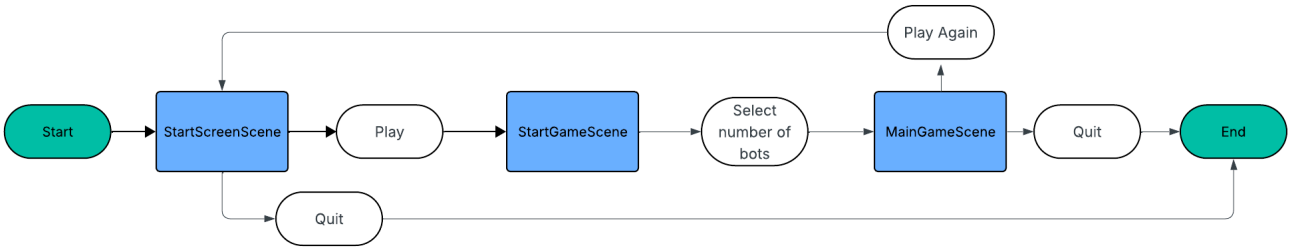
1. Brief Introduction of the Application/System

This project is a Unity-based digital version of the classic card game UNO. It lets one human player compete against AI-controlled opponents in a turn-based setting. The goal follows the standard UNO win condition: get rid of all cards in your hand by playing cards that match the discard pile either by color or by type, while using action cards to slow down or disrupt other players.

The game is designed for casual players who want a quick, familiar UNO experience on a digital platform. It includes the usual UNO card types such as number cards, Skip, Reverse, Draw Two, Wild, and Wild Draw Four. Players move through a simple menu flow where they can start a match, choose how many bots to play against, review the rules, and replay games as often as they want.

2. Architecture Diagram + Explanation of Components/Services



2a. Architecture Diagram

The overall system is split into three Unity scenes, with the main gameplay driven by a singleton **GameManager**. Players begin in **StartScreenScene**, move to **StartGameScene** to pick the number of bots, and then enter **MainGameScene** where the match runs. During gameplay, **GameManager** acts as the central controller for UNO rules, turn sequencing, the **Deck**, the discard pile, and all major UI updates. The Player class provides shared behavior for all players, and **AIPlayer** builds on that base class to handle bot decision making. Cards are defined through **CardData** ScriptableObjects, instantiated visually as **CardController** objects, and managed through each player's hand system.

2b. Component / Service Explanations

<u>Scenes</u>

- StartScreenScene

This is the entry point of the application and contains the main menu and rules UI.

- **MenuController** manages navigation (Start, Rules, Exit).
- **RuleManager** displays UNO rules.

- **SoundManager** plays menu audio and handles basic sound toggles.

- StartGameScene

This scene is used for setting up a match before gameplay begins.

- **GameSettings** persist across scenes and store settings such as AI/bot count.

- MainGameScene

This is where the actual UNO match takes place. It contains the full gameplay loop, card interactions, rule enforcement, and win handling.

<u>Core Gameplay Systems</u>

- GameManager (Singleton)

GameManager is the main coordinator for the match. It is responsible for:

- Building and shuffling a full UNO deck.
- Dealing starting hands to each player.
- Maintaining the discard pile and current top card.
- Enforcing valid-play rules through IsValidPlay.
- Managing turn order, draw/pass rules, and play direction.
- Triggering and resolving special-card effects (Skip, Reverse, Draw Two, Wild, Wild Draw Four).
- Checking win conditions and signaling the UI to show the winner screen.

- Deck

The **Deck** stores the full UNO card collection. It is built from **CardData** ScriptableObjects, shuffled at the start of each game, and used as the source for all draws. When a card is played, it moves from a player's hand into the discard pile controlled by **GameManager**.

- Player (Base Class)
  - The **Player** class represents any participant in the game. It maintains the player's hand and includes shared logic such as checking for playable cards, updating card layout, and managing hand visuals. The human player relies on this base behavior along with manual input.

The human player uses this base behavior with manual input.

- AIPlayer (Derived Class)

AIPlayer inherits from Player and adds automated turn logic. On a bot's turn, it:

- Scans its hand for valid plays.
- Prioritizes plays in this order: type match → color match → wild → wild draw four.
- Draws if no valid play exists.
- Selects a wild color based on the most frequent color in its hand.

<u>Card & Hand Systems</u>

- CardData (ScriptableObject)

Each UNO card is defined using **CardData**. These assets store card properties such as:

- Card color (Red/Blue/Green/Yellow/Wild).
- Card type (Number, Skip, Reverse, Draw Two, Wild, Wild Draw Four).

    - Display values and scoring info.

Using ScriptableObjects makes the deck data-driven and easy to expand.

- CardController (MonoBehaviour)

CardController is the runtime object that shows a card in-game and handles interaction. It:

    - Displaying card face and color.
    - Highlighting/dimming based on playability.
    - Handling user click input for human turns.
    - Calling GameManager.PlayCard() when a valid card is selected.
- Hand Manager

Hand Manager controls how cards are laid out in a player's hand. It manages spacing, ordering, and updating positions whenever cards are drawn or played.

<u>UI System</u>

- UIManager

UIManager controls all match-related UI elements in **MainGameScene**, including:

    - Turn prompts and status messages.
    - Discard pile visuals and top-card display.
    - Draw/Pass/UNO buttons.
    - Wild color-selection UI.
    - Pause panel and winner screen.

UIManager reacts to events/updates from GameManager.

<u>External Services</u>

- None.

The project runs completely locally. There is no networking, backend server, or database dependency.

3. Tech Stack (Frontend, Backend, Databases)

<u>Frontend /Client</u>

- Unity Engine: 2022.3.62f3 (LTS)
- Language: C#
- Unity Packages Used
- com.unity.ugui — UI system.
- com.unity.textmeshpro — text rendering.
- com.unity.feature.2d — 2D tooling (project uses 2D-style card UI).
- com.unity.collab-proxy — Unity version control integration.
- Standard Unity modules for audio, input, physics, etc.

4. Code Implementation

4a. Layout of the Code (Folder Tree)

Assets/

Scenes/
  StartScreenScene.unity
  StartGameScene.unity
  MainGameScene.unity
Scripts/
  MainGameScripts/
    GameManager.cs
    Deck.cs
    Player.cs
    AIPlayer.cs
    CardController.cs
    CardData.cs
    Hand Manager.cs
    UIManager.cs
    PauseManager.cs
    DOTweenInit.cs
  StartScreenScripts/
    MenuController.cs
    RulesManager.cs
    SoundManager.cs
    AspectRatio.cs
  StartGameScripts/
    GameSettings.cs
Plugins/
  Demigiant/DOTween/...
Prefabs/
Art/
Audio/
ScriptableObjects/

4b. Main Gameplay Flow

1. StartScreenScene
    - **MenuController** handles Start / Rules / Exit.
    - Selecting Start loads **StartGameScene**.
2. StartGameScene
    - The player chooses settings (bot count).
    - **GameSettings** persist selection across scenes.
    - Clicking Start loads **MainGameScene**.
3. MainGameScene Setup (GameManager.Start)
    - Creates a Deck from all CardData assets.
    - Shuffles deck.
    - Instantiates players:
        - **humanPlaye**r

        - AI players (left/top/right) based on settings.
    - Deals starting hands.
    - Flips initial discard card (cannot be Wild Draw Four).
    - Starts turn loop.
4. Turn Loop
    - **StartTurn()** decides if the active player is human or AI.
    - **Human turn**
        - UI enables playable cards using **IsValidPlay**.
        - User can:
            - click a valid card to play, or
            - click draw pile to draw (only once per turn), then play if possible, else pass.
    - AI turn
        - **AIPlayer.TakeTurnCoroutine()** executes:
            - finds best playable card
            - draws if none
            - plays or passes
            - chooses a wild color if needed.
5. Card Validation
    - **IsValidPlay(card)** rules:
        - Wild / WildDrawFour is always valid.
        - If a wild color is active, the card must match that color.
        - Otherwise match by color OR type with top discard.
6. Special Card Effects
    - Enforced in GameManager.PlayCard and action handlers:
        - Skip, Reverse, Draw Two, Wild, Wild Draw Four.
        - Wild cards trigger UI color pickers for humans; AI chooses color automatically.
7. Win Condition

- After every play, GameManager checks if a player's hand is empty.
- UIManager.ShowWinnerScreen() displays results.
- MenuController supports Play Again / Return to Start.

4c. Notable Classes and Their Roles

<u>GameManager</u>

- State: current player index, top card, active wild color, direction, draw-once flag, pause state.
- API:
    - PlayCard(Player, CardController)
    - OnDrawPileClicked()
    - IsValidPlay(CardData)
    - turn transition helpers.

- AIPlayer
    - Overrides/extends Player behavior with coroutine turn.
    - Uses playable priority logic and wild selection.
- CardController
    - Mediates user interaction.
    - Updates visuals based on playability and hand activity.

5. Test plans, cases, validation, and report

1. Test Plan

1.1 Introduction

This test plan is aimed at defining the strategy, scope and resources necessary to ensure that our UNO Game functionality is validated. It is concerned with making the core game loop (Draw, Play, Win) work and making sure that the AI is acting within the regulations of UNO.

1.2 Scope of Testing

- **In-Scope:**
    - **GameManager Logic:** Rule enforcement (`IsValidPlay`), Turn management, and Win conditions.
    - **UI Logic:** Button visibility (UNO button), dynamic text updates (Status text), and menus.
    - **AI Behavior:** Automated turn-taking and UNO call probability.
    - **Integration:** Interaction between the 2D UI and 3D Card objects.
- **Out-of-Scope:**
    - Networked multiplayer (Local/Single player only).
    - Performance testing on mobile hardware (PC/Editor focus only).

1.3 Testing Tools & Environment

- **Engine:** Unity 2022.3 LTS.
- **Framework:** Unity Test Framework (based on NUnit) for Unit/Integration tests.
- **Manual Testing:** Play Mode tests within the Unity Editor.
- **Environment:** Windows 10/11 Desktop.

1.4 Test Strategy

1. **Unit Testing:** Verify individual logic methods (e.g., `GetNextPlayer`, `IsValidPlay`) using automated scripts.
2. **Integration Testing:** Verify the handshake between `GameManager` and `UIManager` (e.g., clicking "Draw" updates the hand *and* the UI text).

3. **Regression Testing:** Re-running tests after bug fixes (specifically the UNO button fix and Pause menu Z-fighting) to ensure no new bugs were introduced.

2. Validation (Requirements Traceability)

This section validates that the software meets the intended User Requirements (Rules of UNO). It maps the Game Rules to the specific Test Cases (TC) that prove they work.

Requirement ID	Requirement Description	Validated By (Test Case)	Validation Status
REQ-01	**Legal Move Validation:** A player can only play a card if it matches the color, number, or is a Wild card.	**TC-1.1, TC-1.2, TC-1.3**	**PASS**
REQ-02	**Illegal Move Prevention:** The system must reject cards that do not match the discard pile criteria.	**TC-1.4, TC-1.5**	**PASS**
REQ-03	**UNO Call Rule:** The "UNO" button must *only* be available when the player has 2 cards and can play one.	**TC-2.1, TC-2.2**	**PASS**
REQ-04	**Turn Order:** The game must progress clockwise and handle the "Reverse" card correctly.	**TC-3.1, TC-3.2**	**PASS**
REQ-05	**AI Autonomy:** AI players must take turns without human input and play valid cards.	**Play Mode Test** (Manual)	**PASS**
REQ-06	**Pause Functionality:** The game must completely stop logic execution (AI turns) when paused.	**Play Mode Test** (Manual)	**PASS**

2.1 Validation Summary

The validation process confirms that the internal logic of `GameManager` enforces the official rules of UNO.

- **Verification:** The code was verified via `UnoGameTests.cs` (Unit Tests).
- **Validation:** The gameplay was tested by playing a full match with 3 bots in order to be sure that the game flow is as expected as it was supposed to be. The particular solution of the UNO Button (REQ-03) was also proven and players will no longer be asked to call UNO when they cannot make a move.

3. System Overview & Component Interaction

The system relies on a central `GameManager` (Singleton) that coordinates interactions between `Player` objects (Human and AI), a `Deck` system, and a `UIManager`.

- **GameManager:** Holds the state (Current Color, Discard Pile, Turn Order).
- **Player/AIPlayer:** Holds `CardController` objects and makes decisions.
- **UI:** Reacts to state changes (buttons, indicators).

4. Test Cases

Function 1: `GameManager.IsValidPlay(CardData card)`

Description: Determines if a card selected by the player can be legally played on the current `topCard`.

ID	Type	Test Scenario	Input Data	Expected Outcome
TC-1.1	Functionality	Same Color Match	Input: Red 5 TopCard: Red 8	**True** (Valid Play)
TC-1.2	Functionality	Same Number Match	Input: Blue 7	**True** (Valid Play)

			TopCard: Green 7	
TC-1.3	Functionality	Wild Card on anything	Input: Wild Card TopCard: Yellow 2	**True** (Valid Play)
TC-1.4	Boundary	Wild Color Mismatch	Input: Blue 4 TopCard: Wild (Red declared)	**False** (Must match declared color)
TC-1.5	Error Handling	Invalid Suit & Rank	Input: Yellow 9 TopCard: Red 4	**False** (Invalid Play)

Function 2: `UpdateUnoButtonVisibility()` (Logic Test)

Description: Checks the conditions under which the "UNO" button appears for the human player. *Note: This targets the specific bug fix where the button appeared even if no cards were playable.*

ID	Type	Test Scenario	Input Data	Expected Outcome
TC-2.1	Functionality	2 Cards, 1 Playable	Hand: [Red 5, Blue 2] TopCard: Red 9	**True** (Button Visible)
TC-2.2	Functionality	2 Cards, 0 Playable	Hand: [Yellow 2, Blue 2] TopCard: Red 9	**False** (Button Hidden - *Fix Verified*)

TC-2.3	Boundary	1 Card Remaining	Hand: [Red 5]	**False** (Too late to call, or already called)
TC-2.4	Boundary	3 Cards Remaining	Hand: [Red 5, Red 6, Red 7]	**False** (Too many cards)
TC-2.5	Error Handling	Null Hand/Empty	Hand: Null or Count 0	**False** (Handle exception gracefully)

Function 3: `GetNextPlayer()` / Reverse Logic

Description: Calculates which player acts next, handling the "Reverse" card logic and array wrapping.

ID	Type	Test Scenario	Input Data	Expected Outcome
TC-3.1	Functionality	Standard Turn	Index: 0 (Human), Direction: Normal	Returns Index 1 (Bot Left)
TC-3.2	Functionality	Reverse Turn	Index: 0 (Human), Direction: Reversed	Returns Index 3 (Bot Right)
TC-3.3	Boundary	Array Wrap-Around (End)	Index: 3 (Bot Right), Direction: Normal	Returns Index 0 (Human)
TC-3.4	Boundary	Array Wrap-Around (Start)	Index: 0 (Human), Direction: Reversed	Returns Index 3 (Bot Right)

TC-3.5	Error Handling	1 Player Game	Index: 0, Players: 1	Returns Index 0 (Self)

5. Introduction to Testing Report

This report records the testing procedure of the UNO Game. This was to test the fundamental logic of the game, user interface responsiveness and artificial intelligence behaviors. We narrowed our attention to the aspects of race conditions (spam clicking), logical fallacies of the UNO button visibility, and 2D UI Vs 3D game object rendering.

A. Testing Process

Testing was conducted using a hybrid approach:

1. **Automated Unit Testing:** Created NUnit test scripts (see `UnoGameTests.cs`) to verify the `IsValidPlay` logic and turn calculation math.
2. **Play Mode Integration Testing:** Manual execution of edge cases within the Unity Editor to test visual transitions and input locking.

B. Findings & Anomalies

Failure 1: The "Unplayable UNO" Bug

- **Observation:** The "UNO" button appeared whenever the player had 2 cards, even if neither card was actually playable. This would confuse the player, as they would click UNO but then be forced to draw a card anyway.
- **Status: Fixed.** Logic was added to `UpdateUnoButtonVisibility` to iterate through the hand and verify `IsValidPlay` returns true for at least one card. (See Test Case TC-2.2).

Failure 2: Input Race Condition

- **Observation:** Users could click two playable cards in rapid succession before the first cards animation finished going into the discard pile. This resulted in playing two cards in a single turn which should not be allowed.
- **Status: Fixed.** A boolean flag `isProcessingMove` was made. Input is now locked immediately upon valid click and unlocked only when the next turn begins.

Failure 3: Object Priority Issue on Pause

- **Observation:** When the "Rules" menu (2D Canvas) was opened, the 3D card models in the player's hand clipped through the menu text.
- **Status: Fixed.** Implemented a `SetAllHandsVisibility(false)` function in the Game Manager that disables the `Renderer` component of all cards when full-screen UI is active, essentially making them temporarily invisible.

Failure 4: Pause Menu Leakage

- **Observation:** Pressing pause stopped the game timer, but the AI player continued to take its turn in the background.
- **Status: Fixed.** Added `while(isPaused)` checks within the AI's `TakeTurnCoroutine` to halt execution flow until the game resumes.

C. Recommendations for Improvement

Based on the testing results, the following recommendations are proposed:

1. **Strict State Machine:** Currently, the game relies on boolean flags (`isPaused`, `isProcessingMove`). Migrating to a formal State Machine (e.g., `GameState.PlayerTurn`, `GameState.Animating`, `GameState.Paused`) would reduce the risk of any future similar problems.
2. **Decouple UI from Logic:** The `GameManager` currently handles too much UI logic directly. Moving visual updates to a dedicated `EventManager` system would make unit testing easier, as we wouldn't need to mock the entire UI system to test simple game rules.
3. **Automated UI Tests:** Implement Unity's UI Test Framework to automate the clicking of buttons (Draw, Pass, Uno) to ensure buttons appear/disappear correctly without manual verification.

6. Functional Requirements

1. Game Setup
- The system shall initialize a standard 108-card UNO deck.
- The system shall deal a starting hand to each player.
- The system shall place one valid opening card on the discard pile (not a Wild Draw Four).
2. Card Play Rules
- The system shall allow playing a card if it matches discard color or type.
- The system shall always allow Wild and Wild Draw Four.
- When a wild color is active, the system shall only allow cards of that color.

3. Drawing and Passing
- The system shall allow a player to draw from the deck once per turn if no play is made.
- The system shall allow passing only when no valid play exists after drawing.

4. Special Card Effects
- The system shall enforce Skip, Reverse, Draw Two, Wild, and Wild Draw Four effects.
- The system shall prompt for wild color selection after Wild/Wild Draw Four.

5. AI Gameplay
- The system shall support AI opponents.
- AI shall choose cards using priority: type match → color match → wild → wild draw four.
- AI shall choose wild colors based on the most common color in hand.

6. Win Condition
- The system shall end the game when a player has zero cards.
- The system shall display a winner screen and allow restarting.

7. GITHUB

https://github.com/EmilianoManaloIV/Createngineers-Projects-CPSC-362

8. User Manual

Installation / Running

- Open the project in Unity 2022.3.62f3.
- Load **StartScreenScene** and press Play in editor, or build for your target platform.
- Ensure DOTween plugin is imported (already in Assets/Plugins/).

Main Menu (StartScreenScene)

- Start: goes to the game setup scene.
- Rules: shows UNO rules overlay.
- Exit: quits application.

Game Setup (StartGameScene)

- Select number of AI opponents (stored in **GameSettings**).
- Click Start Game to begin.

Gameplay Controls (MainGameScene)

- Play a card: click any highlighted/playable card in your hand.
- Draw: click the draw pile or Draw button when enabled.
- Pass: click Pass when enabled after drawing and having no valid play.
- UNO button: appears when you have 2 cards and at least one is playable; click before playing your second-to-last card.
- Wild color picker: appears after playing Wild/Wild Draw Four; click desired color.

Pause / End Screens

- Pause menu: appears via UI (project supports fade in/out panels).
- Winner screen: shows winner and provides:

- Play Again → StartGameScene
- Main Menu → StartScreenScene

11. Deployment

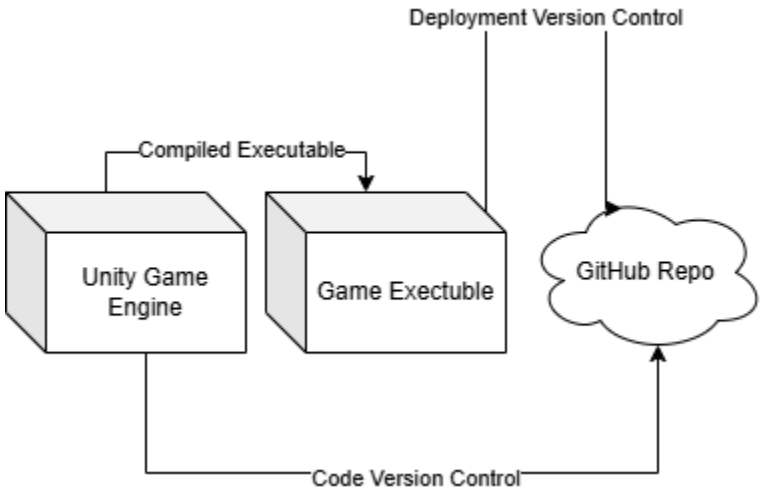
The deployment process involves compiling the Unity game engine editor into a compatible executable file. This executable file type is then updated with the latest version of the main branch. This way, anyone pulling or forking the GitHub repository can access the code and experiment with their own version of the source files as they wish.

To access this game, the following criteria have to be met:

- Be on a Windows 64-bit machine
- Download the UNOVerse Game File

Then, double-click the executable “UNOVerse”, and the application should open.

For our CD/CI pipeline, we chose GitHub as our main source of patching and uploading our application. Through the branch system, we were able to make changes without affecting the main deployed branch. This way, we can update the game collaboratively, and when we feel ready, we upload the executable file.



10. Lessons Learned

Working with **CardData** ScriptableObjects made it much easier to build and maintain a full UNO deck without hard-coding values into scripts. Having a singleton **GameManager** also helped keep rule enforcement consistent for both human and AI turns. Using coroutines for AI behavior (**TakeTurnCoroutine**) gave the bots natural pacing without freezing the game loop. Finally, clear UI feedback (highlighting playable cards and dimming invalid ones) turned out to be essential for making the game feel smooth and understandable.

11. Future Work

A strong next step would be adding multiplayer, either locally or online, by abstracting player input and introducing networking. Another improvement would be persistent tracking for player statistics and match history. The AI could also be expanded into multiple difficulty levels with more strategic play. Additional UNO rule variants (stacking draws, jump-ins, house rules) would make the game more customizable, and more UI polish like drag-to-play, improved discard animations, and better accessibility indicators would further refine the experience.