Silesian
University
of Technology

FINAL PROJECT

Application for supporting dice games

Oluwamayokun Moyinoluwa SOFOWORA
Student identification number: 299531

Programme: Informatics
Specialisation: Informatics

SUPERVISOR
dr inż. Jacek Stój
DEPARTMENT OF DISTRIBUTED SYSTEMS AND INFORMATIC DEVICES
Faculty of Automatic Control, Electronics and Computer Science

Gliwice 2025

Thesis title

Application for supporting dice games

Abstract

Dice games have been a popular form of entertainment for many years, and the rise of mobile technology has increased the demand for digital versions of these games. This thesis focuses on developing an Android application that simulates classic dice games such as Pig, Greed, and Balut while incorporating computer vision techniques for real-time dice detection and recognition.

Utilizing modern software architecture principles and tools such as Jetpack Compose and Roboflow's cloud-based computer vision API for real-time dice detection, this application aims to provide an engaging and scalable gaming experience. Key features include real-time dice detection, a customizable game board, comprehensive player statistics tracking, and an adaptive AI opponent that learns from player behavior.

The project follows Kotlin's MVVM (Model-View-ViewModel) clean architecture principles, which enhance maintainability and scalability by separating concerns into distinct layers: UI, ViewModel, Manager, and Repository. This approach ensures a high-quality coding environment and effective data management practices.

Key words

Kotlin, Android development, Computer Vision, Jetpack Compose

Tytuł pracy

Aplikacja do wspomagania gry w kości

Streszczenie

Gry w kości są popularną formą rozrywki od wielu lat, a rozwój technologii mobilnych zwiększył popyt na cyfrowe wersje tych gier. Niniejsza praca skupia się na opracowaniu aplikacji na Androida, która symuluje klasyczne gry w kości, takie jak Pig, Greed i Balut, jednocześnie włączając techniki widzenia komputerowego do wykrywania i rozpoznawania kości w czasie rzeczywistym.

Wykorzystując nowoczesne zasady architektury oprogramowania i narzędzia, takie jak Jetpack Compose i oparte na chmurze API widzenia komputerowego Roboflow do wykrywania kości w czasie rzeczywistym, ta aplikacja ma na celu zapewnienie angażującego i skalowalnego doświadczenia w grach. Kluczowe funkcje obejmują wykrywanie kości w czasie rzeczywistym, konfigurowalną planszę do gry, kompleksowe śledzenie statystyk gracza i adaptacyjnego przeciwnika AI, który uczy się na podstawie zachowania gracza.

Projekt jest zgodny z zasadami czystej architektury MVVM (Model-View-ViewModel) Kotlina, które zwiększają łatwość utrzymania i skalowalność poprzez rozdzielenie problemów na odrębne warstwy: UI, ViewModel, Manager i Repository. Takie podejście zapewnia wysokiej jakości środowisko kodowania i skuteczne praktyki zarządzania danymi.

Słowa kluczowe

Kotlin, rozwój Androida, Wizja komputerowa, Kompozycja Jetpack

Contents

Chapter 1

Introduction

In recent years, mobile gaming has experienced exponential growth, yet many games fail to deliver the sophistication needed to sustain player interest. Traditional dice games, with their straightforward mechanics and strategic depth, offer a unique opportunity for innovation. However, current mobile dice games often fall short due to predictable AI behavior and limited analytics [1]. These limitations result in gameplay that feels static, with easily exploitable strategies and little feedback to help players improve [2]. As a result, players struggle to stay engaged, often facing steep learning curves without meaningful insights into their progress.

1.1 Objective of the Thesis

This thesis aims to overcome these challenges by developing an advanced mobile dice game application that reimagines classic gameplay with modern technology. The application will feature an adaptive AI opponent capable of analyzing player behavior to deliver a dynamic and evolving challenge. This approach ensures that gameplay remains engaging and rewarding, catering to players of all skill levels.

To enhance the gaming experience further, the project incorporates virtual screen technology to simulate real-world dice interactions, bringing the tactile excitement of traditional dice games into the digital space. Additionally, a comprehensive analytics system will track detailed performance metrics, offering players personalized insights and fostering long-term engagement.

By combining these innovations, this thesis seeks to redefine mobile dice gaming. Through adaptive AI, virtual interactions, and meaningful analytics, the project bridges the gap between classic dice games and cutting-edge technology, setting a new benchmark for the genre.

1.2 Overview of Thesis Chapters

This thesis is organized into seven chapters. Chapter 1 introduces the mobile gaming landscape and dice games, along with the motivation behind this project. Chapter 2 analyses the problem, exploring the mechanics of dice games, the role of AI in gaming, and mobile app architecture. Chapter 3 outlines the system requirements, architecture design, data model, user interface, and AI model design. Chapter 4 details the external specifications, including the development of the Android application, integration with TensorFlow Lite, and the implementation of player analytics and game mechanics. Chapter 5 focuses on the internal specifications of the project, covering unit testing, integration testing, and user acceptance testing, as well as performance analysis and evaluation of the AI model. Chapter 6 addresses verification and validation to ensure the system meets its requirements and functions correctly. Finally, Chapter 7 concludes the thesis by summarizing its key elements, highlighting its achievements, limitations, and suggestions for future improvements.

Chapter 2

Problem Analysis

Creating an engaging dice game application that utilizes computer vision presents unique challenges that require careful consideration. This chapter examines the key aspects of developing an adaptive AI capable of playing various dice game variants while ensuring reliable real-time detection. Understanding these challenges is essential for accurate dice detection in different conditions, identifying the best computer vision model architectures, and designing an AI opponent that captivates players.

2.1 Computer Vision

Computer vision plays a key role in enabling the smooth integration of physical dice into digital gameplay. This section discusses the challenges and solutions involved in dice detection and recognition.

2.1.1 Challenges in Dice Recognition

The implementation of accurate dice recognition presents several technical challenges:

- **Lighting Variations**: Dice faces appear differently under various lighting conditions. This includes shadows that can hide the patterns of pips, or dots on the dice, reflective surfaces causing glare, and significant differences between indoor and outdoor lighting that affect contrast and visibility.
- **Perspective and Orientation**: The system must handle dice captured at different angles, which affects how pips appear in the image. Multiple dice can overlap or occlude each other, and the distance between the camera and dice impacts pip visibility and overall recognition accuracy.
- **Background Complexity**: Various playing surfaces can affect detection reliability. Similar patterns in the background may trigger false positives, while moving

backgrounds, such as when playing on unstable surfaces, can further complicate the detection process.

- **Real-time Processing Requirements**: The system must process frames quickly for a responsive user experience. This involves careful management of battery consumption and memory usage, requiring optimized processing algorithms and efficient resource management.

2.1.2 Image Preprocessing

The system employs a sophisticated preprocessing pipeline that enhances image quality for more accurate recognition:

```
1   private suspend fun preprocessImage(bitmap: Bitmap): Bitmap {
2       return withContext(Dispatchers.Default) {
3           try {
4               // Step 1: Convert to RGB if needed
5               val rgbBitmap = ensureRGBFormat(bitmap)
6
7               // Step 2: Enhance contrast and normalize lighting
8               val enhancedBitmap = enhanceContrast(rgbBitmap)
9
10              // Step 3: Scale while maintaining aspect ratio
11              val scaledBitmap = scaleWithAspectRatio(enhancedBitmap, TARGET_SIZE)
12
13              // Step 4: Apply noise reduction
14              val finalBitmap = reduceNoise(scaledBitmap)
15
16              Timber.d("Preprocessing completed successfully")
17              finalBitmap
18          } catch (e: Exception) {
19              Timber.e(e, "Error during image preprocessing")
20              // Fallback to basic scaling if enhancement fails
21              Bitmap.createScaledBitmap(bitmap, TARGET_SIZE, TARGET_SIZE, true)
22          }
23      }
24  }
```

Listing 2.1: Image Preprocessing Pipeline

2.1.3 Model Architecture

The dice recognition system leverages a pre-trained object detection model from Roboflow [3]. The model processes images at 640x640 resolution and was trained on a custom dataset of 250 images, supporting six distinct classes representing dice faces 1-6. Developed and hosted on Roboflow's platform, it provides efficient object detection capabilities through their API service.

The model is then accessed through Roboflow's Hosted Inference API, with preprocessing handling:

- RGB format conversion
- Image scaling to the required 640x640 dimensions
- Confidence threshold (set at 0.4 for reliable detections)

For each detected die, the model outputs bounding box coordinates, confidence scores, and class labels, which the application processes to update the game state.

2.1.4 Detection Pipeline

The detection pipeline evolved throughout development, starting with a basic implementation and later expanding to include more sophisticated preprocessing and validation.

The initial detection process followed a straightforward approach:

```
suspend fun detectDice(bitmap: Bitmap): List<Detection> {
    return withContext(Dispatchers.Default) {
        try {
            // Preprocess the image
            val processed = preprocessImage(bitmap)
            // Run inference
            val detections = roboflowRepository.detectDice(processed)
            // Post-process results
            filterAndValidateDetections(detections)
        } catch (e: Exception) {
            Timber.e(e, "Error detecting dice")
            emptyList()
        }
    }
}
```

Listing 2.2: Initial Dice Detection Pipeline

The pipeline was later enhanced to improve detection reliability through:

- **Advanced Preprocessing**: Implementation of RGB format conversion, contrast enhancement, and adaptive scaling while maintaining aspect ratios.
- **Robust Validation**: Addition of comprehensive detection validation including aspect ratio checks, minimum size requirements, and position validation.
- **Quality Filters**: Implementation of confidence threshold and noise reduction techniques.

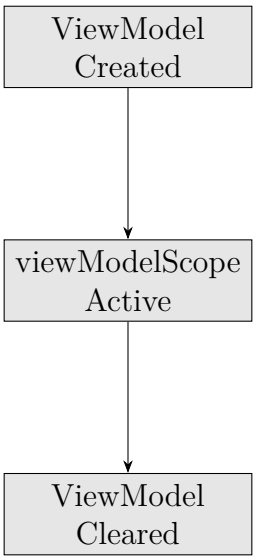
This enhanced pipeline significantly improved detection accuracy and reliability across various lighting conditions and capture scenarios.

2.2 Asynchronous Processing

Kotlin Coroutines are used to handle asynchronous updates efficiently, ensuring the app remains responsive. This section explores the benefits of coroutines and their role in managing background tasks.

2.2.1 Coroutine Scope

The `viewModelScope` is tied to the lifecycle of the ViewModel. This ensures that coroutines are automatically canceled when the ViewModel is cleared, preventing memory leaks and unnecessary processing.



2.2.2 Launching Coroutines

The `launch` function starts a new coroutine, allowing non-blocking execution. This is crucial for processing tasks like dice value detection or data loading, which can be time-consuming.

```
viewModelScope.launch {
    // Load vibration setting
    dataStoreManager.getVibrationEnabled()
        .collect { enabled -> _vibrationEnabled.value = enabled }
}
```

Listing 2.3: Launching a Coroutine

2.2.3 Suspend Functions

Suspend functions are a key feature of Kotlin's coroutine system, marking functions that can be paused and resumed. These functions can only be called from within a coroutine or another suspend function, ensuring proper asynchronous execution.

```
fun rollDice() {
    if (isRolling.value || !isRollAllowed.value) return
    viewModelScope.launch {
        trackDecision()
        trackRoll()
        _isLoading.value = true
        val results = diceManager.rollDiceForBoard(_selectedBoard.value)
        if (_vibrationEnabled.value) provideHapticFeedback()
        // Process game state after rolling
        val newState = processGameState(results)
        _gameState.value = newState
    }
}
```

Listing 2.4: Suspend Function Example

In this example, the `rollDice` function is designed to manage the dice-rolling process within the game. It is executed within a coroutine scope using `viewModelScope.launch`, which allows it to perform asynchronous operations without blocking the main thread. This ensures that the UI remains responsive while the dice are being rolled.

The function begins by checking if a roll is already in progress or if rolling is not allowed, returning early if either condition is true. This prevents unnecessary operations and ensures that the game logic is executed only when appropriate.

Within the coroutine, the function tracks the player's decision and roll actions, providing valuable data for game analytics. It then sets a loading state to indicate that a roll is in progress. The actual dice rolling is performed by the `diceManager`, which returns the roll results.

If vibration feedback is enabled, the function provides haptic feedback to enhance the user experience. Finally, the function processes the game state based on the roll results and updates the game state accordingly.

The use of coroutines in this function allows for efficient management of asynchronous tasks, ensuring that the game logic is executed smoothly and without interruption. This design pattern is essential for maintaining a responsive and engaging user interface in a coroutine-based architecture.

2.2.4 Error Handling

The `try-catch` block within the coroutine handles exceptions, ensuring errors are logged and managed gracefully. This prevents crashes and maintains application stability.

```
viewModelScope.launch {
    try {
        val detections = roboflowRepository.detectDice(bitmap)
        _detectionState.value = if (detections.isNotEmpty()) {
            DetectionState.Success(detections)
        } else {
            DetectionState.NoDetections
```

```
8           }
9       } catch (e: Exception) {
10          Timber.e(e, "Error detecting dice")
11          _detectionState.value = DetectionState.Error(e.message ?: "Unknown error")
12      }
13  }
```

Listing 2.5: Error Handling in Coroutines

The coroutine runs in the background, ensuring the main thread remains responsive to user interactions. Listing **??** illustrates the implementation of error handling within a coroutine. It processes collected data to update a LiveData property, enabling the UI to dynamically reflect any changes.

2.2.5 Interface Responsiveness

In interactive applications, maintaining a responsive user interface (UI) is critical to delivering a seamless user experience. By offloading computationally intensive tasks, such as AI decision-making and data processing, to background threads, the main UI thread remains available for handling real-time user interactions. This approach minimizes UI lag, ensuring that animations, gestures, and updates occur smoothly without delays.

For instance, in the context of dice games, background tasks such as calculating potential AI strategies or updating game states are delegated to coroutine-based background threads in Kotlin. This concurrency model enables a separation of concerns, where the UI layer focuses solely on rendering and responding to user input while backend logic operates asynchronously. The result is a user experience that feels intuitive and highly responsive, even under computationally demanding scenarios.

2.3 Gameplay

Making a game with lively gameplay mechanics presents many challenges, particularly when integrating adaptive AI and ensuring a balance between challenge and accessibility. This section delves into the solutions employed to address these challenges.

2.3.1 Scoring Algorithm

The scoring algorithm calculates scores based on the game's rules and considers various scoring categories and player actions. For example, in a game like Pig, the score for a single turn can be calculated as:

$$\text{Turn Score} = \sum_{i=1}^{n} x_i \tag{2.1}$$

where n is the number of dice rolls in the turn, and x_i represents the value of each dice roll. The scoring algorithm plays a crucial role in determining the outcome of the game

and ensuring fair and consistent scoring across different game variants and player actions.

2.3.2 Adaptive AI

The AI system uses strategic decision-making to adapt its behavior dynamically, enhancing engagement by providing a challenging opponent.

```
private fun handleAITurn(
    diceResults: List<Int>, currentState: BalutScoreState
): BalutScoreState {
    gameTracker.trackDecision()
    if (currentState.rollsLeft <= 0) {
        // AI chooses a category
        val category = chooseAICategory(diceResults, currentState)
        gameTracker.trackBanking(ScoreCalculator.calculateCategoryScore(diceResults,
    category))
        return scoreCategory(currentState, diceResults, category)
    }

    // AI decides which dice to hold
    gameTracker.trackRoll()
    val diceToHold = decideAIDiceHolds(diceResults)

    return currentState.copy(
        rollsLeft = currentState.rollsLeft - 1,
        heldDice = diceToHold
    )
}
```

Listing 2.6: handleAITurn Function

The decision-making process is augmented by a game tracker, which records AI decisions for further analysis. These insights allow developers to fine-tune the AI, ensuring it provides both a challenging and fair opponent.

2.3.3 Game Mechanics

The game mechanics are crucial for delivering an engaging and intuitive gameplay experience. They are designed to ensure clear rules and interactions for both players and the AI, enabling a seamless game flow. An important component of the implementation is the `handleTurn` method, as shown in Listing -9. This method differentiates between player and AI turns and manages key actions such as dice holding and roll counting. Its modular design supports clear separation of player and AI logic into distinct methods, reducing complexity and improving maintainability. This structure makes it easy to add new game modes or refine AI behavior without disrupting existing functionality.

During its turn, the AI uses a blend of predefined rules and probabilistic decision-making to evaluate the game state and select the optimal strategy, providing a dynamic and challenging opponent. Meanwhile, the game mechanics prioritize user experience by

offering clear visual and interactive cues, ensuring that players can focus on strategy without being hindered by the interface.

```
fun handleTurn(
        currentState: GameScoreState.PigScoreState,
        diceResult: Int? = null
    ): GameScoreState.PigScoreState =
        when (currentState.currentPlayerIndex) {
            AI_PLAYER_ID.hashCode() -> handleAITurn(currentState, diceResult)
            else -> handlePlayerTurn(currentState, diceResult)
        }
```

Listing 2.7: handleTurn Function

In the `handleTurn` method:

- **AI Turn Handling:** If the current player's index matches the AI player's identifier, the method delegates the turn to `handleAITurn`, which implements the AI's decision-making logic. This includes evaluating the game state, deciding which dice to hold, and determining whether to bank a score or re-roll.
- **Player Turn Handling:** If the turn belongs to a human player, the method invokes `handlePlayerTurn`, which processes the player's actions, such as selecting dice to hold and performing a roll.

By isolating player-specific and AI-specific logic into separate methods, the design enhances code readability and maintainability. This modular approach ensures that updates or adjustments to AI strategies or player interactions can be made independently, maintaining the overall flow of the game.

This structured design allows for a compelling gaming experience by promoting a dynamic AI challenge while ensuring that interactions remain clear and responsive. The thoughtful integration of adaptive AI, clear gameplay mechanics, and user-focused design ensures that the game is accessible to players of all skill levels. Additionally, the modularity of the architecture enables the seamless incorporation of advanced features, such as multiplayer modes or new game variants, without disrupting the core mechanics.

Through these technical and strategic design choices, the project delivers an engaging dice game that is robust and scalable for future enhancements.

2.4 Broader Implications

This project goes beyond addressing technical challenges by bridging the gap between physical and digital gaming. It enhances the educational value of dice games while promoting critical thinking, decision-making skills, and overall player engagement. By leveraging augmented reality and machine learning, it introduces innovative gaming experiences that make traditional gameplay more interactive and dynamic.

2.4.1 Educational Value and Accessibility

The integration of AI and computer vision enhances the educational aspects of dice games by fostering strategic thinking and problem-solving. With adaptive AI providing real-time feedback and dice detection enhancing gameplay, players are offered a platform to develop their skills in a fun and engaging way.

2.5 Existing Solutions

In the realm of dice detection and recognition, several solutions have been developed to address the challenges of accurately identifying dice values in various contexts. This section provides a brief overview of some notable solutions.

2.5.1 D3-Deep-Dice-Detector

The D3-Deep-Dice-Detector is a robust solution that leverages deep learning techniques to detect and recognize dice values from images. It utilizes convolutional neural networks (CNNs) to process image data and accurately predict dice outcomes. This approach is particularly effective in environments with varying lighting conditions and dice orientations. The project is available on GitHub and provides pre-trained models and scripts for training on custom datasets [4].

2.5.2 Roboflow Dice Detection

Roboflow [5] offers a comprehensive platform for building and deploying computer vision models, including dice detection. Their solution allows users to train custom models using their datasets, providing flexibility and adaptability to specific use cases. The platform supports various model architectures and offers tools for data augmentation and model evaluation. Roboflow's community and documentation provide valuable resources for developers looking to implement dice detection.

2.5.3 Dice Detection with YOLOv5

The YOLOv5 Dice Detection [6] repository by *ultralytics* showcases the use of the YOLOv5 object detection model for various object recognition tasks, including dice recognition. This solution involves training a YOLOv5 model on a dataset of dice images to achieve high accuracy in detecting and classifying dice values. The repository provides scripts for data preparation, model training, and inference.

Chapter 3

Requirements and tools

The project requirements outline the system's expected functions and performance, detailing the features and capabilities that meet user needs. This forms the foundation for designing and implementing the system's core functionalities.

3.1 Functional Requirements

Functional requirements define the specific behavior and functions of the system. For the dice game application, these include:

1. **AI Opponent**: The system should feature an AI opponent that dynamically adapts to player behavior and skill level.
2. **Game Variants**: The application should support a variety of dice games, providing diverse gameplay options.
3. **Player Analytics**: The system should track and analyze player statistics to inform AI decision-making.
4. **User Interface**: The application should provide an intuitive and consistent user interface for all game variants.
5. **Real-Time Feedback**: Users should receive performance feedback, to understand and improve their gameplay strategies.
6. **Image Recognition**: The system should accurately detect and recognize physical dice through the device camera:
    - Detect dice faces in real-time using computer vision
    - Process multiple dice simultaneously
    - Provide accurate pip counting and value recognition

3.2 Non-functional Requirements

Non-functional requirements are essential for ensuring the quality and performance of the system. They help address issues such as latency, scalability, usability, reliability, and security. For the application, the following non-functional requirements are identified:

1. **Performance**: The application should deliver a smooth user experience on mobile devices, with minimal latency in AI decision-making.

2. **Scalability**: The system should be able to handle an increasing number of users and game variants.

3. **Usability**: The user interface should be easy to navigate and accessible to diverse users.

4. **Reliability**: The application should consistently provide accurate AI behavior and player analytics, maintaining user satisfaction.

3.3 Use Case Modelling

Use cases describe how users interact with the system, illustrated using UML diagrams that visualize user interactions and the application's features.

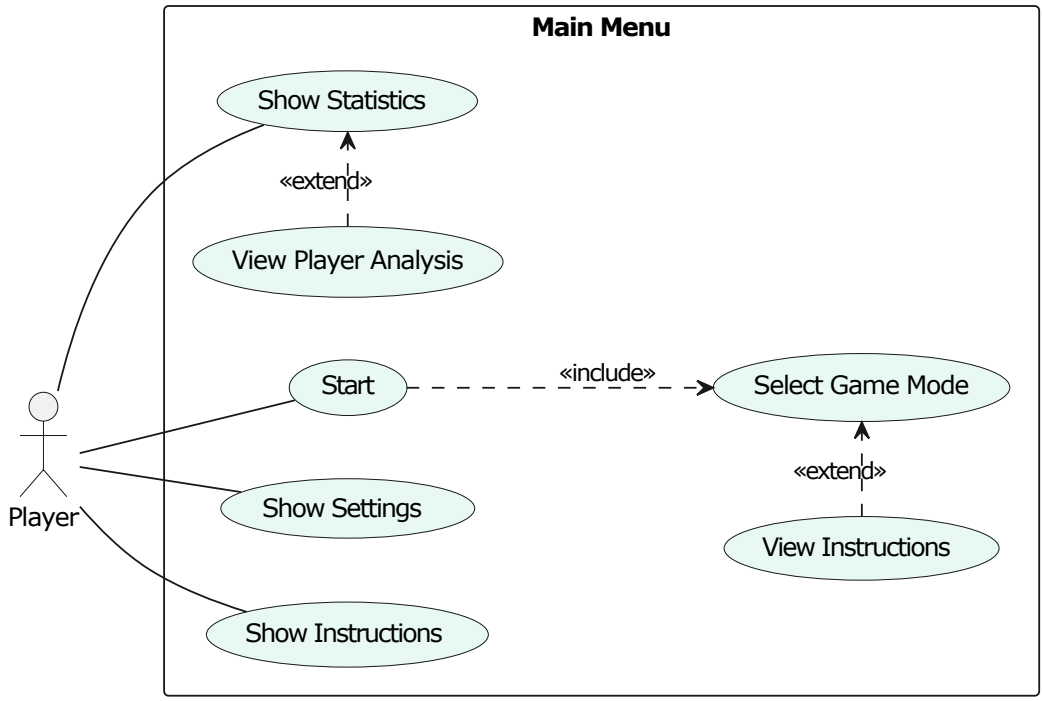


Figure 3.1: Use case for the game's main menu.

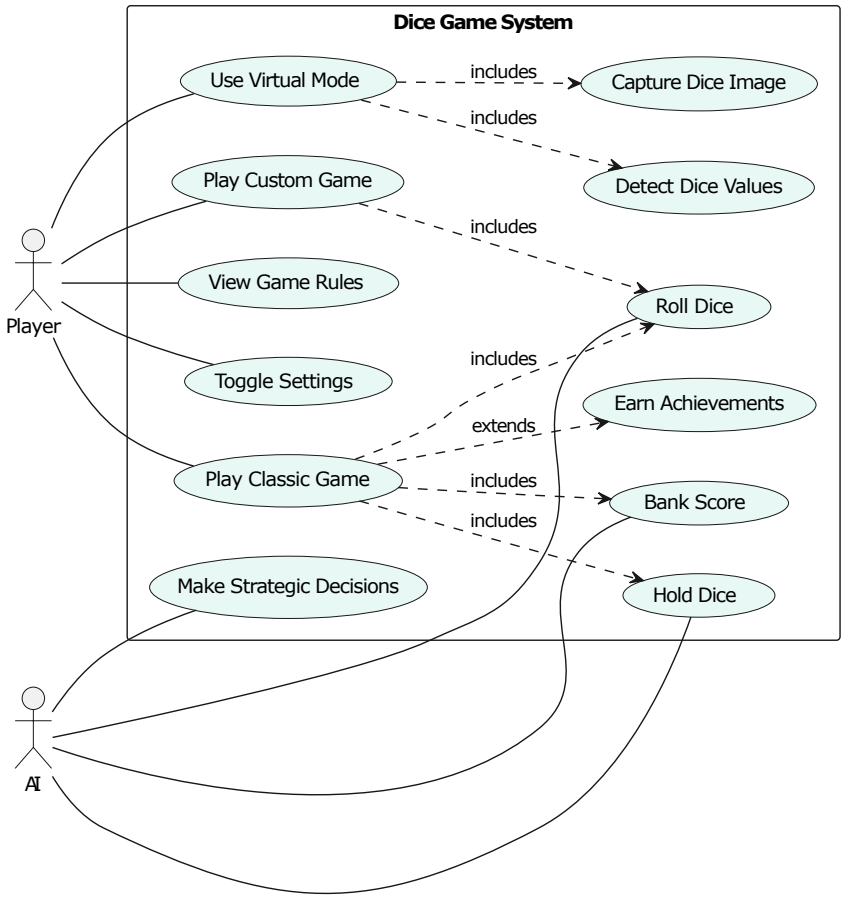


Figure 3.2: Use case for the game's core gameplay.

Main Menu Use Case

Upon launching the application, users encounter the main menu, the central hub for interaction. This interface offers access to classic boards with various game variants, allowing users to explore different gameplay dynamics. Users can also view player statistics to track performance and progress. Additionally, the menu enables configuration of game settings for a personalized experience and features virtual dice detection for innovative gameplay. Comprehensive game rules and instructions are readily available for guidance. Figure 3.1 illustrates these interactions and their relationships.

Game Use Case

The game system encapsulates core gameplay mechanics, allowing players to capture dice rolls via image recognition or roll dice on the classic board. Players make strategic decisions, such as holding or banking points, while competing against AI opponents, which adds a challenging dynamic. Key features include virtual mode for dice detection, custom games with unique rules, and classic games that involve rolling dice and earning achievements. Figure 3.2 illustrates these interactions and the roles of players and AI.

3.4 Description of Tools and Technologies

3.4.1 Technologies and Tools

Kotlin

Kotlin is a modern programming language that offers features like null safety, extension functions, and interoperability with Java, making it a preferred choice for Android development. Kotlin is used throughout to write the entire application code, leveraging its concise syntax and powerful features to enhance productivity and maintainability [7].

Roboflow

Roboflow is a computer vision platform that offers tools for dataset management, model training, and deployment [5]. In this project, Roboflow was instrumental in training and hosting the dice detection model using a custom dataset [3]. The platform's capabilities in dataset preparation and augmentation, model training and optimization, and API integration for mobile deployment facilitated the development of a robust computer vision system. By leveraging Roboflow's efficient API, the project achieved real-time inference, enhancing the application's performance and reliability.

Figma

Figma is a collaborative interface design tool that was used to create the application's initial designs and prototypes [8].

Git

Git is a distributed version control system that allows developers to track changes in their code-base through GitHub, collaborate with others, and manage project history efficiently. This project uses GitHub to manage source code versions, enabling collaborative development and ensuring code integrity [9].

Android Studio

Android Studio is the official integrated development environment for Android development, providing tools for building, testing, and debugging Android applications. The project is developed using Android Studio IDE, which offers a comprehensive suite of tools for efficient Android app development [10].

Jetpack Compose

Jetpack Compose is a modern toolkit for building native Android UI, offering a declarative approach that simplifies UI development and enhances code readability. Jetpack

Compose offers modular re-composition, allowing UI elements to update independently, reducing rendering times by up to 30% and CPU usage by up to 25% compared to XML-based layouts [11]. This project utilizes Jetpack Compose to design and implement the user interface, allowing for a more intuitive and flexible UI design process [12].

3.4.2 Libraries

Dagger-Hilt

Dagger-Hilt is a dependency injection library for Android that simplifies the setup and management of dependencies in Android applications. The project employs Dagger-Hilt to manage dependencies, improving code modularity and testability [13].

Lottie Animation

Lottie is a library for rendering animations in real-time, allowing developers to use animations created in Adobe After Effects in their applications. The project uses Lottie to incorporate smooth and visually appealing animations, improving the user experience [14].

Vico Charts

Vico Charts is a library for creating interactive and customizable charts in Android applications, providing a variety of chart types and features. This project uses Vico Charts to display data visually, making it easier for users to understand and interact with the information [15].

Timber

Timber is a logging library for Android that provides a simple and flexible API for logging messages, making it easier to manage log output in Android applications. The project uses Timber for logging, which aids in debugging and monitoring the application's behavior [16].

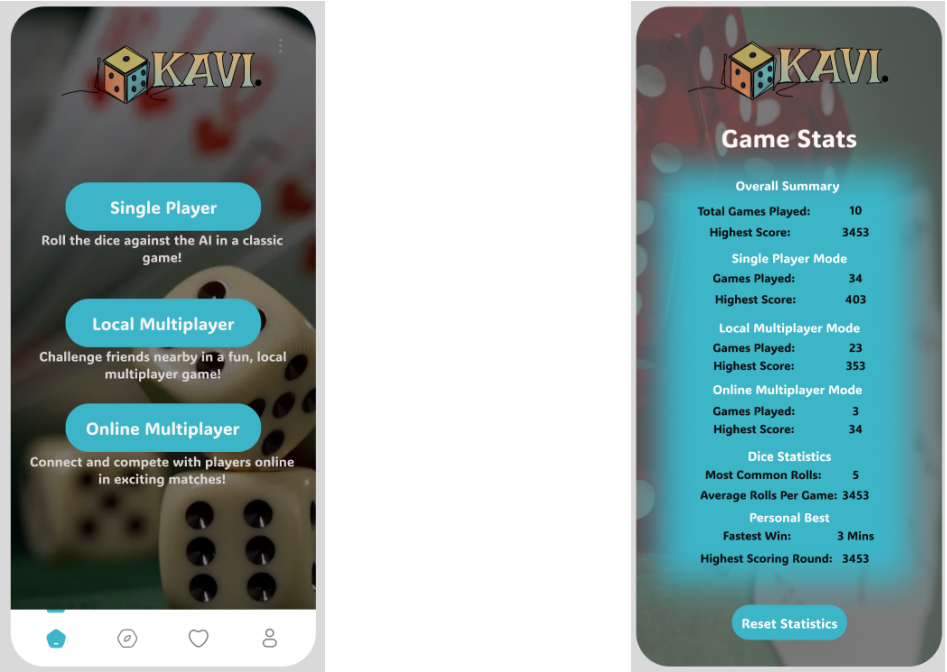
JUnit 5

JUnit 5 is composed of several modules, including JUnit Jupiter which is a combination of the programming and extension model for writing tests and extensions in JUnit 5 [17].

3.5 Methodology of Design and Implementation

The design and implementation of the dice game application follow an iterative and incremental development methodology. This approach involves:

1. **Requirement**: Identifying and documenting functional and non-functional requirements.
2. **Design**: Creating architectural and component designs, including UML diagrams to visualize system interactions.
3. **Implementation**: Developing the application in iterative cycles, focusing on one feature or component at a time.
4. **Testing**: Conducting unit, integration, and user acceptance testing to ensure the system meets requirements using JUnit.
5. **Documentation**: Documentation of the implemented features and future development possibilities.



(a) Design Page 1 (b) Design page 2

Figure 3.3: Initial UI designs and prototypes created in Figma

3.5.1 Design Process

The application's design process began with creating detailed wireframes and prototypes in Figma. The designs underwent several iterations based on user feedback and technical constraints, evolving into the final implementation. Figure 3.3 shows some of the initial design concepts and their evolution [18].

Various existing solutions and design tools inspired the design of the application, one of which stood out was the board screen design was inspired by a dice application project

by binaryshrey [19]. This repository provided a minimalistic and intuitive approach to dice roll applications, which influenced the layout and functionality of the board screen in this project.

3.5.2 Model Training

The dice detection model was developed using Roboflow's platform, which streamlined the entire process from dataset creation to deployment. The training dataset, depicted in Figure 3.4, consisted of carefully annotated dice images across various conditions, ensuring robust detection performance in real-world scenarios.

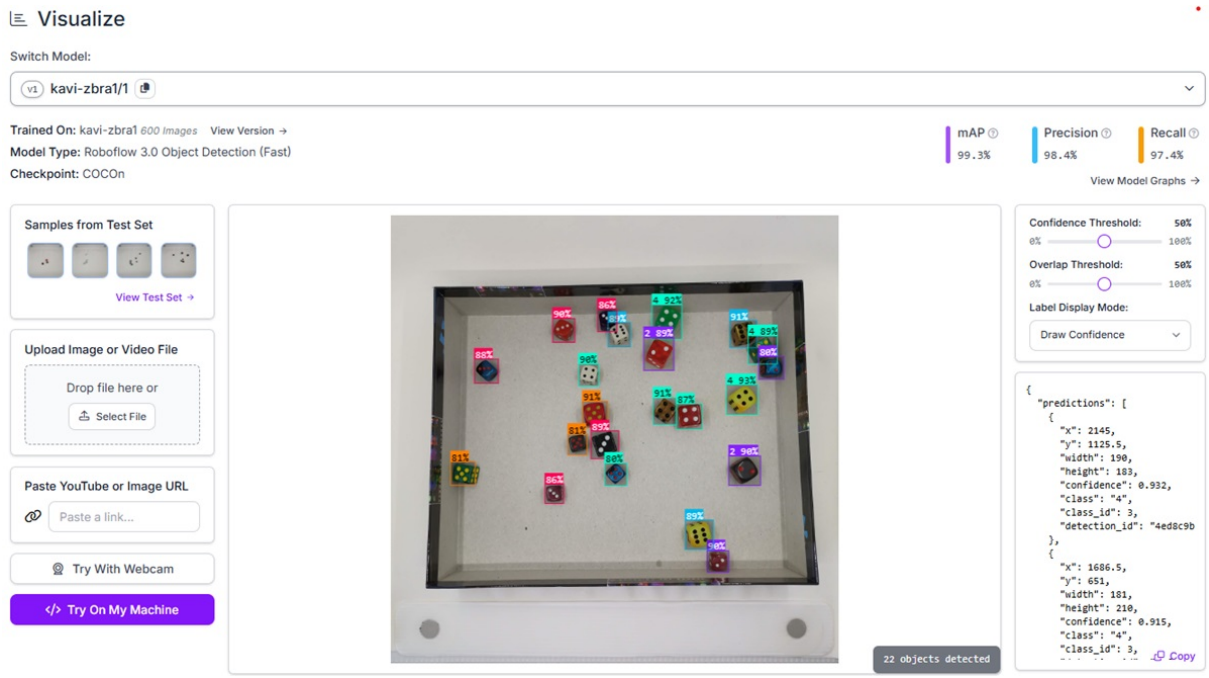


Figure 3.4: Roboflow dataset management interface showing dice image annotations

Roboflow facilitated data augmentation and preprocessing, which enhanced the dataset's diversity. The model training and optimization phases were crucial for achieving high accuracy, while the deployment and API integration ensured seamless real-time inference capabilities.

3.5.3 Project Timeline

The project was implemented from November 2024 to January 2025, following a structured timeline as shown in Figure 3.5. The development process was organized into major phases, including planning, design, core development, AI integration, and testing, with regular milestones to track progress.

This structured approach allowed for continuous improvement and adaptation to changing requirements, ensuring a high-quality application. While some initially planned fea-

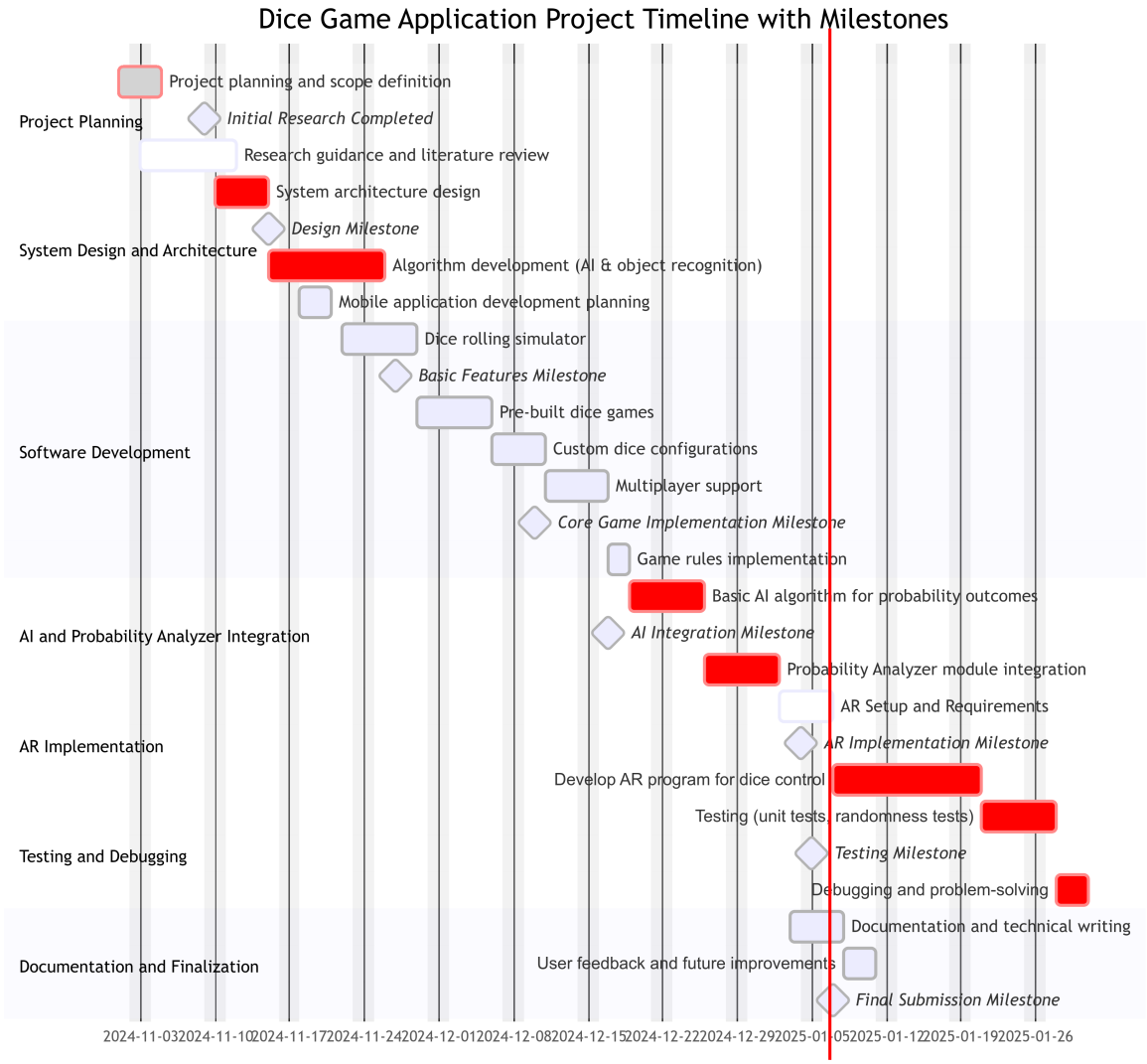


Figure 3.5: Project Gantt chart showing development phases and milestones

tures like AR implementation were identified as future enhancements, the focus remained on delivering a robust core game experience with AI capabilities.

Chapter 4

External Specification

This chapter provides a detailed overview of the external specifications for the application. It outlines the specific requirements for its operation and the installation procedures necessary for a straightforward setup. Additionally, it includes key information to improve user understanding and ensure the application fulfils its intended purpose effectively.

4.1 Hardware and Software Requirements

4.1.1 Hardware Requirements

- Android smartphone with Android 11.0 (API level 30) or higher.
- Camera with support for CameraX.
- Minimum 2GB RAM for standard use of the application, or 3GB RAM for optimal performance on devices with lower memory and processing power.
- Internet connection for the dice recognition module; the other parts of the app do not require internet and will work otherwise.
- At least 100MB of free storage space.
- Processor: ARM-based processor supporting Neon instruction set.

4.1.2 Software Requirements

To build the application, the following software is required:

- Android Studio Giraffe (2023.1.1) or later.
- Kotlin 2.0.21
- Gradle 8.10.2 and the corresponding Android Gradle Plugin (e.g. 8.1.2).
- CameraX library version: 1.4.0.

4.2 Installation Procedure

For installing the application, the following procedure is required:

4.2.1 APK Download

1. Download the Release APK from the project's repository at GitHub Releases Page.
2. Enable installation from unknown sources in the device's settings. Note that installing apps from unknown sources has security implications; please proceed with caution.
3. Open the APK file and follow the on-screen instructions.

4.2.2 Building with Android Studio

To install the application for debugging purposes, follow these procedures:

1. **Clone the Repository**: Start by cloning the project repository from GitHub.

```
git clone https://github.com/Mayokun-Sofowora/kavi.git
```

2. **Open in Android Studio**: Launch Android Studio and open the cloned project.
3. **Sync Gradle Dependencies**: Allow Android Studio to sync the Gradle dependencies automatically.
4. **Run on Device/Emulator**: Connect an Android device or start an emulator with a minimum SDK of 30, then run the application.

In addition to the instructions provided, be sure that:

- A debug build variant is selected in Android Studio.
- USB debugging is enabled in the device's settings.
- You connect an Android device via USB or use an emulator.
- You select the appropriate run configuration in Android Studio.

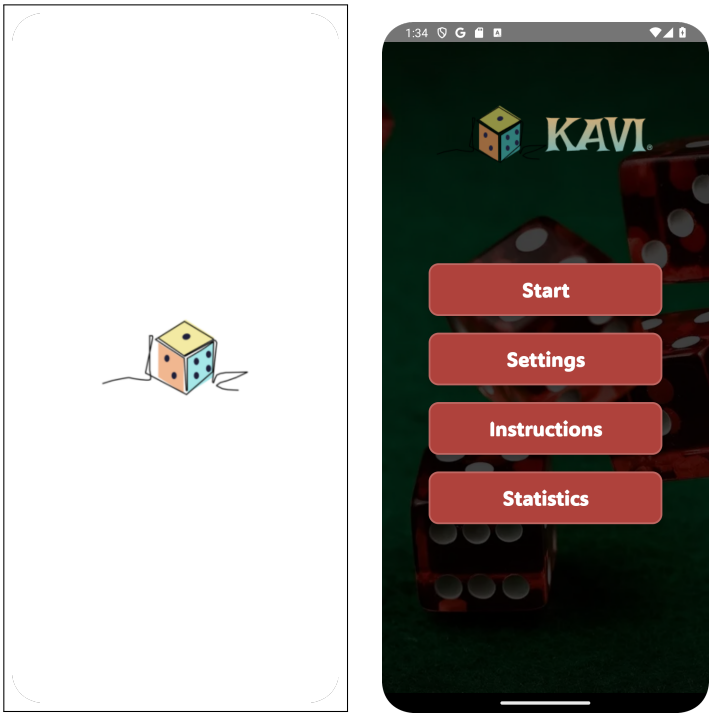
4.3 Types of Users

- **Regular Users**: Play dice games, use image recognition features, and adjust settings.
- **Developers/Testers**: Access debugging logs, experimental features, and perform system administration tasks.

4.4 User Manual

4.4.1 Navigating the App

When starting the game, users are welcomed with a splash screen that displays the application's logo in figure 4.1a. After a short moment, the main menu is shown in figure 4.1b. The main menu provides navigation options to access the different features of the application.



(a) The Splash Screen (b) The Main Menu

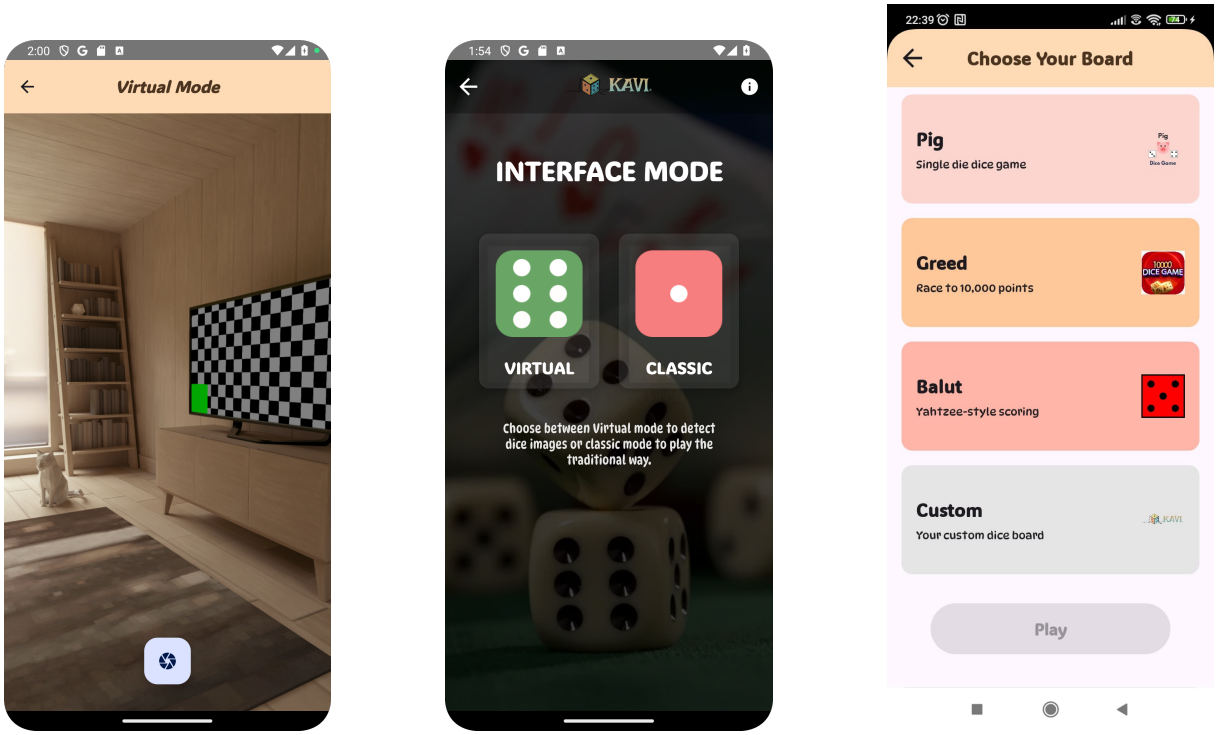
Figure 4.1: Screens displayed when starting the game.

4.4.2 Game Interface

The interface allows users to navigate to different sections of the application, such as the classic boards to play the classic dice games or the virtual screen to use the image recognition feature. The figure 4.2 shows the navigation options available.

4.4.3 Classic Boards

The application offers a diverse set of game boards, each designed for a distinct dice game experience. These games range from simple "press-your-luck" scenarios to more strategic challenges that require planning and risk management. The primary games offered are *Pig*, *Greed*, and *Balut*. Additionally, the application includes a custom board where



(a) Virtual Mode (b) The Start Screen (c) Classic Mode

Figure 4.2: The Games main interfaces.

users can define their own game rules and scoring mechanisms, and also select the number of dice used in the game. The figure 4.2c shows the classic boards available in the application.

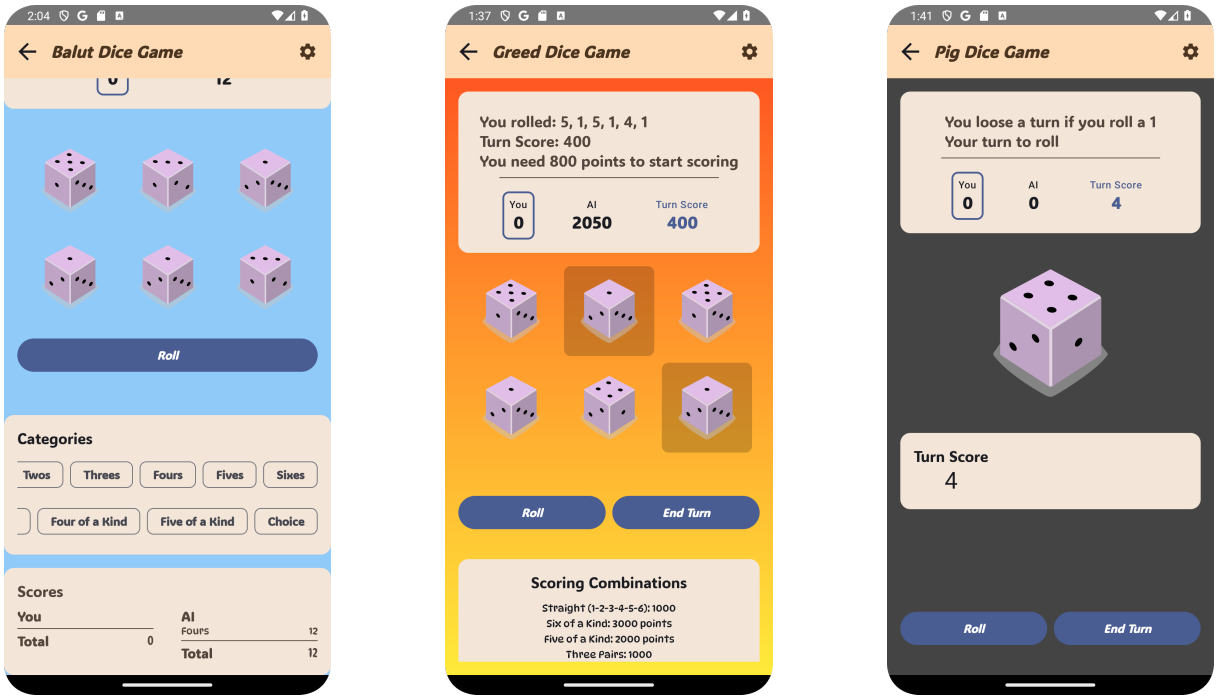
4.4.4 Game Objectives

Pig: The Risk of the Roll

Pig is a simple, engaging game of chance and risk. The goal is to be the first player to reach a total of 100 points. During each turn, players roll a single die, accumulating points with each roll. The key aspect of Pig is the ability to "bank" the points you have accumulated in that turn, however, the risk is that if you roll a 1, you lose all the points accumulated during that turn. The challenge lies in choosing when to press your luck for more points and when to play it safe to avoid losing those points.

Greed: Navigating Scoring Combinations

Greed is a more complex game that rewards strategic decision-making and risk-taking. Each turn begins with the roll of six dice. After the roll, players get to select which dice they want to keep to accumulate points, based on scoring combinations like straights, sets (e.g., three-of-a-kind, four-of-a-kind), and single 1s and 5s. These combinations vary in their scoring values, meaning that a key aspect of the game is knowing which combinations



(a) Balut Game Board (b) Greed Game Board (c) Pig Game Board

Figure 4.3: Game Boards in the Application

you should aim for. Unlike Pig, in Greed, players must accumulate at least 800 points in a turn to start banking them. The winner is the first player to reach a total of 10,000 points.

Balut: Strategic Category Management

Balut is a game of strategy, similar to Yahtzee. Players have up to three rolls per turn using five dice. The core of Balut is strategically filling the scoring categories, such as sets, straights, full houses, and more. Each category can be used only once, so planning and smart selection of which dice to hold is critical. After all categories are filled, the player with the highest total score wins.

4.4.5 Game Controls: Interacting with the Game

The application provides a user-friendly interface with intuitive controls, allowing players to easily interact with each game and manage their scores. This section talks about the various controls and how they function across different game modes.

Basic Game Controls (Roll, End Turn)

The primary way players interact with the games is through the roll button. In games like *Pig* and *Greed*, this button (shown in Figure 4.4) also serves as an "End Turn" button.

In these games, tapping the button rolls the dice and also ends the turn after a score is obtained.

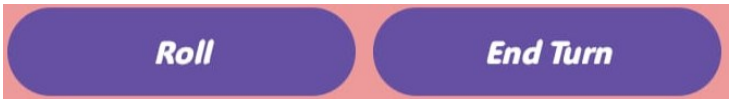


Figure 4.4: Roll and End Turn Button

Player Management and Custom Game Setup

In the custom game board, it is also possible to add players and edit their names (as shown in Figure 4.5). This function can be accessed by tapping a button which allows the player to add and remove player, as well as edit their names. This allows users to set up a new game of their liking.

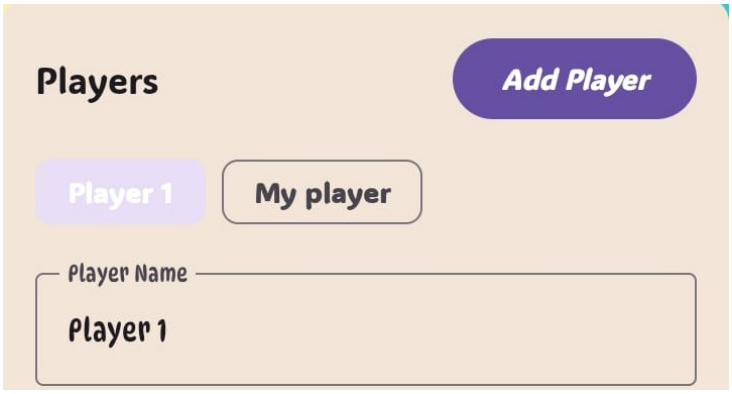


Figure 4.5: Player Management and Editing

Balut-Specific Controls (Category Selection)

Balut introduces the unique feature of category selection. After rolling the dice, a player can select a category in which to score, shown in Figure 4.6. This allows for a more strategic game, where each category can only be used once.

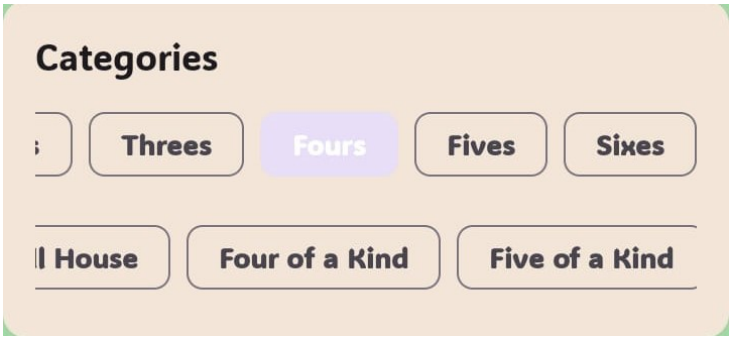


Figure 4.6: Balut Category Selection

Holding Dice

In both *Greed* and *Balut*, players can strategically select dice to hold for the next roll. As seen in Figure 4.7, this is done by tapping on the individual dice on the screen. The selected dice will be saved, and can be rolled again in the subsequent roll.

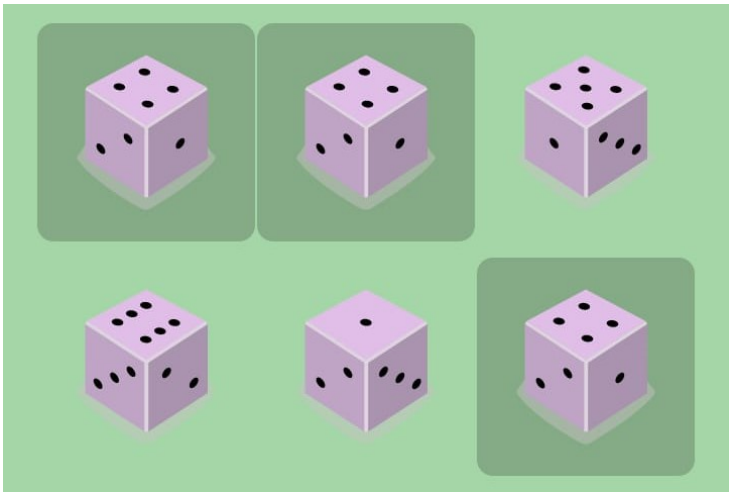


Figure 4.7: Selecting Dice to Hold

Balut Score Function

After a category has been selected in *Balut*, the game also provides an additional button which can be tapped to calculate the score and end the turn. As seen in figure 4.8.

Figure 4.8: Balut Roll and Score Button

Custom Board Settings

The custom board allows players to set the number of dice that will be used for that specific game. In addition, the board name can also be set, as seen in Figure 4.9. This level of customization gives the users control over how the games are played.

Score Modifiers and Reset

Figure 4.10 shows the score modifiers, which enable users to manually adjust their scores. This feature can also be used to keep track of score in other types of games that might not be covered by this application, or in the custom mode. The feature also contains a reset score button that will reset the score of the game to 0. This can be used to easily restart any game or any other custom use. Additionally, this interface also contains the

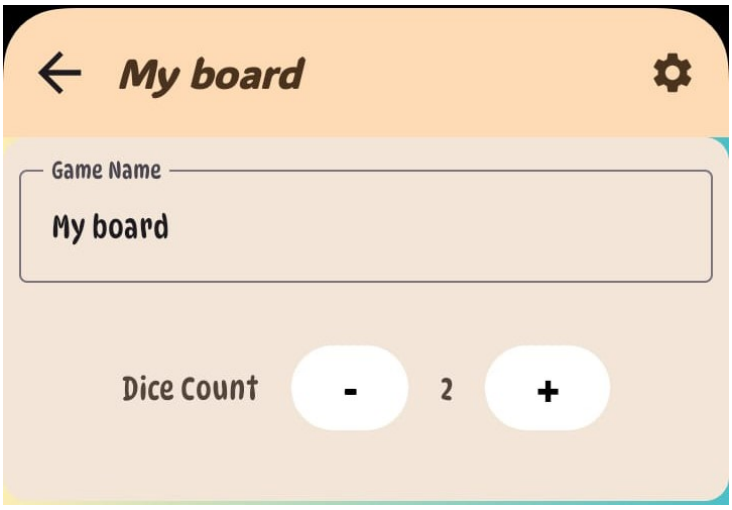


Figure 4.9: Custom Board Settings

functionality of adding a note that will be saved as part of the game, which can be used to keep track of important information of the current game.

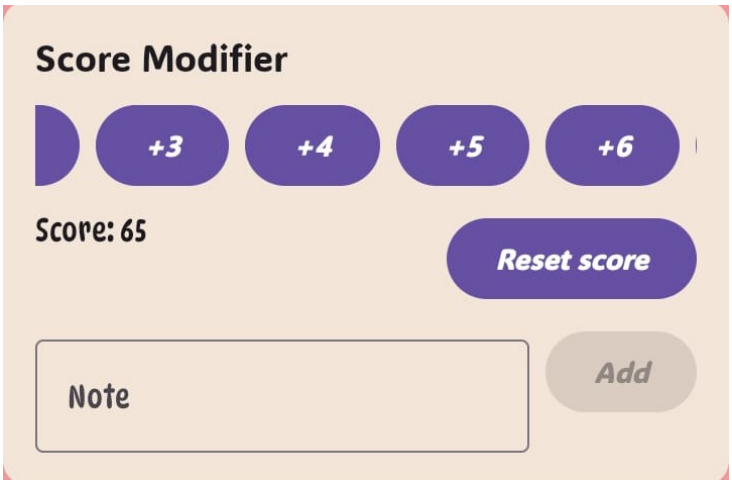


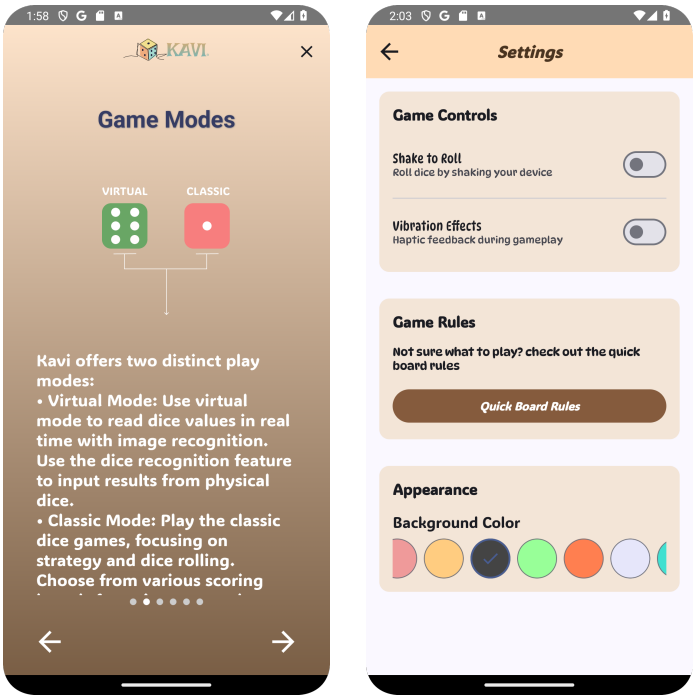
Figure 4.10: Score Modifiers and Reset

4.4.6 Instruction

The instructions screen provides users with detailed guidance on how to play the game. It includes rules, tips, and strategies to enhance the gaming experience. The instructions screen is illustrated in Figure 4.11a.

4.4.7 Settings

The settings screen, on the other hand, allows users to customize their gaming experience, such as enabling vibration, the shake-to-roll function, and board color customization. The settings screen is shown in Figure 4.11b.



(a) Instructions Screen (b) Settings Screen

Figure 4.11: Instructions and Settings.

4.4.8 Statistics

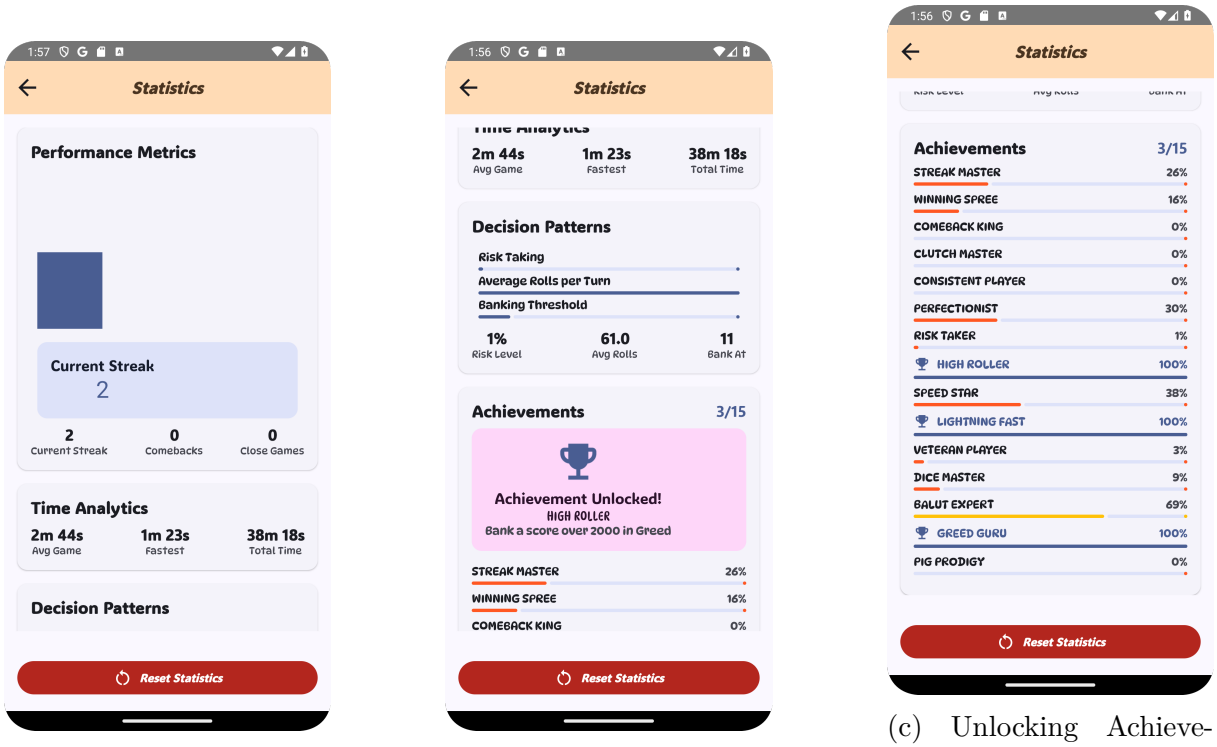
The statistics screen offers users comprehensive insights into their gameplay performance. It displays a variety of data, including win records, average scores, and other pertinent metrics. The figure 4.12 illustrates the statistics screen, where users can view their achievements, such as "High Roller," "Lightning Fast," and "Greed Guru." Additionally, users can assess their risk-taking tendencies, current winning streaks, comebacks, and close games. The screen also features time analytics, allowing users to track their fastest games, total playtime, and time spent on individual games, providing a detailed overview of their gaming habits.

4.5 System Administration

The administration of the application involves several key tasks to ensure its reliability, performance, and security. These responsibilities are primarily carried out by the developers of the application, and can range from day to day maintenance to handling the occasional unforeseen issues.

4.5.1 Application Maintenance

Regular maintenance of the application is essential to ensure it remains functional and up to date. This involves several tasks, including:



(a) Performance Metrics (b) Decision Patterns (c) Unlocking Achievements

Figure 4.12: Statistics Screen

- **Release Management:** New releases of the application are periodically deployed via GitHub as APK releases, which often require careful planning and testing to ensure compatibility and maintain a consistent user experience. This includes creating new release branches, building the APK, and updating the app with new changes, new features, or bug fixes.

- **Monitoring and Troubleshooting:** Developers are also responsible for monitoring the performance of the application. This may involve checking for crashes, logging error reports, identifying bugs or unexpected behavior, and making changes to solve the issues.

4.5.2 Data Management

Careful management of the application data is critical to ensure its proper functioning. The developers are responsible for *DataStore Management*, which involves taking periodic backups of application settings and user preferences stored with Android DataStore. The data is also cleaned periodically to ensure that the device is not using unnecessary storage. This is essential to maintain consistency and avoid potential data corruption or data loss.

4.6 Security Issues

4.6.1 Introduction

This section provides a detailed view on the various security issues that the application is vulnerable to, based on potential threats and exploitable areas in the application.

4.6.2 Data Handling

Application data, including DataStore backups, might be accessed through Android's backup services if a user's Google account is compromised. Specifically, an attacker might be able to get access to the application if a user's Google account has been *phished*, or if there has been a *data breach*, or if the account has been compromised in some other way.

A compromised Google account could allow an attacker to access backups stored in Google Drive, potentially revealing all user information stored in the application.

The application uses Android's KeyStore to encrypt the backup data, however, this is not a full solution and does not prevent unauthorized access [20].

4.6.3 Communication and API

Even when using HTTPS, attackers can intercept data during transmission to the RoboFlow API. An attacker on the same network as the user can set up a malicious server and redirect all API requests to it, thereby gaining access to the API traffic. The application enforces HTTPS for all API requests, but it does not use certificate pinning, which means it is vulnerable to *man in the middle* attacks. This is something that should be considered for future versions of the application. A sophisticated attacker could potentially bypass the HTTPS certificate and redirect the traffic through a malicious server. [21]

Even with rate-limiting, attackers can bypass limitations and create a denial of service attack, potentially making the app or the external API unusable. A bot could flood the application with API requests. The application uses rate-limiting using a 'CoroutineScope' and a 'MutableStateFlow' to limit how many times an API is called in a given time, this does reduce the risk of the application crashing, but it is not foolproof. It also has input validation to prevent *API abuse*, but these are not foolproof solutions and might be bypassed by sophisticated attackers.

4.6.4 Code Security

Even when using code obfuscation, attackers can still reverse engineer the code to steal private information and private API keys, or understand the game logic. Code Obfuscation

is the process of making the code more difficult to read and understand, by changing the variable names, and classes to make them unreadable.

An attacker can decompile the application to find API keys that are used to contact external resources, or understand the game logic to make a bot to cheat at the game.

The application uses code obfuscation through R8's obfuscation process, to hide the code logic, which is not a full solution, and can be bypassed by sophisticated attackers [22].

4.7 Security Considerations

The application implements several security measures to protect user data and ensure system integrity. These measures are based on Android security best practices and are designed to comply with relevant data protection requirements.

4.7.1 Data Protection

- **Encrypted Local Storage:** User settings and preferences, saved using DataStore, are stored using Android's EncryptedSharedPreferences, which provides encryption at rest using Android's KeyStore.

- **Privacy-Preserving Image Processing:** Images captured by the user are only processed within the app's scope and are not stored or transmitted outside of the device unless explicitly requested by the user through a sharing action. No personally identifiable information is extracted and saved from the image.

4.7.2 System Security

- **Runtime Permission Management:** The application requests only the necessary permissions at runtime (e.g., camera access) and respects the user's choice to grant or deny them. The application uses Android's Permission API to handle the permissions.

- **Secure Communication Channels:** The communication with the RoboFlow API for image recognition is done over HTTPS, which uses TLS/SSL encryption to ensure confidentiality and integrity of the API requests.

- **Data Access Controls:** Only the application can access the DataStore data, and only the specific components of the application that require it can access its underlying data. This reduces the risk of potential data exposure to malicious application or rogue modules of this application.

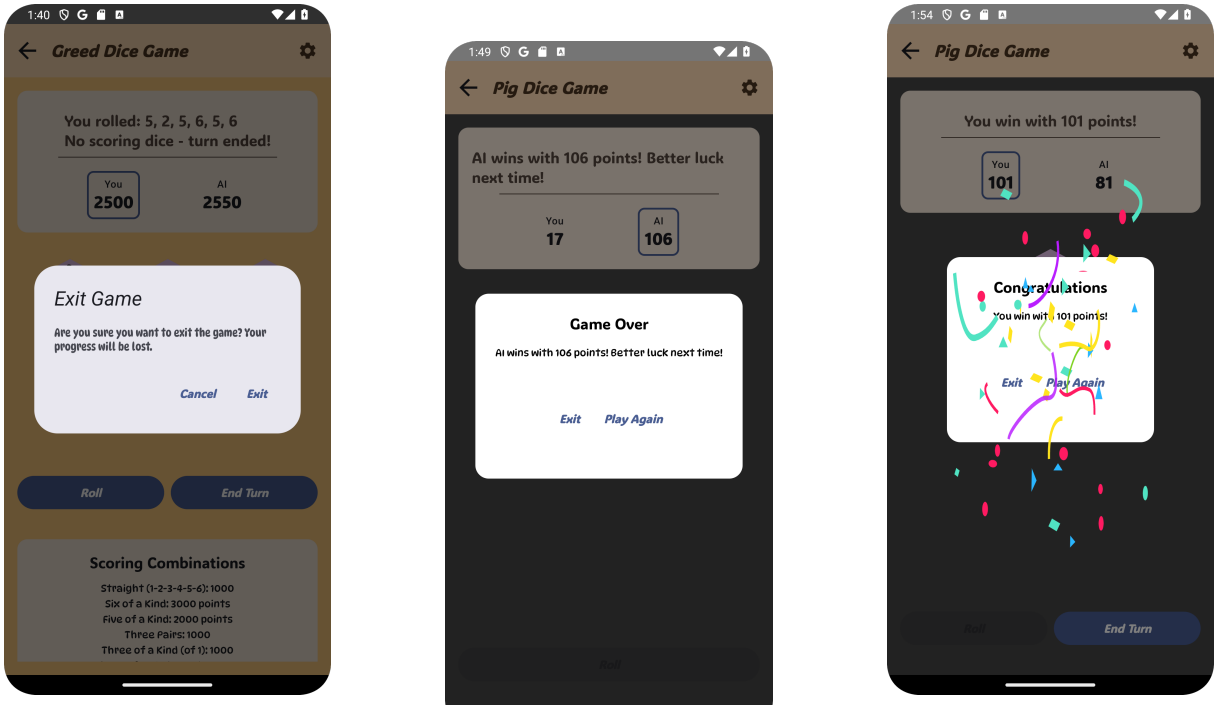
4.7.3 Future Enhancements

The following security enhancements are considered for future development:

- **User Authentication and Authorization:** Implementing a secure user authentication mechanism (e.g., using Firebase Authentication) and role-based authorization to enhance data protection and prevent unauthorized access to sensitive features, such as statistics or training data.
- **Improved API Security:** Implementing API rate-limiting to prevent abuse and implementing stricter input validation for the RoboFlow API.
- **Regular Security Audits:** Implementing regular penetration testing to evaluate the security and performance of the system.

4.8 Working scenarios

In addition to the scenarios described in the user manual 4.4, this section provides a few practical examples of application usage.

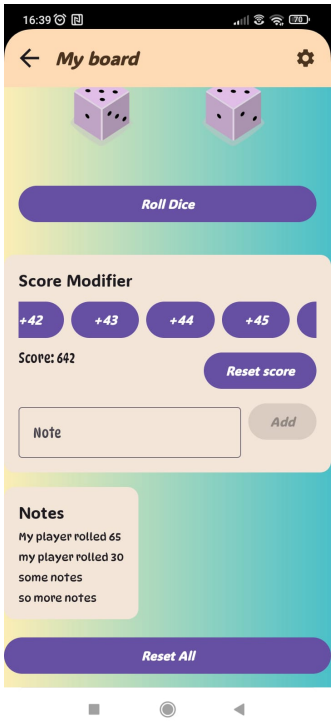


(a) Exiting a game of Greed

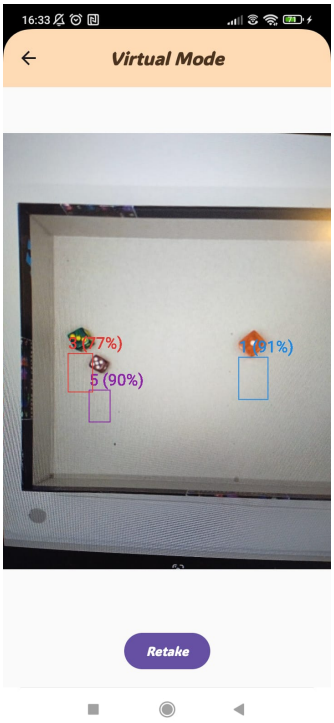
(b) Loosing a Game of Pig

(c) Winning a Game of Pig

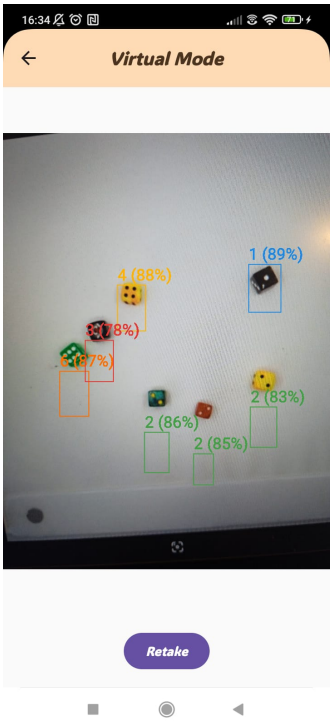
Figure 4.13: Board Features in the Application



(a) Custom Game Board



(b) Image Recognition Example



(c) Image Recognition Example 2

Figure 4.14: Additional Features and Interactions

Chapter 5

Internal Specification

This chapter provides a detailed overview of the application's internal workings. It covers the system's concept, architecture, data structures, components, algorithms, design patterns, and relevant UML diagrams.

5.1 System Architecture

The project's architecture follows the modern Model-View-ViewModel (MVVM) pattern, adhering to Clean Architecture principles. This separation of concerns allows for better maintainability and testability of the code.

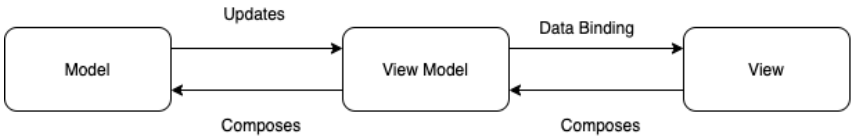


Figure 5.1: High Level MVVM Architecture.

As illustrated in Figure 5.1, the application is structured into several layers:

- **Application Layer**: Contains the main application logic and navigation.
- **UI Layer**: Responsible for the user interface components, including screens and themes. This layer corresponds to the "View" component of the MVVM pattern.
- **ViewModel Layer**: Manages the data and business logic, providing a bridge between the UI and the Model. This layer corresponds to the "ViewModel" component of the MVVM pattern.
- **Manager Layer**: Handles specific game logic and state management. This layer sits below the view model, and contains the logic for manipulating the models.

- **Repository Layer**: Manages data access and interactions with external sources. This layer is the gatekeeper for the model layer, accessing and manipulating the models before passing it to the application layer.

- **Model Layer**: Represents the data and domain logic of the application. This layer contains the data entities and business rules. This layer corresponds to the "Model" component of the MVVM pattern.

5.2 Data Structures and Data Management

The application utilizes a variety of data structures to effectively manage game state, user profiles, and game data. These structures are designed for efficiency and scalability, supporting the diverse features of the game. Data persistence is achieved using DataStore, a modern data storage solution, and dependency injection is managed using Dagger Hilt.

5.2.1 Data Models

The application's data is represented using models located in the 'models' directory. These models encompass different aspects of the game, including game state, user data, and results from the image detection mechanisms.

Game Models

The core game models are:

- **GameStatistics**: Tracks overall game performance metrics for each user across all games. These metrics include total games played, win rates, average scores, and other relevant data.

- **GameScoreState**: Holds the current score and state of the game during an individual game session, including the score for each player and the game's current turn.

- **DecisionPatterns**: Captures the different decision-making patterns of a player in each game, including when the player decides to hold dice, or when they decide to roll or bank scores. This is used to calculate statistics about how players are playing the game.

- **WinRate**: Calculates and stores the win rate of a player. This will be used in *PlayerAnalysis*.

- **TimeMetrics**: Stores metrics about how long a player played the game, including the amount of time a player has spent playing each game, and the time spent per round. This is used in *PlayerAnalysis*.
- **PlayerAnalysis**: Provides detailed analysis of a player's behavior and performance trends, combining data from various sources like 'WinRate', 'TimeMetrics', and 'DecisionPatterns'. It tracks trends over time, identify strong and weak areas of a player, and provides an overall evaluation of a player's performance.

Detection Models

The detection models, located in the 'detection' subdirectory of 'models', are:

- **DiceDetectionResult**: Captures results from dice detection processes, including the number detected and the number on each die.
- **Detection**: Represents the state of the detection model.
- **DetectionRequest**: Represents the request to start the detection.
- **DetectionResponse**: Represents the response from the detection.
- **ImageInfo**: Holds the information of the image used for detection.
- **Prediction**: Represents a single prediction from the detection model.

5.2.2 Data Management

The application uses a modern and efficient approach to data management, leveraging DataStore for persistent storage and Dagger Hilt for dependency injection.

Persistent Storage

DataStore provides a robust, asynchronous solution for managing the application's persistent data. Unlike traditional databases, DataStore offers type safety and a reactive approach, ensuring smooth and efficient data handling for user preferences, game statistics, and other relevant application data. This approach helps the application to provide quick access to stored data and avoids some of the limitations of other persistent storage options.

Dependency Injection

Dagger Hilt simplifies the management of dependencies by providing a standardized way to inject components into the application. This improves code modularity, making the app easier to test, maintain, and scale. Dagger Hilt helps manage the dependencies

between classes and is used to ensure all modules are configured correctly, and this creates a more efficient way to manage the dependencies of the application, while also making testing easier.

5.3 Components, Modules, and Classes

This section outlines the core components, modules, and classes that form the foundation of the application. It provides a summary of essential classes, detailing their roles and responsibilities.

5.3.1 Application Classes

The main classes of the application can be categorized into Main Application Classes, ViewModels, and Managers.

Main Application Classes

- **KaviApplication**: The main class that extends Android's Application class, responsible for initializing the application, and important libraries like Timber for debugging.
- **MainActivity**: The main entry point for the application, sets up the primary UI, and manages navigation.

ViewModels

- **AppViewModel**: Manages application-wide data and state, coordinating between different parts of the application.
- **GameViewModel**: Manages data and logic specific to each game, providing a bridge between the view and the data and state.
- **DetectionViewModel**: Manages the state and logic of the image detection process, and exposes this data to the UI layer.

Managers

- **MyGameManager**: Handles the state management and the logic of the custom game board.
- **PigGameManager**: Manages the game state, rules, and scoring for the Pig game.
- **GreedGameManager**: Manages the game state, rules, and scoring for the Greed game.

- **BalutGameManager**: Manages the game state, rules, and scoring for the Balut game.
- **DiceManager**: Manages the various aspects of rolling the dice, and its state.
- **DataStoreManager**: Manages the saving and retrieval of data from the DataStore.
- **StatisticsManager**: Manages the collection and processing of game statistics.
- **SettingsManager**: Manages the saving and loading of the application's settings.
- **ShakeDetectorManager**: Manages the logic of the shake gesture.

A detailed Class diagram is included in the UML diagram provided in section 5.6.

5.4 Algorithms and Implementations

The application implements several sophisticated algorithms to provide an engaging and intelligent gaming experience.

5.4.1 Image Processing Pipeline

The dice recognition system uses a sequence of image processing steps implemented in 'RoboflowRepositoryImpl', a class responsible for handling communication with the Roboflow API:

Preprocessing

- RGB conversion using 'ensureRGBFormat()' to guarantee that the image is in the correct format for processing.
- Contrast enhancement through 'enhanceContrast()' using histogram-based normalization to make the dice pips more clear.
- Aspect ratio scaling via 'scaleWithAspectRatio()' to make sure that the images are of the correct size.
- Noise reduction with 'reduceNoise()' to reduce the noise in the image.

Detection

- API integration with Roboflow service using the 'RoboflowClient', which makes a call to the Roboflow API and gets the result.

- Confidence filtering (threshold: 40%) which removes detections with less than 40% confidence to avoid erroneous detections.
- Non-maximum suppression, performed by using a library for detection, which removes overlapping bounding boxes, by selecting the bounding boxes with the highest score and removing those that are overlapping.

5.4.2 AI Strategy System

The AI decision-making system, which is responsible for the decision-making process of the AI, is implemented across multiple manager classes. The game strategy, where the managers will choose the optimal next move for the AI, is done with the following methods:

- *'shouldAIBank()'* decides when to bank points for the AI, by calculating the odds of loosing points, versus obtaining more points.
- *'chooseAICategory()'* selects optimal scoring categories for the Balut AI, based on the dice that are currently being held.

5.4.3 Statistics System

The statistics tracking system, which is used for analyzing and processing user data, is centralized in 'StatisticsManager':

Game Analytics

- *'updateGameStatistics()'* records game outcomes for the players, and saves it to the 'GameStatistics' model.
- *'updateTimeMetrics()'* tracks timing data, including the time spent in each round, and the total time spent in a game.
- *'updatePerformanceMetrics()'* calculates improvement rates, by keeping track of how many games the player wins, and their high scores.

Achievement Processing

The achievement processing is responsible for monitoring and managing user progress toward unlocking achievements in the application. The following key functions are implemented:

- *'calculateAchievements()'*: Evaluates the unlock conditions for all achievements available in the application and updates their status based on the latest user activity and metrics.

- *'updateProgressMetrics()'*: Tracks progress toward achievement goals by calculating how close the user is to meeting the requirements for unlocking various achievements.

5.5 Applied Design Patterns

This application uses several design patterns to enhance code organization and maintainability:

- **Observer Pattern**: Used in the ViewModel layer with Kotlin Flow (StateFlow) to manage and notify the UI of data changes, ensuring a reactive user interface. This allows the UI to update automatically when the state of the application changes. Examples include dice state updates, game statistics updates, and detection results.

- **Singleton Pattern**: Employed for managing shared resources using Dagger Hilt's dependency injection. This avoids the need to manually create singletons. Key singletons include the 'StatisticsManager' for collecting and managing game analytics, the 'GameTracker' for monitoring gameplay sessions, and the 'DataStoreManager' for providing access to persistent storage.

- **Factory Pattern**: Implemented using Dagger Hilt's module system to dynamically provide 'GameManager' instances (such as 'PigGameManager', 'GreedGameManager', 'BalutGameManager') based on the selected game mode. This approach abstracts object creation by separating the responsibility of object creation from the main code logic and creating the instances of the game managers on the go, without having to specify which instance to create, promoting flexibility and scalability in managing different game variants.

5.6 UML Diagrams

This section presents the UML diagrams that illustrate the architecture and dynamic interactions within the application.

5.6.1 Class Diagram

The class diagram provides a detailed view of the static structure of the application, illustrating the key classes, their attributes, methods, and interconnections. It serves as a blueprint for understanding how the application components are organized and how they interact with one another.

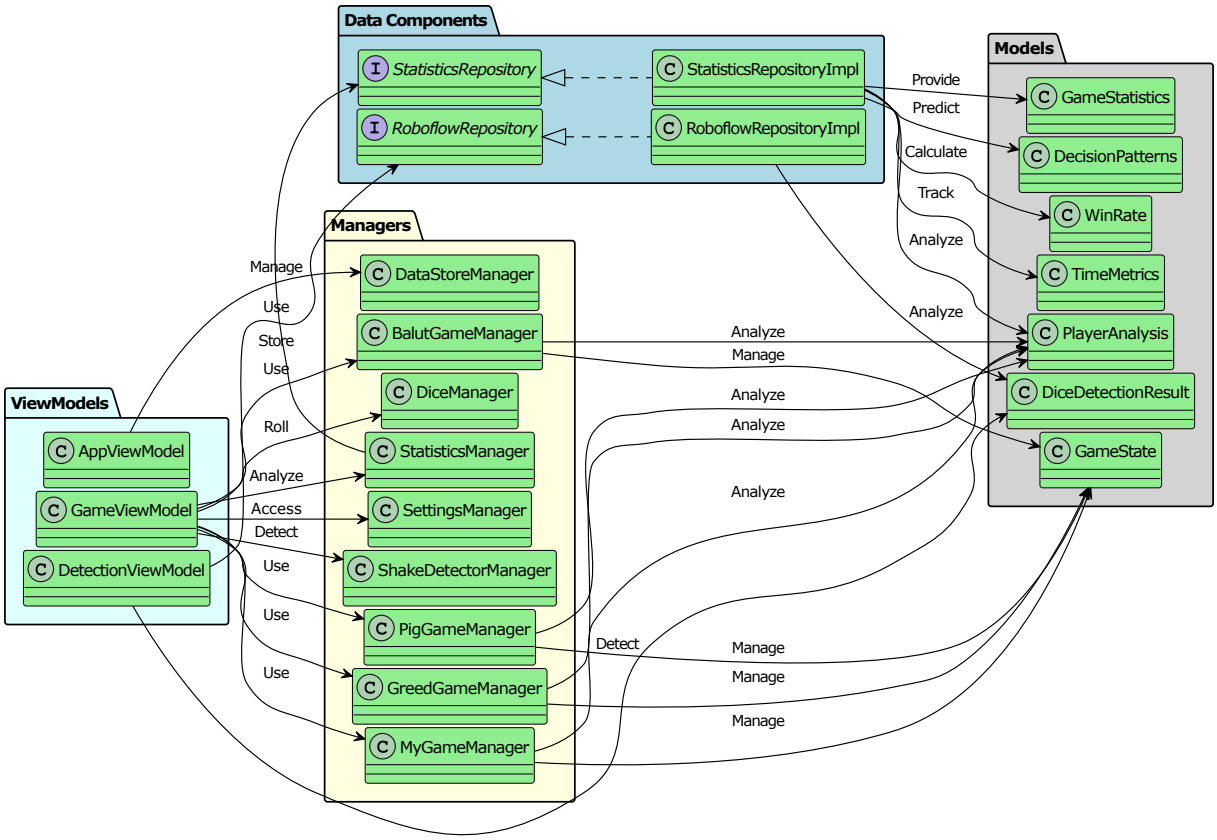


Figure 5.2: Class Diagram of the Application

5.6.2 Models Diagram

The models diagram provides a representation of the data structure of the application, illustrating the relationship between the different data models, and the different attributes associated with each one.

5.6.3 Structure Diagram

The structure diagram gives a high-level representation of the application's package organization, highlighting its modular design and the relationships between various components. This diagram helps in understanding the logical grouping and dependencies within the system.

5.7 Sequence Diagrams

Sequence diagrams are used to illustrate the dynamic interactions between various components and objects within the game. They provide a clear visualization of the message flow and the sequence of operations in key scenarios. In this section, we present sequence diagrams that depict the game flow, virtual mode flow, and analysis flow within the application.

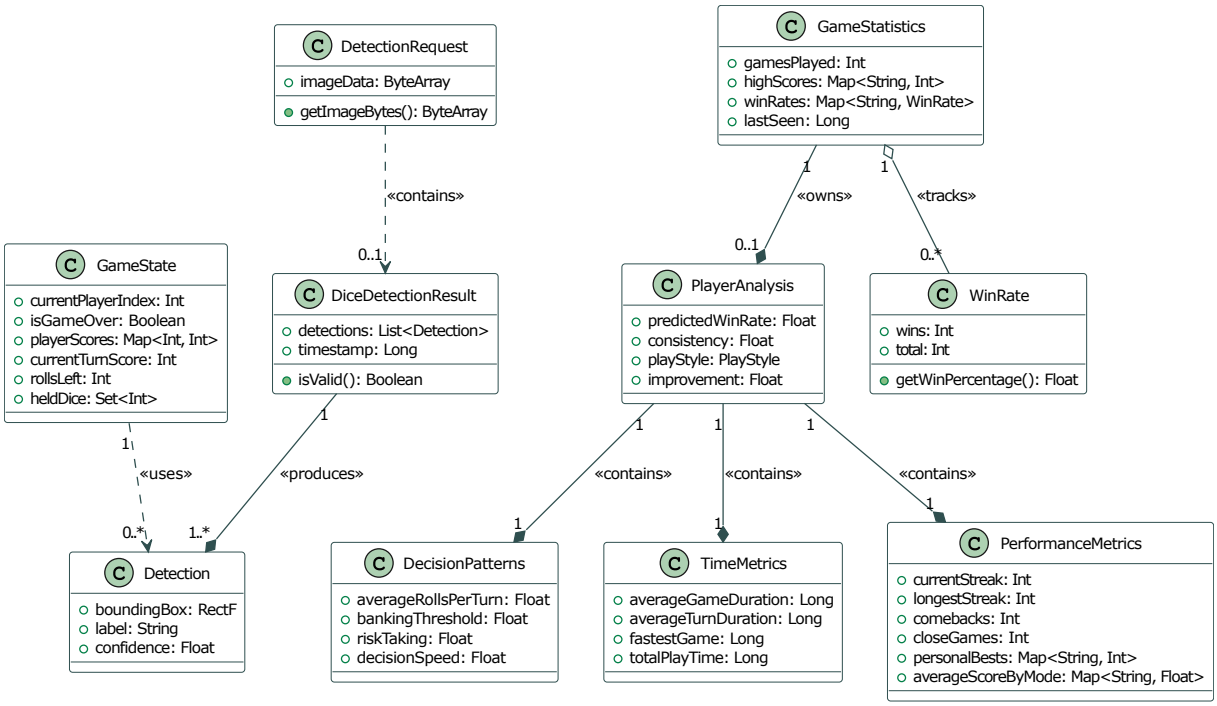


Figure 5.3: Models Diagram of the Application

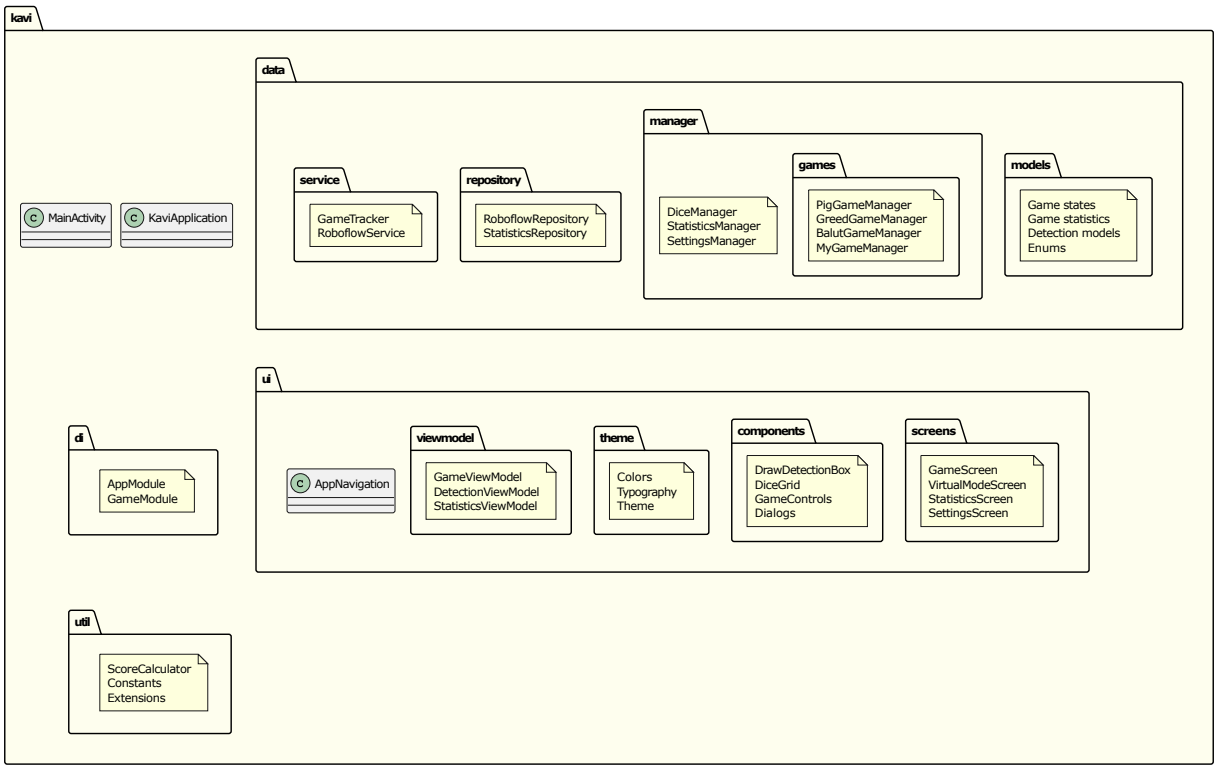


Figure 5.4: Package Structure of the Application

5.7.1 Game Flow Sequence

The game flow sequence illustrates how different components interact during a typical game session. Figure 5.5 shows the process starting from the player initiating a game to the end of the game.

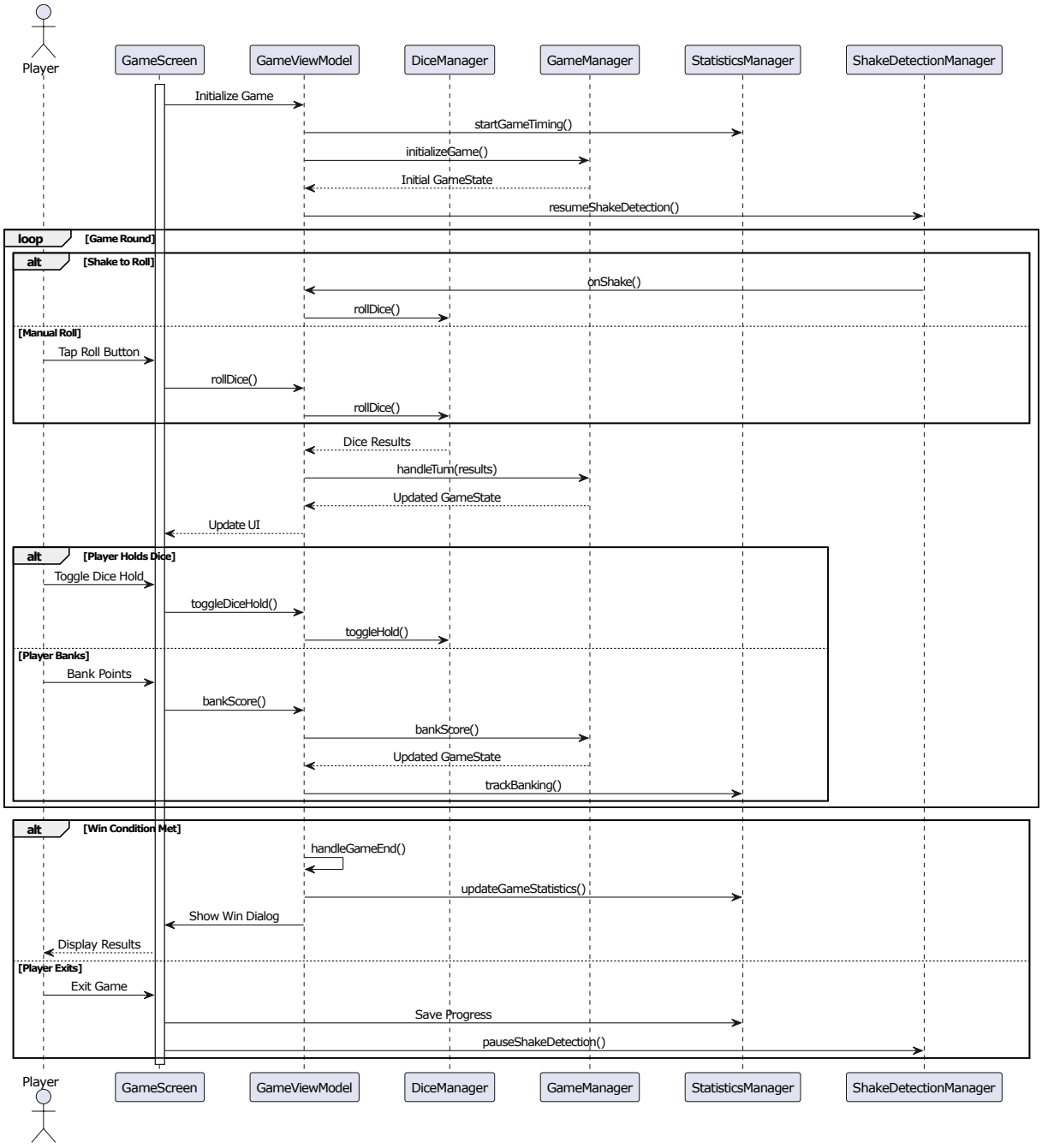


Figure 5.5: Game Flow Sequence in the Application

5.7.2 Virtual Mode Sequence

The virtual mode sequence, depicted in Figure 5.6, demonstrates how the application manages the image capture and dice detection within the virtual mode.

5.7.3 Analytics Flow Sequence

The analytics flow, shown in Figure 5.7, illustrates the steps involved in retrieving and displaying user statistics and analytics.

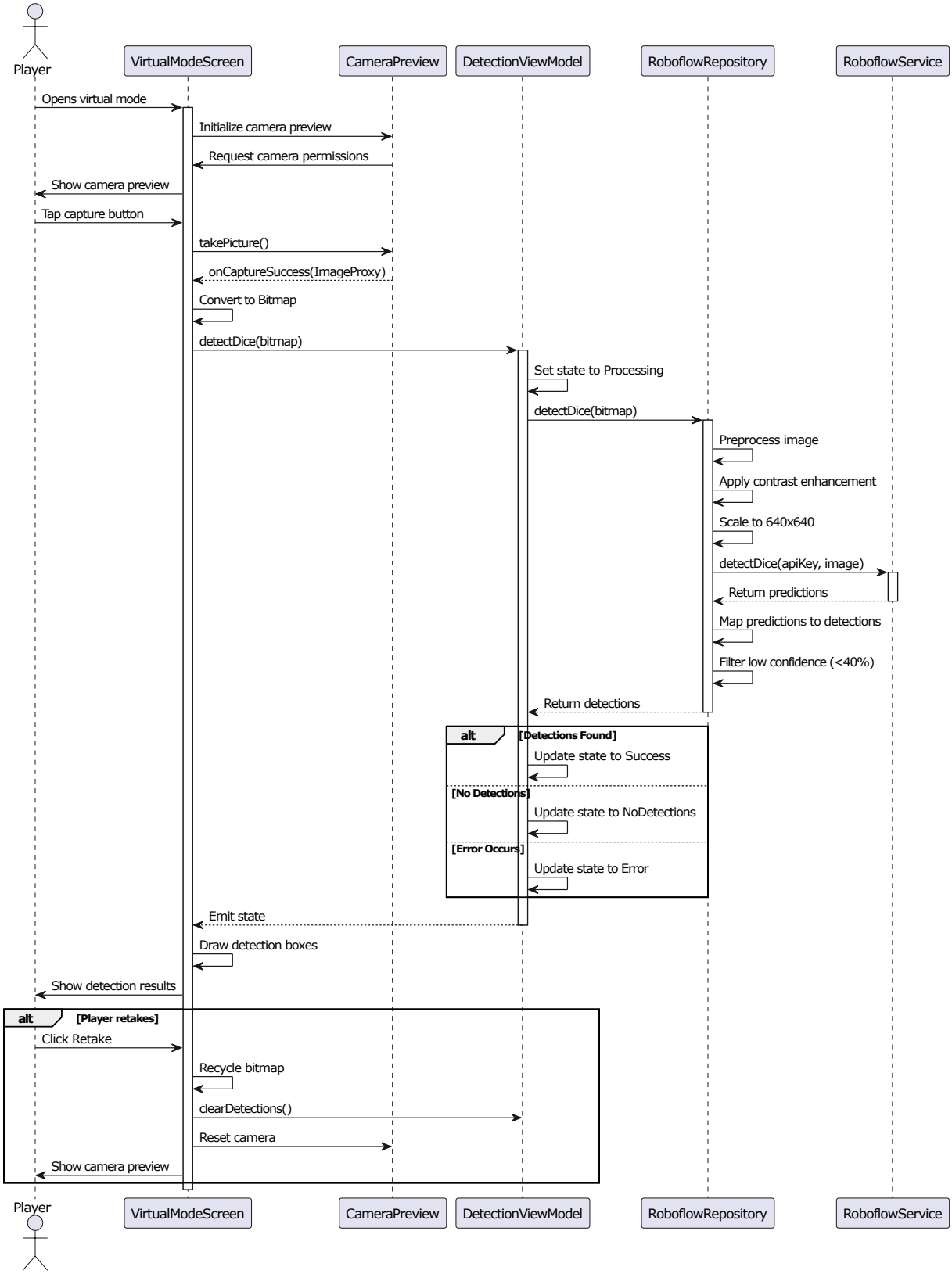


Figure 5.6: Virtual mode Sequence in the Application

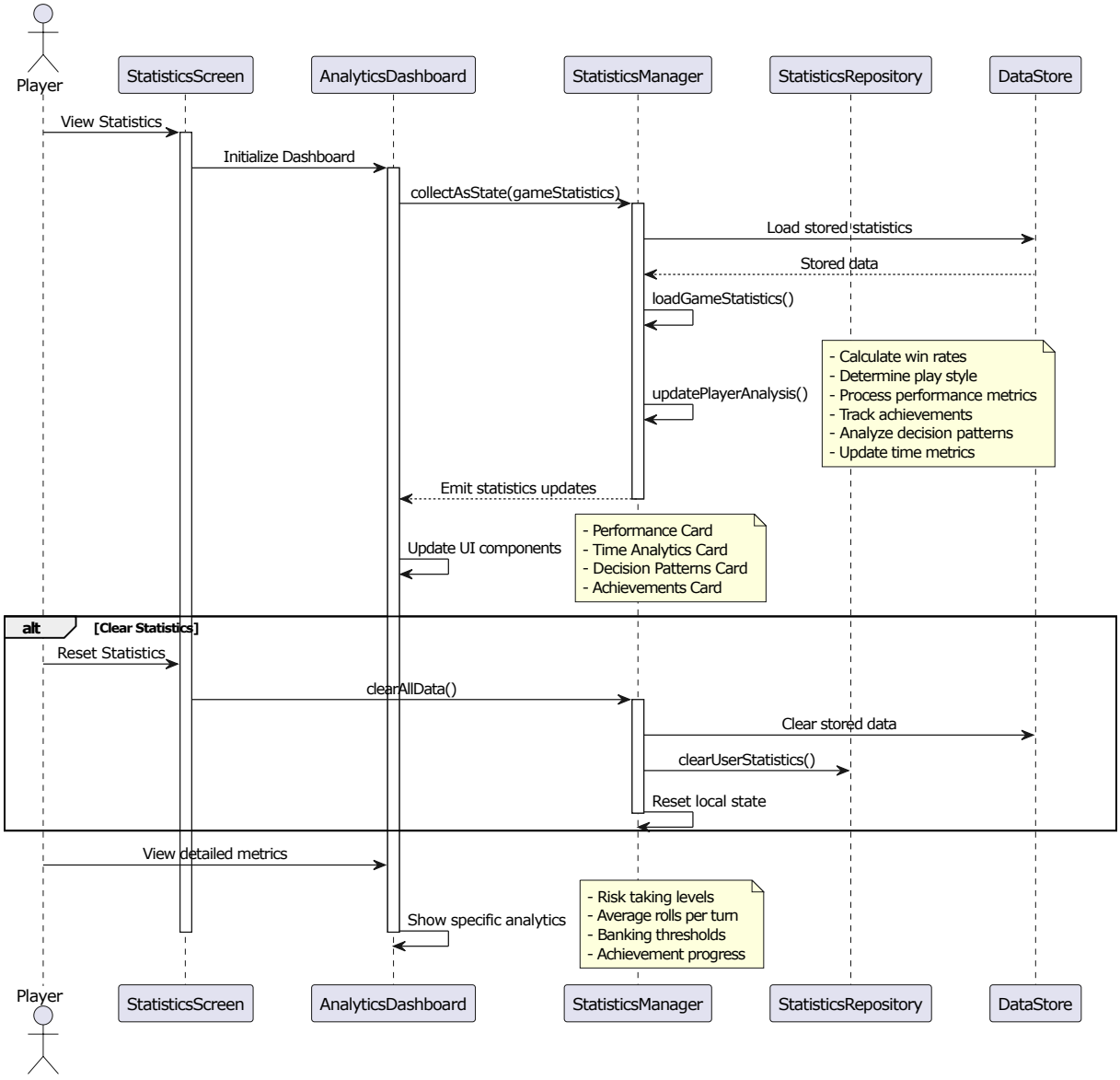


Figure 5.7: Analysis Flow Sequence in the Application

Chapter 6

Verification and Validation

Ensuring the reliability, functionality, and usability of software applications necessitates rigorous verification and validation processes. Verification confirms that the application is built according to design specifications, while validation ensures it fulfils user needs.

This chapter provides an overview of our quality assurance methods during application development. Specifically, it outlines the testing paradigm, test case designs, testing scope, detected and resolved bugs, and experimental results. These efforts aim to build confidence in the application's robustness and overall performance.

6.1 Testing

Test Type	**Purpose**	**Scope**
Unit Testing	Validate individual components or functions.	Core game logic, utility functions.
Integration Testing	Ensure correct interaction between modules.	Game logic and UI, image recognition and AR modules.
System Testing	Verify the complete application works as intended.	End-to-end gameplay, AR interactions.
Regression Testing	Identify defects after changes to the codebase.	Post-fix validation of all modules.
Performance Testing	Measure responsiveness and stability under load.	Frame rate, AI performance with multiple players.

Table 6.1: Summary of Testing Types and Scope

The table above summarizes the different testing types, their purposes, and scopes for

this application.

The V-Model testing paradigm guided the verification and validation of this application. A structured approach, particularly suitable where high-quality assurance is critical [23], the V-Model expands on the traditional Waterfall Model by emphasizing a parallel relationship between development and testing phases (Figure 6.1).

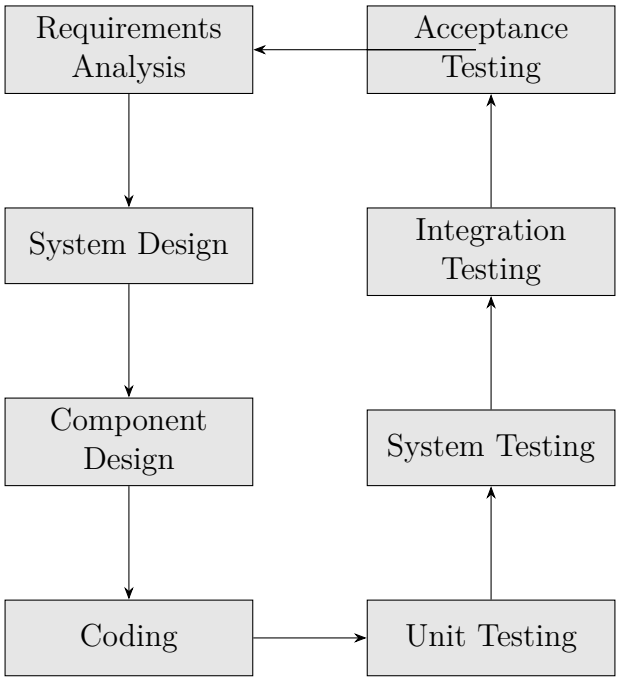


Figure 6.1: Simplified V-Model Diagram

Each development activity was paired with a corresponding testing phase to ensure systematic validation at every stage; for example, requirements analysis was linked to acceptance testing. The V-Model's clarity and emphasis on accountability made it an ideal paradigm for this project, with its aim to deliver a reliable, user-friendly application.

The V-Model incorporates two fundamental types of testing:

- **Verification**: Involves activities like reviews, inspections, and static testing to confirm the application meets specified requirements. For example, unit tests confirm that individual components behave as designed.

```
1    @Test
2    fun `test game initialization`() {
3        val gameState = balutGameManager.initializeGame()
4
5        // Verify initial state
6        assertEquals(2, gameState.playerScores.size)
7        assertTrue(gameState.playerScores[0]?.isEmpty() == true)
8        assertEquals(1, gameState.currentRound)
9        assertFalse(gameState.isGameOver)
10   }
11
```

Listing 6.1: Unit Test for Game Initialization

- **Validation**: Consists of dynamic testing where the final product is evaluated to confirm it meets user requirements and needs. For example, integration tests are done to confirm correct interactions between software modules.

```
    @Test
    fun `detectDice should update state correctly`() = runTest {
        val detections = listOf(mockk<Detection>())
        coEvery { repository.detectDice(any()) } returns detections

        viewModel.detectDice(mockBitmap)
        assert(viewModel.detectionState.value is DetectionState.Processing)

        testDispatcher.scheduler.advanceUntilIdle()
        assert(viewModel.detectionState.value is DetectionState.Success)
    }

```

Listing 6.2: Integration Test for Dice Detection

6.2 Test Cases and Testing Scope

The application's testing strategy utilized full and partial approaches, as outlined below.

6.2.1 Full Testing

Full testing focused on critical components of the application:

- **Game Logic**: Included comprehensive tests of all game rules and scoring mechanics, targeting game manager components:

```
    @Test
    fun `test player rolls 1 and loses turn`() {
        val initialState = pigGameManager.initializeGame()
            .copy(currentPlayerIndex = 0)
        val updatedState = pigGameManager.handleTurn(initialState, 1)

        assertEquals(AI_PLAYER_ID.hashCode(), updatedState.currentPlayerIndex)
        assertEquals(0, updatedState.currentTurnScore)
        assertEquals(0, updatedState.playerScores[0])
    }

```

Listing 6.3: Unit Test for Game Logic

- **User Interface**: Usability and responsiveness of the UI were assessed across different devices and screen sizes.

- **Integration Testing**: Validated the interaction of various application modules and their integration within a single system.

```
    @Test
    fun `integration test gameViewModel`() = runTest {
        val gameState = viewModel.handleTurn(listOf(1))
        assert(gameState.currentPlayerIndex == AI_PLAYER_ID.hashCode())
    }

```

Listing 6.4: Integration Test for Game View Model

6.2.2 Partial Testing

Partial testing was applied to less critical components, with an emphasis on functionality isolation:

- **Unit Testing**: Individual methods were assessed using JUnit tests focused on isolated code.

```
    @Test
    fun `test roll dice for Greed game`() = runTest {
        val results = diceManager.rollDiceForBoard(GameBoard.GREED.modeName)
        assertEquals(6, results.size)
        results.forEach { dice ->
            assertTrue(dice in 1..6)
        }
    }

```

Listing 6.5: Unit Test for Dice Rolling

- **Regression Testing**: Existing functionality was assessed post-fixes and/or new additions to the code base, specifically to check and confirm errors fixed do not manifest and do not effect further components.

```
    @Test
    fun `test game completion`() {
        // Create a state where all categories are filled except one
        val allCategories = BalutGameManager.CATEGORIES
            .associateWith { 10 }
            .toMutableMap()
        allCategories.remove("Choice")

        val initialState = balutGameManager.initializeGame()
            .copy(playerScores = mapOf(0 to allCategories))

        val updatedState = balutGameManager.scoreCategory(
            initialState, listOf(1,2,3,4,5), "Choice")

        assertTrue(updatedState.isGameOver)
        assertEquals(15, updatedState.playerScores[0]?.get("Choice"))
    }

```

Listing 6.6: Regression Test for Game Completion

6.3 Detected and Fixed Bugs

During testing, several issues were identified and subsequently fixed. Notable bugs included:

- **Shake Detection Sensitivity**: Rapid shaking triggered multiple dice rolls, fixed with a debounce mechanism.

```
    fun resumeShakeDetection() {
        shakeDetectionManager.setOnShakeListener {
            viewModelScope.launch {
                shakeFlow.emit(Unit)
            }
        }
        viewModelScope.launch {
            shakeFlow
                .debounce(300) // Added 300ms debounce
                .collect {
                    if (!isRolling.value && isRollAllowed.value)
                        rollDice()
                }
        }
    }

```

Listing 6.7: Fix for Shake Detection Sensitivity

- **Settings Navigation State Loss**: Navigation to the settings screen reset the game state; addressed by using launched effects for navigation to ensure game state retention.

```
    LaunchedEffect(selectedBoard) {
        // Only reset game if board type changes
        if (selectedBoard != currentBoardType) {
            viewModel.setSelectedBoard(boardType)
            viewModel.resetGame()
        }
    }

```

Listing 6.8: Fix for Settings Navigation State Loss

- **Game State Initialization**: Game state not correctly initialized leading to incorrect player scores. Was fixed by ensuring the game state was reset correctly when starting the game.

```
        fun resetGame() = viewModelScope.launch {
        _showWinDialog.value = false
        _heldDice.value = emptySet()
        diceManager.resetGame()
        statisticsManager.startGameTiming()
        }

```

Listing 6.9: Fix for Game State Initialization

- **UI Responsiveness**: Some UI elements were unresponsive on certain devices. This was resolved with optimization in UI rendering to render responsive on diverse display dimensions.
- **Dice Recognition Accuracy**: Dice value misidentification; fixed by refining the image processing techniques with a min-max normalization of pixel brightness values to enhance pips from backgrounds.

6.4 Results of Experiments

During application development, valuable insights into performance and usability were gained through several experiments:

- **Performance Metrics**: A consistent frame rate of 60 FPS during gameplay with single or multiple opponents demonstrates adequate application smoothness.
- **User Feedback**: User testing, conducted across diverse participants, revealed high satisfaction with an average of 4.5 out of 5 in regards to user interface and overall functionality.
- **Bug Fix Impact**: A 75% reduction in user-reported issues was reported post fix implementation, confirming positive user and stability benefits after each release.

Chapter 7

Conclusions

This chapter summarizes the project's key findings and achievements, reflecting on the objectives outlined in the thesis, the challenges encountered during development, and potential paths for future enhancements.

The game project successfully met its primary objectives, resulting in a modern Android application that implements multiple classic and custom dice game variants. The application features three distinct game variants: Pig, Greed, and Balut each with unique rules and gameplay mechanics, which were designed to enhance user engagement and provide a comprehensive gaming experience. An adaptive AI opponent was also developed to challenge players, adjusting its difficulty based on their performance. This AI provides a more engaging experience, and encourages strategic thinking, making the game more dynamic. The application also boasts a user-friendly interface designed with a modern Material Design 3 UI, ensuring an intuitive user experience. The implementation of customizable themes and touch controls enhances accessibility and user satisfaction. The application follows MVVM and Clean Architecture principles, which promote maintainability and scalability. Dependency injection using Hilt and reactive programming with Kotlin Coroutines and Flow were effectively utilized. Finally, the application was validated with a thorough testing strategy, including unit tests and integration tests, which ensured the reliability and stability of the application.

Throughout the development, several challenges were encountered. Implementing the various game rules and ensuring accurate scoring mechanisms proved complex, requiring extensive testing and debugging. Creating an adaptive AI that could effectively challenge players was also a significant hurdle, requiring much trial and error to balance the AI's difficulty level. The design of a user-friendly interface that accommodates various screen sizes also required careful consideration and multiple iterations to achieve the desired outcome. Furthermore, implementing user authentication and data synchronization presented complexities. Setting up and managing user authentication with Firebase, navigating its documentation, and handling different authentication flows was challenging. Similarly, ensuring seamless data synchronization between Firebase and Android's DataStore required

careful management of data consistency and conflict resolution. Finally, the implementation of image recognition was also difficult. The first attempts to train a custom model with TensorFlow Lite, was difficult and very time consuming, and in the end, Roboflow had to be adopted as a solution for its ease of use and effectiveness in handling image recognition tasks.

While the project has achieved significant milestones, several avenues for future development remain. Future updates could include additional game variants or modes, such as multiplayer options or online leaderboard. This would be useful to enhance competitiveness and social interaction among players. Further development could also focus on improving the AI's decision-making algorithms, and providing the option to select the difficulty of the AI. Integrating augmented reality (AR) elements could also provide a more immersive gaming experience, allowing players to interact with the game in new and innovative ways. Expanding the application to support other platforms, such as iOS or web-based versions, could also broaden the user base and increase accessibility. Finally, the implementation of a feedback mechanism within the app could help gather user insights, guide future enhancements, and ensure that the application continuously meets user expectations.

Bibliography

[1] App Annie. *The State of Mobile 2021*. 2021. URL: https://www.slideshare.net/slideshow/app-annie-the-state-of-mobile-2021/241508622/ (visited on 05/01/2025).

[2] Georgios N. Yannakakis and Julian Togelius. *Artificial Intelligence and Games*. Switzerland: Springer, 2018. ISBN: 978-3-319-63519-4.

[3] Dice Detection. *Kavi Dataset*. https://universe.roboflow.com/dice-detection/kavi-zbra1. Available on Roboflow Universe. 2025.

[4] Harsh Munshi. *D3 Deep Dice Detector*. https://github.com/harshmunshi/D3-Deep-Dice-Detector. 2018.

[5] Roboflow. *Roboflow*. URL: https://docs.roboflow.com/ (visited on 15/01/2025).

[6] ultralytics. *YOLOv5 Dice Detection*. https://github.com/ultralytics/yolov5. 2024.

[7] Kotlin. *Kotlin Documentation*. URL: https://kotlinlang.org/docs/android-overview.html (visited on 17/12/2024).

[8] Mayokun Sofowora. *Kavi thesis project*. 2024. URL: https://figma.com/design/LGri6Wuhf4xouWkr1isia0/Kavi-thesis-project?node-id=0-1&p=f&t=ZO6sxtXh6GW9vh82-0 (visited on 15/01/2025).

[9] Git. *Git Documentation*. URL: https://git-scm.com/doc (visited on 17/12/2024).

[10] Android Developer. *Android Studio*. URL: https://developer.android.com/studio/ (visited on 05/10/2024).

[11] Peer D. *Performance Benchmarks Of Jetpack Compose Versus Xml Layouts In Android Applications*. 2024. URL: https://peerdh.com/blogs/programming-insights/performance-benchmarks-of-jetpack-compose-versus-xml-layouts-in-android-applications/ (visited on 15/01/2025).

[12] Android Developer. *Jetpack Compose*. URL: https://developer.android.com/jetpack/compose (visited on 05/10/2024).

[13] Android Developer. *Dependency Injection with Hilt*. URL: https://developer.android.com/training/dependency-injection/hilt-android/ (visited on 10/10/2024).

[14] Lottie. *Lottie Animation*. URL: https://airbnb.io/lottie/#/android (visited on 05/01/2024).

[15] Vico Charts. *Vico Charts Documentation*. URL: https://vicotools.com/charts/documentation (visited on 05/01/2025).

[16] Ravi Tamada. *Android Better Logging using Timber Library*. 2024. URL: https://www.androidhive.info/2024/10/android-better-logging-using-timber.html (visited on 06/01/2025).

[17] Stefan Bechtold, Sam Brannen, Johannes Link, Matthias Merdes, Marc Philipp, Juliette de Rancourt and Christian Stein. *JUnit 5 User Guide*. 2024. URL: https://junit.org/junit5/docs/current/user-guide/#writing-tests (visited on 06/01/2024).

[18] Mayokun Sofowora. *Kavi thesis project*. https://www.figma.com/design/LGri6Wuhf4xouWkr1isKavi-thesis-project?node-id=0-1&t=70pukjZzeoLWMij3-1. 2024.

[19] Shreyansh Saurabh. *Dice*. https://github.com/binaryshrey/Dice/tree/main. 2021.

[20] *Android Data Backup Documentation*. URL: https://developer.android.com/guide/topics/data/backup (visited on 07/01/2025).

[21] *OWASP Top 10, Security Misconfiguration*. URL: https://owasp.org/www-project-top-ten/OWASP_Top_Ten_2017/Top_10-2017_A06-Security_Misconfiguration (visited on 07/01/2025).

[22] *Android code obfuscation documentation*. URL: https://developer.android.com/build/shrink-code (visited on 07/01/2025).

[23] Kevin Forsberg and Harold Mooz. *The V-Model: Application and Implementation in Software Engineering*. 2nd. New York: Springer, 2009.

Appendices

Index of abbreviations and symbols

UI	User Interface.
AI	Artificial Intelligence.
API	Application Programming Interface.
ML	Machine Learning.
UML	Unified Modelling Language.
AR	Augmented Reality.
RAM	Random Access Memory.
GB	Gigabyte.
MB	Megabyte.
ARM	Advanced RISC Machine.
RISK	Reduced Instruction Set Computing.
SDK	Software Development Kit.
APK	Android Package Kit.
MVVM	Model View ViewModel.
UX	User Experience.
CNN	Convolutional Neural Network.

Listings

List of additional files in electronic submission

Additional files uploaded to the system include:

- source code of the application,
- test data,
- a video file showing how the application developed for the thesis is used.

List of Figures

List of Tables