Posts and Telecommunications Institute of Technology

Faculty of Information Technology- Software Engineering 1

Final report
Mobile Application Development

Class	**:** E22CNPM01
Group	**:** 5
Member	**:** Nguyễn Nhật Hưng - B22DCDT148
	: Nguyễn Minh Đức - B22DCCN234
	: Hồ Anh Dũng - B22DCVT090
	: Vũ Minh Đức - B22DCVT164
Topic	**:** Mini-Game Hub
Lecturer	**:** Đào Ngọc Phong

No	Name	Shared work	Assigned Tasks	%
1	Nguyễn Nhật Hưng (Team Leader)	- Scouting for suitable topic for the project - Vote for final topic of the project - Scouting for suitable game for the project - Developing games for the project -Write individual result final report	**07 Modules:** - 1. Main Menu & Intro System - 2. Noughts And Crosses - 3. Minesweeper - 4. 2048 - 5. Pong - 6. MonsterBattler (Complex RPG w/ Database) - 7. Firebase real-time database **Report writing:** - Preface - I.Topic overview - III. Final conclusion **General project leader work:** - Workflow organizing to distribute missions/works to member - Final reviewer/ tester and after review code (all modules)	40%

2	Nguyễn Minh Đức		**02 Modules:** - 1. Bubble Shooter. - 2. Whack-A-Mole.	18%
3	Hồ Anh Dũng		**01 Modules:** - 1.Tetris	12%
4	Vũ Minh Đức		**05 Modules:** - 1. Arkanoid - 2. Flappy Bird - 3. Memory Card - 4. Snake - 5. TowerBloxx	30%

Preface

I have been a gamer my whole life, and i am proud of it. It all started with my mom's old Nokia. It was small, the buttons were extremely clicky, but to me it was everything. Western kids experienced Assassin's Creed as an open-world game, but i had it as a side-scrolling, almost 1942-style shooter where the character rode his eagle and destroyed battleships.

My gaming spark faded pretty early because my family was strict and believed gaming made people more violent. But everything changed around 2012 when an old friend brought over his red PSP. I was mesmerized by the sight—like a Native American getting the colonized-treatment from the Spanish, and so i thought to myself: the sky is the limit; there are unlimited games to try and so much more technology to explore.

After that fateful day, my dream shifted from wanting a new bicycle to wanting something like a PSP to play games on. The dream was crushed the moment i looked at my parents, but it wasn't dead. I kept it inside until 11 years later, when i finally used my own money to buy a handheld console called the Miyoo Mini Plus. Using it exclusively to play retro games, i felt like a kid again—even though 80% of its library is older than me.

While playing on it, i started to notice that some developers were still creating homebrew titles, and some were even making brand-new games in a retro style, built specifically for these old systems. I thought they were incredibly talented. And then a question sparked: could I do something like that too? Not just retro games—games in general?

That question began as a tiny spark of imagination, but the more i think about it, the more exciting it becomes. I started imagining what the game-making process would feel like, what it would be like to turn the ideas in my head into a world someone could actually play. Before i realized it, that spark had grown into a new hobby—something that now sits alongside my love for gaming and my passion for tech devices.

That is the reason why this project was created. This may not be the first game i have ever made, but the passion—and the imagination behind it—are still exactly the same as they were on day one.

Contents

I. Topic overview

1. Background and Motivation

As the leader for Group 5, one of my mission is to select a topic that not only fits the group but also provides the best chance to successfully meet the all the requirements.

After careful consideration, I have decided the project will be **"Mini-Game Hub"** as the group's topic. This topic might be unique, but i hope to bring a new wind to the conversation, the project also suited to address all three CLO:

- **Analysis & Design (CLO 1):** We will design a "hub" architecture, create an interface for game selection, and then analyze and design the mechanics for numerous distinct mini-games (including AI for some) and a database for scoring.
- **Build & Deploy (CLO 2):** The project will be entirely in kotlin programming language due to the exsistence of jetpack compose - a useful tool that allow user to make layout without having to make .xml file.
- **Presentation (CLO 3):** A game hub with multiple games will be a great way to present because the variety of mechanics to present is vast and nothing like others

Furthermore, my personal expertise in this domain will be a significant asset. It allows me to provide clear direction, effectively guide team members through challenges, and organize the project's workflow effectively.

2. Market scouting

Even though this is an unique idea to present, the idea of a mini game-hub with tons of games is not new, and that is how we get our idea from aside from the already present single minigame, one of the most noteworthy examples is “Offline Games – No Wifi Games”, and i will analyze its pros and cons, in order to maximize the project’s potential:

The pros:

- Easy to approach: Family friendly design from interface to the game itself.
- Flexible switch from wifi to no-wifi mode: The app will behave differently between 2 mode, even though the different is not that significant.
- Vast library of games: It dealt with other game option with big gameplay loop, larger scale game with its number of game, which is about 30, and is update regularly.
- Simplicity: The app solve the user demand of “pick and play” by having straight forward design, just turn on the game, go through the game library tab and pick a game.

The cons:

- Quality: Common mistakes of every app similar, the demand of numbers make quality no where to be found. Even though there are 30 game or more, the similarity of some game remain debatable. The mechanics of some game often overlaps, make it looks more just like a reskin of another… All game but no game.
- Annoyingly flexible: the idea of flexible switch is there, but no breakthrough, it is just nothing when there is no wifi and tons of advertising when there is, when there could be so much more that can be done such as adding a leaderbroad for more competitve game environment, or maybe it can expand to multiplayer range.

From the analyze work above, we have learn some crucial point for our project:

- We will be on the middle ground between quality and quantity: While maintance the number of games, we will absolutely focus more on polishing and optimizing than blindly making clone or reskin.
- Develop a unified design: We aim to have a family friendly user interface, targeting all ages.
- Integrate various online feature: Leaderboard

3. Requirements

3.1. Functional Requirements

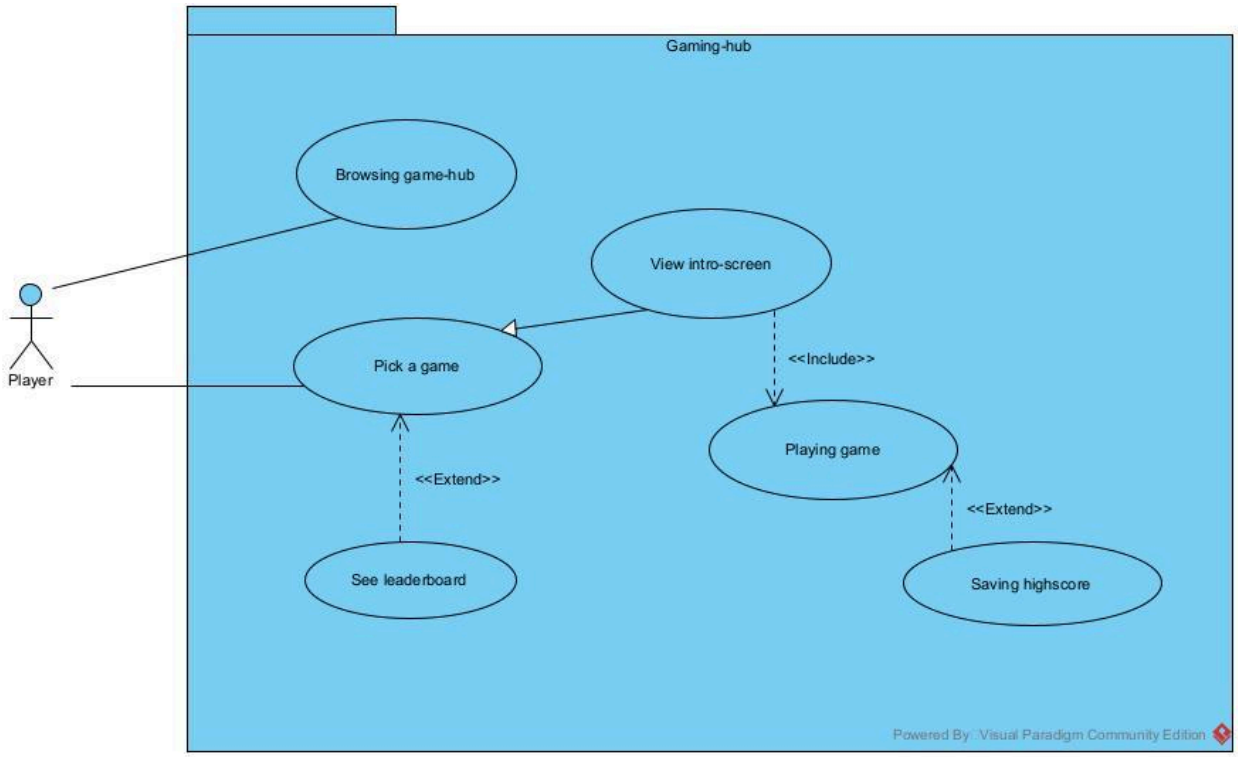
- **Centralized Game Hub:** Provide a main interface acting as a launcher, allowing users to browse, view details, and select games from a unified menu.
- **Diverse Game Library:** Offer a variety of mini-games with distinct mechanics and visual styles to ensure varied gameplay experiences. Not just a reskin of another game is our motto.
- **Global Leaderboard System:** Integrate a cloud-based scoring system using Firebase. This includes capturing player scores upon game completion, validating user input (3-character name), and displaying real-time global rankings.
- **Game State Management:** Ensure distinct game states such as Play, Pause, Resume, and Game Over are handled correctly across all games.

3.2. Non-functional Requirements

- **User-Friendly Interface:** Design an intuitive and accessible interface suitable for users of all age groups, utilizing clear navigation and consistent visual elements.
- **Modern UI Architecture:** Using **Jetpack Compose** for UI development, eliminating .XML file dependencies.

- **Performance & Stability:** Ensure smooth gameplay with a stable frame rate and minimal latency during game transitions and state changes.
- **Modularity & Scalability:** Implement a decoupled architecture where each mini-game functions independently.

4. Complete use-cases diagram



Propose Actors.

- One primary user: the **Player**.

Propose Use Cases.

- Based on the "Hub" features: **Browsing Game Hub**, **Select game**.
- Based on Gameplay: **Playing games**.
- Based on Data/Social features: **Saving high Score**, **View leaderboard**.

Refine Use Cases.

- **Generalization:** Abstract Use Case called **"Playing Game"** for every game
- **Relationships: <<include>>** relationship because you need to access the game after the intro-screen. "View Leaderboard" and saving to the leaderboard is an optional action the player can take, so we use the **<<extend>>** relationship

5. Architectural and Technology Analysis

5.1. Technology Selection

- **Programming Language: Kotlin**.
    - Seamless integration with modern Android Jetpack libraries.
- **UI Toolkit: Jetpack Compose**.
    - Fully **declarative UI model**, replacing the legacy XML view system.
    - **Interoperability:** For arcade-style games requiring high-speed rendering (e.g., Tetris, Arkanoid), we utilize the **AndroidView** composable to embed **SurfaceView**. This hybrid approach combines the ease of Compose UI with the raw performance of the legacy Canvas rendering loop.
- **Backend & Data Persistence: Firebase Cloud Firestore**.
    - The system was upgraded to a cloud-based solution using firebase instead of using sqlite or room
    - **Firestore** is implemented to manage the **Global Leaderboard System**. It provides real-time data synchronization, allowing score data to be stored as JSON-like documents.
- **State Management: Finite State Machine (FSM)**.
    - Game logic is controlled via sealed classes representing distinct states (e.g., GameState.Ready, GameState.Playing, GameState.GameOver). This ensures deterministic behavior and easier debugging of game flows.

5.2. Architectural Pattern

The application follows the **MVVM (Model-View-ViewModel)** architecture pattern combined with **Modular Monolith** styled project structure.

5.2.1. Modular Monolith Structure (Package-by-Feature):

- **Benefit:** One single big folder with many subfolder, easy and simple.

5.2.2. MVVM Implementation:

- **Model (Data & Logic):** Contains data classes, repository objects, and pure game logic. It handles *how* the game works and communicates with Firebase Firestore.
- **View (UI):** Jetpack Compose functions and SurfaceViews. View is strictly passive; it only renders the current state provided by the ViewModel and captures user input.
- **ViewModel (State Holder):** Bridge between Model and View.
    - It holds state using MutableState or StateFlow.

- Ensures the UI automatically recomposes whenever the game state changes (Reactive UI)

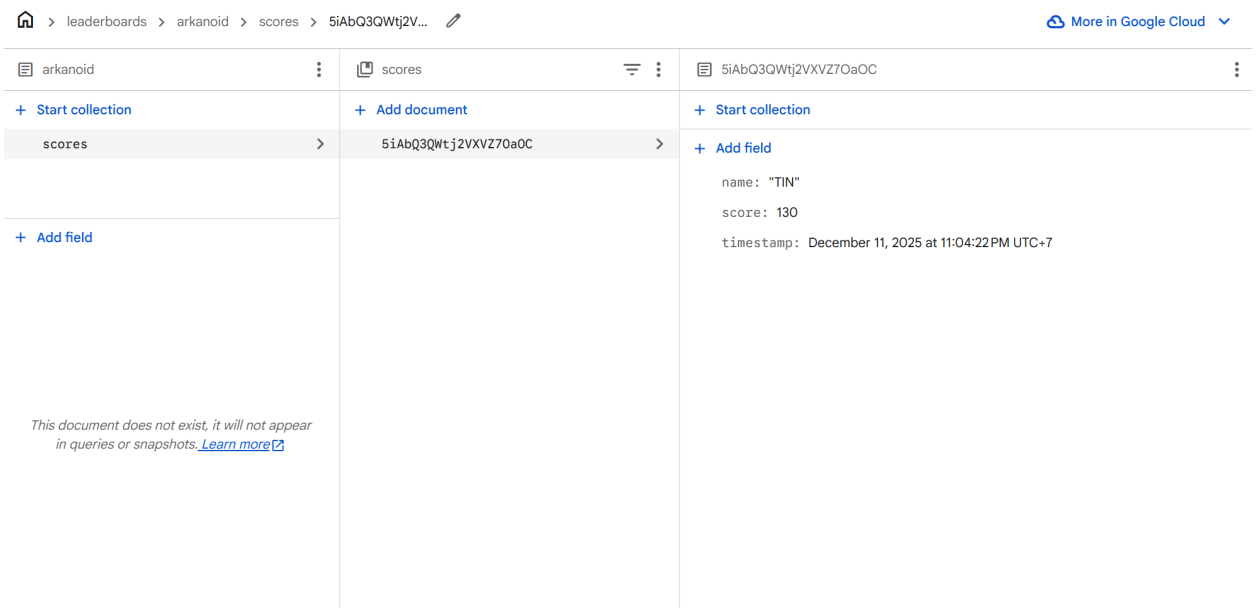
6.Database and storage system

6.1. Cloud Data Storage (Firebase Cloud Firestore)

The Global Leaderboard system utilizes **Google Firebase Cloud Firestore**. This NoSQL document-oriented database was selected for its real-time synchronization capabilities, allowing scores to be updated instantly across all client devices without manual refreshing.

Data Structure Strategy (Multi-tenancy): Instead of using separate tables for each game, the system uses a hierarchical document structure. This allows a single database instance to serve multiple games ("Monster Battler", "Tetris", "Arkanoid") by using the game_id as a document key.

NoSQL Schema Visualization: The data is organized into a root collection containing game-specific documents, which in turn contain sub-collections for score entries.



Field Specifications:

- **name (String):** The player's display name. Enforced validation restricts this to 3 uppercase characters (e.g., "AAA").
- **score (Integer):** The final score achieved in the game session.

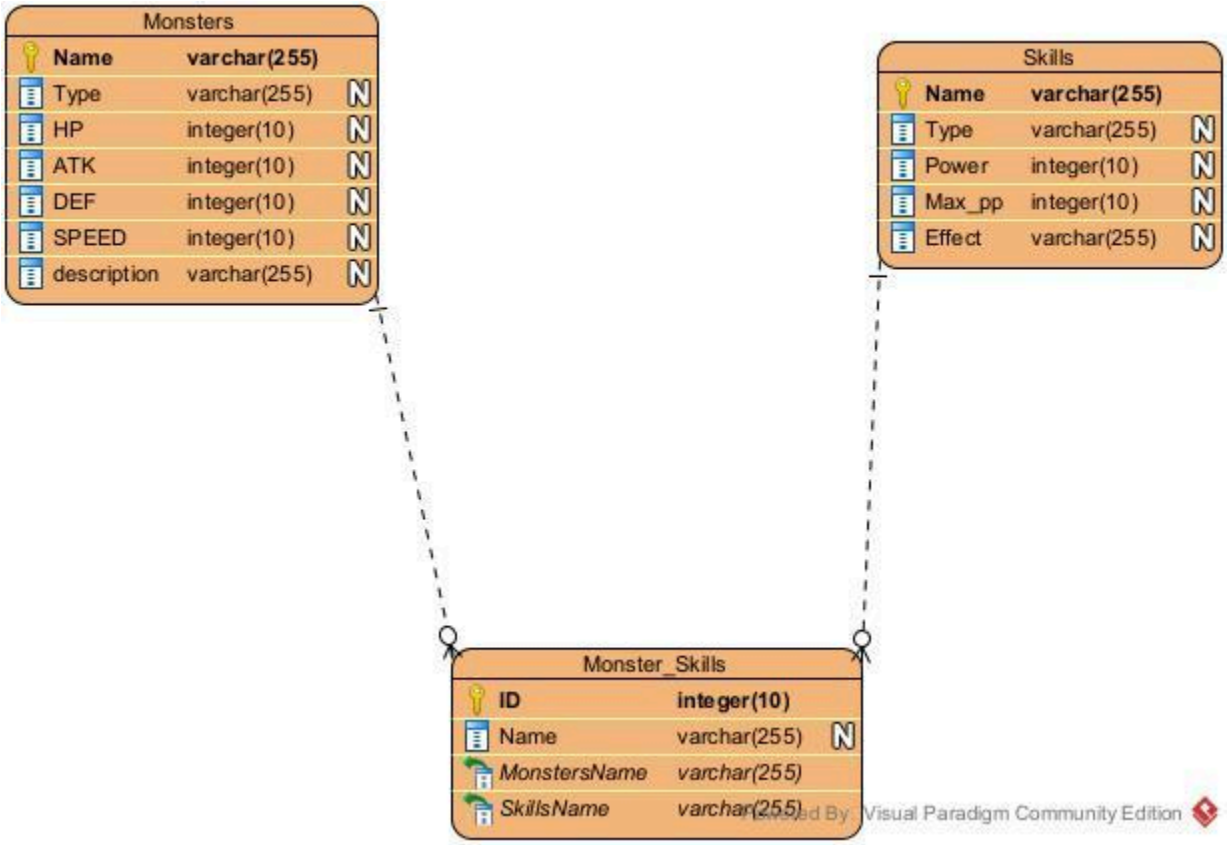
- **timestamp (Server Timestamp):** Records the exact time of submission, used for sorting tie-breakers (earlier score ranks higher).

6.2. Local Data Storage (SQLite - Monster Battler Module)

6.2.1. Purpose:

- **Static Data Management:** Stores immutable game assets such as Monster stats (HP, Atk, Def) and Skill definitions.
- **Data Integrity:** Ensures strict relationships between monsters and their elemental types or available skills.

6.2.2. Entity-Relationship Diagram (ERD): The local database schema consists of three main tables to handle the Many-to-Many relationship between Monsters and Skills.



7. New and Challenging Implementations

7.1. New Features & Implementation Approaches

Using Jetpack compose or new language like kotlin has definitely been a challenge for us at the start, especially when we have to get used to some newer approach on designing such as:

- **Hybrid Rendering Architecture (Compose + SurfaceView):**
    - Unlike traditional Android apps that rely solely on XML or solely on Jetpack Compose, our application implements a **Hybrid UI approach**.
    - We successfully embedded legacy **SurfaceView** components (optimized for high-frequency rendering) inside modern **Jetpack Compose** layouts using AndroidView wrappers. This allows us to leverage the modern UI/UX capabilities of Compose for menus and HUDs while maintaining raw performance (60 FPS) for intensive arcade game loops (Tetris, Arkanoid).
- **Unified "Game Hub" Ecosystem:**
    - The application functions not merely as a single game but as a scalable **Game Console Platform**. It features a centralized launcher that manages multiple distinct game modules.
    - The **Global Leaderboard System** is designed as a "Multi-tenant" service. A single Firebase collection structure serves all games simultaneously by dynamically filtering via game_id, creating a unified competitive ecosystem for users.
- **Hybrid Data Strategy (SQL + NoSQL):**
    - We implemented a sophisticated dual-database strategy. **SQLite** is used for complex, relational, and static RPG data (Monster stats, Skills) ensuring offline capability, while **Firebase Firestore** (NoSQL) is used for real-time social features (Leaderboards). This combines the data integrity of SQL with the scalability of the Cloud.

7.2. Challenges

- **Custom Physics Engine & Collision Detection:**
    - It was really tricky to figure what to do with the physic-based game at first because my only experience is with dedicated game engine such as godot and unity, but after spending some days and some more youtube tutorial the group have figure out what to do.
    - Challenge: Implementing arcade mechanics (Bubble Shooter, Arkanoid) without using game engines like Unity or LibGDX.

    - Solution: Raw mathematical algorithms for **vector calculation**, **trajectory prediction** (ball bouncing,..). Optimizing these calculations to run every 16ms without causing stutter was a major technical challenge.
- **Troubles in using specific function:**
    - While developing games that need a dedicated database to store various types of information. “Room” was first chosen, but the project ran into unexplained compilation and runtime issues. The solution was to switch to SQLite — less modern, but stable and it gets the job done.

7.3. Future Work

- **Log in/out, general account function:**
    - Make way for more advanced administrator function and other useful things
- **More refined firebase storage:**
    - To use alongside account, firebase must be more refined to use with more advanced security protocol..
- **Dynamic Content Delivery (DLC):**
    - More content for existing games, change appearance for some game, more interactable interface and new game in general
- **Social Features:**
    - With account function, we can also create a healthy social media tab that user could add friend, make a friend lists, chat systems,...

8. Conclusion

The project is created in hope to to not only meet the CLOs given but also to demonstrate our ability in creating a polished and entertaining software product.

Technically, the project is also our group's attempt in embracing modern Android development standards. By using Kotlin and Jetpack Compose, we can move away from .XML file dependencies, creating a flexible and reactive UI.

Throughout the development process, the team overcame significant challenges, from implementing custom physics engines without third-party game libraries to resolving database stability issues. These obstacles served as valuable learning opportunities, strengthening our skills in logic design, problem-solving, and team coordination.

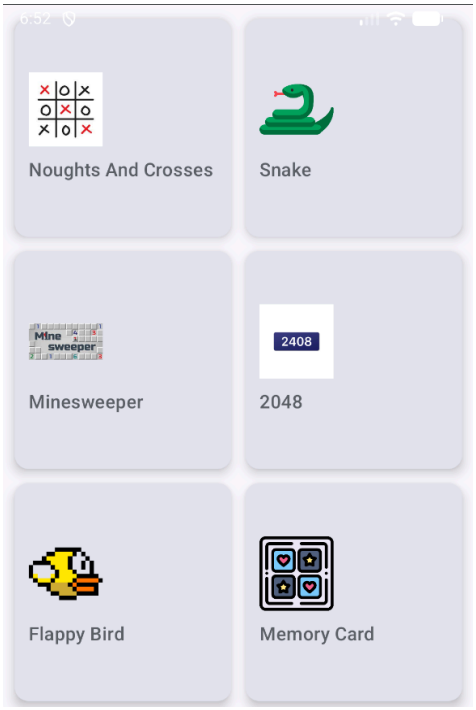
II. Individual result

1. Nguyễn Nhật Hưng

A. Main menu + game intro

Execution Flow:

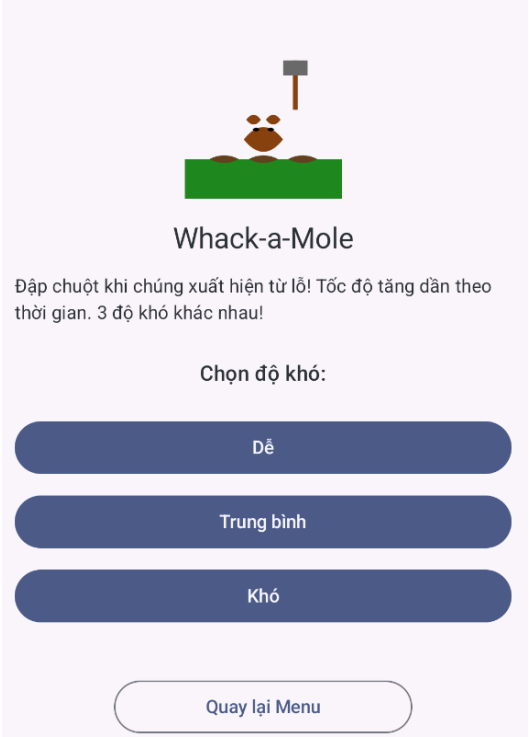
1. **Application Launch:** When the user opens the application, the system loads the MainActivity.
2. **Game List Display:** The system displays a list of all available games in a grid format consisting of two columns. Each grid item contains an icon and the name of the game.



Main game interface (Scroll down for more game)

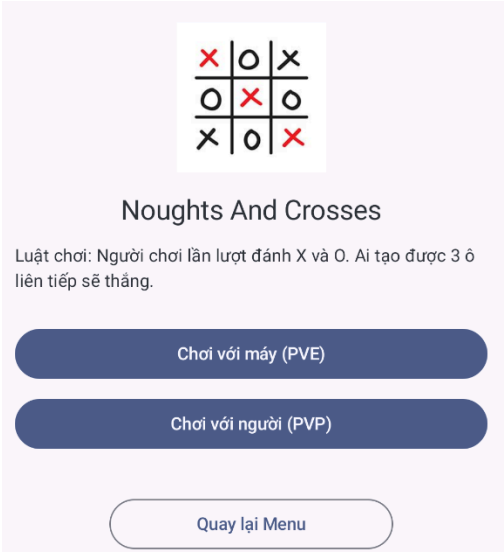
3. **Game Selection:** The user clicks on any game tile.
4. **Screen Transition:** The system captures the game information (Title, Rules, Image Resource, and the Target Class for the game screen) and transitions to the GameIntroActivity screen via an Intent.
5. **Intro Display:** The Intro screen displays detailed information and checks the game type to render the corresponding functional buttons:

- If the game has difficulty levels (e.g., Whack-a-Mole, Tetris): Display three buttons for difficulty selection (Easy, Medium, Hard).



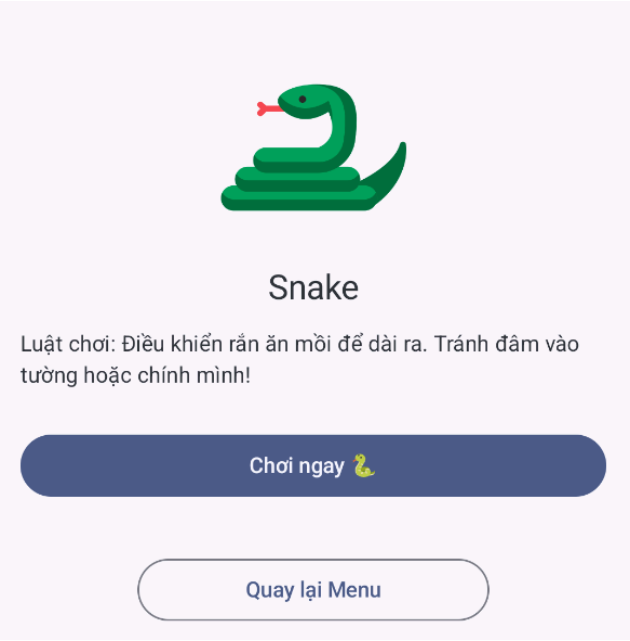
Intro screen for game with difficulties

- If the game is a versus game (e.g., Pong, NoughtsAndCrosses): Display options for Player vs. Environment (PVE) or Player vs. Player (PVP).



Intro for game with pvp and vs-bot gamemode

- If the game is a single-player game (e.g., Snake, Monster Battler, Arkanoid): Display a "Play Now" button with a specific icon.



Standard intro screen with 1 button

6. **Start Game:** When the user clicks a mode selection button, the system launches the Activity of the corresponding game.
7. **Return:** From the Intro screen, the user can press the "Back to Menu" button(if given) or just press the default return button on your phone to return to the initial game list.

Business Constraints:

- **Data Integrity:** Data transmitted between MainActivity and GameIntroActivity must be complete (non-null). If the destination class (targetClass) is not found, the system must handle the error gracefully rather than crashing.
- **Adaptability:** The interface must automatically adapt to the addition of new games to the list without requiring significant modifications to the layout structure.

Main Interface (MainActivity.kt):

- **Layout:** Utilizes a vertical grid (LazyVerticalGrid) with two fixed columns.
- **Visual Components:** Square Cards with elevation to create depth. Inside each card:
  - Game illustration image (Size: 64dp).

    - Game title centered below the image.
- **Spacing:** Items are separated from each other and the screen edges by a padding of 12dp.

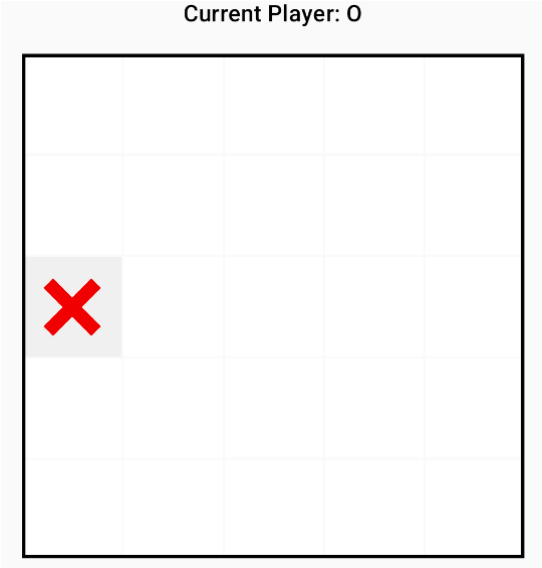
Introduction Interface (GameIntroActivity.kt):

- **Layout:** Columnar structure (Column), with elements centered both vertically and horizontally.
- **Visual Components (Top to Bottom):**
    1. Large representative image of the game (Size: 120dp).
    2. Game Title (Typography: TitleLarge).
    3. Brief Rules Description (Typography: BodyMedium).
    4. Control Button Area (Dynamically changes based on the game): Includes Buttons for game modes or difficulty levels.
    5. "Back to Menu" button (OutlinedButton) located at the bottom, visually separated from the functional buttons.

B. Noughts And Crosses

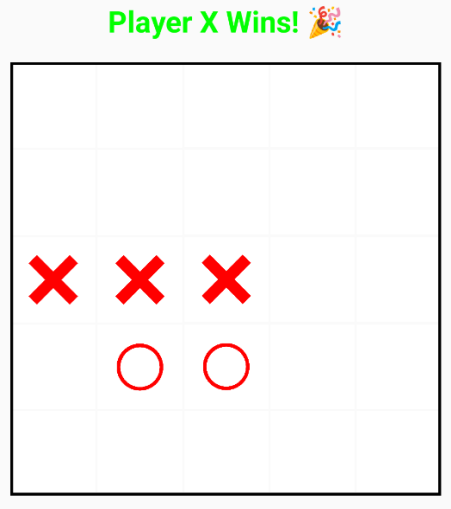
Execution Flow:

1. **Game Initialization:** From the intro screen, the game will behave accordingly to the gamemode, vs-bot or pvp.
2. **Player Turn(Applied to every gamemode):**
    - The player taps on an empty cell on the grid.
    - The system validates the move (checks if the cell is empty and if the game is still active).
    - If valid, the cell updates to "X"(go first condition), and the system checks for a win or draw condition.



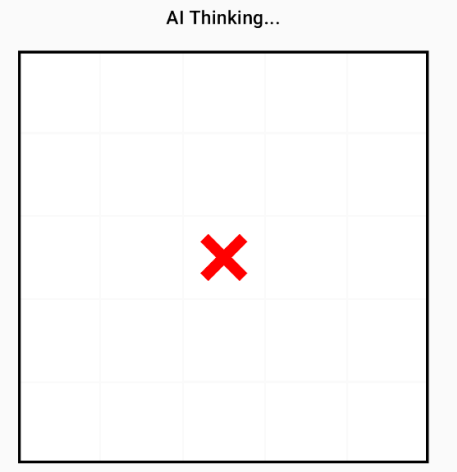
Player click on 11x3, system check condition and fill the blank

3. **Result Validation:** After every move, the system scans the entire board:
    - If there are 3 identical symbols in a row (horizontal, vertical, or diagonal): The current player is declared the winner.
    - If the board is full and no one has won: The game is declared a Draw.
    - If no result is reached: The turn switches to the next player.



Condition checked, confirm winner

4. **AI Turn (Only in PVE Mode):**
    - When it is "O's" turn, the system activates the Minimax algorithm.
    - The AI calculates the optimal move to either block the player's potential win or to build its own winning line.
    - The board is updated, and the system checks for a result similar to the human turn.



Bot is processing

5. **Game Over and Reset:** When a result is determined, the screen displays a notification (Win/Loss/Draw). The player can press "New Game" to clear the board and restart, or "Back" to return to the main menu.

Business Constraints:

- **Board Dimensions:** Fixed at 5x5 (25 cells).
- **Win Condition:** A sequence of 3 identical symbols (WIN_CONDITION = 3).
- **Turn Order:** Player (X) always moves first; the AI (O) moves second.
- **Performance:** The AI must generate a move within a reasonable time frame to ensure the user interface (UI) does not freeze.

Screen Layout (NoughtsAndCrossesActivity):

- **Header:** The title "Noughts and Crosses - 5x5" is displayed in bold at the top.
- **Status Information:** The current game mode (PVP/PVE) and win condition are displayed below the title.
- **Turn/Result Notification:**
    - During gameplay: Displays "Current Player: X" or "AI Thinking...".
    - Game Over: Displays a large status text with specific coloring (Green for X Win, Red for O/AI Win, Gray for Draw).
- **Game Board (Grid):**
    - Implemented using LazyVerticalGrid with 5 fixed columns.
    - Total size is approximately 350dp x 350dp.
    - Each cell is a white square with a black border.

    - "X" and "O" pieces are rendered using Images (Image composable) rather than text for better visual appeal.
- **Control Panel:**
    - "New Game" Button: Resets the game state.
    - "Back" Button: Gray-colored button to finish the activity and return to the menu.

Artificial Intelligence (AI) Algorithm:

- Implemented the **Minimax algorithm with Alpha-Beta Pruning**.
- **Purpose:** To traverse the game tree and find the optimal move (maximizing the AI's score while minimizing the opponent's score) while discarding irrelevant branches to improve speed.
- **Heuristic Evaluation:** Since a full traversal of a 5x5 board is computationally expensive, the algorithm uses a **Depth Limit** (MAX_DEPTH = 4) and a heuristic evaluation function. This function scores the board based on factors like controlling the center and forming chains of two symbols.

Concurrency: Utilizes **Kotlin Coroutines** (LaunchedEffect and withContext(Dispatchers.Default)). The heavy AI calculation runs on a background thread to prevent the main UI thread from blocking (lagging).

Challenges Encountered:

- At first the game is designed for quick X-O action but have limited strategy potential, the solution is to increase the board from 3x3 to 5x5, which lead to some tweak in the A.I system that was meant for 3x3 game at first
- The A.I speed is acceptable, processing speed varies between different opening moves from the player, after that it went smoothly from then on.

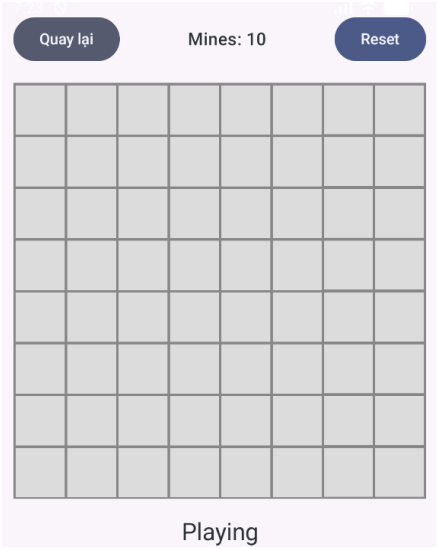
Future Development Potential:

- Existing development: Refined A.I system
- Potential development: None, game is too simple, the a.i is the most advance thing developing-wise that can be implement

C. Minesweeper

Execution Flow:

1. **Game Start:** Upon opening the Minesweeper screen, the system initializes an 8x8 grid containing 10 hidden mines placed randomly.

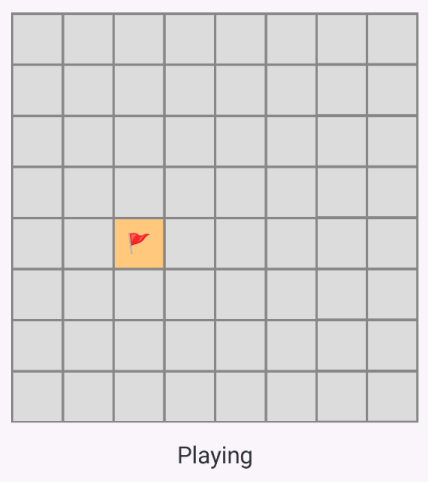


Game screen, mines is place randomly

2. **Player Interaction:**
    - **Single Tap (Reveal):** The player taps a cell to reveal it.
        - If the cell is a **Mine**: The game ends immediately ("Game Over"), and all mines on the board are revealed.

Press on a mine = game over

        - If the cell is **Empty (0 neighbors):** The system automatically reveals that cell and recursively reveals all adjacent empty cells (Flood Fill effect).
        - If the cell is a **Number:** The cell reveals a number indicating how many mines surround it (1-8).
    - **Long Press/Hold (Flag):** The player holds down on a hidden cell to place a Flag, marking it as a suspected mine. Long-pressing again removes the flag.



Flag placed

3. **Win Condition:** The game tracks the number of revealed cells. The player wins when all safe cells are revealed using the flag.
4. **Reset:** The player can press the "Reset" button at any time to generate a new board with randomized mine positions.

Business Constraints:

- **Grid Size:** Fixed at 8 rows x 8 columns.
- **Mine Count:** Fixed at 10 mines.
- **Safety Rule:** A flagged cell cannot be revealed by accident (Single Tap is disabled on flagged cells).
- **Boundary Logic:** The neighbor calculation must handle corners and edges correctly to avoid IndexOutOfBoundsException.

Screen Layout (MinesweeperScreen):

- **Header Bar:**
    - "Back" Button (Left): Returns to the main menu.

    - "Mines Left" Counter (Center): Displays the number of mines minus the number of placed flags.
    - "Reset" Button (Right): Restarts the game.
- **Game Grid:**
    - Implemented using LazyVerticalGrid with 8 fixed columns.
    - The grid maintains a strictly square aspect ratio (aspectRatio(1f)).
    - **Cell Styling:**
        - **Hidden:** Light Gray background (0xFFE0E0E0).
        - **Revealed:** Darker Gray background (0xFFC0C0C0) to distinguish from hidden cells.
        - **Flagged:** Orange-tinted background (0xFFFFCC80) with a flag icon.
        - **Numbers:** Color-coded text based on danger level (1: Blue, 2: Green, 3: Red, 4: Dark Blue, etc.).
        - **Mines:** Displayed using a bomb emoji.
- **Footer Status:**
    - Displays the current game state text (e.g., "Playing", "You Win! 🎉", "Game Over 💥") centered at the bottom using TitleLarge typography.

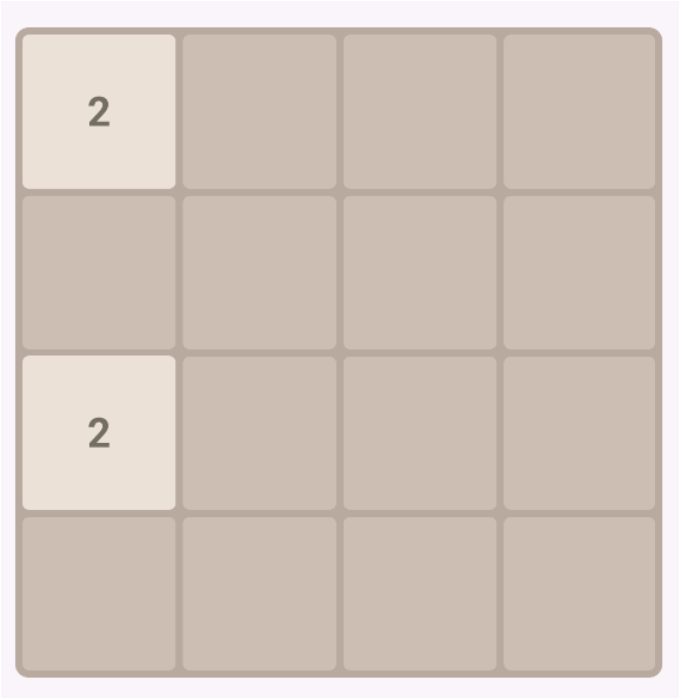
Technical Solution for Function Construction

- **Architecture:** The game data class is immutable; every action (reveal, flag) returns a *copy* of the game object with the updated board state. This ensures thread safety and predictable UI updates in Jetpack Compose.
- **Algorithm - Recursive Flood Fill:**
    - The core logic lies in the revealEmptyCells function.
    - When a cell with 0 surrounding mines is clicked, the algorithm checks all 8 neighbors. If a neighbor is also empty and unrevealed, the function calls itself recursively. This creates the satisfying "opening up" effect characteristic of Minesweeper.
- **Data Structure:** The board is represented as List<List<MinesweeperCellState>>. Each cell stores its own properties: isMine, isRevealed, isFlagged, and minesAround.

D. 2048

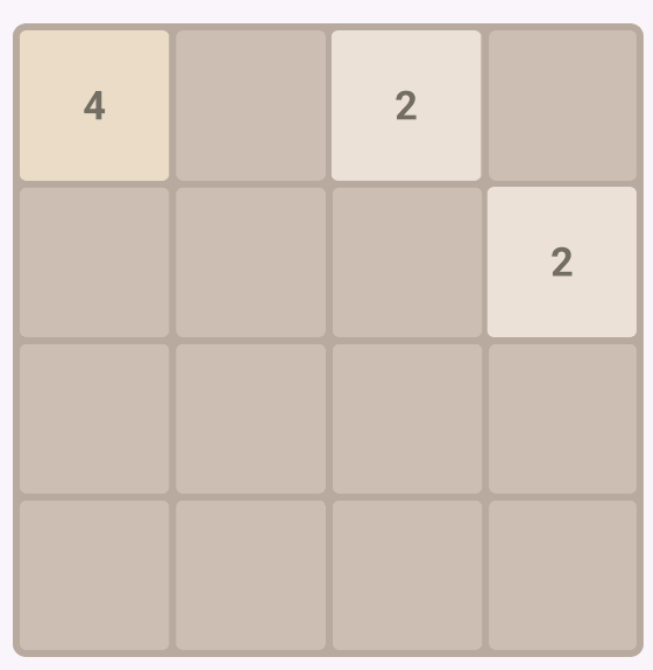
Execution Flow:

1. **Game Initialization:** The game starts by creating a 4x4 grid. Two tiles with a value of "2" are spawned at random positions.



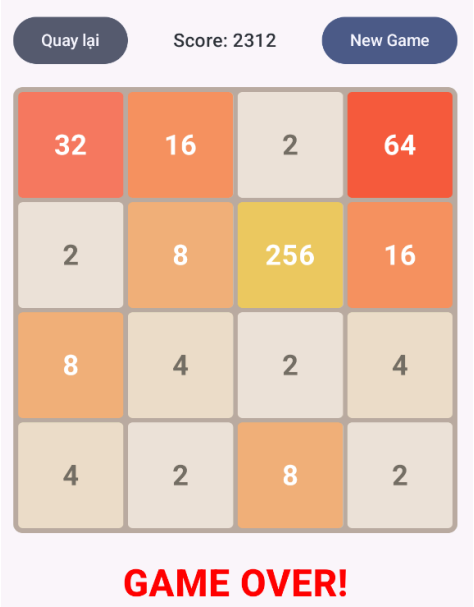
Game screen, number pre-generated at random position

2. **Player Interaction (Movement):**
    - The player swipes the screen (Up, Down, Left, Right) or uses the on-screen direction buttons.
    - **Sliding:** All tiles slide as far as possible in the chosen direction until they hit the wall or another tile.
    - **Merging:** If two tiles of the *same value* collide during the slide, they merge into a single tile with double the value (e.g., 2+2=4). The score increases by the value of the new tile.
    - **Spawning:** If the board state changes (i.e., tiles moved or merged), a new tile (value 2 or 4) appears at a random empty location.



Player swiped up, merging 2 identical number, the game generate random number at the same time

3. **Win Condition:** The game checks after every move if a tile with the value **2048** exists. If found, a "YOU WIN! 🎉" message is displayed, but the player can continue playing.
4. **Game Over Condition:** The game ends when the board is full (16 tiles) and no legal moves are possible (no adjacent tiles have the same value). A "GAME OVER!" message appears.



Unable to continue calculating in any direction

5. **Reset:** The "New Game" button allows the player to restart the session, resetting the score to 0 and clearing the board.

Business Constraints:

- **Grid Size:** Fixed at 4x4.
- **Merging Rules:** A tile can only merge once per turn.
- **Spawn Probability:** New tiles are usually "2" (90% chance) and occasionally "4" (10% chance).
- **Animation:** Tile movements must be animated smoothly to allow the user to track the flow of the game.

Screen Layout (Game2048Screen):

- **Header:**
  - **Title:** "2048" displayed prominently in bold (HeadlineLarge).
  - **Control Bar:** Contains the "Back" button (left), current "Score" (center), and "New Game" button (right).
- **Game Board Area:**
  - A square container with a light brown background (#BBADA0) and rounded corners.
  - The grid contains 16 slots separated by padding (4dp).
- **Tiles (TileView):**
  - **Colors:** Each number has a distinct background color (e.g., 2 is light beige, 8 is orange, 2048 is gold).
  - **Text:** Dark text for small numbers (2, 4) and White text for larger numbers (8+) for readability.
  - **Animations:**
    - **Movement:** Tiles glide to their new positions using animateDpAsState with an EaseOut curve.
    - **Spawn:** New tiles appear with a "pop" effect using a Spring animation (Spring.DampingRatioMediumBouncy).
- **Footer:**
  - Status messages (Win/Game Over) appear below the board.
  - Directional buttons are provided at the bottom as an alternative input method for users who prefer tapping over swiping.

Description of Solution and Techniques:

- **Architecture:** The game data class holds the entire snapshot of the game (tiles, score, status). Functions like move and addNewTile are pure functions that take the current state and return a *new* state, ensuring thread safety and predictable UI rendering.
- **Grid Logic - Matrix Transformation:**
    - Instead of writing four separate logic blocks for Up/Down/Left/Right, the solution uses coordinate mapping. The traversals list defines the order in which cells are processed based on the direction.
    - The slideAndMerge function processes a single line (row or column), handling the mathematical logic of merging identical values.
- **Animation System:**
    - **Separation of Logical and Visual Position:** The logical grid (row/col in data) updates instantly. However, the UI (TileView) uses animateDpAsState to interpolate the pixel position from the old coordinates to the new ones over 150ms.
    - **Globally Positioned Layout:** Used onGloballyPositioned to dynamically calculate the exact pixel size of the game board, ensuring the grid renders correctly on screens of different densities (DPI).
- **Gesture Handling:** Utilizes detectDragGestures within a pointerInput modifier to calculate swipe direction based on the X and Y drag distance. A minimum threshold (minDragDistance) prevents accidental swipes.

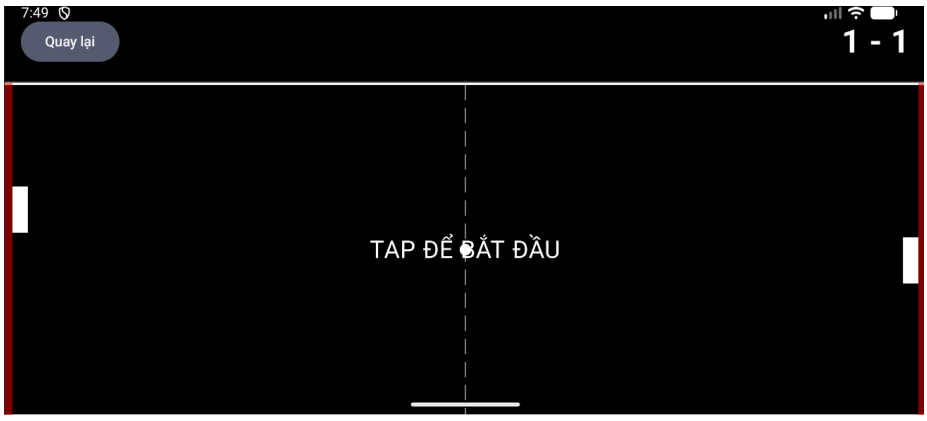
Challenges Encountered:

- Even making simple animation like sliding prove to be a great challenge due to unfamiliarity with the android engine
- Solution: The exact size of a single tile and the spacing between them are dynamically calculated. Using these coordinates, animateDpAsState automatically interpolates the value from the old offset to the new offset over a duration of 150ms, creating the visual illusion of sliding

E. Pong

Execution Flow:

1. **Game Setup:** Receive gamemode from intro screen to use gamemode accordingly.
2. **Gameplay Mechanics:**
    - **Start:** The game enters a "Ready" state. Tapping the screen launches the ball with a random velocity vector(random angle).
    - **Paddle Movement:**
        - **PVE Mode:** The player drags vertically on the screen to control the left paddle. The right paddle is controlled by an AI script that tracks the ball's Y-position.
        - **PVP Mode:** The screen is conceptually divided into two halves. Dragging on the left half controls Player 1; dragging on the right half controls Player 2.
    - **Physics & Collision:**
        - **Walls:** If the ball hits the top or bottom edge, its vertical velocity is inverted.
        - **Paddles:** When the ball hits a paddle, its horizontal velocity is inverted. Crucially, the **angle of reflection** changes based on where the ball hits the paddle (hitting the edges creates a sharper angle). The ball's speed also increases by 10% (SPEED_INCREASE_FACTOR) after every paddle hit to escalate difficulty.
3. **Scoring:**
    - If the ball passes a paddle and enters the "Goal Zone" (Red Zone), the opponent scores a point.
    - The ball resets to the center, and the game pauses briefly before the next round.



Ball reset to the middle after one player score, score board also get updated

4. **Win Condition:** The first side to reach **7 points** (WIN_SCORE) wins the match. A victory message is displayed, and tapping resets the scores for a new game.

Business Constraints:

- **Paddle Constraints:** Paddles must not move beyond the top or bottom screen boundaries.
- **Speed Limits:** The ball speed starts at a manageable level but scales up every time it bounces off. The AI speed is capped (AI_PADDLE_SPEED) so it is beatable.
- **Fairness:** In PVP mode, the input system must differentiate between touches on the left vs. the right side.

Screen Layout (PongScreen):

- **Header:**
    - "Back" Button (Top-left).
    - **Scoreboard:** Large, bold text (e.g., "3 - 5") centered at the top, displaying the current match state.
- **Game Canvas (The Arena):**
    - **Background:** Deep Black (Color.Black) for a retro arcade aesthetic.
    - **Center Line:** A dashed white line running vertically down the middle to visually separate the courts.
    - **Goal Zones:** Semi-transparent red strips at the far left and right edges to indicate danger zones.
    - **Game Objects:**
        - **Paddles:** White rectangles (PADDLE_WIDTH x PADDLE_HEIGHT).
        - **Ball:** A white circle (BALL_RADIUS).
- **Overlays:**
    - **"TAP TO START":** Displayed when the game is waiting for the user to serve.
    - **Victory Text:** Large Green (Player Win) or Red (AI Win) text displaying the result at the end of the match.

Description of Solution and Techniques:

- **Architecture: Real-Time Game Loop** pattern implemented within Jetpack Compose.
    - Continuous updates using LaunchedEffect with a while(true) loop that calculates the **Delta Time (dt)** between frames.
    - **Physics Update:** The updateGamePhysics function moves the ball by Velocity * dt every frame. This ensures the game runs at the same speed on both slow and fast devices (frame-rate independence).
- **Geometry & Trigonometry:**

- **Reflection Math:** Instead of simple 45-degree bounces, I implemented a dynamic angle system. The reflection angle is calculated using the formula:

$$\text{Angel} = \frac{\text{PaddleCenter(y)} - \text{Ball(y)}}{\text{PaddleHeight/2}} \times \text{MaxAngle}$$

Challenges Encountered:

- **The "Tunneling" Effect:** At high speeds, the ball moves so many pixels per frame that it might "skip" over the paddle entirely (passing through it without triggering a collision).
  - Solution: While a full continuous collision detection (CCD) was too complex, I mitigated this by capping the maximum ball speed and ensuring the paddle thickness (PADDLE_WIDTH) is sufficient relative to the max step size.
- **Performance vs. State:** Updating the Compose State 60 times a second triggers frequent recompositions. Optimizing the Canvas drawing block to be lightweight was crucial to prevent the UI from stuttering (dropping frames).

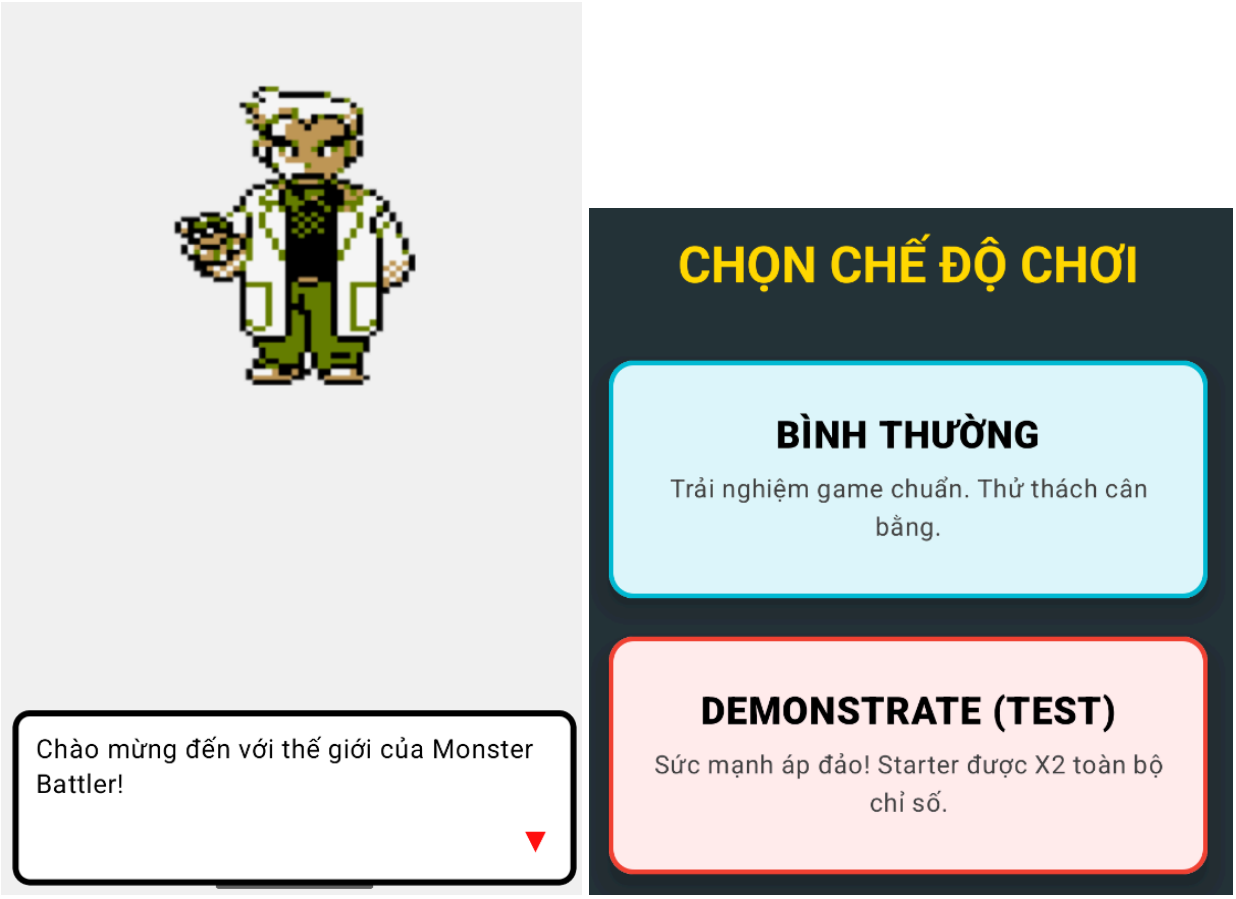
Future Development Potential

- **Power-ups:** Spawn items on the field that enlarge the paddle or slow down time when hit by the ball.
- **Network Multiplayer:** Upgrade the PVP mode from local (same screen) to online (using Firebase or WebSockets) so two users can play on separate devices.

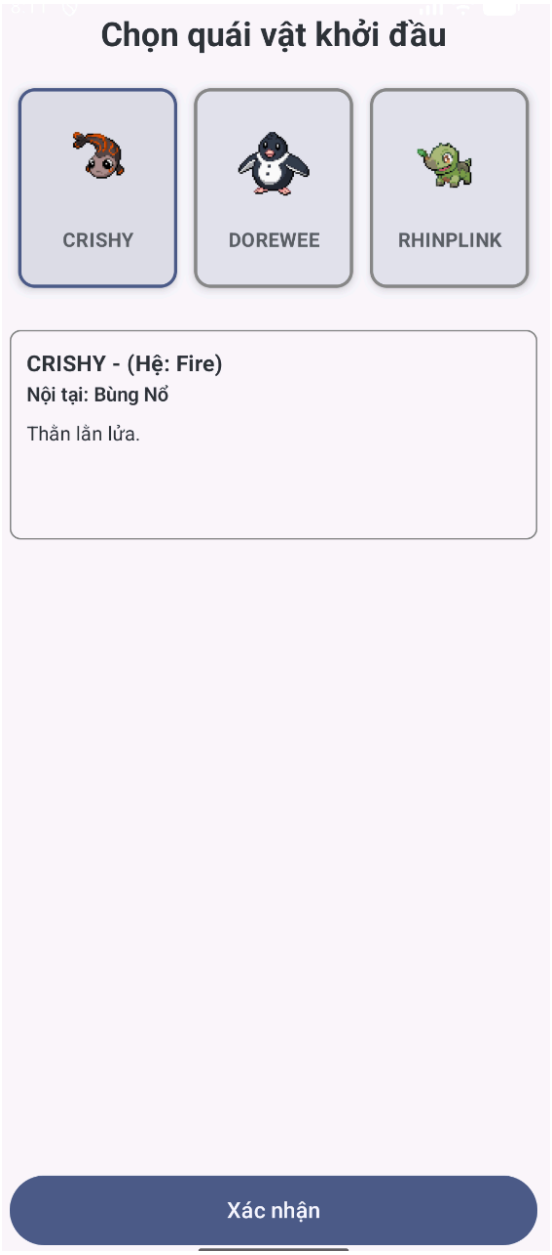
F. MonsterBattler

Execution Flow:

1. **Initialization:** The game begins with an **Intro Screen** featuring narrative dialogue. The player then selects a Game Mode (Normal or Demonstrate) and chooses a "Starter Monster" from a pool of available creatures.

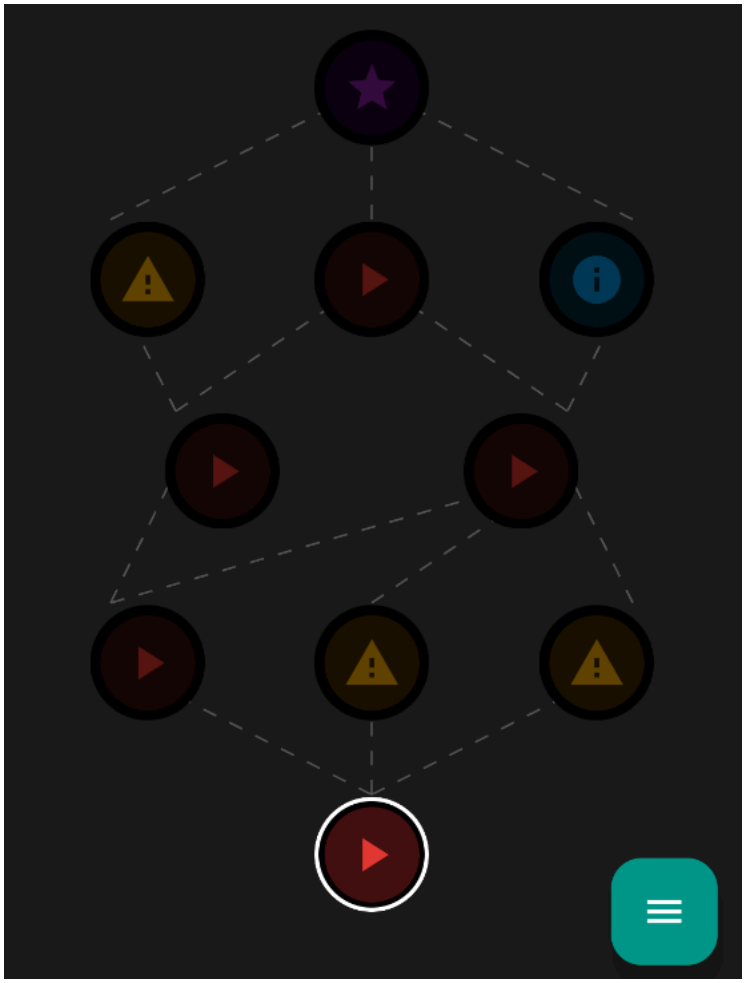


After the intro screen, the game transition to gamemode screen



Starter monster select screen, confirm to pick

2. **Gauntlet Map System:**
    - The core loop revolves around a procedurally generated map (GauntletMapScreen) consisting of 5 levels.
    - **Node Types:**
        1. **Battle(red):** Standard combat against wild monsters.
        2. **Elite(yellow):** Harder combat with buffed enemies (+20% stats).
        3. **Rest(blue):** A text-based event where the player can rest (Heal 10%) or gamble HP in a Dice Roll minigame (Win: Heal 50% / Lose: Damage 10%).
        4. **Boss(Purple):** The final node of the map (Level 5) with a significantly stronger enemy (+30% stats).

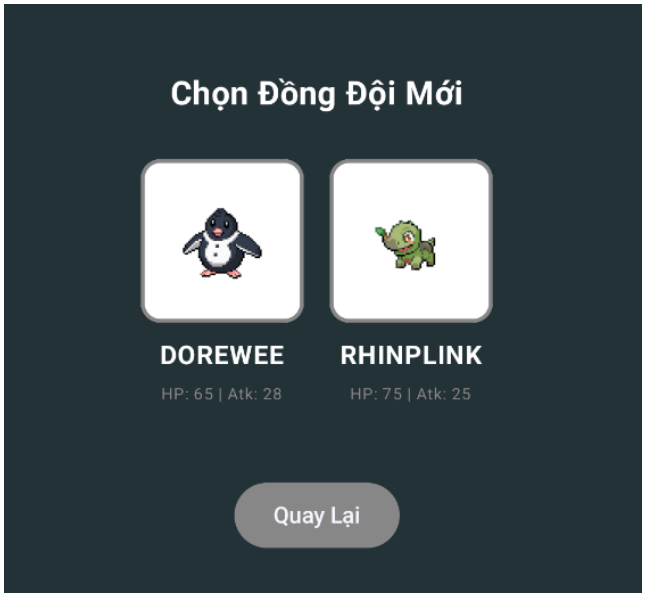


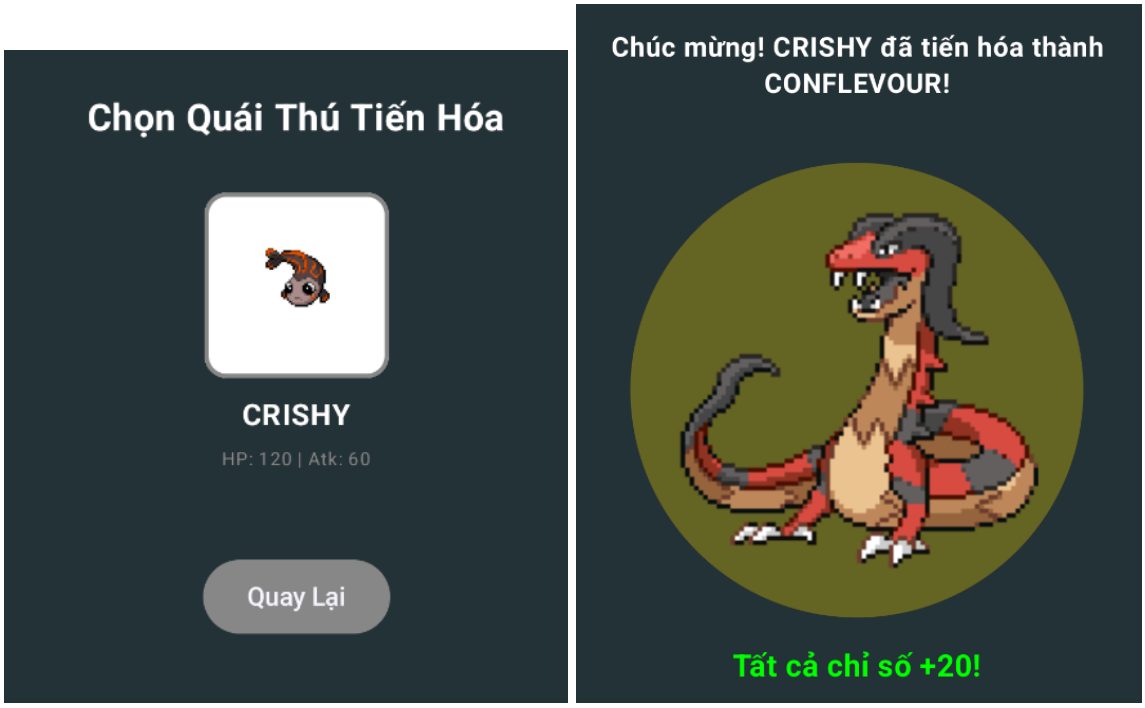
Slay the spire-inspired map screen

3. **Combat System (BattleScreen):**
    - **Turn-Based Logic:** The player selects a Skill or Switches monsters. The Player attacks first, followed by the Enemy AI.
    - **Mechanics:** Damage is calculated based on Attack, Defense, Skill Power, and **Elemental Type Effectiveness** (Fire > Leaf > Water > Fire).
    - **Status Effects:** Skills can inflict effects like **Burn** (DoT), **Stun** (Skip turn), **Break Def** (Halve Defense), or **Weaken** (Halve Attack).
4. **Progression & Evolution:**
    - **Victory Rewards:** After winning a battle, the player drafts a reward: Stat Upgrade, Heal, or a Passive Buff (e.g., "Any attack has 10% chance to Burn").

You can pick 1 upgrade after normal fight and 2 after elite fight

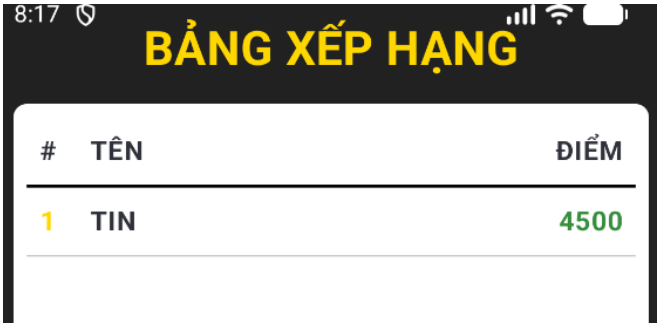
- **Major Upgrade:** Defeating a Boss triggers a major event allowing the player to:
    1. **Evolve:** Transform a monster into a stronger form (e.g., Crishy → Conflevour) with +20 to all stats.
    2. **Recruit:** Add a new monster to the team.
    3. **Empower:** Grant +10 stats to a current member.





Major upgrade screen examples

5. **Game Over & Leaderboard:** If the entire team is defeated, the run ends. The total score is calculated and uploaded to **Firebase Firestore**. Users can view the global Top 10 rankings, you can see the leaderboard every time you are not in a fight



Business Rules:

- **Permadeath (Roguelite):** If a monster's HP reaches 0, it "faints" and cannot be used until revived (currently only via specific rare events). If the whole team faints, the game is over.
- **Move Limitations:** Skills consume PP (Power Points). If PP hits 0, the skill cannot be used.

- **Evolution Logic:** Evolution is restricted to specific species defined in the EVOLUTION_MAP (e.g., only Rhinplink can evolve into Rhitain).

Map Interface (GauntletMapScreen):

- **Layout:** A vertical scrolling list representing dungeon levels.
- **Visuals:** Nodes are circular icons connected by lines.
    - **Active Path:** White solid lines indicating accessible nodes.
    - **Locked Path:** Grey dashed lines indicating inaccessible routes.
- **Feedback:** The current node is highlighted with a green border and a pulsing animation.

Battle Interface (BattleScreen):

- **Split Layout:**
    - **Top-Right:** Enemy sprite, HP Bar, and Status Icons.
    - **Bottom-Left:** Player back-sprite, HP Bar, and active Buff indicators.
- **Control Panel:** Located at the bottom (0xFF2D3436 background). It dynamically switches between:
    - **Menu:** "Attack" and "Switch Monster" buttons.
    - **Skill Select:** A grid of 4 buttons displaying Skill Name, Type (colored text), and remaining PP.
    - **Log:** A text box describing battle events (e.g., "Critical Hit!").
- **Animations:** Uses Animatable for attack lunges (sprites move forward/backward) and flashing effects for damage.

Mystery & Minigames (MysteryScreen):

- **Dice Game:** Displays two large dice boxes (Player vs. Enemy). When rolling, the numbers shuffle rapidly before settling on a result.
- **Colors:** Uses Blue (0xFF0277BD) for the theme to distinguish it from combat.

Leaderboard (LeaderboardScreen):

- **Data Presentation:** A clean LazyColumn list. The Top 3 players are highlighted in Gold, Silver, and Bronze colors respectively.

Technical Solution for Function Construction

Description of Solution and Techniques:

- **Architecture:** The app follows the **MVVM (Model-View-ViewModel)** pattern.

    - GameViewModel: Manages global state (Map progression, Team roster, Inventory).
    - BattleViewModel: Manages the temporary state of a specific combat encounter (Turn order, HP updates, UI logs).
- **Map Generation Algorithm:** The GauntletMapGenerator creates a directed acyclic graph (DAG).
    - **Orphan Fix Logic:** A specific algorithm ensures every node in the next level has at least one parent in the previous level. If a node is "orphaned" (unreachable), the generator forces a connection to the nearest available parent index.
- **Data Persistence:**
    - **Local:** MonsterDbHelper acts as a static repository for Monster stats and Skill data.
    - **Cloud: Firebase Firestore** is used for the Leaderboard. The LeaderboardManager object handles asynchronous read/write operations for high scores.
- **Battle Mechanics Engine:** The BattleMechanics object encapsulates pure functions for logic.
    - *Formula:* Damage = (Attacker.Atk * BuffMultiplier + Skill.Power) - (Defender.Def * DebuffMultiplier / 2).
    - *AI Logic:* A simple heuristic AI checks HP; if HP < 30%, it prioritizes Healing skills; otherwise, it chooses a random offensive move.

Challenges Encountered:

- **Map Connectivity:** Initially, the random map generator created "unreachable islands" where a player could get stuck. Implementing the "reverse-check" logic (scanning from child to parent) in GauntletMapGenerator solved this.
- **Balancing:** Some match-up is extremely hard, but after the first gauntlet everything is a cake-walk.
- When developing “monster battle”, i need a dedicated database to stored various type of info, between “room” and “sqlite”, i have chosen “room” at first, but doing so, the project have caught some difficult compiling and running, the reason until this day is still unexplainable
- Solution: Change to sqlite, less modern compare to room but it gets the job done

Future Development Potential

- **Balancing:** Changing stat for monsters, changing damaging formula

- **Inventory System:** Adding consumable items (Potions, Revives) to the loop.
- **More attack type:** Adding more types (Electric, Earth, Wind) to create a more complex "Rock-Paper-Scissors" combat web.

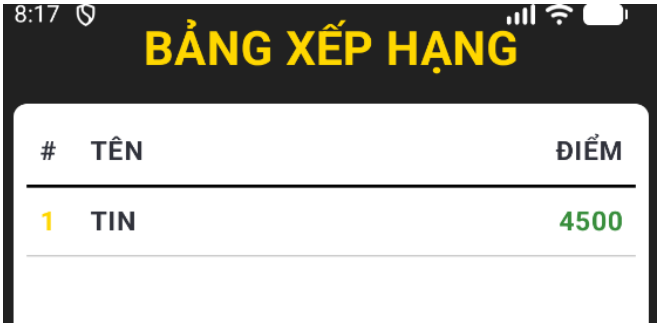
G. Leaderboard

Execution Flow:

1. **Trigger Condition:** The flow initiates immediately when a specific "Game Over" condition is met in any mini-game (e.g., HP reaches 0 in *Monster Battler*, Board full in *2048*, or Time Up in *Whack-A-Mole*).
2. **Data Capture:** The system captures the final integer score achieved during the session.
3. **User Input:** A modal dialog (SaveScoreDialog) appears, prompting the user to enter a display name.
    - **Validation:** The system enforces a constraint where the name is restricted to **3 uppercase characters** (e.g., "AAA", "ABC") to maintain a retro arcade aesthetic and layout consistency.

4. **Data Submission:** Upon clicking "Save", the system creates a data object containing the Name, Score, and Server Timestamp, then transmits it to the **Firebase Cloud Firestore**.
5. **Retrieval & Display:**
    - Immediately after saving, or when accessing the "Leaderboard" button from the menu, the system queries the database.
    - It retrieves the **Top 10 highest scores**, sorted in descending order.

- The data is rendered into a list on the LeaderboardScreen.



Business Constraints:

- **Multi-tenancy:** The system must handle scores for multiple different games simultaneously without data overlap. This is managed by passing a unique gameId (e.g., "monster_battler", "bubble_shooter") during the save/load process.
- **Real-time Synchronization:** Score updates must be reflected immediately across devices without requiring a manual refresh.

Save Score Dialog (SaveScoreDialog):

- **Type:** Modal Alert Dialog (pop-up).
- **Components:**
  - **Header:** "GAME OVER" (Red/Bold).
  - **Score Display:** "Your Score: [X]".
  - **Input Field:** A text box strictly limited to 3 characters, automatically capitalizing input.
  - **Actions:** "Cancel" (Gray) and "Save Score" (Primary Color).

Leaderboard Screen (LeaderboardScreen):

- **Layout:** A vertical list (LazyColumn) on a dark background (0xFF212121).
- **Row Item Structure:**
  - **Rank (#):** The position number. The top 3 ranks are highlighted with specific colors: **Gold (#1), Silver (#2), Bronze (#3)**.
  - **Name:** The 3-character player tag.
  - **Score:** The numeric score aligned to the right (Green color).
- **Loading State:** A circular progress indicator is displayed while data is being fetched asynchronously.

Technical Solution for Function Construction

- **Backend: Google Firebase Cloud Firestore** (NoSQL Database) was selected over SQL solutions because of its document-based structure and native real-time capabilities.
- **Pattern:** The LeaderboardManager is implemented as a **Singleton Object**. This decouples the database logic from the UI, allowing any game activity (Activity) to call LeaderboardManager.saveScore() or LeaderboardManager.getTopScores() without knowing the underlying implementation details.
- **Data Structure:** The data is organized hierarchically: root_collection("leaderboards") -> document(game_id) -> sub_collection("scores") -> document(entry). This structure allows for scalability where new games can be added simply by defining a new game_id string.

Future Development Potential

- **User Authentication:** Implement log in/out function at start, removing the need to input name after game over

2. Nguyễn Minh Đức

A. Bubble Shooter

a. Detailed Function Description

This game is a variant of the classic Bubble Shooter/Puzzle Bobble genre, developed on the Android platform using **SurfaceView** with a custom game thread for rendering and game logic.

Core Functionality:

- **Gameplay (Bubble Matching):** The player controls an aiming mechanism at the bottom of the screen to shoot colored bubbles upward. Bubbles attach to the hexagonal grid at the top. When 3 or more bubbles of the same color connect, they are destroyed.
- **Objective:** Clear bubbles by matching colors and prevent them from reaching the bottom of the screen.
- **Scoring System:** The player gains 100 points per bubble destroyed. Additional bonus points are awarded for floating clusters that fall due to disconnection from the ceiling.
- **Power-Up System:** The game includes two special bubble types: Bomb Bubble (destroys all neighboring bubbles) and Rainbow Bubble (automatically matches the most common neighboring color). Each power-up has a limited quantity (3 per game).
- **Progressive Difficulty:** Every 3 shots, a new row of bubbles is added from the top, pushing existing bubbles downward.
- **Leaderboard System:** The game integrates with Firebase Realtime Database for global score tracking. After the game is over, players can save their scores with a custom name. The leaderboard displays top scores for competitive gameplay.

Execution Flow

1. **Startup:** The application launches, BubbleShooterActivity initializes, BubbleShooterScreen composable is displayed, BubbleShooterView (SurfaceView) is created.

2. **Initialization State:** The game initializes the hexagonal grid (12 columns × 16 rows), fills the top half with random colored bubbles, and prepares the player's current and next bubbles.

3. **Ready State (GameState.READY):** The player can aim by touching the screen above the shooter. The aiming line updates in real-time based on touch position.

4. **Shooting Action:** When the player releases touch (ACTION_UP), the bubble is launched in the aimed direction.

5. **Shoot Bubble State (GameState.SHOOT_BUBBLE):** The bubble travels at 1800 pixels/second. Wall bouncing is handled by reversing the horizontal velocity. Collision detection checks intersection with existing bubbles using circle-circle intersection.

6. **Snap and Cluster Detection:** When the shooting bubble collides or reaches the top, it snaps to the nearest grid position. The cluster-finding algorithm (BFS) identifies connected bubbles of the same color.

7. **Remove Cluster State (GameState.REMOVE_CLUSTER):** If cluster size ≥ 3, bubbles fade out with alpha animation. Floating cluster detection identifies bubbles no longer connected to the ceiling. Floating bubbles fall with gravity simulation and fade out.

8. **Turn Progression:** Turn counter increments. Every 3 turns, a new row is added from the top.

9. **Game Over State (GameState.GAME_OVER):** Triggered when any bubble in the bottom row exists. The onGameOver callback is invoked via Handler on the main thread, passing the final score to trigger the save score dialog.

10. **Score Saving Flow:** After game over, SaveScoreDialog appears allowing the player to enter their name. Upon confirmation, the score is saved to Firebase Realtime Database under the game ID "bubble_shooter".

11. **Leaderboard Display:** After saving, LeaderboardScreen displays showing top scores fetched from Firebase, sorted by score in descending order.

Business Constraints

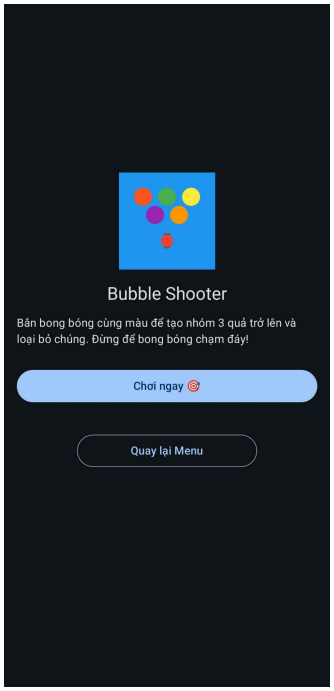
- **Grid Dimensions:** The bubble grid is fixed at 12 columns × 16 rows with hexagonal offset positioning.
- **Bubble Speed:** Shooting bubble travels at a constant speed of 1800 pixels/second.

- **Aiming Angle Limits:** The player's aiming angle is constrained between 10° and 170° to prevent horizontal or downward shooting.
- **Cluster Minimum:** A minimum of 3 connected same-colored bubbles is required for destruction.
- **Power-Up Limits:** Bomb and Rainbow power-ups are limited to 3 uses each per game.
- **Row Addition Frequency:** A new row of bubbles is added every 3 shots to increase difficulty progressively.
- **Color Consistency:** New bubbles (both for player and new rows) are randomly selected only from colors currently existing on the grid to ensure playability.
- **Leaderboard Integration:** Game uses Firebase Realtime Database with game ID "bubble_shooter" for score persistence. Score saving is triggered only once per game session using the hasTriggeredGameOver flag.

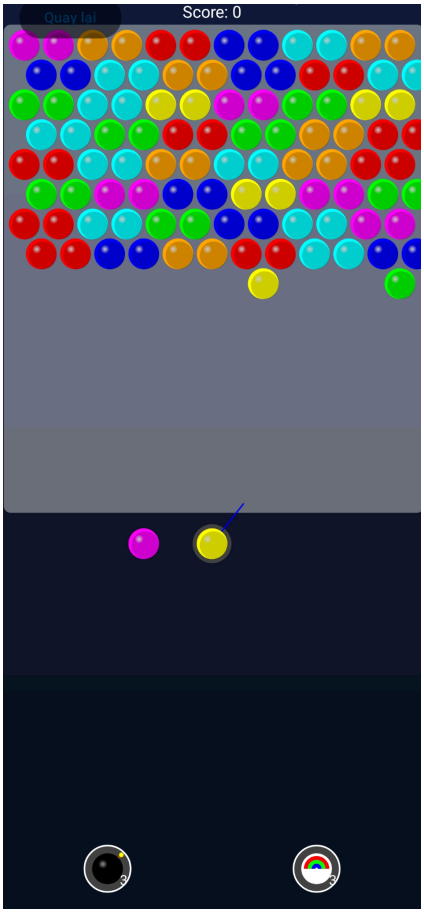
b. Interface Design

Game Canvas

- **Background:** Linear gradient from dark blue to darker blue, creating a deep space atmosphere.



- **Play Area:** Semi-transparent white rounded rectangle containing the bubble grid.



- **Bubbles:** 7 distinct colors: Red, Green, Blue, Yellow, Magenta, Cyan, and Orange. Each bubble features 3D rendering with radial gradient highlights and shadow effects.
- **Special Bubbles:** Bomb Bubble: Black sphere with gray highlight, brown fuse, and yellow spark. Rainbow Bubble: White sphere with concentric rainbow arcs (Red, Green, Blue).
- **Player Shooter:** Dark gray circular base with the current bubble centered. The blue aiming line extends from the center in the aimed direction. The next bubble displayed to the left of the shooter.

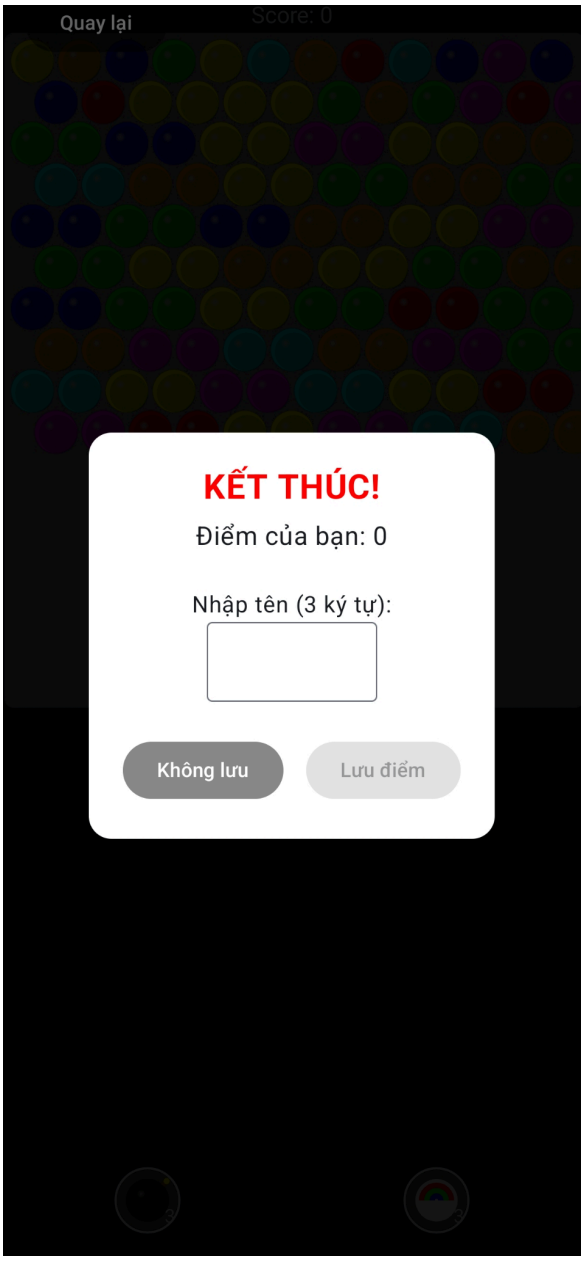
HUD (Heads-Up Display)

- Score display at the top center (white text).
- "Quay lại" (Back) button at the top-left corner with semi-transparent black background.

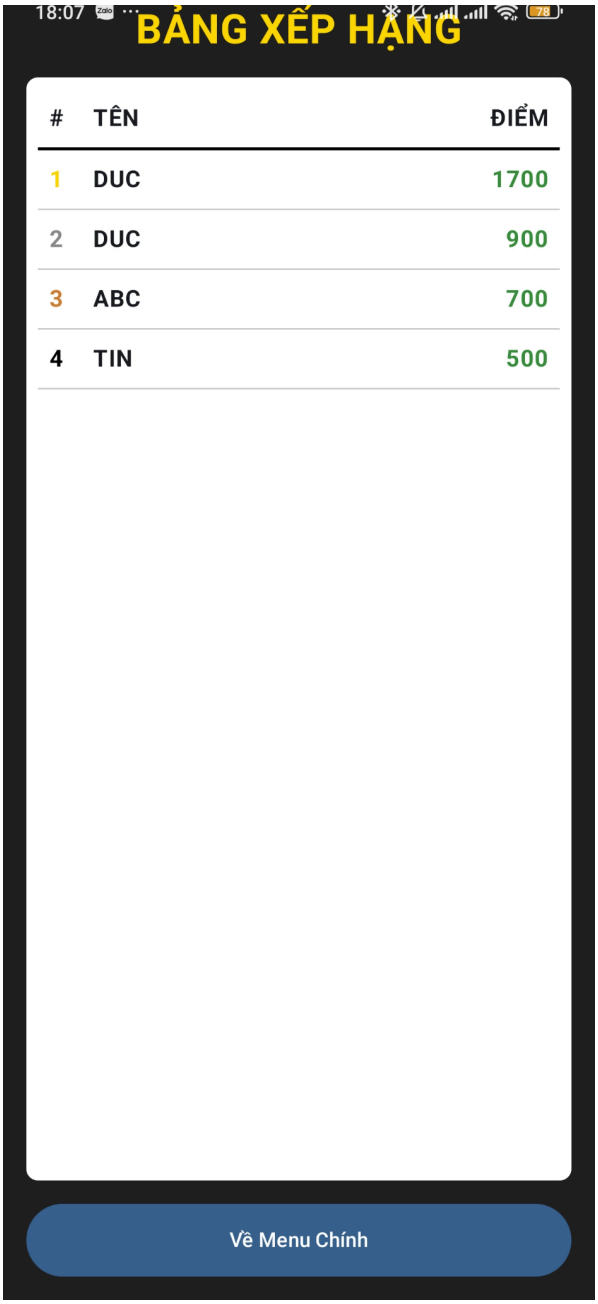
- Two power-up buttons at the bottom of the screen: Bomb (left side) with quantity indicator. Rainbow (right side) with quantity indicator. Buttons become semi-transparent when quantity reaches 0.

Status Screens

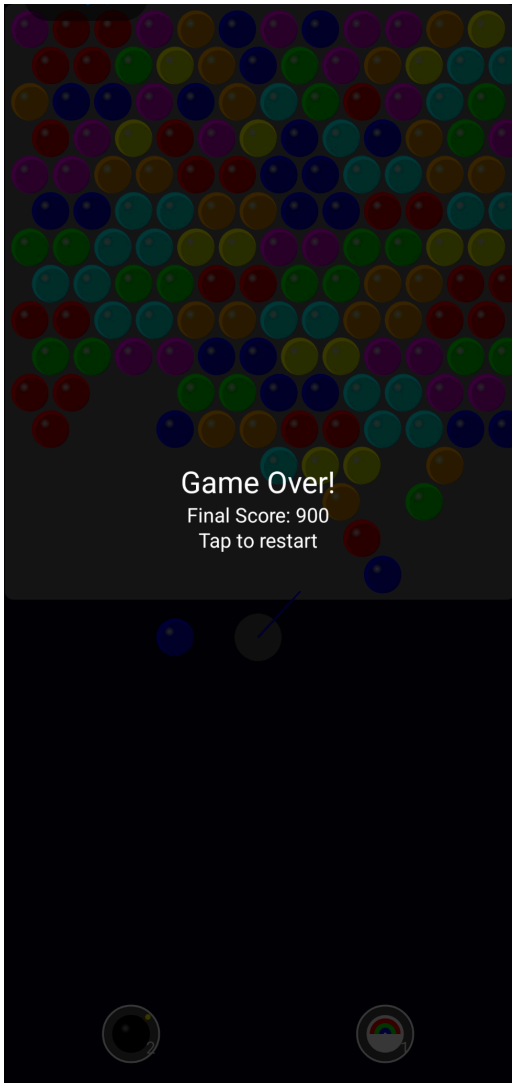
- **Save Score Dialog:** Modal dialog appearing after game over. Text input field for player name. "Lưu điểm" (Save Score) and "Hủy" (Cancel) buttons. Score display showing the achieved score.



- **Leaderboard Screen:** Full-screen overlay with dark background. Title "Bảng xếp hạng" (Leaderboard). Scrollable list of top scores with rank, player name, and score. "Về Menu Chính" (Back) button to return to the game.



- **Game Over Screen:** Semi-transparent black overlay. "Game Over!" text in white. Final score display. "Tap to restart" instruction.



c. Technical Solution for Function Development

Solution and Techniques Used:

1. **Development Technology:**

The game is built using Android with a hybrid approach - Jetpack Compose for the outer container and back button, and SurfaceView with custom Thread for the game rendering loop.

2. Architecture:

- **UI Layer:** BubbleShooterScreen (Composable) provides the container using AndroidView to embed the custom SurfaceView, along with the back button overlay.
- **Game View:** BubbleShooterView extends SurfaceView and implements SurfaceHolder.Callback for lifecycle management.
- **Game Thread:** BubbleShooterThread handles the game loop with delta time calculation, running at approximately 60 FPS using Thread.sleep(1) for timing control.
- **State Management:** Game state is managed using enum class GameState (READY, SHOOT_BUBBLE, REMOVE_CLUSTER, GAME_OVER) with corresponding update and draw methods.
- **Data Structures:** Tile data class stores grid cell information (type, removed, shift, velocity, alpha, processed, x, y). Bubble data class stores bubble properties (x, y, type, visible, angle, speed).

New Features and Implementation Details

- **Hexagonal Grid System:** Implementation of hexagonal grid positioning with row offset alternation. The grid uses a staggered layout where odd rows are offset by half a tile width, creating the classic bubble shooter hexagonal pattern.
- **Dual Power-Up System:** Bomb Bubble: Destroys all 6 neighboring bubbles regardless of color using the getNeighbors() function. Rainbow Bubble: Analyzes neighboring bubble colors using frequency counting (groupBy and maxByOrNull) to automatically become the most advantageous color.
- **3D Bubble Rendering:** Custom draw3DBubble() function creates depth illusion using: Base color circle, RadialGradient highlight positioned at upper-left, Shadow circle positioned at lower-right with transparency.
- **Floating Cluster Detection:** Two-phase algorithm: Flood-fill from ceiling (floodFillConnected using BFS with ArrayDeque) to mark all connected bubbles. The second pass identifies unmarked bubbles as floating clusters.
- **Gravity Simulation for Falling Bubbles:** Floating bubbles receive initial velocity (900f) and acceleration (700f × deltaTime) for realistic falling motion with simultaneous alpha fade-out.
- **Firebase Leaderboard Integration:** The game integrates with Firebase Realtime Database for persistent score storage. Key implementation details:
    - Game ID constant (GAME_ID_BUBBLE_SHOOTER = "bubble_shooter") identifies this game's scores in the database.
    - onGameOver callback pattern: BubbleShooterView constructor accepts a lambda (Int), Unit that receives the final score when game ends.

    - Thread-safe callback: mainHandler.post {} ensures the callback executes on the main UI thread from the game thread.
    - hasTriggeredGameOver flag prevents multiple callback invocations for the same game session.
    - SaveScoreDialog component handles user input for player name and triggers Firebase write operation.
    - LeaderboardScreen component fetches and displays scores from Firebase, sorted by score value.

Difficult Technical Issues

- **Hexagonal Grid Coordinate System:** Converting between screen coordinates and hexagonal grid positions requires careful handling of row offsets. The getGridPosition() and getTileCoordinate() functions must account for alternating row offsets to properly snap bubbles and render them.
- **Cluster Detection Algorithm Optimization:** The BFS-based findCluster() algorithm must efficiently traverse the hexagonal neighbor structure. The neighborOffsets array defines different neighbor positions for even and odd rows, requiring careful index management.
- **Collision Detection Accuracy:** Circle-circle intersection (circleIntersection()) must be precise enough to feel natural while being efficient. The collision check iterates through all existing bubbles each frame during the shooting state.
- **Animation State Synchronization:** Managing multiple animation states simultaneously (cluster fade-out, floating cluster fall, new row addition) requires careful state tracking to prevent visual glitches and ensure smooth transitions.
- **SurfaceView Thread Safety:** Coordinating between the main UI thread (touch events, Compose UI) and the game thread (update/draw loop) requires proper synchronization through the volatile running and paused flags, plus Handler for cross-thread communication.
- **Memory Management:** The tile array and cluster lists must be properly reset and reused to prevent memory allocation during gameplay, which could cause frame drops.
- **Cross-Thread Callback Communication:** The game thread must safely communicate the final score to Compose UI on the main thread. This is achieved using Handler(Looper.getMainLooper()).post {} which schedules the callback execution on the main thread's message queue.
- **Compose State Integration with SurfaceView:** Managing Compose state (showSaveDialog, showLeaderboard) that responds to events from a

non-Compose SurfaceView requires careful callback design. The remember {} block ensures the gameView instance persists across recompositions.

d. Potential Future Developments

- **Additional Power-Ups:** Fire Bubble: Burns through multiple bubbles in a line. Ice Bubble: Freezes bubbles temporarily preventing new row addition. Lightning Bubble: Destroys all bubbles of the same color on the grid.
- **Visual Enhancements:** Add particle effects for bubble destruction. Implement combo visual feedback with screen shake. Add animated backgrounds with parallax scrolling.
- **Audio Integration:** Add sound effects for shooting, popping, and power-up activation. Include background music with multiple tracks. Implement audio settings.
- **Game Modes:** Puzzle Mode - Clear specific patterns with limited shots.
- **Leaderboard Enhancements:** Add player authentication for verified scores. Implement weekly/monthly leaderboard resets. Add player statistics tracking (games played, average score, highest combo). Social features - share scores on social media.

B. Whack-A-Mole

a. Detailed Function Description

This game is a digital adaptation of the classic arcade Whack-a-Mole game, developed on the Android platform using **Jetpack Compose** for a fully declarative UI approach.

Core Functionality:

- **Gameplay (Whacking Moles):** Moles (represented as mice) randomly appear from holes on a grid. The player must tap on visible moles to score points before they disappear.
- **Objective:** Score as many points as possible within the 60-second time limit by successfully tapping on appearing moles.
- **Scoring and Penalty System:** The player gains 10 points for each successful mole hit. The player loses 5 points for tapping an empty hole. The game ends immediately if the score becomes negative.

- **Difficulty Levels:** Three difficulty modes are available: Easy (9 holes, 3×3 grid, slower moles). Medium (16 holes, 4×4 grid, moderate speed). Hard (24 holes, 4×6 grid, fast moles with multiple simultaneous appearances).

Execution Flow

1. **Startup:** The application launches, WhackAMoleActivity receives difficulty parameter via Intent, WhackAMoleGame composable is initialized with the selected difficulty.

2. **Start Screen:** The game displays the title "Whack-a-Mole" with instructions and a "Bắt đầu chơi" (Start Playing) button.

3. **Game Initialization:** When the player taps start, gameStarted is set to true, triggering two LaunchedEffect coroutines: Timer coroutine and Mole appearance coroutine.

4. **Timer Coroutine:** Decrements timeLeft every 1000ms. Increments gameTime for tracking. Sets gameOver = true when timeLeft reaches 0.

5. **Mole Appearance Logic:** Continuously checks and hides moles that exceed their visibility duration. During rest period (no visible moles), waits for the configured restPeriod. Randomly selects holes to show moles based on probability: Single mole (default). Double mole (multipleMouseChance). Triple mole (tripleMouseChance, Hard mode only).

6. **Player Interaction:** Tap on visible mole, score += 10, mole disappears. Tap on the empty hole, score -= 5. If score < 0, gameOver = true.

7. **Game Over State:** Displays "Hết giờ!" (Time's Up) or "Game Over!" if negative score. Shows final score and options: "Chơi lại" (Play Again) or "Thoát" (Exit).

Business Constraints

- **Game Duration:** Fixed at 60 seconds for all difficulty levels.
- **Scoring Rules:** Hit mole: +10 points. Miss (empty hole): -5 points.
- **Negative Score Penalty:** Game ends immediately when score falls below 0.
- **Difficulty Parameters:**

Parameter	Easy	Medium	Hard

Hole Count	9 (3×3)	16 (4×4)	24 (4×6)
Mouse Visibility Duration	1000ms	500ms	200ms
Rest Period	800ms	600ms	400ms

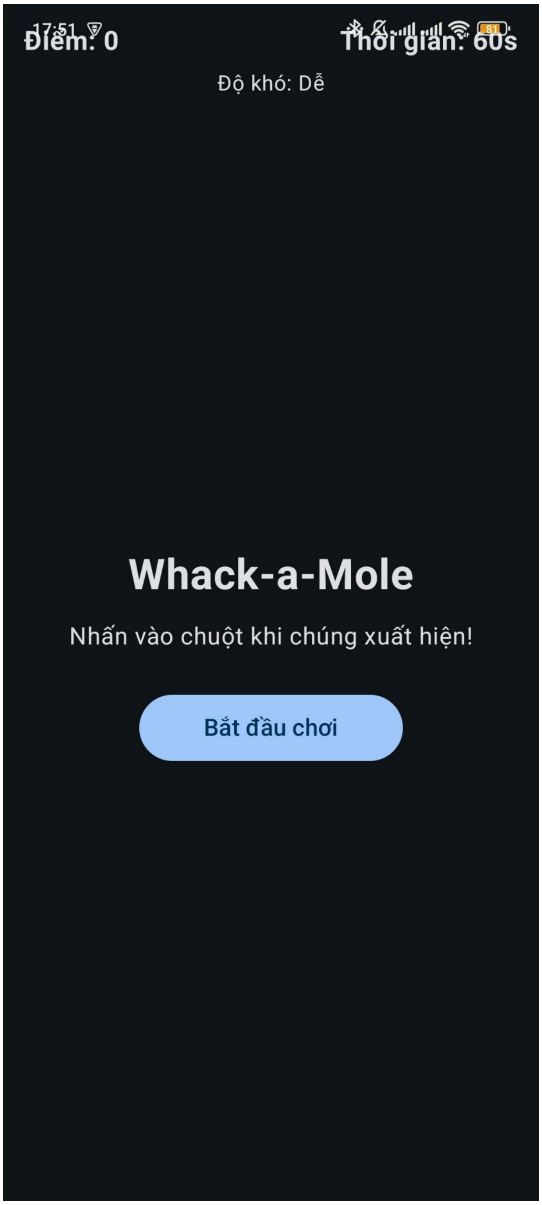
b. Interface Design

Difficulty selection:

- Background: Linear gradient from dark blue to darker blue, creating a deep space atmosphere.
- Simple Column layout. Displays "Chọn độ khó" (Select Difficulty).
- Three primary buttons for: "Dễ", "Trung bình ", and "Khó".
- An OutlinedButton to "Quay lại Menu" (Return to Menu) (exits Activity).

Start Screen

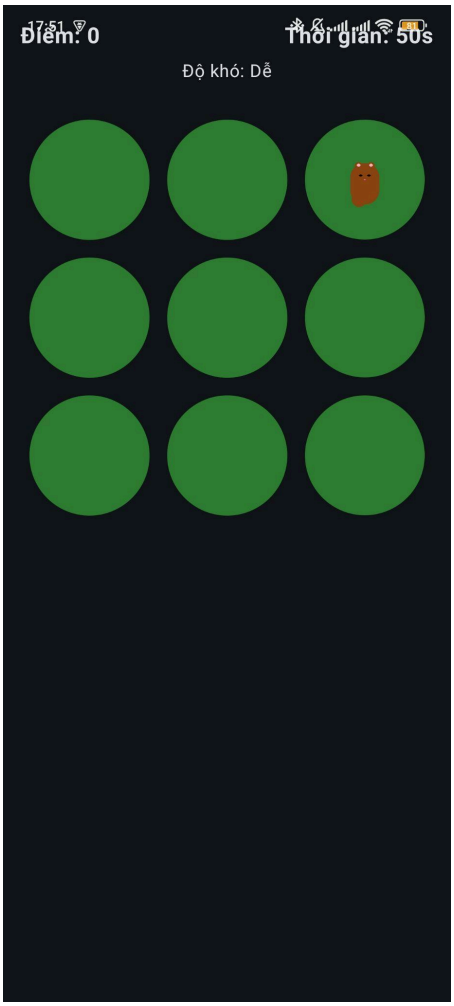
- Title: "Whack-a-Mole"
- Instructions: "Nhấn vào chuột khi chúng xuất hiện!" (Tap on mice when they appear!)
- Start button: "Bắt đầu chơi"



Game Screen Layout

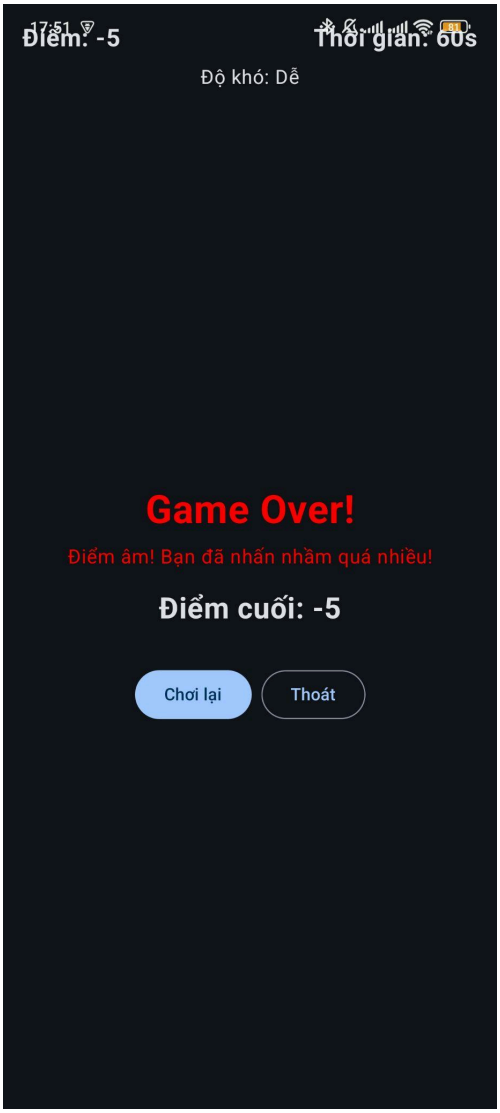
- **Header Section:** Score display on the left ("Điểm: X"). Time remaining on the right ("Thời gian: Xs"). Difficulty indicator below ("Độ khó: [Easy/Medium/Hard]").

- **Game Grid:** LazyVerticalGrid displaying circular holes. Grid columns vary by difficulty (3 for Easy, 4 for Medium/Hard). Dark green circular holes with aspect ratio 1:1. Mouse image appears centered within holes when visible.
- **Hole Appearance:** Empty hole: Solid dark green circle. Active hole: Mouse vector drawable (mouse_vector.xml) centered within the circle. Mouse size scales with difficulty (60dp Easy, 50dp Medium, 45dp Hard).



Game Over Screen

- Status message: "Hết giờ!" (Time's Up) or "Game Over!" (if negative score)
- Penalty explanation if applicable: "Điểm âm! Bạn đã nhấn nhầm quá nhiều!"
- Final score display: "Điểm cuối: X"
- Action buttons: "Chơi lại" (Play Again) and "Thoát" (Exit)



c. Technical Solution for Function Development

Solution and Techniques Used

1. **Development Technology:**

The game is built entirely using Jetpack Compose for declarative UI, leveraging Kotlin Coroutines for asynchronous game timing and state management.

2. Architecture:

- **UI Layer:** Single WhackAMoleGame composable function handles all UI states (start, playing, game over) using conditional rendering.
- **State Management:** Compose state holders: mutableIntStateOf for score and timeLeft. mutableStateOf for gameStarted, gameOver, and holes list. mutableLongStateOf for gameTime tracking.
- **Asynchronous Logic:** Two LaunchedEffect blocks manage: Timer countdown (1-second intervals). Mole appearance/disappearance logic (variable intervals based on difficulty).
- **Data Structures:** Hole data class: Stores isMouseVisible and mouseAppearTime. Difficulty enum class: Encapsulates all difficulty-specific parameters (holeCount, mouseAppearInterval, mouseVisibilityDuration, restPeriod, multipleMouseChance, tripleMouseChance).
- **Grid Rendering:** LazyVerticalGrid with GridCells.Fixed() for responsive grid layout. itemsIndexed for efficient item rendering with index access.

New Features and Implementation Details

- **Configurable Difficulty System:** The Difficulty enum class provides a clean, type-safe way to configure all game parameters. Each difficulty level defines 7 distinct parameters, allowing easy addition of new difficulty levels without code changes.
- **Multiple Mole Appearance System:** Probability-based system for showing multiple moles simultaneously. Uses Random.nextFloat() to determine mole count. shuffled().take(mouseCount) ensures random hole selection without duplicates.
- **Time-Based Mole Visibility:** Each hole tracks its mouseAppearTime using System.currentTimeMillis(). Mole visibility is automatically revoked when current time exceeds appearTime + visibilityDuration. This ensures consistent difficulty regardless of frame rate.
- **Rest Period Implementation:** After all moles disappear, the game waits for a configured restPeriod before showing new moles. This creates a rhythm to the gameplay and prevents overwhelming the player.

- **Immediate Feedback System:** Tapping a visible mole immediately hides it and awards points. Tapping an empty hole provides negative feedback with score deduction. Negative score threshold creates a "survival" element to the gameplay.

Difficult Technical Issues

- **Coroutine Lifecycle Management:** LaunchedEffect must properly handle key changes (gameStarted, gameOver, difficulty) to restart coroutines when game state changes. The while loop conditions must check both !gameOver and timeLeft > 0 to prevent continued execution after game end.
- **State Consistency with Immutable Lists:** Compose requires immutable state updates. The holes list must be replaced entirely using mapIndexed rather than mutated. This pattern: holes = holes.mapIndexed { i, h -> ... } ensures proper recomposition.
- **Timing Precision:** Using delay() in coroutines may not be perfectly accurate due to system scheduling. The mouseAppearTime timestamp ensures mole visibility duration is consistent regardless of delay accuracy.
- **UI Thread Safety:** All state modifications occur on the main thread through Compose's state system. The clickable modifier safely updates state without explicit synchronization.
- **Grid Layout Responsiveness:** LazyVerticalGrid with aspectRatio(1f) ensures holes remain circular across different screen sizes. The grid column count must balance between difficulty visibility and touch target size.
- **Activity Communication:** Receiving difficulty parameter via Intent and safely casting context to Activity for finish() requires proper null safety handling.

d. Potential Future Developments

- **Special Mole Types:** Golden Mole: Appears rarely, worth 50 points. Bomb Mole: Tapping it deducts 20 points (penalty mole). Speed Mole: Appears and disappears extra fast, worth 20 points.
- **Combo System:** Implement consecutive hit tracking. Award bonus multipliers for hitting multiple moles in quick succession. Visual feedback for combo streaks.
- **Visual and Audio Enhancements:** Add hit/miss animations (mole getting bonked, stars, etc.). Implement sound effects for hits, misses, and game events. Add background music that intensifies as time runs out.
- **Progressive Difficulty:** Implement adaptive difficulty that increases mole speed as score increases. Add "waves" where multiple moles appear more frequently over time.

- **Achievement System:** Track statistics (total hits, accuracy percentage, etc.). Unlock achievements for milestones (100 hits, perfect accuracy round, etc.).

3. Hồ Anh Dũng

- ***Tetris***

a. DETAILED FUNCTION DESCRIPTION

1. Objective of the Function

The Tetris module is a mini-game integrated into the Gamin application.
Its main purposes are to implement:

- Block falling mechanics
- Collision detection
- Block rotation
- Line clearing & scoring
- Game loop control
- Touch & gesture-based controls
- Pause / Resume / Game Over states

2. Overall Activity Flow

2.1 Game Initialization

- Create a 10×20 board matrix.
- Generate the first block using getNewBlock().
- Start the game loop using gameLoop().

2.2 Game Loop (gameLoop())

1. Periodically (based on delay), check if the block can fall.
2. If no collision, move the block downward (currentBlock.y++).
3. If collision is detected:
    - Merge block into the board using mergeBlock().
    - Clear completed lines using clearLines().
4. Spawn new block (getNewBlock()).
5. If the new block immediately collides → Game Over.

2.3 Input Processing

- Swipe left → moveLeft()

- Swipe right → moveRight()
- Tap → rotate()
- Downward swipe → soft drop
- Press “Play Again” button → restart game

2.4 Line Clearing (clearLines())

- Scan each row from bottom to top.
- If a row is fully filled → remove the row → shift upper rows downward.
- Score is updated accordingly (if integrated).

2.5 Game Over State

Triggered when a newly spawned block collides immediately.

Actions taken:

- isGameOver = true
- Stop game loop (threadRunning = false)
- Display Play Again and Back buttons

b. INTERFACE DESIGN

1. UI Components

The Tetris game interface consists of the following components:

1.1 Gameplay Screen Components

- Tetris Board (10×20 grid)
  Main area where blocks fall.
- Current Falling Block
  Displayed on the board with its assigned color and shape.
- Score Display
  Shows the player’s current score.
- Level Display
  Indicates current game speed level.
- Next Block Preview
  Shows the upcoming block so the player can plan ahead.

- Move Left Button ("<")
  Located at the bottom-left area.
- Move Right Button (">")
  Located next to the left button.
- Tap Input Area
  Tapping anywhere on the screen rotates the block.
- Long Press Input Area
  Holding the screen makes the block fall faster (soft drop).
- Game Over Overlay
  A semi-transparent dark background covering the game.
- Game Over Dialog
  Includes:
  "Game Over" title
  - Final Score
  - 3-character name input
  - "Save Score" button
  - "Don't Save" button

1.2 Difficulty Selection Screen Components

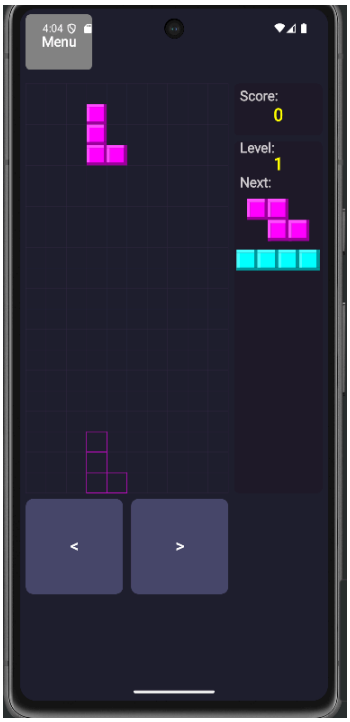
- Game logo
- Title text “Tetris”
- Brief instructions
- Three difficulty buttons
    - Easy
    - Medium
    - Hard
- “Back to Menu” button

1.3 Leaderboard Screen Components

- Ranked list of players
- Player name (3 characters)
- Score value
- Highlight of the most recent score (optional)
- “Back” button to return to menu

2. General Layout Description

2.1 Gameplay Screen Layout



- Left area:
  Large Tetris board grid.
- Right area:
  Vertical information panel showing Score, Level, and Next block.
- Bottom area:
  Two square movement buttons (“<” and “>”).
- Full screen:
  Tap = rotate block
  Long press = soft drop
- Game Over:
  Screen darkens → Popup appears in the center.

2.2 Difficulty Screen Layout

- Logo and title centered at the top.
- Instructions below the title.
- Three difficulty buttons arranged vertically.
- “Back to Menu” button at the bottom.

2.3 Leaderboard Layout

- Centered list of ranked players.
- Columns: Rank – Name – Score.
- “Back” button at bottom center.

3. User Interaction Description

3.1 Rotate Block – Tap

- A short tap anywhere on the screen rotates the current block clockwise.
- Replaces old swipe-based rotation.

3.2 Soft Drop – Long Press

- Holding a finger anywhere on the screen increases the falling speed.
- Releasing returns to normal speed.

3.3 Move Left/Right – Buttons

- Tap "<" to move block left.
- Tap ">" to move block right.

- Ensures precise control on mobile.

3.4 Speed Increase Mechanic

- Every 4 lines cleared cumulatively, the game increases the falling speed.
- Level display updates accordingly.

3.5 Game Over Dialog Interaction

When the game ends:

- A popup appears with score and a 3-character name input.
- The player enters their name.

If “Lưu điểm” is pressed:

- Name + score are saved.
- The system immediately navigates to the Leaderboard Screen.

If “Không lưu” is pressed:

- Score is discarded.
- Player returns to the menu.

3.6 Difficulty Selection Interaction

- Tapping a difficulty button sets the initial speed.
- Player enters the game after choosing difficulty.
- “Back to Menu” returns to previous screen.

3.7 Leaderboard Interaction

- Player views ranked scores.
- “Về menu chính” returns to the main menu.

c. TECHNICAL SOLUTION

1. Description of the solution and techniques used

The Tetris game is implemented using a custom Android View named TetrisView.
This View is responsible for rendering the board, updating game logic, and processing user interactions.
The main techniques used in this function include:

- A manually controlled game loop using a boolean flag to update the block position over time.
- A two-dimensional array representing the Tetris board, where each cell stores the color or type of a block.
- Collision detection based on comparing the block’s shape matrix with the board matrix before movement.
- Rotation logic implemented by switching between predefined shape orientations stored in TetrisBlock.

- Gesture processing using MotionEvent to recognize swipes and taps for movement and rotation.
- Canvas-based rendering to draw blocks, control buttons, and the Game Over overlay.
- Thread management to handle pause and resume operations inside TetrisActivity.

These components work together to ensure that the game behaves consistently with the Tetris rules.

2. New features or improvements introduced by this function

This implementation introduces several improvements compared to a basic Tetris version:

- Gesture-based control, allowing the player to move or rotate blocks using natural touch interactions.
- A soft-drop mechanism with an adjustable delay that increases the falling speed of the block.
- A fully custom layout system, where all UI elements (board, buttons, overlay) are scalable depending on device size.
- Dynamic button hit areas defined by RectF regions instead of fixed pixel positions.
- A self-contained rendering and logic cycle inside one View, making the feature easy to integrate into the larger application.

3. Technical difficulties encountered during development

Several challenges occurred while building this function:

- Synchronizing UI updates and game logic, because the game loop does not run on the main UI thread.
- Ensuring accurate gesture detection across different screen sizes and densities to avoid misinterpreting taps as swipes.
- Implementing rotation near walls, since the current version does not use advanced wall-kick algorithms and may restrict rotation.
- Optimizing performance so that the board and falling blocks can be redrawn smoothly without frame drops.
- Managing state transitions such as Game Over and Play Again without interrupting the rendering loop.

These difficulties required careful adjustment of thresholds, thread handling, and drawing logic.

d. FUTURE DEVELOPMENT

1. 7-Bag Randomizer Biomass System

Replaces the random biomass work with a system of 7 alternating block sessions, ensuring fairness.

2. Sound and Background Music

Adds sound for block rotation, transitions, line deletion, and Game Over.

Integrated option to toggle platform selection.

3. Customizable Interface (Theme/Skin)

Allows changing visual styles (neon, pixel-art, minimalist, classic).

4. Menu Settings

Includes customizations such as:

• controller sensitivity

• soft drop speed

• selection interface

• sound on/off

4. Vũ Minh Đức

a. Arkanoid

i. Detailed Function Description

This game is a variant of the classic brick-breaking genre (Breakout/Arkanoid), developed on the Android platform using Jetpack Compose.

Core Functionality:

- **Gameplay** (Brick Breaking): The player controls a Paddle at the bottom of the screen to deflect one or more Balls that collide with and destroy Bricks arranged above.
- **Objective**: Destroy all bricks in a Wave (level) to advance to the next level.
- **Scoring and Lives**: The player gains points by breaking bricks. The player loses a Life if the ball falls to the bottom of the screen. The game ends when the player runs out of lives or time expires.
- **Advanced Features**: Bricks may contain Power-Ups (e.g., Multi-Ball, Paddle-Grow) or be special brick types (Explosive, Boss). Each wave has a time limit of 180 seconds.

Execution Flow

- Startup: The application launches The LevelSelectScreen is displayed.
- Select Wave: The player chooses a Wave from 1 to 10 Transition to the ArkanoidGameScreen.
- Ready State: The game is in the GameState.Ready state: the paddle, ball, and bricks are initialized.
- Start Play: The player taps the screen The state transitions to GameState.Playing.
- Game Loop (Playing):
  - The elapsed time (Delta Time - dt) is calculated, and the positions of the ball(s), paddle, and Power-Ups are updated.
  - Time Reduction: The Time Left counter is updated. If time runs out, the state transitions to GameState.GameOver.
  - Collision Handling:

- Ball hits Wall: The velocity vector is reversed, and the ball's speed is slightly increased.
- Ball hits Paddle: The Y velocity is reversed. The bounce angle is calculated based on the impact point on the paddle, and the speed is increased.
- Ball hits Brick:
- The brick is destroyed (or its HP is reduced if it is a Boss brick).
- Points are awarded.
- The ball's Y velocity is reversed, and its speed is increased.
- If the brick has a Power-Up: A Power-Up item is spawned and drops down.
- If it is an Explosive brick: Neighboring bricks are destroyed.
- If the brick has a Star: The collected star count is incremented.

▪ Power-Up Handling: Power-Up drops If it hits the paddle The effect is activated (Multi-Ball or Paddle-Grow), and an expiration time is set.

▪ Loss of Life: If all balls fall off the bottom of the screen Life count is reduced (lives--).

- If lives <= 0: Score is saved, state transitions to GameState.GameOver.
- If lives > 0: The ball/paddle positions are reset, state returns to GameState.Ready.

▪ Wave Clear: If all bricks are destroyed Score is saved, state transitions to GameState.WaveClear.

o WaveClear State: A results dialog is displayed The player chooses Continue (next wave) or Return to Level Select.

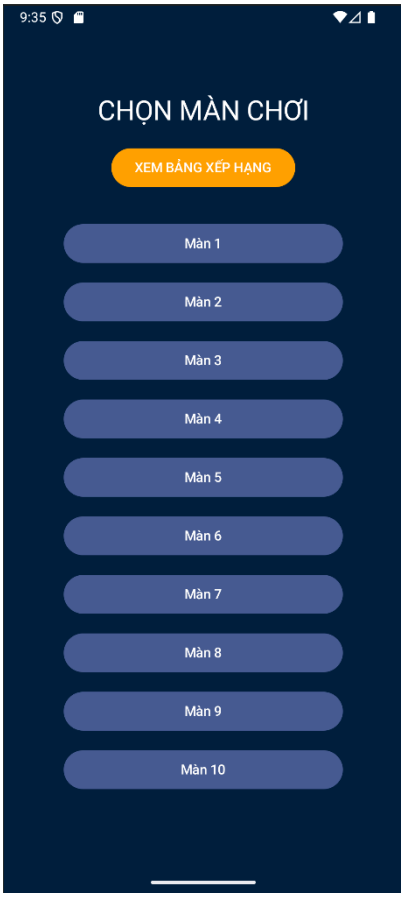
o GameOver State: The "GAME OVER" text is displayed The player Taps to Play Again (restart level).

Business Constraints

o Max Ball Speed: The ball's maximum speed is 2500f (MAX_BALL_SPEED).

- Paddle Position: The paddle's movement is constrained within the screen width boundaries.
- Power-Up Duration: Power-Up effects only last for 5000L (5 seconds).
- Score Saving: The score is only saved if the score is greater than 0.
- High Score Uniqueness: The high score table uses the Room Database constraint onConflict = OnConflictStrategy.IGNORE, ensuring that the combination of scores for the 10 waves is unique.
- Special Bricks:
  - BOSS bricks require 3+ hits (hitPoints) to be destroyed.
  - EXPLOSIVE bricks cannot contain a Power-Up.

ii. Interface Design



1. Game Canvas

a. Background: Gradient from Black to Dark Blue

b. Bricks: Rounded rectangles with diverse colors

   i. **BOSS** bricks are Orange/Yellow:

   ii. **EXPLOSIVE** bricks have a Red circle in the center

   iii. **Star** bricks have a Yellow circle.

c. Paddle: Rounded rectangle in White.

d. Ball: Circular, Yellow

e. Power-Up:

   i. Square shapes: Green for Multi-Ball, Cyan for Paddle-Grow.

2. HUD

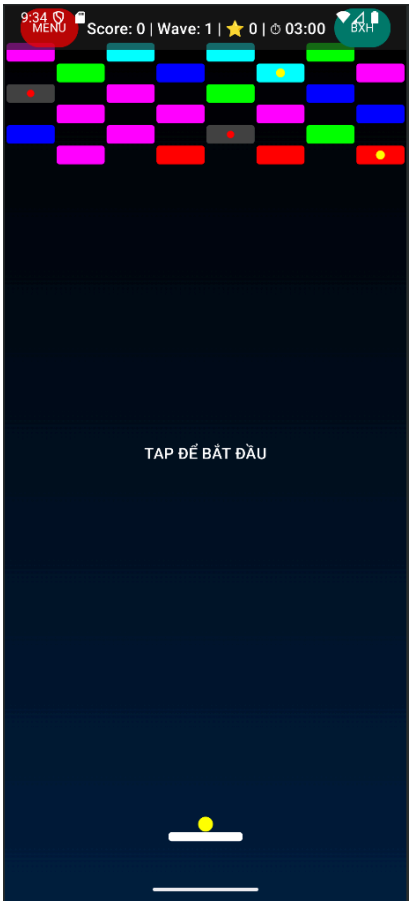
a. A dark horizontal band at the top of the screen

b. It includes two buttons:

   i. “quay lại”: to return to the main selection screen

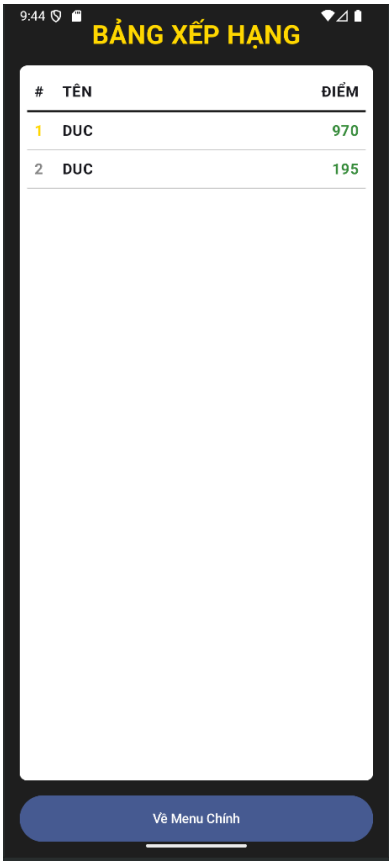
   ii. BXH: to view the score of the player.

3. Status Screen



a. Displays large text in the center of the screen: "TAP ĐỂ BẮT ĐẦU" (TAP TO START) when in the Ready state.

4. **Leaderboard Screen**



a. A dark-colored Card dialog centered on the screen.

i. It displays the title TOP SCORE - MÀN [Wave] (in Yellow)

ii. The score list is shown in a table format with columns for Index (STT), Name(Tên), Score (Điểm).

iii. Technical Solution for Function Development

1. Solution and Techniques Used

a. Development Technology: The game is built using Android with Jetpack Compose for the user interface.

b. Architecture:

i. UI/Rendering (Client): Uses Compose's Canvas to draw game components (balls, bricks, paddle) and

Box with other Composable elements to display the HUD and Pop-ups.

ii. Game Logic: Managed within a LaunchedEffect that executes a game loop using delay(16) (approximately 60 FPS).

iii. State Management: Utilizes Compose's remember mutableStateOf(...) variables to store the game state (balls, bricks, score, lives, game state), allowing the UI to update automatically upon state changes.

iv. Persistence (Storage): Uses the Android Room library to store High Scores.

c. Data Connection: Connects to the AppDatabase (Room Database) via ScoreDao to perform operations:

i. insertScore: Saves the score for the current wave.

ii. getTopScoresForWave: Queries the top 100 highest scores for a specific wave.

2. New Features and Implementation Details

a. Advanced Brick System (Wave Generator): In addition to normal bricks, the game includes Explosive bricks (causing chain reactions) and BOSS bricks (requiring multiple hits to destroy).

b. Advanced Ball Dynamics:

i. Continuous Speed Boost: The ball's speed is slightly increased after every collision with walls and bricks, and significantly increased upon hitting the paddle, capped at MAX_BALL_SPEED. This gradually increases difficulty over time.

ii. Flexible Bounce Angle: The bounce angle from the paddle is calculated based on the impact point relative to the paddle's center, allowing the player to control the ball's trajectory more precisely.

c. Per-Wave Score Accumulation System: The database is designed to store individual scores for 10 different waves

(wave_1_score to wave_10_score) within the same ScoreRecord, instead of just storing the total score.

d. Stars Collected System: Bricks may contain Stars, and the collection of these stars (starsCollected) can be used for future unlocking features or rewards.

e. Time Limit: The addition of a time limit (180 seconds) pressures the player to complete the wave quickly, increasing the element of urgency and difficulty.

3. Difficult Technical Issues

a. Collision Synchronization and State Update: The game loop must accurately and simultaneously handle:

i. Ball-Brick Collision: Ensuring the ball bounces only once per frame, managing the simultaneous destruction of neighboring bricks (for Explosive bricks), and updating the score/brick state (for Boss bricks).

ii. Multi-Ball Management: When Multi-Ball is activated, the system must efficiently handle collisions and state updates for multiple ball objects simultaneously.

b. Performance Optimization (Jetpack Compose): Although Compose is not primarily designed for games with continuous update loops, the use of Canvas and mutableStateOf requires efficient management of recompositions. The delay(16) loop attempts to maintain a target of 60 FPS.

c. Database Design for Multi-Wave Leaderboard: Storing 10 scores in a single record and building dynamic queries using RawQuery (getCustomScores) to fetch high scores for each specific wave is a technical challenge to ensure flexibility and query performance.

d. Power-Up Effect Management: Tracking and automatically revoking extended effects (like Paddle-Grow) based on real-time (expirationTime) requires continuous state checking logic within the game loop.

iv. Potential Future Developments

1. Items and Upgrades:
    a. Add new Power-Ups (e.g., Ball Slow, Fire Ball, Laser Paddle).
    b. Use the number of Stars Collected (starsCollected) to unlock new features, skins, or more powerful Power-Ups.
2. New Game Modes and Levels:
    a. Develop more levels beyond Wave 10.
    b. Add new game modes such as "Survival" (Endless) or "Time Attack"
3. Gameplay Improvement:
    a. Add Sound Effects and Background Music (BGM) to enhance engagement.
    b. Improve Visual Feedback when the ball collides or a Power-Up is activated.

b. Flappy Bird

Detailed Function Description: Execution Flow

This is a single-player game based on the popular Flappy Bird mechanics, developed on the Android platform using Jetpack Compose.

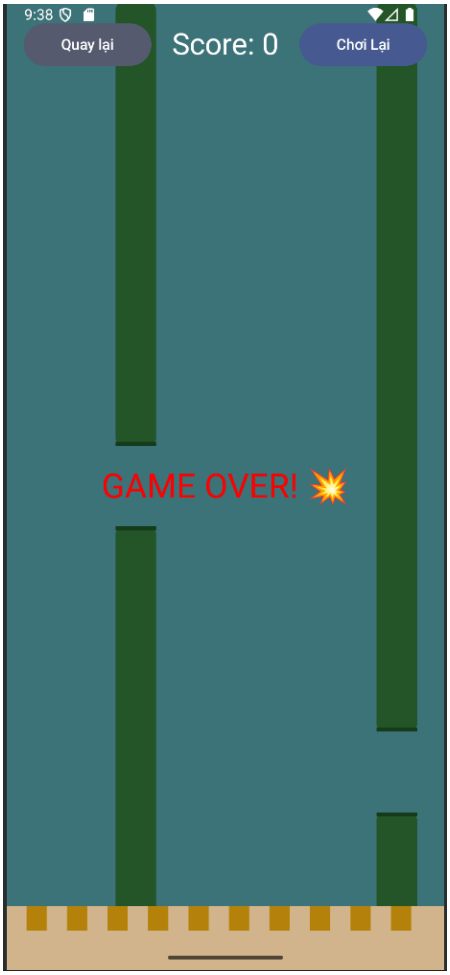
- **Core Functionality**
  - **Gameplay** (Vertical Scrolling/Tapping): The player controls a Bird whose horizontal position is fixed, but whose vertical position is controlled by gravity and player taps. The goal is to navigate the bird through gaps in randomly generated pipes that scroll horizontally across the screen.
  - **Physics**: The game implements simple gravity (GRAVITY = 1000f) and a jump velocity (JUMP_VELOCITY = -500f).
  - **Scoring**: The player gains one point for successfully passing through the gap of a pipe.
  - **Game Over**: The game ends when the bird collides with a pipe, the ground, or the top edge of the screen.
- **Execution Flow (Game Loop)**
  - **Startup**: The FlappyBirdActivity launches → The FlappyBirdScreen Composable is displayed.
  - **Ready State**: The game starts in the GameState.Ready state.
    - The bird is initialized at the center of the screen's height.
    - A list of pipes is initialized off-screen (1.5 x screen width).
  - **Start Play:** The player taps the screen →
    - The state transitions to GameState.Playing
    - The bird's velocity is set to JUMP_VELOCITY (a negative value, causing an upward movement)
  - **Game Loop** (Playing/Crashing): A LaunchedEffect continuously runs the game loop with a delay of 16ms (60 FPS)

- **Time Calculation**: Delta Time (dt) is calculated for frame-rate independence.
- **Bird Update**:
  - New Velocity: Velocity (new) = Velocity (old) + (GRAVITY x dt)
  - New Position: Y(new) = Y (old) + (Velocity (old) x dt)
  - Rotation: The bird's rotation is calculated based on its velocity, simulating a dive or upward flap
- **Pipe Update:**
  - Pipes scroll horizontally: X(new) = X(old) - (PIPE_SPEED x dt)
  - Ground visual scrolls: groundOffset is updated for a parallax effect.
  - New pipes are added off-screen when the last pipe's position is sufficiently far to the left.
- **Collision Detection:**
  - The bird's Rect collides with the Rect of the top or bottom pipe segments.
  - The bird hits the ground (Y + BIRD_SIZE > GameHeight)
  - If a collision is detected, the state transitions to GameState.Crashing.
- **Scoring**: If a pipe's right edge passes the bird's fixed X position and it hasn't been scored yet, score++ and the pipe's scored flag is set to true
- **Crashing State**: If the state is GameState.Crashing, the bird continues to fall under gravity until it hits the ground

- **Game Over**: When the bird hits the ground in the GameState.Crashing state, the state transitions to GameState.GameOver. The screen displays "GAME OVER!"

    - **Restart**: The player taps the "Chơi Lại" button → tapAction() calls resetGame(), setting the state back to GameState.Ready

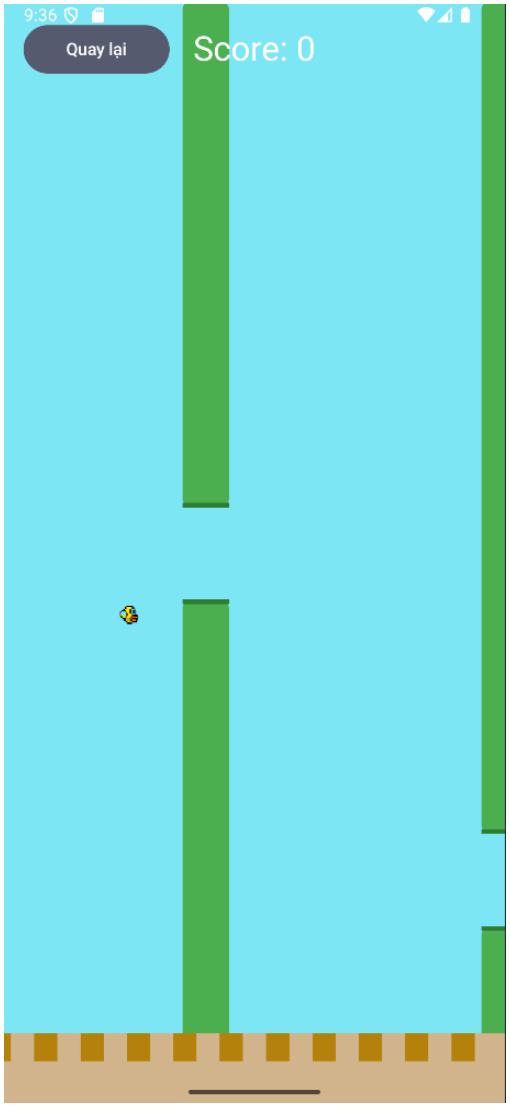
- **Business Constraints**
    - Fixed Parameters: Game physics and dimensions are defined by constants: GRAVITY (1000f), JUMP_VELOCITY (-500f), PIPE_WIDTH (100f), and PIPE_GAP (200f).
    - Vertical Boundaries: The bird's vertical position is constrained between 0f and GameHeight. Hitting the top boundary instantly sets velocity to 0 and position to 0f.
    - Pipe Generation: Pipe gaps are generated randomly but constrained between 20% and 80% of the game height.
    - Movement Control: The only movement input allowed is a tap/click to apply an upward velocity (JUMP_VELOCITY).
- **Interface Design**
    - Game End



- **Game Canvas**

- Background: Solid light blue color.
- Pipes: Drawn as rounded rectangles with a slightly darker color at the top of the bottom pipe and bottom of the top pipe to simulate shadow/thickness.
- Bird: Rendered as an ImageBitmap using drawImage and rotated according to its velocity using withTransform.
- Game Over Overlay: A semi-transparent black overlay covers the screen upon crashing/game over.

- **HUD**



- A Row at the top of the screen displays:
  - A "Quay lại" (Back) button to exit the activity.
  - Current Score (Text: "Score: [score]", White color).
  - A conditional "Chơi Lại" (Play Again) button only visible during GameState.GameOver.

- **Status Text**
  - Large text centered on the screen based on game state:
    - Ready: "TAP ĐỂ BẮT ĐẦU!" (TAP TO START!).
    - Game Over: "GAME OVER!" (Red color).

    - **Ground**
        - A fixed-height Box at the bottom, colored sand/brown. It contains a Canvas that draws alternating darker brown rectangles which scroll horizontally using groundOffset to simulate ground movement.

- **Technical Solution for Function Development**
    - **Solution and Techniques Used**
        - Development Technology: Built using Android with Jetpack Compose.
        - Core Game Loop: The entire game logic, including physics updates and collision detection, is executed inside a coroutine using a LaunchedEffect. This ensures the game runs on a fixed time interval (16ms/frame) while respecting Compose's composition model.
        - Rendering: The main action is drawn using the Canvas Composable, which provides low-level access to drawing methods (rectangles, circles, images). The withTransform block is crucial for applying the calculated rotation to the bird image.
        - Input Handling: User input is processed via the pointerInput(gameState) modifier, which efficiently captures taps and triggers the jump action.
        - State Management: Game variables (birdState, pipes, score, gameState) are managed using Compose's observable state (mutableStateOf, mutableIntStateOf), ensuring the UI automatically redraws on state changes.
    - **New Features and Implementation Details**
        - Physics-Based Rotation: The bird's rotation is directly proportional to its vertical velocity, making the visual feedback more realistic and dynamic compared to a simple fixed-angle sprite.
        - Parallax Ground Effect: The ground's texture is rendered inside a separate Canvas and scrolled using a ground offset

variable, providing a simple yet effective parallax scroll effect that enhances the illusion of movement.

- Clean State Transitions: The game effectively utilizes four discrete states (Ready, Playing, Crashing, GameOver) to control the flow, ensuring actions like tapping or pipe generation are only processed when appropriate. The Crashing state provides a brief, gravity-only fall animation before the final GameOver screen appears.

- **Difficult Technical Issues**
  - Frame-Rate Independent Physics: Ensuring that the physics (gravity and speed) are calculated using the Delta Time (dt) between frames is essential. If dt is not used, the game speed would vary wildly across devices or under load, a common challenge in game development.
  - Efficient Collision Detection: Although the game uses simple bounding box collision (Rect.overlaps), integrating this check efficiently within the high-frequency game loop (delay(16)) and managing the state transition to Crashing immediately upon detection is critical for responsive gameplay.
  - Pipe Management and Spawning: The logic to scroll pipes off-screen and accurately spawn a new pipe at the correct distance (gameWidth - (PIPE_SPEED * 1.8f)) to maintain consistent spacing requires precise timing and calculation within the game loop.
- **Potential Future Developments**
  - Score Persistence and Leaderboard: Implement a Room Database (similar to the Arkanoid example) to save the highest score locally.
  - Sound and Music: Add sound effects for jumping, scoring, and crashing, along with background music, to significantly improve player experience.
  - Assets and Animation: Replace the placeholder ic_bird image with a sprite sheet and implement simple

wing-flapping animation to make the bird look more dynamic.

c. MemoryCard

Detailed Function Description: Execution Flow

This is a classic Memory Card Game developed on the Android platform using Jetpack Compose.

- **Core Functionality**
  - Objective: The player must find all matching pairs of cards by flipping two cards at a time.
  - Gameplay: The game involves flipping cards. If the two flipped cards have the same hidden content ID, they are marked as matched and remain face-up. If they do not match, they are automatically flipped back over after a short delay.
  - Scoring/Tracking: The game tracks the number of Moves made by the player (a move is defined as flipping two cards).
  - Win Condition: The game ends when all cards are successfully matched
  - Difficulty: Players can select the grid size (difficulty) at the start: Easy (2x3), Medium (4x4), or Hard (5x6).
- Execution Flow
  - Startup: The MemoryCardActivity launches →The MemoryCardGameRoot Composable is displayed.
  - Difficulty Selection: The initial screen state is GameScreenState.DifficultySelect. The player selects the desired grid size
  - Game Initialization: The state transitions to GameScreenState.Playing(rows, cols).
    - The MemoryGame model is initialized with the selected rows and cols.
    - A list of CardState objects is created, ensuring there are (rows * cols) / 2 unique content IDs, and the resulting card list is shuffled randomly.

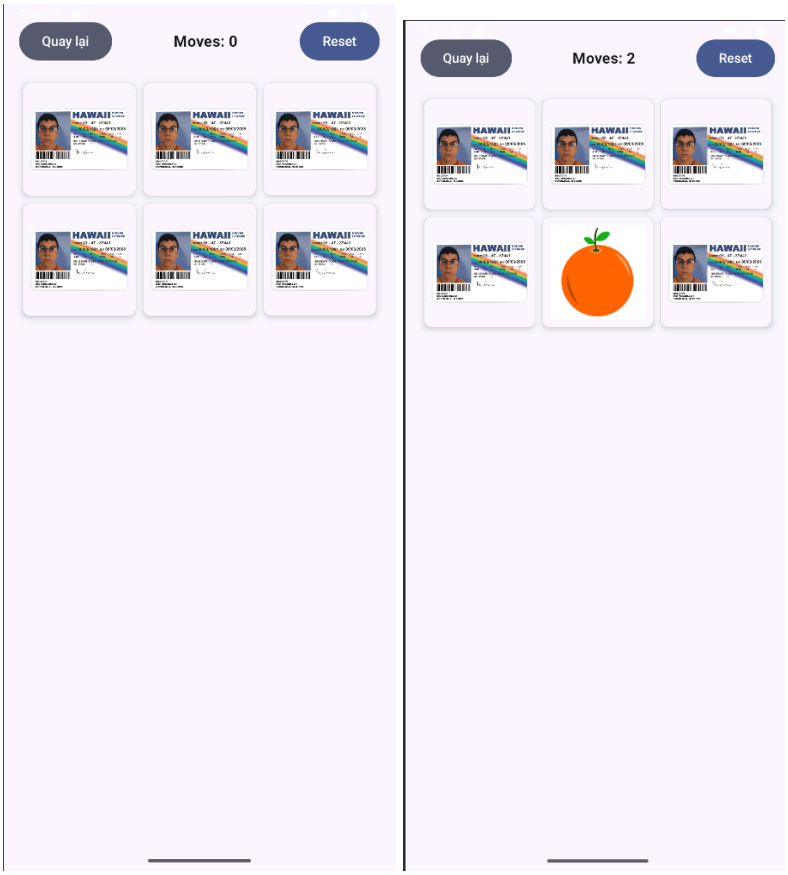
        - The game status is set to "Playing"
    - Card Flip:
        - The player clicks an unflipped, unmatched card.
        - The flipCard(index) function is called.
        - The selected card is set to isFaceUp = true.
        - Case 1: First card flipped: The card is flipped, and its index is stored in currentlyFlippedIndices.
        - Case 2: Second card flipped:
            - The Moves counter is incremented.
            - Match Check: If firstCard.contentId == secondCard.contentId:
                - Both cards are permanently set to isMatched = true.
                - currentlyFlippedIndices is cleared.
                - Win Check: If allMatched is true, the status is set to "You Win!".
            - Mismatch Check: If the cards do not match:
                - Both cards remain face-up.
                - Their indices are held in currentlyFlippedIndices
    - Handling Mismatch (System Action):
        - A LaunchedEffect detects when currentlyFlippedIndices.size == 2.
        - The system pauses for a short delay (HIDE_DELAY_MS = 1000ms).
        - The hideMismatchedCards() function is called, which flips the two cards back over (isFaceUp = false) and clears currentlyFlippedIndices.
    - Win State: If the status is "You Win!":

        - An AlertDialog is displayed, showing the final move count.
        - The player can choose "Chơi lại" (Play Again) (resets the game) or "Chuyển độ khó" (Change Difficulty) (returns to the selection screen).
- Business Constraints
    - Flipping Rules: Cards can only be flipped when the game status is "Playing" and the card is not already face-up or matched.
    - Mismatch Delay: Mismatched cards are only flipped back after a 1000ms delay (HIDE_DELAY_MS).
    - Grid Dimensions: The grid size must be an even number of cards to ensure complete pairs.
- Interface Design
    - Difficulty Selection

        - Simple Column layout. Displays "Chọn độ khó" (Select Difficulty).
        - Three primary buttons for: "Dễ (2x3)", "Trung bình (4x4)", and "Khó (5x6)".

- An OutlinedButton to "Quay lại Menu" (Return to Menu) (exits Activity).

- Game Grid Screen



- Main layout is a Column.
- Top section contains the HUD (Moves counter, Back/Reset buttons).
- The grid of cards is displayed using a LazyVerticalGrid with fixed columns equal to game.cols.

- HUD

- A Row at the top of the grid displays:
  - Moves counter (Text: "Moves: [moves]").
  - "Quay lại" (Back) button to change difficulty.
  - "Reset" button to restart the current game.

    - Memory Card
        - Appearance: Card with RoundedCornerShape and an aspectRatio (square).
        - Flip Animation: Uses graphicsLayer and animateFloatAsState to perform a 3D flip animation over 500ms.
        - Matched Cards: Have a light green background color.
        - Content: Shows a card back image when rotated. Shows the content icon (based on contentId) when rotated
    - Win Dialog
        - An AlertDialog triggered by game.status == "You Win!".
        - Displays the winning message and final move count. Provides Play Again and Change Difficulty buttons.
- Technical Solution for Function Development
    - Solution and Techniques Used
        - Development Technology: Built using Android and Jetpack Compose.
        - State Management (Core): The entire game state is encapsulated within the immutable data class MemoryGame. All player actions (flipCard, hideMismatchedCards) return a new instance of MemoryGame. This ensures state immutability, which is ideal for Compose's reactive UI model.
        - UI Structure: Uses Composable components to manage screen flow (Difficulty Select vs. Playing) via a GameScreenState sealed class. The game grid is rendered efficiently using LazyVerticalGrid.
        - Card Animation: Leverages Compose's animation APIs (animateFloatAsState and graphicsLayer) to implement a smooth, 3D card flipping effect (rotation around the Y-axis). A cameraDistance is set on the graphicsLayer to enhance the 3D perspective during the flip.

        - Time Delay: The mismatch delay is handled using a coroutine with delay(HIDE_DELAY_MS) inside a LaunchedEffect, demonstrating asynchronous logic within the Compose lifecycle.
    - New Features and Implementation Details
        - Immutability-Driven Game Logic: The strict use of immutable MemoryGame state and pure functions simplifies debugging and ensures the UI correctly reflects the latest state, which is a modern, complex pattern in app development.
        - 3D Card Flip Animation: The implementation of the 3D rotation animation using graphicsLayer is a novel visual feature, moving beyond simple 2D state changes. The rotation value is used to conditionally render the front or back of the card, synchronized perfectly with the flip.
        - Flexible Grid and Content: The game dynamically creates the grid and shuffles the card contents based on the selected difficulty (rows and cols). The card content is mapped from a numerical contentId to a specific image resource (R.drawable.mem_icon_X), allowing for easy scaling of content.
    - Difficult Technical Issues
        - State Synchronization with Animation and Delay
        - Handling Clickability during Flip: Preventing the user from clicking a card that is currently flipped or in the process of flipping (during the animation) requires logic in the clickable modifier based on the rotation state, which adds complexity to input handling.
        - Recursive State Update (Win Condition): The flipCard function must handle the immediate transition to the "You Win!" status after setting the final match, which involves a nested conditional check within the main update function.
    - Potential Future Developments

- **Timing and Leaderboard:** Implement a timer to track how quickly the player finishes the game. Integrate a local (Room) or cloud (Firebase) leaderboard to save and compare high scores (based on moves and time).
- **Card Content Customization:** Allow the player to upload their own images to be used as card content, making the game more personalized.

d. Snake

Detailed Function Description

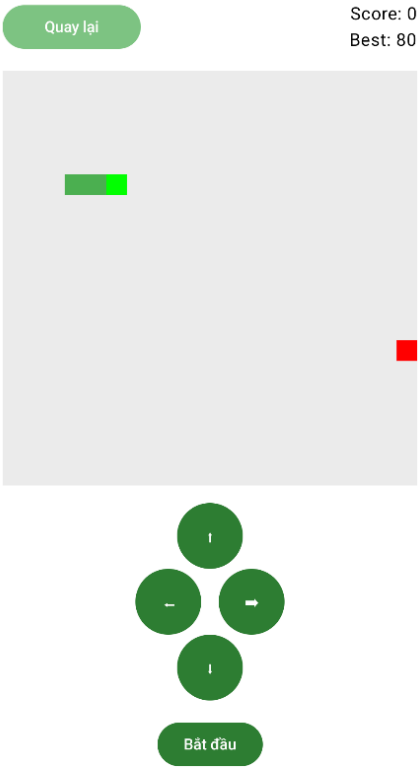
This is a classic Snake Game implemented on a grid, developed on the Android platform using Jetpack Compose.

- Core Functionality
  - **Gameplay (Grid Movement):** The player controls a snake that moves within a fixed grid size of 20 x 20. The snake continuously moves in the current direction.
  - **Objective:** The goal is to eat the randomly spawned **Food**. Eating food increases the snake's length and the player's score.
  - **Scoring:** Eating one piece of food awards **10 points**. The game also tracks the **Best Score** achieved, which is persisted locally.
  - **Control:** Movement is controlled by directional buttons (Up, Down, Left, Right).
  - **Game Over:** The game ends when the snake head collides with the boundary of the grid (walls) or collides with any part of its own body.
- Execution Flow
  - Startup: The SnakeActivity launches, displaying the SnakeGameScreen. The Best Score is loaded from local SharedPreferences.
  - Ready/Initial State: The game is not playing (isPlaying = false, isGameOver = false). The snake is initialized with a starting length (3 segments) and direction (RIGHT). Food is placed randomly. The "Bắt đầu" (Start) button is visible.
  - Start Game: The player clicks the "Start" button $\rightarrow$ score is reset to 0, isGameOver is set to false, and isPlaying is set to true.

- Game Loop (Playing): A LaunchedEffect runs continuously while isPlaying is true
    - **Time Delay:** The loop pauses for **150ms** between updates (defining the game speed).
    - **Movement**: The newHead position is calculated based on the current direction.
    - Game Over Check:
        - If newHead is outside the grid bounds (0 to gridSize - 1).
        - OR if newHead is already in the snake body list (self-collision).
        - The game ends: onGameOver is called, isPlaying becomes false, and isGameOver becomes true.
    - Update Snake
        - The newHead is added to the beginning of the snake list.
        - Food Check: If newHead == food (food eaten):
            - score is increased by 10.
            - A new food location is generated randomly, ensuring it does not spawn on the snake.
            - The snake does not lose its tail (grows in length).
        - No Food:
            - The snake's last segment is removed (removeLast()).
- Direction Change: The player clicks a directional button. The snake's direction is updated, provided the new direction is not the opposite of the current direction.
- Game Over State:
    - The "Game Over" text is displayed.

        - If the finalScore is higher than the best score, the new best score is updated and saved to SharedPreferences.
        - The "Chơi lại" (Play Again) button is displayed, which resets the score and restarts the game.

- Business Constraints
    - Grid Size: Fixed at 20 x 20
    - Control Restriction: Players cannot immediately reverse direction.
    - Game Speed: Fixed by the delay(150) in the game loop.
    - Best Score Persistence: The highest score is saved using SharedPreferences.
- Interface Design



    - Game Canvas
        - Field: Square canvas with a light background.
        - Grid: Implicitly defined by drawing cells based on gridSize.
        - Food: A Red square.
        - Snake: Head is Green; Body is a slightly darker green.
    - HUD
        - A Row at the top of the screen displays:
            - "Quay lại" (Back) button to exit the activity.
            - Current Score (Text: "Score: [score]").
            - Best Score (Text: "Best: [best]").
    - Direction Buttons
        - Arranged in a cross/D-pad layout below the Canvas. Uses Material 3 Button components labeled with emojis (⬆, ⬇, ⬅, ➡).
    - Status Screen
        - Ready: Displays the "Bắt đầu" (Start) button.
        - Game Over: Displays the text "Game Over" and the "Chơi lại" (Play Again) button.
    - Theme
        - Uses a custom SnakeTheme based on Material 3, with Green as the primary color, fitting the game's aesthetic.
- Technical Solution for Function Development
    - Solution and Techniques Used
        - Development Technology: Android with Jetpack Compose.
        - Game Logic/Loop: The core game logic runs within a coroutine using LaunchedEffect(isPlaying). This ties the game loop's execution directly to the isPlaying state variable, ensuring the game starts/stops correctly.

        - Data Model: The snake is represented as a List<Point>. Food and head positions are managed using the simple Point(x: Int, y: Int) data class.
        - State Management: All core variables (snake, direction, food, score) are managed by mutableStateOf, allowing the Canvas and HUD to automatically recompose upon update.
        - Persistence: SharedPreferences is used for persistent storage of the best_score.
        - Rendering: The entire game world is drawn using the Compose Canvas component, which calculates the cell size (cellPx) based on the available space (aspectRatio(1f)).
- New Features and Implementation Details
    - Immutability-driven Snake Movement: The snake's movement is implemented by creating a new list (newSnake) for each update (mutableListOf(newHead).addAll(snake)), and then modifying the list (removing the tail segment if no food is eaten).
    - Grid-based Collision and Boundary Logic: The game uses explicit checks (newHead.x !in 0 until gridSize and newHead in snake) to handle wall and self-collision, which is the foundational logic for Snake games.
    - Local Best Score Persistence: The SharedPreferences integration ensures that the player's achievement (best_score) is remembered across sessions.
- Difficult Technical Issues
    - Input Race Condition Prevention: The direction change logic must prevent the snake from reversing into itself. For example, checking if (direction != Direction.DOWN) before allowing an UP movement is crucial. In high-speed scenarios, input events arriving faster than the game update rate must be managed carefully.
    - Random Food Generation: The randomFood function ensures that the food spawn location is genuinely empty by looping (do...while) until the generated point is not part of the snake's current body segments.
- Potential Future Developments

- Swipe/Gesture Controls: Implement swipe gestures as an alternative input method to the on-screen buttons, especially for touch devices.
- Difficulty Scaling: Add variables to adjust delay(ms) and gridSize based on player preference or score
- Advanced Food/Obstacles: Introduce different types of food or temporary obstacles on the grid to increase complexity.
- Audio Integration: Add sound effects for eating food and game over.

e. TowerBloxx

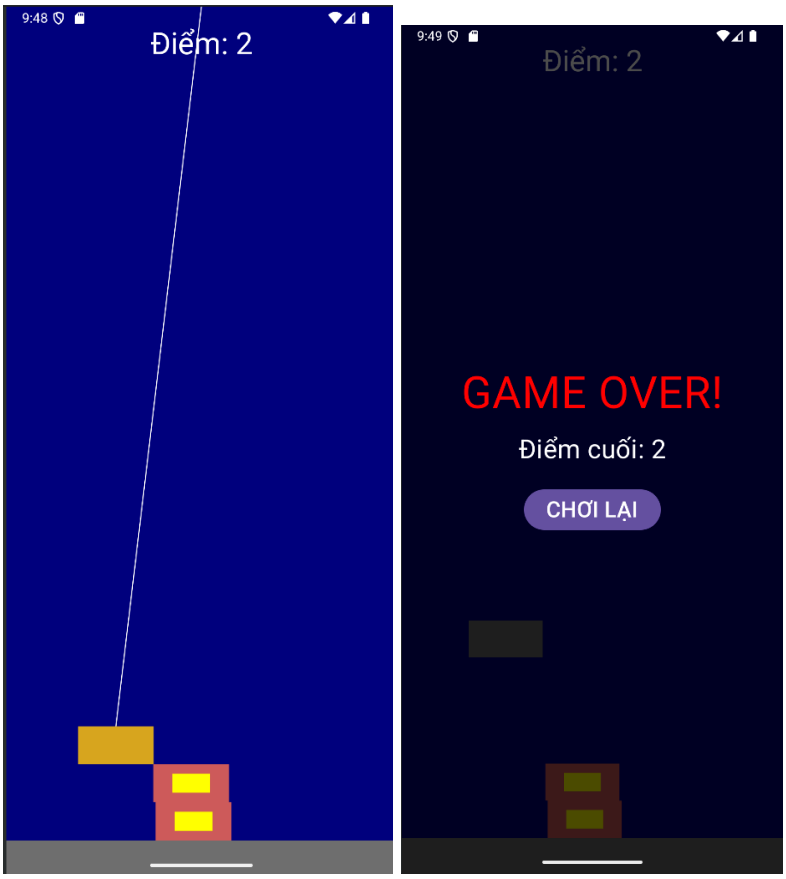
i. Detailed Function Description: Execution Flow

This is a variant of the Tower Bloxx or Stacker game, developed on the Android platform using Jetpack Compose.

- Core Functionality
  - **Gameplay (Timing):** The player attempts to drop a swinging block onto the stack of previously placed blocks.
  - **Swinging Mechanism:** A new block swings horizontally across the top of the screen.
  - **Placement:** The player taps the screen to drop the swinging block.
  - **Scoring:** The score increases by one point for every block successfully stacked.
  - **Game Over:** The game ends if the dropped block misses the base entirely. In this variant, the missed block falls to the ground.
- Execution Flow
  - Startup: The TowerBloxxActivity launches $\rightarrow$ The TowerBloxxGame Composable is displayed.
  - **Initial State:**
    - blocks (the tower) is empty.
    - dropping is false, and gameOver is false.
    - The base width is implicitly the initial block width (blockWidth).
  - Swinging Mechanism (Loop): A LaunchedEffect runs continuously while dropping and gameOver are false.

        - Movement: The swingAngle is incremented.
        - Position: currentBlockX is calculated using a sine wave function based on swingCenter, swingAmplitude, and swingAngle.
        - The block swings smoothly back and forth.
    - Drop Block
        - The player taps the screen.
        - dropping is set to true.
    - Drop Animation (Loop): A second LaunchedEffect runs when dropping is true.
        - he currentBlockY is incrementally increased until it reaches the targetY (which is calculated based on the current height of the tower: blocks.size * blockHeight).
        - Placement Check: Once targetY is reached, collision logic runs:
            - Calculate Overlap: Overlap is determined by subtracting the absolute difference between the dropped block's position and the previous block's position from the previous block's width (overlap = baseWidth - abs(currentBlockX - baseX)).
            - Successful Placement (Overlap > 0): The block is added to the blocks list. currentScore is incremented. dropping is set to false, and a new block starts swinging.
            - Miss (Overlap <= 0): The block missed the tower entirely. gameOver is set to true. The missed block's details are stored in missedBlock.
    - Missed Block Fall (Visual): If gameOver is true and missedBlock exists, a coroutine is launched to animate the missed block falling to the ground (startBlockDrop).
    - Game Over State:
        - A semi-transparent black overlay covers the screen.

        - "GAME OVER!" and the Điểm cuối (Final Score) are displayed.
        - The "CHƠI LẠI" (Play Again) button calls restartGame.
- Business Constraints
    - Collision Model: The game only checks for complete overlap/miss. The block is not cut down to the overlapping width, but retains its original width.
    - Swinging: The swinging motion is governed by a fixed frequency (0.05f).
    - Drop Speed: The block drops at a fixed speed (20f per 10ms delay).
    - Base: The game includes a fixed groundHeight of 100f.
- Interface Design
    - Game Canvas



        - Background: Dark Blue
        - Ground: A fixed Grey rectangle at the bottom (groundHeight).
        - Tower Blocks (Stacked): Red rectangles. Each stacked block has a small Yellow square (Color.Yellow) in its center.
        - Swinging/Dropping Block: Gold/Dark Yellow rectangle.
        - Missed Block: Dark Grey rectangle.
        - Wire/Tether: A white line connects the center top of the canvas to the swinging block.
    - HUD
        - Text showing "Điểm: currentScore" (Score) is centered at the top of the screen in White color and large font.
    - Input
        - The entire screen is tap-sensitive to trigger the block drop.
    - Game Over Screen
        - Overlay: Semi-transparent Black background (alpha = 0.7f).
        - Text: "GAME OVER!" (Red, large font) and "Điểm cuối: currentScore" (Final Score, White).
        - Button: "CHƠI LẠI" (Play Again) button.
- Technical Solution for Function Development
    - Solution and Techniques Used
        - Development Technology: Android with Jetpack Compose.
        - Game Logic/Loop: The game relies on multiple decoupled coroutines using LaunchedEffect: one for the swinging motion and one for the drop animation/collision check.
        - Physics Simulation (Swinging): The swinging motion is achieved using the mathematical sin function, providing a simple, continuous, and periodic movement.

        - Rendering: The entire game visualization is handled by a single Canvas component. Position and size are calculated in pixels (Px) based on screen density.
        - State Management: Core game data is managed by Compose's reactive state variables (mutableStateOf), allowing for automatic UI updates during the swinging and dropping phases.

- Difficult Technical Issues
    - Calculating Dynamic Target Y-position: The target drop position (targetY) must be accurately calculated in real-time based on the accumulating height of the tower (blocks.size * blockHeight), which requires careful management of units (dp to px) and offsets (ground height).
    - Collision Formula for Overlap: The core of the game relies on the mathematical calculation of the overlap between the previous block and the new block's center:
    - Simulation physic: to simulating how the box drop and collapse
- Potential Future Developments
    - High Score Persistence: Implement SharedPreferences or Room Database to save the highest score achieved.
    - Difficulty Scaling: Increase the swingFrequency or swingAmplitude as the tower gets taller to make the game progressively harder.
    - Audio Feedback: Add sound effects for a successful placement (bell sound) and game over (crash sound).
    - Camera Movement: Implement vertical camera movement where the canvas shifts down as the tower grows taller, keeping the view centered on the highest blocks.

III. Final conclusion

And that is the end of this project report, i hope you find this report to your standard, understand our project in more detail and understand our passion for this game project.

We have had difficulties along the developing period, some have solution, some have alternations, but from everything that have happened, we understand our limit, our ability and most of all is our ability to go past that limit, like how we go out of our safe zone and experimenting with more modern approach like jetpack compose and new programming language (kotlin)

To be honest, as the group leader writing this, i am not satisfied with what we have done, not because of some foul aspect we all known, but it is my desire to perfect the project, everything that have been written in “Future Development Potential” will be revised and the project will continue to be review and improve, even when it does not matter if i do it or not