Imperial College London

MEng Final Year Project Report

Imperial College London

Department of Electrical and Electronic Engineering

An Intelligent Scrabble Board for Alchemist 2024

Author:
Simon Staal

Supervisor:
Prof. Thomas J W Clarke

Second Marker:
Prof. Christos M Papavassiliou

October 26, 2023

Final Report Plagiarism Statement

I affirm that I have submitted, or will submit, an electronic copy of my final year project report to the provided EEE link.

I affirm that I have submitted, or will submit, an identical electronic copy of my final year project to the provided Blackboard module for Plagiarism checking.

I affirm that I have provided explicit references for all the material in my Final Report that is not authored by me, but is represented as my own work.

I have used ChatGPT v4 as an aid in the preparation of my report. I have used it to automate the formatting of raw data into latex, as well as generate code to parse the data produced by my benchmarks. However, all content in this report, as well as all technical work, were produced by me.

Abstract

Although Scrabble boards capable of automatically tracking their state have been developed in the past, none have been commercially viable for standard tournament use. This report presents a novel, low cost, QR-code based approach to accurately track the state of play, and develops an end-to-end system to directly integrate the data captured with a platform to visualise matches, intended for use in the upcoming Alchemist Cup 2024 tournament. Potential challenges such as sensor failure, network outages or power loss are addressed, with comprehensive error detection mechanisms to ensure that both the players and tournament organisers remain informed on the status of the system. The paper also considers alternative approaches to both the game state capture and system design, and justifies why these were inferior to the final solution presented. The performance of the system is evaluated quantitatively, where metrics such as detection accuracy, average total latency and production cost are compared to the functional requirements set by the project sponsor, whilst the robustness and overall quality of the implementation are analysed qualitatively.

Acknowledgements

I would first like to thank my parents, who have done so much to shape me into who I am today, and without whom I would have never developed the drive and resilience to accomplish anything I set my mind to. They are the reason I am at Imperial, and ultimately the source of all my successes.

I would also like to thank Michael Tang, the project sponsor for his sustained support throughout the year, Professors Thomas Clarke, Christos Papavassiliou and Edward Stott for their insights in the early stages of the project, which were particularly helpful in performing the initial design work, as well as Amine Halimi and Phil Jones for their assistance in developing a hardware prototype.

More generally, I'd like to thank all the students, teaching, and support staff I have met over the last four years studying here, you've all shaped my time here to be a truly memorable experience I will look back upon fondly for the rest of my life. The work you all put in into making Imperial such an incredible place of learning does not go unnoticed, and has pushed me enormously, both academically and personally.

Finally, I'd like to thank Professor David Thomas, without whom I would have never discovered my love for programming.

Contents

Chapter 1

Introduction

Scrabble®is a popular board game in which 2–4 players compete by forming words with lettered tiles on a 15 by 15 grid, interlocking like words in a crossword puzzle. The English version of the game is played with 100 tiles, including all 26 letters of the alphabet, as well as 2 blank tiles which can represent any letter. Despite being created close to 80 years ago [1], the game remains popular today, topping the UK sales charts in 2008 [2], and can be found in over 50% of UK homes [3]. Scrabble also has a thriving competitive scene, with tournaments across the world held each week [4]. The World English Language Scrabble®Players Association (WESPA) is the overarching global body for English-language national Scrabble associations, and provides an official structure for the international Scrabble community [5]. Alchemist Cup 2024 is an international competition in affiliation with WESPA, with 10 different countries competing over 45 matches [6].

The organiser for Alchemist Cup 2024 is sponsoring this project to modernise the way competitive Scrabble is broadcast. Currently, games are live-streamed using several cameras to capture the board, player racks and players themselves, and the state of the game is manually transcribed into presentation software by the broadcasting team. An example of this setup is shown in figure 1.1.

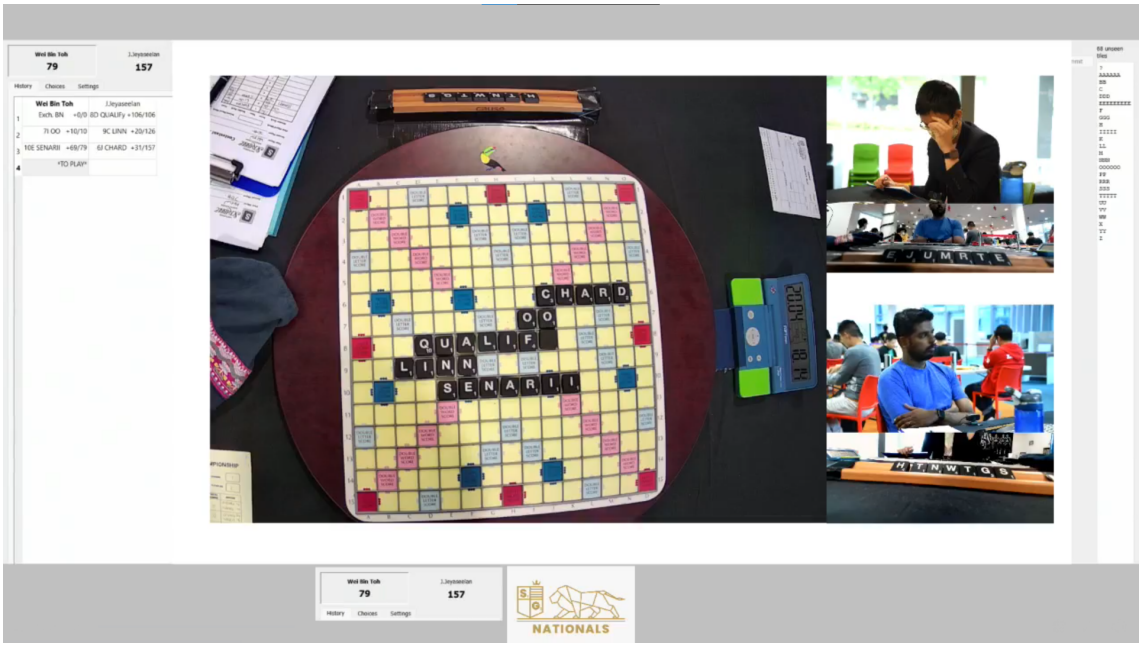


Figure 1.1: Match from the 2022 Singapore National Scrabble Championship

This poses several limitations when compared to tournaments in other games, such as chess, which detract from the viewer experience. Casters are unable to quickly switch between concurrent

matches as only a single game is tracked and visual annotation of future moves is difficult. The goal of this project is to design a solution allowing for the automated digitization of competitive Scrabble, which is implementable for Alchemist Cup 2024.

Recently, several WESPA tournaments have been played on an online Scrabble platform Woogles, including the 2022 Youth Cup [7]. Woogles is developing an API to allow Scrabble moves to be uploaded to the site in real-time, allowing for casters to annotate potential moves, and displaying possible plays using the Macondo crossword AI [8]. The project's sponsor would like the solution developed in this project to interface with the Woogles API so that it can be integrated in the event's broadcast.

1.1 Requirement Capture

Due to the open-ended nature of the project, there are several possible solutions to capture the game state, which are discussed in chapter 3. In order to distinguish between them to find the most suitable one, it was important to obtain a well-defined set of requirements. An iterative approach was employed, in which regular meetings with Michael Tang, the lead organiser for Alchemist Cup and project sponsor, were held, and the various needs of the system were broken down. Although these discussions formed the core the of the project's functional requirements, the importance of code maintainability was highlighted as one of the non-functional requirements to ensure additional development leading the project into a production setting could be performed.

The finalised set of project requirements were broken down as follows in order of importance:

WESPA Conforming Due to its intended use in tournament play, it is critical that the solution correctly encodes and behaves in accordance with WESPA's tournament rules, which are included in Appendix E.

Game State All relevant information regarding the state of the game must be captured with a high degree of accuracy (99.9%), with appropriate processes to verify this performance prior to the start of a match. There should be no more than a 1-second delay between the game state captured and the true state of play, and this information must be represented visually such that it can be included in the tournament live-stream, ideally through the Woogles API. Necessary information includes:

- The tiles played on the board
- The tiles in the players' racks'
- The values of any blank tiles played on the board

Top Scrabble players score $\sim$ 35 points per turn (excluding tile exchanges), and between 330–450 points per game [9], meaning the typical game should take under 26 turns to complete. Given that exchanges are rare [10], it is safe to assume that matches finish within 28 turns. Therefore, a solution capable of detecting the game state correctly with a probability of 99.9% would have 97% likelihood of recording a game without errors. Whilst this means that over the course of the tournament, there is a 75% likelihood that at least one game has an error, they should be rare enough that they could be easily solved through manual intervention, or preferably repeated measurements from the board.

Player Experience The solution must have a minimal impact on the players, ideally conforming to WESPA's equipment preferences:

- Tiles achieve both tactile and visual indistinguishability.
- The board can revolve with minimal disturbance to items on the playing table.
- The board material is non-reflective.
- The number of tiles on a player's rack must be clearly visible to the opponent.

Cost and Production The cost of the solution should not exceed £500 per board, must be manufacturable in a batch of 25 by December 2024.

Word Challenges Competitors may challenge the validity of the words played by their opponent in the previous turn. The solution must therefore ensure that:

- Valid and invalid words are distinguishable from one another.
- A list of currently challengable words must be maintained and made available to competitors when requested.

In addition to these functional requirements, the following non-functional requirements were also imposed:

Error Reporting This system is primarily intended for the broadcasting of matches, and players will still note down moves to keep a manual record of the game. However, any failures in the system should be detected and communicated to both the players and presenters – and ideally easy to recover from.

Maintainability Given that this project is a proof of concept, and may be refined in the year leading up to the tournament, software deliverables should be well architected and documented to facilitate further development.

Chapter 2

Background

As mentioned previously, there are multiple solutions to tackling this problem, and several approaches were considered. As such, the background in this section is quite broad to provide the required context to understand the decisions made in chapter 3.

2.1 Existing Hardware Solutions

2.1.1 DGT Chessboards

Digitizing popular board games for competitive play has been done before, most notably in the case of chess, with the International Chess Federation (FIDE) using electronic boards since 1998 to allow chess games to be followed in real time via the internet [11]. These boards, developed by DGT [12], are able to identify the different chess pieces and their positions on the board using their patent-registered sensor technology [13], solving a very similar problem to identifying the type and position of tiles on a Scrabble board. Chess pieces contain a passive LC circuit with a specific resonance frequency in the 90Hz-350kHz range, depending on its type. The file and rank of a square are selected by analogue switch multiplexers, allowing a particular square to be isolated, and corresponding transmission and reception coils interact with the LC circuit in the piece on the square. The resulting resonance signal obtained on the receiving coil is then converted by the controller firmware into the corresponding chess piece in around 3ms [13]. An example of this operation can be seen in Figure 2.1, where the control device (11) selects a particular transmission coil (18) and reception coil (19) via the multiplexers (10) to measure the resonance signal of the coil in the chess piece (16). Each reception coil comprises several windings (17) to increase its sensitivity.

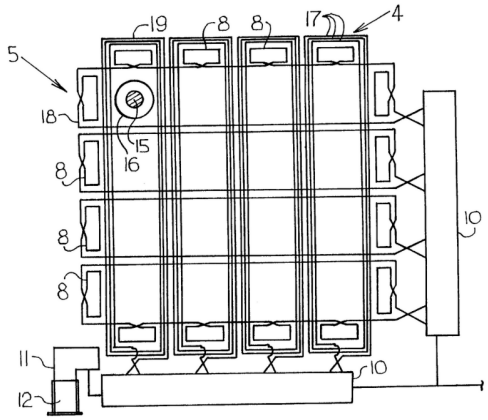


Figure 2.1: Digitised chessboard layout [14]

One issue with this solution is that the transmission and reception coils also generate a resonance

frequency, which interferes with the resonance frequency of the resonance coil in a playing piece. The effect of this interference increases as the topography of the coils around the selected playing square becomes less symmetrical, meaning that the further from the centre a piece is, the greater the inaccuracy. This asymmetry is compensated for by the windings (8) in the transmission coil, which generate an additional magnetic field that supplements the field produced by the transmission coil to render it practically symmetrical. This interference problem would be exacerbated in the case of Scrabble, as not only are the squares roughly a quarter of the size of those in chess, requiring a denser layout of coils, but the play area is a 15 by 15 grid, rather than an 8 by 8 one, leading to greater asymmetry when measuring tiles on the edges of the board. Furthermore, these boards cost over £600 [15], which already exceeds the budget set by the sponsor.

2.1.2 MSI Smart Scrabbleboard

There also exists a smart scrabble board, which was used in the 2012 MSI Prague tournament [16], based on Radio-Frequency Identification (RFID) technology. The tiles are embedded with RFID tags encoding which letter it is, and each square on the board is equipped with an RFID antenna which can detect the tile placed on it. The operation of such an active reader passive tag (ARPT) system is shown in Figure 2.2, where a passive tag consisting of an antenna and application specific integrated circuit (ASIC) chip obtains power from the signal transmitted by the RFID reader. The tag then sends data back by switching its input impedance between a high and low state to modulate the backscattered signal.

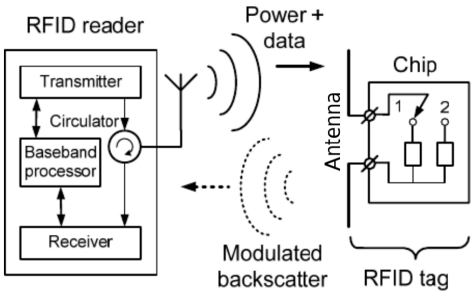


Figure 2.2: ARPT RFID system overview [17]

The MSI board contains nine embedded circuits which cycle through the different antennas to check if a tile is placed on it, and allows a full picture of the board to be constructed in 974ms [18]. The player's racks also included an RFID circuit board, capturing which tiles the players drew. Unfortunately, the combined 100 tags for each tile and 255 antennas for per square on the board make this solution prohibitively expensive, and cost over £20,000 to produce [18].

2.2 Digital Image Processing

With the emergence of low-cost processors and camera peripherals, as well as modern advances in image processing algorithms and computer vision, real-time digital image processing has become more accessible than ever before [19], and offers an alternative approach to tackling the issue of game state capture. The application of digital image processing techniques in scoring scrabble games has already been explored several times [20] [21] [22] [23]. Commercial solutions also exist, such as Scorable [24], a mobile application to automate game scoring. After applying various pre-processing algorithms to isolate the squares on the game board, these solutions use Optical Character Recognition (OCR) to identify the tiles, to varying degrees of success. The use of a k-nearest neighbours algorithm by David Hirschberg from the University of California, based on an earlier work from the University of Sheffield [22] only achieves a classification rate accuracy of 67% [23], and is too unreliable to be considered for this particular application. However, the pre-processing techniques used in capturing the board remain relevant, and are covered in greater depth below. The use of convolutional neural networks (CNN) have a much greater potential, with one solution obtaining over 99.9% accuracy [25]. Although closed-source, the original developer of Scorable, Viliam Vavro, has confirmed via email exchange that their recognition model can perform at practically 100% accuracy in an environment with even lighting and English tiles. As such, an

imaging-based solution appears to be promising, and systems using an embedded Raspberry Pi and camera, as well as the existing camera setup are considered in chapter 3.

2.2.1 Optics

Optics form a key component of machine vision, are and necessary to ensure that the appropriate illumination, lenses, and sensors are selected to capture all the information relevant to a task [26]. The following parameters are of particular importance in ensuring the game state can be captured:

- **Field of View (FOV)** refers to the viewable area of the object under inspection, and is commonly reduced to the horizontal or vertical dimension for ease of calculation. If both values are cited, the horizontal value is given first by convention. For this particular application, the FOV is 322mm by 322mm for the board and 265mm by 20mm for the player racks.

- **Working Distance (WD)** refers to the distance from the front of the surface of the lens to the object under inspection. For optics within the board or racks, the WD is limited to 10cm to avoid becoming too cumbersome for the players. For optics outside the board, the current setup uses a WD of $\sim$ 10cm for the racks and $\sim$ 100cm for the board.

- **Resolution** is an imaging system's ability to reproduce object detail, with a need to capture smaller details requiring a higher resolution. This value can be estimated by dividing the number of horizontal or vertical pixels on a sensor into the size of the object one wishes to observe. The minimum resolution required will depend on the type of image processing performed. Lighting plays an important role in the resolution of an optical system, and is inversely proportional with the wavelength of light; the best results are typically achieved using blue (470nm) light [27].

- **Depth of Field (DOF)** refers to the maximum object depth that can be maintained in acceptable focus, as shown in Figure 2.3. When trying to capture the state of a game of scrabble, DOF can be kept quite low, as the tiles on the racks and board essentially occupy a plane. This means that DOF can be traded off against more important parameters such as resolution, which is inversely proportional to DOF [27].

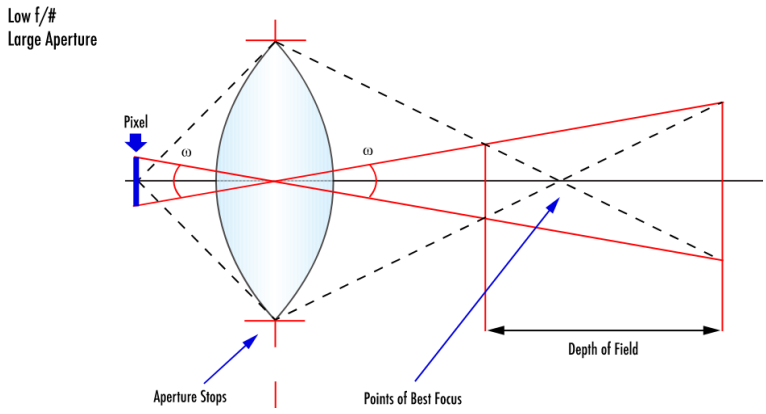


Figure 2.3: Depth of field [28]

To ensure that an optical system is able to perform on a particular FOV with a target WD, an appropriate combination of sensor and lens must be selected. From a physics point of view, the focal length f of a lens is defined as the distance from the back surface of the lens to the plane of the image formed of an object placed infinitely far in front of the lens, meaning that given the length of a sensor l in a given dimension, the angular field of view (AFOV) can be derived as:

$$\text{AFOV} = 2 \times \tan^{-1}\left(\frac{l}{2f}\right) \tag{2.1}$$

Furthermore, as can be seen from Figure 2.4, the AFOV required for a particular FOV and

WD can be found as:

$$\text{AFOV} = 2 \times \tan^{-1}\left(\frac{\text{FOV}}{2 \times \text{WD}}\right) \tag{2.2}$$

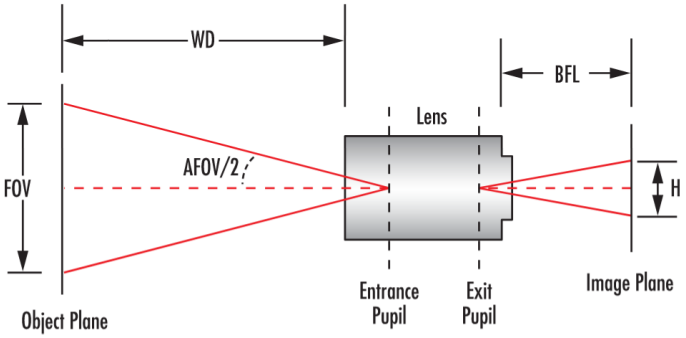


Figure 2.4: Relationship between FOV, WD and focal length for a given sensor [29]

Combining Equation 2.1 and Equation 2.2, one can obtain the focal length required to achieve a target WD and FOV:

$$f = \frac{(l \times \text{WD})}{\text{FOV}} \tag{2.3}$$

In practice, focal length is measured as the distance between the optical centre of a lens and the image sensor, meaning that Equation 2.3 can only serve as an approximation for the appropriate focal length. Furthermore, using an imaging system within the Scrabble board places extreme conditions on the FOV to WD ratio, as ideally the thickness of the board needs to be minimised whilst keeping the area constant, meaning a short focal length is required. Short focal length lenses typically have greater distortion, which can further influence the real AFOV, and have difficulties covering medium to large sensor sizes, restricting the choice of smaller sensors reducing resolution [29]. Additionally, for optimal performance, WD should be 2 to 4 times the FOV [30], leading to further degradation in resolution.

It is important to note that distortion does not reduce the information on an image, but misplaces it geometrically. Rectifying distortions stemming from the use of wide-angle or fisheye lenses has been extensively studied in literature, and there exist algorithms to correct such distortions efficiently [31]. Lighting can also influence distortion, with shorter wavelengths leading to reduced distortion [32]. Given the importance of lighting, controlling the environment of an imaging system significantly improves its reliability and performance [30].

2.2.2 Pre-processing Algorithms

Once a clear image is obtained, there are a range of relevant image processing algorithms that are useful in capturing the game board and tiles. For 2-dimensional images, most algorithms are defined by the 2D convolution of the image itself with a small matrix, often referred to as the kernel. In essence, the kernel defines the function between a pixel in the output image and its neighbourhood of pixels in the input image. Odd-sized kernels are generally used so that the origin of the kernel can be defined as its centre. For some $2b+1$ by $2a+1$ kernel ω, a pixel in the resulting image g from the application of the kernel on the input image f is defined as follows:

$$g(x, y) = \sum_{dx=-a}^{a} \sum_{dy=-b}^{b} \omega(dx, dy) \cdot f(x - dx, y - dy) \tag{2.4}$$

Coloured images are made up of multiple channels of pixels, adding a third dimension to the image. These are processed with concatenations of kernels, called filters, with each kernel assigned to a particular channel. Whilst the additional information that coloured images provide can be useful, and may be required to distinguish between an empty square and a blank tile, they are

nosier and significantly increase the computational complexity of operations [33]. As such, it is often suitable to convert images to greyscale before applying the algorithms below, particularly if an embedded solution with limited computational resources is chosen. This is typically done using the luminosity method, which returns an average of the RGB pixel values weighed by the significance of each colour on human perception [34].

Noise Reduction

Noise in digital image processing refers to unwanted variations in the pixel values caused by factors such as sensor noise, compression artefacts or transmission errors, which degrades the quality of an image and making it more difficult to extract useful information. Given that reliable performance is a critical requirement in this project, reducing noise is key in ensuring that the system is robust, and is often the first step of any image processing algorithm [35]. The most common way to reduce noise is to perform spatial filtering using a Gaussian filter to average pixel values in a local neighbourhood, as it preserves image details more effectively than box filters whilst retaining high computational efficiency. The kernel for this operation is generated by a 2D Normal distribution (as shown in Equation 2.5) around its centre, and is parameterised by its standard deviation σ, which can be increased for a stronger reduction of noise, at the cost of greater smoothing leading to a loss in detail. An example of the Gaussian filter in action is shown in Figure 2.5.

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{\frac{-(x^2+y^2)}{2\sigma^2}} \tag{2.5}$$

Given that the optical system in this project will be dealing with a fixed FOV over a period of time, more complex, non-linear filters incorporating temporal as well as spatial denoising could be relevant if additional noise reduction is necessary [36], although their computational complexity may make them infeasible for an embedded approach.

Figure 2.5: Application of Gaussian filter ($\sigma = 2$) on cameraman standard image

Edge Detection

Edge detection is a technique to determine the boundaries of objects within an image by detecting discontinuities in brightness [37]. This allows physical phenomena to be localised, which is essential for capturing the state of the game board and racks. There have been numerous algorithms that have been developed for this purpose, and it is important to develop a working knowledge of their operation to understand how to maximise their performance. Most edge detection methods rely on finding local maxima of the image gradient, which can be approximated by applying first-order derivative kernels [38]. Such operators include Robert's cross operator, the Prewitt operator and the Sobel operator, although the Sobel operator is the more commonly used for its superior noise-suppression characteristics [39]. To compute the gradient magnitude and direction, the Sobel operator defines a pair of kernels which are applied to the image to obtain the first derivative in the horizontal (G_x) and vertical directions (G_y), which are shown in Equation 2.6.

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} \qquad G_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{2.6}$$

From these resulting images, the edge gradient G and angle θ can be computed as $G = \sqrt{G_x^2 + G_y^2}$, $\theta = \arctan \frac{G_y}{G_x}$. Despite outperforming other operators, this process is still sensitive to noise, producing both false positives and negatives.

The Canny edge detection is a multi-stage algorithm used extensively in engineering applications, relying on the Sobel operator and leveraging several techniques to reduce the impact of noise [40]. The algorithm can be broken down as follows:

1. *Noise reduction:* A Gaussian filter with suitable parameters is applied to the image.

2. *Intensity gradient:* The Sobel operator is applied to determine the magnitude and direction of edge gradients.

3. *Non-maximum suppression:* Candidate edge points are verified to be local maxima in the gradient direction, which is orthogonal to the edge. Non-local maxima are discarded, resulting in a binary image with thin edges.

4. *Hysteresis thresholding:* Two thresholding parameters, $minVal$ and $maxVal$ are used to decide which edges will be kept. Any edges with an intensity gradient $g > maxVal$ are categorized as definite edges, and any edges with $g < minVal$ are definite non-edges. Any pixels in between the two thresholds are only considered edges if they are connected to a definite edge, else they are also ignored.

This algorithm has many tunable parameters affecting the various stages, which are application specific and need to be set correctly in order to ensure reliable performance. However, once these are set correctly, the Canny algorithm provides a computationally efficient, robust method in detecting edges with a high degree of accuracy [40]. OpenCV, a popular python based computer vision library, provides an implementation of Canny edge detection [41] which was used with default parameters to obtain Figure 2.6.

Figure 2.6: Application of Canny edge detection on cameraman standard image

Structured edge detection is a more novel edge detection algorithm which applies random decision forests to predict local edge masks, and once trained was able to achieve superior real-time performance to the Canny algorithm [42]. Although training a model for edge detection in this specific application is outside the scope of this project, should the Canny algorithm prove to be a performance bottleneck in the final system a using structured edge detection with a pre-trained model provides an alternative computationally inexpensive solution.

Contour Detection

Contours are curves joining all continuous points along a boundary, having the same colour or intensity, and are particularly useful for shape analysis and object detection and recognition [43]. Contour detection aims to detect these curves in an image, providing a compact description of its shape including information such as its centroid and bounding rectangle, which in this application is particularly useful for isolating the letters on tiles for categorization. Although several approaches exist, Satoshi Suzuki's border following algorithm is a popular, robust and efficient algorithm building on edge detection [44] which is implemented in numerous image processing libraries such as OpenCV [45]. In order to obtain high accuracy when applying this algorithm, it is important to use a binary image, such as one produced by the Canny edge detection algorithm [46]. As this algorithm does not depend on tunable parameters and is primarily a function of the binary image provided as input, a deep understanding of its operation is not particularly relevant. If its performance is not reliable enough, and cannot be improved through adjustments in upstream processing, alternative contour detection algorithms may be considered.

Feature Detection and Matching

Although no exact, universal definition exists, features typically refer to "interesting" points on an image, which are ideally invariant under changes in illumination, translation, scale and in-plane rotation. Once identified, key elements of this feature are encoded in a feature description, which can then be used to identify the same features in other images, essentially providing a mapping from one image to the other, in this case providing an effective and robust method of isolating the Scrabble board. These feature descriptions are the core part of matching techniques, as perhaps obviously the performance of the matching algorithm is a function of the description quality. The choice of feature detector, which generate these descriptions, is therefore a key parameter in the performance of the overall system. A wide range of feature detection algorithms exist, and developing a working understanding of their properties is essential in ensuring appropriate detectors are used.

Corners are regions in an image with a large variation in intensity in all directions, making them strong features [47] as they are essentially unique points. Harris corner detection is an early feature detection algorithm focused on identifying these corners, relying in part on the Sobel operator mentioned in previous sections to obtain the image derivatives [48]. Improvements to this algorithm were later made by Shi and Thomasi which uses a different selection criterion which is more computationally efficient and ensures better features are chosen [49]. Although they are rotationally-invariant, as corners are unaffected by a change in orientation, changes in scale can affect the properties of a corner, leading to a rapid deterioration in performance as the scale of features is adjusted [50]. This may be an issue for this application, as there may be variance in the distance between the camera and the board, which although minimal could degrade the reliability of the system. Strict constraints on the positioning of the camera could be imposed to mitigate this issue, although the system would remain less robust to mistakes in setup, and hence alternative methods would be preferable.

Scale-Invariant Feature Transform (SIFT) is able to achieve scale-invariance through the use of scale-space filtering [51], in which Gaussian filters smooth the image to different levels of details representing different scales, allowing features stable in scale-space to be identified. This filtering is approximated using a difference of Gaussian, which although more efficient than Laplacian of Gaussian (LoG) that would be ideally required [51] remains quite costly, making the algorithm computationally slow compared to those discussed previously and potentially infeasible for the real-time system this project is designing. However, the descriptors returned are highly robust and perform incredibly well compared to other techniques [52]. Speeded-Up Robust Features (SURF), as the name suggests, is a more efficient version of SIFT, approximating the LoG more aggressively with a box linear filter, as well as other optimisations to obtain a 3 times speed improvement, whilst retaining comparable performance to SIFT [53]. The matching time of these algorithms can be further improved through the transformation and condensing the feature space, allowing a single computation of the Hamming distance and reducing the memory footprint through the application of Binary Robust Independent Elementary Features (BRIEF) in conjunction with these matching algorithms, which lead to a 4- to 13-fold speed-up over SURF depending on the level of

compression [54]. However, this technique is not-designed to be rotation invariant, and although capable of handling small perturbations in angle [54] would lead to significant deterioration in performance, and as such was deemed unsuitable for this application.

If the performance of SIFT and SURF prove inadequate, alternative machine learning-based techniques such as Features from Accelerated Segment Test (FAST), which achieve more rapid classification whilst retaining the rotation- and scale-invariant properties [55]. This algorithm uses a decision tree classifier to speed up the detection of corners, with higher accuracies obtained when trained on the target application domain, although pre-trained general classifiers can also be used, as well as relying on a tunable threshold value for the intensity of corners and suffering a higher sensitivity to noise [55], making it less robust than SIFT or SURF. Binary Robust Invariant Scalable Keypoints (BRISK) builds on FAST, as well as incorporating a BRIEF-like feature descriptor for a matching rate an order of magnitude faster than SURF and almost equivalent reliability, only performing slightly worse for large transformations [56]. As well as inheriting the threshold parameter from FAST, classification speed vs robustness to scale-invariance can be traded off with another parameter which essentially defines the number of scales the image is sampled at to obtain reliable features [56]. It is not possible to determine which of the previously discussed detectors is the most appropriate for this project solely based on theoretical considerations. Rather, the trade-offs between the different algorithms present a systematic method in which they can be tested experimentally to determine the most robust algorithm that meets the performance requirements of the system.

Once a feature description algorithm is selected, matches are typically found by determining the "closest neighbour" to the target feature space, in which the distance metric selected depends on the feature description. For SIFT and SURF, the L2-norm typically provides the best results, whilst binary string descriptors like BRIEF and BRISK can use Hamming norm instead [57]. Open-CV also offers a Fast Library for Approximate Nearest Neighbours (FLANN) based matcher which is optimized for large datasets with high-dimensional features [57], although this would likely be unnecessary for the purposes of detecting the game board. Given enough matches between the two target objects, a homography, which is a 3 by 3 matrix defining the transformation between two points, can be computed to best fit all the matches, allowing the inverse transformation to be applied to the matched object such that it matches the target object. This transformation of the board can introduce a parallax error to game pieces on it, as can be seen in Figure 2.7, although this is unlikely to pose an issue in this application as the camera line of sight will be vertical.

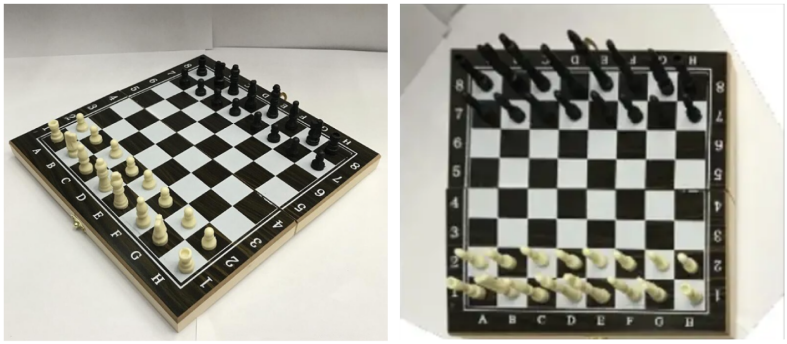


Figure 2.7: Homography applied to obtain bird's eye view of chessboard [58]

2.2.3 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a type of deep learning neural network architecture that is specifically designed for image recognition or other tasks that involve pixel data, inspired by the human visual cortex [59]. The application of CNNs to optical character recognition has been studied extensively [60], and the use of either pre-trained or custom-fit CNNs to the problem of categorizing scrabble tiles is essential in attaining the tight accuracy and reliability requirements of this project.

CNNs are comprised of three primary building blocks, convolutional layers, pooling layers and

fully-connected layers. Convolutional layers employ the familiar convolution operation, but now the elements of the kernel are parameters which are optimised during training, allowing local patterns in small 2D-windows of the input image to be learned [61]. The results of the convolution are then passed through an activation function, which allow non-linear behaviour to be captured. In deep learning applications, the `relu` function, which simply restricts values to be non-negative, is the most popular [61], and as such would most likely be selected. These operations are performed independently over each channel of the image, although in this application only 2D gray-scale images will be used, as colour will be removed in the input pre-processing.

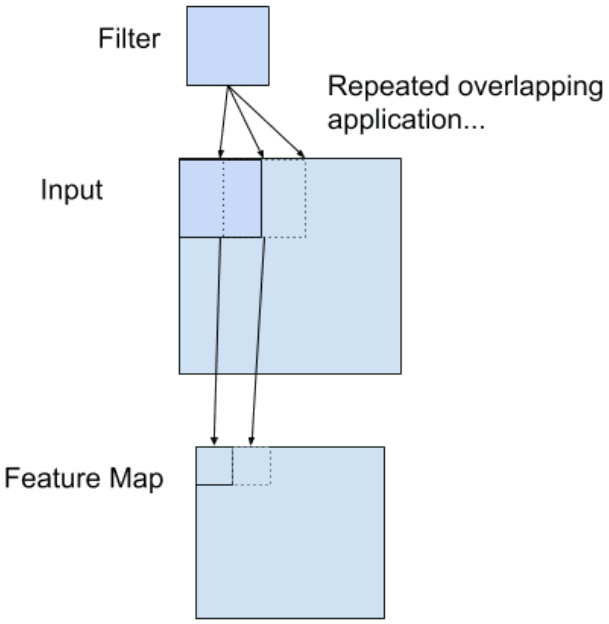


Figure 2.8: An example of a 2D convolution operation [62]

The resulting feature map is then down-sampled using pooling layers, which reduce the spatial resolution to increase the invariance of the features to small translations in the image. The down-sampling also allows spatial-filter hierarchies to be built, where successive convolution layers inspect increasingly larger windows, with the final layer accessing information from the totality of the input. Although this down-sampling can be achieved using strides in the convolution layers, in which the kernel operation "jumps" over elements in the grid, or average pooling which takes the average value of each channel over a given patch, max pooling where the maximum value in each patch is selected tends to perform best [61].

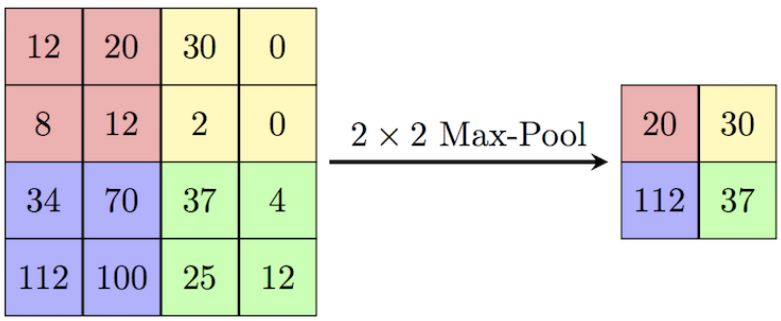


Figure 2.9: An example of a 2 by 2 maxpooling operation [63]

After several layers of convolution and pooling, the feature space is flattened into 1D to be fed into fully connected layers found in traditional neural networks, in which all the outputs from the previous layer are fed into every neuron of the current layer. These layers are ultimately responsible

for the classification of the image, where a softmax activation function in the final layer returns a probability distribution of the possible results. Obtaining probabilities is incredibly powerful, as any uncertain measurements can be repeated on a fresh image.

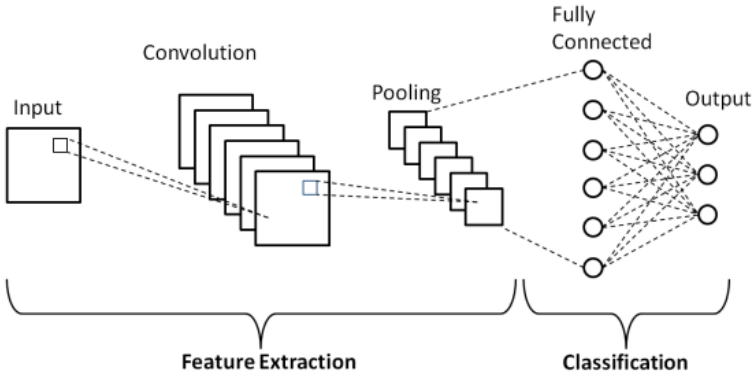


Figure 2.10: Schematic diagram of a basic CNN architecture [64]

F. Chollet describes a simple CNN architecture which achieves a 99.2% classification accuracy on the MNIST database comprised of 28 by 28 pixel binary images of handwritten digits [65]. Not only is performing OCR on printed text much easier than recognising human handwriting [66], but restricting the character class to fewer fonts further reduces the complexity of the problem [67]. Given that the tournament scrabble tiles all use the same font, this categorization problem is therefore much simpler, despite needing to classify between 27 rather than 10 characters. Although tiles which appear similar, such as 'O' and 'Q' could be problematic, the score in the bottom right corner of the tile can help disambiguate between these two. As such, should a custom CNN implementation be required for this project, a similar architecture will be considered.

2.2.4 QR Codes

QR codes are two-dimensional barcodes that store information which can be rapidly and reliably decoded in any orientation by a camera-equipped device. This provides an alternative way to encode the tile information, where a QR code could be printed on the bottom of a tile, and then captured by a camera inside the board. These codes are available in a range of sizes defined by the number of black and white squares, called modules, that make up a row, each capable of storing different amounts of data at different levels of error correction [68].

As these codes must fit on the underside of a standard scrabble tile, their total area is limited to 14mm by 14mm, meaning a trade-off between the number of modules and the module size can be made. More modules allow for more data to be encoded, which is not relevant in this application as only 2 numeric characters are required to represent all possible tile values, but also provide higher levels of error correction. On the other hand, larger modules make it easier for the QR code reader to resolve individual modules, which can help to improve the accuracy and reliability of the code reading process. The different break points for QR code size and error correction, expressed as the percentage of recoverable data bytes, for this particular application are shown in Table 2.1. QR codes also require padding to be detected reliably. Micro QR Codes, which have symbol versions prefixed with 'M', only require 2 modules of padding, whilst standard codes require 4 modules of padding, leading to a larger step in the number of modules between these versions.

Symbol Version	Data size	Total size	Capacity	Maximum Error Correction
M1	11	15	5	0%
M2	13	17	8	15%
M4	17	21	21	25%
1	21	29	17	30%

Table 2.1: QR Code error correction, data capacity in # of digits and size in # of modules for Scrabble tile encoding

Depending on the size of the state space required to encode the tile values, which is evaluated in subsection 3.2.2, further error correction for all symbol versions can be achieved through the design of a custom Reed-Solomon encoding scheme, which fully utilises the capacity of the chosen QR code. The value provided by such a scheme is dependent on the measured detection accuracy of the system, and is discussed in chapter 6.

2.3 Application Architecture

Based on the non-functional requirements discussed in section 1.1, ensuring the software deliverables are well-designed is essential, as this will not only ensure that code is maintainable, but facilitate testing to validate the robustness of the system.

2.3.1 Android

In order to communicate with players, the solution in this paper will use an Android application. The rationale behind this decision is explored later in section 3.3, but this section will provide context on the application life-cycle, state and design patterns necessary to understand the choices made during implementation.

Although there exist a number of development frameworks for building Android applications, this report focuses on Jetpack Compose, a modern toolkit currently recommended by Android as the best way of building native UI [69]. Compose allows developers to describe the application's UI using a declarative syntax, where "composable" functions define what should be rendered based on the state passed into them. Any changes to the state passed into these components causes them to be "re-composed" to display this new state, abstracting away the need to manually update the UI and ensuring it remains synchronized with the underlying data [70]. The application state, as well as the business logic handling it, can be moved outside these components following a design pattern known as "state hoisting", decoupling it from the UI and making the code much easier to maintain [71].

Figure 2.11: The lifecycle of an android activity [72]

State hoisting is also important to ensure that information is not lost across recompositions, where composable functions are effectively called with new arguments, or even activity recreation caused by configuration changes, such as a user rotating the screen. The different stages of an Android application, or activity, lifecycle is shown in Figure 2.11. Whilst there exist mechanisms which can preserve this state within composables, these are much more prone to programmer error,

and using a `ViewModel` to control the UI state is considered better practice as it persists your state, even across process recreation, as well as encapsulating your business logic [73]. The lifetime of the `ViewModel` relative to the various application states is illustrated in Figure 2.12.

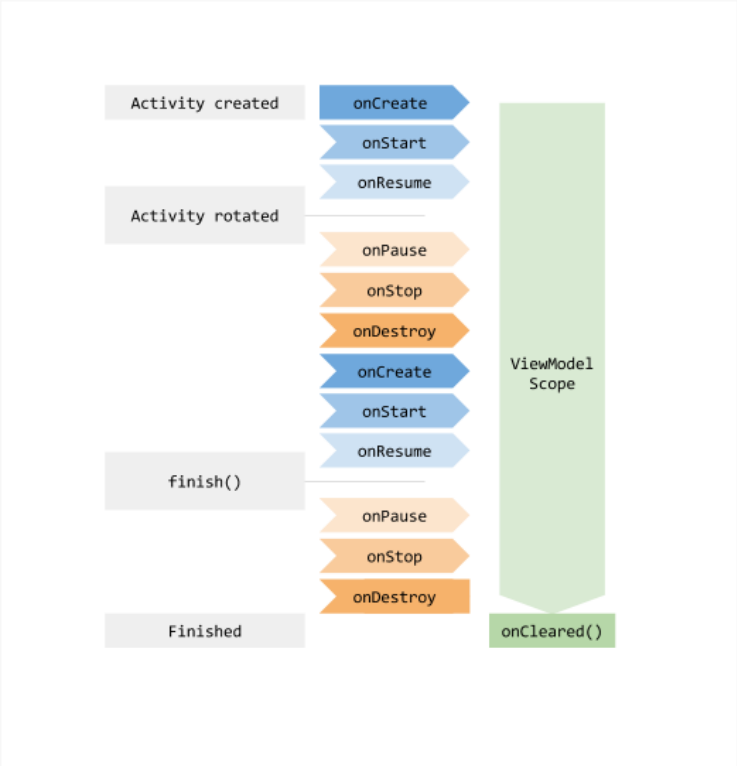


Figure 2.12: The various lifecycle states of an activity and the lifetime of its corresponding `ViewModel` [73]

The decoupling of the composable elements which take in an immutable UI state, and emit events back to the `ViewModel` which then updates the state implements a unidirectional data flow design pattern which presents a number of advantages, allowing both the UI and business logic to be unit tested separately, as well as encapsulating the state into a single source of truth, reducing the likelihood of bugs [74]. The `ViewModel` can then propagate these changes back to the data layer, which in this application would be responsible for networking. An example of an interaction between these different components is presented in Figure 2.13, and provides a foundation for the architecture of the companion application.

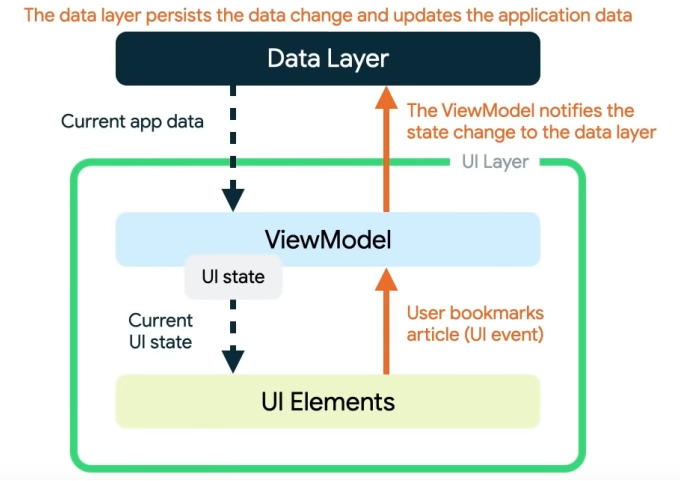


Figure 2.13: An example of the interaction between the data and UI layers [75]

2.3.2 Communication Protocols

Understanding the properties of different networking protocols is essential for selecting the communication methods between the server-client system architecture outlined in section 3.3.

Socket Programming

A socket is a communications connection point which enable inter-process communication, both locally or across networks. Network sockets can be characterised by a local socket address, typically comprising an IP address and port number, transport layer protocol, and when connected, a remote socket address. Once established, the socket API allows two-way communication, meaning both the server and client can send messages to each other. The properties of communicating across sockets is identical to the proprieties of the underlying protocol they use; of which Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) are the two most common [76], whose relevant properties are compared in Table 2.2.

Property	TCP	UDP
Delivery	+ Reliable, ordered	− Unreliable, out-of-order
Overhead	− Larger packets, retransmission	+ Minimal overhead
Congestion control	+ Automatically managed	− User implemented

Table 2.2: Comparison of the properties of UDP and TCP

UDP also utilises fewer OS resources as connections are not maintained, making it more efficient for handling large numbers of concurrent clients. However, this is not particularly relevant in this application, where only $4 * 25 = 100$ connections at most will require simultaneous processing from a central server, orders of magnitude smaller than the expected load of a typical application.

UDP's lower overhead makes it commonly used over TCP in applications which prioritize low latency and real-time communication, such as video streaming or online gaming, where occasional packet loss or out-of-order delivery can be tolerated. In paper [77], the latency incurred by both TCP- and UDP-based protocols is measured under a range of network conditions for IoT applications, allowing the penalty induced by TCP's re-transmission due to packet loss to be estimated. Given expected deployment in a high-quality wireless network environment, the packet loss rate should remain close to 0, but may fluctuate to up to 2.5% [78]. To ensure robust performance across a range of network conditions, losses of up to 10% were considered, which would only be attainable in incredibly high interference environments, corresponding to a penalty of approximately 25ms-75ms depending on packet size [77], which is relatively insignificant in this application, where a 1000ms delay between the state of play and tournament broadcast is acceptable.

Although highly performant, communicating over raw sockets has a substantial development penalty associated with it, as communication is handled at the transport layer, meaning that further processing is typically required to make the packets received useful for application-level processing. As such, it is recommended to avoid communication across raw sockets unless absolutely necessary due to hardware constraints [79]. This can be remedied through the use of serialization and Remote Procedure Call (RPC) frameworks, which consume data from the socket and allow the developer to specify logic at the application level, focusing the definition of the protocol and implementation of the logic associated with it rather than low-level networking challenges such as transforming the stream of packets into meaningful application data. Depending on the selection of framework, these can provide practically zero networking overhead compared to the use of raw sockets, simply performing some additional processing when serializing and deserializing data received from the wire [80][81].

Message Queuing Telemetry Transport (MQTT)

MQTT is a lightweight, application-layer messaging protocol [82] which is commonly used in Internet of Things (IoT) data collection applications [83]. Built on TCP/IP, this protocol follows the publish/subscribe model, in which a dedicated broker passes messages published on a given channel to all clients which are subscribed to that channel. This is particularly beneficial in cases where the same data needs to be disseminated to large numbers of devices, as it shifts necessary

processing for such operations out of a central server into a separate, dedicated component; although this feature of MQTT isn't particularly relevant in this case, as the different components of the system would each have different roles, each requiring individual communication. MQTT also offers quality of service levels for message delivery; 0 – at most once, 1 – at least once, and 2 – exactly once, which can be configured per message. Selecting the reliability level suitable for each type of message can help reduce the overhead of communication, although these reliability mechanisms run on top of the existing TCP stack, making them less efficient than raw sockets in all cases.

HyperText Transfer Protocol (HTTP)

HTTP is a widely adopted communication protocol utilized for web-based interactions between clients and servers. Some of its key features relevant to the design of the system include [84]:

Request-Response Model : A client initiates an HTTP transaction by sending a request to the server, which then provides a response. Although 2-way communication can be emulated with polling, where a client sends frequent requests to the server for data, this is much less efficient compared to the protocols discussed above.

High Level : As an application-layer protocol, HTTP abstracts away many challenges which must be handled by the user in lower-level protocols; the server is guaranteed to have access to a client's full request without worrying about the underlying packets, although it has higher overhead as a result.

Reliable Data Transfer : Given its underlying use of TCP, which is connection-based, HTTP guarantees that messages are received in-order, ensuring any dropped packets are re-transmitted and detecting and correcting data corruption.

Stateless : There is no link between two requests successively carried out on the same connection, although HTTP cookies can be used to create stateful sessions.

Although having substantially greater overhead than any of the other communication protocols discussed above [83] [85], HTTP's ubiquity as a web protocol makes it an attractive option in applications where performance is not critical.

Chapter 3

Design

This chapter outlines the approach to the design of the final product. The different implementations considered in addressing the functional requirements defined in section 1.1 are explored, with a particular focus on accurately capturing the tiles on the board and player racks, as this is the most challenging problem to overcome. The architecture of the overall system is also explored, justifying the selection of communication protocols and technology stack forming the basis of the implementation outlined in chapter 4.

3.1 Hardware Solution

The core issue with the existing hardware implementations discussed in section 2.1 is their high cost. However, the use of an integrated hardware solution remains attractive, providing a compact and intuitive solution requiring minimal effort to set up for tournament use, and having minimal impact on the competitors. A resistive matrix approach was considered, in which the board would contain a 15 by 15 matrix with 2 exposed contact points on the edges of each square. The rows and columns on the board could be selected via a multiplexer to isolate particular squares, allowing the resistance between the contact points to be measured with a potential divider, as shown in Figure 3.1. Each tile would contain a unique resistance encoding its value, with matching contact points on its base to form a closed circuit when placed on a square. An analogue to digital (A2D) converter could then be used to pass the measured value to an embedded computer, which could then classify the tile placed.

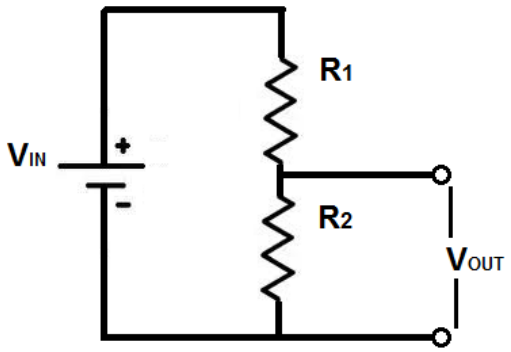


Figure 3.1: Potential divider circuit

To first determine the feasibility of such a solution, one must first verify that a wide enough range of suitable resistor values exist to encode the 27 distinct tile values. The E96 resistor range [86] was considered for this, as the low $\pm 1\%$ tolerance ensures that a high reliability can be achieved. When considering a circuit with an input voltage of $5V$ and an operating current range between $10\mu A - 10mA$, which is typical of such an application, Ohm's Law provides a bound on the total resistance of the circuit of $500 \leq (R_1 + R_2) \leq 500k$. The $5V$ range can now be split

into 28 intervals such that $V_{out_i} = \frac{i}{28}V_{in}$, giving a $179mV$ difference between each tile, and the corresponding resistor value for the i-th tile can be found with Equation 3.1.

$$R_i = \frac{R_1 V_{out_i}}{V_{in} - V_{out_i}} = \frac{iR_1}{28 - i} \tag{3.1}$$

Several values of R_1 were tested when generating the range of R_i values, and optimal performance was found with $R_1 = 10k\Omega$, minimising the power consumption of the voltage divider whilst keeping the total resistance within the acceptable bounds. The optimal R_i was then mapped to its closest E96 resistor, as shown in Table 3.1, such that the worst case error analysis could be performed.

i	Optimal R_i (Ω)	Closest E96 Resistor (Ω)
1	370	3.74e+02
2	769	7.68e+02
3	1200	1.21e+03
4	1667	1.65e+03
5	2174	2.15e+03
6	2727	2.74e+03
7	3333	3.32e+03
8	4000	4.02e+03
9	4737	4.75e+03
10	5556	5.62e+03
11	6471	6.49e+03
12	7500	7.5e+03
13	8667	8.66e+03
14	10000	1.0e+04
15	11538	1.15e+04
16	13333	1.33e+04
17	15455	1.54e+04
18	18000	1.82e+04
19	21111	2.1e+04
20	25000	2.49e+04
21	30000	3.01e+04
22	36667	3.65e+04
23	46000	4.64e+04
24	60000	6.04e+04
25	83333	8.25e+04
26	130000	1.3e+05
27	270000	2.67e+05

Table 3.1: Resistor values for tile encoding

By taking the maximum and minimum resistances given the $\pm 1\%$ tolerance of the E96 series, the maximum, and minimum voltage reading corresponding to a particular tile can be calculated. Comparing these extremes to neighbouring i values then allows the worst-case difference in voltage to be calculated, which can then be compared to the ideal $179mV$ difference between levels. For the resistor values found in Table 3.1, the average worst-case difference was $156mV$, and the smallest worst-case difference was $138mV$. The table containing the full range of values can be found in Appendix B. Compared to the $0.61mV$ quantization error of a 12-bit A2D converter, even in the worst-case scenario the resistor values should be distinct enough to identify accurately.

The results above show that provided the tiles form a strong contact with the board, tile values can be encoded in such a way that they could be easily distinguished from one another. Unfortunately, it is likely that contact resistance would have a substantial distorting effect on the actual resistance values measured by the circuit in practice for a number of reasons. Contact resistance is inversely proportional to the force applied to the contact [87]. Given that standard scrabble tiles weigh $\sim$ 1g, the force applied to the contact from the weight of the tile alone would be

around $0.01N$, which is orders of magnitude lower than the $1N$ minimum commonly required for a reliable contact [88]. A solution with magnets embedded under each square on the board, with metal plates inside each tile could ensure this minimum is reached, but this would add complexity to the construction of the board and tiles, increase costs, and require the tiles to be demagnetized between rounds to avoid having them stick together. Additionally, the contacts would oxidise over time, further degrading their strength over the lifetime of the board. As such, this solution was ultimately deemed too unreliable for this application, although if methods to overcome the contact resistance issue are discovered, future designs following this approach may be useful to consider.

3.2 Optical Solution

Two different optical solutions were explored, one using the existing camera setup used for live-streaming matches with OCR, and one using cameras built into the board with QR codes on the bottom of tiles.

3.2.1 Optical Character Recognition (OCR)

This solution would use the existing camera setup to capture live footage of the games, then use digital image processing and a convolutional neural network to segment the board into squares and determine the tile on a particular square. For the racks, pre-processing would be used to isolate and rotate the tiles in the correct orientation, resulting in images that can be passed into the same neural network for categorization.

Based on comparable OCR problems [65] [25], a minimum resolution of 30 by 30 pixels would be required to accurately identify a particular tile measuring 19 by 19 mm, meaning a pixel density ρ at least 1.58 pixels per millimetre (ppmm) are required. This value needs to be compared with the resolutions obtained using the current streaming setup to ensure that this approach is feasible, which can be accomplished using simple trigonometry. A visualisation of the problem is shown in Figure 3.2.

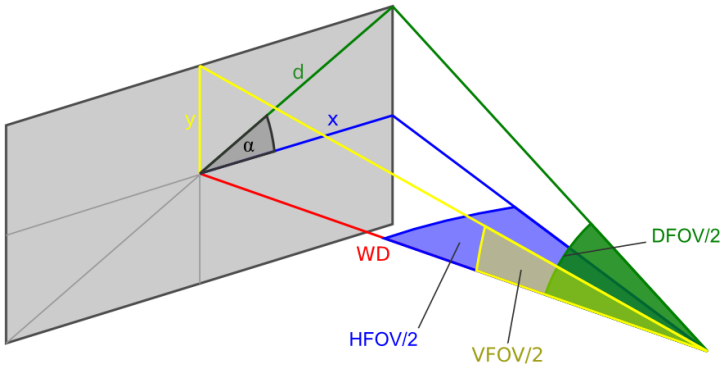


Figure 3.2: Obtaining pixel density from AFOV

Full high-definition webcams are used to capture footage, meaning that images received are 1920 by 1080 pixels, and have a standard diagonal AFOV of 78°. Using this information, the following can be derived:

$$\begin{aligned} \alpha = \arctan\left(\frac{1080}{1920}\right) \approx 29.4^\circ \qquad & d = \mathrm{WD} \cdot \tan(39^\circ) \\ x = d \cdot \cos(\alpha), \qquad & y = d \cdot \sin(\alpha) \\ \rho = \frac{1920}{x} = & \frac{1920}{\mathrm{WD} \cdot \tan(39^\circ) \cdot \cos(29.4^\circ)} \end{aligned} \tag{3.2}$$

Using Equation 3.2, the pixel density ρ can now be found as a function of the working distance (WD). Recalling the working distances of 100 cm and 10 cm for the board and rack respectively

from subsection 2.2.1, $\rho_{\text{board}} = 2.72$ ppmm and $\rho_{\text{rack}} = 27.2$ ppmm, both providing more than sufficient resolution to detect tiles accurately.

The feature description and matching techniques described in chapter 2 can be applied to obtain a normalise image of the board, and an individual square can be isolated by super-imposing a reference grid onto the board. Grid lines can also be determined with a k = 15 k-clustering algorithm as described by Hirschberg [23], although its 94% classification rate is likely to render it infeasible in this application.

Once game squares have been isolated, there are a number of ways to determine the presence of a tile. Given that the tiles and squares have distinct colour values, as shown in Figure 3.3, a simple option would be to compute a histogram of the colour values on a given square, and compare the distance between the peaks for an empty square. Should this difference exceed some error threshold, the square can be categorized as occupied, and individual letter detection can be applied. Unfortunately, Scrabble tournament boards are not standardised, and although the sponsor has guaranteed that all the boards and tiles used in Alchemist Cup 2024 will be identical, this would make the system quite brittle and difficult to apply to other events without adjustments. Instead, a contour detection approach could be considered, where the area of the largest bounding box on the square is used to determine the presence of a tile. As the empty squares do not contain large distinct regions, especially compared to the size of the letters on the tiles, only squares containing sufficiently large contour would be categorized as occupied. Given that contour detection operates on binary images, this method would be more robust to changes in lighting or colouration, although the text on empty special squares may cause interference with the contour detection algorithm. Should this be an issue, a combined approach may be suitable, where colour histograms are used for special squares, which have standardised colours, whilst contour detection is used for the remaining area of the board.

Figure 3.3: Tournament board and tiles for Alchemist Cup 2024

The process above should produce cropped images containing tiles for categorization. If this solution were to be selected, a pre-trained OCR engine such as Tesseract, one of the market leading open-sourced engines [89] would first be tested. Although it is outperformed [66] by Google Cloud Platform's Vision OCR [90] and Amazon's Textract [91], the use of these online APIs would increase project costs, as well as increase the number of external dependencies of the system. If insufficient accuracy is obtained with these tools, a custom CNN would be developed to classify these tiles

using the relevant techniques discussed in chapter 2.

To identify tiles on racks, Canny edge detection can be used to identify tile outlines, from which area computations can be performed to filter out invalid matches. If necessary, the racks can be coloured to increase contrast between themselves, the tiles and the surrounding environment, facilitating rack segmentation, and making it easier to identify the tiles in a particular frame. The contents of the tile can then be categorized using the same OCR method as tiles on the board. However, discussions with the project sponsor have revealed several issues with this approach. Some players rotate the tiles on their racks, meaning that the tile capturing process must accommodate for any orientation, as well as rectifying this orientation to perform OCR detection. As all non-blank tiles contain a numerical score in their bottom-right corner, contour detection could once again be used to ensure that the corresponding centroid is present in the appropriate location. Alternatively, if a custom CNN is used, it could also be trained to identify tiles in any orientation, although this may require a more complex architecture.

The classification of the blank tile poses a potential issue to optical recognition. In its current state, the tile does not contain any distinguishing marks, meaning that the only way it could be identified is through a colour histogram analysis. This issue was discussed with the project sponsor, who was willing to use modified blank tiles, either containing the tournament logo or a question mark character '?', both of which should produce contours which are distinct enough to be recognised optically, especially when using a custom CNN. However, these tiles would still be missing the score in the bottom-right corner meaning that they would be difficult to normalise, and the CNN would therefore need to be trained on all of its possible orientations.

This solution would also need to handle obfuscations of both the board and rack from the players, requiring logic to detect and discard images where insufficient information is visible. In the case of the board, the area of the polygon returned by the feature matching algorithm can be checked to verify that it is within acceptable limits, and only processed if it is. Regardless, the state of the board should only be finalised when a player chooses to end their turn, in which case the board should remain unobstructed, allowing a clear snapshot to be obtained. To handle the player racks, the system can track the number of tiles n that a player should have (typically 7) based on existing game state information, and does not finalise the state of a player rack until an image where n tiles are detected is obtained. Alternatively, a full rack snapshot could be obtained over a series of partial shots, where the system determines which tiles a player has in their rack over time with the appropriate logic to handle repeated letters.

3.2.2 QR Codes

QR codes provide a reliable, machine-readable way to encode tile information, but the only space available to store this data is on the underside of the tile. As such, a new board and rack needs to be designed, containing an optical system capable of detecting this information. This requires the play surface to be built of transparent or semi-transparent material, allowing a camera inside the board to see through the area of play to detect potential QR codes, which could be accomplished relatively easily using different coloured perspex sheets to replicate the patterns on a standard scrabble board. For more rapid prototyping, using a combination of a clear perspex with a printed acetate sheet will be used, allowing different colours and patterns to be experimented with easily should initial versions prove too difficult to see through.

Once an adequate material for the top of the board is selected, a key challenge with this approach is ensuring that a feasible optical system can be designed whilst minimising the thickness of the board is minimised. As per the WESPA requirements outlined in Appendix E, ideally the board should be able to rotate easily, meaning that a large box containing camera equipment would need a custom-built table housing it to allow this behaviour, rendering manufacturing and tournament setup more impractical. After discussions with the project sponsor, a board depth of 10cm was chosen as an upper bound, as anything larger would have a detrimental effect on tournament players. The same upper bound was used for rack depths, and larger racks would prove quite cumbersome to fit on the tables typically used for tournament play. Furthermore, should the new system obstruct a player's view of the number of tiles on the opponent's rack, the solution must provide an alternative way of making this information available to the players to

comply with WESPA's guidelines.

These requirements on the depth of the board and racks essentially provide an upper bound on the working distance (WD) of the system. Recalling the FOV for the board and racks of 322 mm by 322 mm and 265 mm by 20 mm respectively, the focal length required for an optical solution can be derived using Equation 2.3 once a sensor has been selected. For this use-case, the Raspberry Pi High Quality Camera (HCQ) module [92], measuring 6.287 mm by 4.712 mm, was considered, as it is a low-cost sensor which can be fit with custom lenses and can interface easily with an embedded Raspberry Pi CPU that are used extensively in both home and industrial automation applications [93]. For the board, only the vertical FOV needs to be considered, as it is identical to the horizontal FOV but with a shorter sensor length, meaning it will be the bounding constraint in the system, resulting in a focal length $f = 1.46$ mm. This is an incredibly short focal length, and although lenses satisfying this requirement exist [94], they are quite rare and were unavailable for purchase at the time of this report.

The use of multiple camera modules was therefore considered as an alternative to reduce the FOV. Although the Pi only supports a single camera by default, a multi-camera module has been developed, allowing the Pi to multiplex between up to 4 cameras [95]. For the purposes of this initial analysis, only simple arrangements to split up the space keeping the camera orientation fixed were considered, as shown in Figure 3.4, leading to overlaps in tiles on inefficient use of the sensor's dimensions were considered. Should this arrangement prove ineffective in practice, further research will be conducted to determine an arrangement which takes full advantage of the sensor size to reduce the constraints on the focal length optimally. In the 4-camera arrangement, only an 8 by 8 subset of the board needs to be tracked by a single camera, reducing the FOV to 172 mm by 172 mm requiring a focal length $f = 2.74$mm. A 2-camera arrangement focused on a 15 by 8 subset would have an FOV of 322 mm by 172 mm, in which now the horizontal FOV becomes the binding constraint and requiring a focal length $f = 1.95$ mm. A much wider range of lenses exist which meet both of these constraints, and a 1.7 mm lens [96] was selected for prototyping, as the requirements calculated above are only approximations, which as discussed in subsection 2.2.1 are less accurate when dealing with extremely short focal lengths. Ideally, this lens allows both the 4- and 2-camera setup to be tested, as well as enabling the effects of distortion on QR code recognition to be determined experimentally, including after the appropriate corrections [31] are applied. Providing performance is adequate on a 4-camera system, the use of longer focal lenses with lower distortion [97] may prove more desirable if the corrections applied to the 1.7 mm lens are not sufficient in recovering a clear picture.

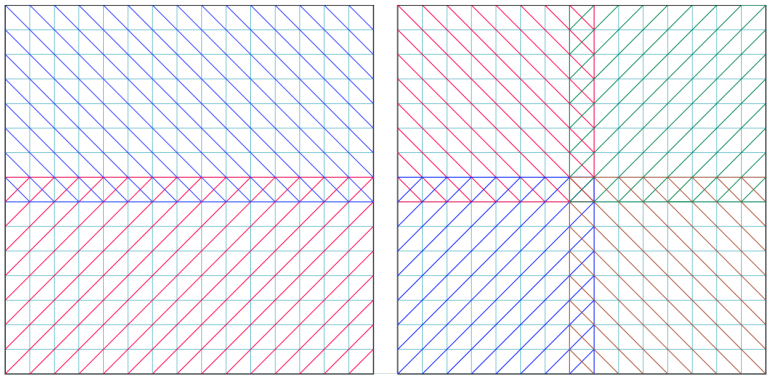


Figure 3.4: The simple 2 and 4 camera arrangements considered

In the case of the racks, assuming the standard dimensions were kept, a focal length $f = 2.37$ mm is required, although given the racks only need to hold 7 tiles, each of which are approximately 19 mm wide, narrower racks of 210 mm would still house these comfortably and allow the focal length to increase up to $f = 3.00$ mm. Discussions with the tournament sponsor confirmed that this would not be an issue for players, meaning that any lens selected for the board should be adequate for rack capture as well.

Provided a reliable optical system can be obtained, the resulting image processing problem

is substantially simpler than in the OCR solution. Although standard techniques will need to be applied to ensure the solution is robust to noise, the enclosed environment provides a much higher degree of control, as LEDs inside the box can help provide constant, effective lighting. As the cameras are inside the board, their orientation relative to the board remains constant, independent of rotations. Therefore, high-contrast etchings can be drawn on the inside of the board to substantially facilitate grid detection, simply using Canny edge detection to isolate the grid lines. A perspective transform can also be applied to the grid contour to help ensure the system is robust to slight variations in sensor placement. Given the camera's fixed position relative to the board, simply cropping the relevant sections of the image provides another, computationally cheaper approach, although this would require more careful hardware assembly to ensure that the camera positioning is kept

Once grid squares are isolated, standard QR code detection and decoding, which has numerous open-source implementations [98] [99], can be applied to identify tiles. These detection algorithms can fail if the resolution of the QR code is too low, and initial tests of the implementations cited above revealed that a minimum of 2 pixels per module is required to detect digital QR code images. For the images captured by an optical system prone to noise, it is likely that a higher resolution will be necessary, although given the 4056 by 3040 pixel resolution of the HCQ module this is unlikely to be an issue. A 2-camera setup would have a pixel density of 12.60 ppmm, resulting in 7.38 pixels per module for version 1 QR Codes, which are the largest considered for this application. As QR codes can be decoded in any orientation, and the libraries above provide capabilities to detect all QR codes in a given frame, the detection of tile in player racks should be extremely simple.

3.2.3 Comparison

Based on the analysis above, both optical solutions present a feasible solution to the problem of capturing game state data. To determine which option would be selected for this project, their performance in various parameters stemming from section 1.1 were compared. Each criterion is discussed in-depth below, and Table 3.2 summarises the conclusions drawn, with factors listed in order of importance.

Detection Accuracy

Although in theory OCR should be able to achieve the target 99.9% accuracy with recognising tiles, this solution faces a number of challenges. Although solutions have been proposed to the blank tile and tile orientation issues, player interference with the rack cameras remains a difficult problem, and could actually make it impossible to identify the tiles a player has drawn in some games. This is an inherent limitation of the system, as instructing players to avoid fiddling with their racks would be highly detrimental to their experience, and may be ineffective as this behaviour would likely be ingrained in their play style. On the other hand, the QR code solution does not suffer from this issue, and its extensive use in part classification for manufacturing [100] proves its extremely high accuracy, likely outperforming a CNN-based classifier. The only real issue it faces in this application is if an optical system of sufficient quality is infeasible for the hard requirements on FOV and working distance. Although such a system is theoretically feasible, the calculations used are approximations, and as such the real performance of this system may differ.

System reliability and robustness to changes in environment

Image preprocessing should render both solutions quite robust to image noise and small variations in camera or board positioning. However, despite the image binarization algorithms considered, lighting could cause problems in the OCR solution, with Scorabble's main developer confirming that shadows and reflective surfaces can substantially degrade the performance of its recognition system. Given that the board surface will be somewhat transparent in the QR code solution, it may also be affected by changes in external lighting, although the use of internal lighting, as well as QR codes error correction capacity when it is partially obstructed should mitigate this. As such, it is considered significantly superior to the OCR solution in this particular area.

Impact on player experience

Given that the OCR solution uses the existing camera setup used for tournament broadcasts, it would have no effect on competitors, which is ideal. Even if the QR code based solution conforms to the depth requirements, the board would still be thicker than those players are used to, which may cause some visual discomfort. The text on the special squares may also have to be removed as these may interfere with QR code recognition, although this may not be an issue, as many online scrabble board do not include this text. If any changes to the board design are necessary, they will be discussed with the project sponsor to ensure that players are not affected.

Another potential issue is that the underside of tiles will now contain a QR code identifying which tile it is. A competitor may hypothetically memorise the QR code patterns corresponding to particular letters, and could be able to determine which tiles their opponent draws if the tile underside is visible to them. Based on previous live-streamed games, this is unlikely to be an issue as players typically place their drawn tiles face-down in front of them, which is not clearly visible to their opponent. Furthermore, as per WESPA tournament rules, players may draw tiles directly onto their rack, which would avoid the problem altogether, although players may not be used to doing this. When exchanging tiles, unwanted tiles must be placed face down on the table before being exchanged, although again these tiles can be placed in front of the player redrawing to avoid giving an opponent any information. However, this was still considered problematic by the sponsor, and so an encoding scheme mapping each letter to multiple QR codes would need to be designed to provide a sufficiently large state-space to render tile memorisation infeasible. Given that there are approximately 280,000 valid scrabble words which require memorisation [101], the QR-code space should be several orders of magnitude larger than this value, which potentially renders M1 QR-codes unsuitable as these can only encode 10^5 different values [68]. Larger symbol versions of M2 and above can store $> 10^8$ possible values, which is more than adequate for this purpose.

Estimated cost

The components required for a QR code solution are as follows:

- 3 Raspberry Pis, one for the board and each rack, costing £25-£40 each depending on the compute power necessary.
- 4-6 High Quality Camera (HCQ) modules costing £40 each, as well as an appropriate lens costing at most £10 each.
- 1 Multi-camera adapter module costing £40 each.

The manufacturing of the board and racks themselves using perspex and acetate should be quite cheap, and would likely cost under £20, leading to total estimated costs of £335-£480 per system.

Although the OCR solution can rely on the current streaming infrastructure, in previous tournaments only one of these was necessary as multiple games could not be broadcast due to the digitization limitations. As such, should this solution be selected additional hardware will need to be purchased to replicate this setup, which according to the tournament broadcast team would be quite expensive and cost over £1000 if the same system, in which all camera feeds are processed by a single computer, is used. Wifi-enabled cameras could be used to stream tournament footage to a cloud computer, which would likely reduce costs, although these would likely be equivalent or more expensive to the QR code approach.

Difficulty of production and setup

Issues with global semiconductor supply chains, largely as a result of the COVID-19 pandemic [102] has led to a shortage of Raspberry Pis available on the market, although supply is expected to return to pre-pandemic levels in Q2 of 2023 [103], meaning that acquiring the necessary components should not be an issue for Alchemist Cup 2024. However, time will need to be spent finding an appropriate manufacturer for the system itself. Although this does not fall strictly under the scope of this project, this is not expected to be a challenge as the construction of the board should be

quite simple, and the expertise of college lab technicians can be relayed to the project sponsor to assist them in finding an appropriate solution. Given that the OCR solution would leverage the existing streaming setup, it would require no additional effort and would therefore be easier than the QR code solution.

Implementation difficulty

This factor needs to be considered primarily to ensure that it is feasible for the project to be completed in an appropriate time-frame. An OCR approach would require more complex preprocessing as well as CNN optimisation, both of which would be non-trivial problems requiring significant development effort. In contrast, the QR code solution should be simpler from a software side, although the optics involved are more complex, and some time would need to be spent experimentally to design an adequate system. Regardless, it is expected that sufficient time is available to complete either solution, although overall the QR code approach should take less time compared to the image processing needed for OCR, meaning that more project time will be available for other components of the system.

Parameter	OCR	QR Code
Detection accuracy	High, potential issue with racks	Extremely high
Reliability	Affected by lighting	Highly robust
Player experience	No effect	Visual discomfort
Estimated cost	>£1000	£335-£480
Setup difficulty	Replicate current setup	Requires manufacturing
Implementation difficulty	Preprocessing and CNN training	Optical system design

Table 3.2: Summarised performance of optical solutions in relevant criterion

Based on this analysis, it was concluded that the QR code solution was better suited for this project, as its superior performance in game state detection and lower costs outweighed the potential negatives from the effects on the player experience. However, this decision hinges on the feasibility of the optical system necessary, which will need to be verified experimentally before additional work is performed on this approach.

3.3 System Design

Although a solution to detecting the game state is the most challenging part of this problem, several other issues remain. Players must be able to provide input to the system to provide the value of blank tiles, as well as to indicate the end of their term, as this information is necessary to broadcast the match but cannot be determined unambiguously from the solutions described above. As such, the use of a tablet, in which a UI would be designed where this information could be provided, was considered necessary. This UI could replace the digital clocks that competitors currently use to end their turns, meaning that no additional effort would be necessary from the players' behalf to capture this information, and when the board detects a blank tile in play, a prompt can be displayed requiring the user to enter this information. Obtaining data from the board also allows the UI to display the number of tiles on each players' rack should the QR code solution obstruct this information, and various other useful features could be included. Scores could be automatically calculated and display, and word challenges could be handled seamlessly; a button could allow the user to select from a list of words that their opponent played in the previous turn and verified against the tournament word list, which for current WESPA tournaments is CSW21 [101].

This companion application would have to be implemented for iOS and Android, and although iOS is typically considered easier from both a development and maintenance perspective [104], the IDE used to write Swift, the language iOS applications are written in, is only available on macOS [105], making it challenging to develop such applications on a Windows computer. Additionally, a typical challenge associated with Android development is the range of different screen resolutions that an application needs to accommodate [104], but for the purposes of Alchemist Cup 2024 a specific tablet model can be selected, mitigating this issue as the application would only need to consider a single resolution. Android applications can also easily be exported to a package which

can be installed on the target device. This is more difficult for iOS applications, which usually need to be published on the App Store, requiring a yearly 99 USD developer fee. Android development was therefore chosen, and discussions will be held with the project sponsor to select the desired tablet this application will target for the tournament. Creating a web-app was another approach that was considered, which has the advantage of being platform-agnostic, but requires an active internet connection to run, unlike native applications which can work offline. Although this should not be an issue in most cases, it does render the system less robust to potential network outages, as it is critical for the time-keeping abilities of the companion application to function in all scenarios to protect the integrity of the match.

One limitation of the approach considered above is that by registering the end of turns on a separate device, a data race condition is introduced between the game state snapshots and the end-of-turn signal. Should the system managing the state of play receive this signal before all game data has been received, it may introduce a mismatch between the system state and the actual state of play. Unfortunately, given that a distributed system is necessary to capture both the board and racks, this issue cannot be solved by simply embedding the companion application into the board itself, which could be done through the Raspberry Pi Touchscreen module [106].

Once all necessary information has been captured, only two core challenges remain: combining all game data to determine a full picture of the state of play, and pairing all the devices together when setting up a match. One option to solve this issue is through a client-server architecture, where a central server could listen for connections from sensors and the companion application, allocate them based on requests from the companion application, and act as a single source of truth for the system, responsible for resolving deltas between the racks and board, calculating scores, resolving challenges and encoding the general game logic, and passing this data up to the Woogles API. An alternative, server-less approach involved pairing sensors to the companion application via Bluetooth, which could then communicate with each other via sockets [107]. Given the computationally intensive digital image processing performed on the embedded Pis, the companion application would be the most suitable candidate to inherit the responsibilities of the server. Although this has the advantage of removing the server as a central point of failure in the system, this solution would be significantly more challenging to monitor in real-time, as the game data for each match would be available on a different device. Having a single source of truth for all match data allows much greater flexibility in designing error correction methods, as well as unifying all the data in the tournament. Furthermore, cloud providers have strong service level agreements (SLAs) on the availability of compute instances, often guaranteeing 99.99% uptime [108] [109], which the sponsor deemed sufficiently reliable.

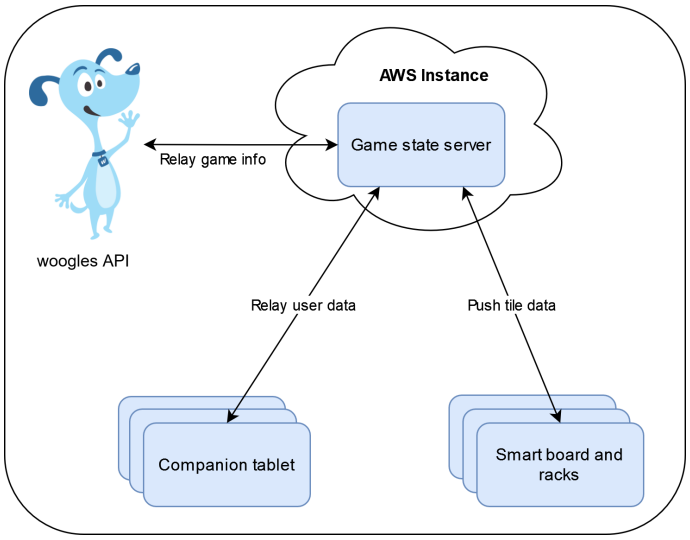


Figure 3.5: High level system architecture

The design of the server architecture will be discussed in greater depth in chapter 2, but a top-level overview of the system is outlined in Figure 3.5. Given that the AWS was already used to manage the domain names associated with the event, it was a natural choice of cloud provider

as it could simplify dynamically mapping a subdomain to a server IP. Using a cloud provider is much cheaper than purchasing hardware, particularly in the prototyping phase, when the exact specifications required by the server are unknown, could even render the system more reliable in production through elastic load balancing [110] and auto-scaling [111] as sufficient resources are allocated to process match data, and that this traffic is evenly spread among these resources. These features can also render the service more resilient to outages by ensuring the server instances are spread across availability zones, mitigating the effects of a network or power outage in a single data centre [112].

3.3.1 Deliverables

Based on the system design outlined above, the core technical deliverables required for this project are summarised below:

- Hardware prototype of the smart board and racks with accompanying embedded software.
- Mobile application for tablet responsible for timekeeping and end-of-turn and blank tile capture.
- Server component responsible for resolving information from hardware and tablet components into a single, unified game-state, as well as publishing these changes to the Woogles API.
- Clear documentation outlining how to reproduce this setup for Alchemist Cup 2024.

3.3.2 Technology Stack

Although this project requires multiple different components, a similar technology stack should be used whenever possible to simplify development effort and facilitate future maintainability. The use of Python was considered ideal as a primary language for the development of this system; its ease of use enables rapid development, allowing for quick prototyping and testing, and has extensive community support, with access to many libraries for image processing and computer vision, as well as networking frameworks. Although it is not ideal for high performance, which may be required for embedded applications, these libraries are typically written in highly performant languages with python bindings, meaning that the bulk of the computationally intensive workload is completed efficiently. Furthermore, there are very few alternatives for interfacing with cameras on a Raspberry Pi, as, although it is possible to use the C++ based libcamera library, the Raspberry Pi foundation only provide a Python API which is tuned specifically to address the capabilities of the device's built-in camera and imaging hardware [113].

OpenCV is a powerful, open-source, widely used computer vision library [114], providing implementations for all the techniques discussed in chapter 2. It is written in C++, providing high levels of performance which are essential for an embedded application, and has bindings in many languages, with benchmarks on a Raspberry Pi 3B+ yielding almost identical performance [115], providing minimal native python code is used in the processing. Unfortunately, it is only capable of detecting standard QR codes, meaning that a different library would be required to decode micro-QR codes. Currently, BoofCV, a library for real-time computer vision written in Java, is the only open-source library which supports these codes [116]. Both of these libraries have bindings in Python, and are evaluated further in subsection 4.1.4.

Python has packages for all the communication protocols explored in subsection 2.3.2, meaning that the availability of library support was a non-factor when determining the optimal network structure. The selection of packages used for networking is discussed in section 4.2, with the rationale behind the protocols chosen explored in subsection 3.3.3.

An added benefit of Python is that it is the most popular language for machine learning [117], meaning that should a switch to OCR become necessary, any existing preprocessing done with OpenCV can be kept, and powerful Python machine learning libraries for convolutional neural networks such as Keras [118] can be used. Tesseract, an open-source OCR engine which can detect any Unicode character [89] also has a python wrapper allowing it to be used in conjunction with OpenCV image objects, allowing for this option to be tested easily as well.

As mentioned in chapter 2, Jetpack Compose will be the toolkit used for the companion application development, which is written in Kotlin, a cross-platform, statically typed, general-purpose high-level programming language recommended by Google for Android application development [119]. Whilst it is possible to develop mobile applications in Python, with transpilers such as the one provided with BeeWare, which converts CPython Bytecode to Java-compatible bytecode [120], sacrificing the extensive developer support provided by Android for Kotlin would render development substantially more difficult and harm maintainability, justifying the cost of including another language in the technology stack.

3.3.3 Server Architecture

The match data server has a number of responsibilities, and as such it is important that these functions are compartmentalised appropriately to render the code more modular, as well as facilitate testing. The server must perform the following high-level tasks:

- Assign sensors to an instance of the companion application in preparation for a match.
- Receive game data from the sensors, as well as synchronise the state of the sensors with the current state of the match.
- Process requests from the companion application, including end-turn events which signify an update to the game state, as well as challenges and blank tile information.
- Codify the game logic to correctly compute scores and challengable words.
- Present match data via the Woogles API.

Each of these functions can be encapsulated using the appropriate objects following the OOP paradigm, with a bridge component acting as an interface between the server and Woogles, as well as a package for handling the game logic. Whilst Woogles already codifies the Scrabble rules, its API for casting matches is rather limited, making it quite difficult to obtain scores or challengable words from it. Furthermore, ensuring these operations are performed on the match data server removes a critical dependency on Woogles for the system to work properly from the perspective of the players, as the Woogles API would only be used for presenting the matches, rendering the system more robust. Although there exist python packages to handle the game logic [121], these often focus on determining optimal moves or playing matches, making them unsuitable in this application. As such, a custom Scrabble package would be developed for this purpose.

The workflow of assigning sensors and tablets to matches was a more challenging problem to overcome. Given that the companion application is the only device in the system with a graphical user interface (GUI), it is logical for setup requests to be initialised from the tablet, where the player names for the match could also be easily provided. This could then send a request to the server, which would be maintaining a pool of available sensors which could be used for the game, select a board and two racks and group these all together with a unique match ID, passing this ID back to the tablet. Once assigned to, the sensors could flash a built-in LED for several seconds to indicate to the tournament organisers which devices have been paired, using different colours to distinguish between the racks for the first and second player. Once a pairing has been established, the server can then prioritise re-using the same board and racks together for subsequent games provided all components remain connected, which would minimise the intraday operational overhead. An alternative approach considered was to set up matches directly on the server via a command-line tool, which would now also maintain a pool of available tablets. This method was deemed worse for a number of reasons, as not only does this method require a higher level of technical literacy, making it more difficult for the tournament sponsor, but organisers would also need to identify which tablet was selected by the server.

Communicating with Companion Application

With the added responsibility of setting up games, the communication between the companion application and the server was well suited for a request-response model, where events on the companion app trigger behaviours on the server side. Stateful parameters, such as the match ID,

can be provided by the tablet, allowing the connection with the server to remain stateless, making it more resilient to network issues. Based on the properties of the various options analysed in subsection 2.3.2, HTTP is ideal for this purpose, as the simplicity and strong reliability guarantees it provides are well worth its extra overhead over other protocols, particularly as the companion application is not responsible for particularly intensive computation and messages will be relatively infrequent. By the same reasoning, a text-based serialization format such as JSON fits best, enabling more rapid troubleshooting of errors. Given the importance of informing the players of potential errors in the system, server responses contain either a body with the expected data, or an error message to be displayed to the user. A summary of the interactions between the table and server can be found in Table 3.3.

Request	Parameters	Response	Logic
Setup match	Player names	Match ID	• Assigns sensors to tablet • Initialises match on Woogles API
End turn	Match ID Turn number Player time	Score # of blanks `isMatchOver`	• Verifies turn number is synchronised with internal state • Determines move from sensor deltas • Applies move to internal state • Informs board sensor of accepted state • Publishes move to Woogles
Send blanks	Match ID Turn number Blank values	None	• Ensures request corresponds to previous move containing blanks • Updates internal state with blank tile values • Re-publishes move to Woogles
Get challenge words	Match ID Turn number	Challenge words	• Ensures request corresponds to previous move and blank values have been provided • Calculates words formed using internal state
Challenge	Match ID Turn number Words	Outcome Penalty	• Verifies validity of words using CSW21 dictionary • Determines penalty depending on outcome of challenge • Publishes event to Woogles

Table 3.3: Communication between companion application and server

Using an HTTP server has the added flexibility of easily exposing match data to other applications. For example, developing a monitoring tool could be helpful for the tournament organisers to have an overview of the state of different matches, notifying them of errors and potentially allowing them to resolve these on the fly. The API above could be extended to support such a tool with relatively little development effort

Communicating with Sensors

Given that the companion application drives changes in the system, ensuring two-way communication between the server and sensors is essential so that information is able to flow through the system efficiently. There are a number of protocols which were discussed in subsection 2.3.2

support this type of communication, namely RPC via UDP or TCP sockets and MQTT, which were compared for suitability. A priority of this system is to accurately detect sensor errors to ensure that tournament organisers and players are aware that the internal match state is no longer reliable. To accomplish this, when first booting up, a sensor will attempt to register itself with the server, providing a unique identifier which remains constant between all sessions. The server can track if this identifier is already allocated to a particular match, in which case it can reply with a match ID, allowing the sensor to immediately continue sending data, or add it to a pool of available sensors.

To assert the reliability of the connection, periodic heartbeats will between the server and sensor to ensure the connection remains active, with either end severing the connection once a timeout is reached. It is important to ensure that other messages from the server, such as match allocation notifications or board delta confirmations, are received reliably by the sensor, as dropping these messages could introduce errors into the system. However, it is less essential to ensure messages from the sensors containing the state of the match are always received, as these are sent relatively often. The target time between sensor updates was estimated to be in the 200-600 ms range, as this is frequent enough to provide real-time match data, whilst avoiding sending unnecessary updates to the server, as it is unlikely for the state of the board to change significantly in this time window.

All three protocols considered can support the functionality described above, although with varying degrees of success. The reliability provided by TCP-based approaches is ideal in ensuring that nodes in the system are aware if they have been disconnected, although the use of TCP sockets allows the connection to be closed directly, which is both simpler and more efficient than passing a disconnection message to the client via a broker. Furthermore, once a sensor is assigned to a match it receives information only related to its particular match, meaning MQTT's many-to-many IoT communication features are not particularly relevant in this case, and communications via the broker only add latency, as well as another potential point of failure. Given the properties of UDP, there is no guarantee that a disconnection message would be received by a client, meaning in the worst case would need to wait for application-level timeouts such as the lack of heartbeat message. This may require shorter timeouts for this mechanism in the UDP case, which is undesirable for a number of reasons, as it would require more frequent heartbeats to be sent which consumes both network and compute resources unnecessarily, or increase the probability of a false alarm.

UDP's main advantage is when publishing sensor data, as TCP's operating-system level reliability guarantees may cause it to ignore a more recent delta due to the packet belonging to an older delta being lost. This causes the average latency of communication for TCP-based protocols to increase under poor network conditions, with higher rates of packet loss as the operating system waits for packets to be re-transmitted. However, as discussed in subsection 2.3.2, this latency difference is not particularly significant in the context of this project, where a delay of up to 1 second is acceptable. Furthermore, using application-level logic to ensure reliability in UDP-based protocols is typically less efficient than TCP implementations, which lead to TCP-based protocols having smaller latency in cases where reliable delivery is required [77]. Initial testing of a TCP-based approach indicated that the size of serialized protocol messages to be in the 100-300 byte range, meaning that an average delay of 30 ms relative to a UDP-based approach could be expected [77]. The analysis of the protocols above is summarised in Table 3.4, with properties listed in order of importance.

Protocol	Advantages	Disadvantages
UDP Socket	+ Lowest overhead, smallest latency + Control over timeout characteristics	– Complex application layer logic to handle dropped packets – Slower for reliable delivery – Most challenging to implement and debug

TCP Socket	+ Reliable and ordered delivery + Low overhead + Maintains state of connection	– Ordered delivery may cause delays if single packet is lost – Difficult to implement
MQTT	+ Configurable reliability + Simple implementation	– Requires additional logic to handle disconnections – Broker represents additional point of failure – Highest overhead

Table 3.4: Comparison of various 2-way communication protocols

Ultimately, the use of RPC over a TCP socket was deemed most suitable for sensor communication, as the guarantee of reliable communication greatly simplifies the application-level logic at the cost of a minimal latency penalty. The logic required to handle disconnections in the case of MQTT already partially mitigates the additional development effort of TCP sockets, and was further justified by removing the need for an additional broker component, as well as eliminating latency overhead, although this difference is likely to be relatively insignificant. Using a mixture of UDP for game state data, and TCP for other messages may have been a more optimal choice from a performance point of view, as it combines the best properties of both approaches, but complicated both the design and implementation of the system to an undesirable extent, particularly as these performance gains would be minimal.

Once the communication between the different nodes of the system had been established, the architecture of the match data server could be established, as shown in Figure 3.6. Player interaction via the tablet, which communicates via the HTTP drive events in the rest of the server, causing deltas accumulated from the sensors to be applied to the game state and updates propagated via the Woogles bridge.

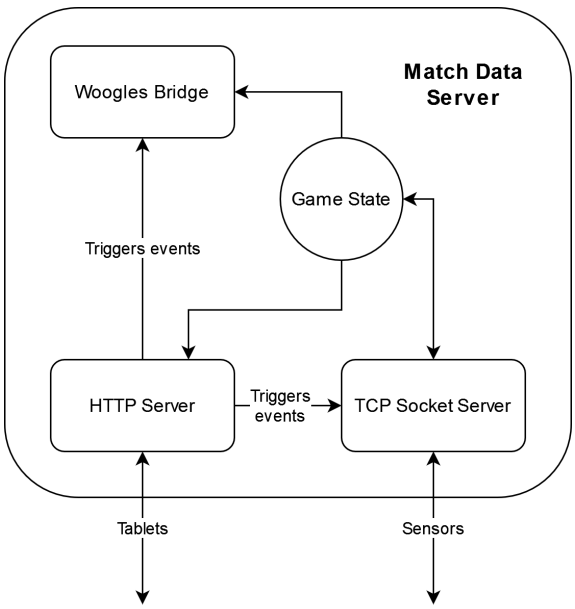


Figure 3.6: Server architecture

The mechanisms through which the different server components interact with one another is discussed in section 4.2.

Chapter 4

Implementation

This chapter explores the decisions made in the implementation of the system designed in chapter 3, and why these choices were superior to others in achieving the requirements outlined in section 1.1, as well as providing a detailed overview of how each part of the system functions, and how implementation challenges for each of these subsystems were overcome.

4.1 Game State Capture

The following section outlines the implementation of the QR code based approach outlined in subsection 3.2.2. Board capture was first implemented for a quarter of the game board, as a working proof of concept with a single camera was desired before purchasing hardware to support the entire board. Adapting a quarter board implementation to handle all four quadrants would be relatively simple; both the Raspberry Pi 3B+ and 4B System-on-Chip (SoC) contain quad-core processors [122], each supporting 1 hardware thread, meaning multiprocessing [123] can be exploited to handle board state detection for each quadrant in parallel once the image is taken. Note that separate processes, rather than threads, are required for a Python implementation to achieve true parallelism to circumvent Python's Global Interpreter Lock (GIL) [124], which only allows a single python thread to execute at a time.

Rack capture would be performed similarly to capture for a quarter board, following the same configuration and preprocessing steps, but would use a different detection algorithm, as the tile positions are no longer restricted to grid squares, and the relative position of the tiles does not need to be captured.

Surprisingly, despite standardisation in ISO/IEC 23941:2022 [125], minimal open-source support exists for micro-QR code detection, with BoofCV as the only mature library providing this functionality [116] at the time of writing. Although feature requests to provide support for this capability were opened with other libraries early in the project timeline [126], these were not completed prior to project submission, and as such, version 1 QR codes were selected for initial feasibility testing to provide greater flexibility in the choice of detection library. However, the use of version M2-4 micro-QR codes was briefly tested in subsection 4.1.4, and is discussed further in subsection 6.1.2.

Due to difficulties in hardware manufacturing, the physical board and rack prototypes were unable to be completed prior to the submission of the report, although this work is expected to be completed in time for the demonstration. Although this means that the parameters used for game state detection are not available, as these will differ between the initial testing setup and final prototypes, the feasibility of the image processing pipeline discussed below is asserted, and will be replicated on the hardware once completed.

4.1.1 Camera Configuration

As mentioned in subsection 3.3.2, the Picamera2 library [113] was used to interface with the camera system in-code, where a `Picamera2()` object can be used to configure the camera module and capture frames. Ensuring the camera is configured correctly is essential to ensuring the appropriate performance, with some notable parameterisation including:

`queue` : Allows the camera object to store the last frame received, returning it immediately when a capture request is received rather than remaining idle until the next frame arrives. Although this means that the frame processed can be delayed by up to one frame period, this delay is negligible in this application, representing ~ 17 ms at 60 frames per second (fps).

`size` : Represents the width and height in pixels of the image received, which in the case of the HCQ module can be up to 4056 by 3040. However, larger image sizes not only lead to slower frame-rates due to the need of copying more data from the camera image buffer, but slower image processing, making it undesirable in this application. `Picamera2()` can request for down-sampled images, which is done using a dedicated Image Signal Processor on the Pi, making it much more efficient than performing these manipulations in software, enabling faster processing and reducing sensor noise. It is worth noting that due to hardware restrictions, the image dimensions cannot be selected arbitrarily, as non-standard sizes need to be copied more frequently [113], meaning that a square resolution matching the board quadrant could not be used directly. For the prototype implementation, a 1280 by 720 resolution was selected, which after cropping would result in a square quadrant of approximately 700 pixels wide – 88 pixels for each square, resulting in approximately 4.19 pixels per module for version 1 QR codes, double the minimum threshold obtained in subsection 3.2.2.

`format` : For reasons explored in section 2.2, greyscale images are optimal for image processing in this application. Unfortunately, `Picamera2` does not support taking greyscale images directly, with the HQC module capturing coloured images which are digitally encoded using the format specified. OpenCV assumes coloured images are encoded using RGB, an additive colour model in which the red, green, and blue primary colours of light are added together to produce the image [127], but with the byte order of the three channels in reverse as $[B, G, R]$ [128]. Initially, this colour format was used to capture images, which were then passed to OpenCV and converted to greyscale for further processing, which was measured to operate at an average of 44 fps using the parameters specified above. This was slower than desired, as assuming no multiplexer overhead implied that almost 100 ms would be required to simply capture a full image of the board, without any additional processing.

To address this issue, YUV, an alternative colour model which tracks a luminance component (Y), and two chrominance components for blue (U) and red (V) projections [129] was considered. This format allows the greyscale image to be extracted by simply cropping the array to select the Y components, which can then be subsequently processed by OpenCV directly, avoiding the need to perform slow software conversions on the image. `Picamera2()` supports a YUV 4:2:0 format, meaning that the chrominance projections are down-sampled by a factor of 2, requiring only 12 bits per pixel instead of the standard 24 for RGB, meaning that not only is the greyscale conversion performed using the Pi's image processing hardware, but the amount of data copied from the image buffer is reduced dramatically, improving operation to 105 fps, a 239% performance improvement.

`Picamera2()` also expose runtime controls, enabling attributes such as exposure time and analogue gain to be updated programmatically whilst the camera is running, which could potentially allow the board capture software to dynamically adjust itself based on its surrounding lighting, which it could estimate through the average brightness of each pixel. The exact parameters for such an algorithm could not be derived due to the lack of hardware prototype, nor could the system's adaptability to different lighting conditions by default, but these controls could help enhance the robustness of detection in the deployed system.

4.1.2 Fisheye Rectification

Lenses with incredibly short focal lengths, such as the one selected in subsection 3.2.2, produce fish-eye distortion, as shown in Figure 4.1, which require correction before subsequent processing. A collection of calibration images using a standard 9 by 6 checkerboard was captured from various angles, which were then used to determine the parameters used by OpenCV to rectify the distortion [130]. It is important to note that although these parameters are intrinsic to each lens, meaning that re-calibration will be necessary if other lenses are tested, as well as alternate aspect ratios [131]. Using identical parameters for different lenses of the same model was tested, and appeared to effectively undistort the image feed, meaning that calibration does not need to be performed to account for minor deviations in the lenses from manufacturing.

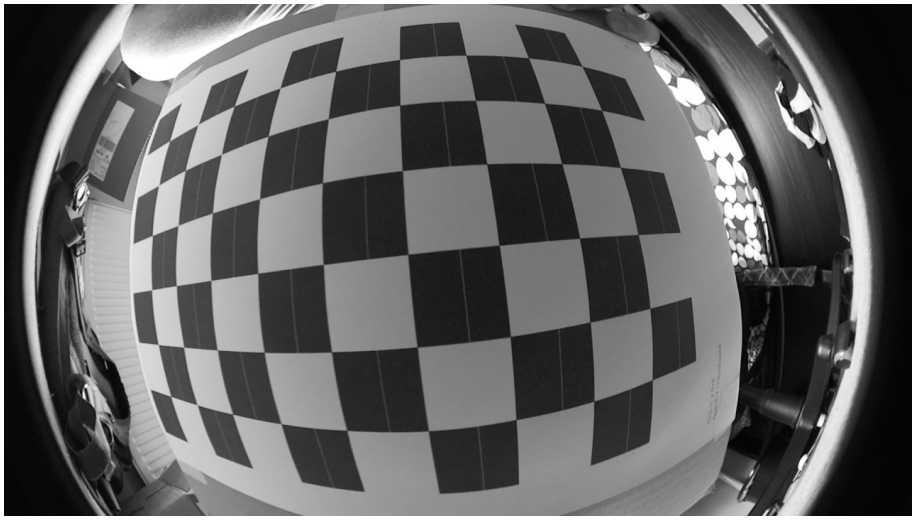


Figure 4.1: Raw image of 9 by 6 checkerboard from camera configuration

Once the camera parameters are obtained, a rectifying map must be computed, which defines how pixels from the raw image will be transformed to produce the resulting undistorted image. Although there exist open-source tools which automate this process [132], these were found to be ineffective at reliably and accurately removing distortion. Instead, various OpenCV functions were used to accomplish this, with additional parameters which needed to be selected:

`balance` specifies the proportion of pixels from the original image should be kept, cropping out the "best part" of the image when set to 0 and preserving all pixels when set to 1, leading to overstretched corners and black areas.

`dim2` is the dimension of the box that will be kept after undistorting the image, and must match the aspect ratio of the original image. This parameter works in tandem with `balance` to determine the resulting image's appearance

`dim3` is the dimension of the final array that OpenCV places the undistorted image in. This is typically identical to `dim2`, but can be adjusted to crop away parts of the image, although unfortunately this cropping is done from the top-left corner of the image, rendering it unusable in this scenario.

The impact of these parameters is further illustrated in Figure 4.2. In order to optimally detect the board segment, it was important to ensure that the maximal amount of vertical pixels in the centre of the image were preserved, as these would enable the board to remain closer to the sensor. Setting `dim2` to match the original dimensions of the image and `balance` to 0.3 was found to be ideal to accomplish this, and unnecessary pixels could then be cropped away leaving only the rectified board suitable for QR-code detection. The derivation of the rectification map is computationally intensive, and as such these values are computed from the parameters specified on startup and re-used for all images processed by the Pi. Figure 4.3 illustrates the application of the mapping derived on images obtained from the HQC module on an initial prototype, without the exact board pixel dimensions for cropping.

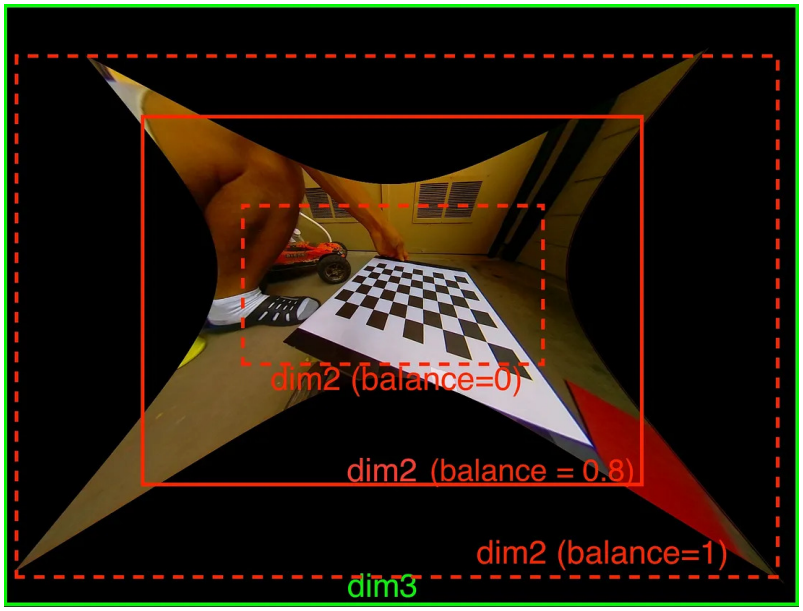


Figure 4.2: Effects of `balance`, `dim2` and `dim3` on undistorted image [131]

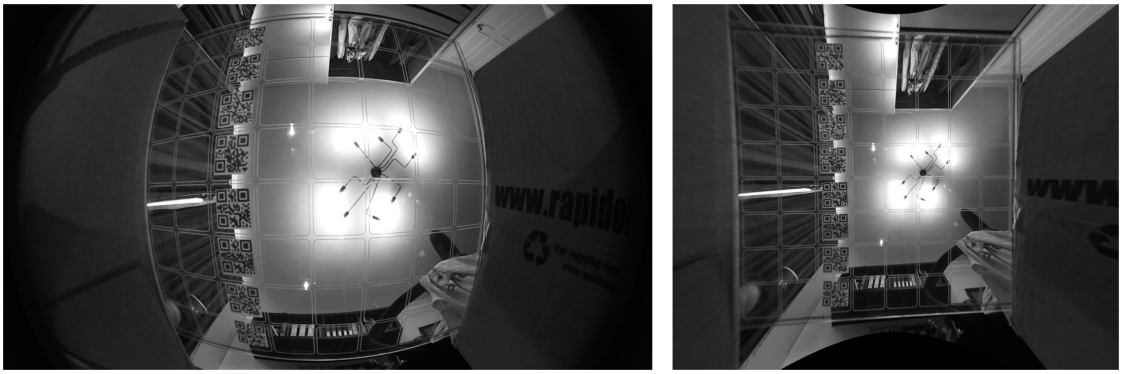


Figure 4.3: Application of undistortion algorithm to raw image on test setup.

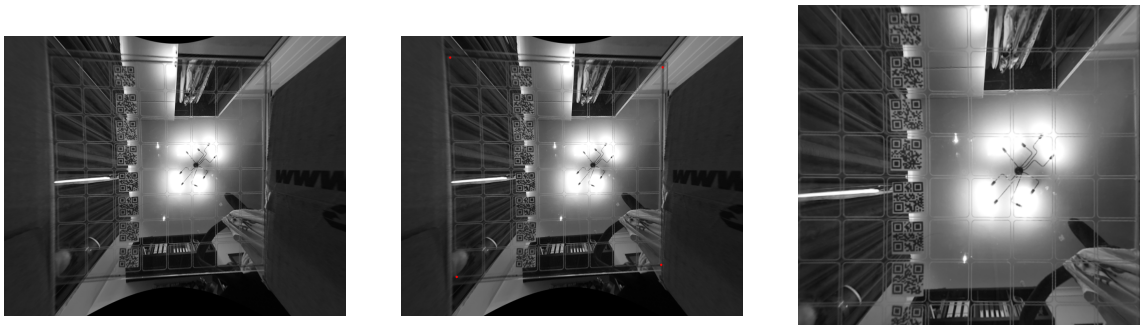
One potential issue identified by this stage of implementation for the larger-scale production of boards for the tournament involved the positioning of the lens on the HCQ. The M12 mount on the module allows the position of the lens relative to the sensor to be adjusted, and must be set at a specific height to obtain a clear image with the correct working distance. This may require manual adjustment for each camera used in the final design, complicating the production process, and will need to be fixed in place with the appropriate adhesive to ensure this position remains fixed.

4.1.3 Board Detection Algorithm

Given a cropped isolated board image from the fisheye rectification stage, a simple algorithm to obtain the scrabble board state was defined:

1. De-noise and binarize image
2. Segment the board image evenly into an 8 by 8 array containing the corresponding image sub-arrays.
3. Apply QR code detection to each sub-array.
4. If detected, decode QR code value to corresponding letter.

To test the board detection algorithm on a simple prototype, some further pre-processing needed to be performed to ensure that an image only containing the board was used, emulating the cropping which would be used on the final version. This was accomplished using techniques described in section 2.2, applying OpenCV implementations of contour detection and perspective transform to isolate the board, which is shown in Figure 4.4.



(a) Undistorted board (b) Board corner detection (c) Isolated board

Figure 4.4: Board isolation process for prototype testing

The final results of the board detection algorithm on the isolated image are shown in Figure 4.5, with the green bounding boxes denoting the locations of the QR codes detected, and the red letter corresponding to the decoded character value of the tile. The black border separating each board square is separated purely for visualisation purposes. The detection of the final letter “e” appears to have failed, most likely due to part of the QR code being obscured due to misplacement on the board. This should not be an issue in the final design, where board squares will be delimited by a physical rim following the WESPA guidelines, ensuring that tiles are always positioned accurately. Regardless, the initial performance of an untuned detection algorithm on the prototype confirmed that a QR-based approach was feasible.

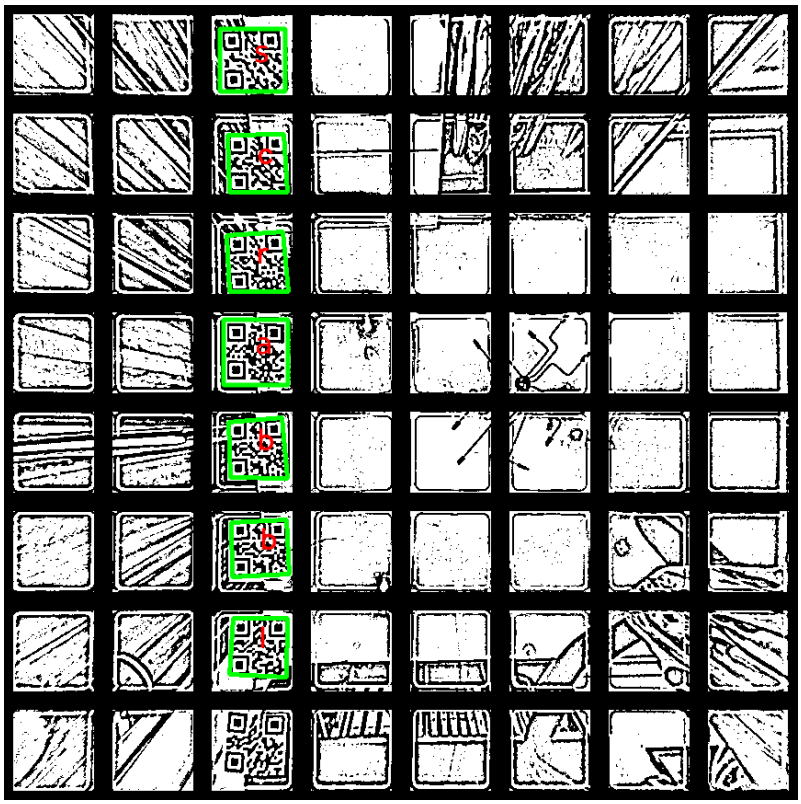


Figure 4.5: Result of board detection algorithm for prototype testing on the played word "scrabble"

Once the state of the board was encoded as an 8 by 8 array of values, differences between the measured state and previously accepted state can trivially be computed, generating deltas containing the new tiles and positions affected which can be published to the server. However, the overall performance of the image processing pipeline on the Pi was incredibly slow, averaging approximately 12 seconds to process a single frame, with the bulk of the time spent on QR-code detection. This was unacceptable, violating the maximal 1-second delay between board state and broadcast, as well as failing to meet the update target derived in subsection 3.3.3.

A simple workaround considered was to instead upload the images to the cloud, performing the image processing pipeline on the central server, or another dedicated component, effectively moving this compute-intensive operation to more performant hardware. However, this is undesirable for a number of reasons, most notably of which is the impact on the network bandwidth. The size of the image after post-processing is roughly 500kB, which is roughly 5 orders of magnitude larger than deltas which would be sent if image processing was done on-chip, including at most 7 tile position pairs taking up 3 bytes each. Given that these updates are sent multiple times per second, by 3 sensors per match for 25 simultaneous games, this dramatic increase in delta size could cause network congestion issues, degrading the performance of the overall system in production.

Another consideration was how such a change would degrade the architecture design of the overall system, as splitting up the image processing across disjoint components would render the game state capture pipeline more brittle and difficult to update, as separate codebases would need to be synchronized between one another when performing changes to the pipeline. As such, optimising the performance of image processing such that it is capable of running on the Raspberry Pi was prioritised.

4.1.4 Performance Optimisation

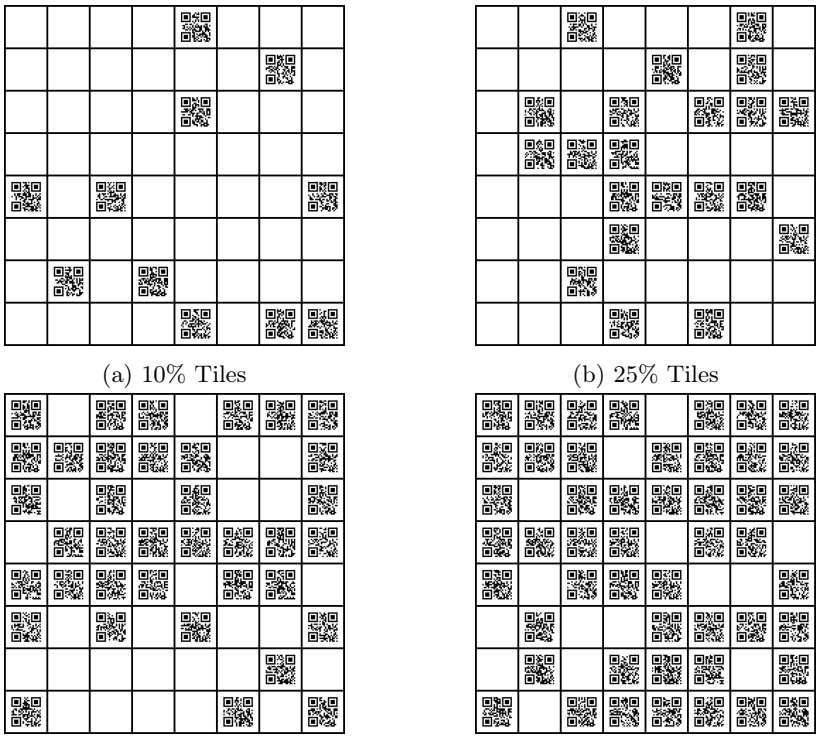
Performance concerns could be partially mitigated through the use of better hardware, as due to supply chain issues, testing was performed on a Raspberry Pi 3B+ rather than the Raspberry Pi 4B which would be used for the tournament. The more recent model provides a number of improvements, performing approximately 3 times faster than its predecessor [133], as well as boasting larger cache sizes and RAM [134]. However, further optimisation was necessary to ensure that sensor updates could be produced frequently enough.

The use of tools which provide performance improvements, such as PyPy [135] which uses a JIT compiler to provide performance improvements over the standard CPython interpreter, or cppyy [136] which provides automatic generation of C++ Python bindings allowing for order of magnitude speed-ups [137], were considered. However, given that 98% of the execution time took place in the board state detection function, corresponding to item 2, item 3 and item 4 of the board detection algorithm described above, which rely on the efficient C / C++ implementations of NumPy and OpenCV, these options would provide minimal benefit, and in some cases can even lead to slower performance [138]. As discussed in subsection 3.3.2, the use of these libraries in Python has minimal performance overhead over pure C++, incurring a worst case penalty of 4% [139]. As such, algorithmic changes to these sections of code were subsequently explored to address the performance problems.

In order to systematically test changes, a test bench was designed to randomly construct board states matching the dimensions of the preprocessed images, including random QR codes encoding the values $r \in [0, 9999999]$ which were generated using Segno [140], an open-source QR and micro-QR code generation library. The generated state was also parameterised by the fraction of tiles present, allowing the performance of the algorithm across the full duration of a match to be modelled, with examples of the board states shown in Figure 4.6. It is worth noting that even in cases where no tiles are present, substantial processing time is expected, as the QR code detection algorithms must still examine the image to confirm the absence of tiles. As the number of tiles increase, execution time is expected to scale linearly, as these algorithms must spend additional time decoding the QR codes once they have been identified.

Both the time taken by the detection algorithm to fully process the board state, and the results obtained were measured, with the latter compared with the reference output generated

alongside the random board state to verify that the algorithm retained high detection accuracy, which was computed as the fraction of correctly identified tiles in the board. Given that "perfect" QR codes were used in the benchmark, unaffected by any visual defects or noise, a sub-100% detection accuracy would make it unlikely that a sufficiently high reliability could be obtained when operating on real images. This lack of visual noise was a notable limitation of the test bench, particularly for the empty tiles containing background objects, as can be observed in Figure 4.5, but is addressed when assessing the accuracy of the detection system in subsection 5.1.1. However, the rapid and robust testing on a wide range of inputs enabled by the test bench made it significantly more practical than manual images for benchmarking purposes.



(a) 10% Tiles

(b) 25% Tiles

(c) 50% Tiles

(d) 75% Tiles

Figure 4.6: Random board states with range of tiles present.

The benchmark was run on both WSL2 running Ubuntu 22.04, configured with 4 GB of RAM on an AMD Ryzen 9 5900HX, as well as a Raspberry Pi 3B+ running Raspbian Bullseye, although more emphasis was placed on the latter's results, with the former serving as a "sanity checker" for the results obtained as more iterations could be run in a shorter time-frame. Plots of the results in the following sections all indicate the 95% confidence interval for each data point, and tend to use floating axes to better illustrate the relative difference between implementations.

Board Segmentation

The initial implementation of the detection algorithm, shown in Listing 4.1, uses a nested native python loop to iterate through the segmented board image. As noted in subsection 3.3.2, these loops should be avoided to take full advantage of the more efficient C++ library implementations, minimising the performance penalty incurred as a result of Python's slower runtime.

A potential improvement considered was to reshape the 2-dimensional `board_img` into a 4-dimensional tensor, containing an 8 by 8 grid of evenly sized sub-arrays, using direct matrix manipulation rather than splitting the array and storing the results in a Python list, removing the need for iteration in the segmentation process. Assuming that the dimensions of the image are square, and perfectly divisible by 8, the approach in Listing 4.2 shows how this could be accomplished. This ensured that the segmented board was available as a NumPy array with minimal copying, which could enable subsequent faster processing by allowing the QR code detection to be performed in a vectorised fashion. Note that in this case, it is easier to reshape the `board_img` into

```python
# Segments board image into squares
squares = []
for row in np.vsplit(board_img, 8):
    squares.append(np.hsplit(row, 8)

# Applies QR code detection to squares
for i, row in enumerate(squares):
    for j, tile in enumerate(row):
        value, _, _ = qcd.detectAndDecode(tile)
        if value:
            board[i, j] = value # Update representation of board state
```

Listing 4.1: Initial iterative detection algorithm implementation

```python
n = board_img.shape[0]
subarray_size = n // 8
squares = board_img
    # Segment image following in-memory data layout of original array
    .reshape((8, subarray_size, 8, subarray_size))
    # Re-arranges order of axes
    .transpose((0, 2, 1, 3))
    # Arrange outermost dimensions to match dimension of board squares
    .reshape((8, 8, subarray_size, subarray_size))
```

Listing 4.2: Board segmentation using matrix manipulation

a 3-dimensional tensor containing all 64 squares in order, then subsequently reshaping the result into an 8 by 8 grid once the vectorised operation has been applied, rather than storing the array as a 4-dimensional tensor which would require vectorisation across two dimensions rather than one. The updated implementation, now only interacting with the matrix via NumPy functions, is shown in Listing 4.3.

```python
# Segments board image
subarray_size = board_img.shape[0] // 8
squares = board_img
    .reshape((8, subarray_size, 8, subarray_size))
    .transpose((0, 2, 1, 3))
    .reshape((64, subarray_size, subarray_size)) # Now 3-dimensional

# Applies QR code detection to squares
def decode_qr(square):
    value, _, _ = qcd.detectAndDecode(square)
    return np.int16(value) if value else np.int16(-1)

board = np.vectorize(decode_qr, signature='(n,n)->()')(squares)
    .reshape(8,8) # Update dimensions of board
```

Listing 4.3: Vectorised board detection implementation

Unfortunately, this is not as performant as typical vectorisation speed-ups in NumPy, which at the time of writing can only be obtained through the application of functions which operate over a 1-dimensional slice of an array, rather than a 2-dimensional one, which are implemented using a simple for-loop, making them essentially equivalent to Python iteration [141]. This was confirmed experimentally, with the two implementations performing almost identically, as shown in Figure 4.7, matching the $O(n)$ performance hypothesized when designing the test bench. The vectorised implementation's improved performance for lower tile fractions are likely due to noise, noting that the 95% confidence intervals for these data points are significantly wider than other points on the graph, and had much closer performance when tested in WSL2. As such, the iterative approach was retained, as its simplicity and ability to operate on arrays of any dimension make it superior to the vectorised approach. However, should NumPy provide access to high-performant multidimensional vectorisation, the method explored above may become superior.

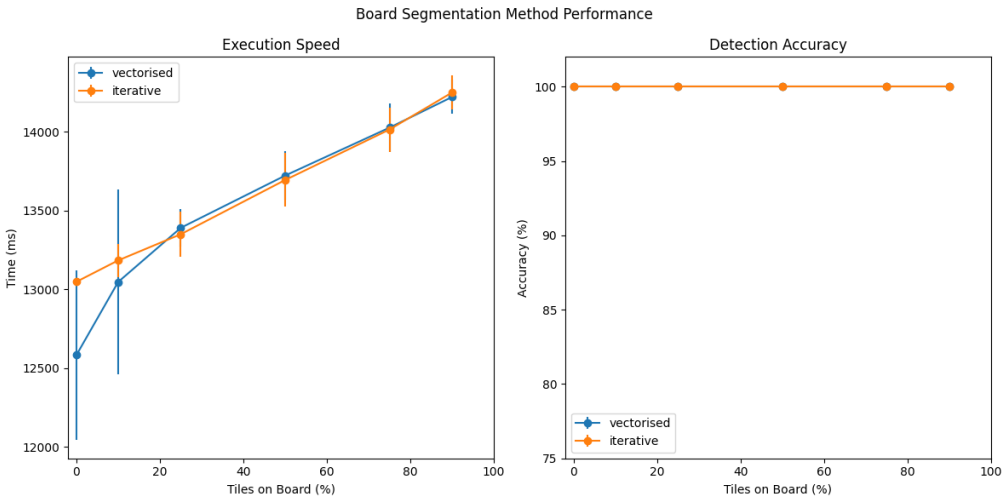


Figure 4.7: Test bench performance of board segmentation approaches on Raspberry Pi (1000 iterations)

QR Code Detection

In order to enhance the performance of the QR code detection stage of the board detection algorithm, various popular open source qr code detection libraries were evaluated. OpenCV served as the baseline library, alongside two Java-based libraries (BoofCV [116] and ZXing [142]) and another C++-based library (ZBar [143]). Official python bindings were available for each of these libraries [144] [145] [146], which were subjected to an initial simplified benchmark to determine suitable candidates for further research. Instead of producing a full board, random QR codes were generated using the same methodology as the main benchmark described earlier in the section, with each code resized to match a grid square in the original image processing pipeline. The performance of the libraries considered on the Raspberry Pi is presented in Table 4.1.

Detector	Mean (ms)	Std. Deviation (ms)
OpenCV	219.298	8.581
BoofCV	155.192	50.998
ZBar	4.274	0.270
ZXing	1953.751	57.966
ZXing-cpp	2.323	0.039

Table 4.1: Benchmark results of QR code detectors on Raspberry Pi (1000 iterations)

ZBar emerged as the most efficient library during this initial test, exhibiting approximately 51 times faster performance than the OpenCV baseline. Notably, BoofCV provided an alternative option that outperformed OpenCV, despite being implemented in Java, which can likely be attributed to a more sophisticated QR code detection algorithm [147]; although was likely too slow to be considered for use, potentially rendering micro-QR codes infeasible. Additionally, it

is worth noting that both BoofCV and ZBar have been found to surpass OpenCV, as well as a number of other libraries, in terms of detection accuracy, particularly in challenging scenarios characterized by variations in brightness, glare, shadows, and blur [147]. As ZXing's performance was orders of magnitude slower than the OpenCV baseline, it was initially deemed unsuitable for the project. However, a third-party C++ rewrite of the library was identified [148], which included its own Python bindings. Subsequently referred to as ZXing-cpp, the initial benchmarking using this new implementation, which is also included in Table 4.1 was incredibly promising, outperforming ZBar by $\sim 90\%$. Although ZXing tended to perform worse than ZBar and BoofCV when exposed to visual artefacts [147], ZXing-cpp has undergone substantial development since its initial fork, making it likely to perform differently to the Java version considered in the study. This version also supported the detection of micro-QR codes, providing a more performant alternative than BoofCV when considering the feasibility of such codes. Nonetheless, ZXing-cpp remained a relatively unproven library, and careful evaluation of its detection accuracy under various real-world conditions would be necessary before it could be considered superior to ZBar. Furthermore, its lack of documentation rendered it challenging to work with, and as such, given the suitability of other libraries, was not tested further.

Surprisingly, the benchmark also revealed bugs in many of the open-source detectors, and exhaustive testing in a reduced state space ($r \in [0, 99999]$) confirmed that small numbers of particular data payloads resulted in specific QR codes which OpenCV, PyBoof and ZXing were unable to detect, leaving ZBar and ZXing-cpp as the only libraries which correctly detected all payloads. These errors were confirmed by library maintainers [149] [150], stemming from false positives in the finder pattern, and are hoping to rectify them in a future release. Interestingly, there was minimal overlap in which QR codes caused issues across detectors, likely as a result of each using different algorithms to recognise the finder pattern. This issue highlights the importance of verifying that the patterns used when manufacturing tiles can be theoretically detected, using the test bench developed, prior to physical assembly.

An issue with segno's QR code encoding was also identified, in which messages with 4-digit payloads contained padding bits which were not standard conforming [151], although most detection libraries were not affected by this as these bits are typically ignored due to very few encoders correctly following the standard [149]. A full report of the problematic codes found for each library is available in Appendix C.

Following these issues, the detection capabilities of BoofCV and ZXing-cpp were also tested on M4 micro-QR codes, in which ZXing-cpp failed to detect 11,420 of the 100,000 tested codes due to the minor distortion caused from resizing the tiles to 88 by 88 pixels [152]. This was solved through tweaks to the parameters of the detector, but suggested that the library's micro-QR code detection capabilities were rather brittle, particularly when considering the aforementioned lack of documentation, providing additional justification to the decision to exclude it from subsequent testing.

Although the estimated performance of ZBar, and by extent ZXing-cpp, was sufficient to achieve the target sensor updates, taking $64 * 4.727 \approx 273$ms for full board detection, alternatives, and optimisations to the detection algorithm were still considered to further reduce the amount of processing time necessary. Rather than performing QR code detection on each segmented square of the board, detection could be performed "simultaneously" on the entire image, using the bounding boxes of the codes returned by the detector to determine the code's position as shown in Listing 4.4.

```
# Derive pixel dimensions of individual squares
subarray_size = np.array(dim / 8 for dim in board_img.shape)
def calculate_index(bounding_box):
    centroid = bounding_box.mean(axis=0)
    col, row = centroid // subarray_size # Maps centroid to 8 by 8 grid
    return int(row), int(col)
```

Listing 4.4: Derivation of QR code square position using bounding box

It was hypothesized that this approach could be more efficient for lower tile fractions, as sections without codes could be excluded more efficiently by the algorithm. Implementations for both the iterative and simultaneous algorithm were benchmarked for OpenCV, BoofCV and ZBar, with the results shown in Figure 4.8. The performance of each library is plotted on separate graphs to clearly illustrate the performance difference of the two approaches.

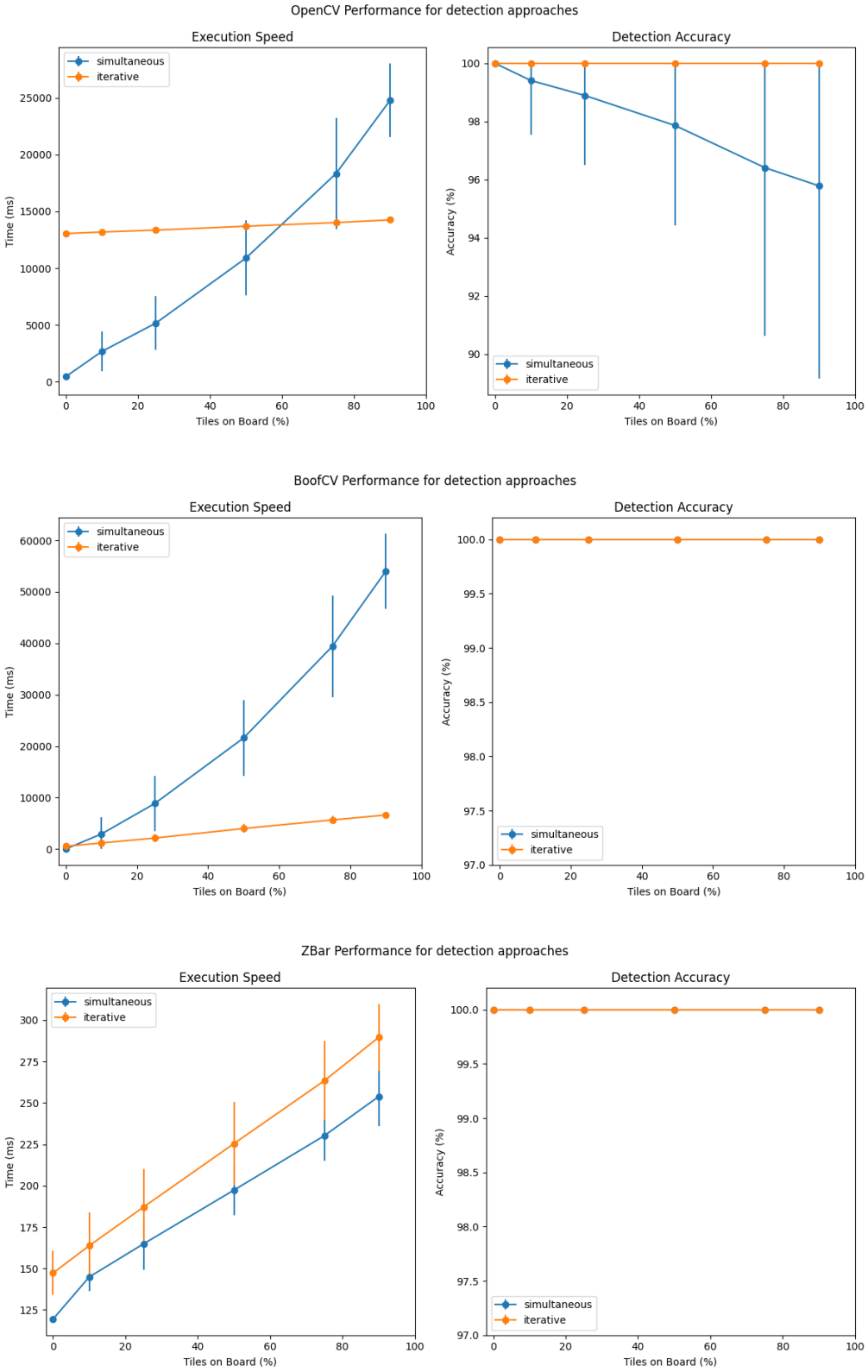


Figure 4.8: Benchmark results of QR detection algorithms for considered libraries (1000 iterations)

Surprisingly, this approach yielded quite different results with each of the 3 libraries, matching the hypothesis in the case of OpenCV, although notably at the cost of detection accuracy, which steadily worsened as the fraction of tiles on the board increased, making it likely that it would not perform adequately on nosier images. In the case of BoofCV, the simultaneous algorithm only provided an improvement for an empty board, quickly becoming more inefficient than the iterative approach by several orders of magnitude, whilst for ZBar it represented a $\sim$ 20% speed-up over the iterative approach across all tile fractions, without sacrificing any accuracy.

These results indicate that the simultaneous algorithm with ZBar provides the best performance, representing a $\sim$ 110-fold improvement over the execution speed of the initial OpenCV implementation, with the best performing approaches obtaining 100% detection accuracy for each of the three libraries illustrated in Figure 4.9. The execution speed of both ZBar approaches, as well as the iterative OpenCV and BoofCV implementations, appear to scale linearly with the number of tiles detected on the board, matching the hypothesis presented earlier. To attempt to maintain a faster detection speed at all stages of the game, a further improvement was considered, in which once the number of tiles acknowledged by the server exceeded 20% of the maximum number of tiles detectable by the board, ZBar's iterative algorithm could be used whilst skipping over confirmed tiles, rendering it more efficient than its simultaneous counterpart, and actually performing faster as more tiles are acknowledged. One limitation of this approach was that it reduced the number of opportunities the system has to detect and correct errors in the measured state, although it is likely that any errors in detection would be identified through the delta resolution mechanism on the server-side. As such, this combined ZBar approach was selected for implementation as the finalised detection algorithm.

An alternative "hybrid" algorithm was also considered, in which the grid could be divided into sub-grids containing multiple squares, each of which could employ simultaneous detection. However, given the superior performance of ZBar's pure simultaneous algorithm on boards with low tile density, this hybrid approach would likely only lead to a deteriorated performance in these cases, whilst lacking the flexibility of the iterative approach to skip individual tiles. As such, this approach was not tested further.

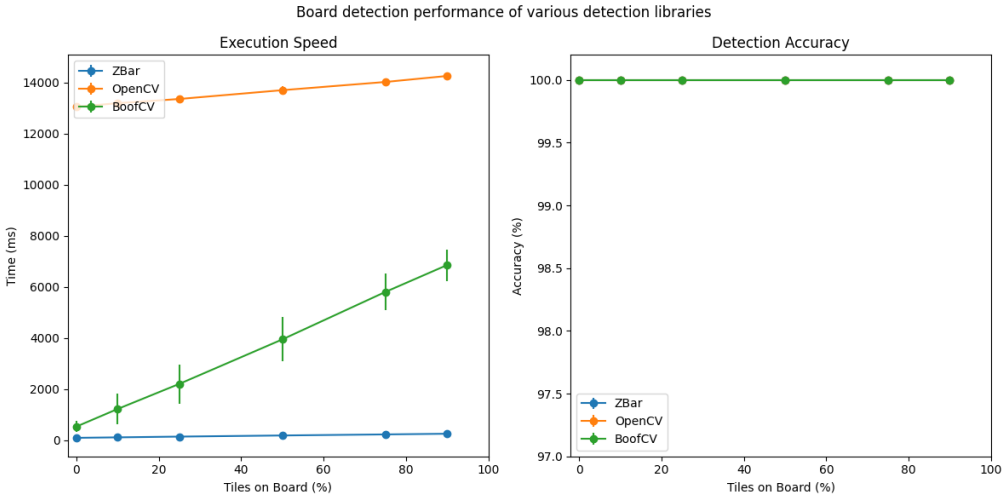


Figure 4.9: Test bench performance of detection libraries on Raspberry Pi (1000 iterations)

4.1.5 Camera Multiplexing

Once the feasibility of game state detection for individual board corners had been established, the extending the functionality to operate on all four board segments simultaneously. As outlined in subsection 3.2.2, ArduCam's camera multiplexer for the Raspberry Pi [95] was identified as a solution to process inputs from multiple camera modules. In order to select which input to capture data from, the Pi must drive the appropriate i2c and GPIO signals to the multiplexer as defined in its documentation [153], as well as restart the `Picamera2()` object to ensure that the updated camera feed is used.

An initial implementation which simply went through each camera and took an image was tested, revealing a ~ 4 second runtime to complete this routine, primarily due to the re-initialisation of the `Picamera2()` objects which were required at each camera change. This was quite surprising, as although the vendors did not provide any performance guarantees for the multiplexer, their demonstrations indicated an average of 10 fps was attainable for full-colour images [154]. Unfortunately, the vendors were not able to provide support to solve the framerate issue, rendering the multiplexing approach non-viable.

However, an alternative hardware solution was identified to overcome this issue, AdruCam's Camarray HAT, which instead of allowing the Pi to multiplex between the various camera modules connected to it merge the image feed from all four cameras into a single image using specialised hardware, allowing the Pi to consume the merged feed without the need for any further camera interface manipulation [155]. The vendors guarantee a minimum frame-rate of 10 fps for the image resolution used by the existing board capture implementation, which is quadrupled when considering the merged image returned from the Camarray. Although this potentially introduces a 100ms delay between the actual board state and the image processed by the detection algorithm, given the performance improvements attained in the previous section, this would still allow the board state on the server to remain within 1 second of the match being played.

Unfortunately, due to sourcing difficulties, it was not possible to obtain this hardware for testing before the completion of the project, which was partially why it was not considered over the multiplexing module in the design stage. An implementation using this module would be scheduled in late-2023, providing ample time for further refinements prior to the 2024 tournament.

4.1.6 Rack Detection Algorithm

The overall rack sensor pipeline followed board detection closely, using identical camera configurations and preprocessing stages. However, unlike the board, the position of the tiles is less restricted, and although the image could be cropped to only include the relevant horizontal and vertical sections, the tiles still possessed significantly more degrees of freedom than previously, as they can be placed in any position on the rack, as well as be orientated in any direction. This suggested that QR detection performed across the entire image would be most appropriate, simply combining all the values of the codes detected to generate the rack state.

Based on the performance of the various QR detection approaches explored in subsection 4.1.4, it was likely that ZBar's multiple code detection functionality would be most suitable, although a test bench similar to the one devised for board detection was developed to verify this. Once again generating random codes with values $r \in [0, 9999999]$, these tiles were randomly placed onto a blank 88 by 860 pixel array, reflecting the estimated dimensions of the rack based on its width relative to a quarter board. Devising a suitable algorithm which randomly distributed the tile positions, whilst ensuring that the tiles did not overlap, was surprisingly non-trivial.

A naive approach which would randomly choose a valid tile position, in which the placed tile did not overlap with any existing tile, and backtracked when unable to place all tiles was first considered, although was not ideal as it could lead to long setup tiles in a benchmark with multiple iterations. Instead, an iterative algorithm was devised which always ensured that all remaining tiles could be placed when considering where to place a particular time. This was achieved by maintaining a set of contiguous empty regions on the resulting rack image which could accommodate at least 1 tile, called "gaps". When placing a tile, the algorithm proceeded as follows:

1. Select a random gap g from the available gaps, following a width-weighted probability distribution as the insertion point for a tile to ensure that intervals in smaller gaps are not oversampled.

2. Compute the number of additional tiles n which g must contain by comparing the total capacity of all gaps with the number of remaining tiles to place.

3. Determine the valid intervals which the tile can be placed in g such that the resulting two new gaps l and r have total capacity $c \geq n$. This can be computed efficiently by considering the maximum amount of space which can be "wasted" by the current tile placement.

4. Randomly select the tile insertion point from the set of valid intervals computed in the previous step.

5. Replace g from the set of available gaps with l and r, provided their respective capacities are > 0.

This approach ensured a fixed time for generating each image with only $O(n)$ time-complexity in the number of tiles tested, which was parameterised to ensure the chosen algorithm performed well on all possible rack states. Examples of some of the rack states generated by the benchmark are shown in Figure 4.10.

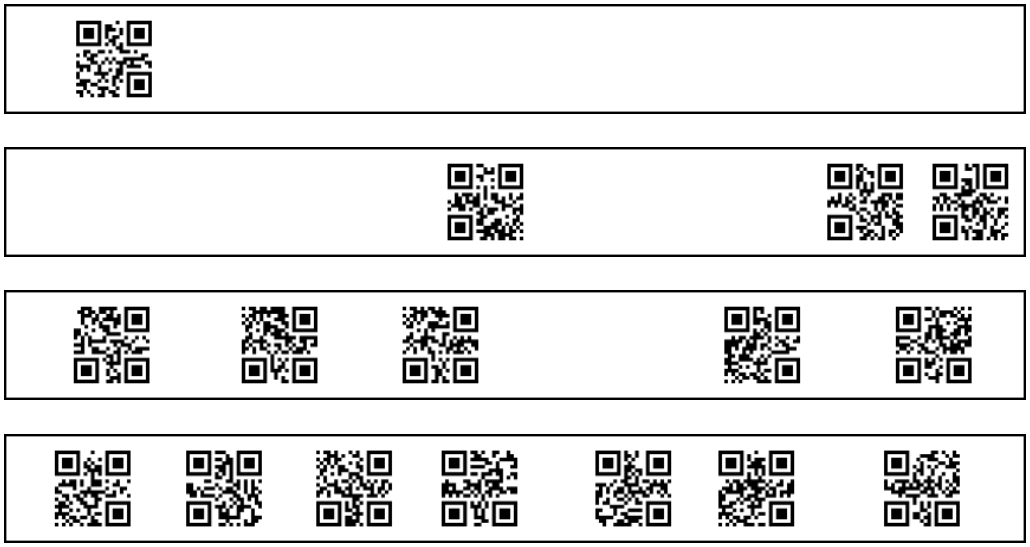


Figure 4.10: Random rack states with different numbers of tiles.

The benchmark evaluated the performance of OpenCV, BoofCV and ZBar, using the F1 score [156] to measure the overall quality of detection in order to penalise false positives and false negatives evenly. The results are presented in Figure 4.11, and confirm ZBar as the most suitable choice, with both the best speed and detection accuracy. Surprisingly, none of the libraries were able to attain a 100% detection accuracy for all rack states, despite BoofCV and ZBar previously performing flawlessly when detecting tiles across the entire board. This suggested that the changes in orientation which the tiles are capable of degrade the detectors' accuracy. However, given the speed at which tile detection is performed, with ZBar taking under 60 ms for any number of tiles, multiple samples of the same state should mitigate these potential additional inaccuracies.

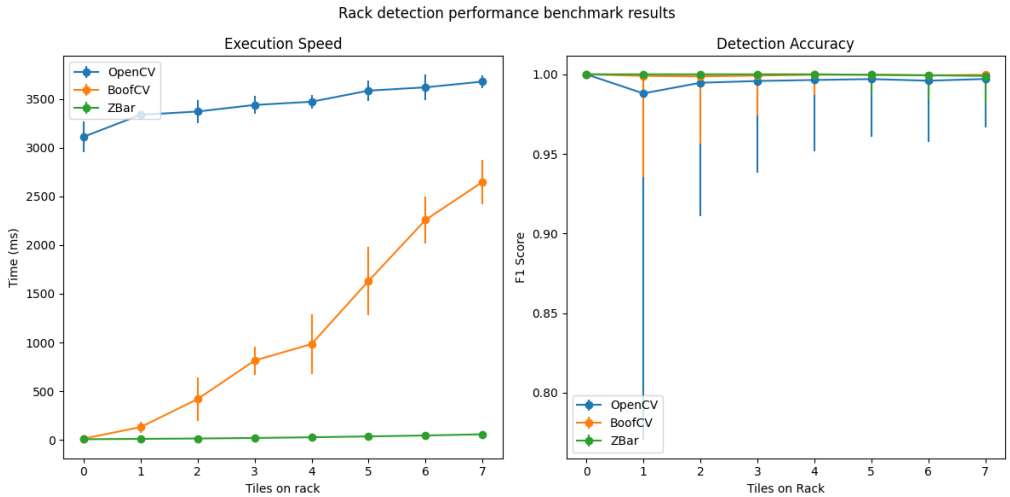


Figure 4.11: Test bench performance of detection libraries on Raspberry Pi (1000 iterations)

Alternatively, the rack image itself could be processed from multiple orientations to in parallel using all four of the Pi's cores, ensuring that at least one of them processes the QR code using its traditional upright orientation. This approach would make better use of the underlying hardware, particularly as three of the rack sensor cores were previously unused. Given that errors using ZBar's detector always stemmed from false negatives, indicated by the perfect precision scores for all tests, the union of each measured rack state could then be taken to produce the overall rack state. This multi-orientation method was benchmarked against the standard implementation above, and the two methods are compared in Figure 4.12. As hypothesized, the parallel processing approach was able to recover perfect detection accuracy over its standard counterpart, at the cost of $\sim$ 25ms for all numbers of tiles, representing the overhead of maintaining the pool of 4 worker processes, although with the overall processing time comfortably under 100ms this execution performance penalty is well worth it. The updated multiprocessing implementation as a result of the integration with asyncio for the sensor client implementation described in subsection 4.2.2 had minimal impact on the performance relative to the benchmarked method, incurring an additional $\sim$ 4ms overhead.

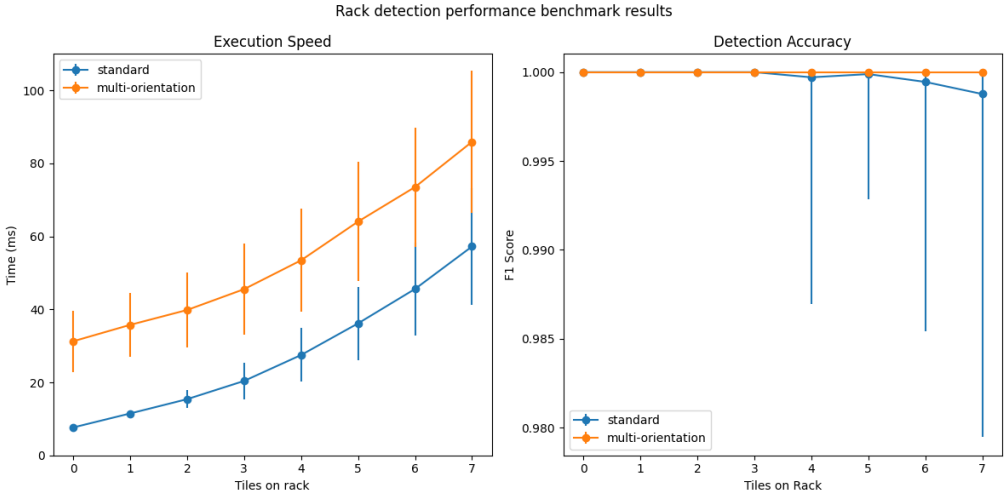


Figure 4.12: Comparison of standard and multi-orientation approaches on Raspberry Pi (1000 iterations)

Unlike board detection, the rack sensor simply publishes all tiles detected directly to the server, as at most 7 tiles, represented as characters, need to be sent, consuming minimal bandwidth. This shifts the tracking of the rack state to be performed entirely on the server side, which is preferable as the central source of truth, rendering the rack sensor almost completely stateless.

4.2 Server

From the server architecture outlined in subsection 3.3.3, it was clear that it would need to be implemented to support communicating with multiple components in parallel, listening on different ports for the HTTP and TCP servers whilst dispatching requests to the Woogles API. Although there are a number of frameworks supporting concurrency in python, including threading [157] and multiprocessing [123], the `asyncio` library [158] is best suited for this application, as it is specifically designed to handle multiple I/O bound tasks, in this case network requests, incredibly efficiently, without any of the overhead of maintaining multiple threads of processes. `asyncio` operates in a similar fashion to other back-end technologies such as Node.js, in which tasks are scheduled on an event-loop, running one at a time. This eliminates other challenges presented by alternative approaches for handling concurrency, as data structures can be shared between asynchronous tasks and are always guaranteed to be updated atomically, since only a single task can execute at any given time, greatly simplifying communication between different parts of the server.

4.2.1 Game Logic

As discussed in subsection 3.3.3, a package encoding Scrabble game logic, capable of calculating scores and supporting challenge requests, was necessary to fulfil the project's functional requirements. This package was implemented in pure Python, as the use of external libraries was considered unnecessary given that the computational complexity of the tasks required was relatively insignificant, simply requiring manipulations on a 15 by 15 array.

Code quality was a core focus of the implementation, both to meet the project's maintainability goals as well as to facilitate testing, which is expanded on in chapter 5. The package was divided into 4 core classes encapsulating their relevant logic:

`BoardPos` manages positions on the board, defined by their row and column. It ensures positions are always valid or in-bounds, and supports the generation of neighbouring positions for a given direction enumeration class: horizontal or vertical.

`Tile` represent the tiles used by players, storing the letter contained and corresponding points score. Also manages blank tile letter designation.

`Move` is a collection of (`Tile`, `BoardPos`) pairs which are played together, which would be formed directly from the sensor data during delta resolution. It is able to check if it is a valid move, meaning that all tiles are played along a particular direction to facilitate error detection.

`Board` handles the state of the game board, updating it by applying `Move`s to it, and verifying that these moves are valid. The score corresponding to each move is also calculated, and methods to update the blanks of the latest play, as well as obtaining the words formed by that play, are available.

When applicable, these classes also contained methods which allowed them to be formatted in a schema consistent with the Woogles API to simplify communication in other parts of the server.

Exploiting regularity in the geometry of the problem was a particularly interesting stage of implementation, ensuring that algorithms were implemented in 1 dimension, and parameterising them by the direction applied in allowed substantial code reuse for both calculating scores and determining words formed by a move. A particularly useful helper function in accomplishing this was `get_tiles_through_pos(anchor: BoardPos, dir: Direction)`, which used the underlying primitive neighbour generation from `BoardPos` to yield the sequence of tiles going through the anchor in word order, top-to-bottom or left-to-right. Listing 4.5 shows the `Board`'s internal score calculation method, and highlights how the abstraction techniques used above greatly simplify this rather complicated business logic into a concise and readable method.

The symmetry of the board about the row = 7 and column = 7 axes also allowed the simplification of encoding the premium tiles on the board. By mapping any position to its corresponding position in the top-left corner of the board (Q4), only the tile types of a quarter of the board needed to be statically configured, rather than all 255 squares.

Once complete, the codebase was converted into a Python package, allowing straightforward inclusion into other parts of the server code.

4.2.2 Communication with Sensors

Given the limited compute resources of the sensors, performance was a core consideration when selecting an RPC framework to be used for TCP socket communication. Capnproto [80] was chosen, in large part due to its interoperability with asyncio sockets, which more popular libraries such as Google's gRPC [159] lack, leading to significantly lower overhead and faster performance [160]. Written in C++ with a focus on speed, Capnproto serializes messages in binary, making them significantly more efficient than text-based serialization schemes such as JSON or CML, in an "in-memory" format, allowing for zero-copy serialization and deserialization. It also supports promise pipelining [161], a technique which minimises the number of network round trips by automatically combining dependent requests into a single trip. Although this feature was not used

```python
def _get_score(self, move: Move):
    def get_score_1D(placed_tiles: List[Pos], dir: Direction):
        score = 0
        word_multiplier = 1
        for tile, pos in self._get_tiles_through_pos(placed_tiles[0], dir):
            letter_multiplier = 1
            if pos in placed_tiles: # Only count multiplier if included in placed tiles
                match Board._get_square_type(pos):
                    case SquareType.LetterX2:
                        letter_multiplier = 2
                    case SquareType.LetterX3:
                        letter_multiplier = 3
                    case SquareType.WordX2:
                        word_multiplier *= 2
                    case SquareType.WordX3:
                        word_multiplier *= 3

            score += letter_multiplier * tile.value

        return score * word_multiplier

    main_score = get_score_1D(move.coordinates, move.direction)
    bingo_bonus = 50 * (len(move.coordinates) == 7)
    cross_score = 0
    for pos in move.coordinates:
        opposite = move.direction.opposite
        if self._forms_new_word(pos, opposite):
            cross_score += get_score_1D([pos], opposite)

    return main_score + cross_score + bingo_bonus
```

Listing 4.5: Score calculation method

in the prototype implementation, it enables the development of more complex communication between the sensors and server in the future to require minimal changes to the existing interface, fitting well with the maintainability goals established in section 1.1.

The communication protocol was defined using Capnproto's Interface Description Language (IDL), which then automatically generates the code required when imported into a Python file. Structures such as `Move` were defined to package data matching the structure of their corresponding types in the game logic package developed in subsection 4.2.1, and interfaces for the server and sensors were designed to expose the desired functionality to the opposing party, which are illustrated in Figure 4.13. In order to enable two-way RPC, a sensor provides an implementation of its target interface, which are passed to the server as a variant when registering to allow it to determine the type of sensor it is connected to. Given the use of stateful connections, the TCP server can track the administrative data associated with each sensor, removing the need for the sensor itself to manage this information. This renders the system more robust to outages affecting the sensors, which are more likely to occur than server issues. The `matchId` parameter in the interfaces shown in Figure 4.13 simply communicated to the sensor whether it had been assigned to an active match or not, as well as to facilitate debugging in early implementation stages. When awaiting a response from the RPC call, a software level timeout can be set to trigger if no response is received in a specified period, which was added as an additional layer of safety to handle cases where issues with the caller's internal state may prevent it from responding appropriately to the request. Although it was likely that events causing such issues would cause the heartbeat mechanism to also fail, inducing a disconnect, this timeout ensured it was impossible for isolated parts of the communication to fail silently.

The TCP server itself does not implement the `MatchServer` interface, but rather creates a separate `SocketHandler` which is responsible for doing so for each new connection and encapsulates the

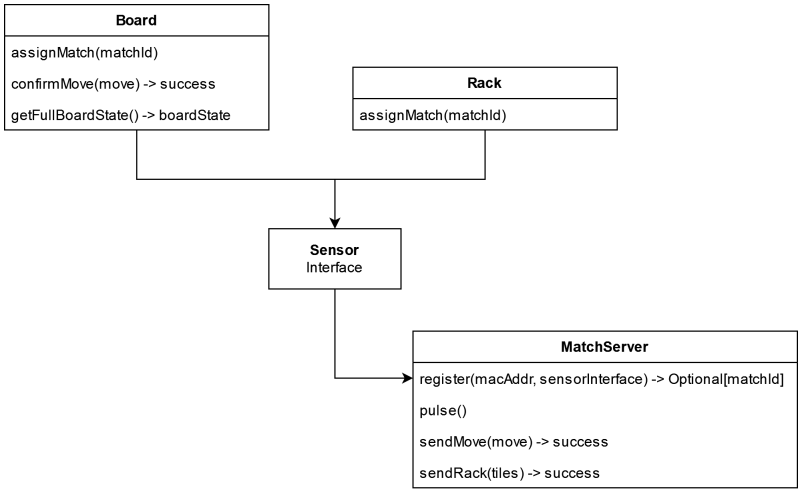


Figure 4.13: RPC interfaces for communication between sensors and server

RPC logic and boilerplate code which connects Capnproto to the asyncio socket reader and writer objects. These `SocketHandler`s are then managed by the TCP server's `ConnectionHandler`, which is responsible for monitoring the currently available sensors to allocate to new matches, as well as maintaining a mapping of sensor connections to ongoing matches, managing the reconnection logic outlined in subsection 3.3.3. The MAC addresses of the sensors were selected as identifiers for this purpose, as they satisfied the requirements of uniqueness whilst remaining static across the lifetime of the device, meaning that the same identifier would be used by a sensor upon reboot, and allowed the `ConnectionHandler` to maintain a mapping of MAC address to "admin" data including match ID and role, for active games.

One limitation of the `MatchServer` interface defined was that it exposed the functionality of both sensors to the device it was connected to, which is not ideal as it allows the connected device to potentially call either method regardless of its type, potentially before even being assigned to a match. This also bloated the `SocketHandler` class, which was required to contain logic maintaining the state for both types of sensors, rather than only for the sensor type it was actually connected to. An alternative design which solved these problems was considered, and is shown in Figure 4.14, which removed the data pushing methods of the `MatchServer` into separate `DataFeed` interfaces, which could be passed back to clients when assigned to a match, or upon registration to handle reconnecting sensors.

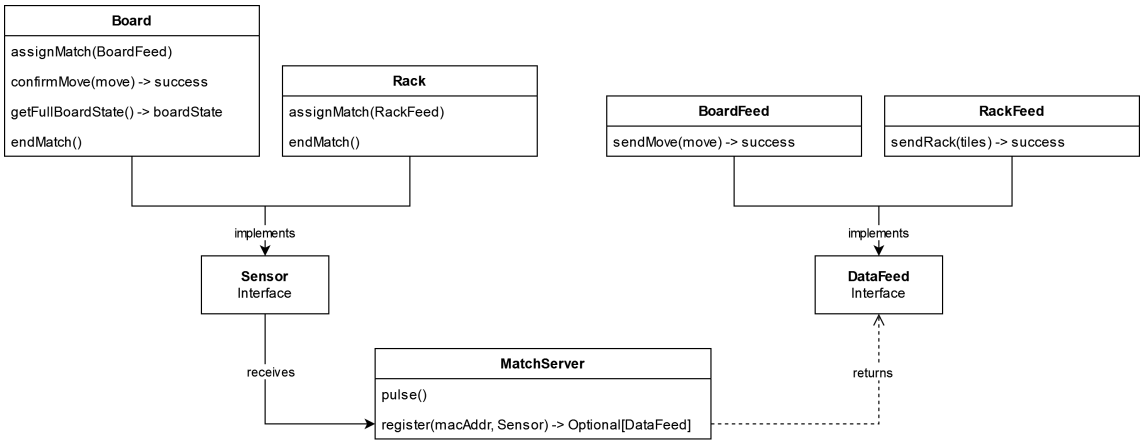


Figure 4.14: Alternative RPC interfaces to enforce safe communication

These `DataFeed` objects could then be responsible for managing the state relevant to the match the sensor is publishing data for, such as the player assigned to a particular rack, removing the need for this logic to be stored in the more generic `SocketHandler`. Although this approach adds

another layer of abstraction, rendering the overall design more complex, the enforced correctness it provides with the design of its protocol and simplifications it brings to the `SocketHandler` are well worth it, and as such the final server implementation was updated to use this updated version.

A singleton `GameStateStore` object was created which would be used to share match data between the TCP server and HTTP server, acting as the single source of truth in the system. This object would contain a mapping of match ID to `GameState`, which would contain the `Board` representing the match from the Scrabble game logic package, as well as the necessary delta resolution logic to handle the incoming information from the sensor `DataFeeds`. The use of the singleton data pattern is important as it ensures that any objects accessing this structure always refer to the same, unique, global instance, reducing the likelihood of implementation errors. Although singleton objects are not available in python by default, they can be implemented relatively easily through the use of metaclasses, as show in Listing 4.6.

```
1  class Singleton(type):
2      _instances = {}
3      def __call__(cls, *args, **kwargs):
4          if cls not in cls._instances:
5              cls._instances[cls] = super(Singleton, cls).__call__(*args, **kwargs)
6          return cls._instances[cls]
7
8  class GameStateStore(metaclass=Singleton):
9      pass
```

Listing 4.6: Implementing singletons in Python using metaclasses [162]

When receiving a request to start a new match, the `ConnectionHandler` performs the following steps:

1. Selects a board `SocketHandler` and two rack `SocketHandlers` from the pool of available connections, assigning each rack to a particular player.
2. Creates a match ID, which is used to uniquely identify the state for a particular match.
3. Initialises `GameStateStore[match_id]` to a blank game state.
4. Updates the active match data MAC address mapping of the selected `SocketHandlers`
5. Instructs the `SocketHandlers` to construct the required `DataFeeds` containing the relevant setup data and assign it to their respective sensors.

When handling a registration from a new connection, the `ConnectionHandler` can simply query its active match data mapping to check if it contains the connecting sensor's MAC address, and if it does proceed immediately to item 5 of the algorithm above, with the `SocketHandler` returning the `DataFeed` directly from the `register` RPC. When handling new information from their sensor, the `DataFeed` can directly access the relevant game state in the `GameStateStore` using its match ID and other admin data, pushing to the appropriate delta stream.

Delta Resolution

A naive approach to the problem of delta resolution would involve storing all incoming deltas in a list, and combine this list into a single change once the end turn signal is received from the companion application. However, this is undesirable, as the $O(n)$ complexity of such an operation, taking place on the core event loop, could delay the processing of other messages, as well as lead to unpredictable update times to the livestream view depending on the duration of a turn, which causes the number of deltas to process to scale linearly. To correct this issue, delta resolution was applied both between sensor updates, and when determining the move played upon receiving an end of turn signal from the companion application, with both operations taking $O(1)$ time-complexity. It can be helpful to conceptualise the intra-delta resolution mechanism as a fold performed over all deltas, with each step of the fold computed as a new incoming delta is received.

Whenever a board delta is published to a game state, the incoming changes are compared to the current state, based on previous accumulated changes, to ensure these are consistent with one another, with repeated snapshots increasing the level of confidence the system has in the measurement. Obviously, it is also possible for players to change the tiles they have placed over the course of a turn, and as such changes to the values of the tiles within a delta are not considered problematic, but if inconsistencies such as changes in the values of confirmed tiles are detected, a warning is logged to indicate to tournament organisers that there may be an issue. Once processed, the new accumulated delta state is stored, ready for use in processing the next delta.

Racks constantly publish a full snapshot of the tiles available, and the changes between these snapshots is also tracked to detect errors in the sensor. At the start of player x's turn, the game state contains a snapshot with all the tiles on their rack which have been detected whilst x was drawing tiles on their opponent's turn. Should there be a mismatch between the number of tiles which should be on their rack, typically 7, and the snapshot in the game state, an error is raised, both to the players and tournament organisers as it indicates either sensor failure, or that the player has drawn the incorrect number of tiles. Should a new snapshot from the rack containing the correct number of tiles be subsequently received, the system assumes the latter case occurred, and is able to proceed. Over the course of x's turn, the server will receive rack snapshots containing varying amounts of tiles as the player considers which to play. The delta resolution mechanism simply verifies that these snapshots are subsets of x's starting rack state, and if any mismatches are detected, another warning is logged.

In both cases, errors are typically only raised during the confirmation of these deltas upon an end of turn event. Any inconsistencies in the measurements from the board and rack which were previously categorized as warnings will now be logged as errors, as these data points are clearly inaccurate and cannot be used in further resolution. The use of low-confidence snapshots, in which few repeated updates were received, is permitted, but warnings are logged to the organisers to keep them informed. In the more common case where these issues do not occur, the latest deltas from the board and player's rack are compared, and the game state verifies that all tiles played are present in the board's delta before generating the move played from the board delta and applying it to the internal game representation. The server tracks the state of remaining tiles in the bag using a dedicated `TileBag` class, which is used by the rack delta resolver to ensure that any new rack states are feasible, updating the contents of the bag with the tiles drawn. The game logic package developed previously also verifies that the move is valid as an additional layer of error detection. Once a move has been applied successfully, both the score of that move, as well as the number of blank tiles it contained are known to the system, and this information can be communicated back to the players, as well as to the Woogles API. If any errors are encountered during this process, both the broadcasting team and players are informed, to ensure they are aware that the game state is no longer accurate, and scores displayed on the companion app are no longer correct.

Confirmed moves are also communicated back to the board sensor via its RPC interface, to ensure that new deltas are published using the updated state. The possibility of receiving a delta based on the previous state after publishing the update was considered, as it is theoretically possible for this to occur despite TCP's guaranteed ordered delivery, as illustrated in Figure 4.15. However, it is worth noting that this can only occur whilst awaiting the move confirmation response, as once this is received ordered delivery guarantees that all subsequent snapshots are sent using the updated state. To handle these cases, the board delta resolution was updated to ignore deltas which contained already placed tiles, provided that these matched the existing state of the board. Although this logic may reduce the server's ability to detect if the board is out-of-sync with the server side state, this approach was preferred over the addition of additional logic in the game state class to track this, particularly as TCP's reliability guarantees ensure that any move confirmations are received by the board.

When player x has completed their turn, they must now draw tiles from the bag back onto their tile. The remaining tiles in the bag are tracked by the server, and if any impossible racks are received from the sensor, then another warning is issues to the organisers. Once a valid snapshot containing the correct number of tiles is received, the server stores this as the current player rack state, and once a sufficient confidence interval is achieved, or x's turn begins, confirms this data, transmitting the update to Woogles. The initial minimum confidence interval was optimistically set

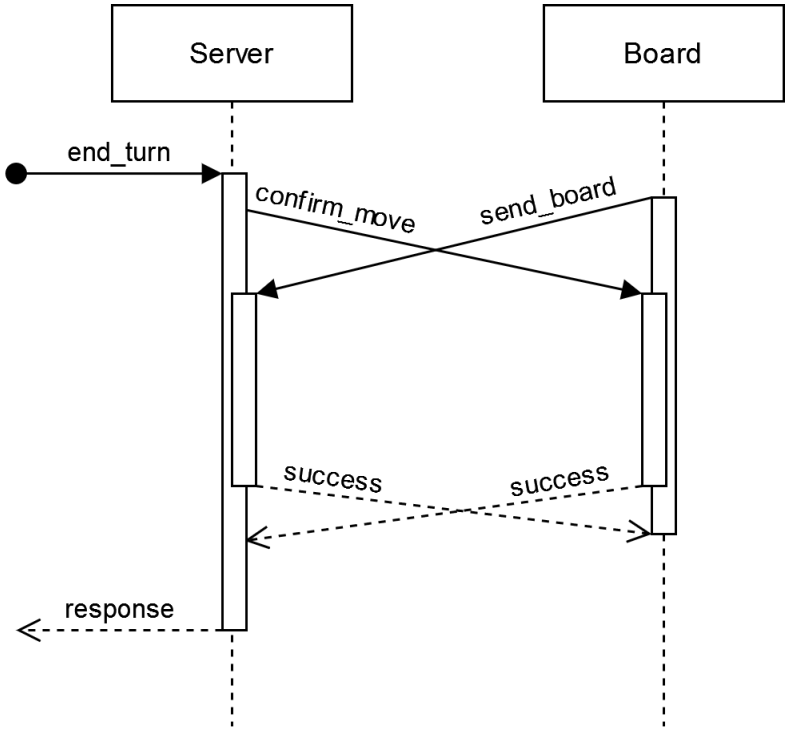


Figure 4.15: Receiving old snapshot after sending move confirmation

at 2 repeated snapshots, as based on initial sensor tests the results appeared extremely accurate, but further sensor testing would be necessary to tune this value optimally. This logic is also used at the start of the game, which begins as soon as the player going first draws a single tile. In this particular case, the system is unable to use the snapshot from the end of the previous turn, and instead uses the snapshot detected during the player's turn.

The delta resolution logic was also configured to support challenges, in which it would verify that a successfully challenged play was removed from the board, and that the tiles contained in the move were added back to the relevant player's rack.

Match Completion

The server determines that a game has completed when the bag of tiles is empty, and one of the players has played all tiles on their rack. When this event occurs, the server calculates the end of game penalty, based on the remaining tiles in the other player's rack, as well as the players' remaining time, to determine the final score of each player. The sensors are placed into idle mode, preventing unnecessary data from being published to the server, and the `SocketHandler`s for each of the sensors are placed back into the pool of idle connections, but remain grouped such that if another match is set up, the same sensors will be used, minimising organisational overhead for tournament staff. Although WESPA rules outline alternative conditions for the end of a match, these were not implemented, with the reasoning behind this decision explored in subsection 6.1.1.

Sensor Client

To enable communication on the sensor side, a client was developed which could operate in tandem with the existing detection approaches. Given that multiprocessing was already used to perform the majority of the CPU bound work, combining the networking and image detection to run in parallel with asyncio was relatively straightforward. One potential issue identified was during image capture, as the `Picamera2()` objects representing the camera array or single sensor in the case of the board and rack respectively needed to be owned by the main process to ensure concurrent attempts to access the camera were impossible. However, capturing images is also an I/O bound task, and the Picamera2 library provides support for non-blocking image capture, in which a callback function can be specified to run on the image array once it has been captured.

Unfortunately, this functionality offered minimal direct integration with asyncio [163]; as such, a workaround was implemented to take full advantage of this asynchronous callback mechanism.

This client had the added responsibility of managing the state of the sensor. Upon startup, the client tries to connect to the server, retrying up to 5 times. If it is unable to connect, the client shuts down, requiring a device restart to try reconnecting once again. Once it is able to connect, the client enters an idle state, simply sending pulses at 5 second intervals to keep the connection to the server alive. When receiving a `DataFeed` interface from the server, the sensor switches into data collection mode, and begins capturing data and publishing it to the server. The rate of data published is throttled to 100 ms intervals for the rack, as the value of each snapshot follows the law of diminishing returns, with more frequent updates unnecessarily consuming server resources. Based on the performance analysis of the board detection, throttling would not be necessary as deltas required an absolute minimum of 125 ms, without considering the time taken for preprocessing and the overhead of multiple processes. Given the small message sizes of 100–300 bytes, this results in an upper bound on the total bandwidth for game state detection at $300\ \text{B} * 10 * 3 * 25 = 225\ \text{kB} \cdot \text{s}^{-1}$, which is negligible compared to what typical modern routers can handle.

The board client performs some extra work to track the confirmed board state on the server, which it uses to generate deltas from the board snapshots obtained on-chip. To separate this logic from the rack client, a base client class was first implemented, which would be responsible for handling the general connection, registration, and heartbeat logic described previously. To ensure these functions are properly suspended on disconnect, the client keeps an inventory of continuously running tasks, which can be cancelled on disconnect. If at any point the connection between the server is severed, the client first cleans up its internal state, then simply tries to reconnect, once again with a maximum number of attempts set at 5, defaulting back to the idle state whenever reconnecting. The logic specific to the rack and board clients were then implemented in derived classes, which could pass tasks down to the base client they contained to execute regularly, ensuring these could also be handled correctly on disconnect.

The approach of maintaining the majority of the state on the server side renders the sensor clients much simpler, rendering the whole system more robust to sensor failure, and allowing the system to recover automatically for small outages. If a sensor is able to recover connection and transmit a new snapshot within the same turn it disconnected, the server is able to continue without issue, as no critical game data has been missed, although the delta resolution algorithms may warn still the tournament organisers if a high-confidence snapshot cannot be generated.

4.2.3 Communication with Companion Application

The AIOHTTP framework [164] was selected to implement the web server which would communicate directly with the tablet. Whilst alternatives such as Flask [165] provide a simpler, more popular alternatives, AIOHTTP was designed to work directly with asyncio, making it much easier to integrate into the top-level server. This integration also allows AIOHTTP to perform significantly faster than Flask, as it scales better when handling large numbers of simultaneous connections or during long periods of blocking I/O [166], which is particularly relevant in this application where in order to handle requests from the companion application, additional communication with the sensors is often necessary.

HTTP end-points were defined to match the communication protocol outlined in subsection 3.3.3, which then triggered the necessary logic in the `GameStateStore` and associated `SocketHandler`s to perform the necessary updates, as well as publishing changes to the Woogles API bridge. Given that requests to the HTTP server trigger the flow of events throughout the system, they often trigger requests of their own. However, in order to minimise the round-trip time between the server and companion app, these requests are typically initiated in the background, allowing the web server to first respond to the client before waiting for these requests to complete. The only notable exception is during match setup, in which the server waits for confirmation from the sensors before responding to the companion application with the newly created match ID, as ensuring that the sensors are initialised correctly is essential.

To handle challenges, another singleton `Dictionary` object is used, which loads the 2021 Collins

Scrabble Word (CSW21) list, avoiding unnecessary copies of this expensive object from being created. The HTTP server can then simply access this object to verify whether a particular word is valid, resolving the challenge according to the 5-point penalty rule [167] used in Alchemist Cup 2024; if any challenged word is invalid, the challenge is successful, meaning that the challenged player must undo their previous move, else the challenge is unsuccessful, and a 5-point penalty is applied for each valid word challenged.

4.2.4 Woogles API Bridge

Due to the substantial scope of the implementation work required in this project, the Woogles API Bridge was not implemented. This work was not considered particularly challenging, as web requests to the API could be integrated with the existing server framework trivially using `aiohttp`, although would likely be time-consuming, as messages for each game event would require relaying to Woogles. Until this feature is supported, the server logs enable the central match state to be observed, which is adequate for the feasibility demonstration this project aims to accomplish.

4.3 Companion Application

In order to include all the required functionality on the application was split into two main screens, an initial setup screen, and the match screen itself. The setup screen, as shown in Figure 4.16 would typically be used prior to the match to by the tournament organisers to prepare the board and racks for the players. Once entering the player names, the app sends a setup request to the server, causing it to allocate sensors through the processes outlined in subsection 4.2.2. If successful, the assigned sensors will blink LEDs to indicate to the organisers which have been assigned to the match, with racks displaying either blue for player 1 or red for player 2. Whilst this does force tournament organisers to set up matches one at a time, as attempting to start multiple games would cause multiple sensors to flash, once games have been set up initially, subsequent matches should be simpler to manage as already grouped hardware is reused whenever possible. Note that player 1 is always assumed to go first, meaning they will always be sat on the left side of the table, which was suitable for the project sponsor.

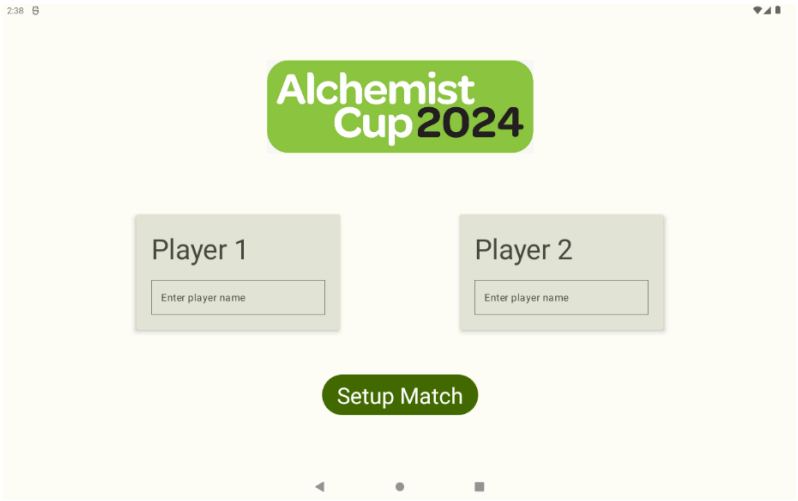


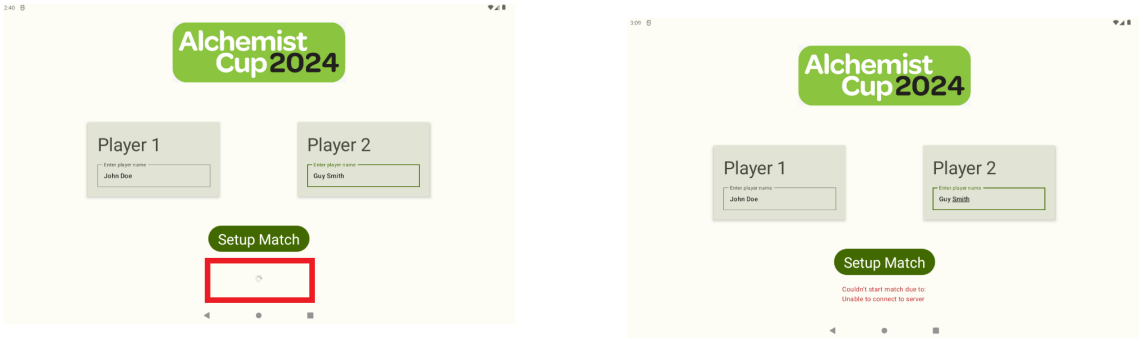
Figure 4.16: Companion application setup screen

The application indicates to the user whenever network requests are being performed, as well as any errors obtained when communicating with the server, as indicated in Figure 4.17. On the setup screen, any errors encountered will prevent the application from progressing to the match screen, as if setup is unable to be performed successfully then game data detection is impossible. The types of errors which may cause this are outlined in Table 4.2.

When errors are instead encountered intra-match, the application will continue to function, albeit with limited functionality. The screen will inform the players that an error in game detection has occurred, meaning that the scores displayed may no longer be correct, and players will no longer

Error	Causes	Message
Bad connection	Server offline, device offline	"Unable to connect to server"
No boards	All boards allocated, board offline	"No available boards"
Insufficient racks	All racks allocated, racks offline	"Insufficient available racks"

Table 4.2: Setup-time errors



(a) Loading icon indicating network communication

(b) Error message displayed

Figure 4.17: Communication with server and error detection

be able to submit challenges through the application. However, timekeeping functionality, which is performed entirely on the client side, will be preserved, which is essential to preserving the integrity of the match. Whilst a detailed error will be logged on the server side for tournament organisers, a generic error will be used for players, as simply informing them of an issue is the system is sufficient, and a more detailed error message is likely to be more confusing than helpful.

Once a match has been set up successfully, the application automatically proceeds to the match screen, displayed in Figure 4.18, which is the main player-facing UI. When ready, players may start the match by pressing the button, which will automatically activate player 1's timer, as well as their end turn button. To facilitate the user experience, inactive buttons are always greyed out, giving a clear indication to the players how they may interact with the application at any given moment.

Figure 4.18: Companion application match screen

Once a player completes their turn, the application sends a request to the server, receiving the score accomplished by the move played, as well as the number of blanks contained. Note that it since blank tiles have a value of zero, it's always possible for the score of the move to be determined prior to receiving information on the blank tiles, allowing this information to always be passed in

the same way, regardless of whether a move contains blanks or not. If the number of blank tiles is greater than zero, the application queries the user for the value of these blanks through a dialogue. As shown in Figure 4.19, error checking on these inputs is performed on the client side to ensure a valid input is provided, with each box accepting a single alphabetical character. Play proceeds for the other player while this dialogue is open, in accordance with WESPA tournament rules, although it is worth noting that this player will be unable to end their turn until the blank dialogue has been resolved, which should not be an issue as minimal time is required to complete it.

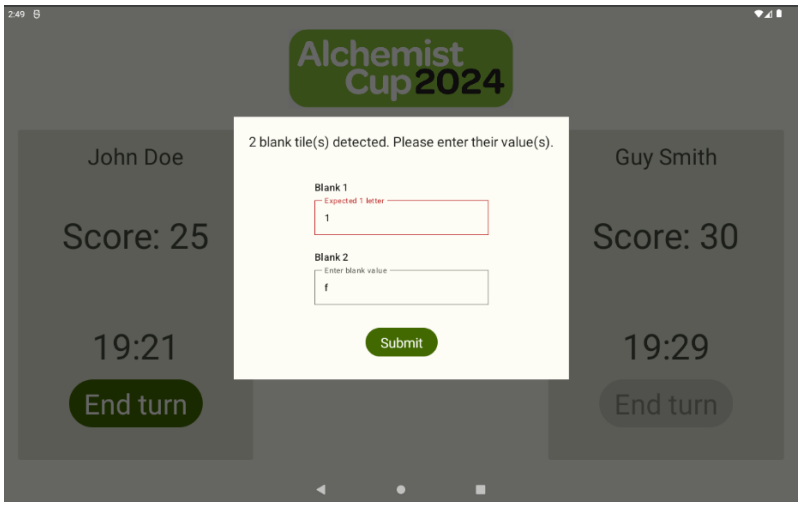
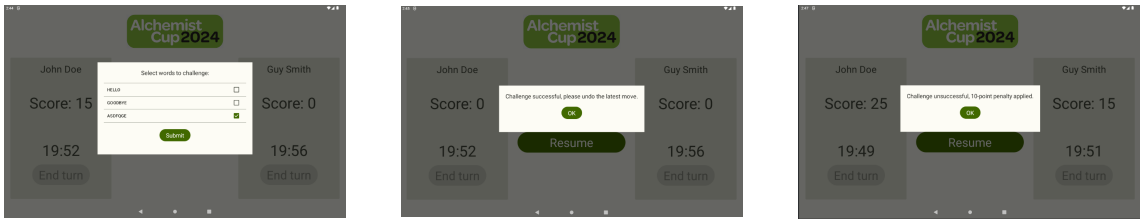


Figure 4.19: Blank tile dialogue

After at least one turn has been played, challenges are enabled, and can be launched by pressing the relevant button. Once a challenge has been launched, it cannot be cancelled, and play is paused until it has been resolved, once again following tournament regulation. The dialogue first presents the players with the list of words formed by the previous move, allowing the challenger to select as many as they wish. Upon submission, the server will resolve the challenge, and display the result to the users and providing them the option to dismiss the dialogue when ready, resuming play. The full challenge workflow is illustrated in Figure 4.20. One potential issue with this functionality is in the unlikely event that a silent failure occurs in the system, causing the list of challengeable words displayed to be incorrect. Since the dialogue can only be dismissed through challenge resolution, the challenger may be forced to challenge a word they do not wish to. In such rare cases, the players should raise this issue with the tournament organisers, who can manually raise the error on the server side, and perform the actual challenge through alternate means.



(a) Challenge selection dialogue (b) Successful challenge (c) Unsuccessful challenge

Figure 4.20: Challenge dialogue

Once the application receives the end of game signal from the server in response to an end turn request, the player timers are frozen, final scores are displayed, and no further interaction is possible until the application is reset to the setup screen for the next match.

As explained in subsection 2.3.1, ensuring the Android application is well-designed is critical not only to keep the codebase maintainable, but to ensure correct functionality is preserved across the various events affecting the activity's state. Jetpack Compose was selected as the framework for app development as the recommended approach to application development, making numerous improvements over older View-based approaches, with extensive community support. Top-level

`ViewModels` were created to manage the UI state for both the setup and match screens, exposing the information down to the UI components as an immutable `StateFlow`, as well as passing down callbacks to enable these components to communicate events back up, implementing unidirectional data flow within the UI layer. Information such as player names and match ID, which are obtained from requests on the setup screen, are propagated to the `MatchViewModel` when constructed as the navigation component transitions across screens. Both the blank tile and challenge dialogues also have state associated with them, and as such sub-`ViewModels` within the `MatchViewModel` were defined, encapsulating the logic for these particular features, with the specific UI state for each propagated down to these composables from the `MatchScreen`.

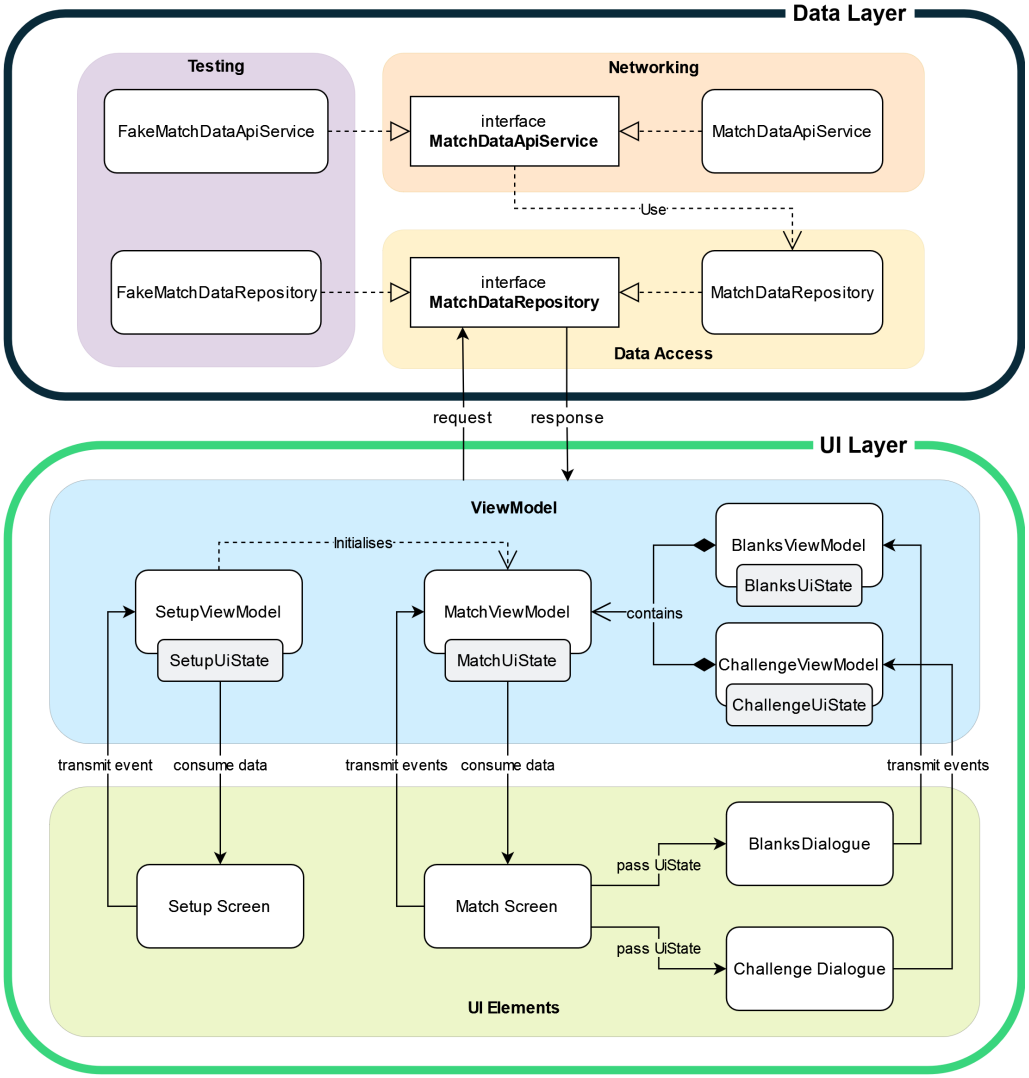


Figure 4.21: Overall Application Architecture

The repository design pattern [168] was used to manage the UI layer's access to data from the HTTP server, which exposes data operations to the `ViewModels`. Although communication with the HTTP server could have been done directly, without the use of a repository, this additional layer helps to organise the codebase, making it much simpler for additional data management solutions such as writing to disk to be included in the future, which could be beneficial to render the application more robust to fatal errors. Dependency injection with an abstract `MatchDataRepository` interface was used to provide `ViewModels` with access to the repository, enabling these `ViewModels` to be unit tested with a "fake" version of the repository returning test-defined data.

The Retrofit2 [169] library, recommended by Android, was used to manage HTTP requests with the server, providing a type-safe, seamless API with support for custom serializers, which enables Kotlin classes to be constructed directly from the HTTP responses, abstracting away the low-level

networking. A generic `ServerResponse<T>` class, matching the generic body-or-error structure of messages from the server was used to standardise how responses from the server were handled, with custom response bodies matching each request to be defined separately. This is illustrated in Listing 4.7, and renders the code highly maintainable. As with the `ViewModels`, the concrete `MatchDataRepository` class is capable of functioning with a "fake" version of the API service to facilitate testing. The structure of the full companion application is summarised in Figure 4.21.

```
interface MatchDataApiService {
    @GET("challenge")
        suspend fun challenge(
            @Query("match_id") matchId: String,
            @Query("turn_number") turnNumber: Int,
            @Query("words") words: List<String>
        ): ServerResponse<ChallengeBody>
}

@Serializable
data class ServerResponse<T>(
    val body: T? = null,
    @SerialName(value = "error")
    private val _error: String? = null
) {
    val success: Boolean
        get() = body != null

    val error: String?
        get() = if (!success && _error == null)
            "Unknown error: response body was null but no error reason was provided"
        else
            _error
}

@Serializable
data class ChallengeBody(
    val successful: Boolean,
    @SerialName(value = "challenger_penalty")
    val challengerPenalty: Int,
    @SerialName(value = "undone_move_score")
    val undoneMoveScore: Int
)
```

Listing 4.7: Example of generic `ServerResponse<T>` class with challenge requests.

Chapter 5

Testing

In order to verify the functionality of the large number of software components making up the overall system implemented in chapter 4, comprehensive testing was required. The design of the code resulted in a decoupled architecture, facilitating this process as the code could be first tested in logical units, prior to larger scale integration and system-wide testing. This chapter further details this process, providing an overview of how functionality was asserted.

5.1 Unit Testing

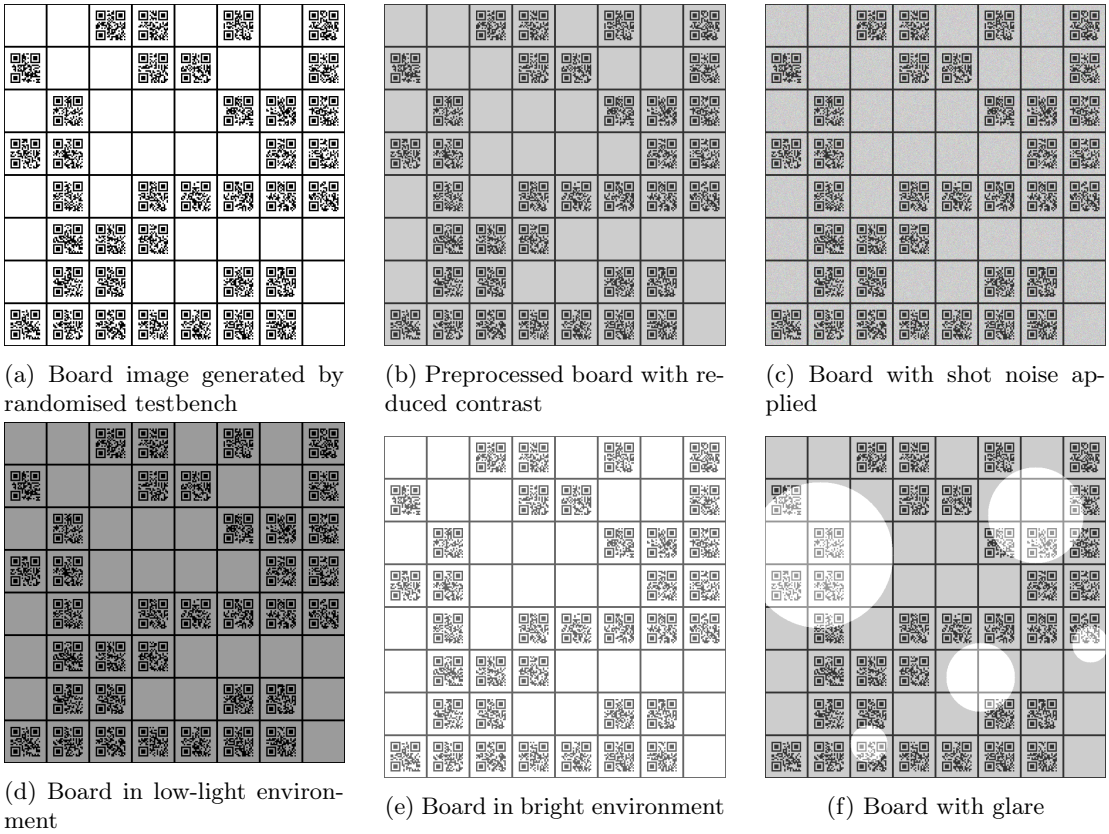
The purpose of unit testing is two-fold, not only verifying the correctness of the overall application at the most granular level, but also to facilitate future development work by providing a record of the expected functionality of each part of the system. As detailed in section 4.3 and section 4.2, various OOP concepts were used to ensure that the sections of the application with more complex business logic were easy to isolate in a unit testing framework. Unit tests were typically written in parallel to code development, allowing implementation bugs to be identified and corrected rapidly over the project duration.

Unfortunately, due to time constraints, not all parts of the application could be unit tested thoroughly, but the functionality of these components was still proven through later testing stages. Explanations for how these components would have been unit tested is still detailed in this section, as well as justification for why providing unit tests of these components was considered less important. In particular, the more networking oriented aspects of the implementation were omitted, and although these could have also been unit tested through the use of mock objects [170], this was considered unnecessary as later integration tests, outlined in section 5.2, were sufficient in ensuring that such behaviour worked as intended. Regardless, this is a limitation of the project, and is discussed further in chapter 6 and section 7.2.

5.1.1 Game State Detection

Both physical, as well as simulated testing was considered to determine the effectiveness of the board and rack state identification pipelines. However, the delays involved in construction of suitable hardware prototypes, as well as the cost associated with capturing a wide range of images which could be used for such tests, rendered it infeasible for such testing to be performed prior to report submission. The viability of the image detection algorithms on real images was asserted in testing on initial prototypes, as shown in section 4.1, and as such, simulated testing was chosen, allowing a much wider, more robust data set to be tested.

The randomised benchmarks used for performance analysis of the board and rack algorithms outlined in subsection 4.1.4 and subsection 4.1.6 provided valuable starting points to perform further simulated testing. A variety of techniques were considered to augment the generated images to contain visual artefacts the system was expected to come across in production:



(a) Board image generated by randomised testbench

(b) Preprocessed board with reduced contrast

(c) Board with shot noise applied

(d) Board in low-light environment

(e) Board in bright environment

(f) Board with glare

Figure 5.1: Effects of various artificial visual artefacts on test bench images

Camera Noise refers to random variations in the image signal which arise from the capture of the image with a physical sensor. Although there exist models which aim to accurately simulate camera noise through various, sensor-specific parameters [171] [172], the applications of such models for this specific use case is especially challenging due to a number of reasons. The IMX477 sensor used in the HQC module uses a number of modern technologies to achieve a high Signal-to-Noise Ratio (SNR) [173], with the downsampling performed on the raw image outlined in section 4.1 further improving this performance. Additionally, the Raspberry Pi hardware applies a number of denoising algorithms to the images via the libcamera technology stack used to pass images from the camera to the user [174], meaning that significant preprocessing is already done on the images received by the image processing pipeline implemented.

As such, the use of a complex model representing all aspects of camera noise was deemed unsuitable; instead, only shot noise, which increases proportionally to exposure time, was modelled to represent the changes in image quality under different lighting conditions. This type of noise, caused by statistical quantum fluctuations in the number of photons sensed, follows a Poisson distribution, and was simulated using `np.random.poisson()`, as illustrated in Figure 5.1c.

Brightness is another important aspect which could vary under external conditions, affecting the overall quality of detection accuracy. This was changed uniformly across the image using built-in OpenCV functionality [175]. The effects of extreme low and high brightness are shown in Figure 5.1d and Figure 5.1e respectively.

Glare could be caused due to direct overhead light sources shining directly into the lens. Although such sources of light would be blocked for a particular square when a tile is placed, the light streaming in from adjacent, empty tiles could still cause these defects. An aggressive approach was taken to simulate these, in which the size and position of the glare effect would be randomly applied over different segments of the image, likely creating images which were

not realistically feasible, but providing a more robust test set. An additional restriction was added to ensure that glare effects could not overlap, as for particularly high intensities these could result in rendering the intersection completely white, erasing any data contained. The application of the glare effect on a test image can be seen in Figure 5.1f.

Rotations were already considered in the existing rack benchmark, and given that all tiles placed on the board must be oriented in the same direction, as per WESPA tournament guidelines (see Appendix E), these were not added to board images.

Blur can occur in the system when players are placing tiles on the rack mid-capture, although it is fair to assume that given the rapid processing rate, non-moving images of tiles places will always be captured eventually. As such, additional blurring of specific tiles was not added, although the tested images will include the blur from the preprocessing stages of the image pipeline.

Before artefacts were applied, the overall contrast was reduced to more closely resemble the colouration of QR codes from real images, as shown in Figure 5.1b. The randomised benchmark pipeline was then updated to include the artefacts listed above, randomly selecting the values they are parameterised in to ensure a wide range of test images were considered. These were then passed into the full detection algorithms used by the sensor, including the de-noising and binarization steps, from which accuracy metrics identical to those used in subsection 4.1.4 and subsection 4.1.6 for the board and rack respectively could be computed.

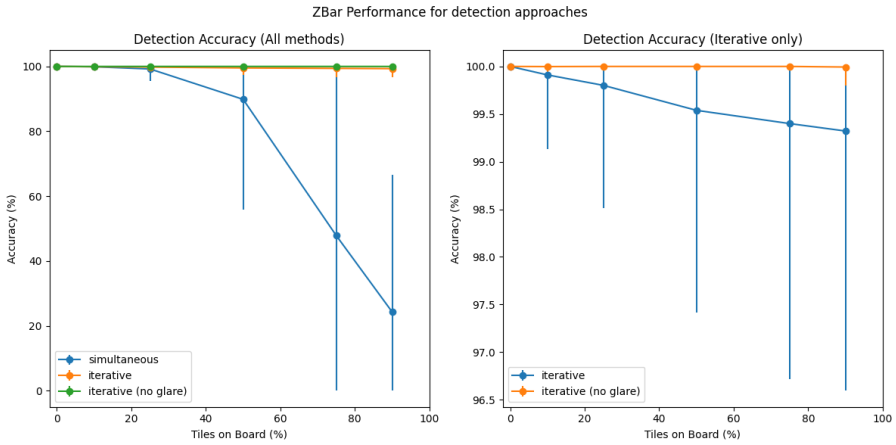
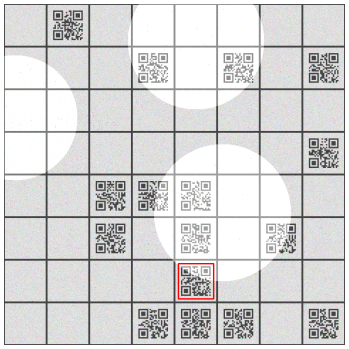


Figure 5.2: Performance of board detection algorithms on randomised test bench (1000 iterations). Iterative algorithms plotted on separate axes to better demonstrate accuracy.

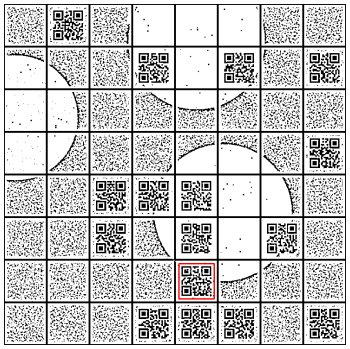
The performance of the board detection algorithm is shown in Figure 5.2, in which the iterative approach achieved above a 99% detection accuracy for all images. Unfortunately, this was below the 99.9% target outlined in section 1.1, and as such, further investigation was performed into identifying cases which the algorithm found particularly problematic. Manual analysis of the problematic cases revealed that sub-100% detection accuracy was commonly obtained on images in which the edge of a glare effect intersected with a QR code's finder pattern, as shown in Figure 5.3a, which after passing through the image preprocessing stage shown in Figure 5.3b resulted in a QR code which was unsuitable for detection. This explained why the performance of the detection algorithm dropped as the fraction of tiles on the board increased, as it was more probable for such a problematic intersection to occur.

Modifications to the adaptive thresholding parameters used in the preprocessing algorithm resulted in images which were successfully detected with 100% accuracy, as shown in Figure 5.3c, but these modified algorithms were found to consistently perform worse on the randomised test bench by up to 0.7%. It is worth noting that the artefacts which cause these detection issues are rather unrealistic, as glare would most likely affect empty sections of the board rather than squares which have direct overhead light blocked by a tile, suggesting that the existing image detection pipeline will likely perform better in production. Tests were repeated on a pipeline without glare,

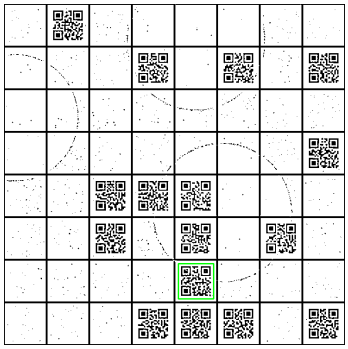
also shown in Figure 5.2, in which a detection accuracy >99.99% for the full confidence interval, lending further credence to the hypotheses above.



(a) Generated board with sub-100% detection accuracy



(b) Preprocessed image passed to QR detection stage (C=2)



(c) Modified preprocessed image (C=6)

Figure 5.3: Common problematic test-case where glare interferes with finder pattern. Problematic QR code is outlined in red (undetected) or green (detected).

Testing was repeated for the rack in a similar fashion, with results plotted in Figure 5.4. The multi-orientation approach selected in subsection 4.1.6 showed very strong results, consistently obtaining an F1-score >0.995. The narrowing of confidence interval widths as the number of tiles increase is expected, as a mistake in decoding a particular QR code on a rack with 1 tile is penalized more harshly than the same mistake on a rack with 7 tiles, resulting in scores of 0 and 0.923 respectively, leading to much larger standard deviations between testcases. In some cases, the standard approach, which only performs QR code detection on the un-rotated image, appeared to perform almost identically to the multi-orientation approach, but this was likely due to variance in the difficulty of the testcases generated by the test bench, as the results obtained in subsection 4.1.6 proved that the multi-orientation approach was more resilient to the rotation of tiles. However, the similarities in performance of the two approaches when facing simulated artefacts suggested that these visual errors may have a more significant impact on detection accuracy than rotation.

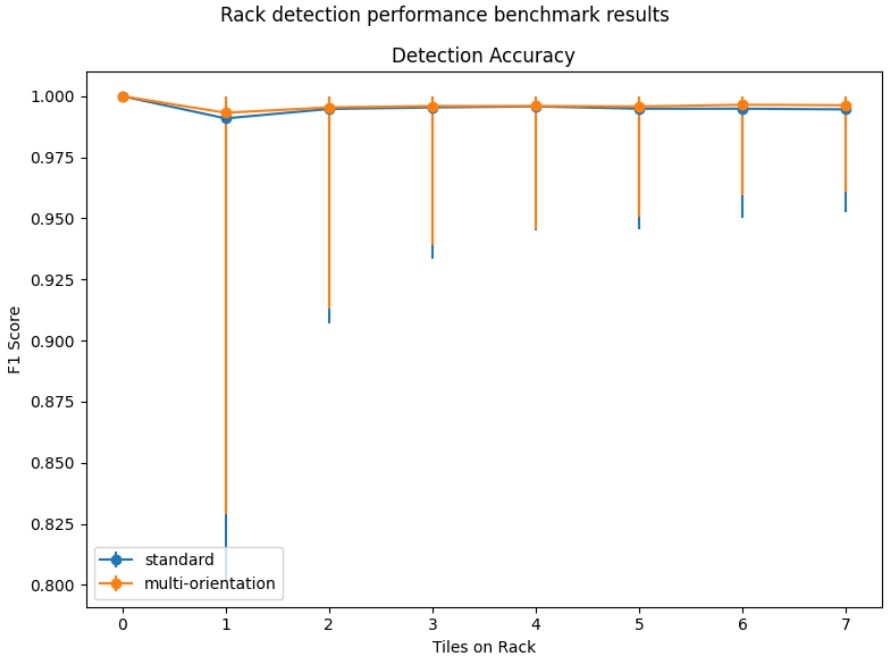


Figure 5.4: Performance of rack detection algorithms on randomised test bench (10000 iterations)

Manual inspection of the failing cases once again confirmed that issues appeared to be primarily caused by intersections between glare effects and QR codes, identical to the case of the boards.

This may explain why the confidence interval of the data points measured appear to remain fixed between different tile numbers, as the increased probability of glare negatively impacting QR code detection due to an intersection offsets the reduction in F1-score penalty, although this may also simply be due to statistical variance. It is also worth noting that given the smaller size of the rack images, such overlaps are more likely to occur than in the board test bench, meaning that the results obtained for the performance of the rack detection algorithm are overly pessimistic with respect to the board.

Nonetheless, the detection accuracy exhibited by both algorithms, even when facing extreme, unrealistic distortions provides a strong level of confidence that not only should it be reasonable to expect such a system in production to achieve the target accuracy defined in section 1.1 for a wide range of lighting environments, but further adjustments could be made to the preprocessing algorithms to fit the model to the tournament environment, achieving even greater results. Although further testing on the hardware prototype, once ready, would also be valuable in verifying the results obtained, the range and severity of the simulated artefacts, combined with the positive results of the initial feasibility tests conducted in section 4.1 make it highly likely for accuracies measured in this section to be at the very least representative of, if not provide a lower bound for, the performance of the sensor implementations in production.

5.1.2 Game Logic

Ensuring the package encoding move validity, score calculations and challenge requests implemented in subsection 4.2.1 worked correctly was vital, and could be accomplished relatively easily using Python's `unittest` framework [176]. The functionality of each of the 4 core classes were tested in separate files, each grouping tests into classes verifying the functionality of a particular method, with each class containing multiple methods testing different scenarios. An example demonstrating the structure of these tests is shown in listing Listing 5.1.

```
1 class TestGetScore(unittest.TestCase):
2     def test_bingo(self):
3         board = Board()
4         # Also tests stacking multipliers (2x Word with 2x Letter)
5         move1 = Move.fromstr('8H ABANDON')
6         self.assertTrue(board.apply_move(move1))
7         self.assertEqual(board.get_score(), 74)
```

Listing 5.1: Example unit test verifying the correctness of score calculation for 7-letter plays

A total of 42 test cases were devised, all passing, in which Woogles was used to provide reference outputs when relevant, which combined with the functional, modular implementation of the code provide a high degree of confidence that all logic was implemented correctly. The majority of the tests for the low-level `BoardPos`, `Tile` and `Move` classes were to verify that the abstractions they provided to simplify the implementation of the top-level `Board` worked correctly, and as such are not particularly relevant, but the top-level tests encapsulating the game logic are summarised below:

TestApplyMove focuses on the ability of the system to correctly detect moves which do not follow standard Scrabble game rules, and reports an error without changing its internal state. Its individual test cases considered the following cases:

- Moves which are not valid in and of themselves, such as when all tiles are not along a principal horizontal or vertical axis, are detected as errors
- Moves which are valid, but not containing an anchor point connecting them to an existing tile, or starting square when the board is empty, are detected as errors.
- Moves which are valid and anchored, but contain a discontinuity resulting in isolated tiles, are detected as errors.

- Moves which are valid and anchored, but contain a discontinuity resulting in formation of multiple words in the same direction of play, as illustrated in Figure 5.5a, are detected as errors.
- Moves which are continuous, valid and anchored are applied successfully.
- Moves which are discontinuous, valid and anchored, forming a single word in the direction of play are applied successfully

TestGetScore verifies the correctness of the score calculation function under a number of scenarios:

- Bingos, in which a player places all 7 tiles, correctly include a 50-point bonus.
- Moves which cover multiple premium tiles correctly compound their multipliers, such as the combination of word and letter multipliers or multiple word multipliers.
- Moves which form words with existing tiles on premium squares do not include these premiums in their score calculation.
- Moves which form words in multiple directions correctly apply square premiums in each word in which they are contained, as shown in Figure 5.5b.
- Blank tiles are always counted as 0 points, irrespective of the letter which they represent.

TestGetChallengeWords verifies that the correct words are presented when a challenge is issued, with individual test cases checking the following behaviours:

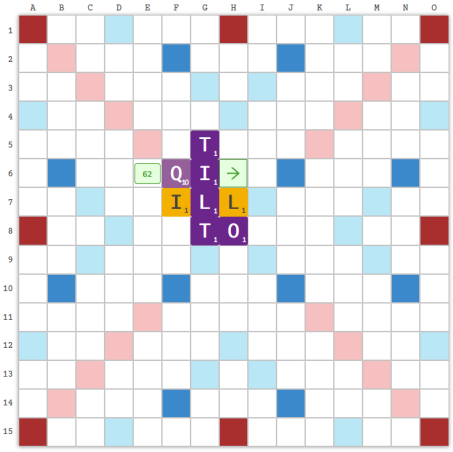
- Challenge words are always formed from the most recently applied move.
- Words are correctly formed in play order, either top-to-bottom or left-to-right.
- All words formed by a move, in both the horizontal and vertical direction, are correctly returned.
- Only unique words formed by a move are provided, ensuring that in cases where the same move is formed multiple times, as shown in Figure 5.5b, each word can only be challenged once.
- Moves containing blank tiles must have had their value specified prior to requesting challenge words, returning an error if this has not been done.
- Words formed from identified blank tiles are treated identically to words formed with the tile matching the blank tile's declared value.

Although testing approaches relying on fuzzing, which randomises data inputs to automatically detect edge cases, were considered to further assert the correctness of the game logic. However, ensuring such a fuzzer was correctly parameterised to adequately test valid play patterns essentially required the game logic to be re-encoded into the fuzzer, as a pure randomised approach would typically result in moves which can be trivially detected as invalid from lower-level logic. The development effort required for such a system was deemed excessive in comparison to the benefits such a fuzzer would provide, particularly when the exhaustive unit testing above, combined with the regularity exploited in the implementation, leave very little room for undetected bugs.

5.1.3 Delta Resolution

The correctness of the delta resolution mechanisms was verified in a similar manner to the game logic package described above. Once again relying on the `unittest` framework, the intra-turn delta resolution mechanisms of both the rack and board delta handlers were first verified independently, along with the `TileBag`, followed by the top-level delta resolution of the overall `GameState`. A total of 26 tests covering a wide range of success and error paths were devised, with the test distribution

(a) Illegal move along a single direction of play, forming multiple words which are all anchored



(b) Move forming the word "QI" in both horizontally and vertically on a triple letter premium, resulting in a 62-point move

Figure 5.5: Board configurations illustrating various edge cases in move application, scoring or challenge word identification. The play triggering the edge case is shown in semi-transparent colouring.

among these different components summarised in Table 5.1. A brief overview of each unit test is available in Appendix D.

Class	Total Tests
`TileBag`	6
Intra-turn rack	9
Intra-turn board	7
Top-level resolution	4

Table 5.1: Test case distribution for management of server-side game state

Not all aspects of the delta resolution protocol were testable, such as verifying that warning logs were emitted for low-confidence snapshots, as these were not reflected in the return types of the methods which are used to interact with a `GameState` class. However, this was not considered particularly problematic, as not only were such features implemented with extremely simple logic, making them less prone to error, but they were non-essential for the correct functionality of the system. Later integration tests, described in section 5.2 verified that these logs were produced when appropriate.

Fuzzing has similar limitations in the testing of delta resolution as it does for the game logic package, once again requiring game logic to be codified to adequately constrain the generation of random inputs. The wide range of scenarios covered by the tests above, combined with the well-structured implementation of delta resolution, provide a high level of confidence in the error detection capabilities of the server.

5.1.4 Connection Management

Verifying the server and clients' capabilities of correctly initiating connections and responding adequately to network errors at all stages of sensor allocation in the context of unit tests was considered problematic. Although this could have been accomplished using mock objects [170] within the `unittest` framework, as mentioned previously, passing in fake network resources to either class representing the other end of communication, it would require substantial development effort. Changes to the existing client implementation would have also been necessary, as the `Reader` and `Writer` objects associated with the socket connection are created and owned by the client to facilitate reconnection logic, unlike the server which is able to tie the lifetime of a `SocketHandler` to the lifetime of a particular connection.

Due to these issues, unit testing for these components were not produced, and instead the correctness and robustness of connection management logic was analysed thoroughly in later integration tests, discussed in subsection 5.2.1.

5.1.5 Companion Application

There are two core unit-testing methods in Android development, "standard" unit testing in which the logic of components such as the `ViewModel` are verified, and instrumentation tests where elements of the UI itself are verified programmatically through visibility or value checks [177]. Testing in Android has substantial third-party support, relying on the JUnit testing framework [178], to provide developers with a streamlined approach to including both types of tests in their project.

The architecture of the companion application outlined in section 4.3 renders the addition of such tests relatively easily, as fake data sources can be provided to enable independent testing of individual classes. Although the addition of isolated unit testing was desirable, ultimately these were not included due to time constraints, with functionality of the application asserted in integration testing described in subsection 5.2.2. As such, the primary value of such tests would not be to provide confidence in the correctness of the application, but more so to improve the maintainability of the code by effectively documenting the expected behaviours of the application, rendering it simpler to verify that future changes do not degrade the functionality of the existing feature set. The impact of this decision on the project's objectives is analysed in subsection 6.2.2, and is considered for future work in section 7.2.

5.2 Integration Testing

Integration testing typically refers to the testing of multiple software components as a group to verify their interoperability. In this project, such testing was split into two major sections: interactions between the companion application and server, and interactions between the sensor clients and server. These tests were also used to ensure the functionality of components which were not thoroughly unit tested, providing a high level of confidence in the correctness of these features.

5.2.1 Sensor Communication

Given that sensor communication was handled through stateful, two-way TCP sockets, it was important to test the various ways in which these connections could fail, ensuring that the server handled each case appropriately. To facilitate this process, modified board and rack clients were created with the goal of simulating various network errors which could be identified by the server. These clients were always run on the same machine as the server, passing fixed, manually assigned MAC addresses to the server upon registration to ensure that these simulated errors were experienced reliably, unaffected by any potential variance in network connectivity. The types of errors simulated by these clients are summarised in Table 5.2, and were intended to exhaustively encapsulate the issues which could be realistically faced in production.

ID	Simulated Behaviour	Representative Scenario
1	Closing the connection	Network outage
2	Delaying heartbeats by $r \in [1, 10]$s	Sensor experiencing heavy CPU load
3	Delaying all communication by $r \in [100, 2000]$ms	Network congestion
4	Crashing and rebooting the client	Sensor experiencing Power outage

Table 5.2: Fake client error simulation capabilities

One board client and two rack clients were run, initially enabling error generating behaviours for only one sensor at a time, exhaustively testing server response to these scenarios for each possible sensor role (`Board`, `Rack_P1`, `Rack_P2`) at the different stages of sensor allocation. Requests to start matches were sent to the server at regular intervals, enabling client role allocation and verifying when this behaviour could take place, testing the full functionality of the `ConnectionHandler`. The

general expected server response for each error are detailed in Table 5.3, with the client attempting to re-initiate the connection whenever disconnected. This initial stage of testing verified that these responses occurred as expected, as well as confirmed that the `ConnectionHandler` performed as follows depending on whether sensors were assigned to a match:

Pre match assignment : Lost connections are removed from the pool of available connections, resulting in insufficient sensors for match allocation. When reconnecting, clients are once again returned to the pool of available connections.

Post match assignment : The relevant `SocketHandler` reflects the disconnection in their state. Any communication attempts with the disconnected `SocketHandler` result in an error. Upon reconnection, the old `SocketHandler` is replaced by the new instance, and the sensor is immediately provided its corresponding `DataFeed`. The server also communicates the current state of the board

ID	Expected Behaviour
1	Connection loss detected by server
2	Delays $\geq$ 5s result in forcibly disconnecting client
3	Delays $\geq$ 1000ms result in forcibly disconnecting client
4	Connection loss detected by server

Table 5.3: Server responses to simulated errors. Note that timings for forced disconnects are approximate due to non-deterministic socket communication timing.

Once the standalone correctness for the management of each sensor was asserted, further testing was conducted in which each sensor could randomly exhibit errors, which notably revealed a subtle bug in the `ConnectionHandler`'s match allocation functionality. When assigning the sensors to a match, the communications with each sensor are done in parallel, resuming execution once all sensors have replied. These responses typically came roughly simultaneously, however, in the particular edge case when one sensor experienced delays in responding, whilst another sensor lost connection after having acknowledged the assignment request, the `ConnectionHandler` did not atomically check that all sensors were still connected prior to confirming the allocation following successful responses from all, causing a disconnected sensor to be allocated to a match. Although such an issue would still be reported to tournament organisers and players once play commences, the possibility of the server allocating a disconnected sensor to a match was undesirable, and an additional check verifying that all `SocketHandler`s remained connected after all match allocation responses were obtained was added to rectify this issue. All other transactions with the sensors were always performed independently of one another, and as such did not suffer from this issue.

Not only did these tests verify the functionality of the server's connection handling logic, which as discussed in subsection 5.1.4 did not have separate unit tests, but they also provided confidence that the general communication protocol devised performed as intended; robust to a wide range of potential communication errors. This testing also asserted the base sensor client's capabilities of automatically restoring server connectivity following an error. As justified in section 3.3, it was assumed that the server would not lose connectivity, but the connection could still be closed from the server side, and responses from the server never be received if the sensor looses connectivity after sending a request, upon which the client's internal state would be cleaned up and a fresh connection would be established. The client's retry limit, in which it would automatically shut down after 5 unsuccessful attempts at initiating a connection with the server, was manually verified by shutting the server down. Larger scale randomised testing, in which larger numbers of fake sensors would be simulated with multiple simultaneous matches was considered, but ultimately was not performed, as almost no new logic would be tested given that concurrent matches were handled by the same functions, making such a test present limited value in terms of the time investment required to perform it.

5.2.2 Companion Application

In order to verify the functionality of the companion application, extensive manual testing was performed, often using a dummy HTTP server implementation which returned standard test data

to the application. Different errors could also be simulated from the server side, verifying that these were correctly displayed by the tablet. Given the high-level, stateless nature of the communication between the app and the server, there was very little scope for errors, with requests either timing out or returning with a success or failure. Integration testing on the application mimicked the standard workflow which would be performed on it in a production environment, exhaustively testing both the success and failure paths for each feature, including:

- Match setup
- Starting game
- Pausing game
- Ending turn
- Submitting blank values
- Challenging words
- Reflecting end of game status

In all cases, it was verified that the UI state matched what was expected, and that the server received the appropriate data from the application.

The majority of testing was performed using Android Studio's integrated emulators, which allowed the successive versions of the application to be rapidly tested on a device matching the screen specifications of the 10" tablet which the sponsor had selected for tournament use. However, these tests were also repeated on the target device with a release version of the software, which replicated the emulator performance. The application architecture outlined in section 4.3 rendered it robust to errors, with several higher order helper functions capturing regularities in processing networking requests, ensuring errors are always handled in the same way; as such, testing was relatively straightforward and yielded no errors.

In general, server-side code for handling events from the companion application were rather simple, simply notifying the game state indexed by the match ID provided in the request of the event, or triggering match allocation logic in the TCP server's `ConnectionHandler`, leaving very little room for implementation bugs. User input is always sanitised on the client side prior to server communication, meaning minimal checks on such inputs needed to be performed. As such, the HTTP server itself was not tested, but its functionality was verified in the top-level system testing described in section 5.3.

5.3 System Testing

Ideally, system-wide testing would be used to ensure the complete functionality of the overall system. Unfortunately, due to the lack of a complete hardware prototype, full system-wide testing could not be performed, but the fake clients developed in subsection 5.2.1 were used to simulate sensors, allowing the full workflow of match setup and gameplay to be tested end-to-end, including events such as blank tile value specification and challenges. Although such testing was not particularly beneficial in terms of bug identification, it was helpful to provided added confidence that the entire system works effectively, and no major issues were experienced across multiple trial runs.

The error reporting capabilities of the server, particularly in regard to delta resolution, were also verified, ensuring that suitable messages were displayed to provide tournament organisers with an accurate picture of the system through its logs. The fake sensors were configured to send various types of invalid information, following the unit test cases used in subsection 5.1.3.

Chapter 6

Evaluation

Although many aspects of the system's design and implementation have already been critically evaluated in chapter 3, chapter 4 and chapter 5, this chapter offers a high-level, comprehensive analysis of the solution developed, focusing on how well it aligns with the objectives specified in section 1.1.

6.1 Functional Requirements

Overall, the solution presented in this report meets all of the functional requirements defined in section 1.1 to a high degree of satisfaction, providing an effective, robust system fit for widespread commercial use in digitizing tournament Scrabble matches. All of the relevant criterion are evaluated in-depth below.

6.1.1 WESPA Conforming

When relevant, the system designed encodes both the standard Scrabble game rules, and WESPA tournament guidelines included in Appendix E, although assumes the use of 5-point penalty as challenge rule [167], which despite being the preferred international norm may not be used in all WESPA conforming tournaments. The impact of these alternative rulesets is discussed in subsection 6.1.5. Although a finalised hardware prototype is not available at the time of writing, requirements on the tiles and board should be met, with the model board using identical ridging to boards used in previous Alchemist Cup tournaments [179]. QR codes can be attached to tiles uniformly using stickers, or potentially etched employing identical manufacturing techniques to those used to print the letter and point value. The timekeeping capabilities of the companion application was also carefully designed to match the WESPA specification, ensuring the functionality of the timer is preserved across any potential networking errors. Although the design of the board and racks will likely prevent players from viewing the number of tiles on their opponent's rack, the companion application's error communication mechanisms would inform the players of an overdraw, which is the reason behind this requirement.

Some aspects of the tournament rules were not codified, including:

Rechallenging Players may typically request the re-adjudication of a challenge. Given that adjudication is performed programmatically by the server, it is effectively impossible for misadjudication to take place, rendering rechallenging unnecessary. If necessary, players may always suspend the timer to request manual adjudication.

Consecutive Zero Scores Matches typically end upon a player "playing out", in which both their rack and the tile bag are devoid of tiles. However, WESPA also stipulate that the game also ends after six consecutive turns scoring zero from any combination of passes, exchanges and successful challenges, upon which each player's final score is reduced by the total value of

tiles on their rack. Logic supporting this scenario was not implemented, as it is an extremely rare occurrence, and can be handled manually with relative ease.

Overtime Forfeiture A game can also end when a player goes over the assigned game time by 10 minutes, resulting in an immediate loss. This is another edge case which is not covered by the system, and according to the project sponsor never happens at the elite level of play.

Whilst the integration of these additional features is considered in section 7.2 as future work, the value they provide is quite minimal, particularly in the context of Alchemist Cup 2024. As such, the solution provided was deemed perfectly adequate in its abilities to follow, and when necessary enforce, the official international tournament guidelines.

6.1.2 Game State

The system presented in this report offer a novel and effective solution to capture all relevant game state information and manage it from a central source of truth. However, the latency and detection accuracy of the solution must first be quantified to fully evaluate its effectiveness with respect to the project goals.

Given the distributed nature of the system, the delay between the server state and each component must first be estimated. The server will likely be based in Jakarta, the city hosting Alchemist Cup 2024, which has a number of availability zones provided by AWS [180] to minimise the distance travelled by the signal between the sensors and server, which given the small packet sizes should take no more than 15 ms for a round trip. Using this assumption, the latency between the server state and each component can be estimated as follows:

Board Latency Using ArduCam's performance guarantees for the Camarray, a delay of up to 100 ms can be expected between the physical state of the board and the image passed into the detection pipeline. A further 150 ms is required for the initial preprocessing, and based on the performances measured in subsection 4.1.4, no more than 175 ms should be needed to handle QR code detection. Factoring in the 15 ms delay for communication, this provides an upper bound of 440 ms as the maximum latency between the server-side board state and the board itself.

Rack Latency Unlike the board, the rack only needs to interface with a single HQC module, which as measured in section 4.1 operates at 105 fps, rendering the delay between the physical rack state and the image captured negligible. Slightly less time than the board is required for preprocessing, as the required image size is smaller, requiring 125 ms, and as measured in subsection 4.1.6, QR detection should take no more than 100 ms. As such, the 100ms throttling discussed in subsection 4.2.2 for rack updates would likely not be required. Combined with the 15 ms network delay, an upper bound of 240 ms can be derived as the maximum delay.

App Latency The companion app performs minimal data processing, meaning that network requests initiated by the user can be viewed as occurring instantaneously. Therefore, only the communication delay needs to be considered for this case. Given the additional overhead of HTTP, which requires an additional round trip setting up the connection, a maximum delay of 30 ms is reasonable.

Woogles Latency Unfortunately, the Woogles servers are based in Ohio, a considerable, 16000 km away from Jakarta. This renders the latency of requests between the server and API challenging to estimate accurately, particularly as the path in which packets between the two are routed is non-deterministic. The bulk of this communication will likely take place through fiber-optic cables, in which the speed of light is reduced by approximately 31% due to the 1.4475 refractive index of the silica core [181], resulting in a speed of light of $\sim 2.07 \cdot 10^8$ ms^{-1}. This leads to an absolute lower bound of 77 ms for photons to travel between the two servers, resulting in a round-trip time of 154 ms.

Given that the Woogles API is HTTP-based, a second round trip is required setting up the connection, resulting in an expensive 308 ms penalty, without accounting for inefficiencies in

network infrastructure. As both servers are hosted in data centres, it is reasonable to assume that high-quality network infrastructure will be available for both; therefore, it is fair to estimate these inefficiencies as no more than an additional 50% of the theoretical optimum, suggesting a maximum total latency of 462 ms. However, connection pooling can be used when making HTTP requests to Woogles from the server, in which existing connections are reused for multiple requests, avoiding the additional round trip overhead of setting up the connection [182]. Given that connections to the Woogles API would be formed when setting up the games, halving the latency for match updates to 231 ms. Note that waiting for the server response, included in the round trip calculation, is not strictly necessary when pushing data, although information from the Woogles server will be required to update the UI state on the broadcast, making it reasonable to include the full round trip time for the purposes of estimating delays between the match state and livestreamed match.

Using the worst-case latency estimations derived above, it is now possible to determine the maximum delay between the true game state, and its representations on both the Alchemist and Woogles servers. As board updates are published at least every 440 ms, with these deltas themselves containing data which is 440 ms old, the match data server's board state will be at most 880 ms behind. Repeating similar logic for the racks, the server state will be no more than 480 ms behind. Both of these values are within the 1 s maximum delay imposed in section 1.1, meaning that delta resolution when receiving the end of turn signal from the companion application should work effectively. Given the compute resources available to the server and $O(1)$ complexity of the delta resolution mechanism implemented in section 4.2, the processing time required for this protocol is negligible, and therefore not included in the latency estimation. Updates to the Woogles API are typically triggered by events transmitted from the companion application, requiring approximately 260 ms to propagate from the app to what is shown on the broadcast. Due to the staleness of match data on the server, this could in rare cases result in board data slightly older than a second being presented on the livestream, but provided that the delta resolution protocol did not identify any issues, it is highly unlikely that this state is not synchronised with the actual board at the end of a given turn.

Based on the conclusion obtained in subsection 5.1.1, both the board and rack detection algorithms are able to achieve incredibly high detection accuracy in a wide range of lighting environments, and are almost certainly capable of achieving the 99.9% target set in section 1.1. Given this high performance, the use of micro QR codes, or custom encoding schemes enhancing the error correction capacity of the QR code as mentioned in subsection 2.2.4, was not considered necessary, but is still highlighted as potential further research in section 7.2.

In its current state, the delta resolution algorithm is capable of handling the majority of sensor errors as they are received, typically ignoring them in favour of feasible data, which should further improve the reliability of the system. One limitation of this algorithm is that unresolvable errors simply cause the system to "fail-fast", leaving the match in an invalid state, rather than determining the most likely correct state, which in some cases could be suboptimal, particularly if an erroneous delta is received immediately prior to an end of turn event. Although it would still be critical for the system to notify the tournament organisers and players of this potential error, if the server has a high degree of confidence in the true play emitting a warning and proceeding could help streamline the management of concurrent matches. Generally, error correction was intentionally left out of the scope of this project, as such mechanisms can increase the likelihood of the server continuing operation on an incorrect game state when compared to the "fail-fast" approach used, but is discussed as future work in section 7.2 in the context of error recovery as a whole.

Nonetheless, the solution developed in this project presents a highly accurate approach with acceptable latencies in capturing all relevant game state information, and was considered to have fulfilled the core functionality requirements.

6.1.3 Player Experience

As already discussed in subsection 6.1.1, although the hardware prototypes will meet WESPA's equipment preferences, the overall solution does have some additional impacts on the player experience. The comparison between the QR and OCR-based approaches in subsection 3.2.3 highlighted

the potential visual discomfort caused by the selected approach as a result of the thicker boards and racks, and internal lighting they will contain. The colouration of the squares which will likely be accomplished through the use of printed acetate, indicating the different types of premiums, may also be slightly different to what players are used to, although this is a feature of boards which is notably not standardised across international play. Regardless, these effects are not particularly significant, particularly for this project which aims to demonstrate the feasibility of an intelligent scrabble system, rather than the creation of polished hardware for commercial use.

Elements of the system which impose potential behavioural restrictions on the players are more relevant when evaluating whether the solution achieves minimal disruption. The large 10^8 state space, rendering QR code memorisation infeasible, combined with the difficulty of manually deciphering the relatively small codes on the underside of the tiles without closer inspection, ensures that players should not need to change how they interact with the tiles such that their opponent is unable to gain any additional information. The core difference for players will be the interaction with the companion application, rather than the more common digital chess clocks used. Although the use of smart device digital timers is permitted by WESPA, integrating score and challenge capabilities into such a device is novel. The intuitiveness of the user interface presented should make it relatively straightforward for players to switch to this new method, yet players using this system for the first time should be briefed on the new procedure, as well as informed of the possibility of error, and how this would be communicated to them via the application. When operating as expected, this intelligent Scrabble system offers a number of improvements to the tournament player experience, as manual score calculations, and leaving the table for challenge adjudications were common pain points for participants highlighted by the tournament sponsor, both of which are now handled in self-contained workflows within the companion application. Should an error be encountered at any time during the match, players should be effectively notified by the application, essentially rendering the system equivalent to the current, non-intelligent tournament setup commonly used, albeit with a slightly different timer.

One potential negative impact of the system proposed is a result of the 880 ms delay between the board and server-side states. Although this is not a particularly long time in regard to human perception, should a competitor end their turn the instant they place their final tile it is possible for the system to not yet have access to the delta containing this final tile, resulting in an error in delta resolution. However, such behaviour is atypical of tournament participants, who tend to use the same hand with which they place tiles on the board to end their turn, meaning that instructing the players to avoid rapidly ending their turn should not significantly affect their play patterns. Another option to avoid this issue could be to update the companion application to delay the end-of-turn request sent by a player by a few hundred milliseconds, providing additional time for the board delta to be published to the server, although the introduction of such arbitrary timeout values represents undesirable code smell [183].

6.1.4 Cost and Production

The cost estimation provided in subsection 3.2.3 remained relatively accurate, although the switch to the Camarray HAT discussed in subsection 4.1.5 combined with the sourcing difficulties mentioned resulted in a £50 increase for a total cost of £530 for the sensor hardware and manufacturing, which was slightly over the target budget. However, it is worth noting that bulk purchasing of components for the tournament, as well as sourcing directly from Asia, will likely lead to price reductions, potentially bringing cost back under £500. The sourcing of a tablet for each setup would also be necessary, with a suitable candidate which was used in development costing £100, although any existing Android tablets with similar screen dimensions would be adequate. The renting of an appropriate AWS EC2 instance would also be required, which based on current pricing should cost no more than $50 USD for the entire duration of the tournament [184]. Overall, the costs required for the manufacturing of the system proposed are perfectly reasonable in the context of existing products described in section 2.1, in line with the price of DGT chessboards despite the significant additional complexity demanded by Scrabble, and over 37 times cheaper than the RFID-based MSI Smart Scrabble board.

6.1.5 Word Challenges

As mentioned in subsection 6.1.1, challenges for the purposes of Alchemist Cup are fully supported. It is also worth noting that the addition of alternative, WESPA-accepted challenge penalties could be accomplished relatively easily, requiring only some minor changes to the companion application and HTTP server. Although considered out of scope for this project, this possibility highlights the extensibility of the implementation, and could enable the wider adoption of the system in the competitive Scrabble community.

6.2 Non-Functional Requirements

Although deemed non-essential by the project sponsor, the system's capabilities of cleanly reporting errors, and ensuring future development is possible through extensible code is also of significant importance in evaluating the success of the solution developed.

6.2.1 Error Reporting

Although the system has been devised to be robust to a wide range of potential hardware and networking errors, and in many cases can recover from these, ensuring that both the tournament organisers and players were well aware of any issues is of critical importance to ensure both a smooth broadcasting of the tournament, and minimal impact on the quality of the match in the case of an error. In the case of the players, the server indicates the existence of an error when responding to client requests, typically providing a generic error message informing the users of its presence, without providing unnecessary technical details, as it is not the players' responsibility to resolve them. More importantly, the message explains that scoring and challenge functionality will be disabled for the rest of the match, which is the only key information that the players need to know.

The match data server employs extensive use of Python's logging module [185] to provide oversight into its operations. All logs use a common format, providing the timestamp of the log, severity, originating class, and if relevant, match ID prior to the actual message, as shown in Listing 6.1, allowing relevant logs to be easily filtered for using standard command line tooling such as `less` or `grep`. Logs with error-level severity were used as sparingly as possible to avoid flooding the terminal with unnecessary information, and precisely described the issue, rendering it incredibly easy for the root cause of errors to be determined, even while the server is running.

```
2023-06-20 17:28:58,355 [ERROR] [ConnectionHandler] [ExampleID] 0 available
↪  boards, unable to assign match
```

Listing 6.1: Example server error log.

Standard procedure for running the matches would require a technically literate member of the tournament organisation team to be directly connected to the server, providing real-time monitoring of the system, where any errors can be quickly identified and reported to members of the broadcast team. Whilst this approach was effective, particularly in the context of a feasibility demonstration, integration with dedicated monitoring tools, such as Grafana [186], to provide live dashboards which could be used to manage errors in the system would substantially improve this process, and is deemed as essential prior to deployment in a tournament context. This work, combined with error correction tooling which could be used to resolve mismatches in game states taking place during a match, was considered outside the scope of this project, primarily due to time constraints, but forms an important part of future work which should be completed prior to deployment in Alchemist Cup 2024 and is further discussed in section 7.2.

One feature which is notably missing from the implementation is a "calibration mode" which could be used when setting up the match to verify that the tile QR codes are correct, and that the system is performing as expected. Such a mode could involve placing all tiles on the board, following some particular arrangement, which could then be manually compared to the server-side representation to ensure that all tiles are correctly identified. Adding an endpoint on the HTTP

server to trigger calibration mode would probably be the simplest way of exposing this functionality to tournament organisers, which could then trigger the TCP server to call the board client's `getFullBoardState()` method, which it can then display, allowing the server side representation to be verified.

6.2.2 Maintainability

Code maintainability was a particularly relevant objective to ensure that the core software deliverables produced were extensible to enable further development leading into its potential use in production. As such, considerable effort was spent in the design and implementation stages of the project to ensure that code was well architected, using appropriate object-oriented and functional paradigms to provide a logical structure resilient to subtle bugs and straightforward to build upon. Although this was generally accomplished extremely well, with details and justifications of the decisions made outlined in chapter 3 and chapter 4, there are a few improvements which could be made. One undesirable aspect of the networking stack for sensor communication is the substantial amount of boilerplate code necessary to integrate Capnproto with the asyncio framework used, requiring multiple conceptual implementation layers which need to be understood before making changes to the TCP client or server. Capnproto's Python port has a pending major release which allows for direct integration with asyncio's event loop, not only dramatically reducing the boilerplate required, but providing substantial performance improvements [187]. Once this new version of Capnproto is released, a rewrite of the relevant parts of the client and server code would be particularly helpful in keeping the code maintainable in the long run.

Other improvements all focus on unit testing, which are incredibly useful to document the intended functionality of the different parts of the code, as well as provide future developers with a straightforward method of verifying that new changes do not break existing functionality. The randomised test benches produced for the sensors, and tests for various logical components in the server are all incredibly beneficial in this purpose, the addition of similar testing for the companion application and networking components to provide rapid feedback on new changes would be invaluable to facilitate further work on these components.

Chapter 7

Conclusion and Future Works

7.1 Conclusion

This project aimed to develop a commercially viable, reliable solution to digitize competitive Scrabble matches, modernising the broadcast of international tournaments to the standard of other popular games such as Chess. The system presented in this report, selected after a rigorous design process, successfully provides a novel, fully functioning proof of concept of such a solution, overcoming numerous technical challenges to provide a robust, maintainable platform capable of autonomously tracking the state of multiple concurrent matches, offering scoring and challenge resolution capabilities via a companion Android application.

Given the open-ended nature of the requirements, narrowing the possible design space into the single, most promising option for implementation and testing was especially challenging, especially as this decision held significant implications in the subsequent work produced. Substantial research, combined with careful considerations of the benefits and limitations of each approach, was essential in the ultimate selection of a QR-code based approach, whose superiority to other designs was strongly justified in chapter 3. The architecture of the overall system was another particularly interesting problem, where once again multiple possibilities were evaluated before the combination of network protocols and central server used in the presented solution were chosen.

The integration of hardware and software engineering disciplines in this project was highly rewarding, bringing together many aspects of the Electronic and Information Engineering degree. The development of the randomised test benches to simulate board and rack states, both used to address the performance issues of the detection algorithm in subsection 4.1.4 and asses the accuracy of code identification in subsection 5.1.1 was another highlight, enabling data-driven decision-making.

7.2 Future Works

Many of the project limitations have been critically evaluated in chapter 6, all motivating future work to be done on the project. This section prioritises which additional features would provide the most value, particularly leading up to professional use in Alchemist Cup 2024.

Although the development of hardware prototypes is expected to be complete prior to the project presentation, additional development work refining the initial designs, as well as tuning the detection algorithms for them, is an essential next step. Testing the impact of different colours on the board squares, as well as general testing of the performance of the hardware system, would complement the analysis performed in subsection 5.1.1.

Enhancements to the delta resolution mechanism to potentially handle error correction and the addition of a “calibration mode”, as detailed in subsection 6.2.1, as well as the addition of tooling to allow manual intervention when errors occur, which could be accomplished relatively easily

by exposing additional endpoints in the HTTP server, are vital in ensuring a smooth broadcast experience. Integration of the system with real-time monitoring tools would also be especially helpful in this regard. Addressing the improvements to the codebase suggested in subsection 6.2.2 would also be particularly beneficial to facilitate future changes to the software implementation.

Further research into the performance of micro-QR codes, or even alternative forms of tile value representation such as Data Matrices [188] could provide more performant alternatives to the version 1 QR codes used in the prototype implementation, but as noted in subsection 6.1.2, the detection accuracy achieved with standard QR codes is more than adequate in this use case, making such further work non-essential. Similarly, the design of custom Reed-Solomon codes which could be used as the payloads for these 2D barcodes also have limited value, particularly as designing such a code to operate over digits in $\mathbb{Z}_{10}$ requires an additional conversion step since it cannot directly form a finite field, which would degrade its error correction capacity.

subsection 6.1.1 and subsection 6.1.5 both note that the augmentation of the system to encode the full set of WESPA rules, as well as parameterising challenge penalties, could be used to extend the solution provided to wider international adoption in the Scrabble community, and although ensuring the system's effective operation at Alchemist Cup 2024 was at the core of the project's goals, provided it is deployed successfully, further development to push its use to other Scrabble tournaments is an exciting space in which additional work could be performed.

Bibliography

[1] Hasbro, "Scrabble History | Making of the Classic American Board Game," 2014. [Online]. Available: https://scrabble.hasbro.com/en-us/history

[2] J. Kollewe, "Scrabble regains No 1 spot," *The Guardian*, Nov. 2008. [Online]. Available: https://www.theguardian.com/business/2008/nov/18/scrabble-top-game

[3] T. Telegraph, "Scrabble: 60 facts for its 60th birthday." [Online]. Available: https://www.telegraph.co.uk/news/newstopics/howaboutthat/3776732/Scrabble-60-facts-for-its-60th-birthday.html

[4] S. Cole, "The wild, misunderstood world of competitive Scrabble," Oct. 2020. [Online]. Available: https://www.huckmag.com/outdoor/sport-outdoor/the-wild-misunderstood-world-of-competitive-scrabble/

[5] WESPA, "World English Language Scrabble Players Association." [Online]. Available: https://www.wespa.org

[6] "Alchemist Cup 2024." [Online]. Available: http://www.scrabble.org.au/ratings/selective/2024Alchemist.html

[7] WESPA, "WESPA Youth Cup 2022." [Online]. Available: https://wysc2022.com/

[8] C. Del Solar, "Macondo." [Online]. Available: http://domino14.github.io/macondo/

[9] Hasbro, "Scrabble FAQ: Answers to Your Scrabble Questions | Hasbro." [Online]. Available: https://scrabble.hasbro.com/en-us/faq

[10] T. Brus, "Predicting the Outcome of Scrabble Games," 2015. [Online]. Available: https://fmt.ewi.utwente.nl/media/124.pdf

[11] Chessify, "New Technology For 3D Chess Board Digitalization." [Online]. Available: https://chessify.me/blog/new-technology-for-3d-chess-board-digitalization

[12] DGT, "Our story." [Online]. Available: https://digitalgametechnology.com/about-us

[13] B. Bulsink, "How electronic chessboards works." [Online]. Available: https://groups.google.com/g/rec.games.chess.misc/c/t97ssfGAnhg/m/8Gr9_7LRNIcJ

[14] B. J. Bulsink, "Device for detecting playing pieces on a board," US Patent US6 168 158B1, Jan., 2001. [Online]. Available: https://patents.google.com/patent/US6168158/en

[15] "DGT USB Walnut e-Board and Pieces." [Online]. Available: https://chess.co.uk/products/dgt-usb-walnut-e-board-and-pieces

[16] WESPA, "MSI Prague 1-4 December." [Online]. Available: https://www.wespa.org/news/2012xprague.shtml

[17] P. Nikitin and K. Rao, "Theory and measurement of backscattering from RFID tags," *Antennas and Propagation Magazine, IEEE*, vol. 48, pp. 212–218, Jan. 2007.

[18] M. Gorman, "Scrabble board packs RFID technology, broadcasts tournaments online in real-time, costs 20,000 pounds." [Online]. Available: https://www.engadget.com/2012-11-15-scrabble-board-rfid.html

[19] K. Firestone, "What is Imaging? | Edmund Optics." [Online]. Available: https://www.edmundoptics.co.uk/knowledge-center/application-notes/imaging/what-is-imaging/

[20] geaxgx1, "Scrabble assistant with computer vision (1/3)," Nov. 2017. [Online]. Available: https://www.youtube.com/watch?v=I3ALe-qEwzM

[21] E. Joseph, "Scrabble OCR using OpenCV and Tesseract-OCR," Mar. 2021, original-date: 2016-01-09T10:01:22Z. [Online]. Available: https://github.com/eladj/ScrabbleOCR

[22] L. Brummitt, "Scrabble for the lazy, using real-time OCR." [Online]. Available: http://brm.io/real-time-ocr/

[23] D. Hirschberg, "Scrabble Board Automatic Detector for Third Party Applications," 2016. [Online]. Available: http://rasdasd.com/projects/Scrabble_Detector/Scrabble_Paper.pdf

[24] KIOS, "Scorable." [Online]. Available: https://www.scorableapp.com

[25] geaxgx1, "Scrabble assistant with computer vision (3/3) : rack acquisition," Nov. 2017. [Online]. Available: https://www.youtube.com/watch?v=52sNvLXmX8k

[26] "What is Machine Vision Optics and How Does It Work? | Synopsys." [Online]. Available: https://www.synopsys.com/glossary/what-is-machine-vision-optics.html

[27] Edmund Optics, "Choosing the Right Machine Vision Lens: Part 1," Mar. 2019. [Online]. Available: https://www.youtube.com/watch?v=rIaCPN24Xyw

[28] G. Hollows and N. James, "Depth of Field and Depth of Focus | Edmund Optics." [Online]. Available: https://www.edmundoptics.co.uk/knowledge-center/application-notes/imaging/depth-of-field-and-depth-of-focus/

[29] ——, "Understanding Focal Length and Field of View | Edmund Optics." [Online]. Available: https://www.edmundoptics.co.uk/knowledge-center/application-notes/imaging/understanding-focal-length-and-field-of-view/

[30] ——, "Best Practices for Better Imaging | Edmund Optics." [Online]. Available: https://www.edmundoptics.co.uk/knowledge-center/application-notes/imaging/11-best-practices-for-better-imaging/

[31] J. Cłapa, H. Blasinski, K. Grabowski, and P. Sekalski, "A fisheye distortion correction algorithm optimized for hardware implementations," in *A fisheye distortion correction algorithm optimized for hardware implementations*, Jun. 2014, pp. 415–419.

[32] G. Hollows and N. James, "Distortion | Edmund Optics." [Online]. Available: https://www.edmundoptics.co.uk/knowledge-center/application-notes/imaging/distortion/

[33] E. R. Davies, *Computer vision: principles, algorithms, applications, learning*, 5th ed. United Kingdom: Elsevier Science, 2018.

[34] I. Farup, M. Pedersen, and A. Alsam, "Colour-to-Greyscale Image Conversion by Linear Anisotropic Diffusion of Perceptual Colour Metrics," in *2018 Colour and Visual Computing Symposium (CVCS)*, Sep. 2018, pp. 1–6.

[35] A. M. Hambal, D. Z. Pei, and F. L. Ishabailu, "Image Noise Reduction and Filtering Techniques," *International Journal of Science and Research*, vol. 6, no. 3, 2015.

[36] S. Hichri, F. Benzarti, and H. Amiri, "Robust Noise Filtering in Image Sequences," *International Journal of Computer Applications*, vol. 50, no. 18, pp. 18–23, Jul. 2012. [Online]. Available: http://research.ijcaonline.org/volume50/number18/pxc3881156.pdf

[37] D. Ziou and S. Tabbone, "Edge Detection Techniques - An Overview," *Pattern Recognition and Image Analysis*, vol. 8, pp. 537–559, Jun. 2000.

[38] B. Li, A. Jevtić, U. Söderström, U. Shafiq, S. Réhman, and H. Li, "Fast Edge Detection by Center of Mass," in *The 1st IEEE/IIAE International Conference on Intelligent Systems and Image Processing*, Sep. 2013.

[39] "Feature Detectors - Sobel Edge Detector." [Online]. Available: https://homepages.inf.ed.ac.uk/rbf/HIPR2/sobel.htm

[40] W. Rong, Z. Li, W. Zhang, and L. Sun, "An improved Canny edge detection algorithm," in *2014 IEEE International Conference on Mechatronics and Automation*, Aug. 2014, pp. 577–582, iSSN: 2152-744X.

[41] OpenCV, "OpenCV: Canny Edge Detection." [Online]. Available: https://docs.opencv.org/4.x/da/d22/tutorial_py_canny.html

[42] P. Dollar and C. L. Zitnick, "Structured Forests for Fast Edge Detection," in *2013 IEEE International Conference on Computer Vision*. Sydney, Australia: IEEE, Dec. 2013, pp. 1841–1848. [Online]. Available: http://ieeexplore.ieee.org/document/6751339/

[43] X.-Y. Gong, H. Su, D. Xu, Z.-T. Zhang, F. Shen, and H.-B. Yang, "An Overview of Contour Detection Approaches," *Machine Intelligence Research*, vol. 15, no. 6, pp. 656–672, Jun. 2018, publisher: Machine Intelligence Research. [Online]. Available: https://www.mi-research.net/en/article/doi/10.1007/s11633-018-1117-z

[44] S. Suzuki and K. be, "Topological structural analysis of digitized binary images by border following," *Computer Vision, Graphics, and Image Processing*, vol. 30, no. 1, pp. 32–46, Apr. 1985. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0734189X85900167

[45] OpenCV, "OpenCV: Structural Analysis and Shape Descriptors." [Online]. Available: https://docs.opencv.org/3.4/d3/dc0/group__imgproc__shape.html#ga17ed9f5d79ae97bd4c7cf18403e1689a

[46] ——, "OpenCV: Contours : Getting Started." [Online]. Available: https://docs.opencv.org/3.4/d4/d73/tutorial_py_contours_begin.html

[47] D. Charnley and R. J. Blissett, "Surface Reconstruction from Outdoor Image Sequences," in *Procedings of the Alvey Vision Conference 1988*. Manchester: Alvey Vision Club, 1988, pp. 24.1–24.6. [Online]. Available: http://www.bmva.org/bmvc/1988/avc-88-024.html

[48] C. Harris and M. Stephens, "A Combined Corner and Edge Detector," in *Procedings of the Alvey Vision Conference 1988*. Manchester: Alvey Vision Club, 1988, pp. 23.1–23.6. [Online]. Available: http://www.bmva.org/bmvc/1988/avc-88-023.html

[49] Jianbo Shi and Tomasi, "Good features to track," in *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition CVPR-94*. Seattle, WA, USA: IEEE Comput. Soc. Press, 1994, pp. 593–600. [Online]. Available: http://ieeexplore.ieee.org/document/323794/

[50] C. Schmid, R. Mohr, and C. Bauckhage, "Evaluation of Interest Point Detectors," *International Journal of Computer Vision*, vol. 37, no. 2, pp. 151–172, 2000. [Online]. Available: http://link.springer.com/10.1023/A:1008199403446

[51] D. G. Lowe, "Distinctive Image Features from Scale-Invariant Keypoints," *International Journal of Computer Vision*, vol. 60, no. 2, pp. 91–110, Nov. 2004. [Online]. Available: http://link.springer.com/10.1023/B:VISI.0000029664.99615.94

[52] K. Mikolajczyk and C. Schmid, "A Performance Evaluation of Local Descriptors," *IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE*, vol. 27, no. 10, 2005.

[53] H. Bay, T. Tuytelaars, and L. Van Gool, "SURF: Speeded Up Robust Features," in *Computer Vision – ECCV 2006*, ser. Lecture Notes in Computer Science, A. Leonardis, H. Bischof, and A. Pinz, Eds. Berlin, Heidelberg: Springer, 2006, pp. 404–417.

[54] D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, M. Calonder, V. Lepetit, C. Strecha, and P. Fua, "BRIEF: Binary Robust Independent Elementary Features," in *Computer Vision – ECCV 2010*, K. Daniilidis, P. Maragos, and N. Paragios, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, vol. 6314, pp. 778–792, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/978-3-642-15561-1_56

[55] E. Rosten and T. Drummond, "Machine Learning for High-Speed Corner Detection," in *Computer Vision – ECCV 2006*, ser. Lecture Notes in Computer Science, A. Leonardis, H. Bischof, and A. Pinz, Eds. Berlin, Heidelberg: Springer, 2006, pp. 430–443.

[56] S. Leutenegger, M. Chli, and R. Y. Siegwart, "BRISK: Binary Robust invariant scalable keypoints," in *2011 International Conference on Computer Vision*, Nov. 2011, pp. 2548–2555, iSSN: 2380-7504.

[57] OpenCV, "OpenCV: Feature Matching." [Online]. Available: https://docs.opencv.org/3.4/dc/dc3/tutorial_py_matcher.html

[58] M. Maulion, "Homography Transform — Image Processing," Feb. 2021. [Online]. Available: https://mattmaulion.medium.com/homography-transform-image-processing-eddbcb8e4ff7

[59] K. Fukushima, "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position," *Biological Cybernetics*, vol. 36, no. 4, pp. 193–202, Apr. 1980. [Online]. Available: http://link.springer.com/10.1007/BF00344251

[60] N. Sarika, N. Sirisala, and M. S. Velpuru, "CNN based Optical Character Recognition and Applications," in *2021 6th International Conference on Inventive Computation Technologies (ICICT)*, Jan. 2021, pp. 666–672.

[61] F. Chollet, *Deep learning with Python*. Shelter Island, New York: Manning Publications Co, 2018, oCLC: ocn982650571.

[62] J. Brownlee, "How Do Convolutional Layers Work in Deep Learning Neural Networks?" Apr. 2019. [Online]. Available: https://machinelearningmastery.com/convolutional-layers-for-deep-learning-neural-networks/

[63] "MaxpoolSample2.png (PNG Image, 750 × 313 pixels)." [Online]. Available: https://computersciencewiki.org/images/8/8a/MaxpoolSample2.png

[64] Phung and Rhee, "A High-Accuracy Model Average Ensemble of Convolutional Neural Networks for Classification of Cloud Image Patches on Small Datasets," *Applied Sciences*, vol. 9, p. 4500, Oct. 2019.

[65] F. Chollet, "Keras documentation: Simple MNIST convnet." [Online]. Available: https://keras.io/examples/vision/mnist_convnet/

[66] C. Dilmegani, "OCR in 2023: Benchmarking Text Extraction/Capture Accuracy." [Online]. Available: https://research.aimultiple.com/ocr-accuracy/

[67] K. Hamad and M. Kaya, "A Detailed Analysis of Optical Character Recognition Technology," *International Journal of Applied Mathematics, Electronics and Computers*, vol. 4, pp. 244–244, Dec. 2016.

[68] D. WAVE, "Micro QR Code | QRcode.com | DENSO WAVE." [Online]. Available: https://www.qrcode.com/en/codes/microqr.html

[69] Android, "Jetpack Compose UI App Development Toolkit." [Online]. Available: https://developer.android.com/jetpack/compose

[70] ——, "Thinking in Compose | Jetpack Compose." [Online]. Available: https://developer.android.com/jetpack/compose/mental-model

[71] ——, "State and Jetpack Compose." [Online]. Available: https://developer.android.com/jetpack/compose/state

[72] ——, "Stages of the Activity lifecycle | Android Developers." [Online]. Available: https://developer.android.com/codelabs/basic-android-kotlin-compose-activity-lifecycle

[73] ——, "ViewModel overview." [Online]. Available: https://developer.android.com/topic/libraries/architecture/viewmodel

[74] ——, "Architecting your Compose UI | Jetpack Compose." [Online]. Available: https://developer.android.com/jetpack/compose/architecture

[75] Android Developers, "Architecture: The UI layer - MAD Skills," Mar. 2022. [Online]. Available: https://www.youtube.com/watch?v=p9VR8KbmzEE

[76] R. J. Bature, "Understanding TCP/IP Transport Layer (Layer 3) protocols - TCP and UDP," Feb. 2021. [Online]. Available: https://www.section.io/engineering-education/understanding-tcp-ip-transport-layer-protocols/

[77] D. P. Proos and N. Carlsson, "Performance Comparison of Messaging Protocols and Serialization Formats for Digital Twins in IoV," in *2020 IFIP Networking Conference (Networking)*, Jun. 2020, pp. 10–18.

[78] S. Cooper, "How to Fix Packet Loss in 8 Steps," Apr. 2019. [Online]. Available: https://www.comparitech.com/net-admin/how-to-fix-packet-loss/

[79] Stephen Cleary, "Raw TCP/IP socket communication," St. Petersburg, Nov. 2020. [Online]. Available: https://www.youtube.com/watch?v=cSlL4nWmZuQ

[80] K. Varda, "Cap'n Proto: Introduction." [Online]. Available: https://capnproto.org/

[81] B. Muller, "RPCUDP : RPC over UDP in Python," Nov. 2022, original-date: 2013-12-31T18:34:41Z. [Online]. Available: https://github.com/bmuller/rpcudp

[82] MQTT, "MQTT - The Standard for IoT Messaging." [Online]. Available: https://mqtt.org/

[83] N. Naik, "Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP," in *2017 IEEE International Systems Engineering Symposium (ISSE)*, Oct. 2017, pp. 1–7.

[84] Mozilla, "An overview of HTTP - HTTP | MDN," May 2023. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview

[85] C. B. Gemirter, Şenturca, and Baydere, "A Comparative Evaluation of AMQP, MQTT and HTTP Protocols Using Real-Time Public Smart City Data," in *2021 6th International Conference on Computer Science and Engineering (UBMK)*, Sep. 2021, pp. 542–547, iSSN: 2521-1641.

[86] "Resistor Values | Resistor Standards and Codes | Resistor Guide." [Online]. Available: https://eepower.com/resistor-guide/resistor-standards-and-codes/resistor-values/

[87] E. Napieralska, K. Komeza, F. Morganti, J. Sykulski, G. Vega, and Y. Zeroukhi, "Measurement of contact resistance for copper and aluminium conductors," *International Journal of Applied Electromagnetics and Mechanics*, vol. 53, pp. 1–13, Jan. 2017.

[88] P. van Dijk, "Critical Aspects of Electrical Connector Contacts," in *2nd International Conference on Reliability of Electrical Products and Electrical Contacts (ICREPEC)*, China, Sep. 2007, pp. 161–168. [Online]. Available: https://www.pvdijk.com/pdf/21thiceccriticalaspects.pdf

[89] "Tesseract OCR," Feb. 2023, original-date: 2014-08-12T18:04:59Z. [Online]. Available: https://github.com/tesseract-ocr/tesseract

[90] G. Cloud, "Vision AI | Cloud Vision API." [Online]. Available: https://cloud.google.com/vision

[91] "Intelligently Extract Text & Data with OCR - Amazon Textract - Amazon Web Services." [Online]. Available: https://aws.amazon.com/textract/

[92] R. Pi, "Raspberry Pi Documentation - Camera." [Online]. Available: https://www.raspberrypi.com/documentation/accessories/camera.html

[93] L. West, "The Rise Of The Raspberry Pi In Industrial Settings - Rowse," Nov. 2022. [Online]. Available: https://www.rowse.co.uk/blog/post/the-rise-of-the-raspberry-pi-in-industrial-settings

[94] Arducam, "Arducam 1/4" M12 Mount 1.05mm Focal Length Fisheye Camera lens M40105M19." [Online]. Available: https://www.arducam.com/product/u0858-1-4-m12-mount-1-05mm-focal-length-fisheye-camera-lens-ls-40180-for-raspberry-pi/

[95] L. Jackson, "Raspberry Pi Multiple Camera Solutions: Adapters, Multiplexers, and Projects," Apr. 2015. [Online]. Available: https://www.arducam.com/multi-camera-adapter-module-raspberry-pi/

[96] Arducam, "Arducam M12 Mount Camera Lens M25170H12." [Online]. Available: https://www.arducam.com/product/m25170h12-2/

[97] "Arducam 1/4" M12 Mount 2.1mm Focal Length Low Distortion Camera Lens M40210M09S." [Online]. Available: https://www.arducam.com/product/arducam-1-4-m12-mount-2-1mm-focal-length-distortion-camera-lens-m40210m09s/

[98] "OpenCV: cv::QRCodeDetector Class Reference." [Online]. Available: https://docs.opencv.org/4.x/de/dc3/classcv_1_1QRCodeDetector.html

[99] "Tutorial QRCodes - BoofCV." [Online]. Available: https://boofcv.org/index.php?title=Tutorial_QRCodes

[100] J. Peltokorpi, L. Isojärvi, K. Häkkinen, and E. Niemi, "QR code-based material flow monitoring in a subcontractor manufacturer network," *Procedia Manufacturing*, vol. 55, pp. 110–115, Sep. 2021.

[101] Collins, "csw21 directory listing." [Online]. Available: https://archive.org/download/csw21/

[102] M. Tamera, "Understanding the Current Global Semiconductor Shortage, Preparing for the Future," Aug. 2022. [Online]. Available: https://www.spglobal.com/engineering/en/research-analysis/understanding-the-current-global-semiconductor-shortage.html

[103] E. Upton, "Supply chain update - it's good news!" Dec. 2022. [Online]. Available: https://www.raspberrypi.com/news/supply-chain-update-its-good-news/

[104] MOOC, "iOS vs. Android App Development: What's the Difference?" [Online]. Available: https://www.mooc.org/blog/ios-vs.-android-app-development

[105] Apple, "Swift Resources - Apple Developer." [Online]. Available: https://developer.apple.com/swift/resources/

[106] R. P. Ltd, "Buy a Raspberry Pi Touch Display." [Online]. Available: https://www.raspberrypi.com/products/raspberry-pi-touch-display/

[107] Android, "Connect Bluetooth devices." [Online]. Available: https://developer.android.com/guide/topics/connectivity/bluetooth/connect-bluetooth-devices

[108] AWS, "Amazon Compute Service Level Agreement." [Online]. Available: https://aws.amazon.com/compute/sla/

[109] A. Polkovnikov, "Azure SLA Board." [Online]. Available: https://azurecharts.com/sla

[110] AWS, "Load Balancer - Amazon Elastic Load Balancer (ELB) - AWS." [Online]. Available: https://aws.amazon.com/elasticloadbalancing/

[111] ——, "Amazon EC2 Auto Scaling benefits - Amazon EC2 Auto Scaling." [Online]. Available: https://docs.aws.amazon.com/autoscaling/ec2/userguide/auto-scaling-benefits.html

[112] ——, "Resilience in Amazon EC2 Auto Scaling - Amazon EC2 Auto Scaling." [Online]. Available: https://docs.aws.amazon.com/autoscaling/ec2/userguide/disaster-recovery-resiliency.html

[113] R. Pi, "The Picamera2 Library," Apr. 2023. [Online]. Available: https://datasheets.raspberrypi.com/camera/picamera2-manual.pdf

[114] OpenCV, "Home." [Online]. Available: https://opencv.org/

[115] StereoPi, "OpenCV: comparing the speed of C++ and Python code on the Raspberry Pi for stereo vision | StereoPi - DIY stereoscopic camera based on Raspberry Pi." [Online]. Available: https://stereopi.com/blog/opencv-comparing-speed-c-and-python-code-raspberry-pi-stereo-vision

[116] P. Abeles, "BoofCV." [Online]. Available: https://boofcv.org/index.php?title=Main_Page

[117] T. Elliott, "The State of the Octoverse: machine learning," Jan. 2019. [Online]. Available: https://github.blog/2019-01-24-the-state-of-the-octoverse-machine-learning/

[118] K. Team, "Keras documentation: About Keras." [Online]. Available: https://keras.io/about/

[119] "Kotlin Programming Language." [Online]. Available: https://kotlinlang.org/

[120] BeeWare, "Write once. Deploy everywhere.— BeeWare." [Online]. Available: https://beeware.org/

[121] B. Crom, "Scrabble," Feb. 2023, original-date: 2017-02-10T20:40:03Z. [Online]. Available: https://github.com/benjamincrom/scrabble

[122] R. Pi, "Raspberry Pi Documentation - Processors." [Online]. Available: https://www.raspberrypi.com/documentation/computers/processors.html

[123] Python, "multiprocessing — Process-based parallelism." [Online]. Available: https://docs.python.org/3/library/multiprocessing.html

[124] ——, "GlobalInterpreterLock - Python Wiki." [Online]. Available: https://wiki.python.org/moin/GlobalInterpreterLock

[125] 14:00-17:00, "Information technology — Automatic identification and data capture techniques — Rectangular Micro QR Code (rMQR) bar code symbology specification," May 2022. [Online]. Available: https://www.iso.org/standard/77404.html

[126] S. Staal, "MicroQR Code Detection Support · Issue #23218 · opencv/opencv," Feb. 2023. [Online]. Available: https://github.com/opencv/opencv/issues/23218

[127] R. Hirsch, *Exploring Colour Photography: A Complete Guide*. Laurence King, 2005, google-Books-ID: 4Gx2WItWGYoC.

[128] "OpenCV: Image file reading and writing." [Online]. Available: https://docs.opencv.org/3.4/d4/da8/group__imgcodecs.html

[129] "YUV," Mar. 2023, page Version ID: 1143386199. [Online]. Available: https://en.wikipedia.org/w/index.php?title=YUV&oldid=1143386199

[130] OpenCV, "OpenCV: Fisheye camera model." [Online]. Available: https://docs.opencv.org/3.4/db/d58/group__calib3d__fisheye.html

[131] K. Jiang, "Calibrate fisheye lens using OpenCV — part 1," Sep. 2017. [Online]. Available: https://medium.com/@kennethjiang/calibrate-fisheye-lens-using-opencv-333b05afa0b0

[132] S. Schmerler, "elcorto/unfish," Apr. 2023, original-date: 2017-08-15T19:36:23Z. [Online]. Available: https://github.com/elcorto/unfish

[133] S. L. Garbett, "Raspberry Pi 3 vs. 4: What's the Difference?" Oct. 2022, section: DIY. [Online]. Available: https://www.makeuseof.com/raspberry-pi-3-vs-4-differences/

[134] R. Pi, "Raspberry Pi Documentation - Processors." [Online]. Available: https://www.raspberrypi.com/documentation/computers/processors.html

[135] T. P. Team, "PyPy," Dec. 2019. [Online]. Available: https://www.pypy.org/

[136] W. T. Lavrijsen, "cppyy: Automatic Python-C++ bindings — cppyy 2.4.0 documentation." [Online]. Available: https://cppyy.readthedocs.io/en/latest/

[137] W. T. Lavrijsen and A. Dutta, "High-Performance Python-C++ Bindings with PyPy and Cling," in *2016 6th Workshop on Python for High-Performance and Scientific Computing (PyHPC)*. Salt Lake, UT, USA: IEEE, Nov. 2016, pp. 27–35. [Online]. Available: http://ieeexplore.ieee.org/document/7836841/

[138] J. Lockwood, "Is PyPy a Faster Version of Python? - CodeSolid.com," Feb. 2022. [Online]. Available: https://codesolid.com/pypy-first-look-a-faster-version-of-python/

[139] A. R. K, "Answer to "Does performance differ between Python or C++ coding of OpenCV?"," Nov. 2012. [Online]. Available: https://stackoverflow.com/a/13433330

[140] L. Heuer, "segno: QR Code and Micro QR Code generator for Python 2 and Python 3." [Online]. Available: https://github.com/heuer/segno/

[141] numpy, "numpy.vectorize — NumPy v1.24 Manual." [Online]. Available: https://numpy.org/doc/stable/reference/generated/numpy.vectorize.html

[142] "zxing/zxing," Jun. 2023, original-date: 2011-10-12T14:07:27Z. [Online]. Available: https://github.com/zxing/zxing

[143] M. C. Chehab, "ZBAR BAR CODE READER," Jun. 2023, original-date: 2018-08-08T10:52:40Z. [Online]. Available: https://github.com/mchehab/zbar

[144] P. Abeles, "PyBoof: Py4J Python wrapper for BoofCV." [Online]. Available: https://github.com/lessthanoptimal/PyBoof

[145] C. Xu, "pyzxing," May 2023, original-date: 2020-05-28T14:27:29Z. [Online]. Available: https://github.com/ChenjieXu/pyzxing

[146] L. Hudson, "pyzbar: Read one-dimensional barcodes and QR codes from Python 2 and 3." [Online]. Available: https://github.com/NaturalHistoryMuseum/pyzbar/

[147] P. Abeles, "Study of QR Code Scanning Performance in Different Environments," Mar. 2019. [Online]. Available: https://boofcv.org/index.php?title=Performance:QrCode

[148] axxel, "ZXing-C++," Jun. 2023, original-date: 2016-04-12T22:33:06Z. [Online]. Available: https://github.com/zxing-cpp/zxing-cpp

[149] S. Staal, "QR Code detector fails for 4-digit integer messages," May 2023. [Online]. Available: https://github.com/lessthanoptimal/PyBoof/issues/23

[150] ——, "Unable to QR Codes with particular payloads · Issue #23810 · opencv/opencv," Jun. 2023. [Online]. Available: https://github.com/opencv/opencv/issues/23810

[151] ——, "Malformed padding when number of bits is a multiple of 8," May 2023. [Online]. Available: https://github.com/heuer/segno/issues/123

[152] ——, "Unable to detect "perfect" micro-QR codes with particular data payloads · Issue #578 · zxing-cpp/zxing-cpp," Jun. 2023. [Online]. Available: https://github.com/zxing-cpp/zxing-cpp/issues/578

[153] ArduCam, "Multi_camera_adapter_v2," May 2023, original-date: 2018-09-05T11:02:14Z. [Online]. Available: https://github.com/ArduCAM/RaspberryPi

[154] Arducam, "4 HQ Camera Modules with One Raspberry Pi: A Multiple Camera Adapter," Aug. 2020. [Online]. Available: https://www.youtube.com/watch?v=I_6uKSLOpNw

[155] ArduCam, "Arducam 12MP*4 477P Quadrascopic Camera Bundle Kit for Raspberry Pi, Nvidia Jetson Nano/Xavier NX." [Online]. Available: https://www.arducam.com/product/12mp4-quadrascopic-camera-bundle-kit-b0397/

[156] "F-score," Apr. 2023, page Version ID: 1148225663. [Online]. Available: https://en.wikipedia.org/w/index.php?title=F-score&oldid=1148225663

[157] Python, "threading — Thread-based parallelism." [Online]. Available: https://docs.python.org/3/library/threading.html

[158] ——, "asyncio — Asynchronous I/O." [Online]. Available: https://docs.python.org/3/library/asyncio.html

[159] gRPC, "gRPC AsyncIO API — gRPC Python 1.55.0 documentation." [Online]. Available: https://grpc.github.io/grpc/python/grpc_asyncio.html#grpc.aio.server

[160] D. Parimi, W. J. Zhao, and J. Zhao, "Datacenter Tax Cuts: Improving WSC Efficiency Through Protocol Buffer Acceleration," *Tech. Rep*, 2019. [Online]. Available: https://people.eecs.berkeley.edu/~kubitron/courses/cs262a-F19/projects/reports/project4_report_ver2.pdf

[161] K. Varda, "Cap'n Proto: RPC Protocol." [Online]. Available: https://capnproto.org/rpc.html

[162] theheadofabroom, "Creating a singleton in Python," May 2022. [Online]. Available: https://stackoverflow.com/q/6760685

[163] S. Staal, "Integrating capture callbacks with asyncio · Issue #714 · raspberrypi/picamera2." [Online]. Available: https://github.com/raspberrypi/picamera2/issues/714

[164] AIOHTTP, "Welcome to AIOHTTP — aiohttp 3.8.4 documentation." [Online]. Available: https://docs.aiohttp.org/en/stable/

[165] Flask, "Welcome to Flask — Flask Documentation (2.2.x)." [Online]. Available: https://flask.palletsprojects.com/en/2.2.x/

[166] Y. Tsutsumi, "Aiohttp vs Multithreaded Flask for High I/O Applications," Sep. 2017. [Online]. Available: https://y.tsutsumi.io/2017/09/23/aiohttp-vs-multithreaded-flask-for-high-io-applications/

[167] Gambiter, "Challenge (Scrabble) - Rules and strategy of scrabble games." [Online]. Available: https://gambiter.com/scrabble/Challenge_scrabble.html

[168] P. Gonzalez Alonso, "Repository Pattern with Jetpack Compose." [Online]. Available: https://www.kodeco.com/24509368-repository-pattern-with-jetpack-compose

[169] Retrofit, "Retrofit." [Online]. Available: https://square.github.io/retrofit/

[170] Python, "unittest.mock — mock object library." [Online]. Available: https://docs.python.org/3/library/unittest.mock.html

[171] K. M. Douglass, "Modeling noise for image simulations," Jul. 2017. [Online]. Available: http://kmdouglass.github.io/posts/modeling-noise-for-image-simulations/

[172] S. R. Alvar and I. V. Bajić, "Practical Noise Simulation for RGB Images," Jan. 2022, arXiv:2201.12773 [cs, eess]. [Online]. Available: http://arxiv.org/abs/2201.12773

[173] Sony, "IMX477 Data Sheet." [Online]. Available: https://www.uctronics.com/download/Image_Sensor/IMX477-DS.pdf

[174] R. P. Trading, "Raspberry Pi Camera Guide," Jan. 2021. [Online]. Available: https://datasheets.raspberrypi.com/camera/raspberry-pi-camera-guide.pdf

[175] OpenCV, "OpenCV: Changing the contrast and brightness of an image!" [Online]. Available: https://docs.opencv.org/3.4/d3/dc1/tutorial_basic_linear_transform.html

[176] Python, "unittest — Unit testing framework." [Online]. Available: https://docs.python.org/3/library/unittest.html

[177] Android, "Test in Android Studio | Android Developers." [Online]. Available: https://developer.android.com/studio/test/test-in-android-studio

[178] JUnit, "JUnit 5." [Online]. Available: https://junit.org/junit5/

[179] S. Aiya, "Boards." [Online]. Available: https://www.samtimer.com/boards

[180] AWS, "AWS in Indonesia." [Online]. Available: https://aws.amazon.com/local/jakarta/

[181] R. Paschotta, "Fibers." [Online]. Available: https://www.rp-photonics.com/fibers.html

[182] R. Thanniru and P. Tian, "The Art of HTTP Connection Pooling: How to Optimize Your Connections for Peak Performance," May 2023. [Online]. Available: https://devblogs.microsoft.com/premier-developer/the-art-of-http-connection-pooling-how-to-optimize-your-connections-for-peak-performance/

[183] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and Why Your Code Starts to Smell Bad," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Florence, Italy: IEEE, May 2015, pp. 403–414. [Online]. Available: http://ieeexplore.ieee.org/document/7194592/

[184] AWS, "EC2 On-Demand Instance Pricing – Amazon Web Services." [Online]. Available: https://aws.amazon.com/ec2/pricing/on-demand/

[185] Python, "logging — Logging facility for Python." [Online]. Available: https://docs.python.org/3/library/logging.html

[186] G. Labs, "Grafana: The open observability platform." [Online]. Available: https://grafana.com/

[187] L. Blaauwbroek, "Integrate the KJ event loop into Python's asyncio event loop by LasseBlaauwbroek · Pull Request #310 · capnproto/pycapnp." [Online]. Available: https://github.com/capnproto/pycapnp/pull/310

[188] "Data Matrix," Jun. 2023, page Version ID: 1158650794. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Data_Matrix&oldid=1158650794

[189] S. Staal, "Unable to detect QR codes with particular data values · Issue #38 · ChenjieXu/pyzxing," Jun. 2023. [Online]. Available: https://github.com/ChenjieXu/pyzxing/issues/38

Appendix A

Code and Experimental Data

All code produced for this project is available on the Alchemist Cup GitHub organisation. Installation instructions and guides to the specific repositories are detailed there.

Appendix B

Worst Case Voltage Values

The table below computes the maximum and minimum voltage values obtained from a potential divider with a $5V$ input and $R_1 = 10k\Omega$ for the maximum 1% deviations in resistor tolerance. The worst-case difference is computed as the smallest difference in voltage between i and its neighbours assuming maximum errors.

i	E96 Resistor (Ω)	Minimum voltage (V)	Maximum voltage (V)	Worst-case difference (mV)
1	3.74e+02	0.179	0.182	171
2	7.68e+02	0.353	0.360	171
3	1.21e+03	0.535	0.545	158
4	1.65e+03	0.702	0.714	158
5	2.15e+03	0.877	0.892	163
6	2.74e+03	1.067	1.084	153
7	3.32e+03	1.237	1.256	153
8	4.02e+03	1.423	1.444	155
9	4.75e+03	1.599	1.621	155
10	5.62e+03	1.787	1.810	145
11	6.49e+03	1.956	1.980	145
12	7.5e+03	2.131	2.155	151
13	8.66e+03	2.308	2.333	153
14	1e+04	2.487	2.512	149
15	1.15e+04	2.662	2.687	149
16	1.33e+04	2.842	2.866	153
17	1.54e+04	3.019	3.043	153
18	1.82e+04	3.215	3.238	138
19	2.1e+04	3.376	3.398	138
20	2.49e+04	3.557	3.577	159
21	3.01e+04	3.744	3.762	154
22	3.65e+04	3.916	3.933	154
23	4.64e+04	4.106	4.121	163
24	6.04e+04	4.284	4.296	159
25	8.25e+04	4.455	4.464	159
26	1.3e+05	4.640	4.646	172
27	2.67e+05	4.818	4.821	172

Table B.1: Deviations in voltage with ±1% resistor tolerance

Appendix C

Problematic QR Codes Identified for Various Open-Source Libraries

This section contains details on the QR code payloads which could not be detected by particular open-source QR-code detection implementations considered in subsection 4.1.4, namely OpenCV (`opencv-python`), BoofCV (`PyBoof`), ZXing (`pyzxing`) and ZBar (`pyzbar`). Tests were run in Ubuntu 22.04 with Python 3.10.6 under a virtual environment containing the following libraries and versions:

```
joblib==1.2.0
numpy==1.24.3
opencv-python==4.7.0.72
py4j==0.10.9.7
PyBoof==0.43.1
pyzbar==0.1.9
pyzxing==1.0.2
segno==1.5.2
six==1.16.0
transforms3d==0.4.1
```

QR codes were generated using the segno library as follows:

```
qrcode = segno.make(data, version=1)
assert qrcode.error == 'H'
qrcode.save(TEST_FILE, scale = 5) # Saves as 145 by 145

img = cv2.imread(TEST_FILE)
img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
img = cv2.resize(img, (88, 88)) # Downsamples image
```

Testing over the entire state space considered for QR codes in this project was computationally infeasible, and as such only values in the range $r \in [0, 10^6]$ were testing exhaustively. It is likely that there exist values outside this range which also result in problematic payloads.

It is worth noting that at the time of writing, `segno` produces malformed padding for generated QR codes with 4-digit codes, and potentially subsequent multiples of 4 [151]. This resulted in an older version of `PyBoof==0.41` to fail in detecting any 4-digit QR codes [149], which was solved in the current version by simply ignoring the padding; the approach used by all other tested libraries which did not suffer from this issue. The values resulting in undetectable QR codes for each library is listed below:

OpenCV

QR detection was performed as follows given an image generated through the methodology outlined above encoding a particular value r:

```
detector = cv2.QRCodeDetector()
detected, points, straight_qrcode = detector.detectAndDecode(img)
if not detected:
    bad_codes.append(r)
else:
    assert str(detected) == str(r) # Verifies correct decoded value
```

This identified the following problematic codes:

9133, 24901, 27002, 28211, 30270, 33094, 36997, 37379, 51851, 62533, 63856, 70558, 71054, 74397, 76142, 79599, 91536

An issue tracking this bug is currently open [150].

BoofCV

QR detection was performed as follows given an image generated through the methodology outlined above encoding a particular value r:

```
detector = pb.FactoryFiducial(np.uint8).qrcode()
img = pb.ndarray_to_boof(img)
detector.detect(img)
if len(detector.detections) == 0:
    bad_codes.append(r)
else:
    # Verifies correct decoded value
    assert len(detector.detections) == 1
    assert detector.detections[0].message == str(r)
```

This identified the following problematic codes:

2474, 4916, 6079, 7243, 7531, 7763, 8138, 8698, 9049, 9158, 9372, 9884, 10110, 10427, 11207, 11303, 11674, 12389, 14244, 14515, 16312, 16456, 18454, 19705, 20415, 20436, 21145, 21749, 21914, 24172, 25165, 25660, 25997, 26322, 26405, 27887, 28255, 29747, 30368, 30726, 33405, 33820, 34006, 35119, 35318, 36409, 37855, 38863, 39885, 41031, 41097, 41688, 42746, 42841, 43425, 50792, 51712, 51764, 51795, 53224, 54567, 55074, 55446, 55522, 55577, 55643, 55881, 57067, 58467, 58561, 59293, 60299, 60857, 60983, 62159, 62861, 64115, 64376, 65192, 65330, 65482, 65699, 65910, 66539, 67010, 67614, 67880, 68250, 68365, 68620, 68752, 69920, 69997, 70120, 70292, 70570, 70708, 70715, 71009, 72280, 72420, 73730, 75275, 75295, 75897, 76185, 77126, 79383, 79737, 79929, 81720, 82106, 83475, 83630, 84087, 84088, 84409, 84848, 86459, 87442, 91170, 91265, 91582, 92247, 92950, 93117, 94101, 94182, 94914, 97164, 99174, 99224

An issue tracking this bug is currently open [149].

ZXing

QR detection was performed as follows given an image generated through the methodology outlined above encoding a particular value r:

```
reader = pyzxing.BarCodeReader()
qr_codes = reader.decode_array(img)
assert len(qr_codes) == 1
if 'raw' not in qr_codes[0]:
        bad_codes.append(x)
```

```
else:
    # Verifies correct decoded value
    assert barcodes[0]['raw'].decode("utf-8") == str(r)
```

This identified the following problematic codes:

1982, 2189, 2429, 4041, 4135, 4598, 4608, 4705, 5468, 6310, 6466, 6620, 6904, 7418, 9983, 10279, 11066, 11561, 13451, 14861, 14940, 15105, 16171, 16697, 17199, 17237, 17367, 18125, 18249, 18251, 18331, 19198, 19295, 19485, 20451, 21842, 22820, 22975, 23030, 23110, 23389, 23502, 23577, 24380, 24403, 24537, 24836, 25114, 25393, 25481, 26545, 26668, 26783, 26872, 27075, 27649, 29342, 29744, 29759, 30473, 31283, 32182, 33687, 33982, 34427, 35288, 36713, 36731, 37872, 38155, 38630, 39141, 39892, 40506, 41855, 42022, 42536, 42918, 43036, 43452, 44255, 44300, 47121, 47681, 48830, 48942, 49808, 49992, 50922, 52182, 53588, 54099, 54441, 54635, 55294, 55540, 55802, 56831, 57003, 57950, 58161, 58240, 58815, 60599, 60826, 60915, 61078, 61569, 61612, 62202, 62457, 62710, 63392, 64002, 64632, 65289, 65742, 66070, 66175, 66495, 67268, 67345, 67861, 68142, 69169, 69772, 69991, 70558, 70585, 72597, 72769, 73250, 73726, 74664, 74729, 75836, 76625, 77483, 77574, 77762, 77906, 78091, 78339, 78350, 78626, 78678, 78889, 78922, 79267, 79726, 79889, 80122, 80483, 80615, 80680, 80886, 80930, 81243, 81315, 81862, 82485, 82509, 82600, 84801, 85762, 86122, 87319, 87729, 89386, 89478, 89996, 90458, 90709, 90938, 91397, 91541, 92479, 93120, 93287, 94030, 94865, 95290, 95342, 95377, 95700, 95815, 96620, 97158, 98549, 98665, 99579, 99794

ZXing is no longer under active development, and as such, a fix is unlikely to be deployed. However, an issue tracking the bug is currently open in the Python wrapper library [189]

ZBar

QR detection was performed as follows given an image generated through the methodology outlined above encoding a particular value r:

```
qr_codes = pyzbar.decode(img)
if len(qr_codes) == 0:
    bad_codes.append(r)
else:
    # Verifies correct decoded value
    assert len(qr_codes) == 1
    assert barcodes[0].data.decode("utf-8") == str(r)
```

All codes in the range $r \in [0, 10^6]$ were detected correctly.

Appendix D

Delta Resolution Unit Test Details

D.1 Tile Bag

Contents correctly updated given tiles drawn This test verifies that the contents of the TileBag class are correctly updated when tiles are drawn from the bag.

Detects invalid draw with too many of a particular letter This test ensures that the TileBag class correctly detects and handles the scenario where an invalid number of tiles for a particular letter is drawn from the bag.

Detects invalid draw containing too many tiles This test checks whether the TileBag class correctly identifies and handles the situation when an excessive number of tiles is drawn from the bag.

Detects invalid draw when containing letter with 0 remaining entries This test validates that the TileBag class can identify and handle the case where a letter with zero remaining entries is attempted to be drawn from the bag.

Correct number of expected tiles returned with many tiles in bag This test verifies that the TileBag class returns the correct number of expected tiles when there are a large number of tiles remaining in the bag.

Correct number of expected tiles returned with <7 tiles in bag This test ensures that the TileBag class accurately returns the correct number of expected tiles when the number of tiles remaining in the bag is less than seven.

D.2 Intra-move Rack Delta Resolution

Test initial snapshot (turn 1) This test verifies that the `RackDeltaResolver` correctly handles processing deltas at the start of the match when Player 1 draws 7 tiles during their turn.

Test redrawing delta resolution (7 expected) This test checks that valid deltas, which are supersets of the confirmed snapshot, are correctly processed, specifically when the expected number of tiles is 7.

Test redrawing delta resolution (<7 expected) This test is identical to the one above, but when the expected number of tiles is less than 7.

Test play delta resolution This test verifies the delta resolution functions correctly when playing tiles from the rack, with incoming deltas always subsets of the confirmed snapshot at the start of the turn.

Test error produced when drawing too many tiles This test validates that an error is produced when drawing an excessive number of tiles.

Test error produced when drawing too few tiles This test ensures that an error is produced when drawing too few tiles.

Test error produced on invalid delta (drawing) This test ensures that an error is generated when the incoming delta is not a superset of the previous rack when drawing tiles.

Test error produced on invalid delta (playing) This test ensures that an error is generated when the incoming delta is not a subset of the previous rack when playing tiles.

Delta with same tiles increases confidence This test checks that the `RackDeltaResolver` correctly increases the confidence in a delta when receiving multiple delta updates with the same tiles.

D.3 Intra-move Board Delta Resolution

Initial delta with empty board resolved correctly This test verifies that the initial delta received when the board is empty is correctly resolved.

Delta with different tiles to previous delta resolved correctly This test ensures that the `BoardDeltaResolver` correctly resolves a delta update with different tiles compared to the previous delta.

Delta with same tiles increases confidence This test checks that the `BoardDeltaResolver` correctly increases the confidence when receiving delta updates with the same tiles.

Test error produced when invalid delta received This test validates that an error is produced when the delta includes changes to already confirmed tiles on the board.

Test error when move is not valid This test ensures that the `BoardDeltaResolver` correctly identifies and handles the scenario where the move indicated by the delta is not valid.

Test correct identification of removed tiles from confirmed state This test verifies that the `BoardDeltaResolver` accurately identifies and handles the removal of tiles from the confirmed state on the board.

Test confirmed information in deltas are ignored This test checks that any already confirmed move information within a delta is ignored, providing it matches the confirmed move.

D.4 Inter-move Delta Resolution

Test successful delta resolution This test verifies that the inter-move resolution logic successfully resolves the deltas when the rack and board deltas match up.

Test error in delta resolution This test checks whether the inter-move resolution logic correctly identifies and handles the error when the rack and board deltas do not match up.

Test end of game This test ensures that the inter-move resolution logic correctly identifies the end of the game and computes the penalty for the relevant player correctly.

Test successful challenge This test validates that the inter-move resolution logic produces the correct error if the appropriate rack is not reverted to its previous state and the relevant tiles are not removed from the board after a successful challenge.

Appendix E

WESPA Tournament Rules

Part 1 – Equipment

1.1 Standard Rules

(a) These Rules apply in addition to the standard game rules ('Standard Rules'). The Standard Rules, which may change from time to time, are set out in Appendix 1.
(b) These Rules override the Standard Rules in the event of a discrepancy. Moreover:
  (i) games played under these Rules must be one on one, with both players keeping score; and
  (ii) games played under these Rules do not end if both players pass twice in succession.

1.2 Word Source

(a) The official word source, listed in Appendix 2, is endorsed by WESPA in consultation with the WESPA Dictionary Committee. It may change from time to time.
(b) Tournaments played under these Rules must not deviate from the official word source.

1.3 The Game Set

1.3.1 Tile Distribution

Both players must check before play that the set contains the correct number and distribution of tiles. Either player may request such a check. Once the game starts, corrections may not be made.

1.3.2 The Tiles

(a) Tiles that best achieve both tactile and visual indistinguishability are preferred.
(b) Any distinguishing marks (such as stickers) must be attached uniformly across the complete set of tiles.
(c) Sets free from tactile or visible irregularities caused by detachment from plastic moulding (especially on the top edges of tiles) are preferred.

1.3.3 The Board

Ordered by descending importance, the hierarchy of preferred attributes is:
(a) boards with edges measuring 33-35cm, which are rigid or can be made rigid for play;
(b) boards with indentations or ridges to prevent tiles from sliding;
(c) boards that do not obstruct a player's view of the opponent's rack;
(d) boards mounted on turntables that revolve with minimal disturbance to items on the playing table;
(e) boards with a non-reflective surface.

1.3.4 Other Equipment

(a) Players may use any rack they wish. However, the number of tiles on the rack must be clearly visible to the opponent.
(b) Tile bags must comfortably accommodate (simultaneously) the set of 100 tiles and a player's hand.

1.3.5 State of Equipment

All equipment in the game set must be in an acceptable state of repair. This includes:

(a) for tiles: clean, legible, not overly worn, hygienic;

(b) for boards: smoothly rotating (if applicable), free from excessively distracting background designs;

(c) for tile bags: opaque, not overly worn, of an appropriate size and design.

1.3.6 Varying the Equipment

Local exigency may at times require departure from the provisions in Rules 1.3.2-1.3.5. Tournament organisers should, however, make every effort to avoid this.

1.3.7 Disputes Over Equipment

The Tournament Director will resolve any disputes concerning equipment in the game set.

1.4 The Timer

1.4.1 Checking the Timer

Both players must check before play that the timer is set correctly and is working properly.

1.4.2 Precedence of Timers

(Note that in the list below, an 'optically passive' LCD screen is an LCD screen that does not emit light, such as the screen of a regular calculator)

If there is a choice of timers, the order of precedence is:

(a) digital timers with optically passive LCD screens with the following standard features:
  (i) countdown from the specified time limit to 00.00;
  (ii) display of overtime in minutes and seconds in a count-up fashion;
  (ii) neutralisation through the depression of a central button or designated area of the screen;

(b) smart device digital timers with a minimum diagonal screen size of 3.5 inches (89 millimetres) and sufficient power for expected use, with all the standard features above;

(c) digital timers with optically passive LCD screens capable only of counting up from 00.00 in such a way that overtime can be accurately calculated in minutes and seconds and which can be neutralised through the depression of a central button;

(d) analogue chess clocks.

- Other timing devices are not suitable but may be considered, at the discretion of the Director, if there is a shortage of suitable timers.
- In general, devices with a reputation of good reliability take precedence over those with a reputation of poor reliability.
- When smart device timers are used, it is recommended that every reasonable measure be taken to prolong battery life.
- All else being equal, a smart device with a larger screen takes precedence over one with a smaller screen.
- Any dispute over timing devices will be settled by the Director.

1.4.3 Neutralisation of the Timer

In these Rules, neutralising a timer means:

(a) for a digital timer: pressing a button or part of the screen whose purpose is to stop the countdown of both digital displays;

(b) for an analogue chess clock: depressing both clock buttons such that they are balanced and neither player's clock is ticking.

1.4.4 Use of a Timer is Mandatory

The use of a timer is mandatory for all games played under these Rules, though if there is a shortage of suitable timers, the Director will, using discretion, decide on a course of action.

1.4.5 Timer Position

The non-starting player may choose the position of the timer.

1.4.6 Malfunction of Timers

The malfunctioning timer must be stopped and the Director called. If the timer cannot be stopped or if the display has malfunctioned then both players must immediately write down their most accurate recollections of the amount of time left for each player at the moment of the malfunction. If the timer malfunctioned due to lack of power then the power source may be replaced or replenished and it may be used again. Otherwise, the timer must be removed from the competition, its owner must be notified, and it must be replaced with a suitable timer. The Director will, in conjunction with the players and, if necessary, any other observers of the game, determine as accurately as possible how much time each player had left. The Director will then assign each player the agreed remaining time on the replacement timer and the game will resume.

1.5 Written Aids

1.5.1 Score Sheets

Players may use either their own score sheets or those supplied by the tournament organisers. Score sheets may incorporate tile-tracking lists and may be double-sided.

1.5.2 Separate Tile-Tracking Lists

Players may prepare separate tile-tracking lists before a game, for use in addition to their score sheets. Such lists must not be designed as memory aids.

1.5.3 Acceptable Materials

The only visible papers allowed in the playing area are blank paper, contestant scorecards, blank and current game score sheets, tile-tracking lists, challenge slips, blank designation slips and result slips. All other papers must be kept invisible and must not be referred to during play. Records of previously played games must be stored in such a way that they are neither readable nor easily accessible (See also 1.6 Use of Electronic and Other Devices During Play).

1.5.4 Writing During the Game

There are no restrictions on what may be written on paper once the game begins.

1.6 Use of Electronic and Other Devices During Play

Apart from during adjudication of a challenge and with the exception of the timer, no electronic devices (including wearable devices) may be used by a player during play, unless for a justifiable reason related to health or simply for telling the time. Non-electronic devices that give an advantage to the user during play (such as calculating devices) are also not permitted. Prior to the start of the game and immediately after the game, electronic devices may be used in a way that does not distract other players. All electronic devices in a tournament venue must be set so that they cannot make distracting sound

Part 2 – Starting the Game

2.1 Determining Who Starts

(a) If no system to predetermine starts is in use, the players draw a tile each. The player whose tile is closest to the beginning of the alphabet, with a blank preceding an A, starts the game. In the event of a tie, each player draws again. No tiles are returned to the bag until the starter is decided. Once a starter is decided, the non-starter must return all tiles to the bag.

(b) Systems to predetermine starts must aim to ensure that all players in a tournament start approximately half their games. Such systems may include:
   (i) assignment of the start in each game by a tournament software program;
   (ii) 'self-balancing starts', in which the players compare their start/reply records before each game. If a player has hitherto started fewer games than his or her opponent, then that player starts. If the records are equal, the standard tile-drawing procedure is used.

(c) When self-balancing starts are in use, any player who knowingly misrepresents his or her start/reply record is considered to be cheating.

2.2 Starting the Timer

The timer of the player going first may be started once that player has removed a tile from the bag.

2.3 Late Arrivals

2.3.1 Duty to be Present

(a) All players must arrive by the scheduled starting time for each round.

(b) A player has officially arrived only when he or she is seated at the playing table ready to commence play immediately.

2.3.2 Both Players Absent

If neither player arrives by the scheduled starting time, the Tournament Director must:

(a) exercising due discretion, start one side of the timer to be used for the game;

(b) when the first player arrives, whether or not that player is due to play first, start the second side of the timer. The first player will be assigned the time showing on the first side of the timer;

(c) when the second player arrives, neutralise the timer. The second player will be assigned the time showing on the second side of the timer minus the time already deducted from the first player.

The game then proceeds as usual. No tiles may be drawn until both players arrive.

2.3.3 One Player Absent

If one player fails to arrive by the scheduled starting time, the Tournament Director must:

(a) exercising due discretion, start the late player's side of the timer;

(b) when the late player arrives, neutralise the timer. The player will be assigned the time showing on his or her side.

The game then proceeds as usual. No tiles may be drawn until both players arrive.

2.3.4 Optional Forfeiture due to Lateness
A late player whose timer has been started may elect to forfeit the game if his or her assigned game time, as calculated under Rule 2.3.2 or 2.3.3, is less than 15 minutes. For consequences see 2.3.6.

2.3.5 Compulsory Forfeiture due to Lateness
A player who fails to arrive before his or her assigned game time expires forfeits that game. For consequences see 2.3.6.

2.3.6 Consequences of Forfeiture due to Lateness
(a) A game forfeited under Rule 2.3.4 or 2.3.5 will count as a win for the opponent by a margin of 100 points. The Tournament Director may increase this margin if strategic lateness is suspected.
(b) A game forfeited due to lateness under Rule 2.3.4 or 2.3.5 will not count towards player ratings for the tournament.

2.4 Shuffling Tiles

Each player may shuffle the tiles within the bag before the game and while on turn. Shuffling tiles excessively and noisily, including hitting the bag on the table during shuffling, is distracting to fellow players and thus constitutes unethical behaviour (see Rule 6.3.1 (b)).

2.5 Special Needs / Disabilities

(a) Players must notify the Tournament Director, and, if relevant, the tournament organisers and venue of any special circumstances, such as physical impediments, that may affect their capacity to comply with any procedures set out in these Rules. Ideally, this notification should be done at least a week before a tournament.
(b) At the discretion of the Tournament Director, alternative procedures may be allowed or arranged to assist or accommodate players with special needs.
(c) At the discretion of the Tournament Director, a player with special needs may be awarded an amount of extra playing time, provided that this extra time will not interfere with the smooth running of the tournament, and with the proviso that the amount of time may be shortened if the tournament is so affected.

Part 3 – The Turn

3.1 Playing a Word

3.1.1 Elements of the Turn

To complete a turn by playing a word, a player must, in this order:

(a) place the tiles on the board (all blanks among said tiles must be properly declared according to rule 3.8 (Declaring a Blank), before the timer is pressed);
(b) announce the score for the turn (this may be computed aloud quietly);
(c) press the timer to start the opponent's time running;
(d) record the score for the turn and the cumulative score in the normal space on his or her score sheet;
(e) draw replacement tiles;
(f) tile track (if desired).

3.1.2 Writing Scores When No Tiles Remain in the Bag

If no tiles remain to be drawn, the writing of scores and cumulative scores is not a required element in completing a turn, so if one player wishes to confirm scores and the opponent has not recorded scores since the bag emptied, the timer may be stopped until both players agree on the scores.

3.1.3 Establishing Orientation

(a) The first play of the game determines the game's orientation with respect to the board's bonus square lettering. If this turn as played does not conform to the natural orientation of the bonus square lettering, then:
    (i) if the error is noticed by or is pointed out to the starting player before his or her turn has ended, then the starting player must correct the error on that player's own time, or;
    (ii) if the error is only noticed by the player going second after the second player’s timer has been started, then the second player may stop the timer and correct the error of orientation, after which the timer of the second player must be started by either player.
(b) If the first play of the game is misoriented but is not corrected before the end of the second move of the game, then the first play of the game determines the orientation of all plays for the rest of the game, so any plays after the first play which are misoriented relative to the first play may be challenged.

3.2 Exchanging Tiles

(For content relating to improper tile exchanges, refer to Rule 3.13.2 (Improper Tile Exchanges))

3.2.1 Elements of the Exchange

To complete a turn by exchanging tiles, a player must, in this order:

(a) check that the bag contains at least seven tiles;
(b) announce the exchange and the number of tiles to be exchanged;
(c) place the unwanted tiles face down on the table;
(d) press the timer to start the opponent's time running, after which no more unwanted tiles may be placed on the table, regardless of what the announced number of tiles to be exchanged was (note that if no tiles were placed on the table prior to pressing the timer then this constitutes a passed turn);
(e) record the exchange on the score sheet;
(f) draw the required number of replacement tiles, keeping them separate from the unwanted tiles;
(g) return the unwanted tiles to the bag;
(h) place the replacement tiles on the rack.

3.2.2 Exchange to Score Zero

An exchange of tiles scores zero points.

3.3 Passing

To complete a turn by passing, a player must, in this order:

(a) announce the pass;
(b) press the timer to start the opponent's time running;
(c) record the pass on his or her score sheet.

Note that pressing the timer so that the opponent can declare his or her blank on his or her own time does not count as a passed turn.

3.4 Significance of Pressing the Timer

3.4.1 Pressing the Timer Concludes Deliberation

(a) By pressing the timer in the course of playing a word, exchanging or passing, a player indicates a final choice of move. The move may not be changed after this act.
(b) A player may alter his or her choice of move at any point before pressing the timer.
(c) A player indicates a final choice of move only by pressing the timer, but if the player neglects to press the timer, final choice of move is indicated when the player places any part of a hand into the bag to draw tiles.
(d) By indicating a final choice of move as in (c), above, a player confers on the opponent an immediate right to challenge the turn.
(e) If the timer was pressed so that a player who just played an undeclared blank could declare the blank on that player's own time, it does not count as a passed turn.

3.4.2 Elements Overlapping with Opponent's Next Turn

(a) By pressing the timer in the course of playing a word, exchanging or passing, a player starts the opponent's next turn. Certain elements of the original turn may therefore overlap with elements of the opponent's next turn.

(b) If a player tile tracks before drawing replacement tiles, and the opponent is thereby delayed from drawing or counting tiles, the opponent may petition the Tournament Director for extra playing time.

(c) The Tournament Director will resolve any disputes concerning misordered turns. See also Rule 3.10.4 (Challenging an Improperly Ordered Turn / Timer Not Pressed After Play).

(d) Where tile drawing or tile counting by the opponent prevent the player on turn from immediately accessing the bag for the purpose of counting tiles, see 3.6.3 (Right to the Bag).

3.4.3 Unintentionally Pressing the Timer

If the timer has been pressed unintentionally, for example, when rotating the board or with a sleeve, the Tournament Director may be petitioned to nullify this action. If the petition is accepted, any time adjustments will be made at the discretion of the Tournament Director if deemed necessary. Play will then proceed as normal.

3.5 Keeping Score

(a) Until the bag is empty, both players must promptly record in the normal spaces on their score sheets both the score for each turn and the cumulative scores.

(b) Once the bag is empty, all further move scores and cumulative scores may be written after the timer is neutralised at the end of the game.

(c) Both players must verify the cumulative scores with reasonable frequency.

(d) Scoring errors may be corrected at any time prior to signing the result slip, though for correction of errors after the result slip is signed, see 5.4.1 (Result Slips Final Once Signed).

3.6 Prerogatives of the Player On Turn

3.6.1 Actions Reserved for the Player On Turn

A player may do the following things ONLY when it is his or her turn:

(a) adjust tiles on the board (errors of misorientation or imperfect placement of tiles may be pointed out to the player on turn, but may only be corrected by a player who is on turn);

(b) rotate or adjust the board; or

(c) ask to verify scores with the opponent, who must co-operate (keeping in mind that if the opponent has not written down the scores since the bag emptied, then the timer may be stopped until both players agree on the scores).

3.6.2 Actions Where the Player On Turn Has Priority

(a) The player on turn has priority for the following:
  (i) shuffling or counting the remaining tiles (see 3.6.3 Right to the Bag)
  (ii) checking the legality of an exchange.

(b) The player not on turn, if doing one of these things, must ensure that the player on turn is minimally disturbed by the act.

3.6.3 Right to the Bag

The player on turn has immediate right to the bag for the purpose of counting tiles and the opponent must promptly surrender the bag upon request, except if the opponent is still drawing tiles or if the opponent already has a hand in the bag and is busy counting tiles, in which case the player on turn may stop the timer until the bag is in the player on turn's possession, after which either player must restart the player on turn's timer.

3.7 Shuffling or Counting the Remaining Tiles

3.7.1 Procedure for Shuffling or Counting Tiles

To shuffle or count the remaining tiles, a player must, in this order:

(a) announce an intention to shuffle or count the tiles;
(b) show the opponent an empty hand (open palm with fingers stretched apart);
(c) hold the bag in a position acceptable for tile-drawing while shuffling or counting (see 3.9.1);
(d) show the opponent an empty hand after shuffling or counting.

3.7.2 Right to Object to Opponent Shuffling Tiles

A player may, only for a legitimate reason, object to the opponent shuffling or counting the remaining tiles. If this occurs, a tournament official may shuffle or count the tiles while the timer is neutralised, notifying both players of the result of the count.

3.8 Declaring a Blank

(a) Blanks must be declared preferably by circling a printed letter or else by writing a capital letter on a neutral sheet of paper, which must remain in clear view of both players for the duration of the game. If neither of the above papers are available for designation of the blank, then the timer may be stopped until one is procured. If a blank has been declared but the opponent is still not certain of the actual designation of the blank, then the opponent may stop the timer and demand that the player repeat the declaration of the blank. Neither oral declarations nor players' records on their personal papers are determinative.

(b) A player who plays a blank must declare it as in (a), above, BEFORE pressing the timer. If a player ends the turn without correctly declaring a blank, the opponent may immediately restart that player's timer and demand that the blank be properly declared. Pressing of the timer by the opponent in this instance does not count as a passed turn.

(c) If the identity of an improperly declared blank that was played on an earlier turn is disputed then the Director must be called. If the Director agrees that the improperly declared blank could have been legitimately mistaken for another letter by the opponent, then the opponent may declare the improperly declared blank to be that letter. All words formed that include this newly declared blank may be challenged.

(d) If a blank is properly declared and its identity is nonetheless disputed at any later time, the Director must be called. The Director will decide if there is a legitimate misunderstanding of the identity of the blank and may permit a move that has just been played based on a misapprehension of the blank's identity to be replayed.

3.9 Drawing Tiles

3.9.1 Bag Position

When drawing tiles, a player must:

(a) hold the tile bag so that its rim is at or above eye level;
(b) avert his or her eyes from the tile bag; and
(c) keep the tile bag in full view of the opponent.

3.9.2 Drawing Protocols

(a) Players need not draw tiles individually.
(b) Players must not put a hand containing tiles into the tile bag. All drawn tiles must be placed on the rack or the table before further tiles are drawn.
(c) Players must show an empty hand both before and after drawing.
(d) Tiles must be drawn with reasonable speed.

3.9.3 Keeping Tiles Above the Table

Players must keep all tiles above the level of the playing table at all times.

3.9.4 Improper Drawing

The Tournament Director will resolve any disputes concerning the propriety of tile drawing.

3.9.5 Overdrawing

If a player draws too many replacement tiles ('overdraws'), the timer must be neutralised and the overdraw corrected as follows:

(a) if NONE of the newly drawn tiles have touched the overdrawing player's rack then:
    (i) the overdrawing player places ONLY the newly drawn tiles face down on the table and shuffles them randomly;
    (ii) if the overdrawing player has 6 tiles on the rack, then the opponent turns <u>all</u> the newly drawn tiles face up, and proceeds to step (iv) of 3.9.5(a);
    (iii) if the overdrawing player has 5 or fewer tiles on the rack, then the opponent turns face up X+2 of the newly drawn tiles, where X is the number of OVERDRAWN tiles;
    (iv) from the face-up tiles, the opponent chooses X tiles and returns them to the bag;
    (v) all remaining tiles are returned to the overdrawing player, leaving that player with the correct number of newly drawn tiles to add to his or her rack.

(b) if AT LEAST ONE newly drawn tile has touched the overdrawing player's rack then:
    (i) the overdrawing player must place the newly drawn tiles AND all his or her other tiles face down on the table and intermix them;
    (ii) where X is the number of overdrawn tiles, the opponent turns face up X+2 tiles;
    (iii) from the face-up tiles, the opponent chooses X tiles and returns them to the bag;
    (iv) the remaining tiles are returned to the overdrawing player, leaving that player with a total of seven tiles to place on his or her rack.

The opponent has 15 seconds to correct the overdraw. If a correction has not been made, the opponent's timer is started. There is no time limit to making a correction except when it exceeds the assigned game time by 10 minutes (see 5.3.3 Overtime Leading to Forfeiture). Once a correction has been decided, the timer is neutralised, and the chosen tiles are returned to the bag. The remaining tiles are returned to the overdrawing player. Play resumes with the player on turn's clock being restarted.

3.9.6 Improperly Corrected Overdraws

If an opponent correcting an overdraw turns too many tiles face up, all exposed tiles must be replaced face down. The opponent then repeats subsection 3.9.5(a)(iii) or (b)(ii) as necessary, but may turn face up only X tiles, and must return those X tiles to the bag.

3.9.7 Duty to Disclose Overdraw

A player who becomes aware that he or she has overdrawn must disclose the overdraw. Non-disclosure is regarded as unethical behaviour (see 6.3 (Level 2 Offences)).

3.9.8 Late-Game Underdrawing

(a) This rule applies if a player underdraws, and the opponent empties the bag in his or her next draw.

(b) If the underdraw is discovered before the player completes his or her next turn, the opponent chooses and gives to the player the appropriate number of tiles from his or her rack.

(c) If the underdraw is discovered only after the player completes his or her next turn, there is neither a correction for the mistake nor a penalty.

(d) Late-game underdrawing is regarded as unethical.

3.9.9 Drawing Out Of Order

(a) If the out of order draw occurs before the opponent has had a reasonable chance to draw replacement tiles, AND leaves fewer tiles in the bag than the opponent would have rightfully drawn, then all of the player's newly drawn tiles are treated as overdrawn tiles to which the following procedure applies:

  (i) the overdraw procedure given in Rule 3.9.5 (Overdrawing) is followed;

  (ii) the opponent draws as many replacement tiles as are needed to complete his or her own draw;

  (iii) any tiles remaining in the bag are replaced on the player's rack.

(b) If the out of order draw does not contravene (a), above, then:

  (i) if any of the tiles drawn out of order have touched the rack of the overdrawing player, then the other player has been too slow to notice the out of order draw and the overdrawing player may replenish the rack without penalty.

  (ii) If a player notices that the opponent has drawn out of order before any of the drawn tiles have touched the rack, then that player must call a halt to the out of order draw and stop the timer. The tiles drawn out of order must be shown to both players and returned to the bag, after which the correct order of drawing must be followed.

3.9.10 No Tile Drawing While Awaiting Adjudication

Players must not draw tiles while awaiting the adjudication of a challenge.

3.10 Accepting and Challenging Turns

3.10.1 Accepting the Turn

(a) Once a player presses the timer under Rule 3.1.1(c), the opponent may:
  (i) issue an immediate challenge (see Rule 3.11 (Procedures for Issuing and Adjudicating a Challenge));
  (ii) call 'hold' (see Rule 3.10.5 (Holds));
  (iii) choose to accept the turn without calling 'hold' or issuing a challenge.

(b) The opponent accepts the turn if he or she neither calls 'hold' nor issues a challenge before any replacement tile drawn by the player is added to the player's rack.
  (i) Once the tile bag is empty, and there are no further tiles to be drawn, there is no time limit with regard to a player accepting a turn and issuing a challenge. If a player goes over their assigned game time by 10 minutes, they will forfeit the game (see 5.3.3 Overtime Leading to Forfeiture).
  (ii) Where a player has 'played out' and the opponent has not, after five seconds, taken one of the three actions outlined in Rule 5.1.2 (Actions to be Taken Upon Playing Out), the opponent's timer is restarted until one of these actions is taken or they forfeit the game due to going overtime (see 5.1.3 Right to Restart the Timer and 5.3.3 Overtime Leading to Forfeiture).

(c) Accepting a turn waives the right to challenge that turn.

(d) Writing by the opponent does not affect acceptance of a turn.

3.10.2 Flash-Drawing

(a) If the player fails to record scores as required by Rule 3.1.1(d) before drawing a replacement tile, or if the player pre-writes the scores, he or she has flash-drawn. The opponent is not considered to have accepted the turn, and may challenge even after a replacement tile is drawn.

(b) Flash-drawing constitutes unethical behaviour (see Rule 6.3.1 (Definition of Unethical Behaviour)).

(c) If a turn is successfully challenged after a flash-draw, then:
  (i) if no flash-drawn tile has touched the player's rack, all flash-drawn tiles are revealed to the opponent and returned to the bag;
  (ii) if a flash-drawn tile has touched the player's rack, the player is overdrawn by the number of tiles drawn in the flash-draw, and Rule 3.9.5 (Overdrawing) applies.

3.10.3 Issuing and Adjudicating a Challenge - (see 3.11 (Procedures for Issuing and Adjudicating a Challenge))

3.10.4 Challenging an Improperly Ordered Turn / Timer Not Pressed After Play

A player who omits to press the timer while making a turn completes that turn by placing any part of a hand in the bag to draw replacement tiles. As soon as this occurs, the opponent may:

(a) compel the player to press the timer immediately, if he or she has not yet done so; and

(b) issue a challenge according to 3.11 (Procedures for Issuing and Adjudicating a Challenge).

3.10.5 Holds

(a) A player considering a challenge must call 'hold', thereby warning the opponent not to draw replacement tiles. The player may take any amount of time to accept or challenge the play after calling 'hold', provided that the amount of time taken does not cause the player to exceed his or her allotted game time by more than 10 minutes (see 5.3.3 Overtime Leading to Forfeiture).

(b) Unambiguous words such as 'accept' or 'okay' must be used to release a hold.

3.10.6 Courtesy Draws

(a) A player whose opponent has called 'hold' may, thirty seconds after pressing the timer, draw and look at replacement tiles. These tiles must be kept separately from the player's rack.

(b) If a challenge is upheld after a courtesy draw, the replacement tiles must be seen by the opponent and returned to the bag. The player is not considered to have overdrawn.

(c) If a challenge is upheld after a courtesy draw and the replacement tiles have (contrary to section (a)) been intermixed with the player's old tiles, the player is considered to have overdrawn, and Rule 3.9.5 (Overdrawing) applies.

3.10.7 Amount of Time Allowed to Challenge When a Player Has Played Out

Once a player has made a move that will end the game if left unchallenged, the opponent may take as much time as desired to challenge that move, provided that the amount of time taken does not cause the player to exceed his or her allotted game time by more than 10 minutes (see 5.3.3 Overtime Leading to Forfeiture). If the opponent does not immediately accept the 'out play' by revealing his or her unplayed tiles as per rule 5.1.2, then after 5 seconds of deliberation by the opponent, the player may restart the opponent's timer.

3.10.8 No Retraction or Concession of a Challenge

(a) A player who verbally expresses an unambiguous intention to challenge AND neutralises the timer is compelled to challenge.

(b) The challenger may change his or her mind about which word or words to challenge at any time before:

  (i) if self-adjudicating via smart device, any letter of any word to be challenged has been typed into the adjudication program;

  (ii) if self-adjudicating at an external device, the challenging player leaves the playing table;

  (iii) if runners are used, the challenge slip is handed to the runner.

(c) A player whose turn is challenged may not concede the challenge prior to adjudication.

3.10.9 Rechallenging

(a) Either player may request the re-adjudication of a challenge.

(b) If such a request is made, in the case of External Adjudication, the original adjudicator should not perform the re-adjudication.

(c) The re-adjudication is final unless it differs from the original adjudication, in which case the Tournament Director will provide a final adjudication.

3.10.10 Erroneous Challenges

If it is discovered that a word written on a challenge slip or entered into the adjudication program does not correspond to a word played on the board in the most recent turn, the challenge may be reissued (subject to 3.10.11 below).

3.10.11 Mis-adjudication of a Challenge

If a move is challenged, and the challenge is discovered to have been mis-adjudicated, the error may be corrected if and only if:

(a) no newly drawn tiles have touched the player's rack, or

(b) no retracted tiles have touched the player's rack.

Otherwise, play continues as normal and no account is taken of the error.

3.10.12 Board Control During Challenge

When the timer is neutralised pending an adjudication, the player whose turn has been challenged retains control of the board.

3.10.13 Challenge Penalties

(a) A player whose turn is successfully challenged loses that turn. The challenger may be penalised only if all challenged words are acceptable.

(b) The penalty for an unsuccessful challenge may vary from tournament to tournament. The following penalty conditions are considered standard:
   (i) five-point penalty per unsuccessfully challenged word (this is the preferred international norm);
   (ii) five-point penalty per unsuccessfully challenged turn;
   (iii) loss of turn ('double challenge').
   (iv) no penalty ('single challenge' or 'free challenge');
   (v) as in subsection (i) or (ii), but using ten point penalties.

(c) Other penalty conditions are not considered standard, and tournaments using non-standard penalty conditions may be considered non-rateable by WESPA. Examples are:
   (i) no penalty for first unsuccessful challenge, loss of turn for subsequent unsuccessful challenges ('dingle challenge');
   (ii) five-point penalty for first unsuccessful challenge, ten-point penalty for subsequent unsuccessful challenges;
   (iii) -5,-10,-20,-30 point (or similar) increasing penalties for unsuccessful challenges;
   (iv) time penalties.

3.11 Procedures for Issuing and Adjudicating a Challenge

A challenge may be adjudicated by the players themselves (Self-Adjudication) or by an external person (External Adjudication). Self-Adjudication is the preferred international norm, though players with legitimate reasons may use External Adjudication. In the case of Self-Adjudication, both players take responsibility for ensuring that the word source is correct and up to date. The Director may not forbid the use of self-adjudication via smart device and must inform participants before the start of a tournament which other method/s of adjudication will be allowed.

After adjudication by any method below, the timer may not be restarted until both players are seated, the move score has been re-announced or the tiles retracted, and the player whose timer would be running after the challenge has either: returned all face down tiles to their rack (recommended); or commenced with any writing on their scoresheet.

3.11.1 Self-Adjudication

(a) Self-Adjudication at an External Device
   (i) the challenger verbally expresses an unambiguous intention to challenge;
   (ii) the challenger neutralises the timer;
   (iii) the challenger clearly informs the opponent which word/s are being challenged;
   (iv) to minimise errors, it is advised that either player writes the word/s down on a challenge slip;
   (v) both players cover or turn the tiles on their racks face down and proceed to the adjudication device;
   (vi) the challenger types in the word/s being challenged into the adjudication program;
   (vii) the opponent verifies that the word/s are correctly typed and executes the adjudication command.

(b) Self Adjudication via Smart Device

(i) the challenger verbally expresses an unambiguous intention to challenge;
(ii) the challenger neutralises the timer;
(iii) the challenger clearly informs the opponent which word/s are being challenged;
(iv) the challenger types in the word/s being challenged into the adjudication program;
(v) the opponent verifies that the word/s are correctly typed and then gives verbal consent for the challenger to execute the adjudication command.

Note that the use of a smart device for self-adjudication must be agreed to by both players. The following requirements with regard to smart device adjudication are mandatory:

- The minimum screen size must be at least 3.5 inches (89 millimetres) measured diagonally.
- The device must be set so that it cannot make distracting sounds or vibrations.
- The device must be used in adjudication mode and may not be used in other modes.
- While in use, the screen must remain visible to both players.
- The device's adjudication program must be capable of accepting multiple words to be challenged at a time.

3.11.2 External Adjudication

(a) External Adjudication Using Runners

(i) the challenger verbally expresses an unambiguous intention to challenge;
(ii) the challenger neutralises the timer;
(iii) the challenger clearly informs the opponent which word/s are being challenged, and must record the word/s legibly on a challenge slip;
(iv) both players must cover or place face down any tiles they may have and the challenger calls for a runner;
(v) the runner takes the challenge slip to the External Adjudicator;
(vi) the External Adjudicator carefully checks the acceptability of the word/s on the challenge slip using the correct software or printed word list, then;
(vii) places a single tick on the challenge slip if all challenged words are acceptable, or a single cross if at least one is not, and returns the slip to the runner.
(viii) When multiple words are challenged, runners and adjudicators must not reveal to players the acceptability of individual words.
(ix) If docket printers are used to print the results of challenges, the printout may be returned to the players in lieu of the original challenge slip.

(b) External Adjudication via Self-Running

The procedure runs as for 3.11.2 (a) (External Adjudication Using Runners), just above, except that the challenger takes the place of the runner.

(c) External Adjudication via Tournament Director's Smart Device

(i) the challenger verbally expresses an unambiguous intention to challenge;
(ii) the challenger neutralises the timer;
(iii) the challenger clearly informs the opponent which word/s are being challenged;
(iv) the Director is called;
(v) the Director types the word/s to be challenged into the adjudication program, shows both players that the words have been typed in correctly, then, after verbal consent from the challenged player, executes the adjudication command.
(vi) the Director may choose to let the players use the Director's device themselves, in which case the procedure runs exactly as in 3.11.1(b) (Self Adjudication via Smart Device).

3.12 Correcting Errors of Misoriented or Imperfectly Placed Tiles

Once orientation of the game has been established (see 3.1.3 (Establishing Orientation)), errors of misoriented or imperfectly placed tiles may be corrected by the opponent as follows:

(a) While the player at fault is still on turn, the opponent may not physically correct the error, but may point out the error.

(b) If the error has not been corrected by the player at fault and that player's turn has ended, then the opponent may stop the timer and correct the error, after which either player may restart the opponent's timer.

Note that playing a tile in a misoriented position or imperfectly placing a tile is considered poor etiquette (see 6.4.1).

3.13 Illegal Moves

3.13.1 Challenging Word Placement

(a) A player may challenge a turn on the grounds that a word has been placed illegally. Illegal word placements include, but are not limited to:
  (i) failure to cover the centre square on the opening play;
  (ii) placing tiles such that the tiles do not all form part of one word;
  (iii) playing a diagonal word;
  (iv) playing a word that extends beyond the 15x15 grid;
  (v) playing a misoriented word after orientation has been established (see Rule 3.1.3 (Establishing Orientation)).

(b) A player wishing to challenge an illegal word placement must neutralise the timer and call the Tournament Director to adjudicate.

(c) There is no penalty for an unsuccessful challenge.

(d) A player is free to refrain from challenging an illegal word placement. In the case of subsection (a)(i), above, if a player so refrains, the centre square retains its double-word-score value for subsequent turns.

3.13.2 Improper Tile Exchanges

(a) A player wishing to challenge an improper or illegal exchange must neutralise the timer and call the Tournament Director to adjudicate.

(b) If an exchange is announced but no number of tiles is specified and no tiles are placed on the table before the timer is pressed, then the turn counts as a pass.

(c) There is no penalty for putting the old tiles into the bag before drawing new tiles.

(d) If the exchanger fails to put the face-down tiles back into the bag, and any new tiles in a turn after the exchange are drawn by either player before the discovery is made, then said face-down tiles must be seen by both players before being returned to the bag.

(e) If the number of tiles placed face down does not equal the number of tiles announced, then the number of face down tiles at the moment the exchanger presses the timer shall be exchanged.

(f) There is no penalty for an unsuccessful challenge of a tile exchange.

(g) A player is free to refrain from challenging an illegal exchange.

(h) An illegal exchange may be challenged at any time before the exchange has been completed according to the steps in Rule 3.2.1 (Elements of the Exchange).

(i) If an exchange is announced and the timer has been pressed but there are less than 7 tiles in the bag then the timer must be neutralised and the following algorithm applies:

(i) If the illegal exchange is only noticed after the non-exchanger has made an accepted move, or passed, or lost a challenge, then the player who made the illegal exchange will have 30 points deducted from that player's game score.

(ii) If NO NEWLY DRAWN TILE has touched the rack and NO ORIGINAL TILES have been placed into the bag, then the exchanger passes the turn. All newly drawn tiles must be shown to the opponent and returned to the bag, while all original tiles are returned to the exchanger's rack.

(iii) If NO NEWLY DRAWN TILE has touched the rack but AT LEAST ONE ORIGINAL TILE has been placed into the bag, then any original tiles on the table must be returned to the rack. The opponent exposes all newly drawn tiles and all tiles in the bag. Within one minute, the opponent replenishes the rack of the exchanger with a tile or tiles of the opponent's choice and the exchanger passes the turn. All remaining tiles on the table are returned to the bag.

(iv) If AT LEAST ONE NEWLY DRAWN TILE has touched the rack, but NO ORIGINAL TILES have been put into the bag, then the opponent exposes all tiles set aside to be exchanged plus all tiles from the exchanger's rack. Within one minute, the opponent chooses 7 of the aforementioned tiles to return to the exchanger's rack, and the rest are returned to the bag. The exchanger passes the turn.

(v) If AT LEAST ONE NEWLY DRAWN TILE has touched the rack and AT LEAST ONE ORIGINAL TILE has been put into the bag then the opponent exposes all tiles on the exchanger's rack, all tiles in the bag and any tiles set aside to be exchanged. From these tiles the opponent chooses 7 (within one minute) to be returned to the exchanger's rack, with the remainder being returned to the bag. The exchanger passes the turn.

3.13.3 Exchanging from Racks of Eight or More Tiles

After announcing an exchange and starting the opponent's timer, if a player is discovered to have had a rack containing eight or more tiles, the following algorithm is to be applied:

(a) If no new tiles have been drawn, the overdraw procedure is applied and the offender loses their turn, i.e. cannot exchange.
(b) If the overdraw is discovered after any new tiles are drawn, but before any tiles have been returned to the bag: all exchanged, kept, and newly drawn tiles are combined. The overdraw procedure will then be applied (see part (b) of 3.9.5 Overdrawing).
(c) If the overdraw is discovered after any tiles have been returned to the bag: combine the originally kept tiles, the replacement tiles, any tiles not yet returned to the bag and the newly drawn tiles. The overdraw procedure will then be applied (see part (b) of 3.9.5 Overdrawing).

3.13.4 Exchanging from Racks of Six or Fewer Tiles

If a player is found to have exchanged from a rack of six tiles or fewer, one of the following steps is taken:

(a) If there would have been seven or more tiles in the bag had the exchanger held a full rack, play continues with no penalty to the exchanger.
(b) If there would have been fewer than seven tiles in the bag had the exchanger held a full rack, implement rule 3.13.2 (i) (Improper Tile Exchanges).

Part 4 – Interrupting the Game

4.1 Neutralising the Timer

The timer may be neutralised during the game for the following reasons:

(a) to issue a challenge;
(b) to resolve a scoring discrepancy;
(c) to correct an overdraw;
(d) to ascertain the game time assigned to a late player;
(e) to call the Tournament Director to resolve a problem;
(f) to deal with an unforeseen event such as a power failure or a spillage of liquid; or
(g) to follow any other procedure which requires neutralisation under these Rules.

4.2 Leaving the Playing Area

(a) Players must obtain the Tournament Director's permission to leave the playing area during a game.
(b) If permission is obtained, the Tournament Director will supervise the following procedure:
    (i) the player wishing to leave must complete a turn, except for drawing replacement tiles;
    (ii) the player may then leave the playing area;
    (iii) while the player is absent, the opponent may complete a turn, except for drawing replacement tiles.
(c) In an emergency, players may leave the playing area without obtaining permission. The opponent must alert the Tournament Director immediately if this occurs (see also 4.9 (Emergencies and Medical/Health Problems)).
(d) Supervision of players who leave the playing area is at the discretion of the Tournament Director. An opponent may request but may not compel supervision.

4.3 Tiles Discovered Out of the Bag

If any tiles (other than those properly in a player's possession) are discovered outside the bag at any time before the result slip has been signed, then:

(a) both players see the tiles;
(b) both players check to ensure that the tiles were not accidentally displaced from the board, especially from the corners and edges (once this is agreed, the board may not be subsequently corrected);
(c) the tiles are returned to the bag;
(d) any tiles removed from players' racks in the belief that the game was over are replaced; and one of the following steps is taken:
    (i) if both players have seven tiles, play resumes as usual;
    (ii) if only one player has seven tiles, that player's opponent draws from the bag; or
    (iii) if neither player has seven tiles, the players ascertain who should have drawn replacement tiles earliest, and that player draws from the bag. If only one player has tiles after this is done, the game is over and the result is recalculated as necessary. Under no circumstances may any moves be replayed.

4.4 Spilled Tiles

If tile/s are spilled from the bag during a game then the timer is neutralised, any tiles on racks are covered, and the Director is called. Players may look at spilled tiles which landed face-up, but may not touch spilled tiles before the Director rules on the spillage.

(a) Spillage in the period before and including replenishing of the rack:

(i) this is managed by rule 3.9.5 (Overdrawing).

(b) Spillage just after the rack has been replenished:

(i) the spilled tiles are turned face up on the table;

(ii) the non-spiller has the option to decline to invoke the overdraw procedure in rule 3.9.5(b), if s/he believes that there is a chance that the spiller may potentially benefit from said procedure, and the Director affirms that said belief is legitimate.

(c) Spillage not related to drawing tiles, with 7 or more tiles in the bag:

(i) this type of spillage is generally deemed to be accidental and carries no penalty; however,

(ii) the non-spiller may petition the Director to scrutinise the circumstances of the spillage, after which management is at the discretion of the Director, with the maximum penalty being invoking the overdraw procedure in rule 3.9.5(b).

(d) Spillage not related to drawing tiles, with less than 7 tiles in the bag:

(i) the Director places the spilled tile/s face up on the table;

(ii) the spiller's rack tiles are placed face down on a separate part of the table and s/he shuffles them randomly;

(iii) the non-spiller chooses two of said rack tiles, turns them face up and adds them to the spilled tile/s;

(iv) the non-spiller chooses any two of the face-up tiles and gives them to the spiller;

(v) the remaining tiles are returned to the bag.

4.5 Upset or Overturned Boards

The Tournament Director must be called immediately and the board and any scattered tiles must not be touched until they arrive. Any tiles knocked off a player's rack may be quickly covered or turned face down to prevent the opponent viewing them. After hearing the testimony of both players and any witnesses of the event, the Tournament Director must decide whether the board was upset accidentally or intentionally.

4.5.1 Accidental Board Upsets

If this occurred due to a table collapse, then all tables must immediately be checked for stability and correct extension of legs.

(a) If the game can be fully reconstructed from recorded plays (subject to approval by the Tournament Director to minimise any impact on tournament scheduling), this must be done and normal play resumes. If the Tournament Director rules that the game cannot be reconstructed due to time constraints, follow the procedure in 4.5.1 (b)).

(b) If the game cannot be fully reconstructed, the game will end. The final game scores will stand at the point at which the board was upset. Where the number of turns played by each player is uneven, the score for the last turn made is subtracted from the appropriate player's total score. Any disputes concerning final game scores will be resolved by the Tournament Director.

4.5.2 Intentional Board Upsets

Upsetting a board intentionally is considered a very serious offence and results in an immediate loss for the offending player. The offending player will also be ejected from the tournament. The spread for the lost game is calculated as follows:

(a) If the offending the player was not in the lead, they will lose the game by the current margin plus 100 points.
(b) If the offending player was in the lead, they will lose the game by a margin of 100 points.

4.6 Tiles Discovered In the Bag After the End of the Game

If any tiles are discovered in the bag, which the players had thought to be empty, before the result slip has been signed, then:

(a) both players see the tiles;
(b) any tiles removed from players' racks in the belief that the game was over are replaced; and
(c) the players ascertain who should have drawn replacement tiles earliest, and that player adds the tiles to his or her rack.

If both players still have tiles after this process, play resumes. If only one player has tiles, the game is over and the result recalculated as necessary. Under no circumstances may any moves be replayed.

4.7 Tiles Noticed to be Missing During Play

If a player notices that a tile has gone missing during play then the Director must be called to confirm this. If confirmed, a thorough search of the surrounding area must be performed, within reasonable limits. If the tile is found then refer to 4.3 (Tiles discovered Out of the Bag). If the tile cannot be found, the game must continue as if the tile was never in the bag, and after the game it must be ensured that the affected table has a full set of tiles for the next round.

4.8 Scope of Uninvited Intervention by the Director in a Game

(a) the Tournament Director may not intervene in the event of any mathematical error/s noticed by the Tournament Director or a third party;
(b) the Tournament Director must intervene when s/he notices cheating, but if the Tournament Director is made aware of alleged cheating by a third party, then s/he must investigate the allegation/s and take action as necessary, using discretion.
(c) The Tournament Director must intervene in any noticed error of procedure that leads to one player being unfairly affected (e.g. an error in challenging procedure).

4.9 Scope of Intervention by a Third Party in a Game

(a) a third party may not intervene DIRECTLY in an observed game in any way, including in errors of mathematics, errors of procedure and even cheating noticed by the third party;
(b) instances of cheating noticed by a third party must be reported to the Tournament Director, who must then investigate the allegation/s and take action as necessary, using discretion;
(c) errors of procedure noticed by a third party should be reported to the Tournament Director.

4.10 Emergencies and Medical/Health Problems

In the event of an emergency (whether medical or not), the affected player's timer must be stopped and the Tournament Director must be called. The Tournament Director must quickly assess the emergency, decide on a course of action and, in order to preserve the smooth running or the tournament, immediately restart the affected player's timer. If the affected player feels sufficiently recovered within the game time left for that player then the player may attempt to finish the game, otherwise the affected player resigns the game due to the emergency (see 5.7 Resigning).

Part 5 – Ending the Game

5.1 'Playing Out'

5.1.1 Procedure for 'Playing Out'

'Playing out' occurs when, after completing a turn, a player has no tiles remaining and no tiles remain to be drawn from the bag.

5.1.2 Actions to be Taken Upon Playing Out

A player attempting to play out must neutralise the timer, rather than starting the opponent's timer.

The opponent must then either:

(a) accept the turn by revealing his or her unplayed tiles;
(b) call 'hold'; or
(c) challenge the turn.

5.1.3 Right to Restart the Timer

(a) If a player has attempted to play out, and the opponent fails to accept the turn within approximately five seconds, then the player is entitled to restart the opponent's timer while awaiting the opponent's action.
(b) If an opponent's timer is so started, the opponent must neutralise the timer after deciding either to accept the turn or to challenge.

5.1.4 Tiles Remaining

(a) When one player has played out, then either:
    (i) his or her score is increased by twice the value of the opponent's unplayed tiles, and the opponent's score is unchanged (this is the procedure recommended by WESPA); OR
    (ii) his or her score is increased by the value of the opponent's unplayed tiles, and the opponent's score is commensurately decreased.
(b) If neither player is able to play out then refer to 5.2 (Six Consecutive Zero Scores End the Game)) just below.

5.2 Six Consecutive Zero Scores End the Game

The game ends after six consecutive turns scoring zero, resulting from any combination of passes, exchanges and successful challenges. If this occurs, each player's final score is reduced by the total value of the tiles on his or her rack.

5.3 Time Penalties

5.3.1 Ascertaining When Time Penalties Apply

A player who exceeds his or her assigned game time incurs time penalties. This occurs once:

(a) the player's timer shows -00.01 (in the case of a digital count-down timer);
(b) the player's timer shows xx.01 (in the case of a digital count-up timer, where xx represents the assigned game time in minutes); or
(c) the flag on the player's side has dropped (in the case of an analogue chess clock).

5.3.2 Application of Time Penalties

A player's score is reduced by 10 points per minute or part thereof (measured in full seconds, and NOT fractions of a second) by which he or she exceeded the assigned game time.

5.3.3 Overtime Leading to Forfeiture

A player who goes over the assigned game time by 10 minutes immediately loses the game (a forfeit loss).

Note that the NON-FORFEITER's final score remains unchanged.

To work out the score to be recorded on the result slip for the forfeiter, first deduct 100 overtime penalty points from the forfeiter's game score, and then:

(i) if, after the deduction from the forfeiter, the NON-FORFEITER has won by 100 points or more, then the forfeiter's score to be recorded on the result slip is the forfeiter's game score minus the 100 overtime penalty points;

(ii) if, after the deduction from the forfeiter, the NON-FORFEITER has NOT won by 100 points or more, then the forfeiter's score to be recorded on the result slip is calculated by deducting 100 points from the NON-FORFEITER's score.

(Note that in the unlikely event of the forfeiter's final score being a negative number after the calculations above, 100 points must be added to each player's score.)

5.3.4 No Additional Time Penalties When Timer Not Neutralised

(a) If the timer is improperly left running at the end of the game, any time penalties that accrue beyond the point at which the timer should have been neutralised are disregarded.

(b) If a player fails to neutralise the timer when playing out, the opponent is taken to neutralise the timer by revealing his or her unplayed tiles.

5.3.5 Standard Game Time

An assigned game time of 25 minutes is considered standard. Tournaments using different times may be regarded by WESPA as non-rateable.

5.4 Result Slip

5.4.1 Result Slips Final Once Signed

A game result slip signed by both players is final, and binds the players and the Tournament Director, unless:

(a) before submitting the sheet, both players agree to correct an error on it;

(b) after submitting the slip, one or both players petition the Director to correct ONLY an error where the final game scores of the winner and loser were accidentally reversed. The Tournament Director must make every reasonable effort to correct this error, but may refuse if doing so will affect the draw or the smooth running of the tournament in a way that s/he deems to be unacceptable or overly complicated. The Tournament Director must ensure that both players are made aware of the amendment to the result slip.

5.4.2 Responsibility of Winner

The winner must hand in the result slip before leaving the playing area and assist with the Tile Check (see 5.6 Tile Check).

5.5 Recounts

5.5.1 Right to Recount
Either player may request a recount at the end of a game, regardless of spread differential, and with the agreement of the Tournament Director.

5.5.2 Recount Procedure
Partial recounts are ONLY acceptable with both players' consent. If one player refuses, a game must be recounted in full, or not at all. The timer remains neutralised during a recount.

5.5.3 Surrender of Score Sheet
A recounting player may request the use of the opponent's score sheet. The opponent may object, but must, if asked, surrender the score sheet to the Tournament Director, who may use it to assist the recounting player.

5.5.4 Tournament Director's Discretion
(a) Since recounts can interfere with tournament scheduling, the Tournament Director may halt a recount if he or she believes it is frivolous or has taken an excessive time.
(b) If the Tournament Director believes that a player is frivolously recounting or deliberately slowing the progress of a recount, then he or she may direct that no changes in that player's favour be made as a result of the recount.

5.6 Tile Check

Before leaving the playing area, BOTH players must ensure that the tiles are left on the board in preferably four 5x5 grids or one 10x10 grid.

5.7 Resigning

(a) A player may not resign a game except in an emergency.
(b) A resigned game is forfeited and cannot be resumed.
(c) The game margin in a properly resigned game is the greater of the following:
    (i) 50 points,
    (ii) the non-resigning player's lead at the time of resignation plus 50 points.
(d) The Tournament Director will determine an appropriate margin for an improperly resigned game.

Part 6 - Conduct

6.1 General Conduct

6.1.1 Expected Standards

WESPA expects players to uphold high standards of both etiquette and ethics. Respectful, courteous, fair and honest play is required. Players are honour bound not to cheat.

6.1.2 Tournament Director's Powers and Responsibilities (see also 4.7 Scope of Uninvited Intervention by the Director in a Game)

(a) In disputes concerning conduct, the Tournament Director's ruling is final.
(b) The Tournament Director must give each player a fair hearing, including (where relevant) taking the testimony of witnesses.
(c) The Tournament Director must resolve factual disputes upon the balance of probabilities.
(d) The Tournament Director may take the smooth running of the tournament into consideration.
(e) In dealing with improper conduct, the Tournament Director has a wide discretion. The appropriate remedy will vary from case to case. The Tournament Director should always act with intelligence and impartiality.

6.1.3 State of Mind

Disputes concerning conduct sometimes require the Tournament Director to form a belief about a player's state of mind. Many different factors may relevantly contribute to such beliefs. Subject to Rule 6.5 (Right of Appeal), the Tournament Director is the first and final arbiter of all such questions.

6.2 Level 1 Offences (Cheating and Abusive Behaviour)

6.2.1 Definition of Cheating

Any deliberate bad-faith violation of these Rules or the Standard Rules is an act of cheating. Cheating includes, but is not limited to:

(a) collusion;
(b) concealing or palming tiles;
(c) knowingly announcing or accepting incorrect move scores or cumulative scores;
(d) knowingly misreporting game results;
(e) using marked tiles;
(f) looking inside the bag;
(g) using accomplices, objects or materials to obtain an unfair advantage;
(h) when a player knowingly misrepresents his or her start/reply record when self-balancing starts are in use.

6.2.2 Suspected Cheating

(a) Players must avoid any personal action that might incur suspicion, and draw to the attention of their opponents any such action on their part.
(b) A player who believes that an act of cheating has occurred in his or her game should call the Tournament Director.
(c) A third party who witnesses an act of suspected cheating may not intervene directly, but must report the incident to the Tournament Director

6.2.3 Definition of Abusive Behaviour

Abusive behaviour includes, but is not limited to:

(a) making unauthorised physical contact with another player or a tournament official that intimidates, threatens or harms that person;

(b) making a statement that intimidates, threatens or insults another player or a tournament official;

(c) performing any other antisocial act that intimidates, threatens, insults or harms another player or a tournament official.

6.2.4 Penalties for Cheating and Abusive Behaviour

If, at the discretion of the Tournament Director, the degree of abuse is deemed to be minor then the penalty is a stern warning and the offender must be informed that any further abusive behaviour could lead to ejection from the tournament. Otherwise, if a player is found cheating or behaving abusively:

(a) the player will be ejected from the tournament and none of the player's games will count towards ratings. Furthermore:
    (i) if the tournament is a round robin, all of the player's games will be considered void; or
    (ii) if the tournament is not a round robin, all games already played by the player will be retrospectively awarded to the opponent with a margin of 150 points (if the opponent achieved a better result, no change will be made), and the player will be moved to the bottom of the standings and treated as a bye for all further games;

(b) the player's conduct will be reported to his or her national association; and

(c) WESPA may restrict the player's participation in future tournaments.

6.3 Level 2 Offences (Unethical Behaviour)

6.3.1 Definition of Unethical Behaviour

Any act which contravenes the spirit of equitable and fair play constitutes unethical behaviour, even if it cannot be classified as a violation of these Rules or the Standard Rules. Unethical behaviour includes, but is not limited to:

(a) impairing the opponent's view of the board;

(b) shuffling tiles persistently and noisily, or otherwise manipulating the board, bag or tiles to distract the opponent;

(c) making statements capable of misleading the opponent or affecting the opponent's play;

(d) talking unnecessarily (including loud computation of the score for a move);

(e) knowingly overdrawing, underdrawing, concealing an overdraw, or drawing out of order;

(f) deliberately flash-drawing;

(g) deliberately drawing slowly, or tracking tiles before drawing, to deny the opponent access to the tile bag;

(h) issuing frivolous challenges to gain thinking time, or calling 'hold' solely to prevent the opponent from drawing;

(i) misrepresenting the number of tiles in the bag;

(j) using devices or materials to gain an unfair advantage;

(k) knowingly misrepresenting the identity of a blank;

(l) checking scores solely in order to gain thinking time or disturb the opponent, or refusing to check scores when properly requested to do so;

(m) intermixing old tiles with tiles drawn in a courtesy draw;

(n) motioning to press the timer, but refraining, in an attempt to gauge the opponent's reaction to a turn;
(o) violating self-adjudication protocols;
(p) abusing game equipment;
(q) improperly leaving the playing area during play;
(r) not notifying an opponent of a timer malfunction.

6.3.2 Behaviour Not Considered Unethical

The following acts are not generally considered unethical:

(a) exploiting an opponent's failure to press the timer at the end of a turn;
(b) playing quickly to render the opponent short of time;
(c) failing to check the opponent's calculation of a score;
(d) the use of non-verbal body language to give a particular impression (for instance, playing a word confidently in order to dissuade the opponent from challenging);
(e) failing to challenge an invalid word for strategic reasons.

6.3.3 Penalties for Unethical Behaviour

(a) A Tournament Director who finds that a player has behaved unethically may choose to deliver an official or unofficial warning, or to impose another penalty.
(b) Possible penalties for unethical behaviour include, but are not limited to:
    (i) official warning;
    (ii) reduction of margin in tournament standings;
    (iii) loss of turn, loss of time or point penalty in the game in progress;
    (iv) forfeiture of a game;
    (v) ejection from the tournament.
(c) The Tournament Director may report unethical behaviour to the national association of the player concerned, or to WESPA.

6.3.4 Privacy of Score Sheets

It is the responsibility of individual players to ensure that private material recorded on their score sheets is adequately concealed.

6.4 Level 3 Offences (Poor Etiquette)

6.4.1 Definition of Poor Etiquette

Any failure to act with due courtesy and respect towards other players and tournament officials constitutes poor etiquette. Poor etiquette includes, but is not limited to:

(a) deliberately arriving late;
(b) rotating the board for the opponent at the completion of a turn;
(c) playing tiles upside down or with imperfect placement;
(d) placing the bag out of the opponent's reach;
(e) conducting lengthy or loud post-game analyses.

6.4.2 Penalties for Poor Etiquette

In general, poor etiquette attracts no penalty beyond an unofficial warning. However, a player aggrieved by poor etiquette may call the Tournament Director, who will assess the case on its merits.

6.4.3 Observational Etiquette (see also 4.8 Scope of Intervention by a Third Party in a Game)

(a) Persons observing a game must not:
  (i) distract the players;
  (ii) audibly discuss the game;
  (iii) do anything capable of passing information about the game to the players;
  (iv) infringe the players' personal space;
  (v) continue to observe, if asked to leave by a player or the Tournament Director.

(b) The Tournament Director has general discretion to ensure that observational etiquette is maintained, including the power to impose penalties.

(c) Annotators must fully understand and comply with observational etiquette. All other annotation arrangements, including the capacity of players to refuse annotation, are matters for the tournament organisers, the Tournament Director and the players concerned.

6.5 Right of Appeal

(a) As the Tournament Director's decision is final during a tournament (see 6.1.2 (a)), any appeal of a decision must commence after the completion of the tournament.

(b) Any parties impacted by a Tournament Director's decision during a tournament, or by a disciplinary decision for improper conduct, has right of appeal.

(c) Any appeal must follow the correct hierarchy of channels, namely:
  (i) the first appeal must be to the appropriate local or provincial body, if any;
  (ii) the second appeal must be to the appropriate national body, if any;
  (iii) only then may the player contact the WESPA Executive Committee, who will arrange for the finding to be reviewed as below.

(d) The WESPA Executive Committee will form a committee of disinterested players to consider the appeal, which will be determined either in person or through written submissions sent by email, fax or letter. Members of the WESPA Executive Committee are not allowed to be considered as "disinterested players". The committee so formed is the sole body that will review the correctness of the finding with the purpose of providing a recommendation to be considered by the WESPA Executive Committee.

(e) If the appeal is partly or wholly upheld, the committee will recommend a course of action to the WESPA Executive Committee. This may include the amendment of tournament results. The WESPA Executive Committee will then review the report with a view to either accept (in part or in whole) or reject the recommendation.

(f) There is no further right of appeal.

Appendix 1 – Standard Rules

Note: In the event of a discrepancy between the Standard Rules and the WESPA Game Rules, the WESPA Game Rules prevail. See Rule 1.1 (Standard Rules).

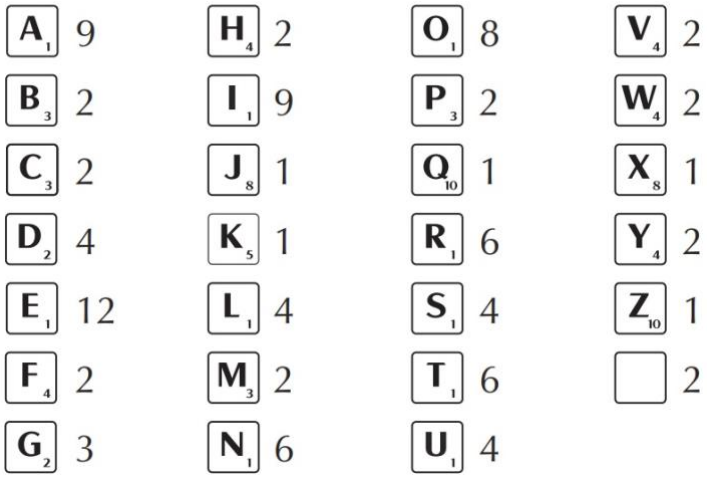
EVERY WORD COUNTS!

SCRABBLE® is a word game for 2, 3 or 4 players. Play consists of forming interlocking words, crossword fashion, on the SCRABBLE® playing board, using letter tiles with various score values. The object of the game is to get the highest score. Each player competes by using their tiles in combinations and locations that take best advantage of letter values and premium squares on the board. The combined total score for a game may range from about 400 points to 800 or more, depending on the skill of the players.

CONTENTS

1 Playing Board
100 Letter Tiles
4 Tile Racks
1 Tile Bag
100 Letter tiles:

- There are 98 tiles with letters of the alphabet and two blank tiles.
- Each of the letter tiles has score values indicated by the number to the bottom right of the letter.
- The two blank tiles have no score value, and can be used as any letter desired. When it is played, the player must state what letter it represents, after which it cannot be changed for the remainder of the game.



SET UP

- Get a pen and paper to keep score.
- Set up the board in the middle of the playing area.
- Each player takes a rack for arranging their tiles and places it in front of them.
- All the tiles are placed in the tile bag. Each player takes a tile out to find out who plays first. The player who has the tile nearest the beginning of the alphabet, with the blank preceding 'A,' plays first. The exposed tiles are put back into the bag and the bag is shaken to shuffle them.
- Each player, in turn, then draws seven new tiles and places them on their racks. Everyone is now ready to play SCRABBLE®. Play proceeds clockwise.

RULES OF PLAY

Keeping score

One player is elected as scorekeeper. They keep tally of each player's score after each turn.

Exchanging tiles

Any player may use their turn to replace any or all of the tiles in their rack. They may do so by discarding them face down, drawing the same number of new tiles, then mixing the discarded tiles with those remaining in the bag. They then await their next turn to play.

Passing (missing a turn)

Instead of placing tiles on the board, or exchanging tiles, a player may also decide to pass, whether or not they are able to make a word (or words).

However, should all players pass twice in succession, the game ends.

Placing the first word

The first player combines two or more of their tiles to form a word and places them on the board to read either across or down with one tile on the centre square (ribbon). Diagonal words are not permitted.

All tiles played in this and subsequent turns must be placed in one continuous line horizontally or vertically.

Permitted words

You may play any words listed in a standard English dictionary except those only spelt with an initial capital letter, abbreviations, prefixes and suffixes and words requiring apostrophes and hyphens. Foreign words in a standard English dictionary are considered to have been absorbed into the English language and are allowed. Prior to starting the game, all players must agree on a dictionary to be used.

Once a tile has been placed on the board, it may not be moved unless the word is successfully challenged.

Challenging words

Once a word has been played, the word may be challenged before the score is added up and the next player starts their turn. At this point only, you may consult a dictionary to check spelling or usage. If the word challenged is unacceptable, the player takes back their tiles and loses their turn.

BOARD Premium Spaces

The playing board consists of 15 x 15 squares in the playing area with grid lines to separate the squares. There are special premium squares on the board with bonus score values:

Premium Letter Squares

A light blue square doubles the score of a letter placed on it.
A dark blue square triples the score of a letter placed on it.

Premium Word Squares

A light red square doubles the score of the word.
A dark red square triples the score of the word.

If a word crosses both premium letter and word squares, all the bonus letter values are added up before the complete word score is double or tripled.

The bonus scores of the premium squares only apply to the turn in which the tiles are placed on them.

When a blank is placed on a Triple or Double Word square, the sum of the tiles in the word is doubled or tripled even though the blank itself has no score value. When it is placed on a Triple or Double Letter square, the value of the blank tile is still zero.

Scoring the first word
A player completes their turn by counting and announcing their score, which is recorded by the scorekeeper.

The score for the turn is calculated by adding up all the values of the numbers on the tiles, plus any premium values from utilising the premium squares.

Ending a turn
At the end of every turn, the player draws as many new tiles as they have played, thus always keeping seven tiles in their rack.

Added 50-point bonus
Any player who plays all seven of their tiles in a single turn scores a premium of 50 points in addition to their regular score for the turn. The 50 points are added on after doubling or tripling a word score.

Next Player's turn
The second player and then each player in turn, has the choice of exchanging tiles, passing or adding one or more tiles to those already played so as to form new words of two or more letters.

All tiles played in any one turn must be placed in one row only across or one column only down the board.

If they touch other tiles in adjacent rows, they must form complete words crossword fashion, with all such tiles.

The player gets full score for all words formed or modified by their play. Include the bonus scores of any premium squares on which they have placed the tiles.

There are five different ways that new words can be formed:

1. Adding one or more tiles to the beginning or end of a word already on the board, or to both the beginning and end of that word.
2. Placing a word at right angles to a word already on the board. The new word must use one of the letters of the word already on the board.
3. Placing a complete word parallel to a word already played so that adjoining tiles also form complete words. In this example, more than one word is formed in the same turn and each word is scored. The common letters are counted (with full premium value, when they are on premium squares) in the score for each word.
4. The new word may also add a letter to an existing word.
5. The last variation would be to "bridge" two or more letters. (This can only happen on the 4th move or later in the game.)

Sometimes a word may cross two premium word squares. The word score is doubled then re-doubled - 4 times the complete word score; or tripled and then re-tripled - 9 times the complete word score!

End of the game

The game ends when

- all the tiles have been drawn and one of the players has used all the tiles in their rack
- when all possible plays have been made
- all players have passed twice in consecutive turns

After all the scores are added up, each player's score is reduced by the sum of his unplayed tiles, and if one player has used all their tiles, their score is increased by the sum of the unplayed tiles of all the other players.

e.g. If Player one has an X and an A left on their rack at the end of the game, their score is reduced by 9 points. The player who used all their tiles adds 9 points to their score. Remember - the game can be won or lost on the last letter in the bag!

RULES CLARIFICATIONS

- If any tile touches another tile in adjacent rows, it must form part of a complete word crossword fashion, with all such tiles.
- The same word can be played more than once in a game.
- Pluralised words are allowed.
- A word can be extended on both ends within the same move e.g. TRAINER to STRAINERS.
- All tiles played in any one turn must be placed in one continuous line only, horizontally or vertically.
- Players may not add tiles to various words, or form new words in different parts of the board in the same turn.
- The bonus scores of the premium squares only apply to the turn in which the tiles are placed on them.
- When more than one word is formed in a single turn, each word is scored. The common letters are counted (with full premium value, when they are on premium squares) in the score for each word.
- If a word crosses two premium word squares, the word score is doubled and re-doubled - 4 times the complete word score; or tripled and re-tripled - 9 times the complete word score.
- When a blank is placed on a Triple or Double Word square, the sum of the tiles in the word is doubled or tripled even though the blank itself has no score value. When it is placed on a Triple or Double Letter square, the value of the blank tile is still zero.
- When one player has used all their tiles and the tile bag is empty, the game is over. In some games, no player succeeds in using all their tiles. In this case the game continues until all possible moves have been made. If a player is unable to move, they pass their turn. If all players pass twice, in consecutive turns, the game ends.
- A dictionary or word guide may not be used while a game is in progress to search for words to fit the tiles on your rack. It may only be consulted after a word has been played and challenged.

Appendix 2 – Official Word Source

From 1st July, 2019, the official word source is Collins Official SCRABBLE® Words, 5th edition, 2019.