chess piece detection, each with varying capabilities and implementation complexities. Broadly, we can categorise these approaches into three fundamental categories based on the level of information they provide:

1. Piece Presence Detection Methods: These are the most straightforward techniques that focus on detecting whether a piece exists on a particular square. They often employ specially modified chessboards and are usually not highly reliable. In scenarios like piece capture, they must detect the actual process of figure exchange to identify the capture reliably. These methods depend on the knowledge of the previous state of the chessboard, and a single detection error can affect the remaining game.

2. Piece Presence And Colour Detection Methods: Typically, these methods require a laboratory environment and/or a static camera. They are capable of identifying the presence of a piece and its colour. Like the first category, they also rely on the knowledge of the previous state of the chessboard, and any single detection error can impact the rest of the game. These algorithms fail to detect pawn promotion (a scenario when a pawn reaches the last rank and changes into a queen, bishop, knight, or rook of the same colour).

3. Piece Presence, Colour, And Type Detection Methods: These are the most sophisticated algorithms, usually based on machine learning techniques. They enable the detection of all chess situations, even without knowledge of the previous state of the chessboard.

Overview of chess piece detection methods

Hardware solutions:

These methods employ specially modified chessboards and chess pieces. In [8], they make use of a reed switch array (a sensor under each of the 64 squares) and chess pieces equipped with magnets. Other popular approaches include utilising mechanical switches, built-in LC oscillating circuits with an antenna in the pieces, and Near Field Communication (NFC) equipped pieces. These methods eliminate the need for complex image processing by requiring a modified chessboard. As our goal is to create a robot that plays on a standard, unmodified chessboard, none of these methods were considered suitable for our purpose.

Standart computer vision approaches:

1. Background/Image Subtraction Methods: These methods work by comparing the current image of the chessboard with a stored image of the empty board or an image from the previous board state [3, 5]. Combined with a Blob Detection algorithm, this approach provides changes on the board as long as a static top-down camera in a controlled environment is used. Usually, a simple computer vision technique, such as thresholding, is used to decide the colour of the moving pieces [19]. The first iteration

of the robot presented in this thesis utilised the method of image subtraction. However, any camera shake or change in lighting caused failure. Due to the positioning of the camera, we also had to deploy complex methods to detect overlapping pieces and pieces that, from the camera's perspective, encroached into other squares.

2. Color Histogram Methods: This technique detects changes in the colour histogram of individual squares after each move to determine which squares have changed [10]. The critical weakness of this algorithm is its sensitivity to light changes and limited performance with pieces on squares of the same colour.

3. Coloured Piece Methods: A popular technique among the research community is using distinctively coloured (e.g., green, red, etc.) chess pieces [4]. By analysing the HSV colour space, the chess pieces can be detected with very high reliability and accuracy. However, this method is limited by the requirement for specifically coloured chess pieces and, typically, a top-down camera positioning.

4. Edge Detection Methods: Edge analysis methods focus on each of the 64 chessboard squares within an image. Using a well-tuned Canny Edge detector [10], for instance, accurately detects chess piece edges, even when the piece's colour closely matches the square behind it. This technique requires the use of a top-down camera and a diffused light source, as it can misinterpret shadows cast on empty squares as the presence of pieces.

5. Shape Descriptor Methods: These methods leverage the symmetrical nature of chess pieces, distinguishing individual pieces based on their unique shapes [20]. The advantage of this approach is that it does not require knowledge of the previous state of the chessboard. The disadvantage, however, lies in the necessary compromise regarding the positioning of the camera. A top-down view does not provide the shape descriptor with sufficient information. A side view offers excellent recognition but also leads to numerous occlusions. Therefore, a well-placed camera that balances these occlusions with the algorithm's reliability is needed.

Machine learning computer vision approaches:

Traditional computer vision techniques often come with a range of restrictions and limitations. The relatively new field of research utilising machine learning brings forth many advantages while eliminating most of the limitations.

1. Convolutional Neural Networks (CNN): The convolutional neural network is a type of deep learning model particularly effective for image analysis tasks and thus is well-suited for chess piece classification tasks. Paper [17] uses the Xception with over 57 000 labelled images and

achieves an accuracy of more than 90%. In the study [21], Google's inception v3 DNN was employed to achieve a classification accuracy of approximately 97.5%, with the lowest accuracy recorded being 92.39%, specifically for the black bishop. The research paper [11] conducts a comparative analysis of CNNs. The researchers utilise a synthetically generated dataset of 4888 chessboard images to train their victorious ResNet model. They claim to have introduced a state-of-the-art solution, given that their system accurately classifies 99.77% of the chess pieces (corresponding to 93.86% of chessboard configurations). Additionally, their system enables the training of the network on a new chess set using just two images.

2. Support Vector Machines (SVM): Support vector machines work by dividing the feature space with multidimensional hyperplanes. Paper [6] uses two separate descriptors - scale-invariant feature transform (SIFT) and histogram of oriented gradients (HOG) for feature extraction with a resulting classification accuracy of 85%. The study [22] uses SVMs combined with a VGG19 convolutional neural network as the feature extract, achieving an accuracy of 98.07%

Machine learning-based methods yield impressive results, but they typically require a substantial amount of training data. There is a lack of quality annotated datasets that are publicly available [10, 6]. In Chapters 3.2.2 and 3.2.3, we will address the creation of both synthetic and real-world datasets for our purposes. In Chapter 3.2.4, we will utilise these datasets to train a neural network and present the results.

Transitioning from Image Subtraction to Machine Learning

During the development of our chess-playing robot, it became clear that the image subtraction method we initially implemented was overly sensitive to external influences and the structural rigidity of the robot. Additionally, the method struggled to accurately detect chess pieces overlapped by others and movements of larger pieces that spanned multiple squares. Despite extensive efforts to stabilise the image, the image subtraction method proved unreliable without a static camera, making it challenging to play an entire chess game without detection errors.

Due to these challenges, we decided to explore a new solution for piece recognition, leveraging machine learning techniques. A common approach to the piece detection problem involves partitioning the chessboard into 8x8 squares. Each square is then separately analysed using a classification neural network. Despite the intuitive appeal of this approach for a grid-like chessboard, the work in [11] found that adding a 50% overlap to each square significantly improved training results. Given the camera positioning on our robot, which results in overlaps and overhangs of individual pieces, we decided to take a different approach.

Our work utilises the neural network You Only Look Once (YOLO), specifically versions YOLOv5 [15] and later YOLOv8 [16]. This network specialises in object detection, offering us several advantages over other methods. YOLO processes the entire image in a single pass, detecting and classifying objects concurrently. This method significantly reduces computational costs, making it an ideal solution for real-time applications like our chess-playing robot.

One of the main strengths of YOLO is its ability to detect partially overlapped objects, an essential capability in our case. Several publicly available pre-trained models allow us to utilise transfer learning techniques to adequately train the neural network, even with a dataset of limited size.

We believe the implementation of YOLOv8 significantly improved the robustness and performance of our robot, as evidenced by the results discussed in Chapter 3.2.4. This approach allowed us to remove most limitations from our application and, in combination with our chessboard detector, to create a very robust algorithm for move recognition.

3.2.2 Generation of the Synthetic Dataset

In the domain of machine learning, particularly where deep learning algorithms are used, the quality and quantity of data play a critical role in the performance and robustness of the model. However, in many real-world applications, acquiring a large, diverse, and well-labelled dataset can be challenging, expensive, and time-consuming. This is particularly true in the case of our chess-playing robot, where we require a large number of images featuring up to 32 chess pieces each. Since manually labelling a dataset of 5000 images would require about 4 months of continuous work, we resort to creating a synthetic dataset. This dataset is generated programmatically, allowing us to produce a virtually limitless and diverse array of chessboard configurations under different lighting conditions and from multiple perspectives.

Obtaining 3D Models of the Chess Pieces

To transport the chess pieces into the virtual world, we utilized the Shining 3D EinScan-SE 3D scanner. This scanner enabled us to capture the geometry and texture of all chess pieces. The scanning process is depicted in Figure 3.16. Each piece was scanned from various angles using a motorized turntable and angled wedge to avoid splits in the model's geometry. This step was most crucial for the rook, whose upper part is hollow and, therefore, difficult to scan. By merging these scans, we obtained a model comprising approximately 3 million polygons. In the subsequent step, we manually reduced all models to approximately 10% of their original size, cleaned them of scanning artefacts, aligned their position and orientation to the world coordinate axes, and rescaled them to match the size of the real-life pieces (Table 3.1).

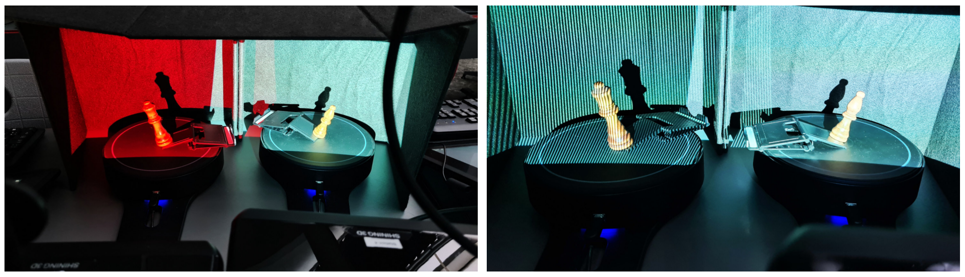


Figure 3.16: The photographs depict the process of 3D scanning using the Shining 3D EinScan-SE. The images feature two chess pieces being scanned concurrently, placed on a motorized turntable, some of them with a wedge for angling the piece. The 3D scanner uses the projected patterns to capture the geometry and texture of the chess pieces.

Chess piece	Height
Pawn	40 mm
Rook	47 mm
Knight	48 mm
Bishop	66 mm
Queen	82 mm
King	88 mm

Table 3.1: Height of all chess piece types.

Due to the glossy surface of the chess pieces, the textures of the scanned models were full of artefacts (pink spots and white reflections). The textures of all pieces had to be manually cleaned of gloss and reflection artefacts. Figure 3.17 depicts a white queen before and after processing.

The outcome of this step is a database of all types of chess pieces in both colours, which we subsequently utilize to create the synthetic dataset and game visualization (Figure 3.18).

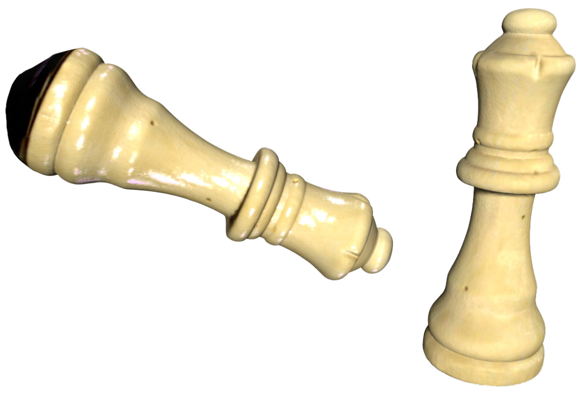


Figure 3.17: Comparison of the 3D model of the white queen before and after manual processing. On the left is a model depicted after merging the scanned point clouds and converting them into a mesh model. On the right is an oriented, scaled model that has been cleaned of artefacts.

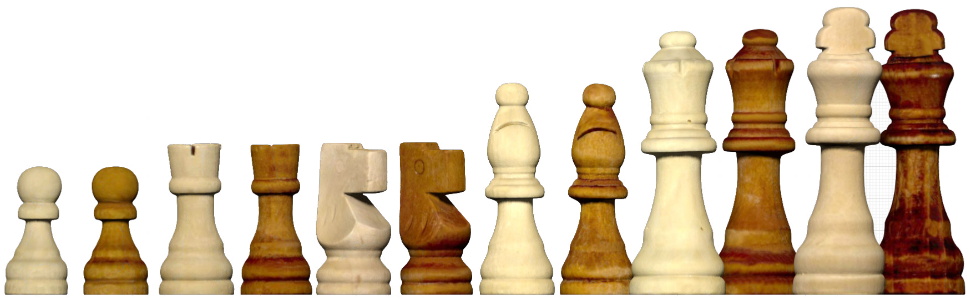


Figure 3.18: 3D models of chess pieces rendered in Blender.

The 3D Environment

To create the synthetic chess dataset, use Unreal Engine 4.27 [23], a powerful game engine known for its realistic visuals and excellent performance. The real-time rasterization rendering greatly expedites the data generation process. To put this into perspective, in a comparable study [10], the researchers employed Blender for image rendering, where each image requires approximately five minutes to render.

Unreal Engine 4 provides advanced features that closely simulate real-world physics and lighting. We utilize the Dynamic Volumetric Clouds with Ray Marching system. This technology allows us to simulate different environmental lighting conditions in a highly realistic manner, leading to a wider variety and better quality of synthetic images (Figure 3.19).

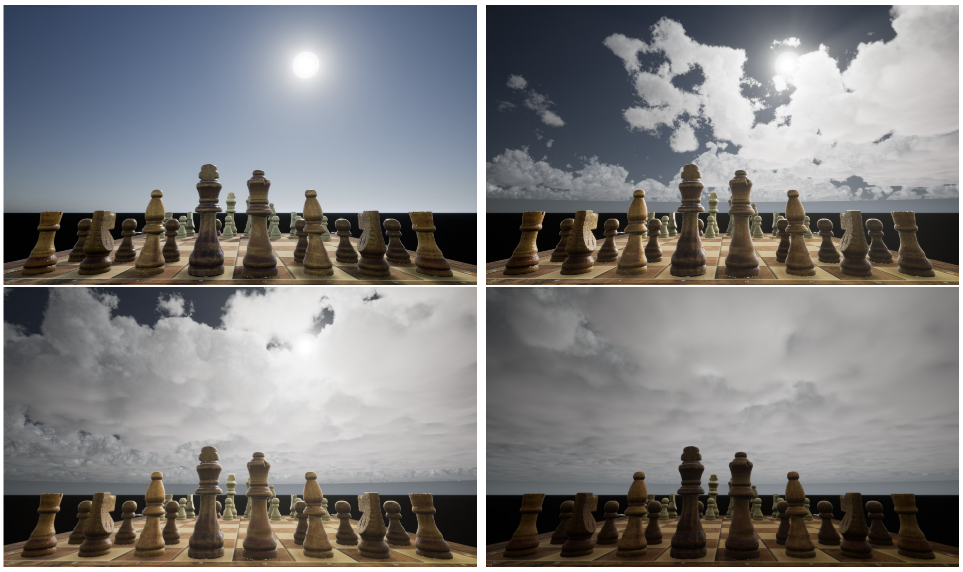


Figure 3.19: The effects of light scattering in dynamic volumetric clouds on the illumination and shadows of the virtual chess pieces.

Data generation

To augment the chessboard and environmental conditions, we utilize UnrealCV [24], a plugin for the Unreal Engine that provides a suite of valuable tools for computer vision applications. The primary advantage of UnrealCV is the associated Python package of the same name, which allows for executing commands in the Unreal Engine from a Python environment.

Our implemented algorithm generates random chessboard configurations involving all 32 chess pieces. This diverges from the method used in [11], where the images are generated according to chessboard configurations from games by the chess grandmaster Magnus Carlsen. In this manner, we are able to avoid some frequent configurations (such as the initial chess configuration) and generally obtain a more balanced dataset. Each of the 32 pieces is slightly adjusted in its position within the square, and a random rotation is applied. The camera's position and orientation are also randomized within a specified window, roughly resembling our real-life configuration. Subsequently, the system modifies the lighting conditions, including the sun's position, the intensity of light emission (both ambient and solar), and the volume and location of the volumetric clouds.

Following these adjustments, a series of 34 high-resolution (1920x1440) images are captured: the rendered chessboard, a segmentation image of all pieces, and 32 individual segmentation images for each piece. The algorithm saves the rendered image, the Forsyth-Edwards Notation (FEN) representation of the chessboard configuration, and the segmentation image featuring all chess pieces. The algorithm then uses the 32 images of individual pieces to create an annotation file describing the location of each piece for object detectors, such as the YOLO neural network. Finally, the algorithm renders one last chessboard image, this time with a random background. The three output images for a single dataset entry can be seen in Figure 3.20.

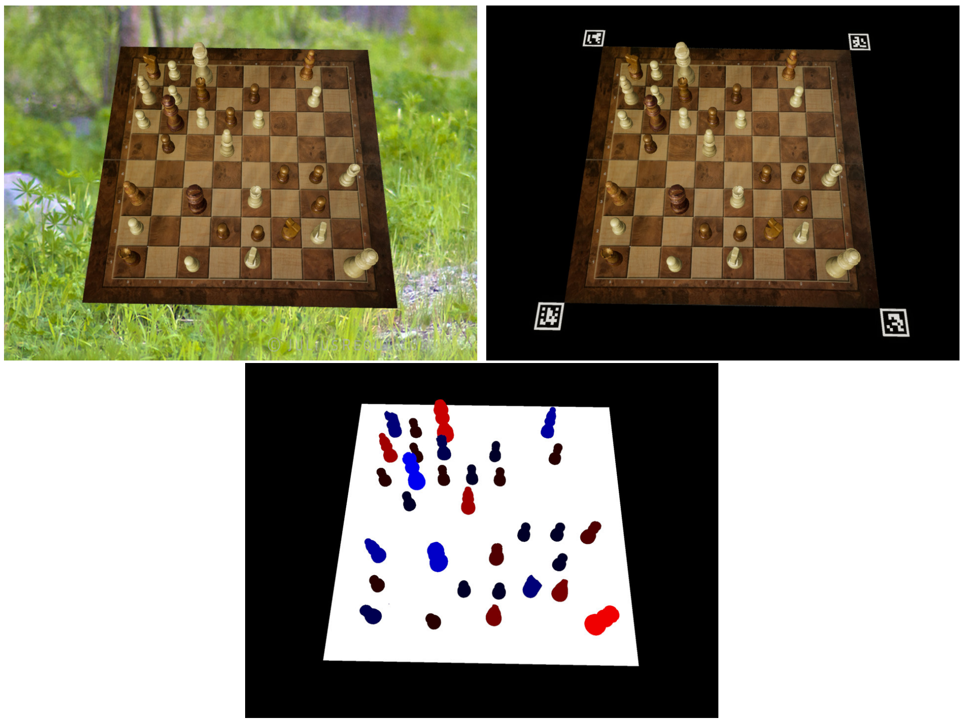


Figure 3.20: The three output images of a single dataset entry. The image with a randomized background (left), the original image with ArUco markers for calibration purposes and a segmentation image.

The background of the rendered images was randomized using images from the COCO dataset [25]. By introducing background variation, we ensure a more robust learning process that generalizes better to real-world conditions, reducing the chance of overfitting. In our research, we do not utilize the segmentation image; it is included in the dataset for the purposes of other scientific studies. The included FEN file allows researchers to identify the board configuration from a single line of text. To illustrate, the corresponding FEN file for the configuration depicted in Figure 3.20 reads as follows:

```
r1P1N2Q/P2ppnN1/b1k1R1p1/5ppR/1p1B4/PqPpP3/BPr1p1P1/nPK3b1 w KQkq – 0 1
```

The annotation file carries information regarding the positions of all 32 chess pieces. The corresponding annotation file for the configuration shown in Figure 3.20 appears as follows:

```
4 0.26536458 0.70868056 0.04739583 0.05625000
```

```
11 0.39765625 0.72812500 0.03385417 0.04652778
9 0.52526042 0.71076389 0.03385417 0.06180556
6 0.75390625 0.72534722 0.07447917 0.08541667
11 0.27864583 0.62152778 0.03333333 0.05000000
5 0.46276042 0.63680556 0.02864583 0.05000000
5 0.53619792 0.64236111 0.02968750 0.04861111
3 0.60677083 0.62951389 0.04166667 0.06597222
9 0.66458333 0.64097222 0.03541667 0.06527778
2 0.27526042 0.52881944 0.04635417 0.07013889
1 0.40677083 0.54652778 0.04479167 0.08611111
10 0.53125000 0.53923611 0.03229167 0.06458333
5 0.66510417 0.56111111 0.03333333 0.05138889
5 0.58906250 0.47638889 0.02812500 0.05555556
5 0.66041667 0.47638889 0.03229167 0.05416667
10 0.73046875 0.47812500 0.04531250 0.06597222
5 0.34635417 0.38958333 0.03020833 0.05694444
8 0.47161458 0.38750000 0.03072917 0.08055556
11 0.29348958 0.32187500 0.03281250 0.05347222
0 0.35651042 0.30486111 0.04947917 0.10833333
11 0.42005208 0.31666667 0.02656250 0.05972222
5 0.47968750 0.31493056 0.02604167 0.05902778
11 0.53802083 0.32048611 0.02604167 0.05486111
8 0.30260417 0.23888889 0.04062500 0.08472222
11 0.36197917 0.25277778 0.03020833 0.05833333
4 0.41927083 0.23819444 0.03229167 0.07361111
5 0.52812500 0.24965278 0.02500000 0.06041667
11 0.65598958 0.25659722 0.02968750 0.06180556
3 0.31250000 0.17256944 0.03750000 0.07708333
11 0.35989583 0.18333333 0.02812500 0.05694444
7 0.41979167 0.16388889 0.04375000 0.12361111
2 0.64088542 0.16840278 0.03281250 0.08958333
```

Each line of the annotation file represents a single chess piece. Each line lists the following details in this order: the ID of the piece type, the x-coordinate of the centre point, the y-coordinate of the centre point, the relative width, and the relative height. The ID number ranging from 0-11 represents the following types and colours of pieces in this order: black queen, black king, black bishop, black knight, black rook, black pawn, white queen, white king, white bishop, white knight, white rook, white pawn.

Results

We have developed a system capable of generating a realistic-looking synthetic dataset. The output of a single iteration, which computes approximately 30 seconds, consists of five files: the FEN configuration, annotation data, an annotation image, a real image with calibration ArUco markers, and a real image with a randomized background. Using this algorithm, we have generated a dataset comprising a total of 5000 images. Sample images from this dataset (specifically those with random backgrounds used in our training of the YOLO model) are depicted in Figure 3.21.

We trained the YOLOv5 on 2,400 images from the synthetic dataset. This

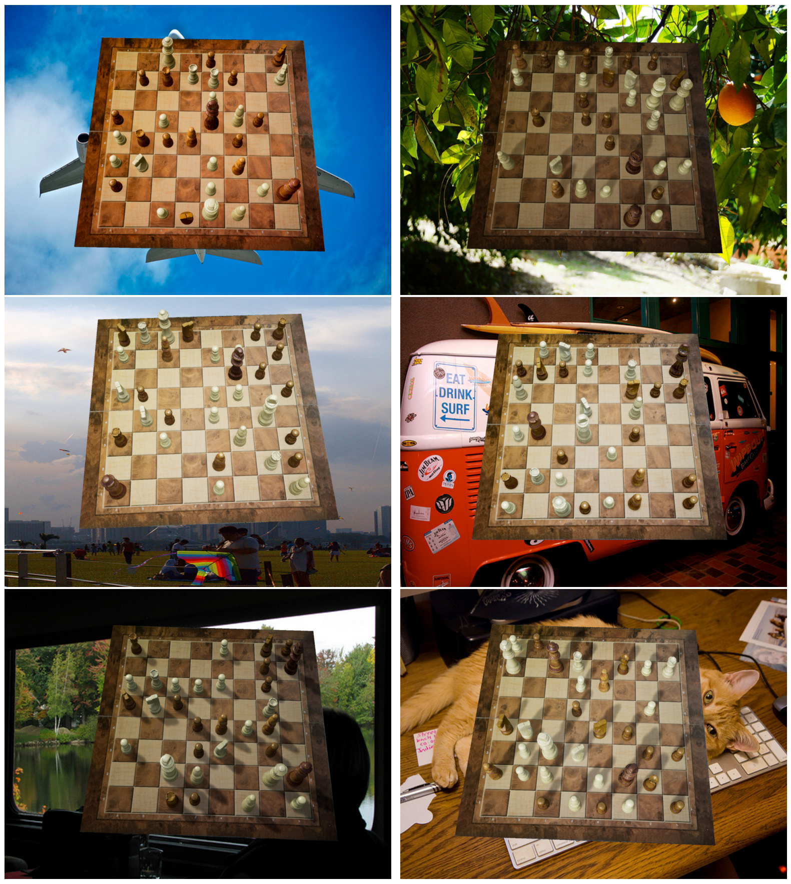


Figure 3.21: An example of training images from the synthetic dataset.

model was built upon the publicly available pre-trained YOLOv5s checkpoint [15] with 7.2 million parameters initially trained on the COCO dataset. Due to the hardware constraints of the computer used at the time, we opted for this smaller model and a reduced number of images. The images were resized to a resolution of 640x480, aligning with the specifications of our utilized camera, and their hue and saturation were further adjusted within a range of $\pm 20\%$. Training this model on an NVIDIA GeForce MX150 graphics card took roughly 24 hours (100 iterations). While the resulting model performed sufficiently well under optimal lighting conditions (Figure 3.22), it struggled in scenarios not covered by the synthetic dataset, as illustrated in Figure 3.23.

Since we aim to develop a robust system that functions under all lighting conditions, these results were unsatisfactory. While object detection performed relatively well, the classification of the individual piece types often failed. We leverage these results in Chapter 3.2.3, where we discuss generating and automatically labelling a real-world dataset using our chess-playing robot.

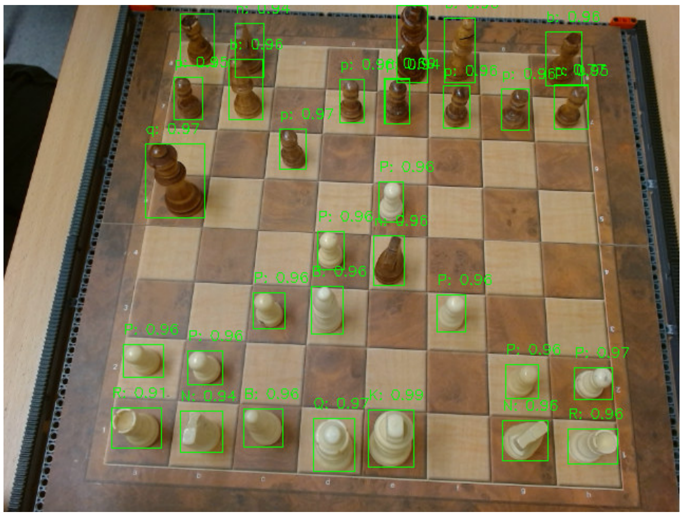


Figure 3.22: The trained model, when recognizing a real chessboard under ordinary lighting conditions, correctly identified the majority of the pieces.

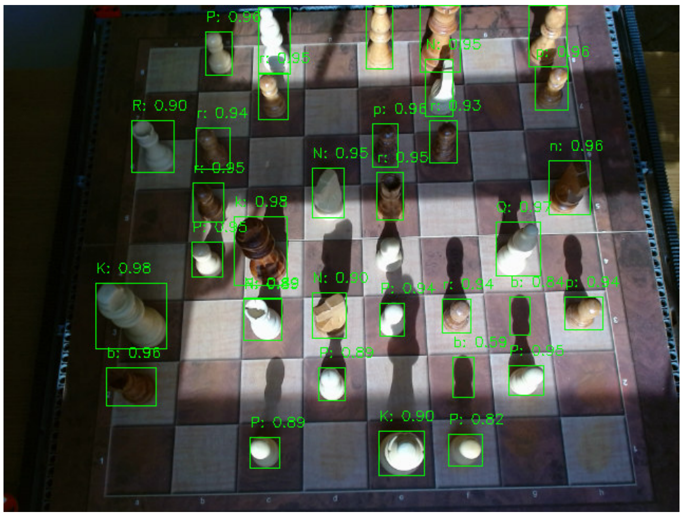


Figure 3.23: The trained model, when recognizing a real chessboard under challenging lighting conditions, fails in classification multiple times. One piece was undetected, while two shadows were mistakenly identified as chess pieces.

The confusion matrix of the model trained on synthetic data, when validated on the testing subset of the real-world dataset from Chapter 3.2.3, is depicted in Figure 3.24.

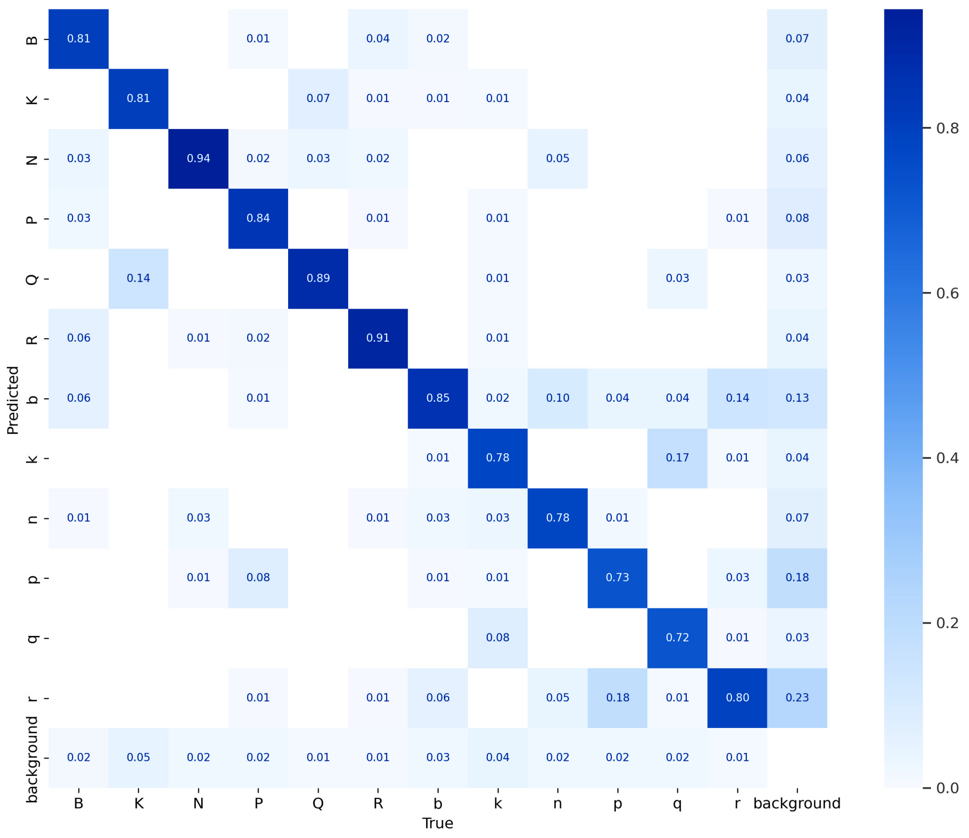


Figure 3.24: The normalized confusion matrix of the model trained on synthetic data when validated on the testing subset of the real-world dataset. Uppercase letters correspond to the white chess pieces.

3.2.3 Generation of the Real Dataset

Given that we had a robust chessboard detection system from Chapter 3.1.2 and a reliable system for marking the chess pieces from the previous chapter, we decided to enhance our results by creating a real-world dataset. Manually annotating the chessboard images would be highly complicated and time-consuming. For this reason, we developed a script for our chess robot, enabling it to generate the dataset autonomously.

The algorithm we used to generate the real-world dataset was similar to the one we employed for the virtual dataset generation. Initially, all the pieces must be arranged according to the standard chess configuration. In a randomized order, the robot grasps each of the 32 pieces and places them on a random empty square. This randomized chessboard serves as the starting state for our data generator. The algorithm records every move made by the robot, maintaining a virtual twin of the real-world chessboard in its memory.

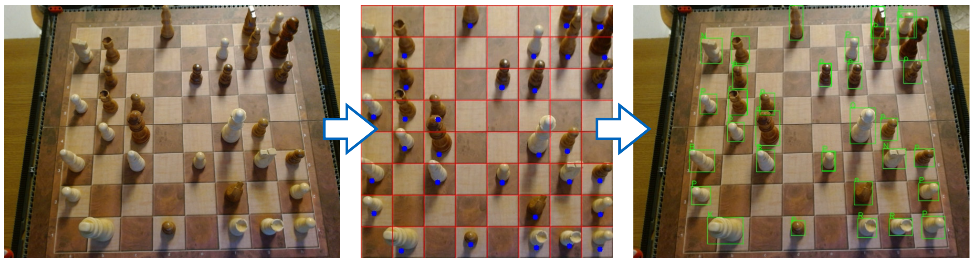
In a continuous loop, the robot performs two completely random moves and then captures an image of the chessboard. The chessboard image is saved along with a file containing the current FEN representation of the board and an SVG visualization of the board for user verification. Using this method, the robot collected 5032 images over several days under various lighting conditions. At night, the chessboard was typically illuminated by a studio light with a softbox. During the day, natural daylight was utilized.

Labeling the data

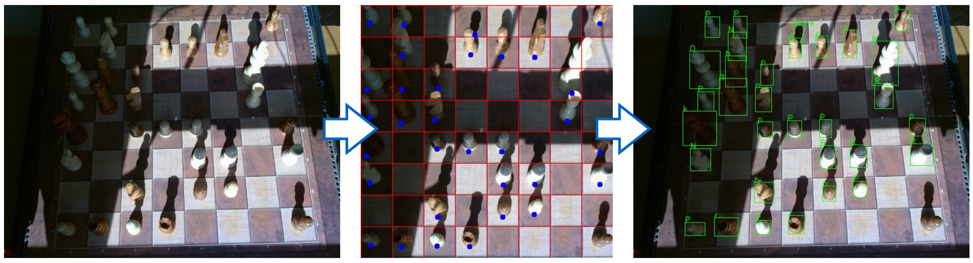
The collected data was labelled using an automatic script. In the first step, the script detects the chessboard using the line-based detector from Chapter 3.1.2. The pieces on the board are then identified using a YOLOv5 model trained on the synthetic dataset. Based on the saved FEN file configurations, each detected piece is correctly labelled by colour and type. If the algorithm fails to detect the chessboard in the image, that image is discarded. Notably, most of the discarded images were of very poor quality, and excluding them from the dataset was on point.

Out of 4814 images that the algorithm automatically labelled, additional 14 images were manually discarded due to their exceedingly poor quality. This left a total of 4800 images with 152,652 annotated pieces. This automated process, owing to the imprecision of the YOLOv5 model, failed to label 948 pieces. These had to be subsequently manually labelled by a person. Most of the undetected pieces were either overexposed or almost entirely obscured behind larger pieces. Figure 3.25 displays samples of images labelled automatically through our annotation system, highlighting its capability under varied conditions.

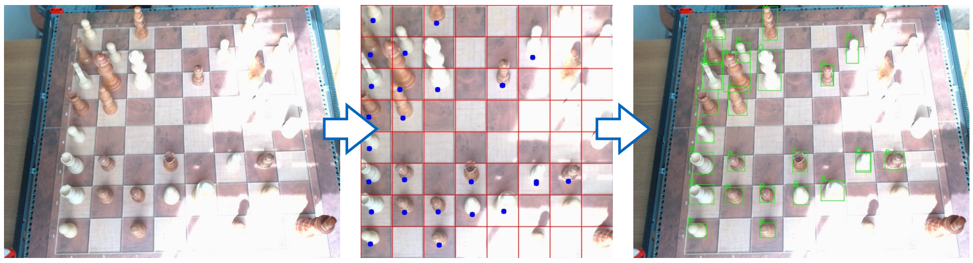
The dataset generated through random placements has a very balanced distribution of pieces, as demonstrated by the annotation heatmap in Figure 3.26. Given the typical distribution of chess pieces, there is an inherent imbalance in the class representation (e.g., eight times more pawns than kings), but YOLOv8 is specifically designed to handle this imbalance. The results achieved in this thesis prove that less frequent pieces were not disadvantaged compared to more frequent ones.



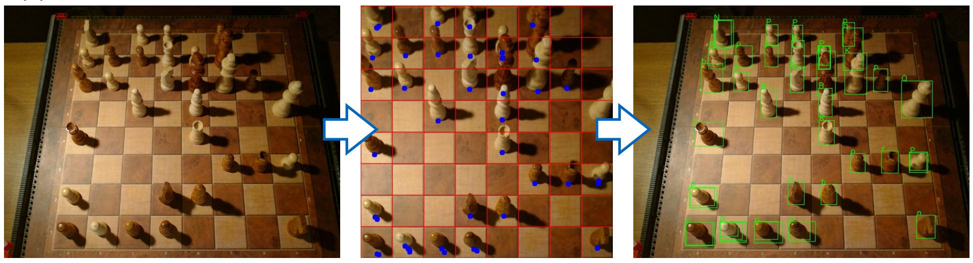
(a) : A standard photo under good lighting conditions - all pieces labelled.



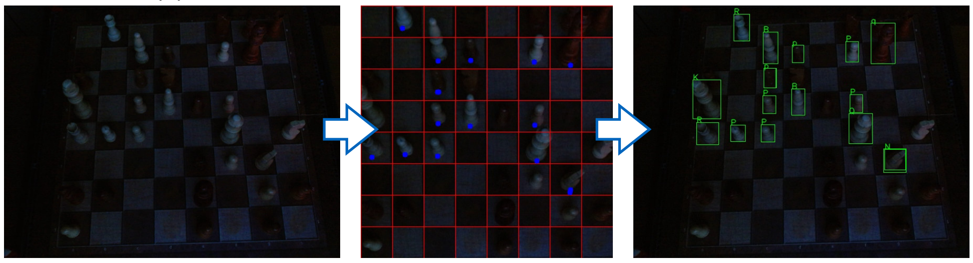
(b) : Photo with high dynamic range - one piece unlabelled.



(c) : An overexposed photo - many pieces unlabelled, hardly recognizable by a human.



(d) : Unbalanced lighting, sharp shadows - all pieces labelled.



(e) : insufficient lighting - many pieces unlabelled.

Figure 3.25: The automated process of labelling images and the achieved results under various lighting conditions. The blue dots at the base of the pieces in the middle image indicate their detection using YOLOv5 trained on the synthetic dataset.

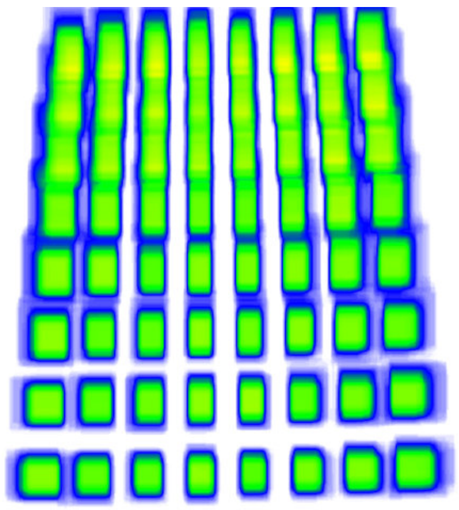


Figure 3.26: Annotation heatmap of the real dataset.

Dataset augmentation

To enhance the properties and size of our dataset, it was expanded threefold through data augmentation. It's essential to mention that the data selected for the validation and test subsets were not augmented. Each augmented image had a 50% chance of being horizontally flipped, underwent a horizontal shear of up to ±5% and a vertical shear of up to ±1%, and adjustments to exposure and saturation within ±5%. In the study presented in [11], researchers managed to retrain a neural network using just two images of a new chess set. They achieved this remarkable result by heavily augmenting these two images. The authors of that study highlighted the shear operation as particularly effective when training chess piece detection since it mimics the natural camera distortion.

The resultant dataset, comprising 12,578 images, was then divided into training (93%), validation (6%), and testing (1%) subsets. Figure 3.27 showcases an original image from the primary dataset alongside its two augmented versions.

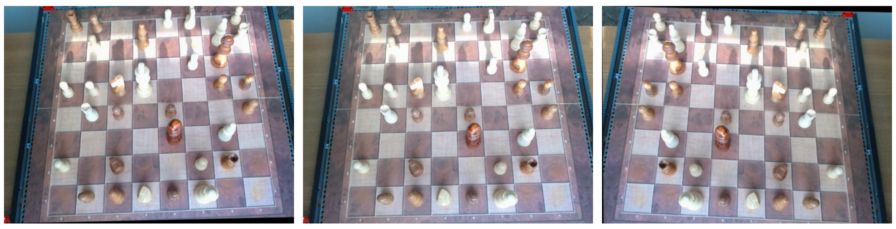


Figure 3.27: The original image (middle) and its two augmentations.

Conclusion

We introduced a highly robust system for the automated generation of real-world chess datasets. This solution could address the frequently mentioned issue among researchers regarding the lack of chess datasets. Only a mere 0.62% of the dataset required manual annotation, primarily for very low-quality images. Thanks to the universal design of our robot's gripper, the algorithm could be easily applied to other chess sets. Furthermore, any robotic manipulator able to reliably grasp and move a chess piece could benefit from this data generation approach. We hope that future works will seize this opportunity, leading to the creation of many more publicly available chess datasets.

In our case, one drawback was the battery life of our robot. It could only sustain the dataset creation process for about 12-16 hours before needing a 6-hour recharge. Unfortunately, the old Li-ion batteries used in our robot can't charge quickly enough to match the high consumption of the motors, as their charging is limited to 700mA at 10V. Another challenge was the frequent failures of our large motors, necessitating a tedious replacement procedure.

3.2.4 Chess piece detector

We trained YOLOv8 on 11,667 training images from the augmented real-world dataset from the previous chapter. The trained model is based on the publicly available YOLOv8m checkpoint [16], which has 25.9 million parameters originally trained on the COCO dataset. Thanks to the use of transfer learning, our model was quickly trained to detect chess pieces (Figure 3.28).

The validation losses begin to stagnate around the 60th iteration while the training losses continue to evolve. This unfavourable trend could suggest overfitting of the neural network to the training dataset. This suspicion is further supported by comparing the 60th and 100th epochs on the testing dataset (i.e., images the neural network has never seen during training) containing over 3000 pieces. While the 60th epoch on the testing dataset made only 4 errors (2 pieces misclassified, 2 false positive detections), the 100th epoch made 7 errors (3 pieces misclassified, 4 false positive detections).

We conducted a comparison of the neural networks YOLOv5 and YOLOv8 trained from publicly available checkpoints YOLOv_n, YOLOv_s, and

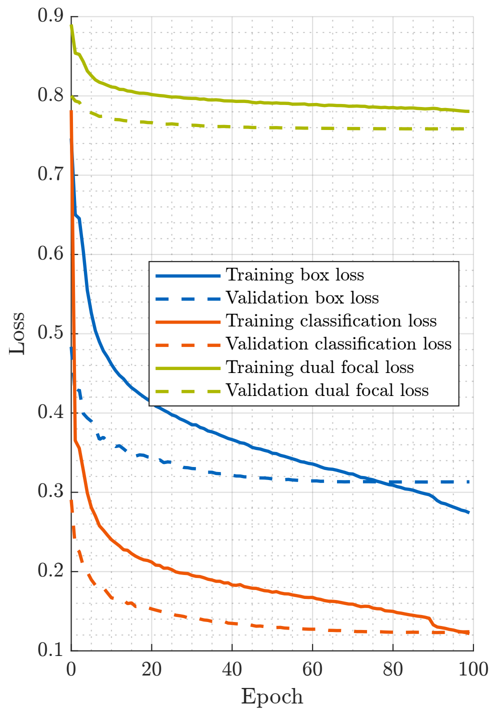


Figure 3.28: Evolution of classification, box, and dual focus losses over 100 epochs of training based on the YOLOv8m checkpoint.

YOLOv_m. The results of the comparison are shown in Table 3.2. We couldn't utilize larger models due to the hardware limitations of our system. The development of validation losses over the 100 training epochs is depicted in Figure 3.29.

	YOLOv5			YOLOv8		
	params	inference	$mAP_{50\text{-}95}$	params	inference	$mAP_{50\text{-}95}$
v_n	1.9 M	1.1ms	0.957	3.2 M	1.1ms	0.959
v_s	7.2 M	2.0ms	0.966	11.2 M	2.1ms	0.967
v_m	21.2 M	4.3ms	0.969	25.9 M	4.8ms	0.969

Table 3.2: Performance comparison of YOLOv5 and YOLOv8 based on checkpoints of various model sizes (n, s, m) after 100 epoch training.

These results indicate that the model size plays a more significant role than the YOLO generation. For our purposes, any of the mentioned models can be used with negligible differences in reliability. Since an inference time of 4.8ms on our control computer was sufficient for our needs, we chose to use the model based on YOLOv8m checkpoint. The normalized confusion matrix for this model is shown in Figure 3.30.

Comparing the confusion matrix 3.30 with the confusion matrix 3.24 from Chapter 3.2.2 reveals that transitioning from a synthetic dataset to a real-

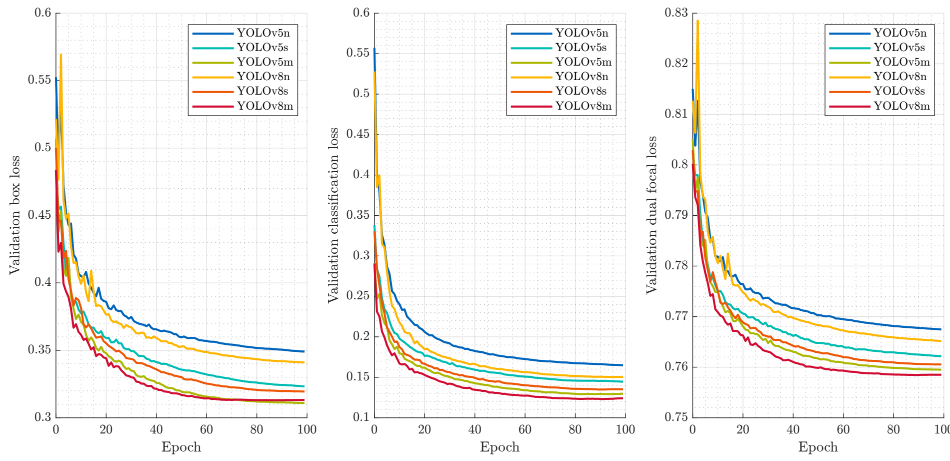


Figure 3.29: The development of box/classification/dual focal validation losses over the 100 training epochs of YOLOv5 and YOLOv8 based on checkpoints of various model sizes (n, s, m).

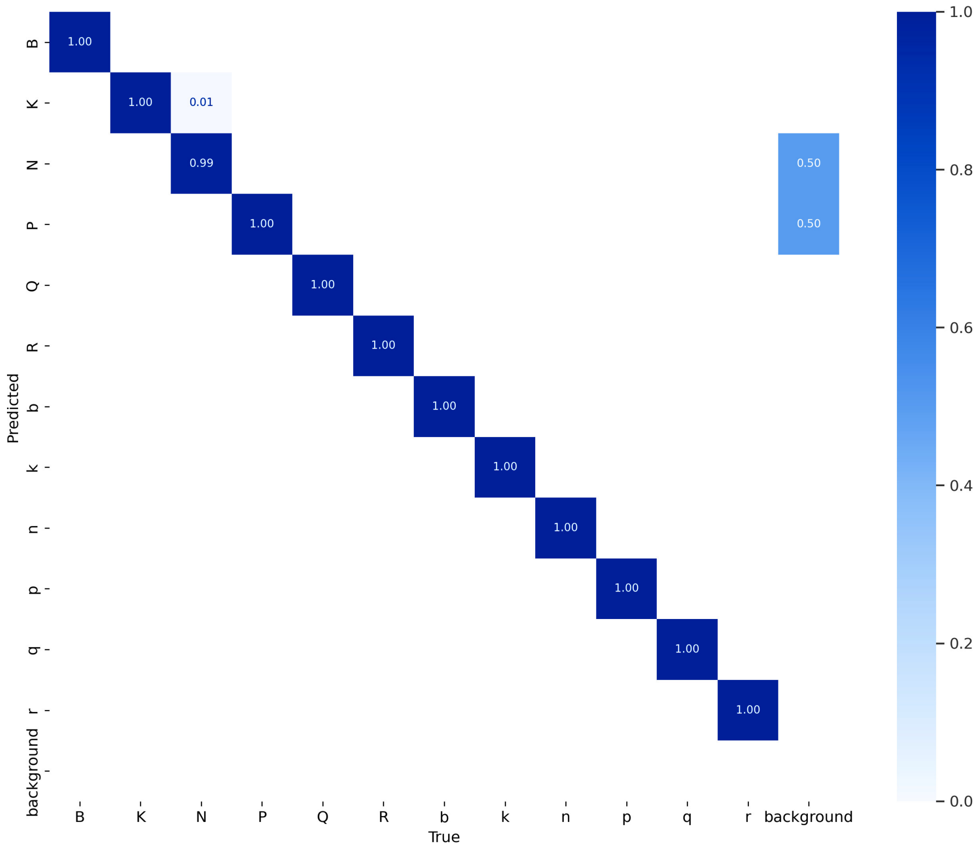


Figure 3.30: Normalized confusion matrix of the trained model based on the YOLOv8m checkpoint, when validated on the testing subset of the real-world dataset. Uppercase letters correspond to the white chess pieces. This figure can be directly compared with Figure 3.24.

world dataset significantly improved training results. Despite our best efforts, the synthetic dataset did not encompass enough image variation. The real-world dataset introduced a countless amount of image imperfections that would be challenging to reproduce in the synthetic dataset generator.

The reliability of the trained model could potentially be further enhanced by a larger dataset, which would also counteract tendencies for overfitting in later training epochs. Nevertheless, given the limited size of our dataset, our achieved results are highly satisfactory (see Figure 3.31). The performance of our system aligns closely with top-tier results from other research studies on chessboard recognition [10, 11].

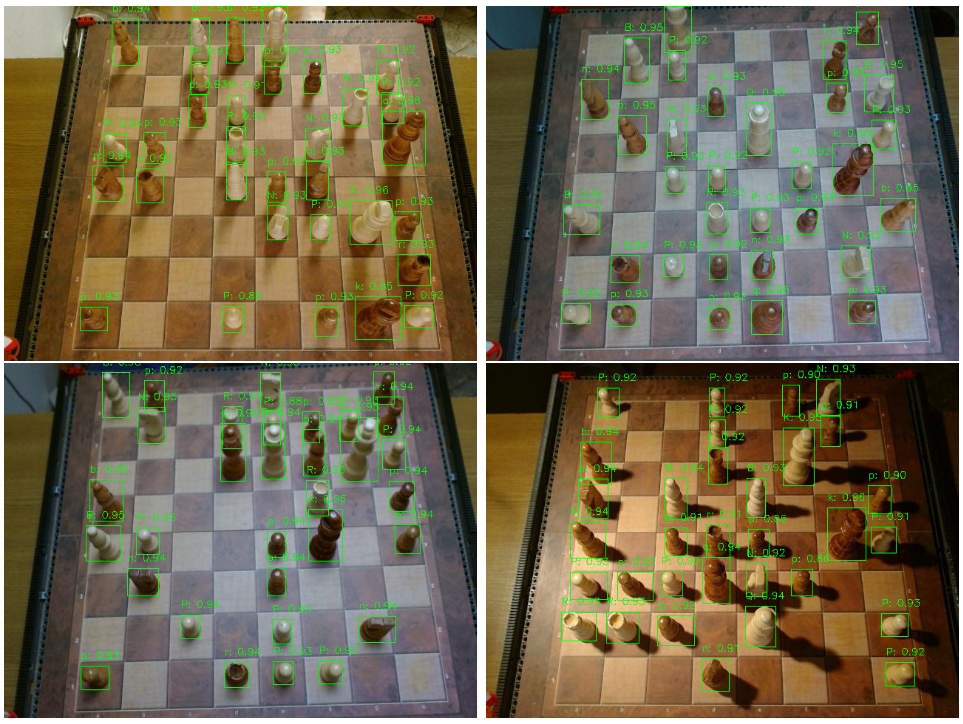


Figure 3.31: Examples of detections from the testing subset of the real-world dataset using a model trained on the YOLOv8m checkpoint. The trained model accurately detects even complex situations caused by poor lighting, misfocus, or overlapping pieces.

3.3 Move detector

Most scientific studies dealing with chessboard detection from camera imagery identify player moves based on recorded changes in the chessboard occupancy. This is often due to the adoption of piece recognition methods that discern only the presence or colour of a piece, not its type. In such cases, the following rules can be used to detect a player's move [4]:

1. No change detected → no move has occurred.
2. Swap of a piece with an empty square → no-capture move.
3. Change in the colour of a piece and disappearance of a piece of the same colour → direct capture move.
4. Two no-capture moves of the same colour → castling move.
5. No-capture move and the disappearance of one of the opponent's pieces → en-passant move.

It's always crucial to verify the legality of the detected moves based on the prior state of the chessboard. Additionally, the fact that each player can have at most 16 pieces, precisely one king, and no more than eight pawns can be used to detect incorrect chessboard recognition. However, these simplistic methods are susceptible to errors and cannot detect complex situations, such as pawn promotion.

Our implemented algorithm for recognizing player moves combines the results obtained in Chapters 3.2 and 3.1. Initially, the algorithm uses the chessboard and piece detectors to assign each square a predicted type and prediction confidence. The piece recognition has a set confidence threshold min_conf of 30%. Squares, where no piece is detected, are thus assigned a min_conf confidence value.

The algorithm generates a list of all legal moves from the previous state of the chessboard. For each legal move, "affected squares" are generated, which can comprise 2 to 4 squares depending on the type of move. The degree of correspondence of all affected squares of the legal moves with the detected chessboard is determined using confidence levels. The legal move with the highest normalized confidence wins - see Algorithm 2.

Algorithm 2: The move detection algorithm

```
Data: previous chessboard state $b$, camera image $img$
Result: highest correspondence move $m_{best}$ or $null$
// 8×8 matrices with detected type and confidence
$T, C \leftarrow$ DetectChessboard($img$);
$m_{best} \leftarrow null$;  $c_{best} = 0$;
for $m \in LegalMoves(b)$ do
    $AS_m \leftarrow$ GetAffectedSquares($b, m$);
    $board_m \leftarrow$ ApplyMove($b, m$);
    $c_m \leftarrow 0$;      // Move correspondence
    for $as \in AS_m$ do
        if $T[as] = board_m[as]$ then
            $c_m \leftarrow c_m + C[as]$;
    end
    $c_m \leftarrow c_m \div |AS_m|$;
    if $c_m > c_{best}$ then
        $c_{best} \leftarrow c_m$;  $m_{best} \leftarrow m$;
end
return $m_{best}$
```

If the algorithm does not detect any legal move, it evaluates whether there has been any change on the chessboard. This step is purely informative and has no more profound significance. If the algorithm detects a change, it notifies the user that an invalid move has been detected. If it doesn't detect any change on the chessboard, the algorithm does not perform any action. The Human Player (see Chapter 4.3) repeatedly attempts to recognize the user's move until it succeeds.

3.4 Hand detector

A hand detection algorithm can significantly enhance the Move detector's reliability and lower the system's overall computational complexity. We leverage the fact that no change can occur on the chessboard without user interaction. The hand detector thus allows tracking the progress and completion of the user's move without the need to repeatedly query the Move detector or use a physical button to confirm the move's end. Moreover, the Move detector may misinterpret the chessboard state when the user partially occludes the camera view with their hand. For this reason, it is desirable to activate the Move detector only after the user's movement has ended.

Many previous research studies have employed hand detection to detect the end of a move and to prevent incorrect move detections. In this section, we will discuss three methods we have tested, outlining their advantages and disadvantages. The last of these methods was implemented in the final system.

3.4.1 Background subtraction

Background subtraction is a technique employed in numerous systems due to its reliability and simplicity. Similar to piece detection using the background subtraction approach (as discussed in Chapter 3.2.1), this method requires a stationary camera and, ideally, a controlled laboratory environment with consistent lighting. Applying this technique to our robot proved challenging. The background subtraction method was sensitive to movements from the robot's flexible structure and surrounding shadows. Most prominently, the user's own shadow frequently triggered false positive hand detections. These issues are illustrated in Figure 3.32.

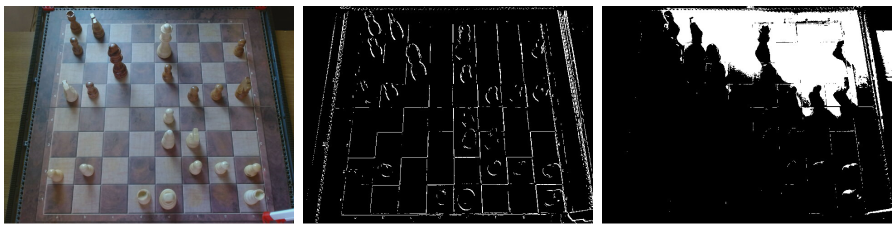


Figure 3.32: The false positive movement detections produced by the background subtraction method. The left image displays the camera output under the given conditions (with the user in front of the chessboard). The middle image illustrates the effects of camera shake, while the right image highlights how the user's shadow is misinterpreted as a movement.

Given that our chess set uses colours similar to human skin tones, the user's hand was typically detected only partially. Amateur chess players often initiate their move by grasping the piece they intend to play with, during which they begin to validate their decisions. In such scenarios, the background subtraction method fails as it gradually adapts to the user's stationary hand. When the hand is removed, the algorithm perceives the chessboard's re-emergence as movement, leading to an undesirable effect. A visualization of this phenomenon is illustrated in Figure 3.33.

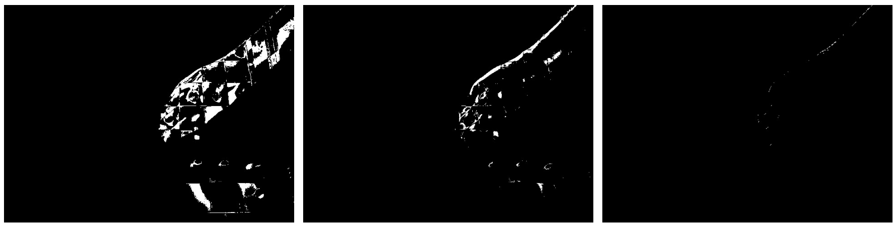


Figure 3.33: The algorithm adjusting over time to the user's stationary hand (from left to right). It can be observed that the hand's shadow is also detected as a movement.

3.4.2 Machine learning

We explored the use of a machine learning approach for human hand detection utilizing Google's MediaPipe Hand Landmark detector [26]. This implementation can reliably detect and track hand movement in a camera image, as demonstrated in Figure 3.34.

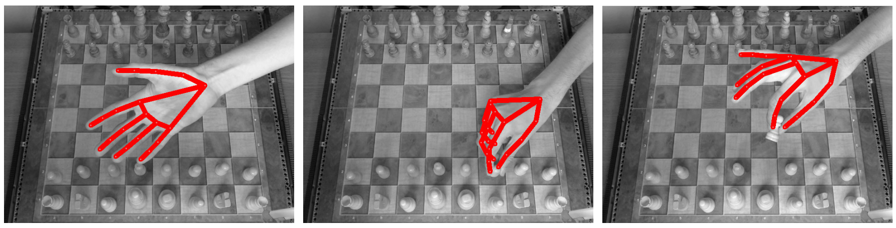


Figure 3.34: The output of Google's MediaPipe Hand Landmark detector.

Unfortunately, even this method was not suitable for our purposes. The MediaPipe Hand detector was primarily trained for gesture recognition, and its reliability significantly decreased when viewing the hand from the top or side. Given the limited field of view of the camera we used, the detector also struggled to identify the hand when picking up pieces from ranks 6 and 7, and it failed to detect the hand entirely when picking up pieces from rank 8. These scenarios are illustrated in Figure 3.35.

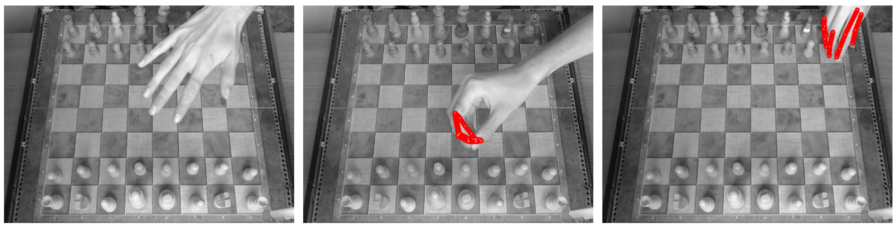


Figure 3.35: Illustrations of described instances where the MediaPipe hand detector struggled or completely failed.

3.4.3 Motion Flow

The final method for hand detection we have tested employs the Gunnar Farnebäck algorithm [27] for calculating the optical flow of an image. Using this algorithm, we obtain an approximation of the magnitude and direction of motion for all image pixels. The normalized magnitude is depicted in Figure 3.36.

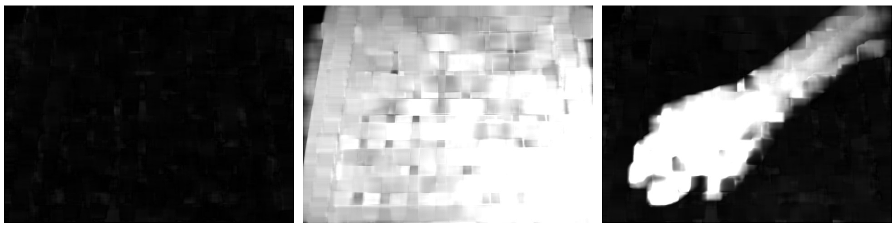


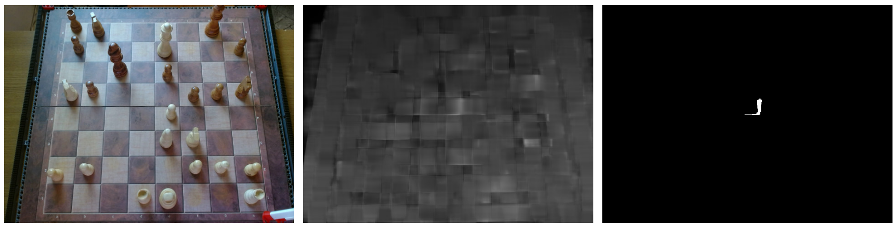
Figure 3.36: Normalized motion magnitude obtained using the Gunnar Farnebäck algorithm. The left image shows the chessboard without interference; the middle image shows the chessboard with a camera shake, and the right image shows the user's stationary hand above the chessboard.

The obtained magnitude is dimensionless, making it unsuitable to use with a fixed threshold to filter out the user's hand. Instead, we employ the interquartile range (IQR) outlier detection to filter the data. This approach reliably differentiates between the background (chessboard) and the foreground (user's hand) under the assumption that the foreground occupies less than 25% of the entire image. We then binarize the filtered image and search for contiguous shapes within it. The normalized area of the largest shape provides the score for the given frame. We calculate a running average score over the last 30 frames. If the average normalized area of the largest shape from the last 30 frames surpasses a fixed threshold (in our case, 0.4%), the frame is labelled as containing a hand.

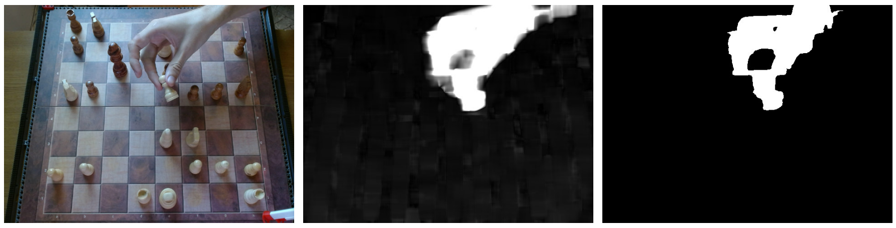
The hand detection method we introduced is capable of detecting a hand under various conditions. Due to its high sensitivity to movement, it can reliably detect a stationary hand placed on a chess piece. It is capable of detecting piece movements in rank 8. Moreover, this method is highly resistant to moving shadows and changing lighting conditions. However, due to the camera's rolling shutter effect, strong camera shakes can sometimes be misinterpreted as movement.

Due to its positive attributes, this method was implemented in the final version of our autonomous chess robot. The Move detector from Chapter 3.3 utilizes the Hand detector to recognize the player's move - it gets activated only after the move is completed. If the Move detector detects no move or an illegal move, it waits for another player interaction using the Hand detector.

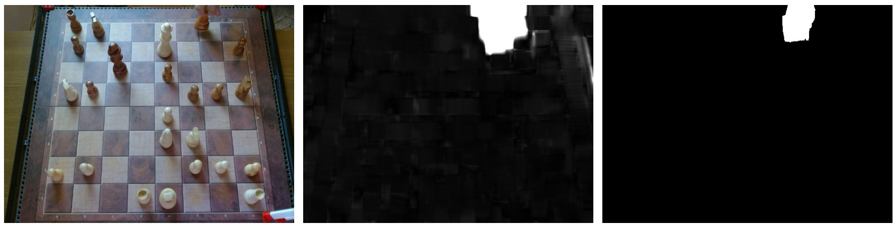
The results of the implemented Hand detector are displayed in Figure 3.37.



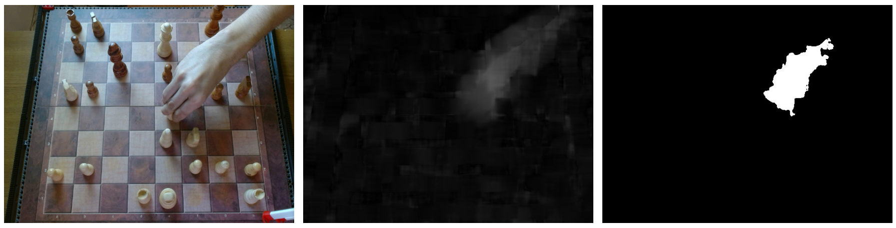
(a) : Chessboard with no hand movement.



(b) : User holding a piece over the chessboard.



(c) : User moving a tall piece in rank 8.



(d) : User holds their hand as still as possible on a chess piece.

Figure 3.37: Visualization of the optical flow based Hand detector's functionality. The left image displays the camera output, the middle image shows the normalized magnitude of movement, and the right image presents the largest contiguous shape after IQR filtering and binarization.

Chapter 4

Implementation

This chapter focuses on the implementation details of our autonomous chess robot. We will break down the overall system, highlighting how different parts work together to make the robot autonomously play chess. Our choice of chess algorithm, the underlying game logic, and the robot's internal algorithm will be discussed in detail. Additionally, we will discuss the implemented motion planner that enables the robot to optimize its path while avoiding collisions with other pieces. Lastly, we will give an overview of the user interface.

4.1 System overview

Given the complexity of the task and the limited performance of the MCUs, a majority of the computations had to be executed on an external control computer (PC), which also runs the GUI (the GUI is not confined to just the control computer and may be run anywhere on the local area network). The implemented system comprises three main modules: the chess-playing robot, the control computer, and the GUI. All these modules communicate with each other over TCP. Figure 4.1 depicts an overview of the modules and their submodules.

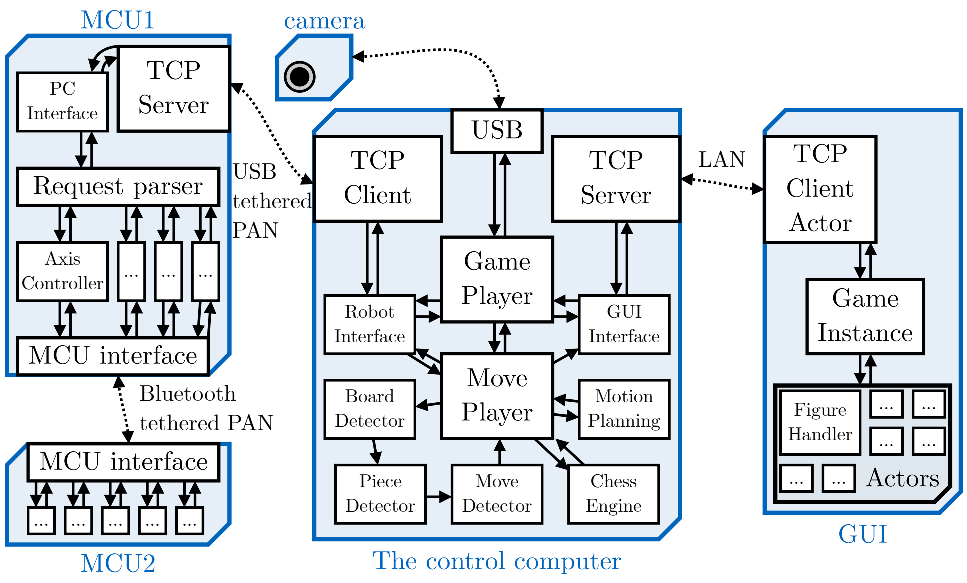


Figure 4.1: An overview of the system, its modules, and the most crucial submodules.

4.1.1 Intermodule communication

The entire system is interconnected using Transmission Control Protocol (TCP). MCU1 establishes its own personal area network (PAN). MCU2 connects to this network via Bluetooth tethering, while the control computer connects through USB tethering. This network operates separately from the local area network (LAN), which the control computer uses to communicate with the graphical user interface (GUI). The camera is connected to the control computer via USB.

The communication framework connecting all modules was designed to ensure maximal independence between the individual modules. This architecture guarantees that each module can respond adequately to connectivity loss. For example, the system can operate without the GUI activated or seamlessly resume gameplay after power restoration if the robot's battery was drained.

MCU1 (host of the PAN) uses a static IP address 10.0.0.1, where it sets up a TCP server for communication. The control computer runs a TCP client that tries to connect to MCU1's server. Once connected, they exchange state information and requests through the "PC Interface" in the robot and the "Robot Interface" in the control computer. Each robot command is treated as a request with its unique ID and arguments. Upon task completion, the robot (MCU1) sends an acknowledgement signal to the control computer, confirming the fulfilment of the request with a given ID. In case of a connection interruption, such as accidentally unplugging the USB cable, the robot autonomously completes buffered requests and then awaits reconnection.

MCU1 and MCU2 communicate over TCP via Bluetooth tethering. They share their actions and synchronize their progress. Simple commands are dispatched to MCU2, such as "light up LED", "rotate the motor at a given speed to a given angle", "report the current motor encoder angle", etc. The "Axis-Controller" submodule running on MCU1 aims to synchronize the movements of all robot axes.

While the MCU1-to-PC connection is responsive and fully meets our requirements (speed: avg. 23.814 Mbps; delay: avg. 1.354ms, max. 3.130ms), the MCU1-to-MCU2 connection is considerably slower. The Bluetooth tethering TCP communication has to handle low transmission speeds and significant transfer delays (speed: avg. 284 kbps; delay: avg. 60.913ms, max. 90.071ms). This caused the motor synchronization between the two MCUs to be highly challenging. The challenges related to slow motor synchronization will be discussed in greater detail in Chapter 4.3.2 about Trajectory Planning).

Both MCUs are powered by the Angstrom 2010.12 operating system, running on the Linux kernel 2.6.33-rc4, with driver support for older external USB Wi-Fi adapters. Unfortunately, at the time of writing, we did not have access to any Wi-Fi adapters compatible with this lightweight operating system.

For communication between the control computer and the GUI, the control computer sets up a TCP server and waits for the GUI client's connection. The computer then updates the GUI with the current game and chessboard state. Simultaneously, the GUI sends back events (similar to requests, but without acknowledgement) to the computer, such as game restart commands. If connection loss occurs, the control computer continues the game while the GUI awaits reconnection.

4.1.2 Camera stream

In our initial endeavours, we aimed to connect the camera directly to one of the MCUs, transmitting images to the control computer via the TCP communication. This section presents our findings and the reasons that ultimately lead us to connect the camera directly to the control computer.

As stated in Chapter 3, the Logitech C920 HD PRO camera was chosen primarily due to its compatibility with the old drivers present in the MCU units. These drivers only support the UVC (USB Video Class) devices and a subset of gspca (Generic Software Package for Camera Adapters) devices. Furthermore, the camera's backward compatibility with USB 1.1, found in the MCU units, made it a preferred choice.

The supported transmission protocols between the camera and the MCU provide raw YUV images or an MJPEG video stream. The raw YUV image transmission is capped at a resolution of 192x144 pixels. The MCU's computational capacity constrained streaming at this resolution to the control computer, resulting in an average of 7 frames per second. It is imperative to note that this small resolution was insufficient for our purposes.

On the other hand, the MJPEG stream from the camera supports a resolution of up to 640x480 pixels. However, the MCU unit requires approximately 3 seconds to capture a single frame, rendering it unsuitable for our requirements. Moreover, the MJPEG introduces compression artefacts, as depicted in Figure 4.2.

Given these constraints, we were ultimately compelled to directly connect the camera to the controlling computer. This configuration enabled us to process images at a resolution of 640x480 pixels at up to 30 fps, devoid of any compression artefacts.

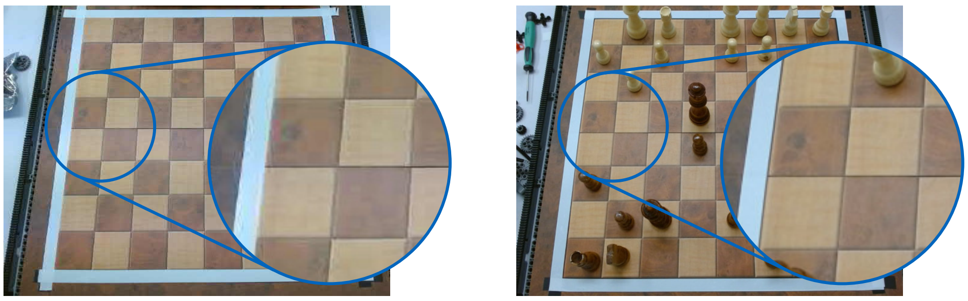


Figure 4.2: Image with artefacts from the MJPEG video stream captured by the MCU1 (left figure) compared to an image taken by the controlling computer (right figure). Both images have an identical resolution of 640x480 pixels.

4.2 Robot algorithm

Due to the highly constrained computational capabilities of the MCUs, the majority of computations are executed on the control computer. Nevertheless, the MCU units handle everything related to the robot's movement. MCU1 secures both the communication with the control computer and the movement synchronization with MCU2. The MCU1 program operates in two phases: axes calibration and request processing (Figure 4.3).

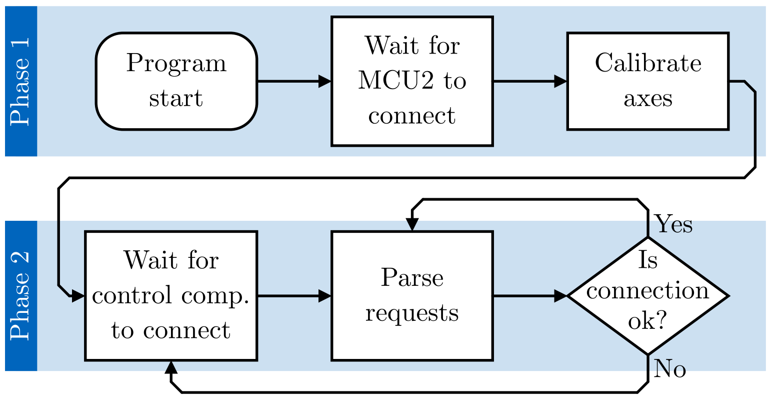


Figure 4.3: A simplified overview of the MCU1 workflow.

Axes calibration

The robot is equipped with seven servo motors. Embedded within these motors are optical encoders, which enable the robot to monitor motor movement. However, these encoders only provide a relative position, meaning their absolute positions are unknown after the program's initialization. A calibration sequence was implemented to avoid manual axis tuning before each game, which autonomously resets the robot to its default position.

Conventionally, end switches are used for the calibration of motorized axes. Unfortunately, we did not have these at our disposal since our choice of parts was limited. For this reason, we deploy a sensorless auto-calibration for 6 out of the 7 motors. This technique leverages the feedback from the motor encoder to detect a motor stall - a situation where the motor is halted due to an opposing external force.

The robot employs PID control for positional control of motors. The PID component values for the medium and large motors are shown in Table 4.1. The integral components are turned off during calibration to prevent unnecessary motor strain.

The robot calibrates individual axes in the sequence listed in Table 4.2. This sequence allows the robot to self-calibrate without knocking over chess

	Proportional	Integral	Derivative
Large motor	4.0	0.04	10.0
Medium motor	8.0	0.04	8.0

Table 4.1: Values used by the positional motor controllers.

pieces on the board. Positional control during calibration enables the synchronization of motors in the Z and Y axes, both of which calibrate two motors simultaneously. Nevertheless, each motor calibrates itself independently of the others. A motor halts its movement if the program detects a sudden spike in the controller error. This purely mechanical solution provides adequate results for our purposes.

Axis	Calibration action
Z (elevation)	Raises to avoid collisions with the chess set
Y (rank)	Reaches mechanical end-stops in front of the chessboard, then returns to home position (rank 1)
X (file)	Calibrates against the wall with the light sensor to allow R-axis calibration
R (gripper rotation)	Calibrates using the light sensor and reflective target on the rotary double gripper
G (gripper opening)	Calibrates to the centre between the actuating pins
X (file)	Moves above the "a" file
Z (elevation)	Lowers to the default position

Table 4.2: Robot's calibration sequence with individual actions of each axis in the order of their execution.

The only motor that uses sensors for its calibration is the R-axis medium motor, which rotates the dual gripper. The use of sensors was necessary, as the dual gripper can rotate indefinitely around its axis (thus, mechanical stops are unfeasible). Consequently, we employ a reflective target affixed to the rotary dual gripper and a light sensor with a built-in LED mounted on the robot's frame. After the X-axis calibration, the light sensor can detect the reflective target. In the first step, the robot quickly rotates the gripper by 360°, storing the position with the highest reflectivity. In the second step, the algorithm, within a 45° radius of this high reflectivity point, executes eight slow and precise sampling rotations, finetuning the relative position of the reflective target and sensor. After the R-axis calibration, the light sensor is permanently disabled.

Request processing

The robot currently supports three types of requests (commands): utility actions, legacy controller, and path controller. Utility actions include changing the colour of the robot's LED indicators and sound signalling.

Legacy controller

The legacy controller has a series of commands allowing the robot to grip and release a piece on a specified square. This controller operates on a very simple principle where the robot moves above the pieces and lowers only for the actual gripping or releasing of a piece. Thus, there's no need to detect potential collisions.

For movement, the legacy controller uses the submodule called "Axis controller", which synchronizes the individual axes. The Axis controller first inquires all axes for a time estimate to complete the planned movement. The worst of these time estimates (unless the user specifies otherwise) is used to set the speeds for all axes. The previously discussed motor position regulators then execute the planned movement (Figure 4.4). A motor's movement is considered complete if the motor approaches within an epsilon distance of the desired position, even though its regulators continue to hold the position.

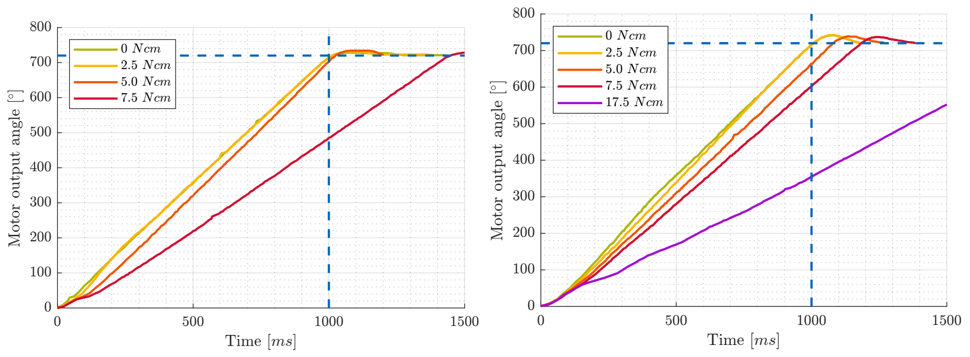


Figure 4.4: Response of the regulator to a request for a 720° motor movement in 1 second under varying loads. On the left is the response of the medium motor, and on the right is that of the large motor. The results correspond to the values from Figure 2.3. It should be noted that a speed of $720°/s$ is the limit for the large motor.

Path controller

The Path controller builds upon the results of the Legacy controller. It takes a sequence of 4D points specifying the positions of its four main axes (X, Y, Z, and R) over time. In our application, we utilize this in conjunction with motion planning, which we delve into in Chapter 4.3.2. The Path controller iteratively processes all input points and adjusts the speeds of individual axes so that their movement follows a straight line between a given pair of points (linear interpolation). The outcome of this system is precise control of movement across all axes, as can be observed in Figures 4.5 and 4.6.

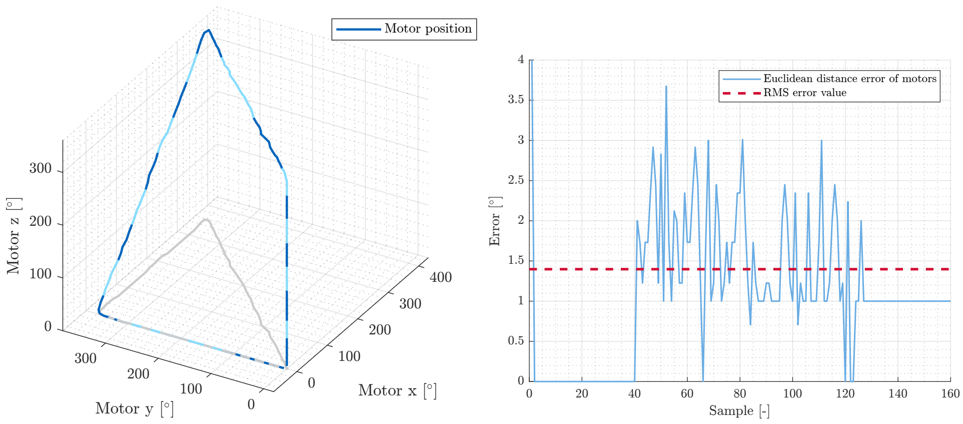


Figure 4.5: Response of the Axis controller following a straight-line path. All of the motors were controlled through a single MCU. The X, Y and Z axis motors were limited to 720, 360 and 180°$/s$ speeds, respectively. The changing colour of the line indicates 1-second intervals (i.e. changing speed).

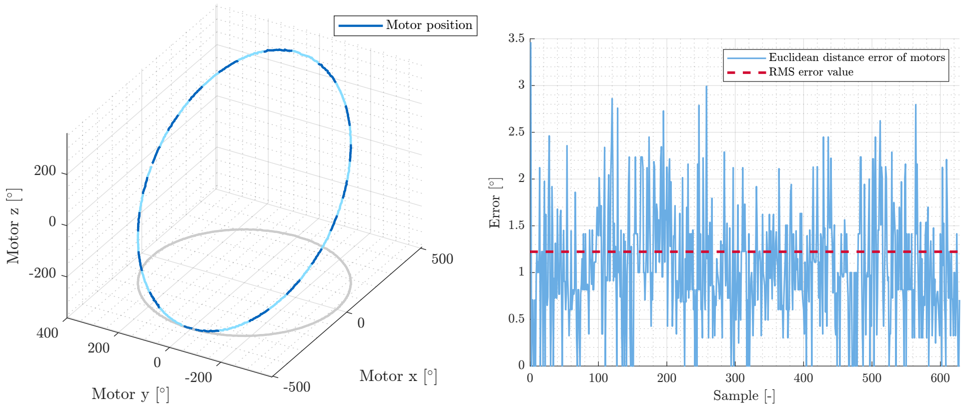


Figure 4.6: Response of the Axis controller following an elliptical path. All of the motors were controlled through a single MCU. The X, Y and Z axis motors were limited to 720, 360 and 180°$/s$ speeds, respectively. The changing colour of the line indicates 1-second intervals (i.e. changing speed).

Since the Bluetooth tethering used for synchronizing motor movement with MCU2 provides a slow connection, we were ultimately forced to limit the sample rate of the Axis Controller to 5 samples per second. This precluded us from tracking complex paths containing numerous points. As a result, we simplify the path generated by the motion planning algorithm by excluding all points which are not required to preserve a collision-free path. We anticipate that more reliable communication between the MCUs would address this issue.

4.3 Control computer aglorithm

The algorithm in the control computer is the core of the presented system. It runs the entire game logic, recognizes and generates moves for both players and coordinates the robot and the GUI. Figure 4.7 shows a simplified workflow of the control computer algorithm.

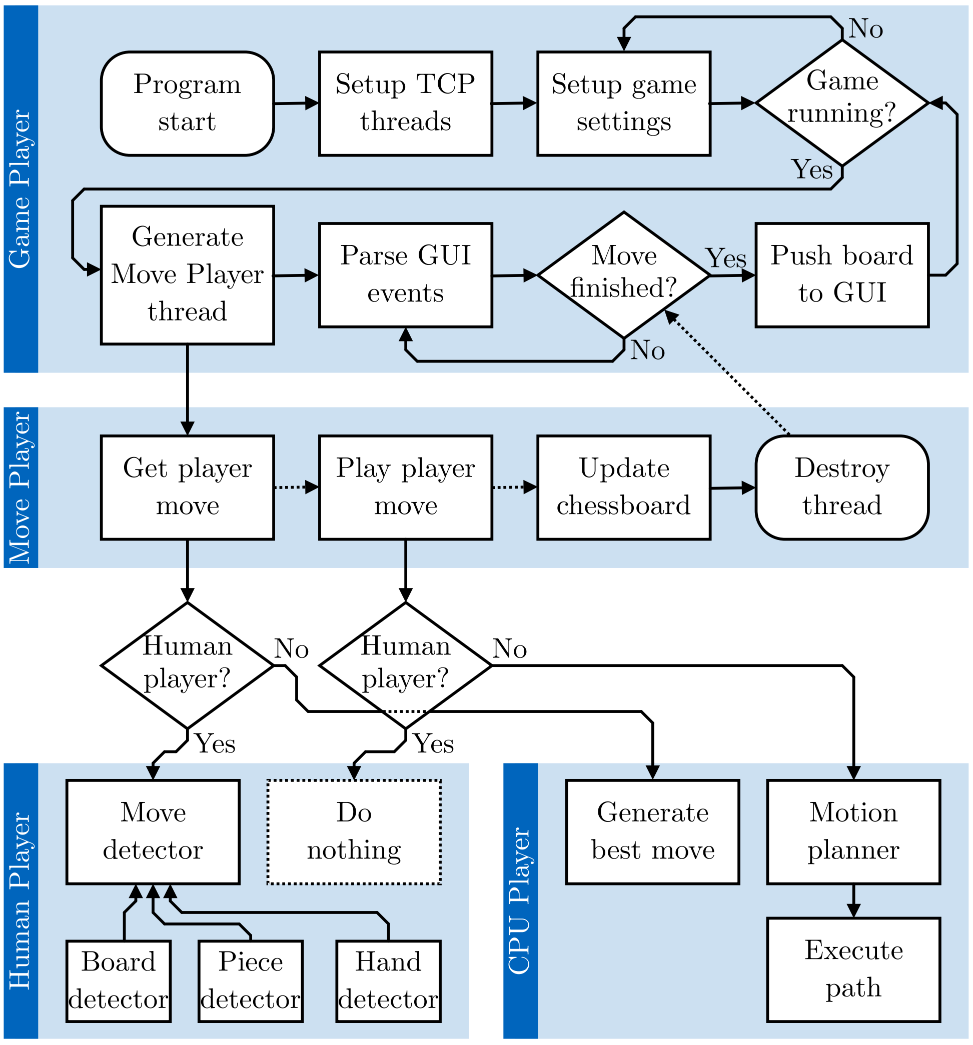


Figure 4.7: A simplified workflow of the control computer algorithm.

The main thread of the algorithm called "GamePlayer" initiates all other threads (for example, threads for communication, camera image processing, etc.) and sets up the default game settings. After initialization, GamePlayer launches its main loop, where it coordinates the course of the game by setting up and starting a new "Move Player" thread for each ply and coordinates its results with the GUI. While the "Move Player" executes the move, the GamePlayer responds to events sent from the GUI. After each move, the

GamePlayer checks if the game has been completed.

Move Player thread is specifically created for each ply. Upon its creation, it is given the current player object and the state of the chessboard. Move Player first obtains the move chosen by the player, then plays this move, records the move on the chessboard, and subsequently, this thread terminates. The updated chessboard is further processed by the Game Player. Thanks to object-oriented programming, both players (human/robot) behave the same despite their implementations being very different.

Human Player

The Human player utilizes algorithms from Chapter 3. Thanks to these robust image recognition methods, the Human player is capable of recognizing a move made on the physical chessboard and transferring it to its virtual representation. The Human player does not perform any further actions, as the user has already physically moved the piece.

CPU Player

The Robotic player (internally nicknamed the CPU player) obtains its move using the chess engine from Chapter 4.3.1, which generates a move based on the set difficulty. Playing a move for the CPU player means physically moving the pieces according to the generated move. In the case of a standard piece move, it's a relatively simple movement. However, there are more complex chess moves, such as castling, which consists of several sub-moves. We distinguish four different types of move, which are listed in Table 4.3.

Move type	Sub-move sequence
No capture	$grab1 \rightarrow release1$
Direct capture	$grab1 \rightarrow grab2 \rightarrow release1 \rightarrow drop2$
Castling	$grab1 \rightarrow grab2 \rightarrow release2 \rightarrow release1$
En-passant	$grab1 \rightarrow release1 \rightarrow grab1 \rightarrow drop1$

Table 4.3: An overview of the generated sub-move sequences for each move type. The *drop* sub-move removes a piece from the chessboard. The number after each sub-move represents the gripper used to perform this action.

The Robotic player then employs motion planning from Chapter 4.3.2, which plans collision-free transitions between the individual sub-moves. The planned trajectory, in the form of a series of 4D points, is sent to the robot via the "Robot interface", where this movement is executed using the "Path controller".

4.3.1 Chess engine

One of the critical decisions in the development of our chess robot was selecting the right chess engine. A chess engine is a computer program/algorithm that aims to generate the strongest possible move based on the current state of the chessboard. Given the long-term and extensive development of chess engines by the research community, creating an engine that would surpass state-of-the-art solutions was beyond the scope of this thesis.

Given that we didn't want our robot to be easily defeated by a human opponent, our choice naturally gravitated towards Stockfish [28]. This engine consistently ranks at the top of leaderboards in all chess engine competitions. In fact, Stockfish gained its 6th consecutive SuperFinal title in the Top Chess Engine Championship in April 2023 [29]. What's more, Stockfish is an open-source and platform-independent chess engine. Stockfish delivers top-tier performance while being resource-efficient, with the option to adjust the state tree search depth, balancing robustness against computational complexity.

This chess engine allows users to choose one of the 20 difficulty settings. Reducing the difficulty level gives the human user a chance to win, which can be desirable in certain situations, for instance, if the user uses the robot to train his chess-playing skills.

4.3.2 Motion Planning and collision avoidance

In robotics, the ability to seamlessly navigate through complex environments devoid of collisions is a highly researched topic. Collision-free motion planning is computationally intensive, especially when operating in high-dimensional spaces. Finding a collision-free trajectory with real-time performance often necessitates the deployment of advanced motion planning algorithms.

We aimed to avoid the conventional approach where the robot manoeuvres exclusively above the chess pieces to circumvent potential collisions, even when such precautions are unnecessary. This simplistic motion planning approach, found among many other chess-playing robots, is also implemented in our robot as the "Legacy controller" from Chapter 4.2. While undeniably functional and reliable, such movements are often perceived as unnatural and tend to be less efficient (for instance, when advancing a pawn).

For these reasons, we decided to use the sPRM (Simplified Probabilistic Road Map) motion planning algorithm, which is probabilistically complete and asymptotically optimal. The motion planning is executed in a 3D virtual environment where the chessboard is represented as a 2D plane, and the individual chess pieces are represented as cylinders of corresponding heights. The rotary dual gripper is represented as a rotating rectangular cuboid in the space above the chessboard, mimicking its real-world motions along the X, Y, Z, and R axes. The described virtual environment is illustrated in Figure 4.8.

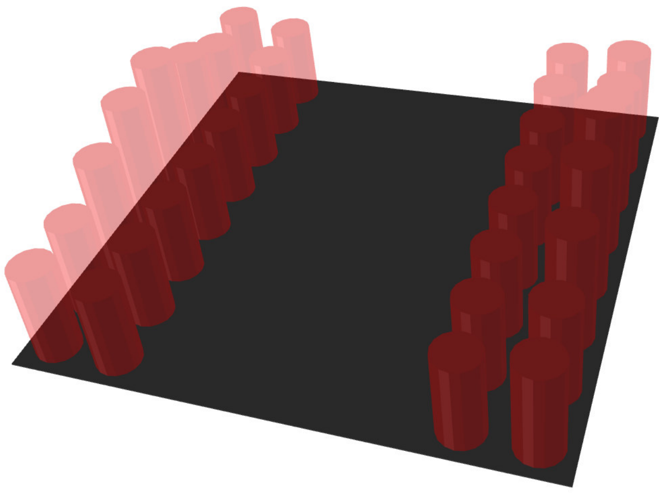


Figure 4.8: The simulated virtual environment for motion planning composed of simple geometric shapes representing the chessboard and individual chess pieces.

The sPRM algorithm (Algorithm 3) constructs a roadmap by randomly sampling points within the robot's collision-free configuration space (in our case, 4-dimensional). Subsequently, it establishes collision-free paths between neighbouring samples in a specified radius, wherever feasible. Through this method, the algorithm systematically explores the robot's configuration space, aiming to find a path that connects the robot's initial and target configurations. While the algorithm is probabilistically complete, meaning that given a sufficiently large sample size, it will find a path if one exists, we have modified the sPRM to guarantee a solution regardless of the sample size. We leverage the fact that the chess pieces are always accessible from the top and that there is a safe height at which the gripper can move without collisions. By adding points above the initial and target configurations at a safe height, we obtain an always feasible fallback solution reminiscent of the "Legacy controller".

Algorithm 3: The modified sPRM algorithm

```
Data: $q_{init}$, $q_{targ}$, number of samples $n$, radius $\rho$
Result: Probabilistic roadmap $G = (V, E)$
$V_{init} \leftarrow \{q_{init}, q^{sh}_{init}, q^{sh}_{targ}, q_{targ}\}$;      // $sh$ = safe height
$V_{samples} \leftarrow \{SampleFree_i\}_{i=1,\dots,n}$;
$V \leftarrow V_{init} \bigcup V_{samples}$;
$E \leftarrow \{(q_{init}, q^{sh}_{init}), (q^{sh}_{init}, q^{sh}_{targ}), (q^{sh}_{targ}, q_{targ})\}$;
for $v \in V$ do
    $U \leftarrow \text{Near}(G = (V, E), v, \rho) \setminus \{v\}$;
    for $u \in U$ do
        if $CollisionFree(v, u)$ then
            $E \leftarrow E \bigcup \{(v, u), (u, v)\}$;
    end
end
return $G = (V, E)$
```

For collision detection, we needed a fast algorithm capable of detecting collisions between two groups of convex objects. Initially, we utilized the RAPID [30] (Robust and Accurate Polygon Interference Detection) collision checking library. However, we later transitioned to Trimesh [31]. Trimesh is a Python library for working with triangular meshes. Trimesh implements the FCL [32] (Flexible Collision Library) for collision detection. FCL is an open-source collision detection program that employs numerous optimization methods to achieve state-of-the-art performance. Trimesh, FCL, and RAPID were tested under identical conditions, detecting collisions between two cuboids in an otherwise empty world. The results of this test can be found in Table 4.4.

Detector	Collision evaluations per second
RAPID	70,949.74
Trimesh	224,035.99
FCL	236,037.44

Table 4.4: Performance evaluation of the tested collision detectors. Both Trimesh and FCL deliver over three times the performance compared to the originally used RAPID library.

To find the shortest path in the graph obtained using the sPRM algorithm, we utilize the popular A* algorithm. Each edge between individual samples in the configuration space is evaluated based on its time requirement. The time requirement is calculated as the maximum time the robot's axes require to transition between configurations, given the current maximum speed setting of the axes. The path with the lowest cost, meaning the lowest time requirement, is chosen.

Due to a limited neighbourhood radius ρ, the obtained path often contains many redundant points. Before sending the path to the robot's Path controller, we simplify the trajectory, eliminating all non-terminal points from the path whose removal does not cause a collision. Simplifying the path generally results in a more natural and efficient movement. Simplifying the path also aids the robot's Axis controller, which is constrained by the transfer speed of Bluetooth tethering between the MCU1 and MCU2 (see Chapter 4.1.1). The output of this algorithm is visualized in Figure 4.9.

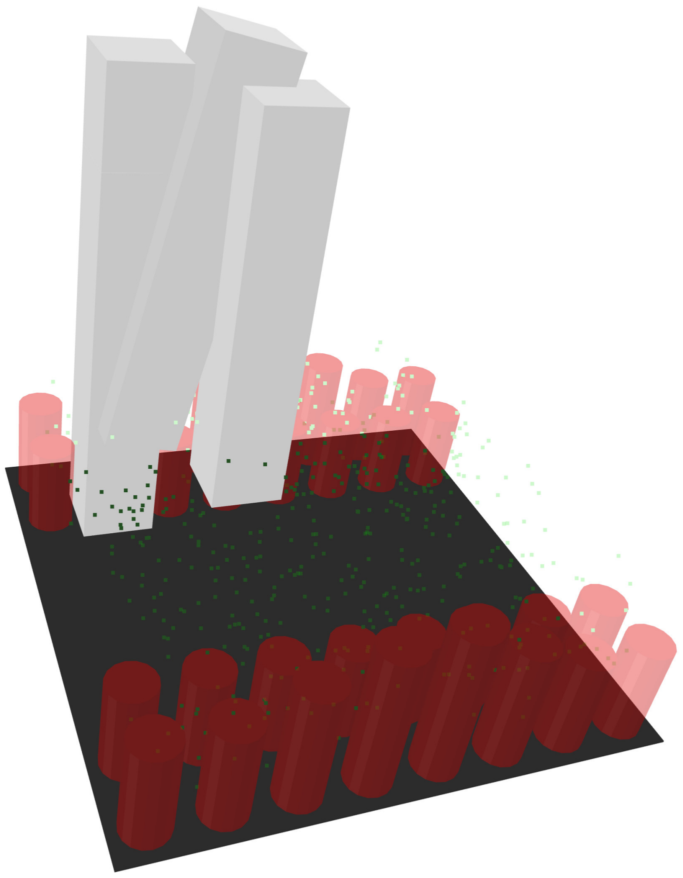
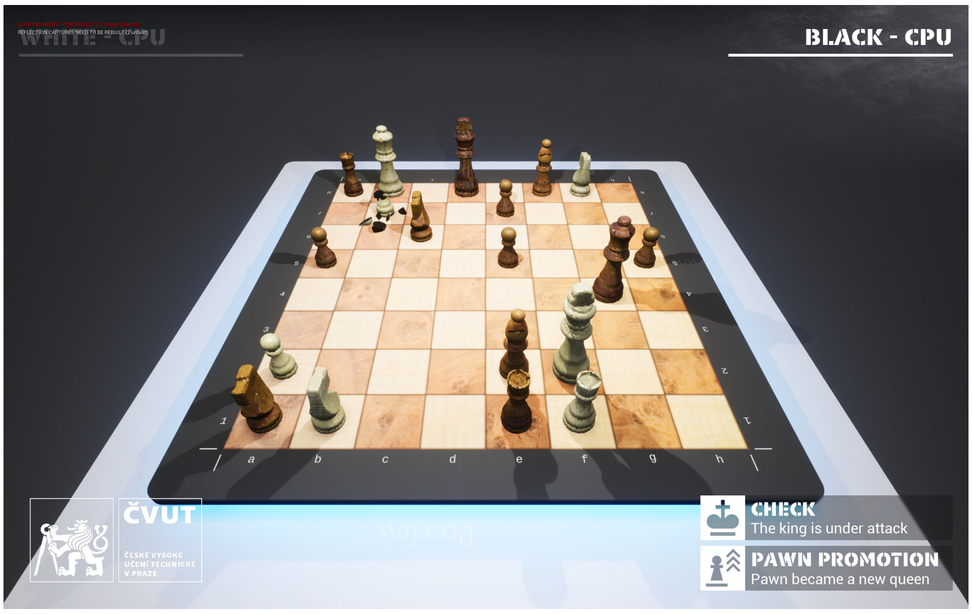


Figure 4.9: Visualization of the output from the implemented motion planning algorithm. Green points represent the sampled configurations. White cuboids represent the configurations of the double gripper from the output path. The configuration on the right is the initial setup of the gripper, aiming to place the piece on the B7 square (left configuration). The algorithm avoids colliding with the piece on the C7 square by adding the middle configuration with a slight gripper rotation.

4.4 Graphical user interface

The implemented graphical user interface allows the user to view the current state of the game and the chessboard. We aimed to create a modern-looking easy to understand interface that can be utilized by both the player and the spectators. Among other things, the interface displays the current player's colour and notifications about critical in-game events, such as a check. The final implementation of the user interface is shown in Figure 4.10.



CHECK
The king is under attack
PAWN PROMOTION
Pawn became a new queen

Figure 4.10: The graphical user interface displaying the state of the chessboard after the white player's pawn has moved to square B8. Notifications shown in the bottom right corner of the screen warn the current player (highlighted at the top of the screen) about the white pawn's promotion and the threat to the black king. The white pawn, having been promoted to a queen, falls apart.

Similar to the synthetic dataset generator discussed in Chapter 3.2.2, the GUI was implemented in Unreal Engine 4.27. Communication with the server on the control computer is facilitated by a TCP client implemented as one of the actors in Unreal Engine. By "actor", we refer to any object that can be placed into the virtual world in Unreal Engine - this includes all assets as well as a significant portion of the scripts.

All information received by the TCP actor is passed on to the Game Instance, which is a high-level manager object that runs throughout the duration of the game. The Game Instance forwards parts of this information to individual actors who are responsible for spawning/moving/destroying pieces, managing notifications, rendering the GUI, changing settings, and more. The Game Instance also prepares responses that are sent back to the control computer, such as user-triggered events (e.g., changes in the settings).

To enhance performance, we utilize models of chess pieces that contain only 0.1% of the triangles from the original 3D scanned meshes. Additionally, a level of detail (LOD) system was implemented for all models, selecting the level of rendered detail based on the model's distance from the camera. We employ a combination of static lights (lighting the chessboard, blue backlighting under the chessboard) and dynamic point lighting, which creates sharp shadows behind the pieces.

A dynamic piece destruction system was implemented to increase user engagement. Every captured piece becomes a breakable object, and the collision with the attacking piece (as it moves in to capture the square) shatters it into many precomputed fragments. These smaller fragments gradually disappear after 30 seconds to maintain clarity on the chessboard. In situations like pawn promotion, where a piece ceases to exist without being captured, the piece shatters without external force applied (see Figure 4.10).

Chapter 5

Results

As described in the preceding chapters, the autonomous chess-playing robot was subjected to a series of tests to evaluate its performance in real-world settings. This chapter presents the results obtained from these tests, highlighting the system's capabilities, strengths, and areas for improvement.

Board detector

The implemented line-based chessboard detector was tested on the real-world dataset from Chapter 3.2.3. Out of the total of 5,032 images, the algorithm was able to detect the chessboard in 4,814, translating to a detection accuracy of 95.668%. Most of the discarded images were of inferior quality. A significant advantage of our algorithm is its ability to detect failures or misalignments with 100% accuracy.

Upon closer analysis, we found that out of the 218 automatically discarded images, 12% were discarded due to the absence of a chessboard grid, 74% were discarded due to template matching failure on the candidate grids, and the remaining 14% were discarded because of 100% piece detection failure (the piece detector is not part of the board_detector, it is used as a failsafe).

Of the cases where the chessboard grid wasn't found, 57% were due to complete darkness (lights were turned off in the laboratory), and 33% were caused by severe camera defocus. The remaining 10% remained unidentified for unknown reasons.

The most frequent cause of failure was a mismatch during template matching. 35% of these mismatches were caused by severe camera defocus, 29% by uneven chessboard illumination (HDR issues), 28% remained unidentified for unknown reasons, 6% of the images were mistakenly out of frame (misaligned camera), and 1% were taken in complete darkness.

The failure to detect pieces was attributed to poor focus in 86% of cases, 8% were due to uneven chessboard illumination (HDR issues), and 5% were taken in complete darkness.

Images with poor focus can be detected using the introduced gradient-descent algorithm from chapter 3.1.1; however, discarding these images might be more appropriate in our case. A potential solution would be to utilize a manually focused camera. Uneven chessboard illumination issues can be addressed by purchasing a camera with HDR support.

Excluding the images discarded by the piece detector (as the piece detector is applied after the chessboard detector as a failsafe) and images for which our algorithm isn't responsible (out-of-focus, 100% darkness, out of frame), only 95 failed detections remain, giving an algorithm accuracy of 98.112%. If we also disregard images with uneven lighting that could be resolved with an HDR-supporting camera, we're left with 48 failed detections, achieving an overall system accuracy of 99.046%.

The measured mean processing time for the real-world dataset was 407 ms per image, with a maximum recorded time of 658 ms. Figure 5.1 displays a histogram of the processing time distribution for images from the real-world dataset.

These findings are consistent with our in-game testing results. During gameplay under regular lighting conditions, we did not observe any instances where the chessboard detection system failed. In the event of a failure, the board detector identifies the error, and the Move Detector retries its chessboard detection attempt until successful. An example of images where the chessboard detector failed is shown in Figure 5.2.

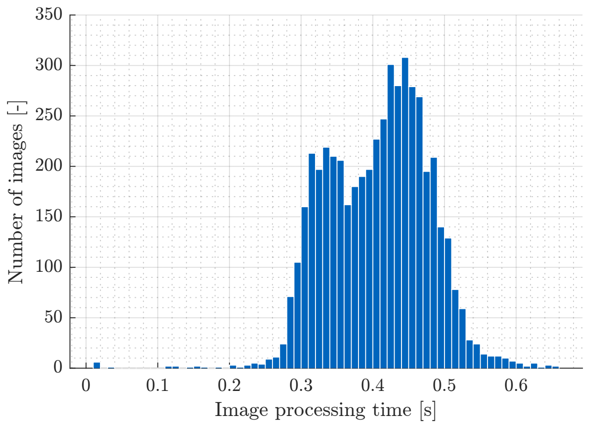


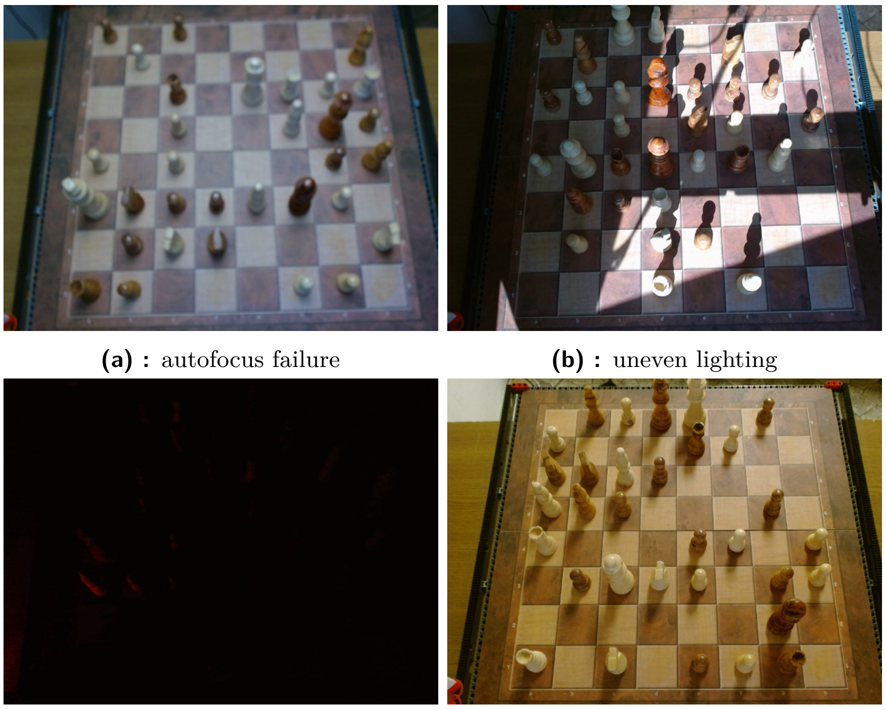
Figure 5.1: Histogram of the processing time distribution for images from the real-world dataset.

Piece detector

The results of the Piece detector have already been discussed in Chapter 3.2.4. Using the real-world dataset, we trained the YOLOv8 neural network, achieving impressive results (refer to the confusion matrix in Figure 3.30). Under normal lighting conditions, it's scarce for the pieces to be undetected or misclassified. In extreme scenarios (e.g., overexposure), a piece may be misclassified. However, in our implementation, the Move detector typically rectifies this by either deeming the move as invalid or deducing the piece's colour and type based on the prior state of the chessboard.

Move detector

The implemented Move detector was tested over 12 games with the chess robot, during which we did not record a single instance of move misidentification (Figure 5.3). Throughout these games, the algorithm alerted the user about an invalid move multiple times, most commonly when the user overlooked a check (despite the GUI warning) or accidentally played with a piece of the wrong colour. The Move detector is capable of detecting invalid moves, but it cannot detect cheating. Suppose a user makes a valid move while simultaneously making an illegal adjustment to the board (removing a piece, moving enemy units, etc.). In that case, this illegal move is ignored by the algorithm. The measured mean processing time of a single image is 606 ms.



(a) : autofocus failure **(b) :** uneven lighting

(c) : darkness **(d) :** unknown (template matching)

Figure 5.2: An example of images where the chessboard detector failed.

Path planning and path execution

The implemented sPRM collision-free path planning algorithm elevates the robot's movements to a new level. By moving the robot in a 4-dimensional state space, it removes the robotic feel that many users initially expect from our system.

Unfortunately, we weren't able to fully utilize path planning. The limited communication bandwidth between MCUs restricts us to primitive paths with a minimal number of waypoints. Moreover, we had to limit the number of sampled state-space points to 200 per path planning. Despite our best efforts, the single-threaded Python implementation was not fast enough to sample more points.

To accelerate path planning, we use a unique system where the path is planned during the robot's actual movement. With the limited number of sampled points, the path is generated faster than the robot can execute it, seamlessly masking the fact that the robot isn't aware of its full path beforehand.

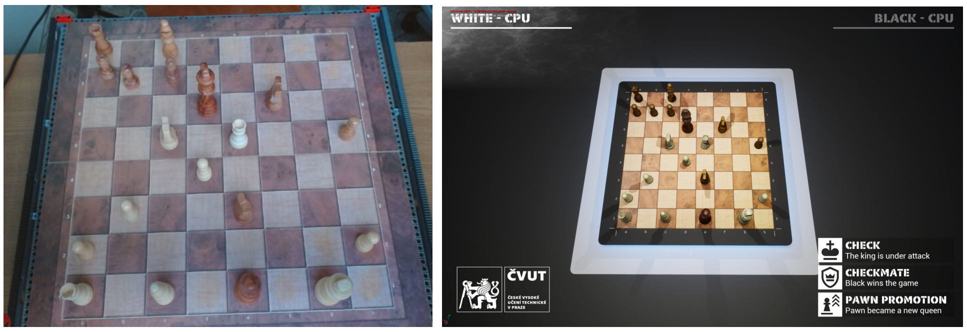


Figure 5.3: A snapshot from the chess robot's camera (left image) and a screenshot of the GUI (right image) after playing an entire chess game using the Move detector.

For future work, we aim to refine the Path planner. Potential improvements include optimizing the distribution of sampled points and transitioning to a multi-threaded implementation. We would also like to replace the Bluetooth tethering between MCUs, allowing us to send more complex paths to the robot.

Except for a relatively high failure rate of the large motors, the path execution is flawless. After all, this system was used to generate a dataset of over 5,000 images autonomously. The implemented AxisController faithfully follows the path created by the Path Planner. A standard no-capture move typically takes about 25-35 seconds to execute. Captures generally take around 100-110 seconds. When the robot plays against a human opponent, returning to the camera position usually takes less than 10 seconds.

User feedback

The system introduced in this thesis has already been presented at public events such as MakerFaire Prague 2023 (Figure 5.4). The feedback from users was overwhelmingly positive. The most frequent suggestion was incorporating a dynamic camera in the GUI to follow events on the chessboard. Users also desired an option to show legal moves and other quality-of-life improvements. On the other hand, the GUI was praised for its modern appearance and real-time representation of the chessboard's state. We plan to implement the suggestions from users in the following GUI update.

A typical user reaction was to test the system with illegal manipulations of the pieces. Some even took pieces directly from the robot's gripper, which unfortunately cannot be detected. Users typically appreciated the seamless gameplay and reliable image recognition.

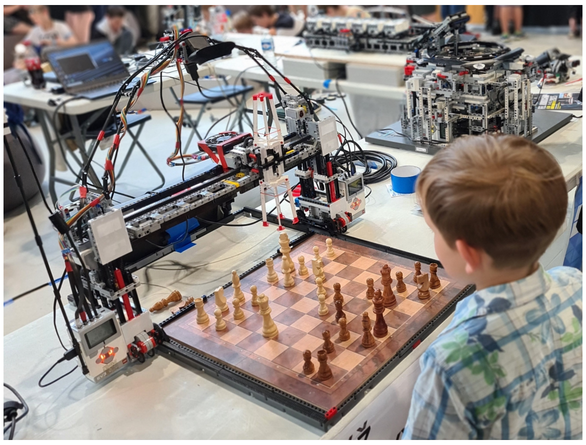


Figure 5.4: Presentation of the developed chess-playing robot at the Maker Faire 2023 event.

Chapter 6

Conclusion

This thesis introduced the challenges associated with the design and development of an autonomous robotic system capable of autonomously playing chess against a human opponent. At the outset, we established several constraining conditions that significantly complicated the implementation process. Our developed system operates with a monocular RGB camera mounted on the robot's moving frame and a wooden chessboard with non-contrasting colours. The focus of this thesis was on system robustness; we presented solutions capable of functioning under diverse lighting conditions.

Chapter 2 delved into the various decisions made during the design and implementation of the presented robotic manipulator with four degrees of freedom. We introduced the individual components that constitute the robot, including the dual MCU system responsible for governing and synchronizing the robot's movements. A unique solution was presented for gripping two chess pieces at once using a rotary dual gripper mechanism. Lastly, we discussed the actuators employed and the challenges associated with them.

Chapter 3 introduced the reader to the challenges of computer vision related to recognizing a player's move from a chessboard image. We discussed a broad range of existing solutions and presented several algorithms we implemented for the detection of the chessboard, chess pieces, player's moves, and the presence of the player's hand in the camera's view. All implemented algorithms are compared, tested, and their results are shared with the reader. An integral part of this chapter was a detailed description of creating synthetic and real-world datasets. The output of our work includes a synthetically generated dataset comprising 5060 images suitable for object detection and segmentation tasks and a real dataset of 4800 images suitable for object detection tasks.

In Chapter 4, we delved into the intricacies of our implementation, spanning from the robot itself, through the control computer, to the graphical user interface. We introduced the reader to the complex system housed within the control computer and its interactions with both the robot and the GUI through a stable communication layer, allowing the system to operate reliably even under unstable connections. We discussed the control, planning, and execution of the robot's movement.

Chapter 5 summarized our achieved results, which, in many respects, are comparable to the top scientific works on the topic of robotic chess players. We introduced the line-based Chessboard detector with a detection accuracy of up to 99.046%, a chess Piece detector based on YOLOv8, which, in our test dataset (98 frames), made only 4 errors, and a Move detector that made no mistakes during 12 test chess games.

Potential continuations of this work include improving the current Path planner, where a wide range of planning algorithms and optimization methods can be implemented. The work can be easily extended to another manipulator, possibly with improved human-robot interactions. Moreover, this study can be easily adapted as a universal chess dataset generator if we measure the detected chessboard using a calibrated camera. This would significantly contribute to the scientific community addressing this topic by providing datasets for machine learning.

Appendix A

Bibliography

[1] F.-H. Hsu, "Ibm's deep blue chess grandmaster chips," *IEEE Micro*, vol. 19, no. 2, pp. 70–81, 1999.

[2] A. T.-Y. Chen and K. I.-K. Wang, "Computer vision based chess playing capabilities for the baxter humanoid robot," in *2016 2nd International Conference on Control, Automation and Robotics (ICCAR)*, pp. 11–14, 2016.

[3] H. M. Luqman and M. Zaffar, "Chess brain and autonomous chess playing robotic system," in *2016 International Conference on Autonomous Robot Systems and Competitions (ICARSC)*, pp. 211–216, 2016.

[4] P. Kołosowski, A. Wolniakowski, and K. Miatliuk, "Collaborative robot system for playing chess," in *2020 International Conference Mechatronic Systems and Materials (MSM)*, pp. 1–6, 2020.

[5] D. A. Christie, T. M. Kusuma, and P. Musa, "Chess piece movement detection and tracking, a vision system framework for autonomous chess playing robot," in *2017 Second International Conference on Informatics and Computing (ICIC)*, pp. 1–6, 2017.

[6] J. Ding, "Chessvision: Chess board and piece recognition," 2016.

[7] N. Banerjee, "A simple autonomous chess playing robot for playing chess against any opponent in real time," 08 2012.

[8] F. Al-Saedi and A. H. Mohammed, "Design and implementation of chess-playing robotic system," *IJCSET*, vol. 5, pp. 90–98, 05 2015.

[9] C. Matuszek, B. Mayton, R. Aimi, M. Deisenroth, L. Bo, R. Chu, M. Kung, L. LeGrand, J. Smith, and D. Fox, "Gambit: An autonomous chess-playing robotic system," pp. 4291–4297, 05 2011.

[10] A. de Sá Delgado Neto and R. Mendes Campello, "Chess position identification using pieces classification based on synthetic images generation and deep neural network fine-tuning," in *2019 21st Symposium on Virtual and Augmented Reality (SVR)*, pp. 152–160, 2019.

[11] G. Wölflein and O. Arandjelović, "Determining chess game state from an image," *Journal of Imaging*, vol. 7, no. 6, 2021.

[12] P. Fuersattel, S. Deitsch, S. Placht, M. Balda, A. Maier, and C. Riess, "Ocpad — occluded checkerboard pattern detector," pp. 1–9, 03 2016.

[13] K. Tam, J. Lay, and D. Levy, "Automatic grid segmentation of populated chessboard taken at a lower angle view," in *2008 Digital Image Computing: Techniques and Applications*, pp. 294–299, 2008.

[14] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[15] G. Jocher and Others, "Yolov5 by ultralytics," May 2020. Version v7.0, DOI: 10.5281/zenodo.3908559, URL: https://github.com/ultralytics/yolov5, License: GPL-3.0.

[16] G. Jocher, A. Chaurasia, and J. Qiu, "Yolo by ultralytics," Jan. 2023. Version: v8.0.0, URL: https://github.com/ultralytics/ultralytics, License: AGPL-3.0.

[17] P. Karia, V. Jain, M. Shah, and S. Rane, "Digitization of chess board and prediction of next move," in *2022 IEEE 7th International conference for Convergence in Technology (I2CT)*, pp. 1–5, 2022.

[18] V. Wang and R. Green, "Chess move tracking using overhead rgb webcam," in *2013 28th International Conference on Image and Vision Computing New Zealand (IVCNZ 2013)*, pp. 299–304, 2013.

[19] T. Cour, R. Lauranson, and M. Vachette, "Autonomous chess-playing robot," *ECOLE POLYTECHNIQUE*, July 2002.

[20] C. Danner and M. Kafafy, "Visual chess recognition," 2015.

[21] R. Srivatsan, S. Badrinath, and G. Lakshmi Sutha, "Autonomous chess-playing robotic arm using raspberry pi," in *2020 International Conference on System, Computation, Automation and Networking (ICSCAN)*, pp. 1–6, 2020.

[22] D. Vegas Romero, "Implementation of a chess playing robot application," Master's thesis, Escola Tècnica Superior d'Enginyeria Industrial de Barcelona, June 2020.

[23] Epic Games, "Unreal engine, version 4.27," Aug. 2021. URL: https://www.unrealengine.com.

[24] W. Qiu, F. Zhong, Y. Zhang, S. Qiao, Z. Xiao, T. S. Kim, Y. Wang, and A. Yuille, "Unrealcv: Virtual worlds for computer vision," *ACM Multimedia Open Source Software Competition*, 2017.

[25] T.-Y. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, C. L. Zitnick, and P. Dollár, "Microsoft coco: Common objects in context," 2015.

[26] C. Lugaresi, J. Tang, H. Nash, C. McClanahan, E. Uboweja, M. Hays, F. Zhang, C.-L. Chang, M. Yong, J. Lee, W.-T. Chang, W. Hua, M. Georg, and M. Grundmann, "Mediapipe: A framework for perceiving and processing reality," in *Third Workshop on Computer Vision for AR/VR at IEEE Computer Vision and Pattern Recognition (CVPR) 2019*, 2019.

[27] G. Farnebäck, "Two-frame motion estimation based on polynomial expansion," vol. 2749, pp. 363–370, 06 2003.

[28] The Stockfish developers (see AUTHORS file), "Stockfish," No year provided. Repository: https://github.com/official-stockfish/Stockfish.

[29] G. M. Sadler, "Tcec season 23 superfinal: Leela chess zero vs stockfish," Apr. 2023. URL: https://tcec-chess.com/articles/Sufi_23_-_Sadler.pdf.

[30] S. Gottschalk, M. C. Lin, and D. Manocha, "Obbtree: A hierarchical structure for rapid interference detection," in *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, (New York, NY, USA), p. 171–180, Association for Computing Machinery, 1996.

[31] Dawson-Haggerty et al., "trimesh," Dec. 2019. Version 3.2.0, URL: https://trimsh.org/.

[32] J. Pan, S. Chitta, and D. Manocha, "Fcl: A general purpose library for collision and proximity queries," in *2012 IEEE International Conference on Robotics and Automation*, pp. 3859–3866, 2012.

[33] D. Pařil, "Online attachments for master's thesis," 2023. Available online: `https://owncloud.cesnet.cz/index.php/s/yPFGvfmEkceqOZB`.

Appendix B

Attached files

Attached to this thesis are all the source files necessary to run the presented system. Due to attachment size limitations, we were compelled to split the appendix, so its complete form can be found only at this link [33]. The tree structure of the file system is illustrated below:

```
root
├─ ControlComputer_Source_code
│  ├─ game_player.py
│  ├─ DatasetGenerator.py
│  ├─ AnnotationTransfer.py
│  └─ ...
├─ MCU_Source_code
│  ├─ main/src/Runner.java
│  └─ ...
├─ Synthetic_Dataset_Generator
│  ├─ DatasetYolo.py
│  └─ ...
├─ Datasets (online-only)
│  ├─ RealWorldDataset
│  └─ SyntheticDataset
├─ pretrained weights (online-only)
│  └─ ...
└─ UnrealEngine (online-only)
   ├─ ChessGame
   └─ CNS
```