The University of Hong Kong

2024 – 25

COMP4801

Final Year Project

Final Report

Autonomous Chinese Checkers Playing Robot Arm

Name	**UID**
LEUNG Ho Ning	3035801453

Abstract

This project presents an autonomous Chinese Checkers playing robot system that integrates computer vision, artificial intelligence (AI), and precision robotics. Chinese Checkers (波子棋/跳棋), deeply embedded in Chinese cultural traditions and family gatherings, holds significant cultural heritage value that deserves preservation and innovation. It holds a special place as a collective memory for many in the Chinese community, symbolising togetherness and tradition. Despite its cultural importance and strategic richness, there has been notably limited research on automating this game compared to Western counterparts like chess or checkers, representing a significant gap in the field of AI-robotic gaming systems.

Our system consists of three integrated components: a vision system that accurately detects the board state using a mobile phone camera with real-time detection system, an AI engine that determines optimal moves using strategic algorithms, and a robotic arm equipped with a vacuum gripper for precise marble manipulation. A custom Android application serves as the central hub, coordinating these components and providing user interactions.

The project tackles unique challenges including the hexagonal board layout and the precision required for physical marble manipulation. Our innovative movement correction algorithm significantly enhances the robotic arm's reliability.

This work demonstrates the feasibility of creating accessible autonomous systems for culturally significant games using readily available components. By combining traditional gameplay with modern technology, our project contributes to both technological advancement in robotics and the preservation of cultural heritage in an engaging, interactive format.

Acknowledgment

I must firstly extend my sincere gratitude to my supervisor, Dr. T.W. Chim, for his ongoing guidance, insightful feedback, and weekly availability throughout this project. His expertise and encouragement were invaluable to the project's success.

Special thanks to Mr. David Lee, whose technical expertise in robotics and mobile application development was crucial for overcoming hardware challenges and implementing the control system. His patient instruction significantly accelerated the progress.

I also appreciate the Faculty of Engineering and Tam Wing Fan Innovation Wing for providing essential facilities, including 3D printers and technical equipment, which were instrumental in bringing our theoretical concepts to practical reality.

Finally, I thank all who contributed their time, knowledge, and support, helping our group achieve our milestone of creating a functional autonomous Chinese Checkers playing robot.

Table of Contents

Lists of Figures

1. Introduction

1.1 Project Background

Board games have traditionally served as platforms for both entertainment and the development of strategic thinking. With advances in artificial intelligence and robotics, there has been growing interest in creating automated systems that can play these games, serving as both opponents and learning tools.

AI has already transformed the landscape of strategic board games, most notably in chess and Go. Systems like Stockfish (Tong et al., 2023) and AlphaGo (Silver et al., 2016) have achieved superhuman performance, demonstrating the potential of AI in complex decision-making environments. The integration of robotics into gaming has further expanded these possibilities, enabling physical gameplay and embodied interactions that enhance both accessibility and engagement.

Despite these developments, Chinese Checkers remains relatively underexplored in the domain of AI-robotic systems. The game holds particular cultural significance within Hong Kong and broader Chinese communities, often played during Lunar New Year and family gatherings. Its strategic complexity—featuring a hexagonal board, multi-directional movement, and chain jumps—makes it both intellectually engaging and culturally resonant. It has also been researched that Chinese Checkers could serve as a strategic thinking development tool in undergraduate education (F1000Research, 2024). Yet, few efforts have been made to automate its play in a meaningful, interactive way.

This project aims to bridge this gap by developing an autonomous system capable of playing Chinese Checkers. The system integrates computer vision to detect the game state, an AI engine to determine optimal moves, and a robotic arm to execute physical movements. Beyond advancing technical capabilities, the project aims to preserve and modernize a culturally significant game, showcasing its potential for educational use, human-robot interaction, and the integration of heritage with contemporary technology

Traditionally, Chinese Checkers could fit into 2 to 6 players. However, for our project at this stage, we would focus on 2-player games, namely human and AI robotics. To eliminate confusion, the rule of game we adopted is attached at the *Appendix*.

1.2 Motivation

The motivation for this project stems from both technological and social considerations. First, there exists a notable gap in research and development surrounding the automation of Chinese Checkers. Unlike well-studied Western games such as chess or Go, Chinese Checkers remains underrepresented in the fields of AI and robotics, despite its strategic richness and cultural relevance.

This gap became even more significant during the COVID-19 pandemic, which disrupted traditional in-person activities and heightened the demand for alternative, remote-friendly forms of engagement. Stand-alone robotic gaming system which is capable of providing entertainment and human-robot interaction could offer a valuable solution, especially for individuals separated by social distancing measures.

Furthermore, many existing robotic game-playing systems rely on expensive, industrial-grade equipment that is often inaccessible for educational or domestic use. In contrast, this project aims to develop a cost-effective, modular solution that leverages readily available components such as mobile phones, open-source software, and consumer-grade robotic hardware. By minimizing the need for specialized tools, the system demonstrates how meaningful automation and cultural preservation can be achieved without prohibitive costs, making it suitable for widespread adoption. Moreover, such modular approach allows us to scale up easily to fit with any resources we have.

1.3 Project Objectives

The primary objective of this project is to develop an autonomous Chinese Checkers playing robot that integrates computer vision, artificial intelligence, and robotics into a cohesive and interactive system. We aim to create a robotic arm system capable of accurate piece movement through inverse kinematics, complemented by a computer vision server that recognizes the board state in real-time. The system will incorporate an AI engine using minimax with alpha-

beta pruning to enable strategic gameplay, allowing the robot to function as an even more challenging opponent than human players.

A parallel objective is to enhance accessibility through intuitive user experiences. This includes the development of a mobile application and physical button that enables users to easily se up and enjoy the game. Emphasis is placed on maintaining the traditional tactical feel of the game while ensuring reliable robotic opponent.

Finally, the system is prioritized with scalability and adaptability. Its modular architecture allows for future enhancements, such as expanding to support additional players, improvement of computing server for quicker and smarter decision, or higher level of hardware for improving gaming experience. By emphasising low-cost, reproducible components, the project seeks to demonstrate that such intelligent robotic systems can be both educational and widely accessible.

1.4 Key Deliverables

Our project delivers the Chinese Checkers playing system through three integrated phases.

The first phase establishes a robust Board Recognition System, making use of mobile phone camera with OpenCV-based self-developed computer vision algorithms to detect the hexagonal board, identify marble positions in real-time, and transform this visual data into a structured game state representation that can be communicated to other system components. This creates the foundation for understanding the physical game environment. It has been deployed to cloud server for easy access.

The second phase implements precise Robotic Arm Control with multi-axis movement capabilities by inverse kinematics calculations. This system enables accurate marble manipulation through carefully calibrated movements. We have added a vacuum gripper for marble grabbing. The robotic control module communicates wirelessly with our mobile application, providing real-time position feedback and movement confirmation, which allows

our game play logic flies as well as one of our highlights, the movement corrections algorithm to greatly reduce the inaccuracy of the arm in real-time.

Our final phase delivers an intuitive Game Logic and integration through an Android application that orchestrates all system components. This AI game engine, which has deployed to the cloud, implements the Minimax algorithm with Alpha-Beta pruning for strategic gameplay. We have enhanced the user experience with a physical button for players to press after completing their moves and integrated audio cues that signal when the robotic arm is ready to execute its turn, creating a more engaging and intuitive gameplay experience. The game process we designed is smooth. Most of the time, after a quick set-up, users do not have to touch the phone or computer, but focus on the marbles and the “turn ends” button.

1.5 Literature Review

This section reviews existing research and technologies relevant to the development of an autonomous Chinese Checkers-playing robot. It highlights key approaches in AI, robotics, computer vision, and system integration that inform and contrast with our proposed solution. This section would only provide high-level overview, relevant areas regarding implementation details would be explained in next chapter, methodology.

1.5.1 Autonomous Game-Playing Robots

The integration of artificial intelligence (AI) and robotics has significantly advanced the development of autonomous game-playing systems, particularly in the realm of chess. Early milestones include IBM's Deep Blue, which defeated a world champion using a combination of brute-force search and evaluation functions (Campbell et al., 2002), and DeepMind's AlphaZero, which employed reinforcement learning to master chess, shogi, and Go without human data (Silver et al., 2018).

Beyond software, there has been substantial progress in creating physical chess-playing robots that combine AI with robotic manipulation. For instance, the OpenChessRobot developed by Delft University of Technology integrates computer vision, a chess engine, and a robotic arm to interact with human players through both verbal and non-verbal cues (Zhang et al., 2025). Similarly, the RoboChess project utilizes a robotic arm to play chess against humans, emphasizing real-time interaction without the need for computer interfaces (Technion, 2025). At the University of Hong Kong, a final year project titled "Playing Chess with Robotic Arm" developed an autonomous chess-playing machine capable of interacting with human players at home as well. (Innovation Academy, 2020).

In the context of Chinese Checkers, such comprehensive robotic systems are notably scarce. It could hardly to find similar stand-alone arm-playing Chinese Checker games. This project aims to bridge this gap by developing an autonomous Chinese Checkers-playing robot that leverages computer vision, AI, and robotic manipulation to provide an interactive gaming experience.

1.5.2 Computer Vision for Board Games – OpenCV

OpenCV (Open Source Computer Vision Library) is a widely adopted open-source computer vision library that offers a comprehensive suite of tools for image processing and object detection. Its extensive functionalities make it a popular choice for developing real-time computer vision applications across various domains, including robotics and gaming (Bradski & Kaehler, 2008).

The library is capable of contour detection, perspective transformation, and color space manipulation are particularly beneficial for recognizing the hexagonal patterns and colored marbles characteristic of Chinese Checkers. Although there is no pre-trained model or built-in functions targeting Chinese Checker, the library’s modular design and flexibility make it well-suited for developing a custom detection pipeline. In fact, the library is implemented in C++ underhood which perfectly meets our real-time performance requirement. Therefore, OpenCV

was selected as the foundation for our board state recognition system due to its balance of functionality and portability.

1.5.3 Robotic Manipulation and Embedded Control

The robotic control system in this project is built around the ESP32 microcontroller, a low-cost, dual-core platform that offers integrated Wi-Fi and Bluetooth capabilities, making it ideal for wireless communication in the embedded robotics field and perfectly fit for our use, wirelessly real-time control for the robot arm (Espressif Systems, 2023). ESP32's real-time performance, combined with its energy efficiency and rich I/O options, enable integration with motor drivers, sensors, and vacuum sucker.

To implement the software side, the PlatformIO ecosystem was adopted, providing a flexible development environment for embedded C applications. Combined with inverse kinematics algorithms, which will be explained in the following chapter, this setup enables the robotic arm to compute and perform precise X-Y-Z positioning to pick up and place game pieces. This configuration ensures repeatable marble moving within the confined geometry of the Chinese Checkers board.

1.5.4 AI Algorithms for Board Games

Artificial intelligence plays a central role in the decision-making aspect of this project. Among various approaches, the minimax algorithm is one of the most widely used methods in board game AI, where it recursively explores future move possibilities and selects the action that minimizes potential losses while maximizing gains (Michalewicz & Fogel, 2004). To improve efficiency, alpha-beta pruning is employed, reducing the number of nodes evaluated by the algorithm without sacrificing accuracy (Russell & Norvig, 2021).

We chose minimax as our primary search strategy due to the deterministic and turn-based nature of Chinese Checkers. As a two-player, perfect-information game with no element of randomness, Chinese Checkers aligns closely with the conditions under which minimax is known to perform well (Chakole & Dey, 2021). The presence of multi-jump moves and long-term spatial planning further benefits from the lookahead search properties of minimax, allowing the algorithm to evaluate deeper positional advantages. Minimax, when paired with a domain-specific evaluation function, offers a balance of interpretability and strategic competitive play (Jiang, 2023).

1.5.5 Smartphone as a Coordination Hub

Smartphones are ubiquitous and accessible devices nowadays., making them ideal candidates for serving as coordination centers in various applications. As of 2025, approximately 4.69 billion people worldwide own a smartphone, reflecting their widespread adoption and integration into daily life (Backlinko, 2025). The affordability of smartphones has also improved significantly, with entry-level devices becoming more accessible to a broader population segment (GSMA, 2024). This extensive reach ensures that utilizing smartphones as central hubs could increase user familiarity and affordable

1.6 Contributions

Task	**Contributors(s)***
Detection algorithms	Hollis
Marble coordination	Alex
Movement correction algorithm	Hollis
3D modelling	Alex, Hollis
Artificial Game Engine	Hollis
Mobile Application development	Alex, Hollis
System Integration	Hollis
Documentations	Alex, Hollis

*Alphabetical order

1.7 Outline

The following chapters detail the design, implementation, and evaluation of our autonomous Chinese Checkers-playing robot.

Chapter 2: Methodology

This chapter presents the system architecture and development approach. It elaborates on the technical implementation of each subsystem, including the computer vision detection pipeline, robotic arm control, AI game engine, cloud server set up, and mobile coordination interface etc.

Chapter 3: Results and Evaluation

This chapter provides an overview of the system's performance and functionality. It presents observations based on real gameplay scenarios, highlighting the strengths and limitations of each component.

Chapter 4: Future Work

Following chapter 3, this chapter identifies areas for improvement and potential extensions of the project. It proposes enhancements in hardware precision, AI strategy, scalability, and user experience.

Chapter 5: Conclusion

The final chapter summarizes the project's achievements and reflects on its broader implications in the fields of robotics, AI, and cultural technology preservation.

2. Methodology

This chapter presents the technical implementation of our autonomous Chinese Checkers playing robot. We describe the system's modular components: a vision-based detection server for board recognition, a self-developed AI engine for strategic gaming, a precision robotic arm with a customized accuracy improvement algorithm for marbles manipulation, and a mobile application that orchestrates these elements. Each component addresses specific challenges in creating a reliable game-playing system, with particular focus on our innovative approaches to marble manipulation and AI game engine.

2.1 Overall System Architecture

Our autonomous Chinese Checkers playing system developed under a modular and distributed architecture that integrates computer vision, artificial intelligence, and robotic control through a central mobile application hub. This design approach allows independent development and testing of components while ensuring seamless interaction during gameplay.

The system consists of four primary components: a detection server for computer vision, an AI server for strategic decision-making, a robotic arm for physical manipulation, and an Android mobile application that coordinates these elements. Figure 1 illustrates the brief high-level architecture and interaction flow between these components, as well as what major tech stacks are used.

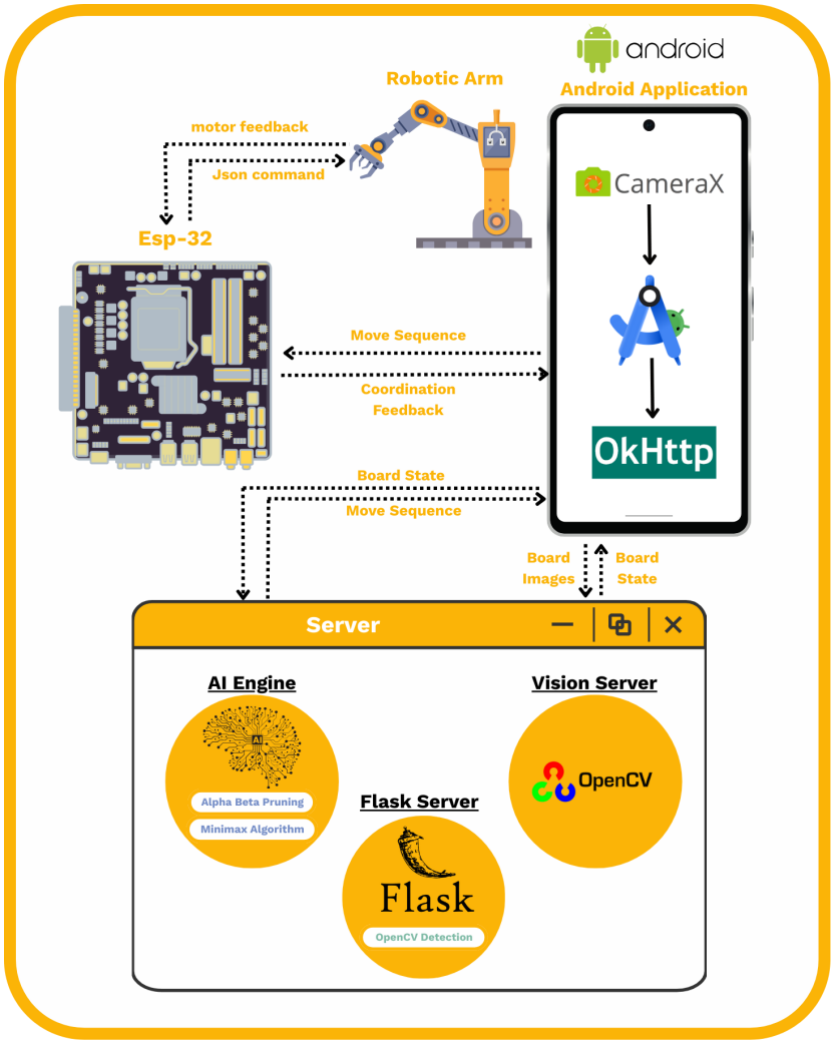


Figure 1 High Level Overview of System

2.1.1 Component Interaction

The mobile application serves as the central coordination hub, initiating all processes and managing the information flow between components. Generally inside a game cycle, the application first captures the board state using the phone's camera and transmits this image to the detection server via HTTP POST requests. The detection server processes this input using OpenCV algorithms to identify the board layout and marble positions, returning a structured representation of the game state.

Once the current board state is established, the mobile application forwards this information to the AI server, which employs a minimax algorithm with alpha-beta pruning to determine the optimal move. This server returns a sequence of coordinates representing the chosen move, which the mobile application then communicates with the robotic arm step by step, translating into a series of precise movements.

The robotic arm, controlled by an ESP32 microcontroller, executes these movements through a combination of positional calculations and vacuum gripper operations. After completing the move, the system would wait for the users' move and verify the new board state through another image capture and detection cycle, ensuring move accuracy before proceeding to the next turn.

2.1.2 Data Flow

Data flows through the system in a cyclical pattern:

1. Visual data (images) → Detection server → Structured board state representation
2. Board state → AI server → Move sequence coordinates
3. Coordinates → Mobile application → Movement commands
4. Commands → Robotic arm → Physical marble movement
5. Wait for another users' move

This cycle repeats with each turn, with the mobile application maintaining the game state and alternating between human and AI moves. The human player indicates completion of their turn by pressing a physical button, triggering the AI and robotic response sequence. Figure 2 provides a detailed illustration of this data flow, mapping the specific interactions between physical components, the mobile application hub, and cloud-hosted servers.

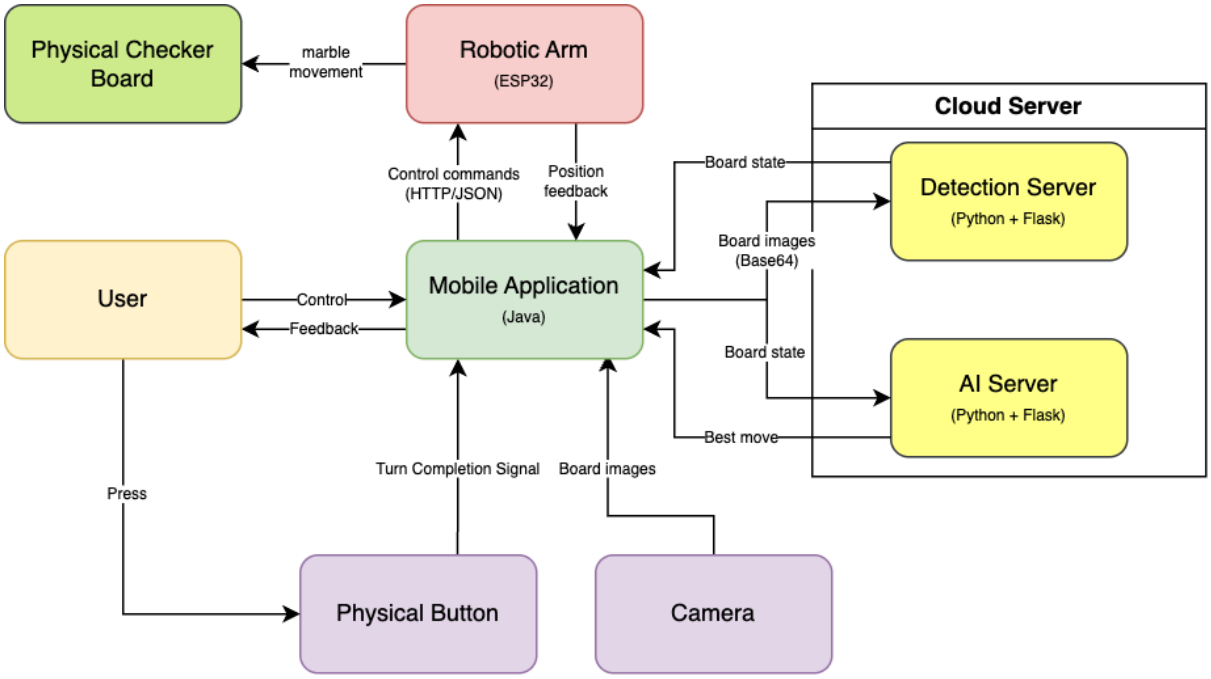


Figure 2 Dataflow Illustration of System

2.2 Detection Server

The detection server forms the foundation of our autonomous Chinese Checkers playing system, providing accurate recognition of the game state through computer vision techniques. Implemented in Python using OpenCV and Flask, this component processes images from the mobile application to identify the hexagonal board layout, locate each cell position, and determine which cells contain marbles and of what color. It involves multiple steps of image processing and computer vision identifying algorithms.

2.2.1 Challenges in Detection

During the development process, we discovered several challenges. The hexagonal grid layout with its star-shaped pattern creates irregular contours that are more complex to identify than rectangular grids. Additionally, the marbles used in Chinese Checkers introduce several detection complications.

Variable lighting conditions pose a significant obstacle, as reflections on the spherical marble surfaces create inconsistent highlights and shadows that interfere with color recognition. Additionally, the narrow spacing between marbles in standard game configurations makes individual piece detection challenging, as boundaries between adjacent marbles can be difficult to distinguish. Given the limited resources of our two-person project team, developing and training a machine learning model capable of adapting to all possible checkerboard configurations and environmental conditions was way beyond our scope. Instead, we opted for a more controlled approach using traditional computer vision techniques that could be precisely calibrated for our specific setup.

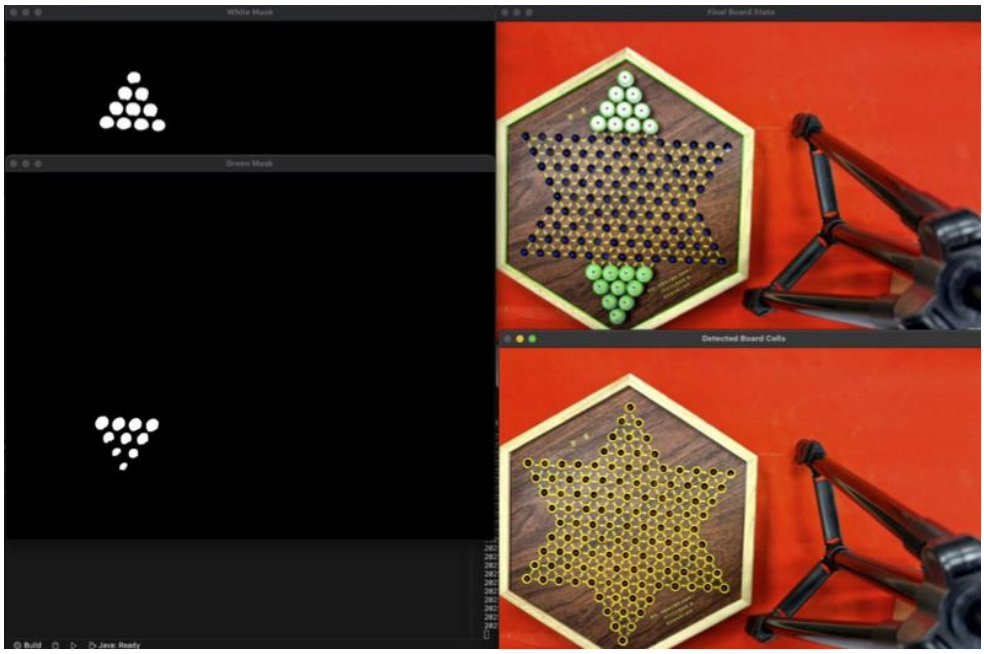


Figure 3 Graph showing the difficulty in detection, narrow spacing

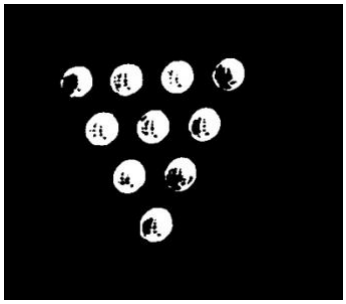


Figure 4 A colour mask showing the difficulty in detection sunlight reflection

2.2.2 3D Modelling - Standardisation

To overcome detection challenges and standardize our development process, we created a custom Chinese Checkers board using Rhino 3D modeling software and fabricated it with a 3D printer. This custom board features slightly increased distances between marble positions, providing better separation for the vision system while maintaining playability.

2.2.3 Computer Vision – Detection Pipeline Overview

To detect the Chinese Checkers board and the marbles positioned on it, we implement a structured image processing and analysis pipeline. The full detection sequence consists of the following key steps:

1. **Image Acquisition and Preprocessing**
    - Adaptive resizing
    - Brightness and contrast adjustment
    - CLAHE enhancement
    - Gaussian blurring
2. **Edge and Shape Detection – Checker Board**
    - Conversion to HSV and grayscale
    - Board contour detection (using Canny edge + contour approximation)
    - Morphological closing
3. **Cell Detection**
    - Hough Circle Transform for candidate cells
    - Filtering by board contour
4. **Marble Detection**
    - HSV-based color segmentation
    - Morphological cleaning
    - Circularity check
    - Filtering by board contour
5. **Mapping and State Encoding**
    - Row-based grouping of cells
    - Cell assignment to the board layout

- Nearest-cell assignment of marbles
- Generation of final board state matrix

2.2.4 Image Processing

To ensure accurate and consistent detection of both the board layout and marble positions, raw images captured by the Android application are subjected to a series of image preprocessing operations. These operations improve visual clarity, normalize contrast and brightness, and suppress noise, laying the foundation for detection in later stages. This section outlines the key steps implemented in our image processing pipeline.

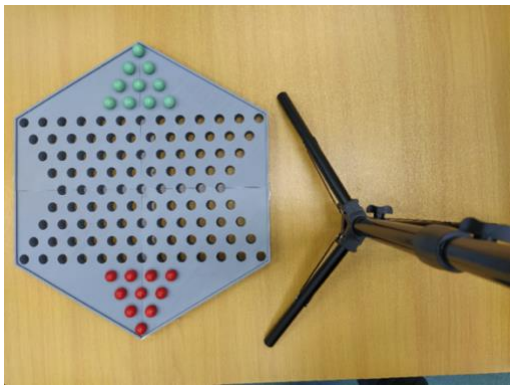


Figure 5 Preprocessed photo of our setup

2.2.4.1 Adaptive Resizing

To standardize input images and optimize computational efficiency, we implement an adaptive resizing strategy.

Procedure:

- **Maximum Dimension Constraint:** A maximum dimension (`max_dim`) of 1600 pixels is established. This constraint ensures that images are neither too large to process efficiently nor too small to lose critical details.

- **Scaling Factor Calculation:** The scaling factor is computed by comparing the image's original dimensions to `max_dim`. This factor maintains the aspect ratio, preventing distortion.
- **Resizing Operation:** The image is resized using OpenCV's `cv2.resize()` function with the `cv2.INTER_AREA` interpolation method. This method is particularly effective for downsampling, as it calculates the average of pixel areas, preserving image quality and reducing aliasing artifacts (OpenCV, n.d.).

This resizing process ensures uniformity across all input images.

2.2.4.2 Brightness and Contrast Enhancement

After resizing, the next step in the image preprocessing pipeline is enhancing the image's brightness and contrast to ensure consistent feature visibility under varied lighting conditions. This process facilitates downstream tasks such as marble detection and contour extraction by equalizing image intensity and improving local contrast.

Grayscale Conversion

The input image is first converted from the BGR (Blue-Green-Red) color space to grayscale using OpenCV's `cv2.cvtColor()` function. This reduces the image to a single channel representing luminance, simplifying intensity-based analysis and reducing computational complexity.

Histogram Calculation and Clipping

The grayscale histogram is computed using `cv2.calcHist()`, which returns the frequency distribution of pixel intensity values from 0 (black) to 255 (white). From this, we derive the **cumulative histogram**, representing the total number of pixels at or below each intensity level. This information is for identifying appropriate clipping thresholds to remove extreme dark or bright pixels that may skew contrast.

A clipping threshold (currently set at 1%) is applied to discard intensity extremes. This process effectively removes illumination artifacts and sensor noise, narrowing the intensity range to focus on useful pixel data (Pizer et al., 1987).

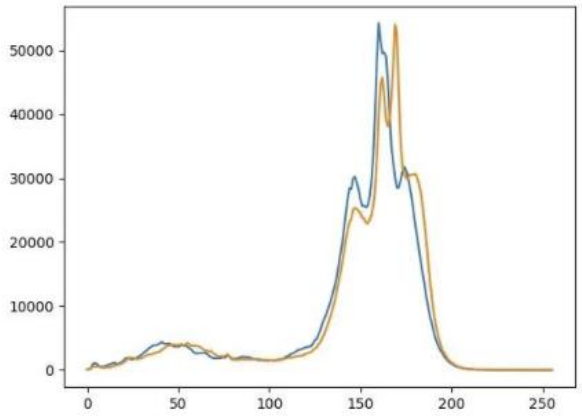


Figure 6 Histogram and cumulative histogram of grayscale

Intensity Scaling (Alpha and Beta Calculation)

The **minimum and maximum gray levels** post-clipping are identified, defining the effective intensity span. Two parameters are computed:

Alpha (α) – the contrast scaling factor, calculated as

$$\alpha = \frac{255.0}{\text{max\_gray} - \text{min\_gray}}$$

This scales the clipped intensity range to span the full 0–255 scale.

Beta (β) – the brightness offset, calculated as

$$\beta = -\text{min\_gray} \times \alpha$$

This re-centers the scaled range to ensure proper brightness alignment.

The image is then transformed using the linear equation:

$$\text{adjusted\_image} = \alpha \cdot \text{image} + \beta$$

The resulting image has improved contrast and dynamic range, making key features—such as the board's holes and marble shadows—more discernible across different lighting conditions. An example processed photo is listed as Figure 7

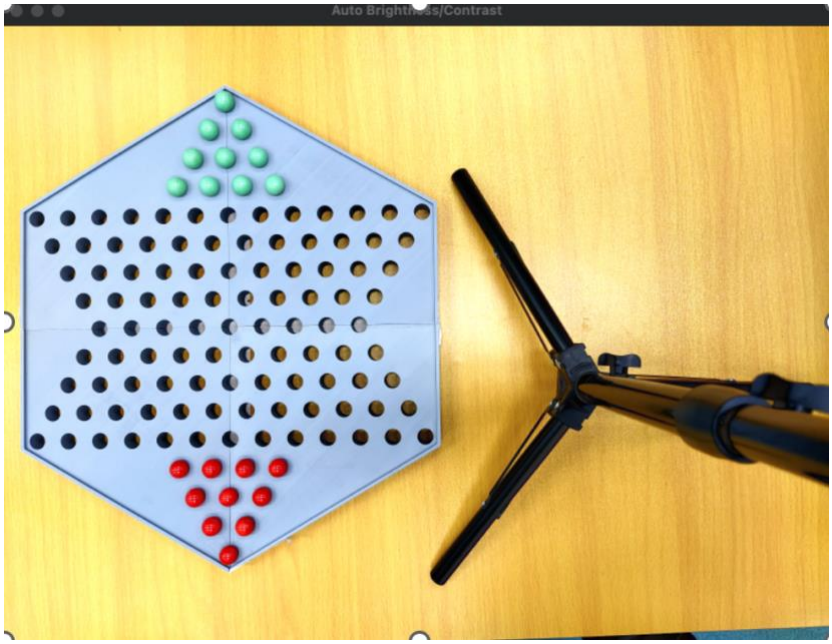


Figure 7 After adjusting brightness and contrast

2.2.4.3 Contrast Limited Adaptive Histogram Equalization (CLAHE)

To further enhance local contrast and mitigate the limitations of global histogram equalization, our system employs Contrast Limited Adaptive Histogram Equalization (CLAHE). This technique is particularly effective in improving the visibility of features in images with varying illumination conditions, such as those encountered in our Chinese Checkers setup.

Theoretical Background

Traditional histogram equalization adjusts the global contrast of an image by redistributing pixel intensity values. However, this approach can lead to over-enhancement of noise and loss of detail in areas where the histogram is not confined to a particular region. CLAHE addresses these issues by dividing the image into smaller regions called "tiles" and applying histogram equalization to each tile individually. This method enhances the local contrast of the image while limiting noise amplification (Pizer et al., 1987).

In CLAHE, the following steps are performed:

1. **Image Tiling**: The image is divided into non-overlapping tiles, typically of size 8×8 pixels.
2. **Histogram Equalization**: Each tile undergoes histogram equalization independently.

3. **Contrast Limiting**: To prevent over-amplification of noise, a clip limit is applied to the histogram of each tile. This limits the contrast enhancement by clipping the histogram at a predefined value and redistributing the clipped pixels uniformly across the histogram.
4. **Interpolation**: To eliminate artificially induced boundaries between tiles, bilinear interpolation is used to combine the neighboring tiles smoothly (OpenCV, n.d.).
5.

Implementation Details

In our implementation, we utilize OpenCV's `cv2.createCLAHE()` function with the following parameters:

- **Clip Limit**: Set to 2.0. This parameter controls the threshold for contrast limiting. A higher clip limit increases the contrast enhancement, while a lower value reduces it.
- **Tile Grid Size**: Set to (8, 8). This defines the number of tiles in the row and column.

The CLAHE algorithm is applied to the grayscale version of the input image, resulting in an image with enhanced local contrast and reduced noise amplification.

2.2.4.4 Gaussian Blur

To further enhance image quality and facilitate accurate detection of board features, our system applies Gaussian blur, a widely used technique in image processing for reducing high-frequency noise and detail. This step could greatly smooth the image, thereby improving the performance of subsequent algorithms such as edge detection and circle recognition.

Theoretical Background

Gaussian blur operates by convolving the image with a Gaussian function, effectively acting as a low-pass filter that attenuates high-frequency components (Shapiro & Stockman, 2001). The two-dimensional Gaussian function is defined as:

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

Where:

- $(x\ ,\ y)$ are pixel coordinates relative to kernel center
- σ is the standard deviation of the Gaussian distribution, controlling degree of blurring

This convolution results in each pixel's new value being a weighted average of its neighbors, with the weights determined by the Gaussian function. The center pixel has the highest weight, and the weights decrease with distance from the center, leading to a smooth blurring effect that preserves edges better than other blurring techniques (Gonzalez & Woods, 2018).

Implementation Details

In our implementation, we utilize OpenCV's `cv2.GaussianBlur()` function with the following parameters:

- **Kernel Size**: A 5×5 kernel is used, which defines the size of the neighborhood considered for blurring. The kernel size is chosen to be odd and positive, as required by the function.
- **Sigma (σ)**: Set to 0, allowing OpenCV to automatically calculate the standard deviation based on the kernel size. This ensures an appropriate level of blurring without manual tuning.

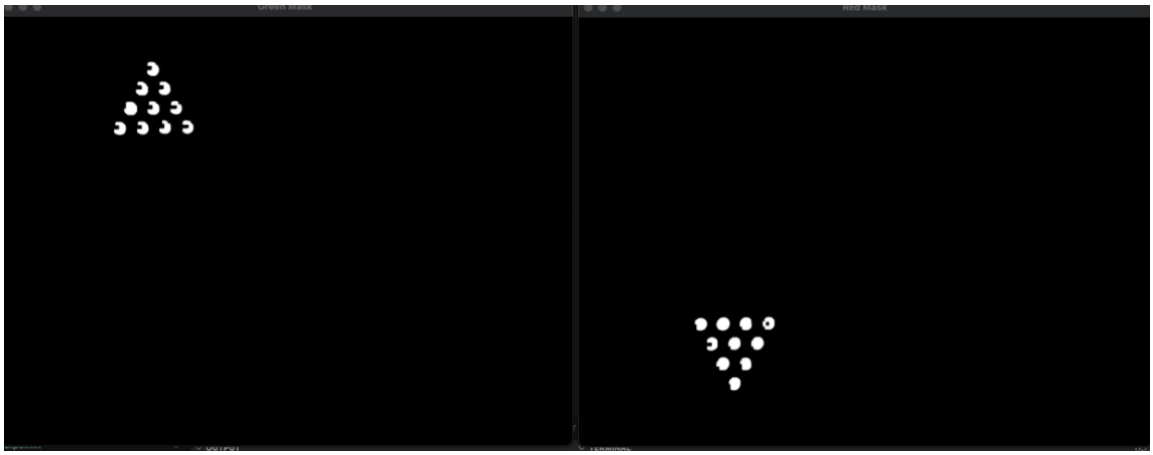


Figure 8 Before applying gaussian blur

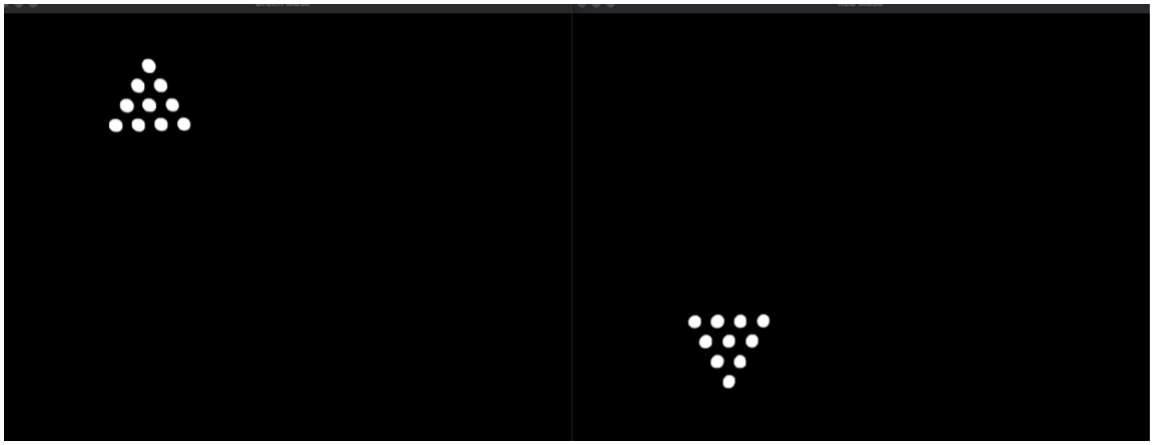


Figure 9 After applying gaussian blur

2.2.5 Edge and Shape Detection – Checker Board

This stage focuses on identifying the structural boundaries of the Chinese Checkers board by analyzing the image's edge features. We combine classic edge detection techniques with morphological operations and contour approximation to robustly isolate the board's hexagonal perimeter under varying lighting and background conditions. The following subsections describe each component in detail.

2.2.5.1 Color Space Conversion and Edge Detection

To accurately identify the structural boundaries of the Chinese Checkers board, our system performed a combination of color space conversion and edge detection techniques.

Color Space Conversion

The initial step involves converting the preprocessed image from the BGR (Blue, Green, Red) color space to grayscale, as explained above

Canny Edge Detection

Following the color space conversion, the system applies the Canny edge detection algorithm to the grayscale image. Developed by John F. Canny in 1986, this algorithm is renowned for its ability to detect a wide range of edges in images through a multi-stage process (Canny, 1986).

The Canny edge detection algorithm comprises the following steps (OpenCV, n.d.)

1. **Noise Reduction**: A Gaussian filter is applied to the grayscale image to smooth out noise and reduce the risk of false edge detection.
2. **Gradient Calculation**: The algorithm computes the intensity gradients of the image using Sobel operators to identify regions with high spatial derivatives. The algorithm calculates the intensity gradients in both horizontal (G_x) and vertical (G_y)directions. The gradient magnitude (G) and direction (θ) are computed as follows:

$$G = \sqrt{G_x^2 + G_y^2}$$

$$\theta = \arctan\left(\frac{G_y}{G_x}\right)$$

3. **Non-Maximum Suppression**: This step thins out the edges by suppressing all gradient values that are not at the maximum, ensuring that edges are retained precisely.
4. **Double Thresholding**: Two thresholds are applied to classify edges as strong edges (above the high threshold), weak (between the thresholds), or non-relevant (below the low threshold). Strong edges are immediately considered as part of the final edge image, while weak edges are included only if they are connected to strong edges.
5. **Edge Tracking by Hysteresis**: This final step eliminates spurious responses by suppressing all edges that are not connected to strong edges.

According to the OpenCV documentation, the ratio of the upper to lower threshold should generally be between 2:1 and 3:1 for optimal results (OpenCV, n.d.). In our implementation, we set the lower threshold to 50 and the upper threshold to 150 based on multiple experiences. By trial and error, 50,150 could capture the hexagonal board clearly without having extra unwanted noise.

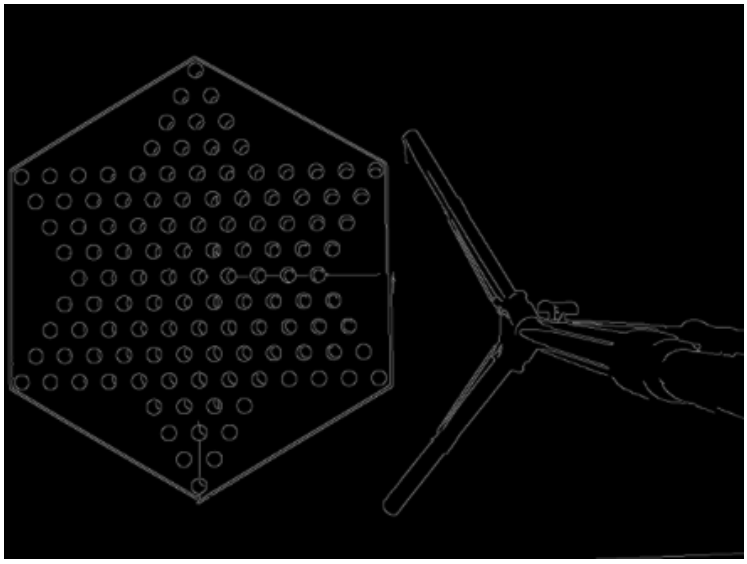


Figure 10 Photo using Lower Threshold:50 Upper Threshold:150

2.2.5.2 Morphological Closing

To enhance the continuity of edges and fill small gaps in the detected contours of the Chinese Checkers board, we decided to apply the morphological closing operation. This technique could refine the binary images resulting from edge detection.

Theoretical Background

Morphological closing is a fundamental operation in mathematical morphology, defined as a dilation followed by an erosion using the same structuring element. The primary purpose of closing is to close small holes and connect adjacent objects in a binary image without significantly altering their area (Serra, 1982).

Mathematically, the closing of a set A by a structuring element B is represented as:

$$A \cdot B = (A \oplus B) \ominus B$$

where:

- $\oplus$ denotes the dilation operation,
- $\ominus$ denotes the erosion operation,
- A is the input binary image,
- B is the structuring element.

Implementation Details

In our implementation, closing is performed on the binary edge map obtained from the `cv2.Canny()` edge detector. The function `cv2.morphologyEx()` is used with the `cv2.MORPH_CLOSE` flag, applying a 5×5 square kernel.

This kernel was chosen to be large enough to bridge gaps in the board's contours, especially at corner transitions, which was found to be optimal for filling the gaps between the edges of the checkerboard without introducing unwanted artefacts.

The result is a binary image (`closed_edges`) in which the outer perimeter of the board is cleanly connected, allowing for accurate polygon approximation and contour filtering in the subsequent steps.

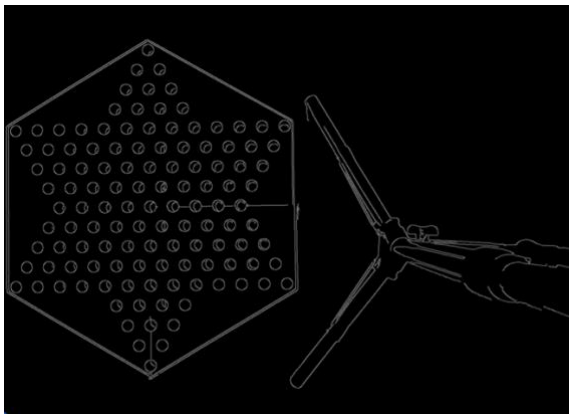


Figure 11 After applying Morphological Closing.

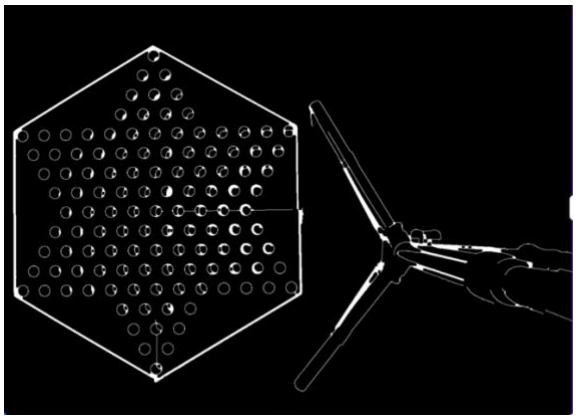


Figure 12 Before applying Morphological Closing.

2.2.5.3 Contour Detection and Polygon Approximation

Accurately detecting the hexagonal boundary of the Chinese Checkers board is a prerequisite for subsequent tasks such as cell filtering and marble mapping. This section outlines how contour detection and polygon approximation are used to extract the board's perimeter from preprocessed images.

Theoretical Background

Contour detection is a widely used method in computer vision to identify closed boundaries in a binary image. OpenCV implements the Suzuki–Abe algorithm, which traces the outermost edges of objects in an image and organizes them hierarchically (Suzuki & Abe, 1985). This

process is fundamental to separating foreground shapes from the background and isolating the region of interest.

To simplify complex contours and match them with known geometrical forms, polygon approximation is employed. OpenCV uses the Ramer–Douglas–Peucker algorithm for this purpose, which approximates a contour to a polygon with fewer vertices based on a tolerance parameter known as epsilon.

Implementation Details

To optimize the contour retrieval process, we employ the `RETR_EXTERNAL` mode, which retrieves only the extreme outer contours, thereby ignoring any nested contours that are not pertinent to the board's boundary. Additionally, we use the `CHAIN_APPROX_SIMPLE` method for contour approximation. This method compresses horizontal, vertical, and diagonal segments, leaving only their end points, which reduces memory usage and enhances processing speed (OpenCV, n.d.).

To approximate the shape of the detected contour to a polygon, we use OpenCV's `cv2.approxPolyDP()` function, which implements the Ramer–Douglas–Peucker algorithm. This algorithm simplifies a curve by reducing the number of points required to represent it, based on a specified precision. In our case, the approximation helps in identifying the board's boundary with a polygon consisting of 6 to 18 points, accommodating minor irregularities while preserving the overall shape.

Once contours are detected, we apply a filtering process to focus only on the most relevant shapes. Contours that occupy less than 5% of the image area are discarded to eliminate noise and insignificant features. Among the remaining contours, only the largest ones are selected for further processing, as they correspond to the board boundary. This approach ensures that the board’s edges could be correctly identified (OpenCV, n.d.)

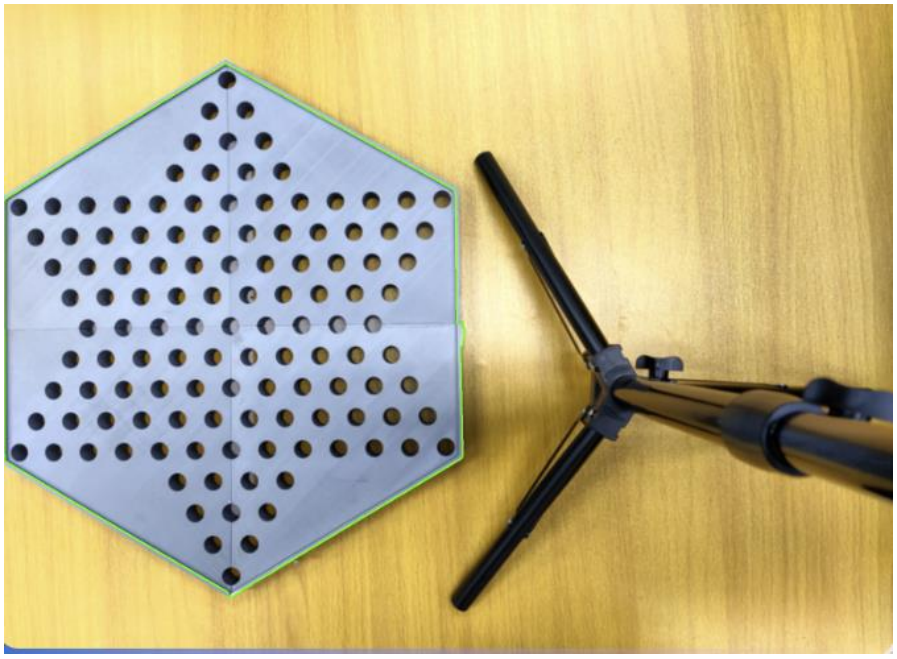


Figure 13 Board Contour

2.2.6 Cell Detection via Hough Circle Transform

To detect the precise positions of all playable cell locations (holes) on the Chinese Checkers board, we employ the Hough Circle Transform—a shape detection technique for circular geometry.

Theoretical Background

The Hough Circle Transform is a parametric voting-based method for identifying circular shapes in images, based on the general equation of a circle:

$$(x - a)^2 + (y - b)^2 = r^2$$

where (a, b) are the coordinates of the circle's center and r is its radius.

In the context of image analysis, each pixel in the edge-detected image space votes in a three-dimensional accumulator space over possible center locations and radii. Peaks in this accumulator indicate the most likely circle candidates. This method is particularly robust in detecting imperfect or partially obscured circles under noisy or non-uniform lighting conditions (Ballard, 1981; Duda & Hart, 1972).

Implementation Details

Building on prior preprocessing steps (Section 2.2.4), our cell detection pipeline consists of the following stages:

1. **Hough Circle Transform Configuration**:

   The grayscale image is passed to `cv2.HoughCircles()` with empirically tuned parameters:

   - `dp = 1.1`: The inverse ratio of the accumulator resolution to the image resolution.
   - `minDist = 30`: The minimum distance between detected circle centers, based on expected cell spacing.
   - `param1 = 500`: Upper threshold for Canny edge detection.
   - `param2 = 10`: Accumulator threshold for circle detection.
   - `minRadius = 15`, `maxRadius = 25`: Bounds for acceptable circle sizes based on board geometry.

   These settings were selected through iterative testing, tailored to our specific board design and camera setup.

2. **Board Contour Filtering**:

To ensure that only valid cells within the playable board area are retained, we filter out circles whose centers fall outside the previously identified board contour. This is achieved using OpenCV's `cv2.pointPolygonTest()`, which computes the signed Euclidean distance between a point and the board's polygonal boundary:

$$d = \min_{E \in C} \| P - E \|$$

where:

- $P(x, y)$ is the point being tested.
- $E(x_e, y_e)$ represents points along the edges of the contour CCC.
- d is the Euclidean distance between the point PPP and a point EEE on the contour.
- $d \geq 0$: Point lies inside or on the contour.
- $d < 0$:: Point lies outside the contour.

This filtering step significantly reduces false positives due to reflections or background textures.

2.2.7 Marble Detection

Building upon the previously identified cell positions, this section outlines the methodology employed to detect marbles, ensuring precise recognition of player moves and game progression.

Color Segmentation

Marbles are detected based on their distinct color characteristics in the HSV color space. For instance, green marbles are isolated using predefined HSV thresholds (e.g., `GREEN_LOWER` and `GREEN_UPPER`), while red marbles require two separate ranges due to the HSV hue wrap-around (e.g., `RED_LOWER1`, `RED_UPPER1`, `RED_LOWER2`, and `RED_UPPER2`). The masks generated for red are combined using a bitwise OR operation to encompass the full red spectrum.

Morphological Operations

To refine the binary masks and eliminate noise, morphological operations are applied:

- **Opening**: Erosion followed by dilation to remove small artifacts.
- **Closing**: Dilation followed by erosion to fill small holes within detected regions.

Ratio-of-Color (ROC) Based Occupancy Classification

To further increase detection robustness—particularly under variable lighting conditions—we implement a secondary approach based on the **ratio of target color pixels within a cell**. This method evaluates whether a circular region centered at each board cell contains a sufficient density of red or green pixels:

```
green_ratio = green_mask[circle].mean() / 255.0
red_ratio   = red_mask[circle].mean()   / 255.0

if   green_ratio > 0.15: occ[(cx,cy)] = 'green'
elif red_ratio   > 0.15: occ[(cx,cy)] = 'red'
else:                    occ[(cx,cy)] = None
```

This ROC-based approach is particularly effective in conditions where traditional contour-based methods may fail due to uneven illumination or partial occlusions. It is simple yet resilient, enabling accurate classification even in the presence of reflected light or slight marble misalignment. For example, Figure 14 shows that even under some light reflections, the marble is still able to be detected.

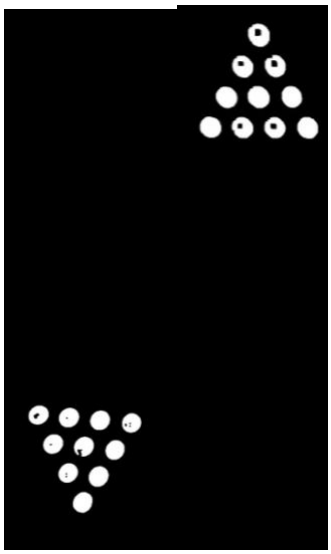


Figure 14 Successful Detection under Reflections

2.2.8 Mapping Logic / Board State

Understanding the current state of the board is essential for strategic decision-making in Chinese Checkers. The system translates the spatial positions of detected marbles into a structured representation of the board state.

Board Layout

The Chinese Checkers board layout is predefined as a 17-row structure, with each row having a specific number of cells. Currently it has a strcutre as follows:

```
board_layout = [
    [None],               # row 0 has space for 1 cell
    [None, None],         # row 1 has space for 2 cells
    [None, None, None],   # row 2 has space for 3 cells
    [None, None, None, None],    # row 3 has space for 4 cells
```

```
    [None, None, None, None, None, None, None, None, None, None, None, None, None],
    [None, None, None, None, None, None, None, None, None, None, None, None
    [None, None, None, None, None, None, None, None, None, None, None],
    [None, None, None, None, None, None, None, None, None, None],
    [None, None, None, None, None, None, None, None, None],
    [None, None, None, None, None, None, None, None, None],
    [None, None, None, None, None, None, None, None, None, None],
    [None, None, None, None, None, None, None, None, None, None, None],
    [None, None, None, None],   # row 13 has space for 4 cells
    [None, None, None],         # row 14 has space for 3 cells
    [None, None],               # row 15 has space for 2 cells
    [None],                     # row 16 has space for 1 cell
]
```

This layout is represented as a 2D array, where `None` placeholders indicate unoccupied spots. The layout serves as the reference for assigning cell coordinates and mapping marble positions.

In our artificial intelligence logic, we adapted a following graph structure in achieve of a path searching.

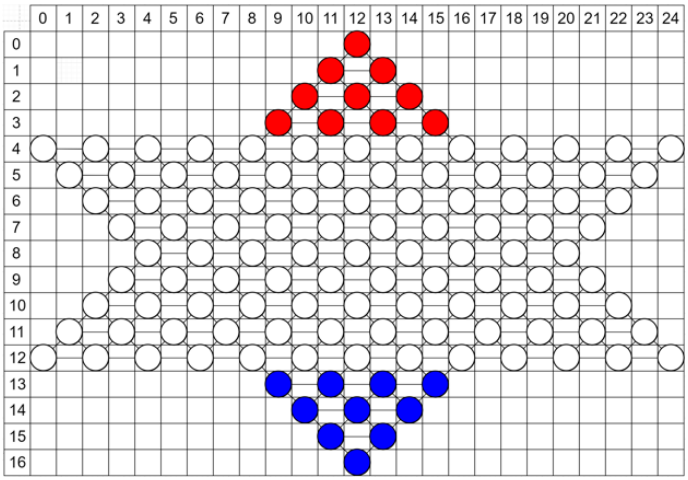


Figure 15 Graph for path searching

Grouping Cells into Rows

Detected cell centers are grouped into rows based on their y-coordinates. The process involves

1. **Sorting**: Cells are sorted by ascending y-coordinates.

2. **Row Thresholding**: Cells with y-coordinate differences within a predefined threshold (e.g., 15 pixels) are grouped into the same row.
3. **Within-Row Sorting**: Cells within each row are sorted by x-coordinates to maintain left-to-right order.

Assigning Cells to the Board Layout

The grouped cells are mapped onto the predefined board layout. This involves aligning each detected row of cells with the corresponding row in the board model. If the number of detected cells does not match the expected count, the system logs a warning, indicating potential detection issues.

Mapping Marbles to Cells

Each detected marble is assigned to the nearest available cell within a dynamic threshold, calculated as the sum of a base threshold and the marble's radius. The Euclidean distance between the marble's center and each cell's center is computed:

$$d = \sqrt{(x_{\text{marble}} - x_{\text{cell}})^2 + (y_{\text{marble}} - y_{\text{cell}})^2}$$

Marbles are assigned to the closest cell within this threshold, ensuring that each cell is occupied by at most one marble.

2.2.9 Communication Architecture

The detection server component of our system uses a client-server architecture to enable efficient communication between the mobile application and the computer vision processing pipeline. Built with Flask, a lightweight Python web framework, the server exposes RESTful API endpoints that handle image data transmission and board state retrieval.

The primary communication flow begins when the mobile application captures an image of the Chinese Checkers board using the device's camera. This image is encoded as a Base64 string

and transmitted to the detection server via HTTP POST requests. Two main endpoints facilitate this process:

1. `/upload_empty_board`: This endpoint receives an initial calibration image of the empty board, allowing the system to establish baseline cell positions before gameplay begins. The server processes this image using the detection pipeline described in previous sections and stores the cell positions for subsequent state recognition.
2. `/detect_current_state`: Once gameplay is underway, this endpoint accepts images of the board with marbles in their current positions. The server processes these images using the same detection pipeline, mapping detected marbles to the stored cell positions, and returns a structured representation of the board state.

The server responds to these requests with JSON-formatted data containing either confirmation of successful processing or a standardized representation of the current board state. For board state responses, each cell's status is encoded using a consistent notation:

- Empty cells: "."
- Green marbles: "G"
- Red marbles: "R"
- Non-playable positions: "X"

For our implementation, we have deployed the detection server on **the Render cloud platform** with **static end point**, though the architecture is designed to work on any suitable hosting environment. This client-server design offers several advantages:

- **Separation of concerns & Reduced mobile processing load**: Processing-intensive computer vision tasks run on the server, while the mobile application focuses on user interaction and coordination. The detection algorithms require significant computational resources that might strain mobile devices.
- **Flexibility for deployment**: The server can be hosted on any suitable infrastructure, from local development environments to cloud platforms.

Naturally, we have different debug endpoints for error handling, including health checks, debug test cases with debug images downloading, etc. However, as it is not supposed to be exposed to users, we would not explain it in detail in this report. Such implementation could be extracted from source code.

2.3 AI Server

The AI Game Engine forms the strategic core of our Chinese Checkers playing robot, enabling it to analyze the board state and determine optimal moves. This section details our implementation of the Minimax algorithm with Alpha-Beta pruning, the evaluation functions, heuristic strategies specific to Chinese Checkers, and outputs optimized move paths.

2.3.1 Board Representation and Normalization

In order to facilitate efficient evaluation and comparison of board states, the AI module utilizes a structured internal representation of the game board. The board is modeled as a two-dimensional array of Tile or coordinate objects, where each cell contains information about its occupancy status (e.g., red marble, green marble, or empty).

Each cell in this representation can be empty (.), contain a player piece (o for Player 1 or x for Player 2). For move generation, we implemented a graph-based model where each cell maintains references to its adjacent cells in six directions: left, right, upper-left, upper-right, lower-left, and lower-right. This structure enables efficient path finding for multi-jump sequences. For the details,p please refer Chapter 2.2.8 mentioned above.

2.3.2 Minimax Algorithm with Alpha-Beta Pruning.

The strategic decision-making core of our Chinese Checkers AI system is implemented using the Minimax algorithm with Alpha-Beta pruning. This classical approach is particularly well-suited for two-player, perfect-information games like Chinese Checkers, where each player has complete knowledge of the game state and there is no element of randomness.

Theoretical Foundation

The Minimax algorithm operates on a fundamental principle: the AI player (MAX) aims to maximize its score, while assuming the opponent (MIN) will play optimally to minimize that score. For a given game state S,

$$S = (P, B, T)$$

Where:

- P represents the current player (P $\in \{1, 2\}$)
- B denotes the board state matrix
- T indicates the turn number

The value function V(s) is recursively defined as:

- **Terminal state** (e.g., goal achieved or game over):

$$V(s) = \text{eval}(s)$$

- **MAX's turn**:

$$V(s) = \max_{s' \in \mathcal{M}(s)} V(s')$$

- **MIN's turn**:

$$V(s) = \min_{s' \in \mathcal{M}(s)} V(s')$$

Where:

- $\mathcal{M}(s)$ is the set of all successor states reachable from state s
- $\text{eval}(s)$ is the evaluation function going to be described in later, Section 2.3.3

Game Tree Structure and Analysis

The game tree in Chinese Checkers presents unique challenges compared to other board games due to its high branching factor and complex movement rules. Each node in the tree represents a game state, and each edge represents a possible move leading to a new state.

Chinese Checkers has a significantly higher branching factors. Firstly, It has Multi-jump sequences: A single turn can include multiple consecutive jumps, creating complex branching patterns. Secondly, **P**iece mobility: Each piece can potentially move in six directions.

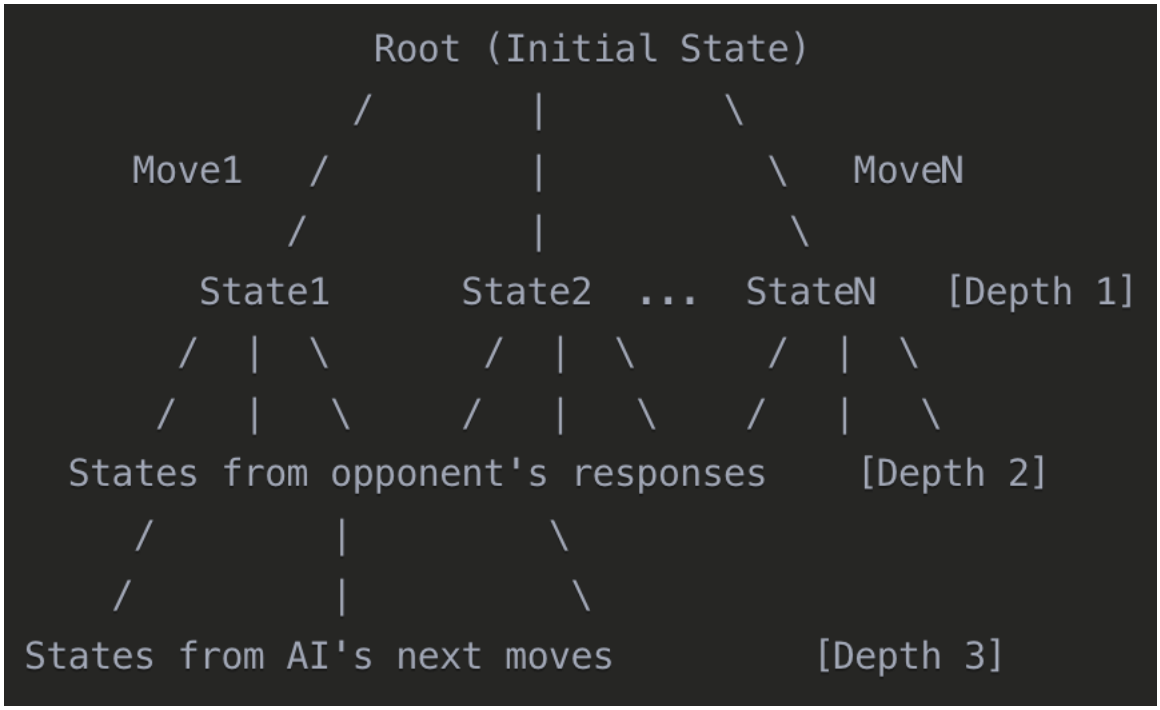


Figure 16 Conceptual Game Tree

This recursive framework naturally alternates roles between the two players:

- **When it's the AI's turn**, the algorithm **maximizes** the score, selecting moves that increase the AI's positional advantage.
- **When it's the opponent's turn**, the algorithm **minimizes** the score, assuming the opponent will play optimally to reduce the AI's chances of winning.

This high branching factor presents significant computational challenges. Our estimation showed that searching to depth 3 (representing one and a half complete turns) would be the best choice in our cases.

	Typical mid-game value	**Rationale**
Branching factor	25 – 35 legal paths per ply	Each piece has ≈ 2–3 adjacent steps plus a handful of multi-jump paths; ten pieces with mutually exclusive destinations give high-20s branching.
Search depth	3 or 4	Depth 3 ≈ one full move by each player + the reply half-move; depth 4 adds the opponent's full reply.

$\mathbf{N \approx b^d}$	b=30 ⇒ N(3)=27,000; N(4)=810,000	Flat-tree count before pruning or transposition savings.

Depth 3 typically lands in the **30 k–40 k** range, depth 4 in the **800 k–1.2 M** range, which will far exceed our computation capacity and our needs. Therefore, we chose to run in search **depth 3.**

Alpha-Beta Pruning Enhancement

To enhance the efficiency of the Minimax algorithm, we implemented Alpha-Beta pruning, which allows the algorithm to skip evaluating branches that cannot influence the final decision, thereby reducing the effective branching factor and enabling deeper search within the same computational constraints.

The pruning maintains two values:

- Alpha (α): The best value MAX has found so far
- Beta (β): The best value MIN has found so far

During search, if:

- At a MAX node, the current value ≥ β, search of this node's remaining children is skipped
- At a MIN node, the current value ≤ α, search of this node's remaining children is skipped

This optimization often reduces the number of evaluated positions by 40-60% without affecting the final decision, allowing our system to search deeper with the same computational resources.

AI Move Selection

For each of the AI's tiles:

- **All valid moves** are generated
- **Heuristic filters** may discard backward or non-progressive moves
- **Each path is simulated** using `apply_path()` and reversed with `undo_path()`
- **Evaluation is recursive**, and the best-scoring path is retained and returned

The AI outputs:

$$\text{BestMove} = \arg\max_{\text{path}\in\mathcal{P}} \text{Score}(s_{\text{result}})$$

Where:

- $\mathcal{P}$: set of all valid move paths from the current state,
- s_{result}: board state after applying a path,
- $\text{Score}(s)$: evaluation of the resulting state using the heuristic.

2.3.3 Evaluation Functions

The performance of a Minimax-based AI heavily depends on how effectively it evaluates board states, especially when the search reaches a depth limit or encounters non-terminal game positions. We developed two distinct evaluation functions which both are working greatly.

2.3.3.1 Evaluation Function 1: Distance-Based Progression

Give every tile a "distance score".

The farest tile right inside your goal triangle is worth **16 points**. One step farther away is worth **15**, the next **14**, … until the farthest possible tile, which is worth **0**.

In addition, if a tile belongs to the goal triangle of a colour, it gets an **extra bonus of +5** for pieces of that colour.

For any Player 1 piece on tile t

$$\text{score}_1(t) = (16 - d_1(t)) + 5*\mathbf{1}_{\text{goal}}(t)$$

Where

- $\mathbf{1}_{\text{goal}}(t)$ = 1 if t already lies **inside that side's goal triangle**, 0 otherwise
- $d_1(t)$ **shortest hop-count** (in tiles) from tile t to the **deepest corner** of Player 1's goal triangle

2.3.3.2 Evaluation Function 2: Board Control / Central – lane progress

The second evaluation function, $E_2(s)$ focuses on controlling key areas of the board, both vertically and horizontally.

Per-piece tile weight

For every marble $i$on a tile t_i define

$$s(t_i) = \underbrace{10\boldsymbol{v}(t_i)}_{\text{vertical progress}} - \underbrace{\delta(t_i)}_{\text{side-step penalty}} + \underbrace{50 * \mathbf{1}_{\text{goal}}(t_i)}_{\text{goal-triangle bonus}}$$

Where

- **Vertical progress** $\boldsymbol{v}(t_i)$= number of rows the marble has advanced **towards its own destination triangle**. For the upward-moving side this is $16 - r(t_i)$; for the downward-moving side it is $r(t_i)$.
- **Side-step penalty** $\delta(t_i) = | \ x(t_i) - c(r(t_i)) \ |$ = horizontal distance (in columns) from the board's centre line in that row.

Goal-triangle indicator

$$\mathbf{1}_{\text{goal}}(t_i) = \begin{cases} 1 & \text{if } t_i \text{ lies inside the player's target triangle} \\ 0 & \text{otherwise} \end{cases}$$

Evaluation Function 2 rewards a marble for three things: (1) **vertical progress**—every row it advances toward its home triangle is worth 10 points; (2) **central alignment**—each column it drifts away from the row's centre line subtracts 1 point, encouraging pieces to stay in the quick "central lane"; and (3) **goal completion**—the moment a marble enters its destination triangle it receives a flat +50 bonus, making arrival vastly more valuable than any single step elsewhere. The score of a position is the sum of these tile values for the side to move minus the same sum for the opponent, so the minimax search simply seeks to maximise this difference.

2.3.4 Heuristic Strategies for Chinese Checkers.

In addition to search and evaluation mechanisms, our AI incorporates domain-specific heuristics to guide move selection more intelligently.

Motivation for Heuristic Strategies

Chinese Checkers has a unique movement structure where:

A single move can involve a chain of jumps,

Excessive lateral or backward movement often stalls game progress.

Given that we have limited computing resources, applying a simple heuristic could not only improve performance by reducing searching paths but also could potentially leads to competitive moves.

Forward Progress Heuristic

The most basic heuristic we employ is a **directional heuristic**. It prioritizes moves that bring the marble closer to the target triangle. This is implemented through a simple function:

```
def heuristic(tile_origin, tile_destination) -> bool:
                # Enforce directional movement based on player's position
                if is_player1:
                    return             self.board.get_row_index(tile_destination)             >=
self.board.get_row_index(tile_origin)
                else:
                    return             self.board.get_row_index(tile_destination)             <=
self.board.get_row_index(tile_origin)
```

This filters out moves that go “backward” (i.e., increase the row index for player 1 or decrease it for player 2). It significantly improves search efficiency in the opening and mid-game by avoiding regressive moves.

Jump Preference Heuristic

A unique aspect of Chinese Checkers is the ability to make multi-jump moves, which can significantly advance pieces across the board. Our AI prioritizes evaluation of jump sequences over basic moves through a specialized ordering mechanism.

When generating possible moves, the algorithm first identifies all multi-jump sequences for a piece, then considers single-step adjacent moves. This prioritization ensures that potentially

more valuable multi-jump sequences are evaluated first during the search process, which has two benefits:

- It increases the efficiency of alpha-beta pruning by establishing strong bounds early in the search
- It ensures that strategically powerful jumping moves are not overlooked due to low search depth

2.3.5 Communication Architecture

The AI engine is deployed as a **modular HTTP-based server** using Flask, enabling our mobile application to request move suggestions and receive results in a structured format. This design ensures modularity and the ability to scale or substitute components without altering the AI core.

Server Structure and Entry Point

The Flask-based AI server exposes a single main route:

- `/get_ai_move` (POST)

  This endpoint accepts a JSON request containing:
  - The current board state (as a list of tiles and colors),
  - The active player,
  - The selected evaluation function and search depth (optional).

Upon receiving the request:

1. It extracts the game configuration from the received JSON data, including board state, player turn, search depth, and evaluation function preference
2. It transforms the text-based board representation into our internal data structures
3. It invokes the Minimax algorithm to determine the optimal move
4. It validates the generated move sequence to ensure it conforms to game rules
5. It converts the move into a standardized coordinate format before returning it to the client

Similarly, we have different debug endpoints for error handling, including health checks, test cases for verification of AI moves, etc. However, as it is not supposed to be exposed to users, we would not explain it in detail in this report. Such implementation could be extracted from source code.

2.4 Robot Arm

The robotic arm component of our autonomous Chinese Checkers playing system translates AI decisions into physical game moves. This section details our implementation of the arm's kinematic model, control system, vacuum gripper design. For our innovative movement correction algorithm which significantly enhances accuracy, it is implemented on the mobile application side and will be explained in the next chapter.

2.4.1 Inverse Kinematics Implementation

The RoArm-M2-S is a 4-degree-of-freedom (4-DOF) smart robotic arm designed for innovative applications. It features a lightweight structure, and customizable end-of-arm tooling (EoAT), making it suitable for various tasks requiring precision and flexibility. However, it does not provide a high-enough accuracy for marble picking such that we have developed our accuracy improving algorithm.

For a 4-DOF robotic arm like the RoArm-M2-S, inverse kinematics can be approached analytically by solving geometric relationships between the arm's links and joints. The goal is to determine the joint angles (θ_1, θ_2, θ_3, θ_4) that achieve a specified end-effector position (x, y, z) and orientation (φ).

Base Joint (θ_1):

The base joint angle is determined by the projection of the end-effector position onto the XY-plane:

$$\theta_1 = \arctan\ 2(y, x)$$

Wrist Center Position:

Calculate the position of the wrist center (the point where the wrist joint connects to the end-effector) by subtracting the end-effector's offset due to its orientation:

$$x_w = x - a_4 \cos\ \theta_1 \cos\ \phi$$

$$y_w = y - a_4 \sin\theta_1 \cos\phi$$

$$z_w = z - a_4 \sin\phi$$

Where:

- x_w, y_w, z_wCoordinates of the wrist center.
- a_4Length of the fourth link (distance from the wrist joint to the end-effector).
- ϕ: Desired orientation angle of the end-effector.

Shoulder and Elbow Joints (θ_2 and θ_3):

Define intermediate variables:

$$r = \sqrt{x_w^2 + y_w^2}$$

$$s = z_w - d_1$$

$$D = \frac{r^2 + s^2 - a_2^2 - a_3^2}{2a_2a_3}$$

Where:

- r: Horizontal distance from the base to the wrist center.
- s: Vertical distance from the base to the wrist center.
- d_1: Offset along the base joint's Z-axis (distance from the base to the shoulder joint).
- a_2: Length of the second link (distance between the shoulder and elbow joints).
- a_3: Length of the third link (distance between the elbow and wrist joints).
- Then, the elbow joint angle is:

$$\theta_3 = \arctan\ 2(\sqrt{1 - D^2}, D)$$

And the shoulder joint angle is:

$$\theta_2 = \arctan 2(s, r) - \arctan 2(a_3 \sin\theta_3\ , a_2 + a_3 \cos\theta_3)$$

Wrist Joint (θ_4):

The wrist joint angle is calculated to achieve the desired end-effector orientation:

$$\theta_4 = \phi - (\theta_2 + \theta_3)$$

The RoArm-M2-S utilizes an inverse kinematics function to compute the necessary joint angles for a given end-effector position. This allows the robotic arm to accurately reach target points by calculating the rotation angle of each joint.

2.4.3 Vacuum Gripper

The end effector of our robotic arm is a **vacuum-based gripper**, purpose-built for the delicate and reliable manipulation of Chinese Checkers marbles. This design provides a compliant and adaptable gripping mechanism that accommodates the smooth, curved surfaces of marbles without requiring mechanical fingers or force calibration.

Gripper Components and Design Rationale

The gripper comprises three primary components:

1. **Bellows-Type Suction Cup**

   A soft, bellows-style suction cup made from silicone rubber is used to form a conformal seal with the marble surface. The bellows design allows slight vertical compression, which increases contact stability and tolerance to minor height differences.

2. **Mini Vacuum Pump**

   A compact brushed DC vacuum pump generates negative pressure required for lifting. The pump is directly driven from a PWM-compatible pin on the ESP32 microcontroller, with sufficient suction strength to lift lightweight plastic or glass marbles without damage.

3. **Servo-Actuated Solenoid Valve**

   An inline micro servo-controlled valve is used to manage airflow between the suction cup and the pump. In its default (closed) position, it isolates the vacuum, ensuring the marble remains held even if the pump is briefly deactivated. When opened (typically to 180°), the vacuum is released, allowing the marble to be dropped cleanly. This design introduces enables precise timing for pick-and-release operations.

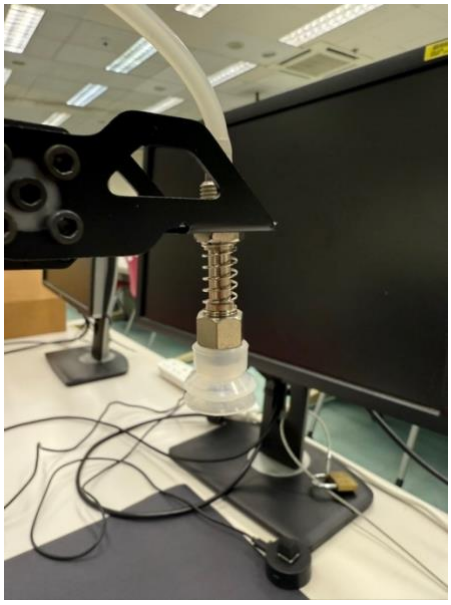


Figure 17 Bellows-Type Suction Cup

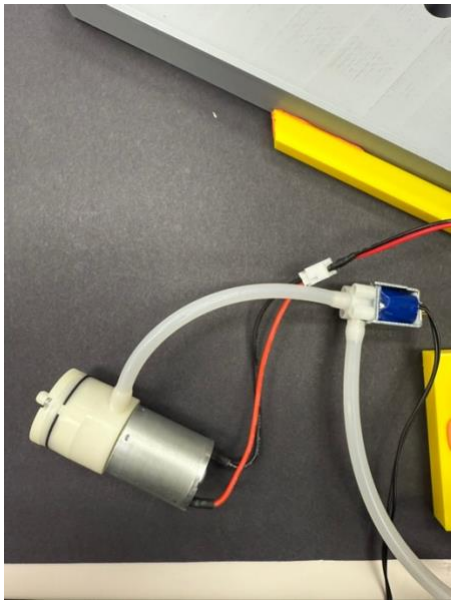


Figure 18 Vacuum Pump & Solenoid Valve

2.4.4 Control System Architecture

The control system for the robotic arm is built around the ESP32 microcontroller, which handles the physical movements based on commands from our mobile application. Rather than directly processing game logic, the ESP32 acts as a movement executor that responds to precise positional commands:

Coordinate Translation Layer

A key component of our architecture is the BoardCoordinatesAdapter class in our mobile application. This adapter translates the AI's internal board moves (e.g., tile-to-tile paths) into real-world coordinates. Each logical board position is mapped to a specific XYZ location on the physical board, along with an orientation angle and optional torque limits depending on board region.

Command Transmission

The ESP32 supports ESP-NOW, a low-latency, peer-to-peer wireless communication protocol. This feature allows multiple robotic arms to communicate and coordinate with each other without the need for a Wi-Fi network infrastructure. Commands are sent over Wi-Fi via HTTP requests, with the payload formatted as JSON objects. Each command contains:

- Command type (`"T"`)

- Position parameters (`"x"`, `"y"`, `"z"`, `"t"`)
- Speed and acceleration settings (`"spd"`, `"acc"`)

Example Commands

Purpose	JSON Command Example
Move to position	`{ "T": 104, "x": 235, "y": 0, "z": 234, "t": 3.14, "spd": 0.25 }`
Activate gripper (suck)	`{ "T": 116, "cmd": 1 }`
Release gripper (drop)	`{ "T": 116, "cmd": 0 }`
Reset to init position	`{ "T": 100 }`

2.6 Mobile App

Our mobile application acts as the **central control interface** and **execution coordinator** for the entire system. It orchestrates communication between the user, detection server, AI server, and robotic arm. Built in **Android (Java)**, the app provides real-time interaction and robust control over robotic execution.

2.6.1 System Architecture Overview

The mobile application adopts an architecture centered around a single core component: the `IntegratedAIGameFragment`. This fragment serves as the **main interface** for the entire gameplay experience, unifying camera control, board recognition, AI integration, and robotic actuation into one streamlined UI.

Responsibilities of the IntegratedAIGameFragment:

- **Camera Integration**: Captures images of the game board using Android's CameraX API.
- **State Processing**: Sends board images to the detection server and retrieves structured board states.

- **AI Move Calculation**: Communicates with the AI server to obtain recommended move sequences.
- **Robot Execution**: Translates AI decisions into physical movement using a multi-step robotic control sequence.
- **Game Logic**: Receive signal from physical button or response with direct click on screen, handle the whole game logic flow, from capturing board to detection, ask for ai, and execute move.

Supporting Controller Classes:

Class	Responsibility
`RobotController`	Handles HTTP communication with the ESP32 robotic arm, including movement and gripper control
`BoardDetectionClient`	Manages image encoding and REST API interaction with the board state detection server
`BoardCoordinatesAdapter`	Maps logical board coordinates (x, y) to real-world (X, Y, Z, Torque) positions for precise robotic movement

This modular structure enables robust fault isolation and simplifies development, testing, and maintenance..

2.6.2 Board State Recognition and AI Movement

The mobile application's board state recognition system serves as the visual perception component for our autonomous Chinese Checkers player. This subsystem captures images of the game board, processes them through the detection server, and transforms the results into a structured representation that can be used for game logic and AI decision-making.

2.6.2.1 Camera Integration and Image Capture

We leverage Android's CameraX for reliable image acquisition, which provides several advantages over older camera interfaces. For example it provides real time preview with

minimal latency and reliable performance across different Android devices • Reliable performance across different Android devices.

For board detection, we configure specific camera parameters to optimize capture quality:

- **Resolution**: 1600×1200 pixels (4:3 aspect ratio) to maintain consistent board proportions
- **Flash control**: Disabled to prevent reflections on the marbles
- **Focus mode**: Auto-focus with tap-to-focus capability for board clarity
- **Exposure compensation**: Slightly increased to enhance visibility of board features

This configuration ensures that captured images contain sufficient detail for accurate board and marble detection.

2.6.2.2 Image Processing and Server Communication

After capture, images are efficiently processed and transmitted to our detection server. The application converts images to JPEG format, encodes them as Base64 strings, and transmits them via HTTP POST requests to our cloud-based detection server.

The BoardDetectionClient class manages this client-server interaction, providing callbacks for both successful detection and error handling scenarios. This design ensures reliable communication with the detection server while maintaining a responsive user experience even when network conditions are suboptimal.

2.6.2.3 AI server movement

After we obtain the board state, the mobile application will query the best moves for the current board state through HTTP requests to the cloud AI server, as explained in the previous paragraphs.

2.6.3 Robotic Movement Execution and Correction

After obtaining the board state from the detection server and receiving the AI's move sequence, the mobile application orchestrates the precise physical movement of the robotic arm. This process involves translating logical game moves into coordinated physical actions, verifying position accuracy, and applying corrections when necessary.

2.6.3.1 Move Sequence Translation and Execution

The AI's suggested move (e.g path = [{"x": 0, "y": 16},{"x": 0, "y": 14},{"x": 4, "y": 12},{"x": 4, "y": 10}] is translated into a sequence of physical positions, including intermediate jump points. The application instructs the arm to:

1. Move above the origin marble location.
2. Descend to the marble's height and activate the vacuum gripper.
3. Retract to a safe height.
4. Move through intermediate jump points (if any), maintaining height.
5. Descend to the intermediate cells but **not** releasing marble
6. Descend to the target location and release the marble.
7. Return to home position (init).

Each step includes a short delay to ensure hardware safety, and each movement is accompanied by position feedback to ensure execution accuracy.

2.6.3.2 Servo Positioning Limitations and Challenges

A fundamental challenge in our implementation stems from inherent limitations in the RoArm-M2-S servo motors. Despite sending identical position commands, the actual position achieved by the arm varies significantly due to several factors:

- **Mechanical Backlash**: Play in the gears and joints causes positional discrepancies when approaching the same coordinates from different directions
- **Torque Inconsistency**: The servos experience varying loads depending on the arm's extended position, leading to position drift under different weight distributions
- **Servo Resolution Limits**: The 12-bit encoders provide theoretical 0.088° positioning resolution, but mechanical factors reduce actual precision. It is actually far from achieving this number.

These limitations result in positioning errors of approximately 3-5mm when attempting to reach precise coordinates, making it **impossible** to reliably pick up Chinese Checkers marbles (which require sub-2mm accuracy) with direct commands alone.

2.6.3.3 Feedback-Based Position Correction Algorithm

To overcome these hardware limitations, we developed an innovative feedback-based position correction system that adaptively compensates for positioning errors through deliberate overshooting.

Position Verification and Error Detection

After sending movement commands, the application requests the actual achieved position from the ESP32 controller and compares it with the intended target. The verification process measures the Euclidean distance between the target and actual positions, determining whether the error falls within an acceptable tolerance (empirically determined to be 1.5mm for reliable marble manipulation).

Through extensive testing, we observed that the robotic arm consistently underachieves its target position—reaching approximately 70-90% of the commanded distance. This systematic undershoot results from mechanical resistance, gear friction, and servo control limitations.

Adaptive Overshoot Strategy

When position errors exceed the acceptable threshold, our correction algorithm employs a progressive overshoot approach:

1. **First Attempt**: Standard movement to target coordinates
2. **Position Verification**: Measurement of actual position achieved
3. **Error Calculation**: Computation of X-Y ***positional error*** from target
4. **First Correction**: Command position at (target + error), ***essentially doubling the movement vector*** to compensate for systematic undershoot.
5. **Second Verification**: Re-measurement of position after correction
6. **Subsequent Corrections**: If still outside tolerance, apply overshoot factors (up to 3 total attempts), namely measure the difference of the actual coordinate now, comparing the original coordinate.

For example, if the arm attempts to reach position (240, 30) but actually arrives at (237, 28), the first correction will command position (243, 32)—intentionally overshooting to compensate for the observed mechanical resistance pattern.

This adaptive correction algorithm has transformed a consumer-grade robotic arm with moderate positioning accuracy into a system capable of reliable marble manipulation:

- **Without correction**: Position errors averaging 3.8mm, resulting in approximately 62% successful marble pickup rate
- **With adaptive correction, errors are** reduced to 1 mm average, achieving a 95% success rate compared to nearly 0% successful rate before the algorithm.

While the verification and correction steps add a 2-3 second overhead to each move operation, this trade-off is justified by the dramatic improvement in overall system reliability, essentially transforming an inconsistent hobby-grade arm into a dependable game-playing system. Of course, it could be updated in future work by upgrading the quality of arm.

2.6.4 User Interface and Experience Design

The mobile application interface is designed with simplicity and functionality in mind, focusing on the essential controls needed to manage the autonomous Chinese Checkers system. The UI consists of three main components: a camera preview area, control buttons for system operation, and status displays.

Main Interface Components

The primary interface displays a real-time camera preview of the game board, allowing users to properly position the phone for optimal board recognition. Below this preview, a sequence of action buttons guides users through the gameplay process.

Configuration options are also available, allowing users to set the robot's IP address for proper communication. During normal gameplay, users primarily interact with the system through the "AUTO PLAY TURN" button or a connected physical button to trigger the robot's turn after completing their own move.

However, as the normal run down, a user only has to click the set robot ip button and capture empty button. After the robot arm ip is set and the capture empty button is set, they can continue to click the physical button after each time they made a move. The others button is for a manually step by step if any errors occurred, as a precaution. Please refer to Figure 19 below.

Status Monitoring and Feedback

The application provides continuous feedback through dedicated status displays:

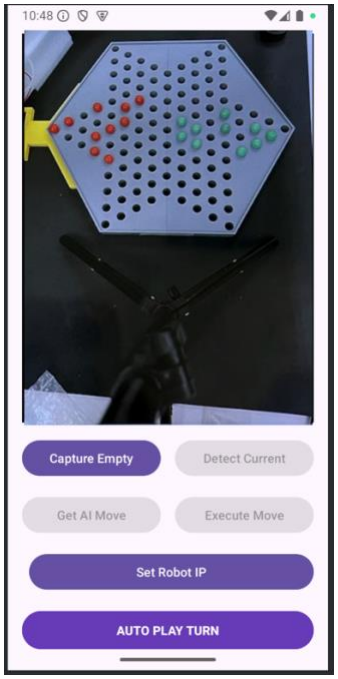
- **Board State Visualization**: Shows the detected game state in a text-based format, with "G" representing green marbles, "R" representing red marbles, and "." indicating empty positions
- **Movement Progress**: Displays detailed information about the movement correction process, including original targets, position errors, and adjusted coordinates
- **Diagnostic Information**: Provides system status indicators such as connection status, movement status, and detection readiness

These feedback mechanisms allow users to monitor the system's operation and verify that the board state has been correctly recognized before proceeding with moves. Please refer to Figure 20, 21.

Debug and Testing Interface

For development and troubleshooting purposes, the application includes additional controls:

- **"Show Debug Info":** Displays system configuration and state variables
- **"Lookup & Move To Position"**: Allows direct testing of specific board coordinates
- **Manual Gripper Control**s: Enables independent testing of the vacuum gripper through separate "Start Gripper" and "Stop Gripper" commands



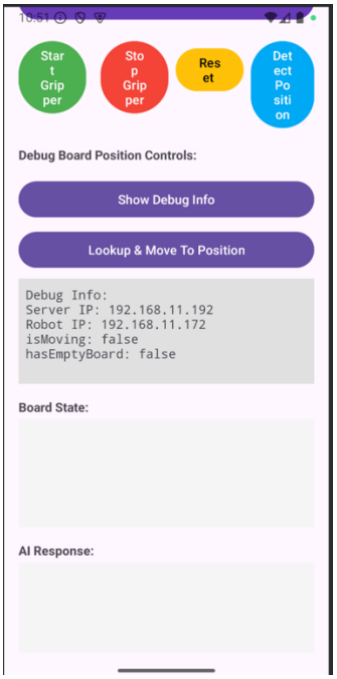


Figure 19 User Interface – top

Figure 20 User Interface – bottom

Figure 21 Example Runtime Screen

3. Results - Game Demonstration

As a product-based project, our primary focus was on the integration and functionality of the complete system rather than isolated component performance testing. Therefore, instead of conducting traditional benchmark testing or performance profiling, we evaluated the system through full gameplay demonstrations of Chinese Checkers.

Before gameplay can begin, several calibration steps must be completed to ensure that all system components work together seamlessly. The setup process involves hardware positioning, camera calibration, and establishing connections between the mobile application and both the detection server and the robotic arm. The brief game flow could refer to Figure 22

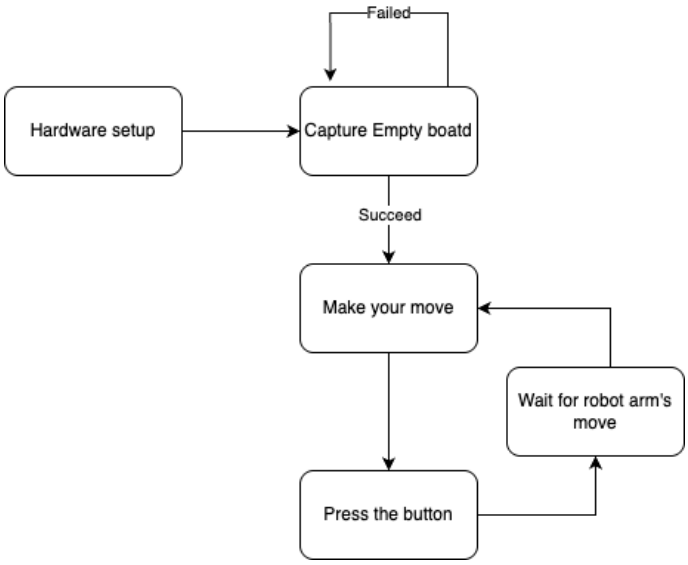


Figure 22 Game Flow

3.1 System Setup and Initialization

The autonomous Chinese Checkers system requires a straightforward setup process that connects all components before gameplay begins.

3.1.1 Physical Setup

The physical setup comprises the following steps (also visualized in Figure 23):

1. **Arm Positioning**: Mount the robotic arm at the edge of the table. Place a yellow spacer to guide consistent board placement.
2. **Board Placement**: Align the Chinese Checkers board flush against the spacer.
3. **Camera Alignment**: Mount the Android phone on a fixed stand, approximately 40cm above the board. Ensure the top edge of the board is horizontally aligned with the camera view.
4. **Power Connection**: Connect the robotic arm to power and turn it on.

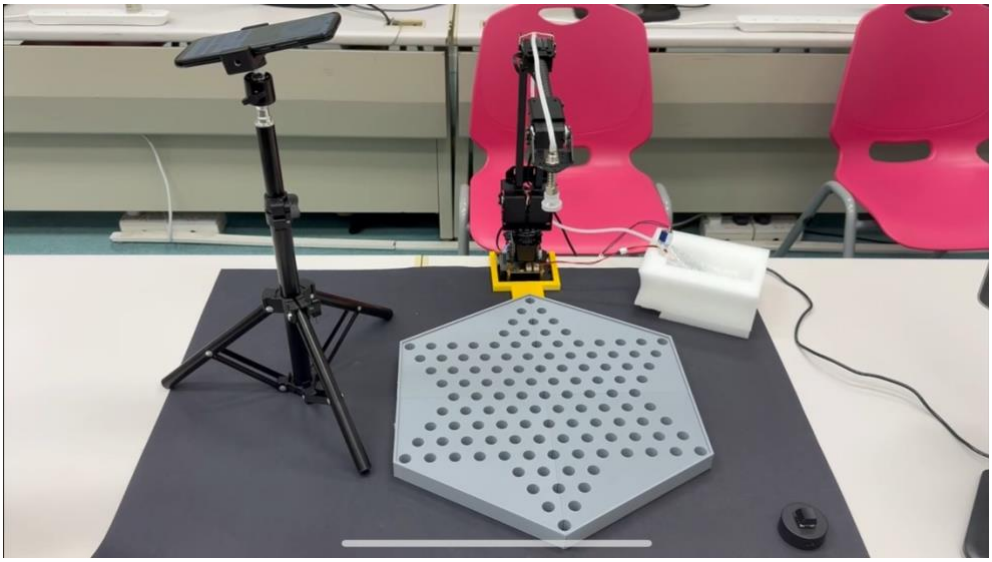


Figure 23 Set Up

3.1.2 Application Initialization

1. **IP Configuration**: The robot IP (e.g. `192.168.11.172`) is displayed on the robotic arm's screen and please manually enter in the app (Figure 24).
2. **Wi-Fi Setup**: Both the phone and arm is suggested be connected to the same local network to prevent any router blockage. Network settings can be adjusted via the web interface (`http://192.168.4.1`) when connected to the `RoArm-M2` access point.
3. **Empty Board Detection**: Tapping "Capture Empty" captures a reference image and sends it to the detection server, establishing a baseline for future marble detection.
4. **Confirmation**: The app displays a success message upon completion. This initialization process typically takes less than a minute and needs to be performed only once at the beginning of a play session.

Figure 24 Robot Arm IP

3.2 Gameplay Walkthrough

After setting up, the game proceeds through repeated turns of human input and robotic response. A typical cycle includes the following:

Step 1 – Capture Empty Board and Place Marbles

The player captures the empty board (Figures 25 and 26) and places marbles in their starting zones while waiting for the confirmation message.

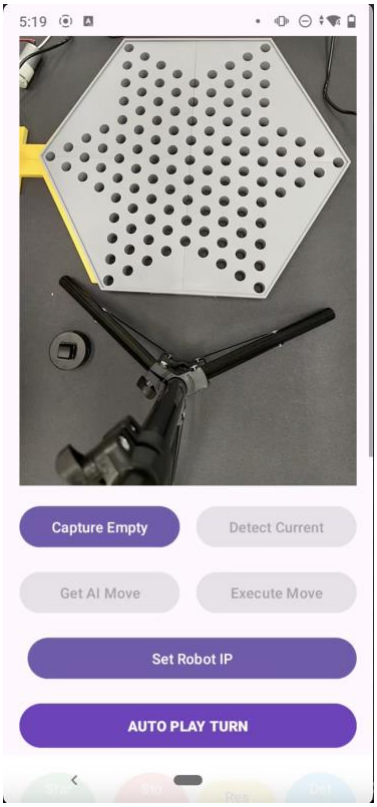


Figure 25 First Step - Detect Empty

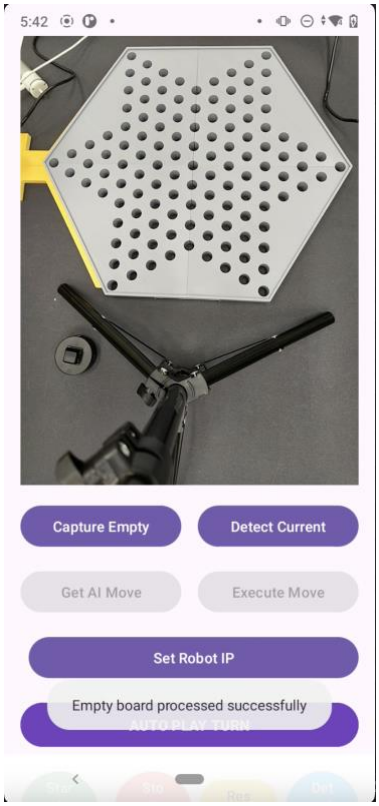


Figure 26 Detection Successful

Step 2 – Human Move and Board Detection

The game begins with the traditional Chinese Checkers setup: 10 green marbles in the top triangle and 10 red marbles in the bottom triangle. For simplicity, the robot arms always play red and the player always play green.

Our game design did not strictly require the game to start at the beginning state. That means you could resume a game paused last time just simply put the marble back. For demonstration, referring to Figure 27 and Figure 28, after placing the marble, the game could start anytime. If

you want to start first (or the next move is your turn), play your move and then click the button. Otherwise, directly press the button. After you click the button, the phone would play a sound effect to confirm “Now it’s the robot’s turn”, confirming the move is being processed. Behind the scene, the detection server would detect where the marbles at and return back to our mobile application, its log is shown on Figure 29.

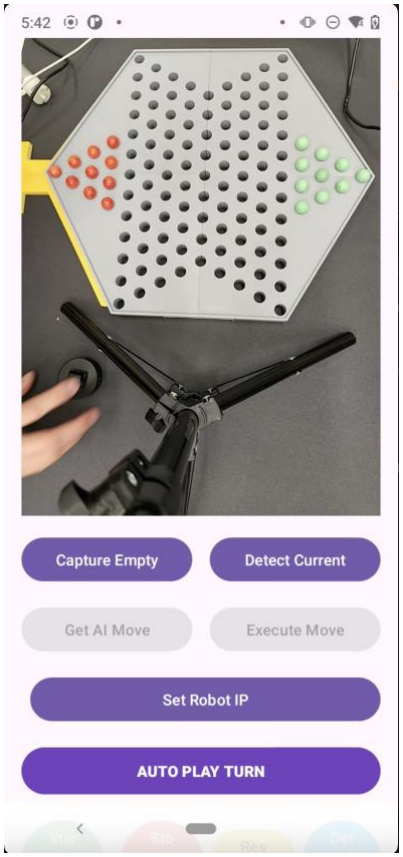


Figure 27 Press Button and Detect Current Board Automatically

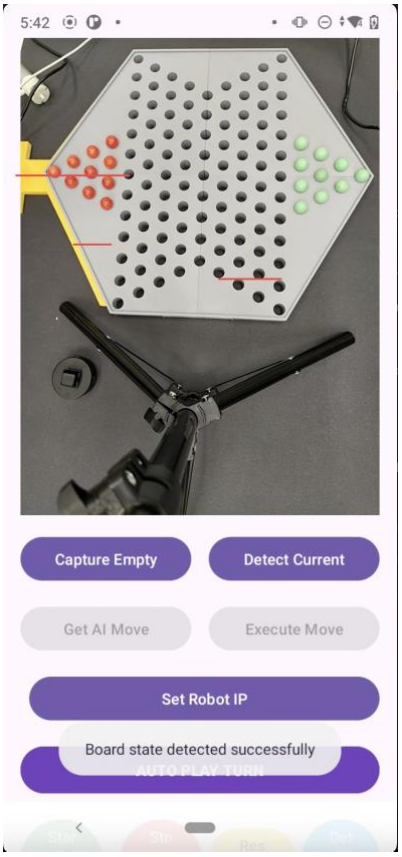


Figure 28 Board State Detected

Figure 29 Cloud Detection Server Log

Step 3 - AI Analysis and Move Selection

After the current board state is detected and analysed, which it will do each turn to ensure the current board state is up-to-date, it would then send the board state to the AI server and get the optimized moves. For example, Figure 30, displays what would the application shows during run-time. However, as users, we actually do not need to touch or even see the screen. The robot would execute itself without our intervention. In the backend, the cloud AI server would analyse the optimal move in real-time, as shown in Figure 31

Figure 30 Application display during runtime

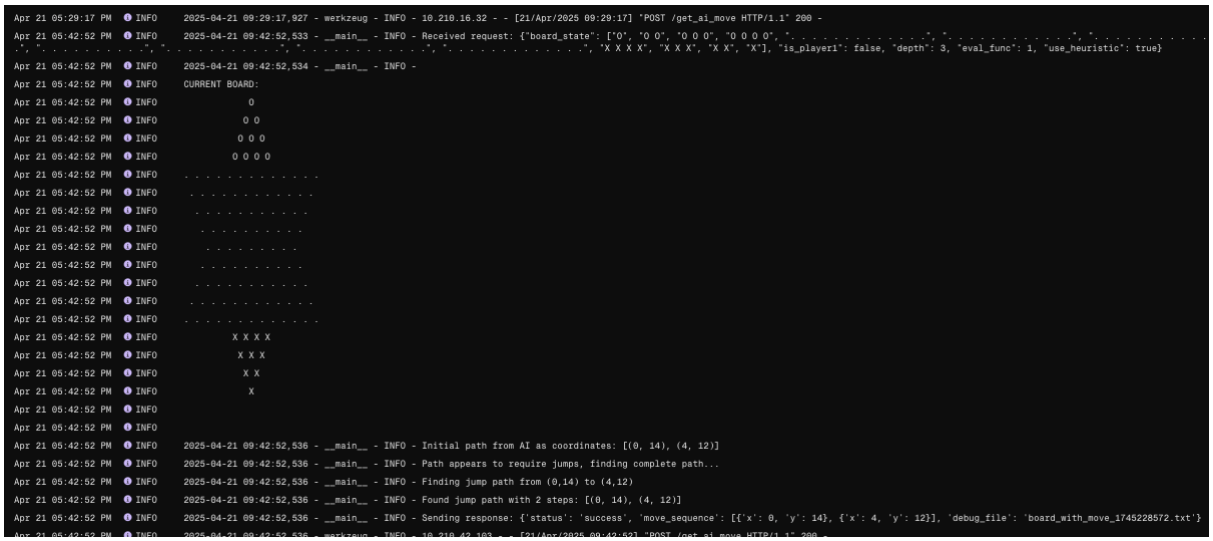


Figure 31 Cloud AI Server Log

Step 4 - Robotic Execution

Once the AI's move is received, the mobile application begins coordinating with the robotic arm to execute it. This process involves:

1. **Arm Positioning**: The arm first moves to a safe height above the origin marble at position. Refer to Figure 32.
2. **Position Adjustment**: The *movement correction algorithm* measures the actual position achieved and calculates necessary adjustments. It is exactly the correction algorithm as mentioned above. It might perform the same action multiple times, for any steps later. It will execute itself to find the best position.
3. **Marble Pickup**: After achieving accurate positioning, the arm descends and activates the vacuum gripper, lifting the marble. As shown in Figure 33
4. **Multi-Jump Path (If applicable)**: The arm sequentially moves through the intermediate jump positions, maintaining a safe height without dropping the marble.
5. **Marble Placement**: At the destination, the arm descends and deactivates the vacuum gripper, releasing the marble. As shown in Figure 34
6. **Return to Home**: The arm returns to its home position, awaiting the human player's next move. As shown in Figure 35

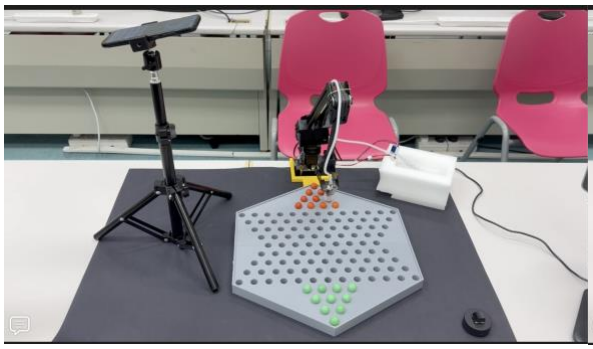


Figure 32 Arm Positioning

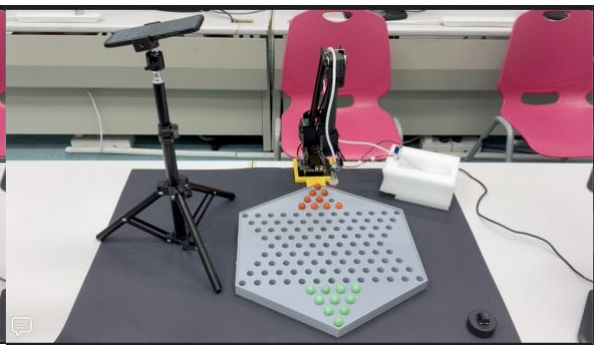


Figure 33 Grab Marbles

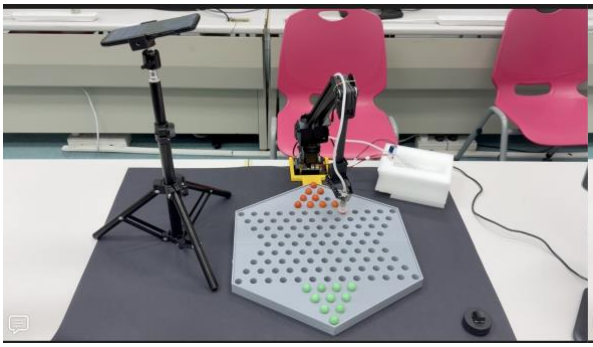


Figure 34 Arrive Destination

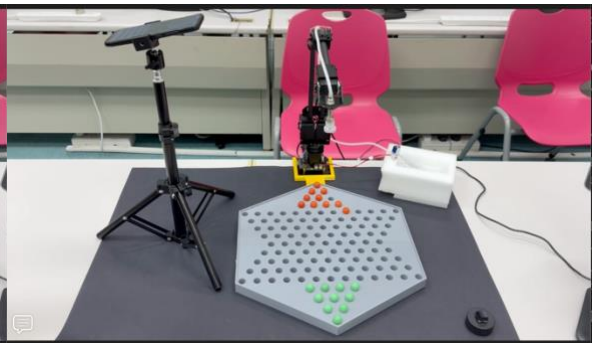


Figure 35 Return Home Position

The abovementioned process would be basically one iteration of the game cycle. Keep continuing the above process to have the game.

3.3 System Performance Observations

Throughout multiple gameplay sessions, we observed several key performance characteristics:

Detection Reliability

The board detection system reliably identified both the empty board structure and marble positions under normal indoor lighting conditions. The most challenging scenario occurred when the lighting environment varies, especially under insufficient lights or directly under sunlight.

The detection process is typically completed within 10 seconds under good network connections, providing responsive feedback during gameplay. It could easily be upgraded by improving the server computing capacity.

3.3.1 Robotic Accuracy

The *movement correction algorithm with overshooting* proved essential for reliable marble manipulation:

Without correction: The arm frequently missed marbles by 3-5mm, making reliable gameplay impossible.

With correction: The arm achieved positioning accuracy within 1mm, enabling successful marble manipulation in more than 95% of attempts.

Occasional pickup failures still occurred at extreme board positions (particularly corner cells), but these represented less than 5% of movements and could be manually assisted when necessary.

3.3.2 Game Flow and User Experience

From a user perspective, the gameplay experience was smooth and engaging. The physical button for ending turns provided intuitive interaction, while the application's status displays kept players informed about the system's activities.

The AI's move calculation typically completed within 1 seconds, with the majority of the turn time (approximately 20-30 seconds) allocated to the physical movement of the robotic arm. While slower than human play, this pacing remained appropriate for casual gameplay and provided ample opportunity to observe and appreciate the robotic manipulation.

3.4 Collaboration Tools

Throughout the development of this project, we employed several collaboration tools to streamline our workflow and ensure efficient system integration.

3.4.1 Version Control

We utilized **Git** as our version control system, maintaining all project components under a single repository. This unified approach facilitated integration testing and ensured consistency across the system's various elements. The repository structure included:

- Computer vision detection algorithms
- AI game engine implementation
- Android application code
- Hardware interface specifications

Using a single repository simplified dependency management and allowed us to track system-wide changes effectively.

3.4.2 Cloud Deployment

For our server components, we leveraged **Render web services** to host both the detection and AI servers. We implemented a Continuous Integration/Continuous Deployment (CI/CD) pipeline that automatically built and deployed our services whenever changes were pushed to the main branch. This automation provided several advantages:

- Immediate deployment of new features and bug fixes
- Consistent build environment
- Automatic validation of code changes

The cloud deployment with automated CI/CD not only enabled reliable connections between our mobile application and server components but also significantly accelerated our development cycle by eliminating manual deployment steps.

3.4.3 Development Environment

Our development process utilized specialized tools for each system component:

- **Android Studio** for mobile application development
- **PlatformIO** for ESP32 firmware programming
- **Rhino 3D** for custom board component design

4. Future Work

While our autonomous Chinese Checkers robot successfully integrates computer vision, artificial intelligence, and robotic actuation into a functional system, several avenues remain for enhancing its robustness and scalability. This chapter outlines proposed improvements across detection, hardware, and user experience.

4.1 Enhanced Board Detection via Machine Learning

Our current detection pipeline relies on traditional computer vision techniques, which, although effective in controlled environments, are sensitive to lighting variability, camera alignment, and partial occlusions. A key improvement would be to replace this with a deep learning-based approach.

Deep Learning-Based Detection

A convolutional neural network (CNN) could be trained to robustly recognize board layouts and marble positions. Benefits include:

- Adaptation to varying lighting conditions without manual tuning
- Improved recognition of partially occluded marbles
- Automatic detection of board orientation and perspective correction

While the development of such a system was beyond the scope of our two-person undergraduate team, it represents a logical next step. This enhancement would require:

- A diverse dataset of board images across lighting and viewing conditions
- Annotation of marble positions and board geometry
- Model training and testing on edge-case scenarios

Such a system could substantially increase recognition reliability.

4.2 Hardware Enhancements for Speed and Precision

The current hardware implementation meets functional requirements but is constrained by the limitations of consumer-grade servos and pneumatic components. Future iterations could benefit from targeted hardware upgrades.

4.2.1 High-Precision Servo Motors

Replacing existing servos with precision actuators (featuring closed-loop control, higher torque, and finer encoders) would yield:

- Greater positional accuracy without reliance on feedback-based correction
- Reduced latency through fewer correction cycles

This would also open the door to smoother movement trajectories and faster turn execution.

4.2.2 Optimized End Effector

The current vacuum gripper could be redesigned with:

- A more responsive pneumatic system for faster grip/release operations
- Integrated proximity or force sensors to fine-tune pickup height

These modifications would significantly reduce average movement duration—potentially lowering execution time from 20–30 seconds per move to under 10 seconds—creating a more fluid and engaging experience.

4.3 Cloud Infrastructure for Scalability and Speed

The backend server currently handles detection and AI computation for single-user scenarios. To support broader deployment, a more scalable architecture is necessary.

4.3.1 Detection Server Performance Optimization

The current detection server is hosted on a basic cloud platform instance, which introduces some latency during the detection phase. Upgrading the cloud infrastructure for the detection server would yield significant performance benefits:

- **Increased computing resources**: Allocating more CPU cores and memory would accelerate image processing operations.
- **GPU acceleration**: Implementing GPU support for computationally intensive tasks like Hough Circle Transform and contour detection would dramatically improve processing speed.
- **Optimized image processing pipeline**: Refactoring the detection algorithms for parallel processing would further reduce latency while maintaining accuracy.

These enhancements would significantly reduce the delay between a player completing their move and the robot beginning its response, creating a more responsive and engaging interaction.

4.3.2 AI Performance Optimization

Upgrading the AI engine could involve:

- Multi-threaded search for faster parallel evaluation
- GPU acceleration for heuristic scoring and deep tree exploration
- Extending search depth beyond 3–4 plies to 5–6 plies for improved strategic foresight

These enhancements would increase AI sophistication while maintaining nearly immediate response times.

4.3.3 Scalable Server Architecture

To support concurrent games or institutional use, the server architecture could be expanded with:

- Dockerized services deployed via Kubernetes
- Auto-scaling based on demand
- Region-specific deployment for latency minimization
- Caching of frequently seen board states for instant retrieval

This would position the system as a deployable multi-user platform for exhibitions, classrooms, or competitive events.

4.4 Additional Gameplay Features

Beyond technical refinements, the following feature enhancements would enrich the user experience:

- **Multi-player support**: Extending AI logic to handle 3–6 player configurations
- **Game recording and replay**: Enabling review, analysis, and learning
- **Interactive tutorial mode**: Guiding users through strategies or robotic operations

Such features would broaden the platform's appeal across educational, casual, and competitive contexts.

4.5 Educational Integration

Given its interdisciplinary scope, the project holds strong potential as an educational tool in robotics, computer vision, and AI. Future efforts may include:

- Developing curriculum-aligned teaching kits and lesson plans
- Tailored tutorial mode for children and elderly
- Designing low-cost kits for hands-on classroom assembly and experimentation

By making the platform accessible to learners, this project could become a practical and inspiring gateway into STEM disciplines, as well as a good leisure buddies for all ages.

5. Conclusion

This project successfully delivers an autonomous Chinese Checkers playing robot that brings together computer vision, artificial intelligence, and robotic control into a unified system. By integrating these modern technologies with a classic board game, we have created a product that is both interactive and culturally meaningful, breathing new life into traditional gameplay.

Over the course of development, we overcame multiple technical challenges. Detecting the hexagonal geometry of the board and accurately locating marbles required customized computer vision pipelines. The mechanical imprecision of consumer-grade servo motors was mitigated through the design and implementation of an adaptive position correction algorithm. Furthermore, the complex nature of Chinese Checkers, with its high branching factor and multi-jump possibilities, called for a carefully optimized AI engine that could provide strategic moves within strict time constraints.

Beyond its entertainment value, this project holds significant educational potential. It serves as a tangible demonstration of how robotics, AI, and computer vision can work together in a real-world application. Its modular, extensible architecture allows each component to be explored and developed independently, making the system well-suited for academic use or future enhancements.

In sum, this autonomous Chinese Checkers robot exemplifies how cutting-edge technology can complement—not replace—traditional cultural experiences. It presents a compelling vision of how classic games can be reimagined for the modern era, blending nostalgia with innovation.

We firmly believe that with continued development, refinement, and community engagement, this system has the potential to grow into a highly polished product. One that not only delights users, but also serves as a meaningful platform for learning, experimentation, and cross-generational enjoyment.

References

Backlinko. (2025). Smartphone Usage Statistics for 2025. https://backlinko.com/smartphone-usage-statistics

Ballard, D. H. (1981). Generalizing the Hough transform to detect arbitrary shapes. *Pattern Recognition*, 13(2), 111–122. https://doi.org/10.1016/0031-3203(81)90009-1

Bradski, G., & Kaehler, A. (2008). *Learning OpenCV: Computer vision with the OpenCV library*. O'Reilly Media, Inc.

Campbell, M., Hoane, A. J., & Hsu, F. (2002). Deep Blue. *Artificial Intelligence*, 134(1–2), 57–83. https://doi.org/10.1016/S0004-3702(01)00129-1

Canny, J. (1986). A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *8*(6), 679–698. https://doi.org/10.1109/TPAMI.1986.4767851

Chakole, M., & Dey, S. (2021). Performance Study of Minimax and Reinforcement Learning Agents in Board Games. *Journal of Intelligent Systems*, 30(1), 1–15. https://doi.org/10.1080/08839514.2021.1934265

Duda, R. O., & Hart, P. E. (1972). Use of the Hough Transformation to detect lines and curves in pictures. *Communications of the ACM*, 15(1), 11–15. https://doi.org/10.1145/361237.361242

Espressif Systems. (2023). ESP32 series datasheet. Retrieved from https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf

F1000Research. (2024). *Chinese Checkers as a strategic thinking tool in undergraduate education*. https://f1000research.com/articles/13-812

Gonzalez, R. C., & Woods, R. E. (2018). *Digital Image Processing* (4th ed.). Pearson.

GSMA. (2024). Analysis to improve handset affordability. https://www.gsma.com/solutions-and-impact/connectivity-for-good/external-affairs/wp-content/uploads/2024/07/Handset-Affordability-GSMAi-Analysis.pd

Innovation Academy. (2020). Playing chess with robotic arm. The University of Hong Kong. Retrieved from https://innoacademy.engg.hku.hk/playchesswithrobot

Jiang, C. (2023). The application of artificial intelligence in board games. *Applied and Computational Engineering*, 4(1), 383–386. https://doi.org/10.54254/2755-2721/4/20230497

Michalewicz, Z., & Fogel, D. B. (2004). *How to solve it: Modern heuristics* (2nd ed.). Springer.

OpenCV. (n.d.). *Geometric Image Transformations*. Retrieved from https://docs.opencv.org/3.4/da/d54/group__imgproc__transform.html

OpenCV. (n.d.). Histogram Equalization. In *OpenCV Documentation*. Retrieved from https://docs.opencv.org/4.x/d5/daf/tutorial_py_histogram_equalization.html

OpenCV. (n.d.). Canny Edge Detection. In *OpenCV Documentation*. Retrieved from https://docs.opencv.org/4.x/da/d22/tutorial_py_canny.html

Pizer, S. M., Amburn, E. P., Austin, J. D., Cromartie, R., Geselowitz, A., Greer, T., ... & Zimmerman, J. B. (1987). Adaptive histogram equalization and its variations. *Computer Vision, Graphics, and Image Processing*, 39(3), 355–368. https://doi.org/10.1016/S0734-189X(87)80186-X

Russell, S. J., & Norvig, P. (2021). *Artificial intelligence: A modern approach* (4th ed.). Pearson.

Serra, J. (1982). *Image Analysis and Mathematical Morphology*. Academic Press.

Shapiro, L. G., & Stockman, G. C. (2001). *Computer Vision*. Prentice Hall.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., ... & Hassabis, D. (2017). Mastering the game of Go without human knowledge. Nature, 550(7676), 354-359.

Suzuki, S., & Abe, K. (1985). Topological structural analysis of digitized binary images by border following. Computer Vision, Graphics, and Image Processing, 30(1), 32–46.

Technion – Israel Institute of Technology. (2025). RoboChess: Chess Playing Robotic Arm. Retrieved from https://crml.eelabs.technion.ac.il/projects/robochess-chess-playing-robotic-arm/

Tong, A., Perez, J., & Stockfish Team. (2023). Stockfish: A strong open source chess engine. arXiv preprint arXiv:2023.xxxxx.

Zhang, R., de Winter, J., Dodou, D., Seyffert, H., & Eisma, Y. B. (2025). An open-source reproducible chess robot for human-robot interaction research. *Frontiers in Robotics and AI*, 12, 1436674. https://doi.org/10.3389/frobt.2025.1436674

Appendices

Rules of Chinese Checkers

Equipment

The Chinese Checkers board is in the shape of a six-pointed star. Each point of the star is a triangle consisting of ten holes (four holes to each side). The interior of the board is a hexagon with each side five holes long. Each triangle is a different colour, and there are six sets of ten marbles with corresponding colours.

Preparation

Chinese Checkers can be played by two, three, four, or six players. For the six-player game, all marbles and triangles are used. If there are four players, play starts in two pairs of opposing triangles. A two-player game should also be played from opposing triangles. In a three-player game, the marbles will start in three triangles equidistant from each other.

Each player chooses a colour, and the ten marbles of that colour are placed in the appropriately coloured triangle.

Objective

The aim of the game is to be the first player to move all ten marbles across the board and into the triangle opposite.

Play

A toss of a coin decides who starts. Players take turns to move a single marble of their own colour. In one turn, a marble may either be simply moved into an adjacent hole or it may make one or more hops over other marbles.

Each hop must be over an adjacent marble and into the vacant hole directly beyond it. Hops may be made over any coloured marble, including the player's own, and can proceed in any of the six directions. After each hop, the player may either finish or, if possible and desired, continue by hopping over another marble. Occasionally, a player will be able to move a marble all the way from the starting triangle into the opposite triangle in a single turn.

Marbles are never removed from the board. A marble may be moved into any hole on the board, including those in triangles belonging to other players. However, once a marble has reached the opposite triangle, it may not be moved out of that triangle—only within it.

Finishing

The first player to occupy all 10 destination holes is the winner.