Department of Electrical Engineering
香港城市大學
City University of Hong Kong

Department of Electrical Engineering

FINAL YEAR PROJECT REPORT

BENGEGU4-CDE-2022/23-LMP-03

Deep Learning based Facial Action Control Mobile Game (Android Version)

Student Name: Cheng Darren
Student ID: 55819461
Supervisor: Dr Po, L M
Assessor: Mr. Ting, Van C W

Bachelor of Engineering in Computer and Data Engineering

Student Final Year Project Declaration

I have read the student handbook and I understand the meaning of academic dishonesty, in particular plagiarism and collusion. I declare that the work submitted for the final year project does not involve academic dishonesty. I give permission for my final year project work to be electronically scanned and if found to involve academic dishonesty, I am aware of the consequences as stated in the Student Handbook.

Project Title: Deep Learning based Facial Action Control Mobile Game (Android Version)	
Student Name: Cheng Darren	**Student ID: 55819461**
Signature:	**Date: 1/4/2023**

Abstract

Facial muscles are muscles located in the face and neck. Just like with other muscles in the body, exercising them can help to tone and firm them up, reducing wrinkles and improving blood circulation around the face. However, facial muscle exercises have always been neglected. Many people find it challenging to regularly do facial exercises on their own.

To promote facial exercise, I think it is best to implement it into a game. As the incorporation of facial recognition technology in mobile gaming may significantly enhance user experience and a game with good user experience provides players with satisfying and immersive experience that keeps them engaged and entertained.

This project involves creating an Android game that features facial action controls. It aims to study which type of facial movements are suitable for mobile game control and to use it as the main controls for the game. We hope to create a new type of mobile game that brings a unique user experience, with the objective of promoting facial muscle exercise to reduce the aging process of the face.

A knife/dagger throwing game was chosen owing to its simplicity and potential for encouraging players to engage in facial movements that stimulate facial muscles, thereby promoting their health.

An existing version of the game (without facial controls) was found on the Internet with free source code (see Appendix A for details). However, the code was written in Java and not suitable for new developers like me. As the project required the use of Kotlin Jetpack Compose, the game was **redeveloped from scratch**, making use of some image resources from the original source, along with my modifications to the UI and entirely new game logics (please note that I only used some of the image resources, but not any of the code provided).

MLKit on the other hand, is a SDK that provides facial detection features for mobile applications. Within the game, all classification modes available in the API have been utilized, and the logic is implemented using Coroutine Scopes, resulting in the development of eye-blinking and smiling motion controls into the game.

Once the game was complete, a questionnaire was conducted to gather feedback from different age groups to determine if the facial muscle exercise promoting features of the game were effective. The results are surprisingly positive, indicating the success of the project.

Overall, this project provides an innovative approach to promote facial muscle exercise, and offered a unique and engaging user experience that encourages players to regularly train their facial muscles. The utilization of facial recognition technology in mobile gaming provides us an exciting opportunity to enhance user experience and promote healthy habits.

Acknowledgements

I would like to express my sincere gratitude to Dr Po L M and Mr. Ting, Van C W for their guidance and support throughout the whole project.

Dr Po organized monthly meetings to support me on different challenges I faced in this project. His monthly meetings have been instrumental in keeping me on track and ensuring that I make steady progress towards my goals. I am very grateful for his suggestions, which have saved me a large amount of time and effort in researching for the tools used in my project. His expertise and insights helped me overcome various challenges and obstacles.

Mr. Ting has been instrumental in keeping me on the right track and ensuring that I focus not only on the technical aspects of the game but also on the user experience. His suggestions have inspired me to think from the user's perspective and consider how to bring the best experience and facial muscle training to users. I am sure that without his guidance, I may have neglected important aspects of the game.

Lastly, I am truly grateful for both professors' unwavering support and encouragement throughout this project. Thank you, Dr Po and Mr. Ting, for your contributions to this project.

Contents

ABSTRACT ..... **I**

ACKNOWLEDGEMENTS ..... **III**

CONTENTS ..... **IV**

LIST OF FIGURES ..... **VI**

LIST OF TABLES ..... **IX**

1. INTRODUCTION **1**

1.0 Dagger Master 3
1.1 Background 4
1.1.1 Impacts of Facial Muscle Exercise *4*
1.1.2 Factors that drive User Engagement *6*
1.2 Objectives 7

2. METHODOLOGY **8**

2.1 Set Up 9
2.2 Install Dependencies 10
2.3 Navigation Set Up 10
2.4 File Structure 12
2.5 Screens 12
2.5.1 Loading Screen *13*
2.5.2 Landing Screen *14*
2.5.3 Game Screen *17*
2.5.4 Shop Screen *18*
2.5.5 Settings Screen *21*
2.6 States 23
2.6.1 Game State *23*
2.6.2 Dagger State *25*
2.6.3 Spinner State *28*
2.6.4 Fruit State *29*
2.6.5 Remaining Daggers State *32*
2.7 Utils 33
2.7.1 Status Bar Util *33*
2.7.2 Orientation Util *34*
2.7.3 Spinner Util *35*
2.7.4 Dagger Util *35*
2.7.5 Permission Util *37*
2.7.6 Settings Util *38*
2.7.7 Game Util and Data Store *41*
2.7.8 Utils Setup *43*
2.8 Global Variables 44
2.9 Game Console 45
2.10 Face Detection 48

2.11 Questionnaire 50

3. RESULTS 53

3.1 Loading Screen 53
3.2 Landing Screen 55
3.3 Game Screen 57
3.4 Shop Screen 59
3.5 Settings Screen 60
3.6 Questionnaire Results 62

4. DISCUSSION 68

5. CONCLUSION 69

APPENDICES 70

Appendix A: Free Source Code Link 70
Appendix B: "Knife Hit" on Google Play Store 70
Appendix C: Navigation Animation Library Link 70
Appendix D: Detail implementation of all UI Components 70
Appendix E: GitHub Source code for Dagger Master 70
Appendix F: Questionnaire 70

REFERENCES: 71

List of Figures

Figure 1	**Flappy Bird game scene**	**p.2**
Figure 2	**Dagger Master game scene**	**p.2**
Figure 3	**Dagger Master Logo**	**p.3**
Figure 4	**Project Creation Page**	**p.9**
Figure 5	**Dependencies added to build.gradle**	**p.10**
Figure 6	**Code snippet for Navigation and Navigation Animation**	**p.11**
Figure 7	**File Structure of Dagger Master**	**p.12**
Figure 8	**Overview of game flow**	**p.12**
Figure 9	**Loading Screen Implementation**	**p.13**
Figure 10	**all Landing Screen Animations**	**p.14**
Figure 11	**Landing Screen general UI items**	**p.14**
Figure 12	**Landing Screen specific UI items**	**p.15**
Figure 13	**Game Mode Screen in Landing Screen**	**p.16**
Figure 14	**Game Screen Top Component**	**p.17**
Figure 15	**Shop Screen Implementation**	**p.18**
Figure 16	**Draw Shop Item Implementation**	**p.20**
Figure 17	**Settings Screen (Part 1)**	**p.21**
Figure 18	**Settings Screen (Part 2)**	**p.22**
Figure 19	**Settings Item**	**p.22**
Figure 20	**Game State data class**	**p.23**
Figure 21	**Game Mode data class**	**p.25**
Figure 22	**Dagger and Sparkle data object in Dagger State**	**p.25**

Figure 23 **Dagger State data class (Part 1)** **p.26**

Figure 24 **Dagger State data class (Part 2)** **p.27**

Figure 25 **Dagger State data class (Part 3)** **p.27**

Figure 26 **Spinner State data class** **p.29**

Figure 27 **Fruit data class** **p.29**

Figure 28 **Fruit State data class (Part 1)** **p.30**

Figure 29 **Fruit State data class (Part 2)** **p.31**

Figure 30 **Remaining Daggers State data class** **p.32**

Figure 31 **Status Bar Util** **p.33**

Figure 32 **Orientation Util** **p.34**

Figure 33 **Spinner Util** **p.35**

Figure 34 **Dagger Util** **p.36**

Figure 35 **Permission Util** **p.37**

Figure 36 **Settings Util (Part 1)** **p.38**

Figure 37 **Settings Util (Part 2)** **p.39**

Figure 38 **Game Data class for Data Store** **p.41**

Figure 39 **Game Util** **p.42**

Figure 40 **Utils Setup** **p.43**

Figure 41 **Global Variables** **p.44**

Figure 42 **Game Console Implementation (Part 1)** **p.45**

Figure 43 **Coroutines in Game Console** **p.46**

Figure 44 **Game Console Implementation (Part 2)** **p.47**

Figure 45 **Face Detection Analyzer** **p.48**

Figure 46 **Draw Camera Implementation** **p.49**

Figure 47 **Questionnaire (Part 1)** **p.50**

Figure 48 **Questionnaire (Part 2)** **p.51**

Figure 49 **Questionnaire (Part 3)** **p.52**

Figure 50 **Loading Screen Demo GIF** **p.53**

Figure 51 **Loading Screen Sword Appear** **p.54**

Figure 52 **Icon bounces to top** **p.54**

Figure 53 **Landing Screen Demo GIF** **p.55**

Figure 54 **Game Mode selection page** **p.56**

Figure 55 **Tap Mode gameplay GIF** **p.57**

Figure 56 **Facial Mode gameplay** **p.58**

Figure 57 **Score Board Overview** **p.58**

Figure 58 **Shop Screen Demo** **p.59**

Figure 59 **Settings Screen Demo GIF** **p.60**

Figure 60 **Settings Screen Overview** **p.61**

Figure 61 **Questionnaire Q1 Result** **p.62**

Figure 62 **Questionnaire Q2-Q4 Result** **p.63**

Figure 63 **Questionnaire Q5-Q6 Result** **p.64**

Figure 64 **Questionnaire Q7-Q10 Result** **p.65**

Figure 65 **Questionnaire Q11-Q14 Result** **p.66**

List of Tables

Table 1	**Impact of Blinking Motion**	**p.4**
Table 2	**Impact of Smiling Motion**	**p.5**
Table 3	**Technology used in Dagger Master**	**p.8**
Table 4	**Game Status and their use cases**	**p.24**
Table 5	**Table showing all Settings**	**p.40**

1. Introduction

This project is centered around using deep-learning-based computer vision technology to create a facial action control mobile game in Android that promotes facial muscle exercise to reduce the aging process of the face. As I have also mentioned in the above Abstract section, this project aims to:

1. Study which types of facial movements are suitable for mobile game control.
2. Use facial recognition technology as the main controls for the mobile game.
3. Create a new type of mobile game that brings a unique user experience.

Facial muscle exercises have been found to offer many positive benefits to our health, while mobile games have become increasingly popular in recent years due to their accessibility and convenience. With smartphones being an integral part of our daily life, mobile games can be accessed pretty much anytime. To promote facial muscle exercises, why don't we simply implement facial action controls into a game, so people can train their facial muscles during gameplay?

However, promoting facial muscle exercises through mobile games is still a relatively unexplored area, there are several unknown factors that I will have to investigate. For instance, we still don't know whether players will enjoy this type of mobile game. As the incorporation of facial action controls into mobile games significantly increases game's difficulty, making even a simple game like Flappy Bird extremely challenging to control. Hence, by the end, this project will serve as an experiment to investigate whether people like this kind of game, and whether they will consistently play it. Through this experiment, we hope to gain insights into the potential of incorporating facial action controls into mobile games and their effectiveness in training facial muscles.

As for the game choice, after thinking about the types of mobile games that are suitable for facial action controls, I have concluded that a knife/dagger throwing game would be the most

appropriate choice for this project as this game offers a simple gameplay mechanic that encourages players to engage in facial movements. Unlike other simple and reactive 2D games like Flappy bird (where you have to keep tapping the screen to survive), this game doesn't force players to perform facial actions. With that said, players can take as much time as they want and perform the facial actions anytime they feel comfortable with it.

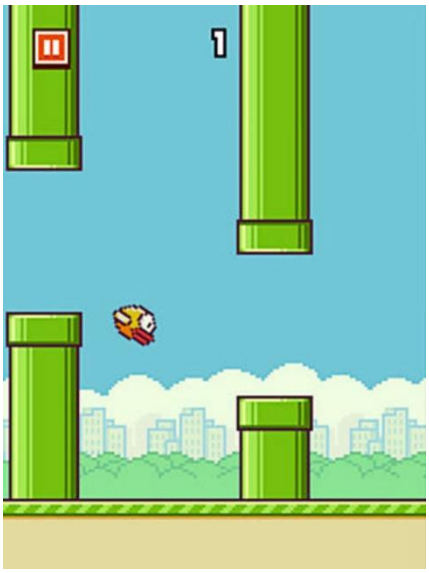


Figure 1. Flappy Bird game scene

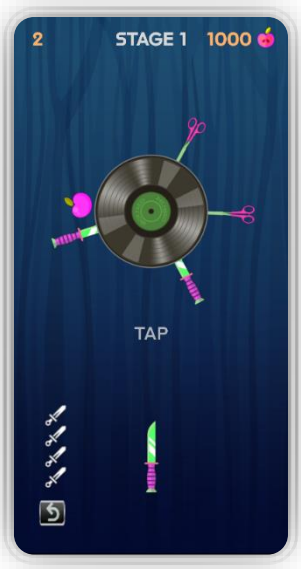


Figure 2. Dagger Master game scene

By the end of this project, in order to assess how effective this game is, I have conducted a survey to gather feedback from individuals across various age groups. This survey comprises of 16 straightforward questions that aim to evaluate the game's responsiveness, feasibility, and effectiveness.

1.0 Dagger Master

Figure 3. Dagger Master Logo

My game is named “Dagger Master” and it is a Dagger throwing game. It challenges players to throw daggers at rotating targets without hitting other daggers that are already on the target. The inspiration of Dagger Master came from a classic, childhood game named “Knife Hit” (see Appendix B for details), which had brought me so much joy and got me addicted in my youth. My hope is that this “enhanced with facial action controls” version will do the same for players around the world.

1.1 Background

1.1.1 Impacts of Facial Muscle Exercise

Facial muscle exercises have been found to help reduce the aging process of the face by:

1. Promoting blood circulation
2. Toning Muscles
3. Reducing Wrinkles

There are in fact a lot of facial action controls available for games, for instance, mouth movements, blinking motion, tongue movements, smiling motion, eyebrow movements, etc. In Dagger Master, I specifically chose Blinking motion and Smiling motion, simply because I like to. Let's briefly look at the impacts and muscle training associated with Blinking and Smiling.

Blinking Motion: Table 1. Impact of Blinking Motion

Facial muscles trained:
• Orbicularis oculi muscle (surrounds the eye). This muscle plays a big role in facial expression, and also in reflexes such as squinting and blinking. [1]
Pros:
• Blinking is an essential action for maintaining healthy eyes and vision, as it moisturises the eyes, keeping your eyes lubricated while providing protection and reducing eye strain [2]. It also improves vision and stimulates brain activity. Besides, research also discovered that gamers tend to blink less frequently [3]. So blinking motion in game solves this problem.
Cons:
• Blinking too repeatedly could be irritating, which can be very uncomfortable and make it difficult to continue playing the game [4]. Also, some people might not have strong control over their blinking (e.g., if they have dry eyes or a medical condition), which could negatively affect the gameplay.

Smiling Motion: Table 2. Impact of Smiling Motion

Facial muscles trained:
• Zygomaticus muscles (raises the corners of the mouth) and orbicularis oculi (causes crow's feet). This muscle is responsible for smiling and is important for facial expression and communication.
Pros:
• Smiling releases endorphins, improves your mood, relieves stress, boosts immune system, lowers blood pressure, and makes you more attractive and approachable [5]. Even if you forcefully or intentionally smile, you can still trigger the same positive benefits associated with a genuine smile [6].
Cons:
• Like blinking, smiling could cause facial fatigue or strain if the player has to do it repeatedly for a prolonged period. Also, some people might find it difficult or uncomfortable to smile for extended periods of time, especially if they have dental braces, sore teeth, or other oral health issues.

We can see that repeatedly performing smiling or blinking actions over a prolonged period of time can have negative effects. To address this issue, I incorporated a game mode system in which players can freely choose from different game modes. This allows them to switch to a different mode if they become tired, enabling them to continue playing without/with less negative consequences.

1.1.2 Factors that drive User Engagement

To promote facial muscle training through mobile games, I believe the most important aspect is to keep players engaged, and addicted. As players won't get any benefits from facial muscle training in this game if they play it less than once a week. Hence, before I started developing this project, I did a little research on the driving factors that boost user engagement.

Mobile game engagement is similar to customer engagement for other products. While in mobile games, this engagement is achieved when users realize value from the product, or when they became so "hooked" and entertained that they want to keep playing [7].

A study identified four key factors of engagement in games, **three of which** are aligned with the subscales of the User Engagement Scale, namely:

1. Perceived Usability,
2. Aesthetics,
3. Focused Attention,
4. Endurability,
5. Felt Involvement,
6. Novelty

And the **fourth** factor is "Satisfaction" [7].

To create an engaging and addictive Android game, I have tried my best to incorporate these factors into the design of Dagger Master.

1.2 Objectives

By the end of this project, Dagger Master is developed. This game is developed to be Android compatible and with facial action controls implemented. This project has 4 main objectives, which are:

1. **Promote facial muscle exercise to reduce the aging process of the face.**
   Our goal is to encourage facial muscle exercise through the incorporation of facial recognition technology into mobile games. Players can benefit from facial muscle training through gameplay, and experience the "satisfaction" of playing the game, creating a "win-win situation".

2. **Study which types of facial movements are suitable for mobile game control.**
   At the conclusion of our study, we hope to determine whether players find the smiling and blinking controls enjoyable in the game. A questionnaire is conducted to collect feedback from various age groups on their preferences, researching on their level of satisfaction with these new features.

3. **Create a new type of mobile game that brings a unique user experience.**
   Dagger Master is a unique mobile game that sets itself apart from traditional games by incorporating facial actions as a means of control. By introducing this innovative gameplay mechanic, I hope it can make a significant contribution to the mobile game industry and offer players a new and exciting game experience.

2. Methodology

The methodology of this project can be divided into two phases: the Development phase and the Research phase. These phases were designed to ensure an efficient approach to the creation and study of Dagger Master.

The development of Dagger Master utilized a wide range of technical tools, with Android Studio serving as the primary Integrated Development Environment (IDE) throughout the project. Under the guidance of my supervisor, Dr. Po, I selected Kotlin as the primary programming language and Jetpack Compose as the framework/toolkit. These choices proved to be highly effective, significantly boosting my productivity and making the development process more streamlined and direct. The OOP nature of Kotlin Jetpack Compose has allowed me to easily understand the flow of the code, and enabled me to produce code that is well-organized and efficient.

Besides the above mentioned tools, external libraries are also utilized. This table below shows briefly all the technology used throughout this project.

IDE	Language	Toolkit	Facial Action Control Library	Tools & Dependencies
•Android Studio	•Kotlin	•Jetpack Compose	•Google ML Kit -> Facial detection	•Coroutines •Navigation Compose •Navigation Animation •Datastore •Camera

Table 3. Technology used in Dagger Master

Google ML Kit's Facial Detection module is the library that powers all the eye blinking and smiling control. **Navigation Compose** and **Navigation Animation** allowed me to carry out smooth and animated navigations throughout the game, **Datastore** allowed Dagger Master to

store game data in local storage, **Camera** library allowed the use of camera in games, and **coroutines**, the most important library in Dagger Master, managed all the asynchronous tasks and state changes throughout the game.

In the upcoming subsections, you will see a lot of functions starting with the word “Draw”. They are composable functions with the source code in utils/UIComponent.kt. Since there are a lot of components and each of them contains mainly groups of images and texts. I will not go through every single one of them in this report. If you are interested in any component’s implementation, please check Appendix D.

2.1 Set Up

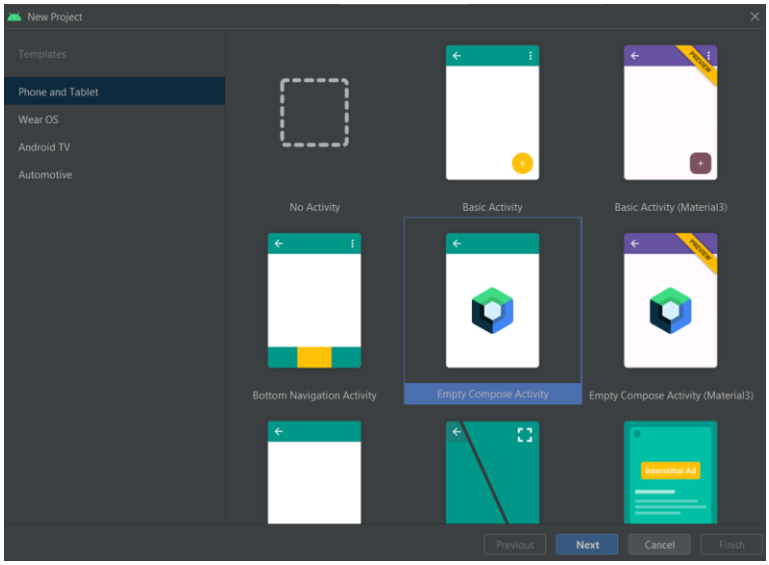


Figure 4. Project Creation Page

To start with, create a new project with “Empty Compose Activity” template in the center. It will directly set up everything, with a file named “MainActivity.kt” as the starting point. With that said, the project is good to go.

2.2 Install Dependencies

The next step would be installing the necessary dependencies (check Table 3 Tools and Dependencies field for details). To install dependencies, go to file “build.gradle” in Gradle Scripts. There are 2 Gradle files, 1 for the project, and 1 for the app. Choose the Gradle file for the app and add the below lines (figure 5) into the dependencies object. Also, any dependencies can be updated if necessary.

```
//Navigation API
implementation("androidx.navigation:navigation-compose:2.5.3")
implementation "com.google.accompanist:accompanist-navigation-animation:0.28.0"
//DataStore API
implementation "androidx.datastore:datastore-preferences:1.0.0"
//Google ML Kit
implementation "androidx.camera:camera-camera2:1.2.2"
implementation "androidx.camera:camera-lifecycle:1.2.2"
implementation "androidx.camera:camera-view:1.2.2"
implementation "com.google.mlkit:face-detection:16.1.5"
```

Figure 5. Dependencies added to build.gradle

2.3 Navigation Set Up

During the initial stages of development, the establishment of Navigation was prioritized, as it allowed me to categorize the project according to different screens, which allowed me to create a new file for each corresponding screen for further development. Figure 6 demonstrates how to set up the routes to different screens within the game. To further enhance user experience, Navigation Animation is also utilized, which allowed me to customize enter and exit animations of each screen.

The library of Navigation Compose Animation (see Appendix C for details) provided me with a wide range of animations for customizing the enter, exit, pop enter, and pop exit transitions within Dagger Master.

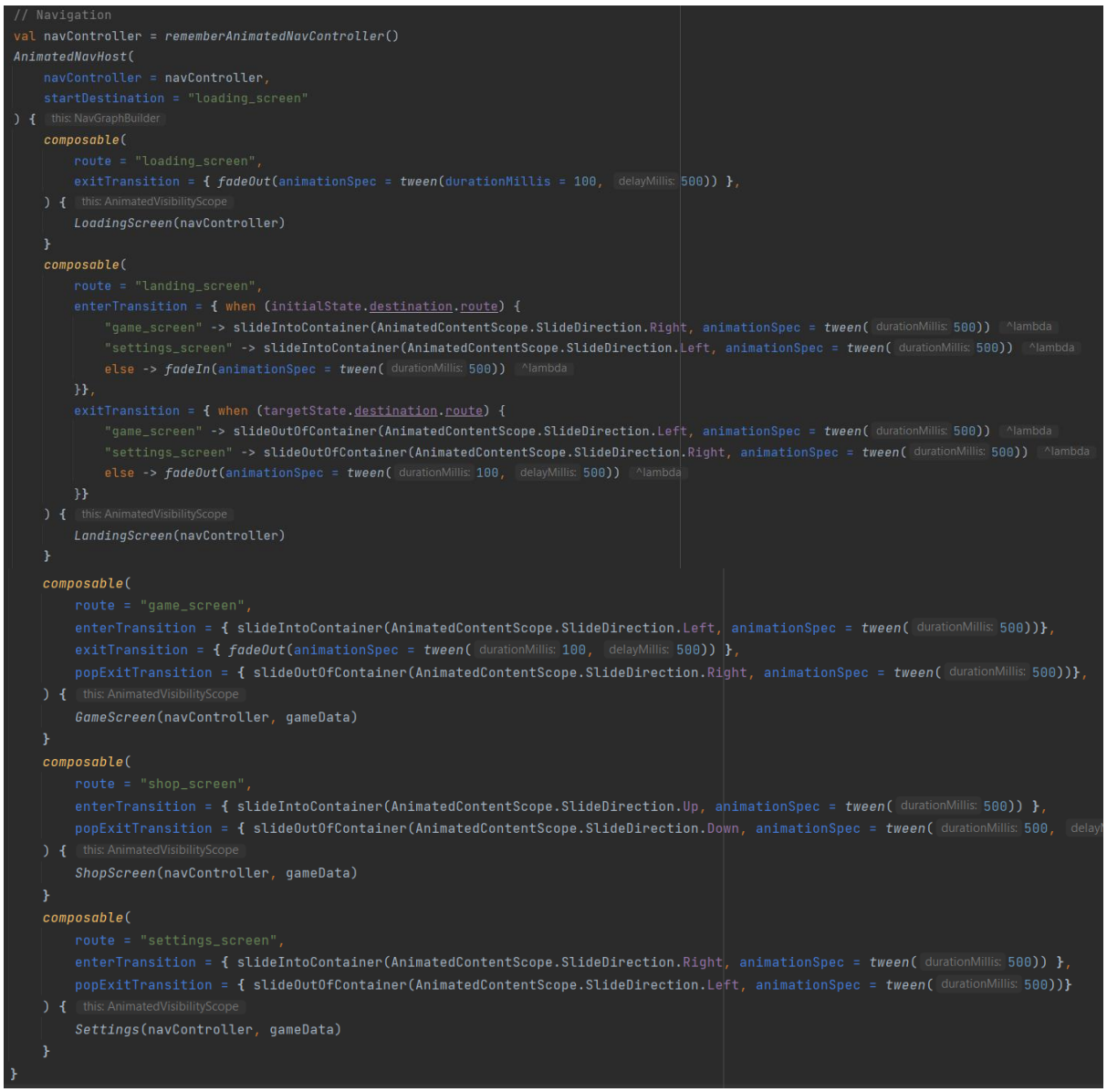


Figure 6. Code snippet for Navigation and Navigation Animation

From the above code snippet (figure 6), I have set up 5 routes in total, with the loading screen as the starting destination of the app.

2.4 File Structure

Before I start introducing the implementations, I would like to introduce the file structure of Dagger Master. MainActivity.kt is the starting point of the app. In the **screens** folder, we have 5 files corresponding to each screen. The **states** folder stored all the states used during gameplay. While the **utils** folder stored utils and all the general items that are shared among different files.

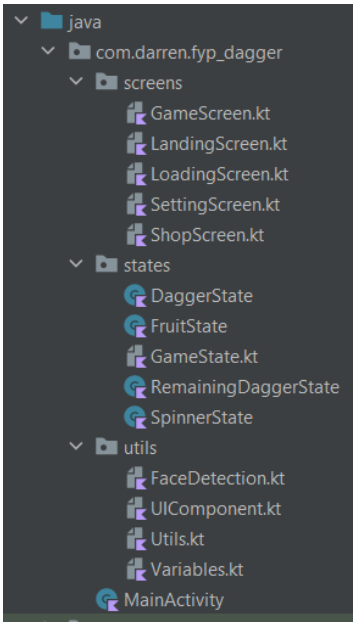


Figure 7. File Structure of Dagger Master

2.5 Screens

This section will focus on the UI/UX development for each screen. Each screen has different designs and serves specific purposes. There are 5 screens in total, which corresponds to Loading Screen, Landing Screen, Game Screen, Shop Screen and Settings Screen.

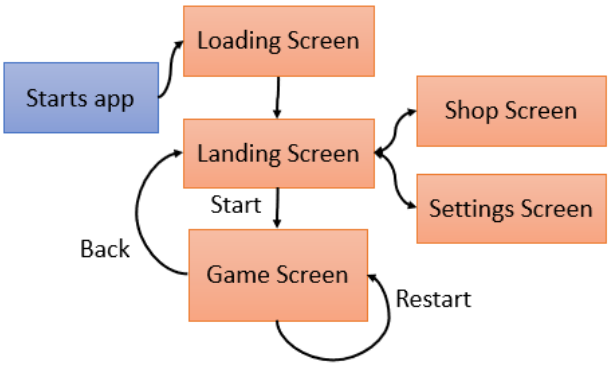
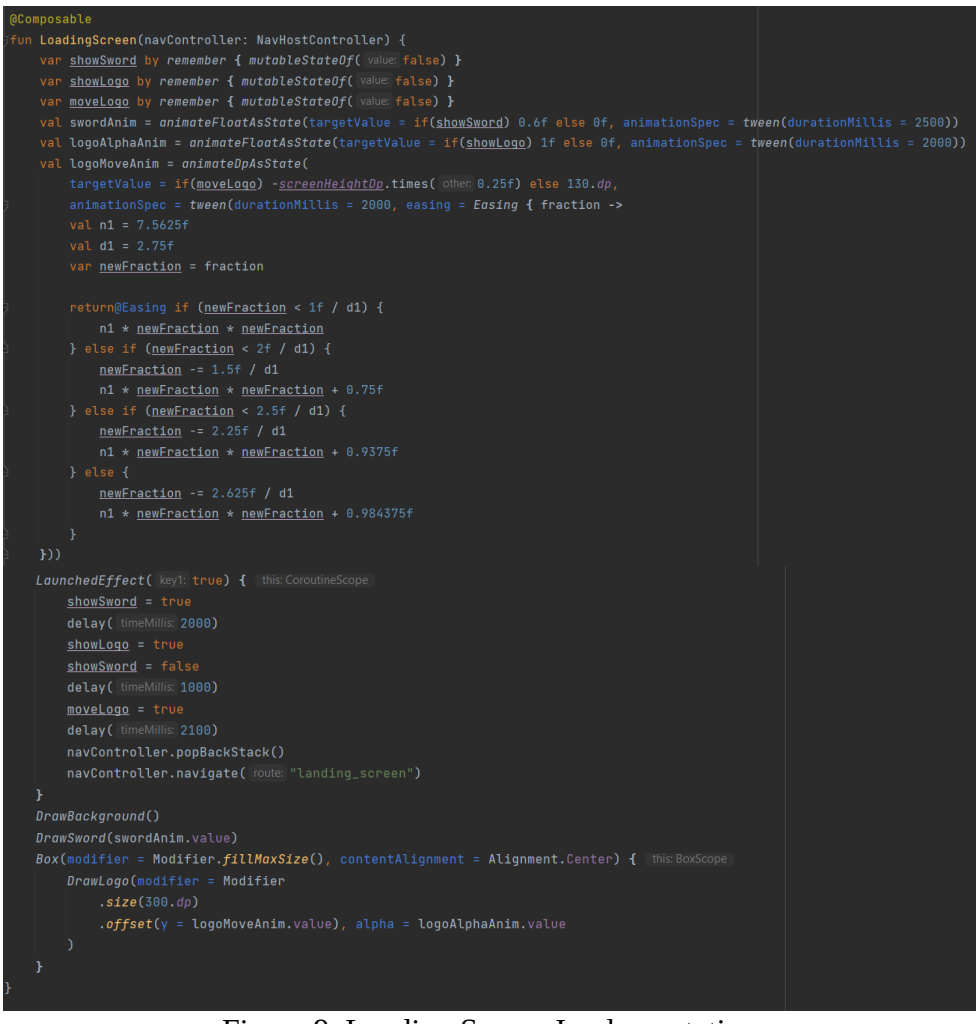


Figure 8. Overview of game flow

2.5.1 Loading Screen

The loading screen is designed to provide a cool looking UI with animations that attract players once they enter the game. The upper part of figure 9 illustrated how animations are initialized using three boolean states that trigger the “animateXXXAsState” function to animate a value of type “XXX” whenever the boolean state changes. The lower part of figure 9 shows the UI components. The background is drawn first, followed by a sword with an alpha animation attached, and lastly, the game logo. “LaunchedEffect” with key of true means it only runs once every time the loading screen recomposes. Within this block, the boolean states are toggled, activating the animations, then navigating to the landing screen.



```
@Composable
fun LoadingScreen(navController: NavHostController) {
    var showSword by remember { mutableStateOf( value: false) }
    var showLogo by remember { mutableStateOf( value: false) }
    var moveLogo by remember { mutableStateOf( value: false) }
    val swordAnim = animateFloatAsState(targetValue = if(showSword) 0.6f else 0f, animationSpec = tween(durationMillis = 2500))
    val logoAlphaAnim = animateFloatAsState(targetValue = if(showLogo) 1f else 0f, animationSpec = tween(durationMillis = 2000))
    val logoMoveAnim = animateDpAsState(
        targetValue = if(moveLogo) -screenHeightDp.times( other: 0.25f) else 130.dp,
        animationSpec = tween(durationMillis = 2000, easing = Easing { fraction ->
        val n1 = 7.5625f
        val d1 = 2.75f
        var newFraction = fraction

        return@Easing if (newFraction < 1f / d1) {
            n1 * newFraction * newFraction
        } else if (newFraction < 2f / d1) {
            newFraction -= 1.5f / d1
            n1 * newFraction * newFraction + 0.75f
        } else if (newFraction < 2.5f / d1) {
            newFraction -= 2.25f / d1
            n1 * newFraction * newFraction + 0.9375f
        } else {
            newFraction -= 2.625f / d1
            n1 * newFraction * newFraction + 0.984375f
        }
    }))
    LaunchedEffect( key1: true) {  this: CoroutineScope
        showSword = true
        delay( timeMillis: 2000)
        showLogo = true
        showSword = false
        delay( timeMillis: 1000)
        moveLogo = true
        delay( timeMillis: 2100)
        navController.popBackStack()
        navController.navigate( route: "landing_screen")
    }
    DrawBackground()
    DrawSword(swordAnim.value)
    Box(modifier = Modifier.fillMaxSize(), contentAlignment = Alignment.Center) {  this: BoxScope
        DrawLogo(modifier = Modifier
            .size(300.dp)
            .offset(y = logoMoveAnim.value), alpha = logoAlphaAnim.value
        )
    }
}
```

Figure 9. Loading Screen Implementation

2.5.2 Landing Screen

The landing screen is the main screen where players can access every functional part of the game. I have specifically designed two UIs in one screen, where the game mode screen is the other UI. I did this because I wanted to change only part of the UI when entering the game mode screen, while the background, settings and fruit bar remains in the same position (figure 11), and I noticed that the Navigation library doesn't support this feature.

As for the screen changing animation (figure 10), I implemented an animation to change the offset and alpha of both screens, and "showLevelMenu" is the boolean state controlling the animations. If the play button is clicked in the main screen, "showLevelMenu" becomes true, main screen gets pulled downwards and became invisible, while game mode screen pops up from the bottom, at the same time.

```
val moveUp = remember { mutableStateOf( value: false) }
val logoOffset = animateDpAsState(targetValue = if(moveUp.value) 10.dp else 0.dp, animationSpec = tween(durationMillis = 1000))
val showLevelMenu = remember { mutableStateOf( value: false) }
val landingScreenAlpha = animateFloatAsState(targetValue = if (showLevelMenu.value) 0f else 1f, animationSpec = tween( durationMilli
val levelMenuOffset = animateDpAsState(targetValue = if (showLevelMenu.value) 0.dp else screenHeightDp + 20.dp, animationSpec =
val sparkleAnim = remember { mutableStateOf( value: 1) }
var lastClickTime by remember { mutableStateOf( value: 0L) }
LaunchedEffect( key1: true) { //Logo Anim
    while(true) {
        moveUp.value = true
        delay( timeMillis: 1000)
        moveUp.value = false
        delay( timeMillis: 1000)
    }
}
LaunchedEffect( key1: true) { //Sparkles Anim
    while(true) {
        delay( timeMillis: 500)
        sparkleAnim.value = (sparkleAnim.value + 1) % 2
    }
}
```

Figure 10. all Landing Screen Animations

```
DrawBackground()
DrawTopFruit()
DrawSettingsIcon {
    navController.navigate( route: "settings_screen")
}
```

Figure 11. Landing Screen general UI items

Other than the screen changing animations, figure 10 also showed 2 other animations. They are logo animations (logo moving up and down repeatedly) and sparkles animations respectively.

The code snippet below corresponds to the main landing screen. The components are lined up from the UI's top to bottom for better readability. First, I drew a logo, then drew 2 sparkles controlled by sparkleAnim (from figure 10). These 2 sparkles will appear one at a time, and randomly in a new location, which creates an illusion that there are many sparkles. Afterwards, I drew a large dagger in the center, with a Play button and shop button underneath it.

```
//Main Landing Screen
Box(modifier = Modifier
    .fillMaxSize()
    .alpha(landingScreenAlpha.value)
    .offset(y = screenHeightDp + 20.dp - levelMenuOffset.value), contentAlignment = Alignment.Center) {
    DrawLogo(modifier = Modifier
        .align(Alignment.Center)
        .size(300.dp)
        .offset(y = -screenHeightDp.times( other: 0.25f) - logoOffset.value)
    )
    DrawSparkle(id = 0, sparkleAnim)
    DrawSparkle(id = 1, sparkleAnim)
    DrawDagger(modifier = Modifier
        .size(140.dp)
        .offset(y = screenHeightDp.times( other: 0.03f))
    )
    DrawButton(text = "PLAY", offsetY = screenHeightDp.times( other: 0.22f)) {
        if (SystemClock.elapsedRealtime() - lastClickTime > 1000L) {
            showLevelMenu.value = true
            lastClickTime = SystemClock.elapsedRealtime()
        }
    }
    DrawShopButton(offsetY = screenHeightDp.times( other: 0.35f)) {
        if (SystemClock.elapsedRealtime() - lastClickTime > 1000L) {
            navController.navigate( route: "shop_screen")
            lastClickTime = SystemClock.elapsedRealtime()
        }
    }
}
```

Figure 12. Landing Screen specific UI items

As for the game mode screen (figure 13), the word "game mode" is first drawn on top, along with 3 game mode items. Each game mode item consists of a beautifully designed button that navigates users to the game screen, along with corresponding game mode description that introduces the reward system to users. Last but not least, a return button is drawn for users to navigate back to the landing screen by simply toggling "showLevelMenu" state.

```
//Game Mode Screen
Box(modifier = Modifier
    .fillMaxSize()
    .offset(y = levelMenuOffset.value))
{  this: BoxScope
    Text(
        text = "GAME MODE",
        fontSize = 40.sp,
        color = white,
        fontFamily = myFont,
        fontWeight = FontWeight.Bold,
        modifier = Modifier
            .align(Alignment.Center)
            .offset(y = -screenHeightDp * 0.3f)
    )
    DrawGameModeItem(buttonText = "TAP", descriptionText = "Tap screen to shoot", rewardText = "x1", offsetY = -screenHeightDp * 0.18f) {
        if (SystemClock.elapsedRealtime() - lastClickTime > 500L) {
            gameDifficulty.value = 0
            gameMode.value.setTap()
            gameState.value.setWipe()
            navController.navigate( route: "game_screen")
            lastClickTime = SystemClock.elapsedRealtime()
        }
    }
    DrawGameModeItem(buttonText = "SMILE", descriptionText = "Smile to shoot", rewardText = "x2", offsetY = screenHeightDp * 0.05f) {
        if (SystemClock.elapsedRealtime() - lastClickTime > 500L) {
            if (PermissionUtil.hasPermission()) {
                gameDifficulty.value = 1
                gameMode.value.setSmile()
                gameState.value.setWipe()
                navController.navigate( route: "game_screen")
                lastClickTime = SystemClock.elapsedRealtime()
            } else PermissionUtil.requestCameraPermission()
        }
    }
    DrawGameModeItem(buttonText = "BLINK", descriptionText = "Blink both eyes to shoot", rewardText = "x3", offsetY = screenHeightDp * 0.23f) {
        if (SystemClock.elapsedRealtime() - lastClickTime > 500L) {
            if (PermissionUtil.hasPermission()) {
                gameDifficulty.value = 2
                gameMode.value.setBoth()
                gameState.value.setWipe()
                navController.navigate( route: "game_screen")
                lastClickTime = SystemClock.elapsedRealtime()
            } else PermissionUtil.requestCameraPermission()
        }
    }
    DrawReturnButton(offsetY = (-50).dp) {
        showLevelMenu.value = false
    }
}
```

Figure 13. Game Mode Screen in Landing Screen

With that said, the landing screen is complete.

2.5.3 Game Screen

The game screen is the trickiest part during development, where everything has to be responsive and smooth. As it directly determines whether players will like this game. Everything on the screen reacts perfectly as soon as the player takes any action.

```
@Composable
fun GameScreen(navController: NavHostController, gameData: GameData) {
    //UI
    var lastClickTime by remember { mutableStateOf( value: 0L) }
    DrawBackground()
    GameConsole(navController, gameData)
    //Top and bottom Bar
    DrawTopBar()
    DrawTopFruit()
    DrawReturnButton(offsetY = (-50).dp) {
        if (SystemClock.elapsedRealtime() - lastClickTime > 500L) {
            navController.popBackStack()
            lastClickTime = SystemClock.elapsedRealtime()
        }
    }
    //Camera
    DrawCamera()
    if (textHelperOption.value) DrawControllerIcons()
}
```

Figure 14. Game Screen Top Component

Since the game screen consists of too many components. I have organized it into several composable functions. First, the background is drawn, followed by the game console, top bar, return button, and the camera. This makes the code organized and easy to understand. Controller Icons are text helpers that guide players during gameplay. It is created to tell them what control/mode they are using. This feature can be turned off in the settings.

More specifically, Game Console is the largest component of the game screen, where it controls all the game logic, and is responsible for all the state changes, animations that are occurring during gameplay. I will introduce the methodology of Game Console in the upcoming sections after I introduce all the states. Besides, I will also introduce the Camera component along with Facial action controls.

2.5.4 Shop Screen

The shop screen is where I implemented the shop system.

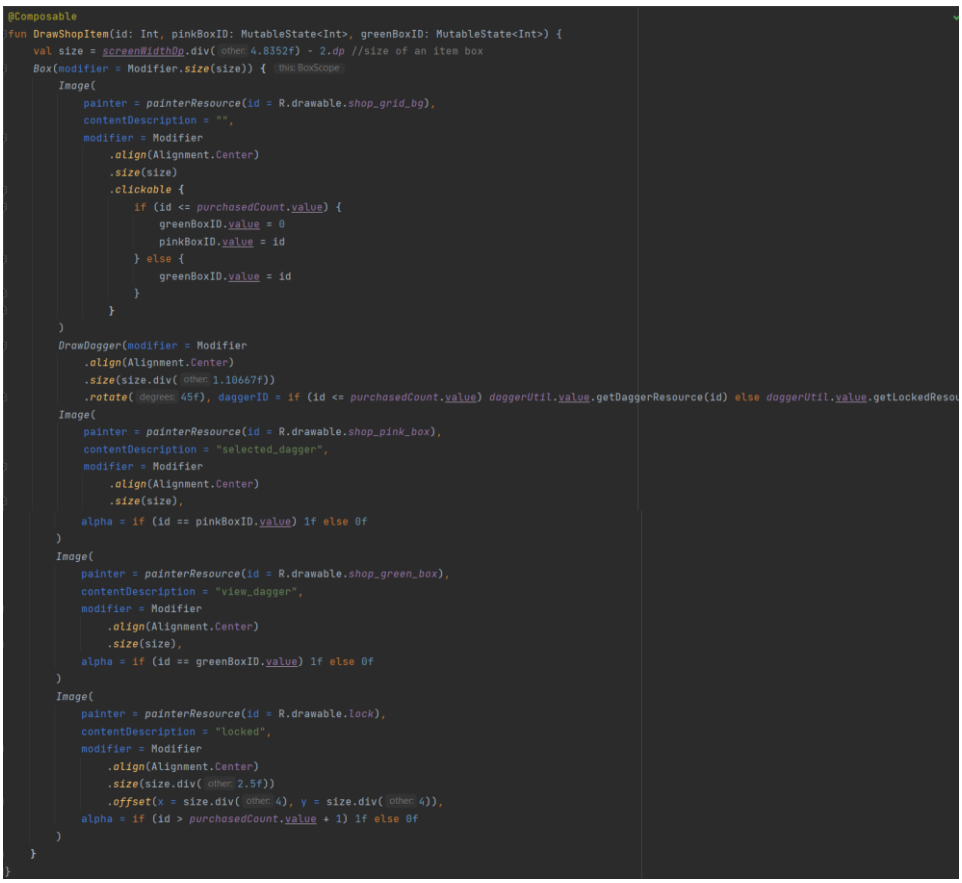
```
@Composable
fun ShopScreen(navController: NavHostController, gameData: GameData) {

    val pinkBoxID = remember { mutableStateOf(daggerUtil.value.getDaggerInUseID().value) }
    val greenBoxID = remember { mutableStateOf( value: 0) }
    val purchaseAction = remember{ mutableStateOf( value: false) }
    var lastClickTime by remember { mutableStateOf( value: 0L) }

    LaunchedEffect(purchaseAction.value) {  this: CoroutineScope
        if (purchaseAction.value) {
            gameData.saveFruitCount(fruitCount.value)
            gameData.savePurchasedCount(purchasedCount.value)
            purchaseAction.value = false
        }
    }
    LaunchedEffect(pinkBoxID.value) {  this: CoroutineScope
        daggerUtil.value.setDaggerInUseID(pinkBoxID.value)
        gameData.saveDaggerInUseID(pinkBoxID.value)
    }

    DrawBackground()
    DrawTopFruit()
    DrawReturnButton(offsetY = (-50).dp) {
        if (SystemClock.elapsedRealtime() - lastClickTime > 2000L) {
            lastClickTime = SystemClock.elapsedRealtime()
            navController.popBackStack()
        }
    }
    Box(modifier = Modifier.fillMaxSize()) {  this: BoxScope
        DrawShopLights()
        DrawDagger(modifier = Modifier
            .align(Alignment.Center)
            .offset(y = -screenHeightDp.times( other: 0.3f))
            .size(screenWidthDp.div( other: 2.935f))
            .rotate( degrees: 50f), daggerUtil.value.getDaggerResource(pinkBoxID.value))
        DrawShopBanner(-screenHeightDp.times( other: 0.1f))
        Column( //4x4 Grid
            modifier = Modifier
                .align(Alignment.Center)
                .size(screenWidthDp.div( other: 1.2088f))
                .offset(y = -screenHeightDp.times( other: 0.1f) + (screenWidthDp.times( other: 0.462296f) + 5.dp)),
            verticalArrangement = Arrangement.spacedBy(2.dp),
            horizontalAlignment = Alignment.Start
        ) {  this: ColumnScope
            repeat( times: 4) { x ->
                Row(
                    modifier = Modifier.size(screenWidthDp.div( other: 1.2088f),  height: screenWidthDp.div( other: 4.8352f) - 2.dp),
                    verticalAlignment = Alignment.CenterVertically,
                    horizontalArrangement = Arrangement.spacedBy(2.dp)
                ) {  this: RowScope
                    repeat( times: 4) { y ->
                        val id = x * 4 + y + 1
                        DrawShopItem(id, pinkBoxID, greenBoxID)
                    }
                }
            }
        }
        ShopPurchaseButton(offsetY = screenHeightDp.div( other: 2.4086f), greenBoxID) { price ->
            if (greenBoxID.value == purchasedCount.value + 1 && fruitCount.value >= price) {
                purchasedCount.value++
                fruitCount.value -= (greenBoxID.value - 1) * 15
                pinkBoxID.value = greenBoxID.value
                greenBoxID.value = 0
                purchaseAction.value = true
            }
        }
    }
}
```

Figure 15. Shop Screen Implementation

In the upper part of the code implementation, I defined several state variables named pinkBoxID, greenBoxID, and purchaseAction using mutableStateOf. They are used to keep track of selected purchased items (pink), selected unpurchased items (green), and whether a purchase has been made. A coroutine (LaunchedEffect) is used to save game data to datastore when a purchase is made, and to update the selected dagger. Another coroutine is used to update the user selected dagger whenever the pink box is changed.

Whereas for the UI, I first drew the basic elements such as background, top fruit and return button to establish a consistent look and feel. Then, I created more complex components such as shop lights that highlight the selected dagger, and a banner with a 4 by 4 grid that provides a clear and organized way for users to browse through all the available daggers. Lastly, to enable users to make purchases, a purchase button is displayed whenever a non-purchased dagger is selected.

But now you might be asking, how does the composable function DrawShopItem work? As there is only this one line of code in the above code snippet that is responsible for all the complicated logic behind the scenes. For instance, it determines the color of box to be shown for that exact dagger item, and it adds a lock to the item if the dagger is not yet available to be purchased. Below is the implementation of this composable function.



```
@Composable
fun DrawShopItem(id: Int, pinkBoxID: MutableState<Int>, greenBoxID: MutableState<Int>) {
    val size = screenWidthDp.div( other: 4.8352f) - 2.dp //size of an item box
    Box(modifier = Modifier.size(size)) { this: BoxScope
        Image(
            painter = painterResource(id = R.drawable.shop_grid_bg),
            contentDescription = "",
            modifier = Modifier
                .align(Alignment.Center)
                .size(size)
                .clickable {
                    if (id <= purchasedCount.value) {
                        greenBoxID.value = 0
                        pinkBoxID.value = id
                    } else {
                        greenBoxID.value = id
                    }
                }
        )
        DrawDagger(modifier = Modifier
            .align(Alignment.Center)
            .size(size.div( other: 1.10667f))
            .rotate( degrees: 45f), daggerID = if (id <= purchasedCount.value) daggerUtil.value.getDaggerResource(id) else daggerUtil.value.getLockedResou
        Image(
            painter = painterResource(id = R.drawable.shop_pink_box),
            contentDescription = "selected_dagger",
            modifier = Modifier
                .align(Alignment.Center)
                .size(size),
            alpha = if (id == pinkBoxID.value) 1f else 0f
        )
        Image(
            painter = painterResource(id = R.drawable.shop_green_box),
            contentDescription = "view_dagger",
            modifier = Modifier
                .align(Alignment.Center)
                .size(size),
            alpha = if (id == greenBoxID.value) 1f else 0f
        )
        Image(
            painter = painterResource(id = R.drawable.lock),
            contentDescription = "locked",
            modifier = Modifier
                .align(Alignment.Center)
                .size(size.div( other: 2.5f))
                .offset(x = size.div( other: 4), y = size.div( other: 4)),
            alpha = if (id > purchasedCount.value + 1) 1f else 0f
        )
    }
}
```

Figure 16. Draw Shop Item Implementation

As states can be passed into composable functions, we can update pink and green box ID directly from this function (the concept is similar to passing pointer reference as argument in C). A clickable image with a grey background act as the grid's background and it will change the corresponding box ID on click. Then, daggers with corresponding ID and both color boxes are drawn. The color boxes are shown only when the color ID is same to grid ID.

On the other hand, to ensure a seamless user experience, I have also made everything fully responsive and consistent. With that said, the layout, design, and functionality of the screen will remain the same regardless of the device being used.

2.5.5 Settings Screen

The settings screen is for players to modify “Game” and “Camera” settings. It starts by defining a state variable named saveToDataStore for indicating whether to save changes to datastore, where a coroutine is responsible to perform this action whenever the state is changed to true. The UI consists of a background, some decorative elements, and two sections for the two types of settings. Each section has a sticky header with a section name, and a list of settings items. Whenever the user interacts with a settings item, saveToDataStore flag is set to true and everything is saved to datastore through a coroutine.

Figure 17. Settings Screen (Part 1)

```
@OptIn(ExperimentalFoundationApi::class)
@Composable
fun Settings(navController: NavController, gameData: GameData) {

    var lastClickTime by remember { mutableStateOf( value: 0L) }
    val saveToDataStore = remember{ mutableStateOf( value: false) }

    LaunchedEffect(saveToDataStore.value) {  this: CoroutineScope
        if (saveToDataStore.value) {
            gameData.saveSettings()
            saveToDataStore.value = false
        }
    }

    DrawBackground()
    Box(modifier = Modifier.fillMaxSize()) {  this: BoxScope
        DrawShopLights()
        Image(
            painter = painterResource(id = R.drawable.settings3d),
            contentDescription = "Settings 3D icon",
            modifier = Modifier
                .align(Alignment.Center)
                .offset(y = -screenHeightDp.times( other: 0.3f))
                .size(screenWidthDp.div( other: 4.935f))
        )
        val sections = listOf("Game", "Camera")
        Box(modifier = Modifier
            .align(Alignment.Center)
            .offset(y = screenHeightDp.times( other: 0.16f) - 45.dp)
            .size( width: screenWidthDp - 10.dp,  height: screenHeightDp.times( other: 0.68f) - 90.dp)
            ) {  this: BoxScope
            LazyColumn(
                verticalArrangement = Arrangement.spacedBy(5.dp),
                contentPadding = PaddingValues(2.dp, 2.dp, 2.dp, 2.dp)
            ) {  this: LazyListScope
                sections.forEach { section ->
                    stickyHeader {  this: LazyItemScope
                        Box(modifier = Modifier
                            .size(screenWidthDp, 70.dp)
                            .background(Color( color: 0xFF181D31))) {  this: BoxScope
                            Text(
                                text = section,
                                fontSize = 30.sp,
                                color = yellow,
                                fontFamily = myFont,
                                fontWeight = FontWeight.Bold,
                                modifier = Modifier.align(Alignment.Center)
                            )
                        }
                    }
                    val sectionItems = settingsUtil.value.getSectionItems(section)
```

```
                items(items = sectionItems) { settingsItem ->
                    Box(modifier = Modifier
                        .size(screenWidthDp, 100.dp)
                        .background(color = Color( color: 0x4F181D31))) {  this: BoxScope
                        Text(
                            text = settingsItem.title,
                            fontSize = 20.sp,
                            color = white,
                            fontFamily = myFont,
                            fontWeight = FontWeight.Bold,
                            modifier = Modifier
                                .align(Alignment.CenterStart)
                                .padding(30.dp, 0.dp, 0.dp, 0.dp)
                        )
                        when (settingsItem.type) {
                            SettingsUtil.Type.Switch -> {
                                DrawSwitch(settingsItem.onText, settingsItem.offText, settingsItem.switchOn) {
                                    saveToDataStore.value = true
                                }
                            }
                            SettingsUtil.Type.Slider -> {
                                Slider(
                                    modifier = Modifier
                                        .width(screenWidthDp.times( other: 0.8f))
                                        .align(Alignment.Center)
                                        .offset(y = 25.dp),
                                    value = settingsItem.slideValue.value,
                                    onValueChange = { settingsItem.slideValue.value = it },
                                    onValueChangeFinished = { saveToDataStore.value = true },
                                          valueRange = settingsItem.min..settingsItem.max,
                                          steps = ((settingsItem.max - settingsItem.min) / settingsIte
                                    )
                                    Text(
                                        text = if (settingsItem.interval < 1) String.format("%.1f",
                                        fontSize = 20.sp,
                                        color = white,
                                        fontFamily = myFont,
                                        fontWeight = FontWeight.Bold,
                                        modifier = Modifier
                                            .align(Alignment.CenterEnd)
                                            .offset(x = (-30).dp)
                                    )
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
```

Figure 18. Settings Screen (Part 2)

Settings are mainly categorized into two types, slider, and switch. Each settings item consists of title, type, interval, min, max, onText, offText, and initial states like slideValue and switchOn.

```
data class SettingsItem(
    var title: String,
    var type: Type,
    //slider
    var interval: Float = 0f,
    var min: Float = 0f, var max: Float = 0f,
    //Switch
    var onText: String = "On", var offText: String = "Off",
) {
    var slideValue = mutableStateOf( value: 0f)
    var switchOn = mutableStateOf( value: false)
}
```

Figure 19. Settings Item

2.6 States

States are important for game control, as they are variables that can be used to hold and manage data that changes over time. States are used in defining parts of the UI that need to be recomposed whenever the data changes. In Dagger Master, there are 5 major states. They are Game State, Dagger State, Spinner State, Fruit State and Remaining Daggers State respectively.

2.6.1 Game State

In GameState.kt, there are two distinct states, one of which is game state itself, and the other is game mode. These two states will be used mainly by landing screen to set game mode, and by game screen to monitor the status of game.

```
data class GameState(private val status: Status = Status.RESET) {
    enum class Status {
        WIPE,
        RESET,
        RUNNING,
        SHOOTING,
        LEVELING,
        LOSING,
        OVER,
    }

    fun isWipe() = status == Status.WIPE
    fun isReset() = status == Status.RESET
    fun isRunning() = status == Status.RUNNING
    fun isShooting() = status == Status.SHOOTING
    fun isLeveling() = status == Status.LEVELING
    fun isLosing() = status == Status.LOSING
    fun isOver() = status == Status.OVER

    fun setWipe() { gameState.value = GameState(Status.WIPE)}
    fun setReset() { gameState.value = GameState(Status.RESET) }
    fun setRunning() { gameState.value = GameState(Status.RUNNING) }
    fun setShooting() { gameState.value = GameState(Status.SHOOTING) }
    fun setLeveling() { gameState.value = GameState(Status.LEVELING) }
    fun setLosing() { gameState.value = GameState(Status.LOSING) }
    fun setOver() { gameState.value = GameState(Status.OVER) }
}
```

Figure 20. Game State data class

There is a total of 7 game status: Wipe, Reset, Running, Shooting, Leveling, Losing, Over. Below functions are the getter and setter for each status. Below are the use cases of each status.

WIPE	Resetting everything, where all information such as level, score, etc., are reset to initial status. Used by landing screen when start game, and by game screen when player needs to restart game.
RESET	Reset only level information, where level information are randomly generated again, then reset spin speed of spinner, use another random spinner, and reset all game states (fruit, spinner, dagger state). Used whenever game is wiped or level up.
RUNNING	Normal state of game play, everything runs.
SHOOTING	Whenever control action is detected (like tapping, smiling, blinking), the state is set to SHOOTING, and shooting animations are shown. User can only perform actions when game state is not SHOOTING. After everything is complete, the game enters RUNNING again.
LEVELING	After all the remaining daggers are shot out, the game enters level up status and level up animations are shown. The game is set to RESET afterwards.
LOSING	When a dagger hits other daggers on the spinning target, the game status indicates LOSING. Where the dagger thrown falls off, and enters OVER.
OVER	Game is OVER and the score board appears. Players can decide to restart, or leave.

Table 4. Game Status and their use cases

As for Game Mode, there are a total of 3 modes: TAP, SMILE, BOTH. The figure on the next page shows the setup of GameMode data class and their corresponding getter and setter functions. Game mode is usually set by landing screen to determine which mode the game is in.

```
data class GameMode(private val mode: Mode = Mode.TAP) {

    enum class Mode {
        TAP,
        SMILE,
        BOTH
    }

    fun isTap() = mode == Mode.TAP
    fun isSmile() = mode == Mode.SMILE
    fun isBoth() = mode == Mode.BOTH

    fun setTap() { gameMode.value = GameMode(Mode.TAP) }
    fun setSmile() { gameMode.value = GameMode(Mode.SMILE) }
    fun setBoth() { gameMode.value = GameMode(Mode.BOTH) }
}
```

Figure 21. Game Mode data class

2.6.2 Dagger State

In the dagger state, there are two sub states: Dagger and Sparkle, where I would like to store the image bitmap, rotation and translation of each dagger object, and some Sparkle data. These two data classes are located inside the dagger state data class.

```
data class Dagger(val image: ImageBitmap = daggerUtil.value.getDaggerBitmap()) {
    var rotation = 0f
    var translation: Float = 700f
}

data class Sparkle(val left: Boolean, var transX: Float = 0f, var transY: Float = 0f) {
    val spreadFactor = (10 ≤ .. ≤ 30).random().toFloat()
    val fallFactor = (50 ≤ .. ≤ 70).random().toFloat()
    val fadeFactor = (85 ≤ .. ≤ 90).random().toFloat().div( other: 100)
    var scale  = 1f
}
```

Figure 22. Dagger and Sparkle data object in Dagger State

Figure 23-25 below demonstrate the implementation of dagger state. The state variables include spinSpeed, remainingDaggers, uiAlpha, hitOffset, and sparkleImg. These states are used to manage the rotation speed of daggers on target, number of remaining daggers to be thrown and transparency of UI (invisible when game level up or player loses).

```
data class DaggerState(
    val spinSpeed: MutableState<Float>,
    val remainingDaggers: MutableState<Int>,
    val uiAlpha: State<Float>,
    val hitOffset: State<Float>,
    val sparkleImg: ImageBitmap,
) {
    private val imgWidth = 70
    private val imgHeight = 140
    private val imgSize = IntSize(imgWidth, imgHeight)

    private val shootVelocity = 175f
    private val rotationMargin = 8f
    private val dropVelocity = 60f

    private var currentDagger: Dagger = Dagger()

    private val daggerList: MutableList<Dagger> = emptyList<Dagger>().toMutableList()
    private val sparkleList: MutableList<Sparkle> = emptyList<Sparkle>().toMutableList()
fun reset(): List<Float> {
    daggerList.clear()
    currentDagger = Dagger()
    val randomDagger = daggerUtil.value.getRandomDagger()
    val rotationList: MutableList<Float> = emptyList<Float>().toMutableList()
    (1 ≤ .. ≤ gameLevel.value+1).forEach{ _ ->
        val dagger = Dagger(randomDagger)
        dagger.rotation = rotationMargin * (1 ≤ until < (360/rotationMargin.toInt())).random().toFloat()
        while(dagger.rotation in rotationList) {
            dagger.rotation = rotationMargin * (1 ≤ until < (360/rotationMargin.toInt())).random().toFloat()
        }
        rotationList.add(dagger.rotation)
        daggerList.add(dagger)
    }
    return rotationList
}
fun shoot(daggerHit: () -> Unit) {
    currentDagger.translation -= shootVelocity
    if (gameState.value.isShooting() && currentDagger.translation <= 0f) { //Arrived wood
        //Check collision
        if (daggerList.any { dagger ->
                val degree = abs( x: dagger.rotation % 360)
                degree in 0f..rotationMargin || degree in (360f-rotationMargin)..360f  ^any
            }) {
            gameState.value.setLosing()
        } else {
            if (remainingDaggers.value <= 1) {
                gameState.value.setLeveling()
            } else {
                daggerList.add(currentDagger)
                currentDagger = Dagger()
                gameState.value.setRunning()
            }
            gameScore.value++
            remainingDaggers.value--
            (1 ≤ .. ≤ (2 ≤ .. ≤ 4).random()).forEach {  it: Int
                val left = (0 ≤ .. ≤ 1).random() != 1
                when (it) {
                    1 -> sparkleList.add(Sparkle( left: true,  transX: -20f,  transY: -20f))
                    2 -> sparkleList.add(Sparkle( left: false,  transX: 20f,  transY: -20f))
                    else -> sparkleList.add(Sparkle(left, (-30 ≤ .. ≤ 30).random().toFloat(), (-40 ≤ .. ≤ 0).random().toFloat()))
                }
            }
            daggerHit()
        }
    }
```

Figure 23. Dagger State data class (Part 1)

```
fun drop() {
    currentDagger.rotation += 10f
    currentDagger.translation += dropVelocity
    if (currentDagger.translation > 1200f) {
        gameState.value.setOver()
    }
}

fun draw(drawScope: DrawScope) {
    drawScope.drawCanvas()
}
```

Figure 24. Dagger State data class (Part 2)

```
private fun DrawScope.drawCanvas() {
    // Current Dagger
    drawDagger(currentDagger, currentDagger.translation,  trans2: 0f, if (uiAlpha.value == 0f && !gameState.value.isLeveling()) 0f else 1f)

    // Daggers on the wood
    daggerList.forEach{ dagger ->
        dagger.rotation += spinSpeed.value
        drawDagger(dagger,  trans1: -250f - hitOffset.value,  trans2: 260f, uiAlpha.value)
    }

    val removeList: MutableList<Sparkle> = emptyList<Sparkle>().toMutableList()
    sparkleList.forEach{ sparkle ->
        sparkle.transX += if (sparkle.left) -sparkle.spreadFactor else sparkle.spreadFactor
        sparkle.transY += sparkle.fallFactor
        sparkle.scale *= sparkle.fadeFactor
        withTransform({  this: DrawTransform
            translate(sparkle.transX, sparkle.transY)
            scale(sparkle.scale, sparkle.scale)
        }) {  this: DrawScope
            drawImage(
                image = sparkleImg,
                srcOffset = IntOffset.Zero,
                srcSize = IntSize( width: 23,  height: 23),
                dstOffset = IntOffset( x: midX().toInt() - 23,  y: midY().toInt() - 23),
                dstSize = IntSize( width: 46,  height: 46),
            )
        }
        if (sparkle.scale < 0.1f) removeList.add(sparkle)
    }
    removeList.forEach{ sparkle -> sparkleList.remove(sparkle) }
}

private fun DrawScope.drawDagger(dagger: Dagger, trans1: Float, trans2: Float, alpha: Float) {
    withTransform({  this: DrawTransform
        translate( left: 0f, trans1) //mark to center
        rotate(dagger.rotation)
        translate( left: 0f, trans2) //provide offset
    }) {  this: DrawScope
        drawImage(
            image = dagger.image,
            srcOffset = IntOffset.Zero,
            srcSize = imgSize,
            dstOffset = IntOffset( x: midX().toInt() - imgWidth,  y: midY().toInt() - imgHeight),
            dstSize = imgSize * 2,
            alpha = alpha
        )
    }
}
```

Figure 25. Dagger State data class (Part 3)

Dagger State is responsible for keeping track of all the daggers' position and rotation, and to draw everything out precisely. There are two types of daggers, one controlled by the player, and the others on the spinner. Daggers on the spinner only need to spin, and the dagger held by the player shoots forward until it hits the spinner. Hence, I have implemented a dagger list to collect all the daggers that reach the spinner target.

Besides, Dagger State class also includes functions such as reset, shoot, drop, and draw. They are used to reset the game, shoot daggers towards spinner, drop daggers when game loses, and most importantly, draw everything on to the canvas.

In the draw() function, it first draws the current dagger to be thrown according to its translation, then draws all the daggers on the spinning target. Finally, it draws the sparkles that will appear whenever a dagger hits the target. The sparkles will be removed when it runs out of bounds. With that said, the sparkles animations are done, and the dagger state is ready to be used.

2.6.3 Spinner State

Spinner State is relatively easier to implement than Dagger State. As there is only one spinner in the middle of the UI. Just like the dagger state, Spinner State has state variables image, spinSpeed, uiAlpha, hitAlpha and hitOffset. Image and cover correspond to the image bitmap of the spinner and its cover respectively.

I want the spinner to spin according to spin speed and have a cool hit animation whenever a dagger hits it. The spinner will be covered by a white cover, and offset a bit when it is being hit, for 50 milliseconds.

The Spinner State data class includes 2 functions: reset and draw. They are used to reset the game, and draw the spinner on to the canvas respectively. The spinner will update its rotation by spinSpeed every time the draw() function is called.

```
data class SpinnerState(
    val image: MutableState<ImageBitmap>,
    val cover: ImageBitmap,
    var spinSpeed: MutableState<Float>,
    val uiAlpha: State<Float>,
    val hitAlpha: State<Float>,
    val hitOffset: State<Float>
) {
    private val imgWidth = 220
    private val imgHeight = 220
    private val imgSize = IntSize(imgWidth, imgHeight)

    private var currentRotation = 0f

    fun reset() {
        currentRotation = 0f
    }

    fun draw(drawScope: DrawScope) {
        drawScope.drawCanvas()
    }

    private fun DrawScope.drawCanvas() {
        currentRotation += spinSpeed.value
        withTransform({ this: DrawTransform
            translate( left: 0f,  top: -250f - hitOffset.value)
            rotate(currentRotation)
        }) {  this: DrawScope
            drawImage(
                image = image.value,
                srcOffset = IntOffset.Zero,
                srcSize = imgSize,
                dstOffset = IntOffset( x: midX().toInt() - imgWidth,  y: midY().toInt() - imgHeight),
                dstSize = imgSize * 2,
                alpha = uiAlpha.value
            )
            drawImage(
                image = cover,
                srcOffset = IntOffset.Zero,
                srcSize = imgSize,
                dstOffset = IntOffset( x: midX().toInt() - imgWidth,  y: midY().toInt() - imgHeight),
                dstSize = imgSize * 2,
                alpha = hitAlpha.value
            )
        }
    }
}
```

Figure 26. Spinner State data class

2.6.4 Fruit State

During gameplay, there are fruits rotating around the spinner target, and when a dagger hits the fruit, the fruit will crack and circulate around the spinner for more than 160 degrees, then run to the score on the top bar. First, a fruit data class is defined inside the fruit state data class.

```
data class Fruit(var rotation: Float, var updateGained: Boolean = true) {
    var transX: Float = 0f
    var transY: Float = 0f
    var scale: Float = 1f
}
```

Figure 27. Fruit data class

```
data class FruitState(
    val image: ImageBitmap,
    val image_crack: ImageBitmap,
    val spinSpeed: MutableState<Float>,
    val uiAlpha: State<Float>,
    val hitOffset: State<Float>,
    val gameData: GameData
) {
    private val imgWidth = 56
    private val imgHeight = 56
    private val imgSize = IntSize(imgWidth, imgHeight)
    private val rotationMargin = 20f
    private val fruitList: MutableList<Fruit> = emptyList<Fruit>().toMutableList()
    private val completedFruitList: MutableList<Fruit> = emptyList<Fruit>().toMutableList()

    fun reset(rotationList: List<Float>) {
        fruitList.clear()
        // 40% chance 0, 20% chance 1 or 2, 10% chance 3 or 4
        var noOfFruit = (0 ≤ until < 10).random()
        noOfFruit = when (noOfFruit) {
            in (0 ≤ .. ≤ 3) -> return
            in (4 ≤ .. ≤ 5) -> 1
            in (6 ≤ .. ≤ 7) -> 2
            8 -> 3
            else -> 4
        }
    (1 ≤ .. ≤ noOfFruit).forEach { _ ->
        var rotation: Float
        run { while (true) {
            rotation = (20 ≤ .. ≤ 339).random().toFloat()
            var crashed = false
            rotationList.forEach{ if (!crashed && rotation in (it - rotationMargin .. it + rotationMargin)) crashed = true }
            fruitList.forEach{ if (!crashed && rotation in (it.rotation - rotationMargin .. it.rotation + rotationMargin)) crashed = true }
            if (!crashed) return@run
        } }
        val fruit = Fruit(rotation)
        fruitList.add(fruit)
    }
  }

  fun hit() { //check collision of fruit
      val removeList: MutableList<Fruit> = emptyList<Fruit>().toMutableList()
      fruitList.forEach{ fruit ->
          val degree = abs( x fruit.rotation % 360)
          if (degree in 0f..rotationMargin || degree in (360f-rotationMargin)..360f) {
              removeList.add(fruit) //so that multiple fruit being hit can be handled correctly
              if (fruit.rotation < 0) fruit.rotation = 360 - abs( x fruit.rotation % 360)
              else fruit.rotation = abs( x fruit.rotation % 360)
              completedFruitList.add(fruit)
          }
      }
      removeList.forEach{ fruit -> fruitList.remove(fruit) }
  }
```

Figure 28. Fruit State data class (Part 1)

Figure 28 included initializations of the data class, with fruitList storing all the fruits that are on the spinner target, and completedFruitList storing all the fruits that are hit by dagger. This figure also showed how I included reset and hit function, which are used to reset the game and handle fruits that are being hit. From reset(), you can see that fruits are randomly generated with different rotations. If the fruit is initialized in a place that collides with the daggers, it will randomly generate a new rotation for the fruit again.

Figure 29. Fruit State data class (Part 2)

In figure 29, addBonusFruit and draw functions are introduced. Add bonus fruit is called during level up in smiling and blinking mode, due to the reward system. While the draw function updates the fruits positions and renders again onto the Canvas every time it's called.

2.6.5 Remaining Daggers State

Last but not least, the remaining daggers state is an indicator that will be placed in the left bottom corner of the game screen to notify players of the number of remaining daggers for a specific level. It consists of only one function named draw(), and it is responsible for drawing the indicator images.

```
data class RemainingDaggerState(val image: ImageBitmap, val remainingDaggers: MutableState<Int>, val uiAlpha: State<Float>) {
    private val imgWidth = 30
    private val imgHeight = 30
    private val imgSize = IntSize(imgWidth, imgHeight)

    fun draw(drawScope: DrawScope) {
        drawScope.drawCanvas()
    }

    private fun DrawScope.drawCanvas() {
        (1 ≤ .. ≤ remainingDaggers.value).forEach{ it: Int
            drawImage(
                image = image,
                srcOffset = IntOffset.Zero,
                srcSize = imgSize,
                dstOffset = IntOffset(45.dp.toPx().toInt(), y: (size.height * 0.88).toInt() - it * 80),
                dstSize = imgSize * 3,
                alpha = uiAlpha.value
            )
        }
    }
}
```

Figure 30. Remaining Daggers State data class

2.7 Utils

There are mainly 7 utils in Dagger Master and each of them solves a specific problem for dagger master. They correspond to Status Bar Util, Orientation Util, Spinner Util, Dagger Util, Permission Util, Settings Util, and Game Util and Data Store. By the end of the utils section, I will talk about Utils Setup, where I demonstrate how I implemented all the utils.

2.7.1 Status Bar Util

By default, the status bar of the phone/emulator will be showing on the top. I think it is very ugly and hugely affects the user experience. Hence, I have decided to make it transparent.

```
object StatusBarUtil {
    fun transparentStatusBar(activity: Activity) {
        with(activity) {  this: Activity
            window.clearFlags(WindowManager.LayoutParams.FLAG_TRANSLUCENT_STATUS)
            window.addFlags(WindowManager.LayoutParams.FLAG_DRAWS_SYSTEM_BAR_BACKGROUNDS)
            val option = View.SYSTEM_UI_FLAG_LAYOUT_STABLE or View.SYSTEM_UI_FLAG_LAYOUT_FULLSCREEN
            val vis = window.decorView.systemUiVisibility
            window.decorView.systemUiVisibility = option or vis
            window.statusBarColor = Color.TRANSPARENT
        }
    }
}
```

Figure 31. Status Bar Util

2.7.2 Orientation Util

During development, I discovered that changing phone's orientation can lead to state loss. It could be a bug in Android phones, but I will just lock it, nevertheless. To do it, simply launch a disposable effect, and lock screen orientation based on current activity.

Figure 32. Orientation Util

2.7.3 Spinner Util

The spinner util is responsible for initializing and managing a list of spinner images. The Init function is a composable function that adds 16 image bitmaps to the spinnerList. The getRandomSpinner function is used by the game screen to get a random spinner to be used on every level.

```
data class SpinnerUtil(val totalSpinners: Int = 16) {
    private val spinnerList: MutableList<ImageBitmap> = emptyList<ImageBitmap>().toMutableList()

    @Composable
    fun Init() {
        spinnerList.clear()
        (1 ≤ .. ≤ totalSpinners).forEach{ it: Int
            spinnerList.add(ImageBitmap.imageResource(id = getSpinner(it)))
        }
    }
    fun getRandomSpinner(): ImageBitmap = spinnerList[(0 ≤ until < totalSpinners).random()]

    private fun getSpinner(id: Int): Int {
        return when(id) {
            1 -> R.drawable.spinner1
            2 -> R.drawable.spinner2
            3 -> R.drawable.spinner3
            4 -> R.drawable.spinner4
            5 -> R.drawable.spinner5
            6 -> R.drawable.spinner6
            7 -> R.drawable.spinner7
            8 -> R.drawable.spinner8
            9 -> R.drawable.spinner9
            10 -> R.drawable.spinner10
            11 -> R.drawable.spinner11
            12 -> R.drawable.spinner12
            13 -> R.drawable.spinner13
            14 -> R.drawable.spinner14
            15 -> R.drawable.spinner15
            16 -> R.drawable.spinner16
            else -> R.drawable.spinner0
        }
    }
}
```

Figure 33. Spinner Util

2.7.4 Dagger Util

Just like the spinner util, the Dagger util is utilized to initialize and manage a list of dagger images. Since image bitmaps can only be accessed in composable functions, I have created the Init function to add all the images into dagger and locked list. Locked images are the

images that will be shown to players in the shop screen when they haven't bought the daggers. While normal images will be used by landing screen, shop screen, and game screen.

```
data class DaggerUtil(val totalDaggers: Int = 16) {
    private val daggerList: MutableList<ImageBitmap> = emptyList<ImageBitmap>().toMutableList()
    private val lockedList: MutableList<ImageBitmap> = emptyList<ImageBitmap>().toMutableList()
    private val daggerInUseID: MutableState<Int> = mutableStateOf( value: 0)
    @Composable
    fun Init(daggerToUse: Int) {
        daggerList.clear()
        lockedList.clear()
        daggerInUseID.value = daggerToUse
        (1 ≤ .. ≤ totalDaggers).forEach{ daggerList.add(ImageBitmap.imageResource(id = getDagger(it))) }
        (1 ≤ .. ≤ totalDaggers).forEach{ lockedList.add(ImageBitmap.imageResource(id = getLocked(it)))}
    }
    fun setDaggerInUseID(daggerID: Int) { daggerInUseID.value = daggerID }
    fun getDaggerInUseID() = daggerInUseID
    fun getDaggerResource(daggerID: Int = daggerInUseID.value): Int = getDagger(daggerID)
    fun getLockedResource(daggerID: Int) = getLocked(daggerID)
    fun getDaggerBitmap(daggerID: Int = daggerInUseID.value): ImageBitmap = daggerList[daggerID - 1]
    fun getRandomDagger() = daggerList[(0 ≤ until < totalDaggers).random()]
   private fun getDagger(id: Int): Int {
        return when(id) {
            1 -> R.drawable.d1
            2 -> R.drawable.d2
            3 -> R.drawable.d3
            4 -> R.drawable.d4
            5 -> R.drawable.d5
            6 -> R.drawable.d6
            7 -> R.drawable.d7
            8 -> R.drawable.d8
            9 -> R.drawable.d9
            10 -> R.drawable.d10
            11 -> R.drawable.d11
            12 -> R.drawable.d12
            13 -> R.drawable.d13
            14 -> R.drawable.d14
            15 -> R.drawable.d15
            16 -> R.drawable.d16
            else -> R.drawable.d1
        }
    }
    private fun getLocked(id: Int): Int {
        return when(id) {
            2 -> R.drawable.d2s
            3 -> R.drawable.d3s
            4 -> R.drawable.d4s
            5 -> R.drawable.d5s
            6 -> R.drawable.d6s
            7 -> R.drawable.d7s
            8 -> R.drawable.d8s
            9 -> R.drawable.d9s
            10 -> R.drawable.d10s
            11 -> R.drawable.d11s
            12 -> R.drawable.d12s
            13 -> R.drawable.d13s
            14 -> R.drawable.d14s
            15 -> R.drawable.d15s
            16 -> R.drawable.d16s
            else -> R.drawable.d1
        }
    }
}
```

Figure 34. Dagger Util

2.7.5 Permission Util

Permission Util is used to query for user's approval on Camera permissions. During the start of Dagger Master, permission util will be utilized to check whether Dagger Master has camera permissions. If not, Dagger Master will request for it with the requestCameraPermission function.

```
object PermissionUtil {

    private val hasCameraPermission = mutableStateOf( value: false)
    private lateinit var launcher: ManagedActivityResultLauncher<String, Boolean>

    fun hasPermission() = hasCameraPermission.value

    @Composable
    fun Initialize(context: Context) {
        hasCameraPermission.value = ContextCompat.checkSelfPermission(context, android.Manifest.permission.CAMERA) == PackageManager.PERMISSION_GRANT
        launcher = rememberLauncherForActivityResult(ActivityResultContracts.RequestPermission()) { granted ->
            hasCameraPermission.value = granted
        }
        LaunchedEffect( key1: true) {  this: CoroutineScope
            if (!hasCameraPermission.value) launcher.launch(android.Manifest.permission.CAMERA)
        }
    }

    fun requestCameraPermission() {
        Log.d( tag: "game",  msg: "request camera perm")
        launcher.launch(android.Manifest.permission.CAMERA)
    }
}
```

Figure 35. Permission Util

2.7.6 Settings Util

Settings util is a data class that aids settings screen. From figure 36, Settings util consists of a list of camera and game settings. As I have mentioned in the settings screen, all settings can be categorized into an Enum class named Type with either Slider or Switch, with a Settings Item data class storing all the information of each setting.

```
data class SettingsUtil(var initialized: Boolean = false) {
    private val cameraSettings: MutableList<SettingsItem> = emptyList<SettingsItem>().toMutableList()
    private val gameSettings: MutableList<SettingsItem> = emptyList<SettingsItem>().toMutableList()

    enum class Type {
        Slider,
        Switch
    }

    data class SettingsItem(
        var title: String,
        var type: Type,
        //slider
        var interval: Float = 0f,
        var min: Float = 0f, var max: Float = 0f,
        //Switch
        var onText: String = "On", var offText: String = "Off",
    ) {
        var slideValue = mutableStateOf( value: 0f)
        var switchOn = mutableStateOf( value: false)
    }
    fun getSectionItems(section: String): List<SettingsItem> {
        return when(section) {
            "Camera" -> cameraSettings
            "Game" -> gameSettings
            else -> emptyList()
        }
    }
```

Figure 36. Settings Util (Part 1)

Figure 37 demonstrated how I initialize all the settings. The methodology is quite simple, where I manually input all the required fields in settings item class, then add it to the respective list. By the end, the boolean variable “initialized” is set to true.

```
    fun initializeSettings() {
        cameraSettings.clear()
        gameSettings.clear()
        var s = SettingsItem(
            title = "Show Camera in Game",
            type = Type.Switch,
            onText = "Yes", offText = "No"
        )
        s.switchOn = showCameraSettings
        cameraSettings.add(s)
        s = SettingsItem(
            title = "Rotation",
            type = Type.Slider,
            interval = 90f, min = 0f, max = 360f
        )
        s.slideValue = cameraRotationSettings
        cameraSettings.add(s)
        s = SettingsItem(
            title = "Outer Scale",
            type = Type.Slider,
            interval = 0.1f, min = 0.5f, max = 1.5f
        )
        s.slideValue = cameraOutScaleSettings
        cameraSettings.add(s)
        s = SettingsItem(
            title = "Inner Scale",
            type = Type.Slider,
            interval = 0.1f, min = 0.5f, max = 1.5f
        )
        s.slideValue = cameraInScaleSettings
        cameraSettings.add(s)
        s = SettingsItem(
            title = "Facing",
            type = Type.Switch,
            onText = "Front", offText = "Back"
        )
        s.switchOn = lensFacing
        cameraSettings.add(s)
        s = SettingsItem(
            title = "Left Eye Sensitivity",
            type = Type.Slider,
            interval = 0.1f, min = 0f, max = 1f
        )
        s.slideValue = faceLeftSensitivity
        gameSettings.add(s)
        s = SettingsItem(
            title = "Right Eye Sensitivity",
            type = Type.Slider,
            interval = 0.1f, min = 0f, max = 1f
        )
        s.slideValue = faceRightSensitivity
        gameSettings.add(s)
         s = SettingsItem(
             title = "Smiling Sensitivity",
             type = Type.Slider,
             interval = 0.1f, min = 0f, max = 1f
         )
         s.slideValue = faceSmileSensitivity
         gameSettings.add(s)
         s = SettingsItem(
             title = "In Game Text Helper",
             type = Type.Switch,
             onText = "Enabled", offText = "Disabled"
         )
         s.switchOn = textHelperOption
         gameSettings.add(s)
         initialized = true
    }
}
```

Figure 37. Settings Util (Part 2)

Camera Settings		
Show Camera in Game	Switch	onText = Yes, offText = No
Rotation	Slider	Interval = 90f, min = 0f, max = 360f
Outer Scale	Slider	Interval = 0.1f, min = 0.5f, max = 1.5f
Inner Scale	Slider	Interval = 0.1f, min = 0.5f, max = 1.5f
Facing	Switch	onText = Front, offText = Back
Game Settings		
Left Eye Sensitivity	Slider	Interval = 0.1f, min = 0f, max = 1f
Right Eye Sensitivity	Slider	Interval = 0.1f, min = 0f, max = 1f
Smiling Sensitivity	Slider	Interval = 0.1f, min = 0f, max = 1f
In Game Text Helper	Switch	onText = Enabled, offText = Disabled

Table 5. Table showing all Settings

This table shows all the settings along with their type and constraints.

2.7.7 Game Util and Data Store

Datastore requires quite a lot of setups. In this section, I am going to demonstrate how I set up Datastore. I have created a class named GameData and it is responsible for managing and storing game data using Datastore. It contains a companion object, which defines the keys used to store and retrieve game data. Below it consists of a lot of get and save functions that act as getter and setter functions for each key. Note that the save functions are suspend functions due to the fact that it requires coroutine scopes to update the data to local storage.

```
class GameData(private val context: Context) {
    companion object{
        private val Context.dataStore: DataStore<Preferences> by preferencesDataStore( name: "data_storage")
        val MAX_SCORE = intPreferencesKey( name: "max_score")
        val FRUIT_COUNT = intPreferencesKey( name: "fruit_count")
        val DAGGER_IN_USE_ID = intPreferencesKey( name: "dagger_in_use_ID")
        val PURCHASED_COUNT = intPreferencesKey( name: "purchased_count")
        val SHOW_CAMERA = booleanPreferencesKey( name: "show_camera")
        val CAMERA_ROTATION = floatPreferencesKey( name: "camera_rotation")
        val CAMERA_OUT_SCALE = floatPreferencesKey( name: "camera_out_scale")
        val CAMERA_IN_SCALE = floatPreferencesKey( name: "camera_in_scale")
        val CAMERA_FACING = booleanPreferencesKey( name: "camera_facing")
        val FACE_LEFT = floatPreferencesKey( name: "face_left")
        val FACE_RIGHT = floatPreferencesKey( name: "face_right")
        val FACE_SMILE = floatPreferencesKey( name: "face_smile")
        val TEXT_HELPER = booleanPreferencesKey( name: "text_helper_options")
    }
    val getMaxScore: Flow<Int> = context.dataStore.data.map { it[MAX_SCORE] ?: 0 }
    val getFruitCount: Flow<Int> = context.dataStore.data.map { it[FRUIT_COUNT] ?: 0 }
    val getDaggerInUseID: Flow<Int> = context.dataStore.data.map { it[DAGGER_IN_USE_ID] ?: 1 }
    val getPurchasedCount: Flow<Int> = context.dataStore.data.map { it[PURCHASED_COUNT] ?: 1 }
    val getShowCamera: Flow<Boolean> = context.dataStore.data.map { it[SHOW_CAMERA] ?: true }
    val getCameraRotation: Flow<Float> = context.dataStore.data.map { it[CAMERA_ROTATION] ?: 0f }
    val getCameraOutScale: Flow<Float> = context.dataStore.data.map { it[CAMERA_OUT_SCALE] ?: 1f }
    val getCameraInScale: Flow<Float> = context.dataStore.data.map { it[CAMERA_IN_SCALE] ?: 1f }
    val getCameraFacing: Flow<Boolean> = context.dataStore.data.map { it[CAMERA_FACING] ?: false }
    val getFaceLeft: Flow<Float> = context.dataStore.data.map { it[FACE_LEFT] ?: 0.5f }
    val getFaceRight: Flow<Float> = context.dataStore.data.map { it[FACE_RIGHT] ?: 0.5f }
    val getFaceSmile: Flow<Float> = context.dataStore.data.map { it[FACE_SMILE] ?: 0.5f }
    val getTextHelper: Flow<Boolean> = context.dataStore.data.map { it[TEXT_HELPER] ?: true }
    suspend fun saveSettings() { context.dataStore.edit {  it: MutablePreferences
        it[SHOW_CAMERA] = showCameraSettings.value
        it[CAMERA_ROTATION] = cameraRotationSettings.value
        it[CAMERA_OUT_SCALE] = cameraOutScaleSettings.value
        it[CAMERA_IN_SCALE] = cameraInScaleSettings.value
        it[CAMERA_FACING] = lensFacing.value
        it[FACE_LEFT] = faceLeftSensitivity.value
        it[FACE_RIGHT] = faceRightSensitivity.value
        it[FACE_SMILE] = faceSmileSensitivity.value
        it[TEXT_HELPER] = textHelperOption.value
    }}
    suspend fun saveMaxScore(score: Int) { context.dataStore.edit { it[MAX_SCORE] = score } }
    suspend fun saveFruitCount(count: Int) { context.dataStore.edit { it[FRUIT_COUNT] = count }}
    suspend fun saveDaggerInUseID(id: Int) { context.dataStore.edit { it[DAGGER_IN_USE_ID] = id }}
    suspend fun savePurchasedCount(count: Int) { context.dataStore.edit { it[PURCHASED_COUNT] = count }}
}
```

Figure 38. Game Data class for Data Store

```
object GameUtil {
    @Composable
    fun loadData(context: Context, gameMode: String): GameData {
        val gameData = GameData(context)
        LaunchedEffect( key1: true) {  this: CoroutineScope
            when (gameMode) {
                "god" -> {
                    gameData.saveFruitCount( count: 999)
                    gameData.savePurchasedCount( count: 16)
                    gameData.saveDaggerInUseID( id: 16)
                }
                "reset" -> {
                    gameData.saveFruitCount( count: 0)
                    gameData.saveMaxScore( score: 0)
                    gameData.savePurchasedCount( count: 1)
                    gameData.saveDaggerInUseID( id: 1)
                }
                "rich" -> {
                    gameData.saveFruitCount( count: 1000)
                }
            }
        }
        maxScore.value = gameData.getMaxScore.collectAsState(initial = 0).value
        fruitCount.value = gameData.getFruitCount.collectAsState(initial = 0).value
        purchasedCount.value = gameData.getPurchasedCount.collectAsState(initial = 1).value
        showCameraSettings.value = gameData.getShowCamera.collectAsState(initial = true).value
        cameraRotationSettings.value = gameData.getCameraRotation.collectAsState(initial = 0f).value
        cameraOutScaleSettings.value = gameData.getCameraOutScale.collectAsState(initial = 1f).value
        cameraInScaleSettings.value = gameData.getCameraInScale.collectAsState(initial = 1f).value
        lensFacing.value = gameData.getCameraFacing.collectAsState(initial = false).value
        faceLeftSensitivity.value = gameData.getFaceLeft.collectAsState(initial = 0.5f).value
        faceRightSensitivity.value = gameData.getFaceRight.collectAsState(initial = 0.5f).value
        faceSmileSensitivity.value = gameData.getFaceSmile.collectAsState(initial = 0.5f).value
        textHelperOption.value = gameData.getTextHelper.collectAsState(initial = true).value
        daggerUtil.value.Init(gameData.getDaggerInUseID.collectAsState(initial = 1).value)
        spinnerUtil.value.Init()
        return gameData
    }

    fun updateLevelInfo(randomSpeed: MutableState<Boolean>, clockwise: MutableState<Boolean>, spinSpe
        val minusFactor = when (gameDifficulty.value) {
            1 -> 1
            2 -> 1
            3 -> 2
            else -> 0
        }
        val level = gameLevel.value
        clockwise.value = (0 ≤ .. ≤ 1).random() == 1
        randomSpeed.value = level >= 3
        spinSpeed.value = (2 ≤ .. ≤ 4).random().toFloat()
        minSpeed.value = ((2+level/4) ≤ .. ≤ (3+level/4)).random()
        maxSpeed.value = ((4+level/4) ≤ .. ≤ (6+level/4)).random()
        remainingDaggers.value = level / 2 + (5 ≤ .. ≤ 7).random() - minusFactor
    }
}
```

Figure 39. Game Util

Game util consists of 2 major uses. One of which is to load all the data that are stored in datastore into a bunch of global variables, the other is to update difficulty for each level.

The loadData() function is a composable function that takes a context object and game mode as parameters, and returns a GameData object. The game mode denoted here is not same with the one in game state. It is a feature for developers like me to overwrite game data, while players will be playing in normal mode.

On the other hand, the updateLevelInfo() function is a difficulty system that I implemented. This system randomly decides the rotational direction, spin speed, minimum and maximum spin speed, as well as the number of remaining daggers.

2.7.8 Utils Setup

At the beginning of the methodology, I have illustrated how to set up Navigation in Main Activity. In this section, I am going to demonstrate how to set up all the utils. First, before entering the setContent function, the status bar had to be set transparent. The next thing to do is initialize local context and configuration to current activity, and obtain the screen size for further calculations. Afterwards, I utilized orientation util, permission util and settings util to lock screen orientation, request for permission, and initialize settings, respectively. Last but not least, game util is applied with “normal” game mode to load all data to global variables.

```
class MainActivity : ComponentActivity() {

    @OptIn(ExperimentalAnimationApi::class)
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        StatusBarUtil.transparentStatusBar( activity: this)
        setContent {
            // Initialization
            val context = LocalContext.current
            val configuration = LocalConfiguration.current
            screenHeightDp = configuration.screenHeightDp.dp
            screenWidthDp = configuration.screenWidthDp.dp
            screenHeightInt = with(LocalDensity.current) { screenHeightDp.toPx().toInt() }
            screenWidthInt = with(LocalDensity.current) { screenWidthDp.toPx().toInt() }
            OrientationUtil.LockScreenOrientation(orientation = ActivityInfo.SCREEN_ORIENTATION_PORTRAIT)
            PermissionUtil.Initialize(context)
            settingsUtil.value.initializeSettings()

            // available gameModes, "god" "reset" "normal" "rich"
            val gameData = GameUtil.loadData(context, gameMode = "normal")
```

Figure 40. Utils Setup

2.8 Global Variables

Although setting global variables is not the best practice of Software Engineers, I have made use of its nature to make Dagger Master's development efficient and convenient.

```
//Game Data (temporary)
val gameState: MutableState<GameState> = mutableStateOf(GameState())
val gameMode: MutableState<GameMode> = mutableStateOf(GameMode())
val gameDifficulty: MutableState<Int> = mutableStateOf( value: 0)
val gameScore: MutableState<Int> = mutableStateOf( value: 0)
val gameLevel: MutableState<Int> = mutableStateOf( value: 1)
val cameraReady: MutableState<Boolean> = mutableStateOf( value: false)
val showCamera: MutableState<Boolean> = mutableStateOf( value: false)
val leftP : MutableState<Float> = mutableStateOf( value: 1f)
val rightP : MutableState<Float> = mutableStateOf( value: 1f)
val smileP : MutableState<Float> = mutableStateOf( value: 0f)
val showTopScore : MutableState<Boolean> = mutableStateOf( value: false)
val showScoreBoardButton : MutableState<Boolean> = mutableStateOf( value: false)

//Game Data (Permanent)
val maxScore : MutableState<Int> = mutableStateOf( value: 0)
val fruitCount : MutableState<Int> = mutableStateOf( value: 0)
val fruitGained : MutableState<Int> = mutableStateOf( value: 0)
val purchasedCount : MutableState<Int> = mutableStateOf( value: 1)
val showCameraSettings: MutableState<Boolean> = mutableStateOf( value: true)
val cameraRotationSettings: MutableState<Float> = mutableStateOf( value: 0f)
val cameraOutScaleSettings: MutableState<Float> = mutableStateOf( value: 1f)
val cameraInScaleSettings: MutableState<Float> = mutableStateOf( value: 1f)
val lensFacing : MutableState<Boolean> = mutableStateOf( value: false)
val faceLeftSensitivity: MutableState<Float> = mutableStateOf( value: 0.2f)
val faceRightSensitivity: MutableState<Float> = mutableStateOf( value: 0.2f)
val faceSmileSensitivity: MutableState<Float> = mutableStateOf( value: 0.5f)
val textHelperOption: MutableState<Boolean> = mutableStateOf( value: false)
//Utils
val spinnerUtil = mutableStateOf(SpinnerUtil())
val daggerUtil = mutableStateOf(DaggerUtil())
val settingsUtil = mutableStateOf(SettingsUtil())

//Variables
val mutex = Mutex()
var screenHeightDp: Dp = 0.dp
var screenWidthDp: Dp = 0.dp
var screenHeightInt: Int = 0
var screenWidthInt: Int = 0

val myFont = FontFamily(Font(R.font.nineteenth))
val white = Color( color: 0xFFD8D8D8)
val yellow = Color( color: 0xFFFFB26B)
```

Figure 41. Global Variables

These global variables are utilized everywhere throughout the game.

2.9 Game Console

After I introduced all the methodologies for states, utils and variables. It's time for Game Console. In figure 42, I have initialized animations, game settings, as well as game resources.

```
@Composable
fun GameConsole(navController: NavHostController, gameData: GameData) {
    //Animations
    val animation = remember{ Animatable(initialValue = 0f) }
    val uiAlpha = animateFloatAsState(targetValue = if(gameState.value.isOver() || gameState.value.isLeveling
    val uiAlpha2 = animateFloatAsState(targetValue = if(gameState.value.isOver() || gameState.value.isLevelin
    val daggerHit = remember{ mutableStateOf( value: false) }
    val hitOffset = animateFloatAsState(targetValue = if (daggerHit.value) 20f else 0f, animationSpec = tween
    val hitAlpha = animateFloatAsState(targetValue = if (daggerHit.value) 0.6f else 0f, animationSpec = tween

    //Game Settings
    val randomSpeed = remember{ mutableStateOf( value: false) }
    val clockwise = remember{ mutableStateOf( value: false) }
    val spinSpeed = remember{ mutableStateOf( value: 0f) }
    val minSpeed = remember{ mutableStateOf( value: 0) }
    val maxSpeed = remember{ mutableStateOf( value: 0) }
    val remainingDaggers = remember{ mutableStateOf( value: 0) }

    //Game Resources
    val spinner = remember{ mutableStateOf(spinnerUtil.value.getRandomSpinner()) }
    val cover = ImageBitmap.imageResource(id = R.drawable.spinner0)
    val remainingDagger = ImageBitmap.imageResource(id = R.drawable.remaining_dagger)
    val fruit = ImageBitmap.imageResource(id = R.drawable.fruit)
    val fruitCrack = ImageBitmap.imageResource(id = R.drawable.fruit_crack)
    val sparkle = ImageBitmap.imageResource(id = R.drawable.sparkle)
    val daggerState = remember { DaggerState(spinSpeed, remainingDaggers, uiAlpha2, hitOffset, sparkle) }
    val spinnerState = remember { SpinnerState(spinner, cover, spinSpeed, uiAlpha, hitAlpha, hitOffset) }
    val remainingDaggerState = remember { RemainingDaggerState(remainingDagger, remainingDaggers, uiAlpha) }
    val fruitState = remember { FruitState(fruit, fruitCrack, spinSpeed, uiAlpha, hitOffset, gameData)}
```

Figure 42. Game Console Implementation (Part 1)

Within the function, various animations are created, where they are then used to control the opacity and position of various UI components based on the state of gameplay. In the game resources section (figure 42), image bitmap of spinner, cover, remaining dagger, fruit, fruit crack and sparkle are defined first, then dagger, spinner, fruit, and remaining dagger state are initialized with the corresponding resources. These states manage the positions and animations of the game elements on Canvas.

```
// game animation and camera controller coroutine
LaunchedEffect( key1: true) {  this: CoroutineScope
    animation.animateTo(
        targetValue = 1f,
        animationSpec = infiniteRepeatable(animation = tween(durationMillis = 250, easing = LinearEasing))
    )
}
// game state handling for important events
LaunchedEffect(gameState.value) {  this: CoroutineScope
    if (gameState.value.isWipe()) {
        gameLevel.value = 1
        showCamera.value = gameDifficulty.value != 0
        cameraReady.value = false
        showScoreBoardButton.value = false
        delay( timeMillis: 100) //Delay for score board scores to show longer
        showTopScore.value = false
        gameScore.value = 0
        fruitGained.value = 0
        gameState.value.setReset()
    } else if (gameState.value.isReset()) {
        GameUtil.updateLevelInfo(randomSpeed, clockwise, spinSpeed, minSpeed, maxSpeed, remainingDaggers)
        spinSpeed.value *= if(clockwise.value) 1f else -1f
        spinner.value = spinnerUtil.value.getRandomSpinner()
        val rotationList = daggerState.reset()
        fruitState.reset(rotationList)
        spinnerState.reset()
        gameState.value.setRunning()
    } else if (gameState.value.isLeveling()) {
        gameLevel.value++
        var loopCnt = when (gameDifficulty.value) { 1 -> 2; 2 -> 3; 3 -> 5; else -> 0 }
        while(loopCnt > 0) {
            fruitState.addBonusFruit()
            delay( timeMillis: 100)
            loopCnt--
        }
        delay( timeMillis: 500)
        gameState.value.setReset()
    } else if (gameState.value.isOver()) {
        if (maxScore.value < gameScore.value) {
            maxScore.value = gameScore.value
            gameData.saveMaxScore(maxScore.value)
            showTopScore.value = true
        }
    }
}
// Smile Control
LaunchedEffect( key1: gameMode.value.isSmile() && smileP.value > faceSmileSensitivity.value) {  this: CoroutineScope
    if (gameMode.value.isSmile() && smileP.value > 0.5f && gameState.value.isRunning()) gameState.value.setShooting()
}
// Both Eye Control
LaunchedEffect( key1: gameMode.value.isBoth() && leftP.value < faceLeftSensitivity.value && rightP.value < faceRightSensiti
    if (gameMode.value.isBoth() && leftP.value < 0.2f && rightP.value < 0.2f && gameState.value.isRunning()) gameState.va
}
// spin speed controller coroutine
LaunchedEffect(randomSpeed.value) {  this: CoroutineScope
    while(randomSpeed.value) {
        spinSpeed.value = (minSpeed.value ≤ .. ≤ maxSpeed.value).random().toFloat() * (if(clockwise.value) 1 else -1)
        delay( timeMillis: (1 ≤ .. ≤ 6).random() * 500L)
    }
}
// hit animation
LaunchedEffect(daggerHit.value) {  this: CoroutineScope
    if (daggerHit.value) {
        delay( timeMillis: 50)
        daggerHit.value = false
    }
}
```

Figure 43. Coroutines in Game Console

From the above figure (figure 43), a total of 6 coroutines are implemented to control the flow of the whole game. The first coroutine function is used to repeatedly animate the game by updating the “animation” object. The second coroutine function is used to handle game state changes. Whenever the value of gameState changes, it will go through the conditions to match the corresponding actions. The third and fourth coroutine function detects smiling or blinking using probabilities given by MLKit API (details will be given in Face Detection section). The fifth coroutine function generates a random spin speed within the min speed and max speed provided in figure 42, for every random period of seconds. The last coroutine is an animation that occurs for 50 milliseconds every time a dagger hits the target.

```
Canvas(modifier = Modifier
    .fillMaxSize()
    .pointerInput(Unit) {  this: PointerInputScope
        detectTapGestures(
            onTap = {  it: Offset
                if (gameMode.value.isTap() && gameState.value.isRunning()) gameState.value.setShooting()
            }
        )
    }
) {  this: DrawScope
    animation.value //use to maintain animation loop
    if (gameState.value.isShooting() || gameState.value.isLeveling()) {
        daggerState.shoot {
            daggerHit.value = true
            fruitState.hit()
        }
    } else if (gameState.value.isLosing()) {
        spinSpeed.value = 0f
        daggerState.drop()
    }
    daggerState.draw( drawScope: this)
    fruitState.draw( drawScope: this)
    spinnerState.draw( drawScope: this)
    remainingDaggerState.draw( drawScope: this)
}
//Score Board
DrawScoreBoard(navController, fruitState)
}
```

Figure 44. Game Console Implementation (Part 2)

The code snippet above consists of a canvas that draws the game interface. It has a modifier with pointer input that listens for tap gestures (tap motion). If game mode is tap and game state is RUNNING, game state is set to SHOOTING. “animation.value” is used to animate the Canvas to keep it recomposing. And every time it recomposes, the 4 states draw onto the Canvas. Last but not least, the score board is drawn.

2.10 Face Detection

To start doing face detection, we have to set up a face detection analyzer first. This analyzer overrides the analyze function in ImageAnalysis and uses the face detector to detect faces in the image. If a face is detected, smiling, left eye, and right eye probability are retrieved from the face object, and are used by the game console to detect blinking and smiling.

```
class FaceDetectionAnalyzer(
    private val faceDetector: FaceDetector,
    private val lensFacing: MutableState<Boolean>,
) : ImageAnalysis.Analyzer {
    @androidx.annotation.OptIn(androidx.camera.core.ExperimentalGetImage::class)
    override fun analyze(imageProxy: ImageProxy) {
        val mediaImage = imageProxy.image
        if (mediaImage != null) {
            cameraReady.value = true
            val image = InputImage.fromMediaImage(mediaImage, rotationDegrees: 0)
            faceDetector.process(image).addOnSuccessListener { faces ->
                for (face in faces) {
                    if (face.smilingProbability != null) smileP.value = face.smilingProbability!!
                    if (face.leftEyeOpenProbability != null) leftP.value = if (!lensFacing.value) face.rightEyeOpenProba
                    if (face.rightEyeOpenProbability != null) rightP.value = if (!lensFacing.value) face.leftEyeOpenProb
                    Log.d( tag: "gameValues", msg: "face values: ${leftP.value}, ${rightP.value}, ${smileP.value}")
                    break
                }
                imageProxy.close()
            }
        }
    }
}
```

Figure 45. Face Detection Analyzer

The facial detection analyzer itself won't perform any actions. We have to know how to use it. Figure 46 on the next page demonstrates how to set up the camera. The methodology is in fact difficult but understandable, where it first retrieved the current lifecycle owner and local context, then created a cameraProviderFuture and cameraExecutor. It then created a FaceDetector object from ML Kit Vision library for face detection. The face detector options are set to retrieve all classification modes (for smiling, blinking probabilities). Next, if camera permission is granted, and when the value of showCamera is true, an AndroidView is used to display the camera preview. The camera view is then modified according to preferences and is set with lens facing in the settings. Afterwards, analyzer is set up in the Image Analysis object, and our FaceDetectionAnalyzer is initialized and utilized. Lastly, everything is bound to the lifecycle defined above.

```
@Composable
fun DrawCamera() {
    val lifecycleOwner = LocalLifecycleOwner.current
    val context = LocalContext.current
    val cameraProviderFuture = remember { ProcessCameraProvider.getInstance(context) }
    val cameraExecutor = remember { Executors.newSingleThreadExecutor() }
    val faceDetector = remember {
        FaceDetection.getClient(
            FaceDetectorOptions.Builder()
                .setLandmarkMode(FaceDetectorOptions.LANDMARK_MODE_NONE)
                .setContourMode(FaceDetectorOptions.CONTOUR_MODE_NONE)
                .setClassificationMode(FaceDetectorOptions.CLASSIFICATION_MODE_ALL)
                .setPerformanceMode(FaceDetectorOptions.PERFORMANCE_MODE_FAST)
                .build())
    }
    if (PermissionUtil.hasPermission() && showCamera.value) {
        val cameraAlpha = animateFloatAsState(targetValue = if (gameState.value.isOver()) 0f else 1f, animationSpec = tween( durationMillis: 500))
        val cameraOffset = animateDpAsState(targetValue = if (!cameraReady.value) screenWidthDp else 0.dp, animationSpec = if (cameraReady.value) tween
        Box(modifier = Modifier.fillMaxSize()) {  this: BoxScope
            AndroidView(
                modifier = Modifier
                    .size(130.dp)
                    .offset( x: screenWidthDp.times( other: 0.65f) + cameraOffset.value, screenHeightDp.times( other: 0.75f))
                    .rotate( degrees: -90f + cameraRotationSettings.value)
                    .scale(cameraOutScaleSettings.value)
                    .clip(CircleShape)
                    .scale(1.5f * cameraInScaleSettings.value)
                    .alpha(if (showCameraSettings.value) cameraAlpha.value else 0f),
                factory = { ctx ->
                    val previewView = PreviewView(ctx).apply {  this: PreviewView
                        implementationMode = PreviewView.ImplementationMode.COMPATIBLE
                    }
                    cameraProviderFuture.addListener({
                        val cameraProvider = cameraProviderFuture.get()
                        val preview = Preview.Builder()  Preview.Builder
                            .setTargetResolution(Size(previewView.width, previewView.height))
                            .build()  Preview
                            .also {  it: Preview
                            it.setSurfaceProvider(previewView.surfaceProvider)
                        }
                        val cameraSelector = CameraSelector.Builder()
                            .requireLensFacing(if (lensFacing.value) CameraSelector.LENS_FACING_FRONT else CameraSelector.LENS_FACING_BACK)
                            .build()
                        val imageAnalysis = ImageAnalysis.Builder()
                            .setTargetResolution(Size(previewView.width, previewView.height))
                            .setBackpressureStrategy(ImageAnalysis.STRATEGY_KEEP_ONLY_LATEST)
                            .build()
                        imageAnalysis.setAnalyzer(cameraExecutor, FaceDetectionAnalyzer(faceDetector, lensFacing))
                        try {
                            cameraProvider.unbindAll()
                            cameraProvider.bindToLifecycle(
                                lifecycleOwner,
                                cameraSelector,
                                preview,
                                imageAnalysis
                            )
                        } catch (exc: Exception) {
                            Log.e( tag: "game",  msg: "camera preview failed", exc)
                        }
                    }, ContextCompat.getMainExecutor(ctx))
                    previewView  ^lambda
                }
            )
        }
    }
}
```

Figure 46. Draw Camera Implementation

2.11 Questionnaire

After explaining all the mechanics and methodologies of my game, I am interested in measuring its effectiveness in terms of training facial muscles. I would like to assess how well the game improves facial muscle tone and strength and gather feedback and insights from players on the effectiveness of the game for this purpose. I have conducted a survey to gather feedback for my game from individuals across various age groups.

This questionnaire, created using Google Forms (see Appendix F for details), consists of 16 straightforward Chinese questions that aim to evaluate the game's responsiveness, feasibility, effectiveness in terms of facial action controls and gameplay, and overall rating. The questions are designed in Chinese because many elderlies can't read English. The questions are designed to gather valuable feedback from players and assess their experience playing the game, including how well it improves facial muscle training and toning. Without further ado, let's look at the questions.

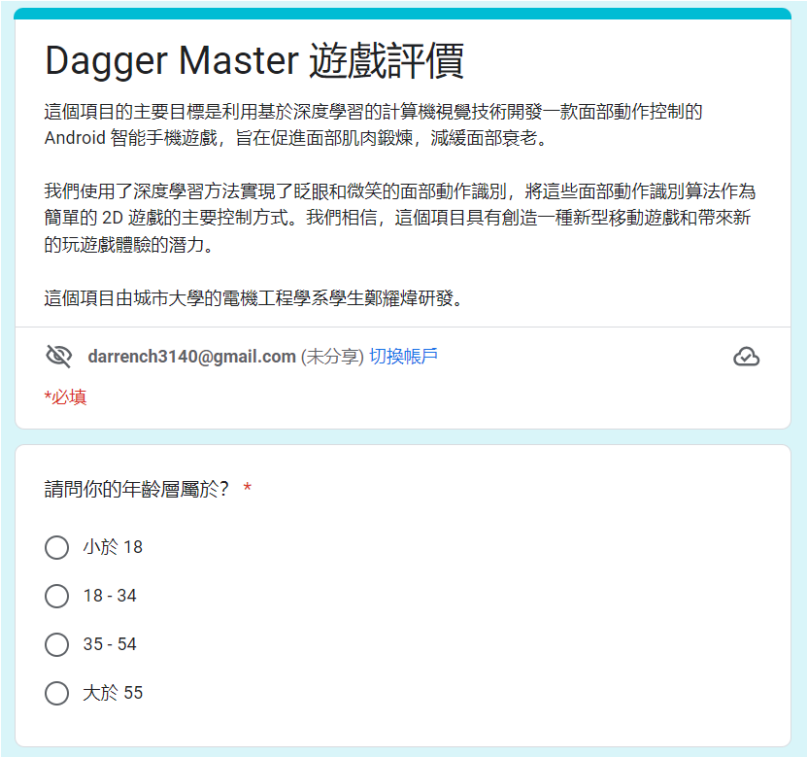


Figure 47. Questionnaire (Part 1)

Figure 48. Questionnaire (Part 2)

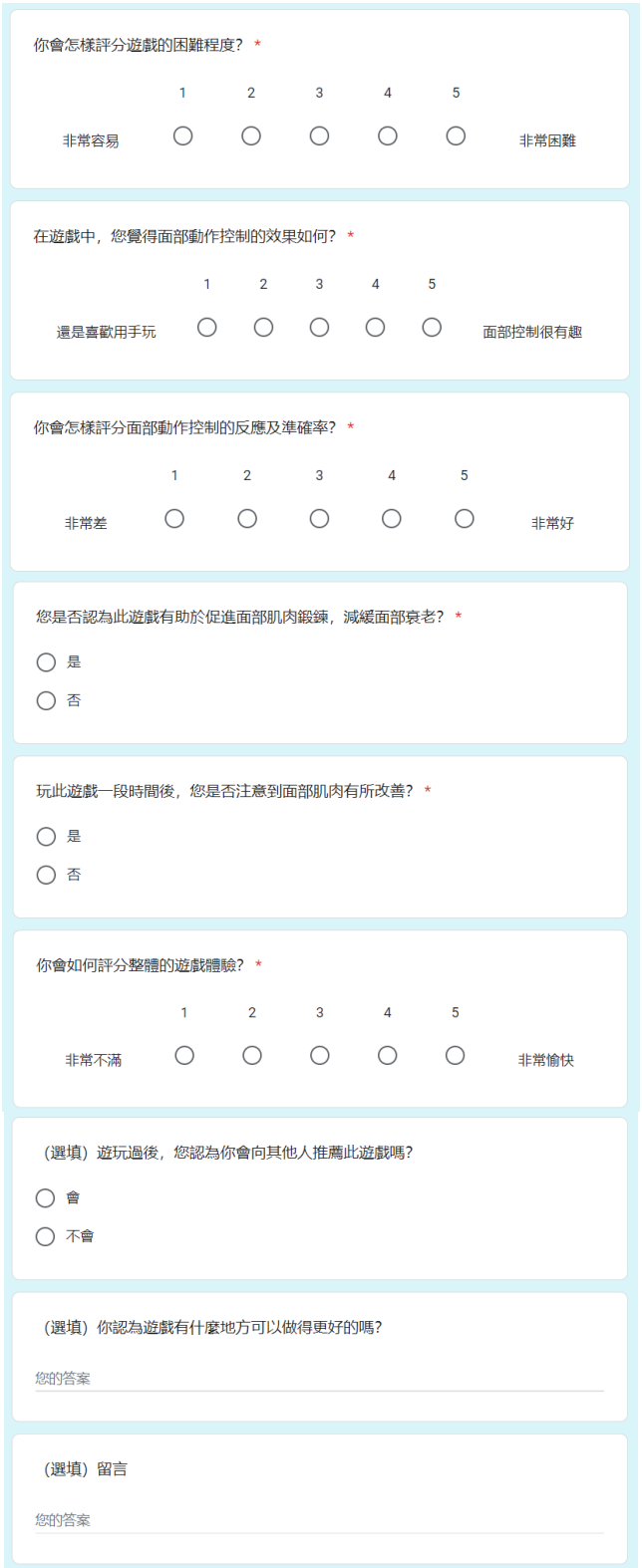


Figure 49. Questionnaire (Part 3)

3. Results

In this project, I have successfully developed Dagger Master (check Appendix E for GitHub source code), as well as obtained positive feedback from the questionnaire. I have also managed to meet all the objectives of this project. Without further ado, let's take a look at the finishing results of Dagger Master and feedback from the questionnaire.

NOTE: Some figures in the upcoming sections are in GIF format. If it doesn't animate on your screen, you can simply tap on the image and go to the hyperlink provided.

3.1 Loading Screen

The loading screen consists of a sword that shows and disappears, along with a logo popping out that bounces onto the top, as if gravity is reversed.

Figure 50. Loading Screen Demo GIF

Figure 51. Loading Screen Sword Appear

Figure 52. Icon bounces to top

3.2 Landing Screen

The landing screen consists of two main pages: the main page and game mode selection page. In the main page, the logo moves up and down consistently, and sparkles shine beside it. The mode selection page consists of 3 modes, where smile and blink mode have extra rewards.

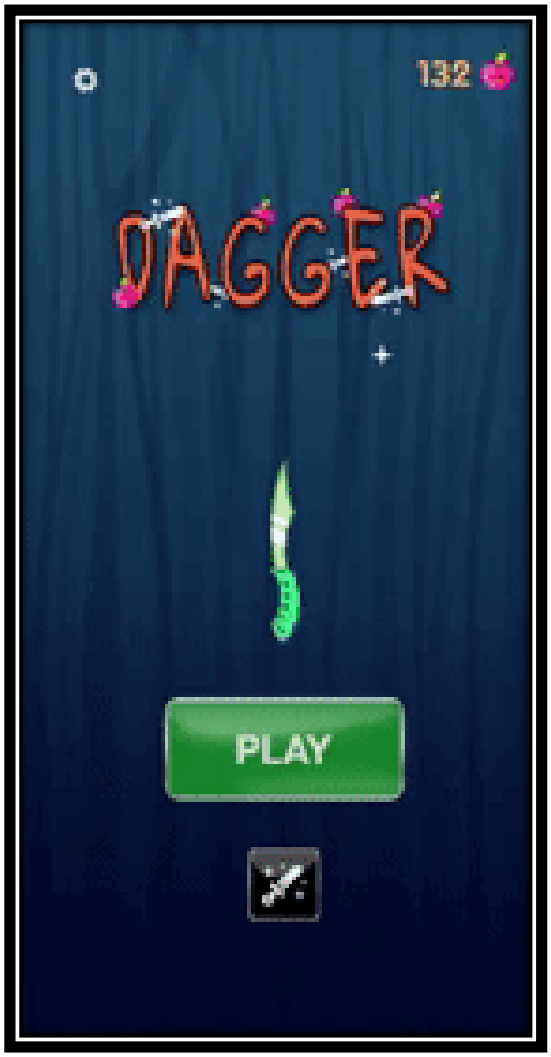


Figure 53. Landing Screen Demo GIF

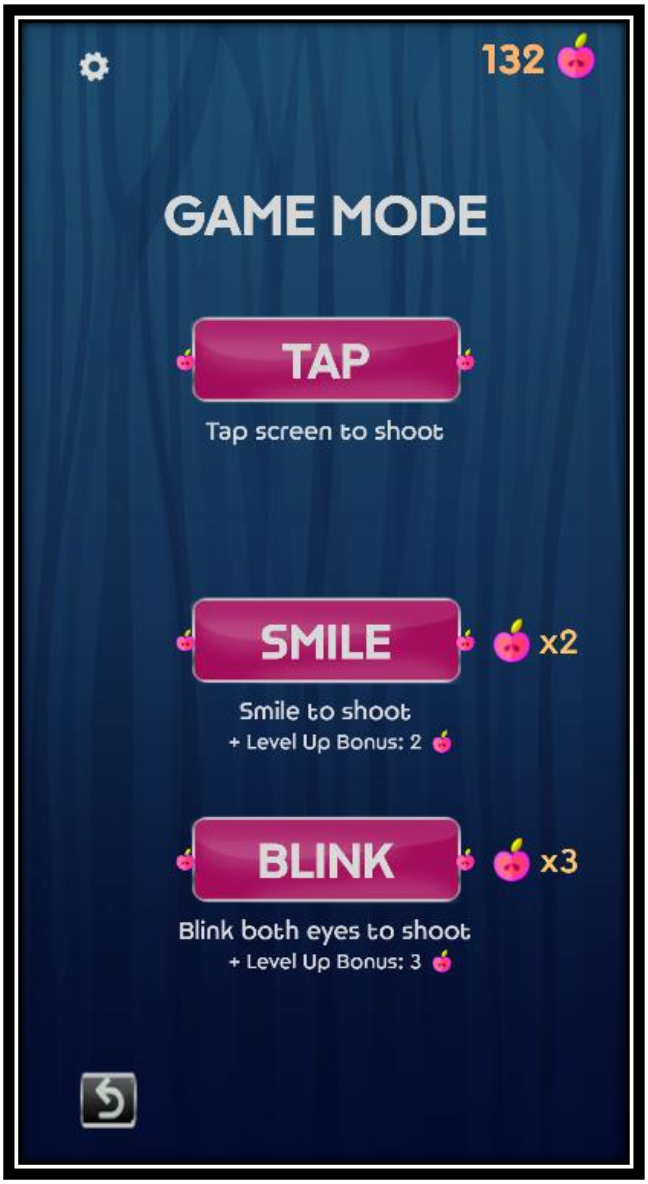


Figure 54. Game Mode selection page

3.3 Game Screen

After a game mode is selected in the landing screen, the game screen slides in from the right. Everything is ready and spinning in the middle of the UI. As the player takes action, the dagger shoots onto the spinner target, sticks to it and starts spinning. The remaining daggers indicator on the bottom left decreases as soon as the dagger is shot out. Animations make everything smooth and realistic. Besides, if a fruit is hit, it cracks and runs towards the top fruit bar. Lastly, when the player loses, the dagger drops, and the score board appears.

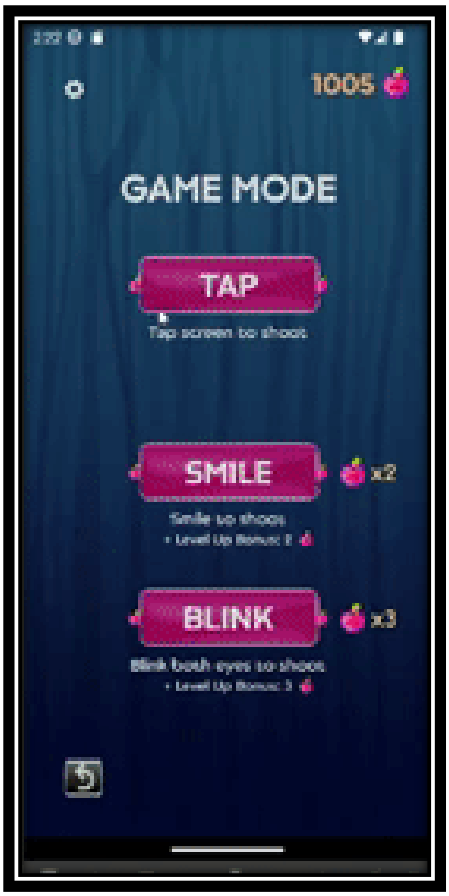


Figure 55. Tap Mode gameplay GIF

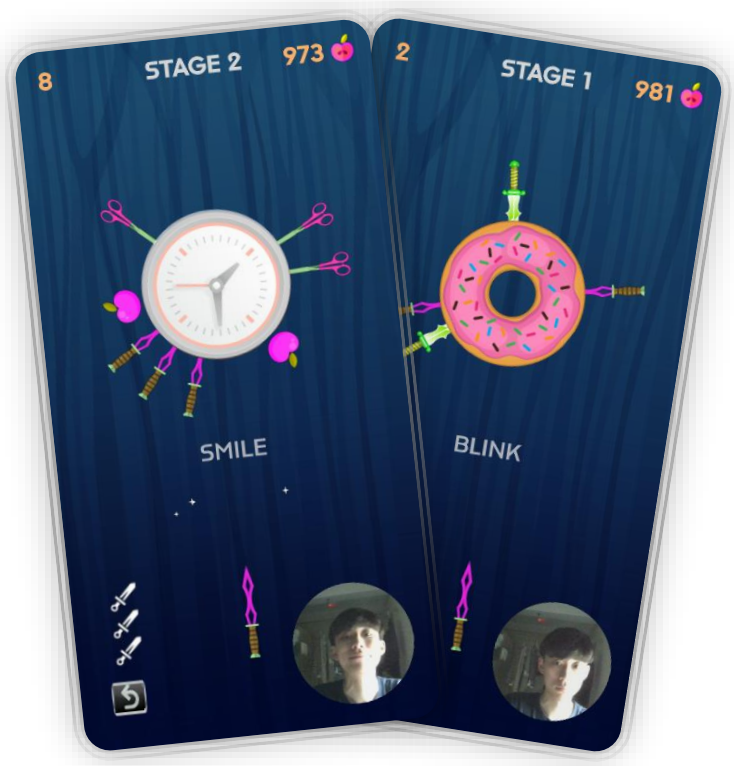


Figure 56. Facial Mode gameplay

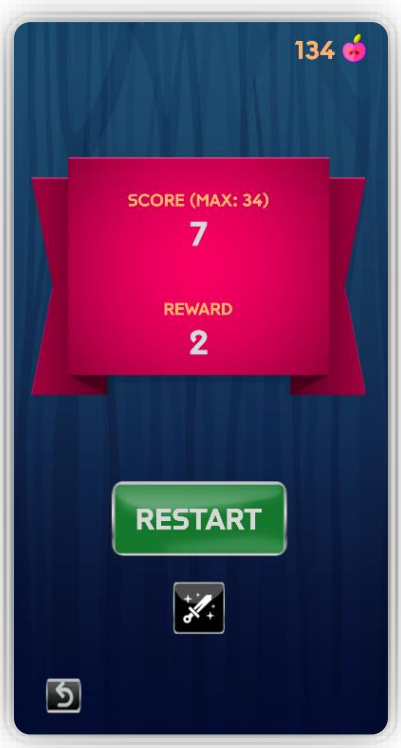


Figure 57. Score Board Overview

Figures 56 demonstrates both smile mode and blink mode gameplay. It can be seen that a text helper of SMILE and BLINK is shown in the middle of the UI to notify players of the type of motion controlling the gameplay. Next to it is the score board overview (figure 57), it gives players information on the max score, current score, as well as number of fruit rewards. Underneath it consists of a restart button, return button and a shop button, allowing players to restart the game, buy daggers, or go back to the main screen, with more convenience.

3.4 Shop Screen

The shop screen has a total of 16 daggers. A large spotlight on the top indicates the purchased and selected dagger, where this dagger is also surrounded by the pink box in the grid. Unpurchased daggers have “locked image” and are surrounded by the green box when the user clicks on it. Finally, a purchase button at the bottom allows players to purchase daggers when they have enough fruits.

Figure 58. Shop Screen Demo

3.5 Settings Screen

The settings screen consists of game and camera settings.

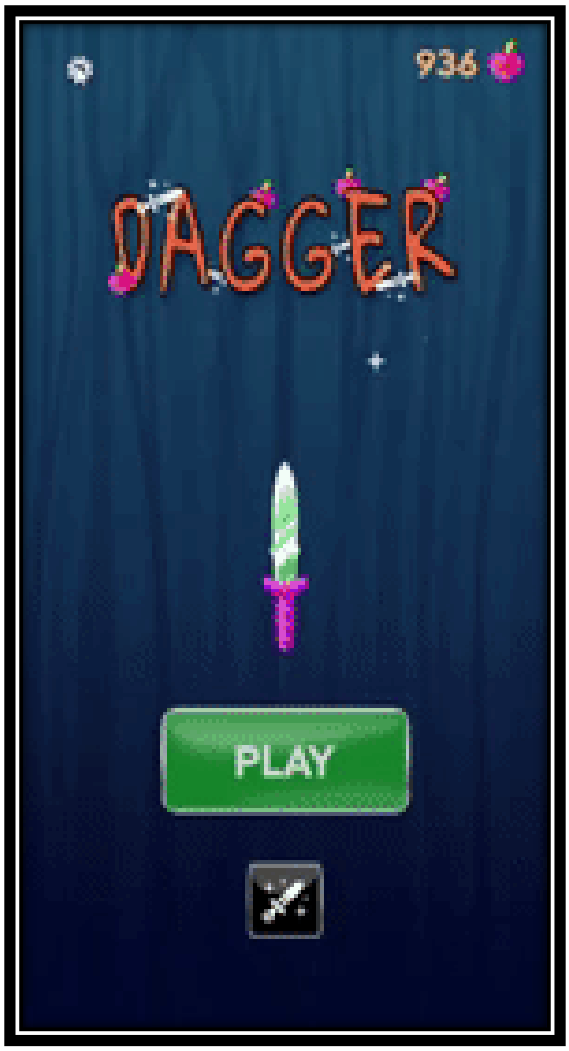


Figure 59. Settings Screen Demo GIF

Figure 60. Settings Screen Overview

The settings can be scrolled manually by users. The section titles “Game” and “Camera” are sticky headers. Hence, you can see that “Game” section stayed there even if left eye sensitivity is scrolled over already. When “Camera” section is scrolled to the top, it takes over the Game section and sticks on top.

3.6 Questionnaire Results

There are a total of 14 contestants participating in this questionnaire. Let’s take a look at the results.

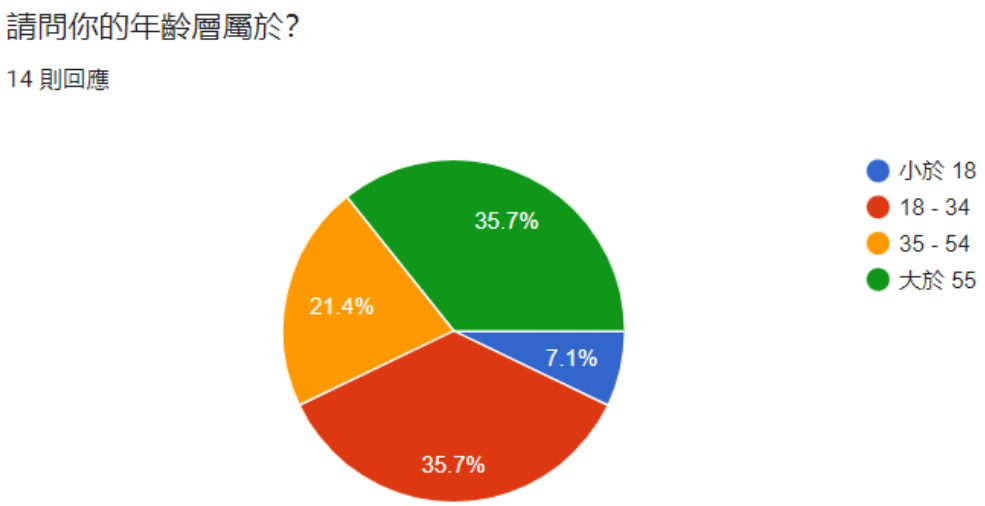


Figure 61. Questionnaire Q1 Result

This question asks about the age group of participants. From the results, the majority of participants are 18-34 years old, or larger than 55 years old.

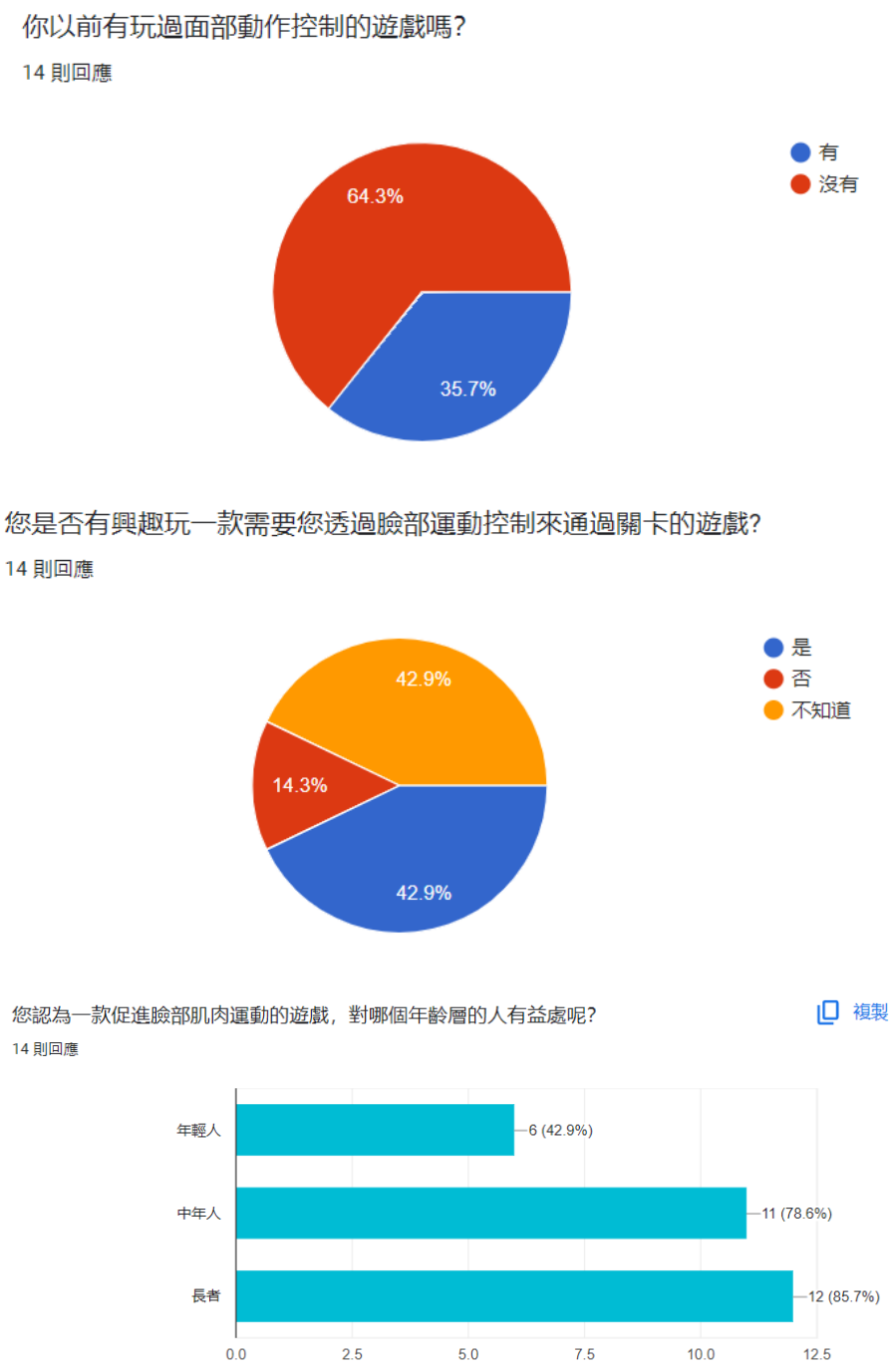


Figure 62. Questionnaire Q2-Q4 Result

In Q2, 64.3% of participants have never played games that are controlled by facial action controls. In Q3, 42.9% of participants are not sure about whether they will be interested in this type of mobile game, while another 42.9% say they are. From Q4, it appears that most

respondents believe that the game would be beneficial for middle-aged, and elderly people as compared to teenagers.

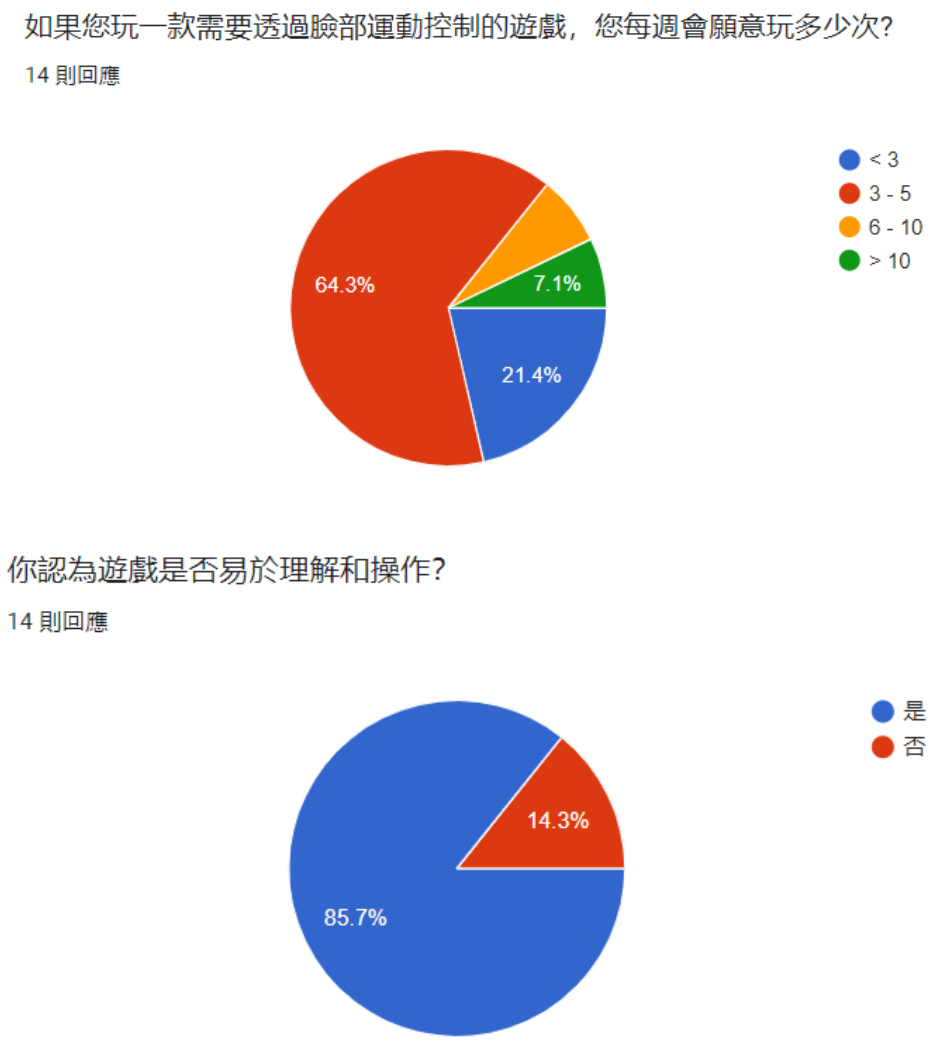


Figure 63. Questionnaire Q5-Q6 Result

In Q5, it appears that the majority of respondents are willing to play this game for 3-5 times per week. While in Q6, more than 85% of respondents agree that the game is easy to understand and operate.

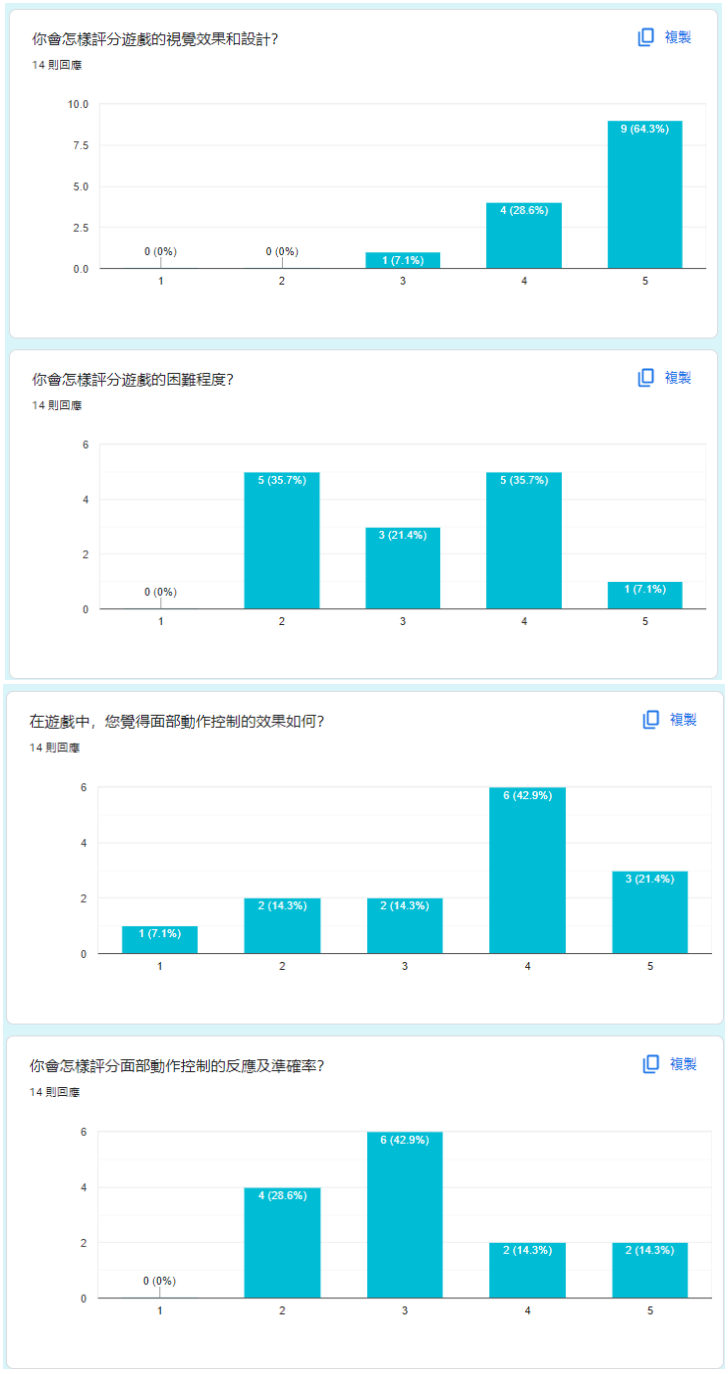


Figure 64. Questionnaire Q7-Q10 Result

In Q7, around 65% of respondents like the design and visual effects of Dagger Master. While in Q8, 35.7% of the respondents think the game is fairly difficult, and another 35.7% thinks it's fairly easy. Q9 suggests that 9 out of 14 participants think the facial action controls are

doing great. Lastly, Q10 asks about the facial action control’s responsiveness and preciseness in Dagger Master, and the majority chose the middle range.

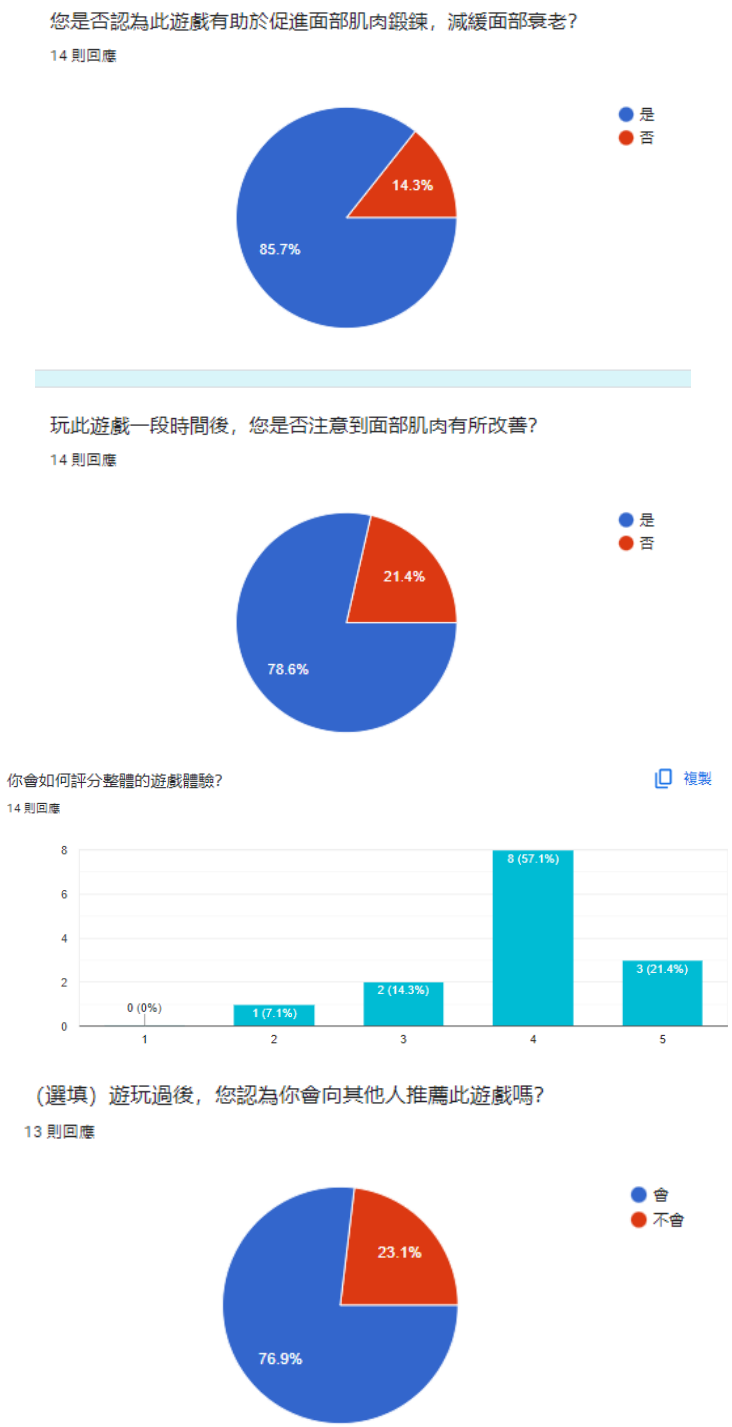


Figure 65. Questionnaire Q11-Q14 Result

In Q11 and Q12, 85.7% of the participants agreed that this game has helped them train their facial muscles, with 78.6% respondents noticing changes in their facial muscles. Last but not least, in Q13 and Q14, more than half of the respondents enjoyed the game experience and around 77% of them claimed that they might recommend this game to others.

4. Discussion

Based on the results of the questionnaire, I have obtained some notable findings and trends in Dagger Master. Firstly, the majority of participants are either in the 18-34 age group, or are larger than 55 years old. This suggests that Dagger Master can actually appeal to a wide range of age groups.

Next, in terms of familiarity with facial action control games, I noticed that a large portion of participants had never played this type of game before. However, a considerable number of respondents expressed interest in this type of mobile game, indicating a potential market for Dagger Master.

In terms of the game's design, a majority of participants liked the design and visual effects of Dagger Master. However, when it came to the game's difficulty level, opinions became mixed, with almost equal numbers of respondents finding the game fairly difficult or fairly easy. It might be hard for me to think how to fine tune the difficulty of this game.

I noticed that facial action controls in Dagger Master are doing great, and most respondents noticed changes in their facial muscles after playing the game. Participants enjoyed the game experience, and they are willing to recommend it to others.

Overall, the findings in this project suggest that Dagger Master has the potential to be a successful game for promoting facial muscle training and toning. However, small sample size and potential biases of respondents are the limitations of this study. To provide a better result, further research may be necessary to confirm these findings, allowing me to further optimize the game's design and features.

5. Conclusion

To conclude, this project has been successful in achieving its objectives. The development of Dagger Master has been a significant accomplishment, where I implemented more and more features during the way of development, starting from simple animations to industrial level animations, as well as from the implementation of level systems, reward systems, to shop and settings systems.

It has been challenging along the path, where I faced a lot of challenges and did a lot of debugging. But everything turns out to be worth it, where the positive feedback received from the questionnaire, and appraisals from peers and supervisors are proof of this app's success.

Moving forward, I plan to continue improving Dagger Master, exploring new features and functionalities, to bring a better user experience for gamers to improve their facial muscles.

Appendices

Appendix A: Free Source Code Link

I discovered the source code from this YouTube link:

https://www.youtube.com/watch?v=wz9e8RfQgeg

The link to the source code:

https://drive.google.com/file/d/13fh70vq5cD4YyKE6dYp7AAHyoOe_cPk7/view

Appendix B: "Knife Hit" on Google Play Store

The app is currently available on the Google Play Store:

https://play.google.com/store/apps/details?hl=en&id=com.ketchapp.knifehit

Appendix C: Navigation Animation Library Link

The documentation of this library available on:

https://google.github.io/accompanist/navigation-animation/

Appendix D: Detail implementation of all UI Components

If you are interested in any UI components shown in the report, feel free to check out this link:

https://github.com/darrench3140/fyp_dagger/blob/master/app/src/main/java/com/darren/fyp_dagger/utils/UIComponent.kt

Appendix E: GitHub Source code for Dagger Master

The source code of Dagger Master can be found in:

https://github.com/darrench3140/fyp_dagger

Appendix F: Questionnaire

The questionnaire can be found here: https://forms.gle/jDVSFbBQFDPxvWKj6

References:

[1] Rad, A. (2022, October 21). Orbicularis Oculi, Kenhub. Retrieved April 1, 2023, from https://www.kenhub.com/en/library/anatomy/orbicularis-oculi

[2] Arora, D. (2018, August 6). 5 benefits of blinking which you miss out on while staring at your smartphone. TheHealthSite. Retrieved April 1, 2023, from https://www.thehealthsite.com/news/benefits-of-blinking-your-eyes-which-you-miss-out-on-while-staring-at-your-smartphone-d0818-589376/

[3] Lazarus, R. (2021, December 23). The truth about gaming and your child's vision. Optometrists.org. Retrieved April 1, 2023, from https://www.optometrists.org/childrens-vision/healthy-eyes-for-life-8-ideas-to-teach-children/the-truth-about-gaming-and-your-childs-vision/

[4] Crooks, F. (Ed.). (2019, August 8). When Eye Blinking Is a Problem. Healthline. Retrieved April 1, 2023, from https://www.healthline.com/health/eye-health/eye-blinking

[5] Stibich, M. (2023, February 17). 10 Big Benefits of Smiling. Verywell Mind. Retrieved April 1, 2023, from https://www.verywellmind.com/top-reasons-to-smile-every-day-2223755

[6] Spector, N. (2018, January 10). Smiling can trick your brain into happiness - and boost your health. NBCNews.com. Retrieved April 1, 2023, from https://www.nbcnews.com/better/health/smiling-can-trick-your-brain-happiness-boost-your-health-ncna822591

[7] Chaldea, N & Lupiyoadi, R. (2018). Driving Mobile Game Engagement: Factors and User Metrics. [Online]. Available: https://download.atlantis-press.com/article/125918995.pdf