Developing "Turf War": A Step-by-Step Guide for Junior Unity Developers

This report provides a comprehensive, step-by-step guide for developing "Turf War," a virtual tactical card game, using the Unity engine and C#. Tailored for junior developers, it offers a practical roadmap from initial project setup to implementing core gameplay mechanics, user interface, visuals, and audio. The approach emphasizes modularity, data-driven design utilizing ScriptableObjects, and clear explanations of every code snippet and Unity concept. This guide aims to empower a junior developer to follow along, understand the underlying logic, and build a functional prototype, while also laying groundwork for future expansion into a robust, scalable interactive experience.

1. Laying the Foundation: Unity Setup & Project Structure

Establishing a solid foundation is paramount for any game development project, especially one with the strategic depth and potential for expansion like "Turf War." This section guides through the initial Unity environment setup and the establishment of a robust project architecture.

1.1. Understanding "Turf War": Game Concept & Core Mechanics

"Turf War" is conceptualized as a virtual Tactical Card Game, functioning as a turn-based strategy Player vs. Player (PVP) experience.[1] The game's core mechanics revolve around strategic deployment of units, card collection, deck building, and guild participation, with a Player vs. Environment (PVE) element also envisioned.[1] The primary objective for players is to capture three secretly assigned turfs on a randomly generated hexagonal board within a 15-20 minute time limit.[1] In the event of a draw on assigned turfs, the player with more total turfs claims victory.[1]

The game consists of four primary elements:

- **The Board:** Described as a flat drawing with a fixed border, housing 36 hexes that are filled with randomly distributed "blocks." The visual aspiration is a "stair" effect, where rows on the bottom appear on top, creating a smooth 3D plane.[1] An isometric map of a town, with seamlessly merging tiles, is the visual goal.[1]
- **The Cards:** Two main types are defined. "Deploy Cards" allow players to place units (tokens) on the map, ranging from levels 2-4, and can only be played once per turn during the player's turn. These cards are intended to have a pastel color pattern to convey a less urgent feel, forming the bulk of the player's army.[1] "Event Cards," conversely, enable "game-changing maneuvers" and can be played on the player's turn or reactively during other players' turns within a "reaction window." These are designed with more vibrant colors to emphasize urgency.[1]
- **The Tokens:** These are objects spawned from deploy cards onto chosen blocks. There are four levels of tokens (1-4), with larger tokens capable of being placed over smaller tokens or empty blocks, effectively removing the existing token. Level 3 tokens are special, used to deploy unique units with game-altering effects, while the Level 4 token, known as the "alpha," is the single strongest and has only one use per player.[1]
- **The Leader:** An icon chosen before the game that grants the player a passive ability throughout the game. A common ability mentioned is the free deployment of one rank 1 token, with unique abilities yet to be decided.[1]

The gameplay loop initiates with a setup phase where the board is randomly generated, each player is secretly revealed 3 objective blocks, players draw 4 cards, a coin flip determines the first player, and players pick their leader and alpha token. Decks are auto-shuffled, and 3 random turfs are secretly assigned to each player.[1] During a player's turn, they draw one card and can perform actions at will and in any order: play one deploy card, use available leader skills, and play as many event cards as desired. Each interaction pauses the turn for 10 seconds, allowing the other player to respond.[1] The game concludes when the time limit is reached, with scoring based on turf capture.[1] Abilities include a wide array of card effects such as Draw, Shuffle, Deploy, Remove, Negate, Immunity, Deny, Knock back, Recycle, Discard, Peek, Skill, Copy, and Decoy, alongside various map location ideas with unique tile effects.[1]

1.2. Setting Up Your Unity Project

The journey begins by setting up the Unity development environment. First, download and install the Unity Hub, a crucial tool that streamlines Unity installations, project management, and version control.[2] From within the Hub, install the latest Long-Term Support (LTS) version of the Unity Editor. When installing, it is important to select all necessary modules, such as build support for target platforms like Windows, Mac, Linux, and WebGL, as well as documentation, to ensure a comprehensive development environment.[3]

Once the Unity Editor is installed, proceed to create a new 3D project. In the Unity Hub, select "New project" and choose a "3D Core" or "3D Sample" template. A critical practice at this stage is to name the project without spaces and avoid excessively long file paths. Adhering to these naming conventions helps prevent potential errors and simplifies project management, especially as the project grows.[3] Upon the project opening in the Unity Editor, immediately create a new scene by navigating to

File > New Scene and save it in a dedicated Scenes folder within your project's Assets directory. A newly created scene typically includes a Main Camera and a Directional Light, providing a basic starting point for development.[3]

Choosing the right engine and establishing a proper initial structure are foundational decisions that significantly impact a game's future. Unity, with its versatility for both 2D and 3D development, C# scripting, extensive asset store, and large community support, is particularly well-suited for a junior developer embarking on a project like "Turf War".[4] While other engines might offer higher-end graphics capabilities, Unity's accessibility and robust community resources provide a more supportive learning curve and development environment.[5] The "Turf War" concept explicitly mentions elements such as card collection, guilds, deck building, and a Player vs. Environment (PVE) mode.[1] These features are often characteristic of "Live Service Games" (LSGs), which rely on continuous updates and long-term player engagement.[7] This implies that "Turf War" is designed to evolve over time, making a flexible engine and a modular architecture essential from day one. By prioritizing correct initial setup and architecture, the project is positioned for future scalability, maintainability, and the ability to expand into a live service model without requiring extensive and costly refactoring down the line. This proactive approach instills good development habits and provides a clear understanding of the underlying rationale behind best practices.

1.3. Establishing a Robust Project Architecture

A well-organized project architecture is vital for maintaining clarity and efficiency throughout development. Begin by organizing the Assets folder into logical sub-folders such as Scripts, Prefabs, Materials, Models, Textures, Audio, Scenes, ScriptableObjects, and UI. This structured approach ensures that assets are easy to locate and manage, which is particularly beneficial in larger projects.[10]

From the outset, adopt modular design principles. This involves creating assets and systems that can be easily rearranged, combined, or replaced without affecting other parts of the game.[11] For example, each game object or system should ideally encapsulate its own functionality.[13] This practice is critical for ensuring the project's scalability and facilitating collaboration, even in a solo development scenario.

A fundamental principle to implement is the separation of concerns, akin to a Model-View-Controller (MVC) pattern. This means striving to keep game logic distinct from its visual representation and user interface elements.[14] The core game rules and state should operate independently of how they are rendered or displayed on screen. This separation is especially critical for future multiplayer implementation, as it forms a strong defense against cheating by ensuring that the server, not the client, is the ultimate authority on game state.[14]

The "Turf War" concept explicitly defines the game as a "turn base strategy PVP" experience.[1] This inherent multiplayer aspect necessitates designing the game with networking in mind from the very beginning. Attempting to integrate multiplayer functionality after a game has been developed for single-player can be a significant challenge, often requiring extensive rewrites.[15] A core best practice for multiplayer games is to separate game logic from visuals and UI, and to employ an authoritative server model where the server maintains the definitive game state to prevent client-side manipulation.[14] Modular design naturally supports this separation by encouraging the creation of self-contained components and systems.[11] By structuring the project with multiplayer and modularity as core considerations from day one, the development effort is not merely focused on building a prototype, but on laying the robust groundwork for a competitive online game. This foresight helps prevent costly and time-consuming refactors in the future, directly addressing the "PVP" requirement of "Turf War" and preparing the game for a live service model.

2. Building the Game's Core Data Systems

This section focuses on creating the foundational data structures that will define all game elements, ensuring flexibility, easy balancing, and maintainability.

2.1. Centralizing Game Data with ScriptableObjects

ScriptableObjects are serializable Unity classes specifically designed to store large quantities of shared data independently from script instances.[13] Unlike MonoBehaviour scripts, which must be attached to a GameObject in a scene, ScriptableObjects are saved as assets directly within the project.[18]

Their utility for "Turf War" is multifaceted:

- **Data-Driven Design:** ScriptableObjects empower game designers to define and balance various game elements—such as card statistics, token properties, and leader abilities—directly within the Unity Inspector without needing to modify source code.[13] This is particularly advantageous for a card game, which typically involves a large number of unique entities and frequently evolving mechanics.[1]
- **Memory Efficiency:** Instead of duplicating data across numerous MonoBehaviour instances (e.g., every card on the board holding its own copy of an "Attack" value), ScriptableObjects store data once, and all instances reference that single, shared source in memory.[18] This significantly reduces memory footprint.
- **Modularity & Reusability:** They inherently promote a modular development approach by clearly separating data from game logic, which makes it easier to reuse components and systems across different parts of the game.[13] For example, a
  CardDisplay script can be designed to dynamically render information from any CardDataSO asset it receives.

To begin, a base ScriptableObject class can be created to provide common properties for all game data assets:

C#

```
// Scripts/ScriptableObjects/Base/BaseGameDataSO.cs
using UnityEngine;

// This is a base class for all our game data ScriptableObjects.
// It provides common properties like a unique ID and a display name.
public abstract class BaseGameDataSO : ScriptableObject
{
    // The attribute exposes the private backing field
    // of the auto-property in the Unity Inspector, allowing it to be set there.
    // 'public string Id { get; private set; }' defines an auto-property
    // that can be read publicly but only set internally within the class.
    public string Id { get; private set; } // Unique identifier for this data asset
    public string DisplayName { get; private set; } // Name to show in UI

    // OnValidate is a Unity callback method invoked when the script is loaded
    // or a value is changed in the Inspector. It's useful for validation.
    // Here, it ensures that if the 'Id' field is left empty, it automatically
    // defaults to the asset's file name, promoting data consistency.
    protected virtual void OnValidate()
    {
        if (string.IsNullOrEmpty(Id))
        {
            Id = name; // Auto-assign name as ID if not set
        }
    }
}
```

2.2. Designing Card, Token, and Leader Data Structures

Building upon the BaseGameDataSO, specific ScriptableObjects are defined for the core game elements, allowing for detailed property definition and inter-referencing.

CardDataSO (for Deploy and Event Cards)

This ScriptableObject defines the properties common to all cards, as well as specific fields for Deploy and Event card types.

C#

```
// Scripts/ScriptableObjects/CardDataSO.cs
using UnityEngine;
using TMPro; // Required for TextMeshProUGUI if used in UI

// The CreateAssetMenu attribute adds an option to the Unity Editor's "Create" menu,
// allowing easy creation of instances of this ScriptableObject asset.
// fileName: default name for new asset. menuName: path in Create menu. order: position in menu.

public class CardDataSO : BaseGameDataSO
{
    // Enums provide a way to define a set of named integral constants.
    // Here, they categorize card types and factions, improving readability and maintainability.
    public enum CardType { Deploy, Event } // Distinguish between card types [1]
    public enum Faction { Cats, Dogs, Neutral } // Factions from lore [1]

    // Header attributes organize fields in the Inspector, making complex ScriptableObjects easier to
manage.
    [Header("Card Properties")]
    private CardType cardType; // The type of this card (Deploy or Event)
    private Faction faction; // The faction this card belongs to
    private Sprite cardArtwork; // The visual sprite/image for the card's artwork
    private string cardDescription; // A multi-line text area for the card's description or effect


    private int manaCost; // The cost to play this card, potentially for a mana-based system [1]

    // Fields specific to Deploy cards:
    // A direct reference to a TokenDataSO asset, linking a Deploy card to the type of token it creates.
    // This demonstrates how ScriptableObjects can reference each other, building a data graph.
    private TokenDataSO tokenToDeploy;
    // Fields specific to Event cards:
    // An array of AbilityDataSO objects, allowing an Event card to trigger multiple effects.
    private AbilityDataSO abilities;

    // Public properties (read-only) to access the private serialized fields.
    // This is good practice for encapsulation.
    public CardType Type => cardType;
    public Faction CardFaction => faction;
    public Sprite Artwork => cardArtwork;
    public string Description => cardDescription;
    public int ManaCost => manaCost;
    public TokenDataSO TokenToDeploy => tokenToDeploy;
    public AbilityDataSO Abilities => abilities;
}
```

TokenDataSO

This ScriptableObject defines the characteristics of the tokens that are deployed onto the game board.

C#

```
// Scripts/ScriptableObjects/TokenDataSO.cs
using UnityEngine;


public class TokenDataSO : BaseGameDataSO
{
    // Enum for different token levels as described in the game design.[1]
    public enum TokenLevel { Level1, Level2, Level3, Level4_Alpha }


    private TokenLevel tokenLevel; // The level of this token
    private GameObject tokenPrefab; // The GameObject Prefab representing the visual of this
token
    private int attackValue; // Example: The attack power of the token
    private int healthValue; // Example: The health points of the token
    private bool isSpecialUnit; // Flag for Level 3 tokens, indicating special effects [1]
    private bool isAlphaToken; // Flag for the unique Level 4 Alpha token [1]

    // Public properties for read-only access.
    public TokenLevel Level => tokenLevel;
    public GameObject Prefab => tokenPrefab;
    public int Attack => attackValue;
    public int Health => healthValue;
    public bool IsSpecial => isSpecialUnit;
    public bool IsAlpha => isAlphaToken;
}
```

LeaderDataSO

This ScriptableObject defines the properties of the player's chosen leader.

C#

```
// Scripts/ScriptableObjects/LeaderDataSO.cs
using UnityEngine;


public class LeaderDataSO : BaseGameDataSO
{
    [Header("Leader Properties")]
    private Sprite leaderIcon; // The visual icon representing the leader
    private string passiveAbilityDescription; // Description of the leader's passive ability
    private AbilityDataSO passiveAbility; // Reference to the AbilityDataSO defining the passive
ability

    // Public properties for read-only access.
    public Sprite Icon => leaderIcon;
    public string PassiveDescription => passiveAbilityDescription;
    public AbilityDataSO PassiveAbility => passiveAbility;
}
```

AbilityDataSO (for generic abilities/effects)

This ScriptableObject serves as a flexible template for various in-game abilities and effects, allowing for diverse gameplay interactions.

C#

```
// Scripts/ScriptableObjects/AbilityDataSO.cs
using UnityEngine;


public class AbilityDataSO : BaseGameDataSO
{
    // Enums to categorize when an ability triggers and what type of effect it has,
    // directly mapping to the "Abilities" section of the game design.[1]
    public enum AbilityTriggerType { OnPlay, OnDeploy, OnTurnStart, OnTurnEnd,
OnReaction }
    public enum AbilityEffectType { Draw, Shuffle, Deploy, Remove, Negate, Immunity,
Deny, Knockback, Recycle, Discard, Peek, Copy, Decoy, UpgradeToken, RollDice }

    [Header("Ability Properties")]
    private AbilityTriggerType triggerType; // When this ability activates
    private AbilityEffectType effectType; // The type of effect this ability performs
    private int effectValue; // A generic value (e.g., number of cards to draw, spots to knock back)
    private float duration; // For effects with a time duration (e.g., Immunity, Deny)
    private string targetTag; // For abilities targeting specific types of tokens or tiles

    // Public properties for read-only access.
    public AbilityTriggerType Trigger => triggerType;
    public AbilityEffectType Effect => effectType;
    public int Value => effectValue;
    public float Duration => duration;
    public string TargetTag => targetTag;
}
```

The approach of utilizing ScriptableObjects for core game data is a powerful architectural choice. The "Turf War" document details numerous card effects and token types.[1] Without a data-driven system, hardcoding these elements would lead to an unmanageable and inflexible codebase. ScriptableObjects are specifically designed for efficient storage of static data.[13] Their ability to reference one another (e.g., a

CardDataSO referencing a TokenDataSO or AbilityDataSO) creates a robust, interconnected data graph. This allows game designers to create new cards, tokens, and abilities, and even combine them in novel ways, directly within the Unity Editor without requiring a single line of C# code for new content.[13] This method also significantly aids in debugging, as data can be inspected and modified at runtime, with changes persisting outside of Play mode.[13] This data-driven approach dramatically reduces development time for content creation and balancing, empowers non-programmers (such as game designers), and makes the game highly extensible. This flexibility is a cornerstone for building a "live service game," where new content is introduced regularly to maintain player engagement.[8]

2.3. Implementing a Game State Management System

Turn-based games inherently follow a sequence of distinct states, such as Game Setup, Player Turn, Opponent Turn, Reaction Phase, and Game End. A Finite State Machine (FSM) pattern is an ideal architectural choice for managing these transitions, ensuring that only valid actions can occur within a given state.[19] This pattern compartmentalizes game logic, leading to cleaner, more maintainable code that is easier to debug.[20]

A central GameManager script will orchestrate the overall game flow. This manager should be implemented using the Singleton pattern, which ensures that there is only one instance of the GameManager accessible globally throughout the application.[22]

C#

```
// Scripts/Managers/GameManager.cs
using UnityEngine;
using System.Collections; // Required for Coroutines for time-based operations
using System.Collections.Generic; // For List<T>

public class GameManager : MonoBehaviour
{
    // Singleton instance: This static property provides a global access point to the single GameManager
instance.
    // 'get; private set;' ensures it can be read publicly but only set within this class.
    public static GameManager Instance { get; private set; }

    // PlayerID enum can be used to distinguish between players in a multiplayer context.
    public enum PlayerID { Player1, Player2 }

    // GameState enum defines the distinct phases of the game, crucial for turn-based logic.
    public enum GameState
    {
        GameSetup,      // Initial phase to set up the board, decks, etc.
        PlayerTurn,     // Active player's main action phase
        OpponentTurn,   // Non-active player's phase (AI or remote player)
        ReactionPhase,  // Special phase for reactive plays [1]
        GameEnd         // Game conclusion, scoring, and winner declaration
    }

    // exposes the CurrentState property in the Inspector for easy monitoring.
    public GameState CurrentState { get; private set; }

    // C# event: A powerful way to implement loose coupling. Other scripts can subscribe to this event
    // to be notified when the game state changes, without needing direct references to the
GameManager.
    public static event System.Action<GameState> OnGameStateChanged;

    // References to other managers/systems (assigned in Inspector)
    private HexGridGenerator hexGridGenerator;
    private CardManager cardManager;
    private TurnManager turnManager; // Reference to the TurnManager script
    private LeaderDataSO defaultLeaderSO; // For initial leader assignment
    private TokenDataSO defaultAlphaTokenSO; // For initial alpha token assignment
    private List<CardDataSO> allAvailableCards; // All cards in the game, for deck initialization

    // Player-specific data (simplified for prototype)
    public List<CardDataSO> Player1Deck = new List<CardDataSO>();
    public List<CardDataSO> Player2Deck = new List<CardDataSO>();
    public List<Hex> player1Turfs = new List<Hex>();
    public List<Hex> player2Turfs = new List<Hex>();
    public LeaderController player1Leader; // Reference to Player 1's LeaderController
    public TokenController player1TokenAlpha; // Reference to Player 1's Alpha TokenController
    // Add similar for Player 2

    private bool isPlayer1Turn; // Tracks whose turn it is

    private void Awake()
    {
        // Singleton enforcement: Ensures that only one GameManager instance exists.
        // If another instance is found, destroy this one.
        if (Instance!= null && Instance!= this)
        {
            Destroy(gameObject);
        }
        else
        {
            Instance = this;
            // DontDestroyOnLoad keeps the GameManager GameObject alive across scene loads,
            // useful if it manages global game state from a main menu to game scene.
            DontDestroyOnLoad(gameObject);
        }
    }

    private void Start()
    {
        // Start the game in the initial setup state.
        UpdateGameState(GameState.GameSetup);
    }
    // Central method to change the game's state.
    // It triggers specific handler methods and notifies all subscribers.
    public void UpdateGameState(GameState newState)
    {
        CurrentState = newState; // Update the current state variable

        // Use a switch statement to execute logic specific to each game state.
        switch (newState)
        {
            case GameState.GameSetup:
                StartCoroutine(HandleGameSetup()); // Start setup as a coroutine
                break;
            case GameState.PlayerTurn:
                turnManager.StartPlayerTurn(); // Delegate turn handling to TurnManager
                break;
            case GameState.OpponentTurn:
                turnManager.StartOpponentTurn(); // Delegate turn handling to TurnManager
                break;
            case GameState.ReactionPhase:
                // ReactionHandler will subscribe to OnGameStateChanged and manage this phase
                break;
            case GameState.GameEnd:
                HandleGameEnd();
                break;
            default:
                Debug.LogError("Unknown game state: " + newState);
                break;
        }

        // Invoke the event, notifying all listening scripts about the state change.
        OnGameStateChanged?.Invoke(newState);
    }

    // Placeholder methods for state-specific logic, actual implementation details will be in other
managers.
    private IEnumerator HandleGameSetup()
    {
        Debug.Log("Game State: Setting Up Game...");
        // 1. Generate Hex Grid and Distribute Blocks [1]
        // Assuming hexGridGenerator is assigned in the Inspector.
        hexGridGenerator.GenerateGrid();
        hexGridGenerator.DistributeBlocksRandomly();
        yield return new WaitForSeconds(1f); // Simulate some loading time or animation

        // 2. Initialize Player Decks (simplified for prototype)
        // In a full game, this would load from player's collection data.
        Player1Deck = new List<CardDataSO>(allAvailableCards); // Example: populate with all
available cards
        Player2Deck = new List<CardDataSO>(allAvailableCards);
        ShuffleDeck(Player1Deck);
        ShuffleDeck(Player2Deck);
        Debug.Log("Decks shuffled.");
        yield return new WaitForSeconds(0.5f);

        // 3. Draw Starting Hand [1]
        for (int i = 0; i < 4; i++)
        {
            cardManager.DrawCard(DrawCardFromDeck(Player1Deck)); // Assuming
DrawCardFromDeck exists in GameManager
            cardManager.DrawCard(DrawCardFromDeck(Player2Deck)); // Draw for Player 2 as
well
        }
        Debug.Log("Starting hands drawn.");
        yield return new WaitForSeconds(0.5f);

        // 4. Random Coin Flip for First Player [1]
        isPlayer1Turn = (Random.Range(0, 2) == 0);
        Debug.Log($"Coin flip: Player {(isPlayer1Turn? "1" : "2")} goes first.");
        yield return new WaitForSeconds(0.5f);

        // 5. Leader and Alpha Selection (Simplified: assume pre-selected or basic UI interaction) [1]
        // For prototype, you might just assign default leaders/alphas.
        // Ensure player1Leader and player1TokenAlpha GameObjects exist in scene and are referenced.
        player1Leader.Initialize(defaultLeaderSO);
        // Place alpha on a starting hex (e.g., center hex or a specific spawn point).
        // This Hex(0,0) is an example, you'd need a proper spawn point.
        player1TokenAlpha.Initialize(defaultAlphaTokenSO, new Hex(0, 0));
        //... similar initialization for Player 2
        Debug.Log("Leaders and Alphas selected.");
        yield return new WaitForSeconds(0.5f);

        // 6. Secret Turf Assignment (Simplified: assign random hexes as turfs) [1]
        AssignRandomTurfs(player1Turfs, 3);
        AssignRandomTurfs(player2Turfs, 3);
        Debug.Log("Turfs assigned secretly.");
        yield return new WaitForSeconds(0.5f);

        Debug.Log("Game Setup Complete. Starting First Turn.");
        // Transition to the first player's turn.
        UpdateGameState(isPlayer1Turn? GameState.PlayerTurn :
GameState.OpponentTurn);
    }

    // Helper method to shuffle a list (Fisher-Yates shuffle algorithm)
    private void ShuffleDeck(List<CardDataSO> deck)
    {
        for (int i = deck.Count - 1; i > 0; i--)
        {
            int rnd = Random.Range(0, i + 1);
            CardDataSO temp = deck[i];
            deck[i] = deck[rnd];
            deck[rnd] = temp;
        }
    }

    // Helper method to draw a card from the top of the deck
    private CardDataSO DrawCardFromDeck(List<CardDataSO> deck)
    {
        if (deck.Count == 0)
        {
            Debug.LogWarning("Deck is empty, cannot draw card.");
            return null; // Handle empty deck scenario
        }
        CardDataSO card = deck; // Get the top card
        deck.RemoveAt(0); // Remove it from the deck
        return card;
    }

    // Helper method to assign random turfs to a player
    private void AssignRandomTurfs(List<Hex> turfList, int count)
    {
        // This is a very basic random assignment. In a real game,
        // ensure turfs are valid (e.g., not occupied by starting tokens) and distinct.
        List<Hex> allHexes = new List<Hex>(hexGridGenerator.hexTiles.Keys); // Get all
available hex coordinates
        for (int i = 0; i < count; i++)
        {
            if (allHexes.Count == 0)
            {
                Debug.LogWarning("Not enough unique hexes to assign turfs.");
                break;
            }
            int randomIndex = Random.Range(0, allHexes.Count);
            turfList.Add(allHexes[randomIndex]); // Add the randomly selected hex as a turf
            allHexes.RemoveAt(randomIndex); // Remove it from the pool to prevent duplicate
assignments
        }
    }

    private void HandleGameEnd()
    {
        Debug.Log("Game State: Game Over!");
        // Calculate scores, determine winner [1]
        // This logic would be more detailed, potentially in a separate ScoreManager.
        CalculateAndDeclareWinner();
    }

    private void CalculateAndDeclareWinner()
    {
        // Placeholder: You'd need a way to track which player owns which hex tile throughout the game.
        // For example, each HexTile could have a 'PlayerID owner' property that is updated when a token
is deployed.

        int player1AssignedTurfsCaptured =
CalculateCapturedAssignedTurfs(player1Turfs, PlayerID.Player1);
        int player2AssignedTurfsCaptured =
CalculateCapturedAssignedTurfs(player2Turfs, PlayerID.Player2);

        int player1TotalTurfs = CalculateTotalTurfs(PlayerID.Player1);
        int player2TotalTurfs = CalculateTotalTurfs(PlayerID.Player2);

        string winnerMessage = "It's a Draw!";

        // First, compare assigned turfs [1]
        if (player1AssignedTurfsCaptured > player2AssignedTurfsCaptured)
        {
            winnerMessage = "Player 1 Wins!";
        }
        else if (player2AssignedTurfsCaptured > player1AssignedTurfsCaptured)
        {
            winnerMessage = "Player 2 Wins!";
        }
        else // Assigned turfs are tied, then compare total turfs [1]
        {
            if (player1TotalTurfs > player2TotalTurfs)
            {
                winnerMessage = "Player 1 Wins (more total turfs)!";
            }
            else if (player2TotalTurfs > player1TotalTurfs)
            {
                winnerMessage = "Player 2 Wins (more total turfs)!";
            }
        }

        Debug.Log(winnerMessage);
        // Trigger end game UI display (Section 6.1)
    }

    // Placeholder methods for calculating turf ownership.
    // These would iterate through all HexTiles and check their 'owner' property.
    private int CalculateCapturedAssignedTurfs(List<Hex> assignedTurfs, PlayerID player)
    {
        int count = 0;
        foreach (Hex turfHex in assignedTurfs)
        {
            HexTile tile = hexGridGenerator.GetHexTile(turfHex);
            // Assuming HexTile has an 'OwnerID' property
            // if (tile!= null && tile.OwnerID == player)
            // {
            //     count++;
            // }
        }
        return count;
    }

    private int CalculateTotalTurfs(PlayerID player)
    {
        int count = 0;
        foreach (HexTile tile in hexGridGenerator.GetAllHexTiles()) // Assuming GetAllHexTiles
method exists
        {
            // if (tile.OwnerID == player)
            // {
            //     count++;
            // }
        }
        return count;
    }
}
```

The Finite State Machine (FSM) serves as the central nervous system of the game. "Turf War" is explicitly a "turn base strategy PVP" game [1], which inherently demands a structured sequence of player actions and phases. Without a clear state management system, game logic can quickly devolve into a complex and unmanageable web of

if statements and boolean flags.[20] The FSM pattern, a well-established solution for managing sequential behaviors and transitions in games, directly addresses this challenge.[19]

Implementing the GameManager as a Singleton provides a single, authoritative point of control for the game's overall state.[24] This centralized control is vital for a turn-based game, especially one incorporating a "reaction window" [1] that requires precise timing and synchronization between players. Furthermore, the use of C#

events, such as

OnGameStateChanged, allows different parts of the game—including the user interface, the board, and player input systems—to react to state changes without being tightly coupled to the GameManager itself.[13] This promotes loose coupling, which enhances modularity and makes the system more flexible. A robust FSM within a Singleton

GameManager is not merely a coding pattern; it is the architectural backbone that ensures the game's rules are enforced consistently, turns progress correctly, and all interconnected systems remain synchronized. This structure is paramount for delivering a fair and predictable competitive PVP experience.

Table 2.2: Core Game Data ScriptableObjects

This table provides a quick reference for the game's data model, illustrating how various game elements are defined and interconnected through ScriptableObjects.

ScriptableObject Name	Purpose	Key Properties	Relationships
BaseGameDataSO	Provides common properties for all game data assets.	Id, DisplayName	Base class for all other GameDataSOs.
CardDataSO	Defines properties for all game cards (Deploy & Event).	CardType, Faction, Artwork, Description, ManaCost	References TokenDataSO (for Deploy Cards) and AbilityDataSO (for Event Cards).
TokenDataSO	Defines properties for all deployable tokens.	TokenLevel, Prefab, Attack, Health, IsSpecial, IsAlpha	Referenced by CardDataSO.
LeaderDataSO	Defines properties for player leaders.	Icon, PassiveDescription	References AbilityDataSO (for passive ability).
AbilityDataSO	Defines generic abilities and effects.	TriggerType, EffectType, Value,	Referenced by CardDataSO and

		Duration, TargetTag	LeaderDataSO.

Table 2.3: Game State Machine Transitions

This table outlines the main game states and the events that trigger transitions between them, providing a high-level overview of the game's operational flow.

Current State	Triggering Event(s)	Next State(s)
GameSetup	All players loaded, board initialized, decks shuffled, hands drawn, leaders/alphas selected, turfs assigned.	PlayerTurn (based on coin flip)
PlayerTurn	Player ends turn, turn timer expires, game time limit reached, critical objective captured.	OpponentTurn, ReactionPhase, GameEnd
OpponentTurn	Opponent completes actions, opponent turn timer expires, game time limit reached, critical objective captured.	PlayerTurn, ReactionPhase, GameEnd
ReactionPhase	Opponent responds/dismisses, reaction timer expires.	PlayerTurn, OpponentTurn (based on previous state)
GameEnd	Game time limit reached, all turfs captured, specific win condition met.	(No further game state transitions; leads to score screen/exit)

3. Constructing the Hexagonal Battlefield

This section details the creation of the "Turf War" game board, from the underlying hexagonal grid mathematics to its 3D visualization and player interaction.

3.1. Hex Grid Mathematics and Coordinate Systems

Hexagonal grids are a popular choice in strategy games due to their unique properties, such as having six equidistant neighbors, which can simplify movement and area-of-effect calculations.[26] Unlike traditional square grids, hex grids require specific coordinate systems for efficient management of positions and distances.[27] While offset coordinates are commonly used, axial or cube coordinates offer more elegant symmetry and simplified algorithms for calculating distances and identifying neighbors.[27] For "Turf War," the primary system utilized will be

Axial Coordinates (q, r), as they are intuitive and can be easily converted to Cube coordinates (q, r, s, where s = -q-r) and subsequently to Unity's world positions.

Converting hex coordinates to Unity's 3D world space (Vector3) is essential for accurately positioning game objects on the board. This conversion involves defining the size and orientation (flat-top or pointy-top) of the hexagons within the game world.[28]

C#

```csharp
// Scripts/HexGrid/Hex.cs
using UnityEngine;
using System.Collections.Generic;

// Represents a single hexagonal cell in axial coordinates.
// allows instances of this struct to be displayed and edited in the Unity Inspector.

public struct Hex
{
    public readonly int q; // The Q-coordinate (horizontal axis in axial system)
    public readonly int r; // The R-coordinate (diagonal axis in axial system)
    public int s => -q - r; // The S-coordinate, derived for cube coordinate compatibility (q + r + s = 0)

    // Constructor to initialize a Hex with its q and r coordinates.
    public Hex(int q, int r)
    {
        this.q = q;
        this.r = r;
    }

    // Defines the outer radius of a hexagon in Unity world units.
    // This value should be adjusted based on the size of your 3D models for hex tiles.
    public static float HEX_SIZE = 1f;

    // Basis vectors for converting axial coordinates to world coordinates.
    // These are derived from hexagonal geometry, specifically for a pointy-top orientation.
    // (Reference: Red Blob Games - https://www.redblobgames.com/grids/hexagons/)
    private static readonly Vector2 Q_BASIS = new Vector2(Mathf.Sqrt(3f) * HEX_SIZE, 0f);
    private static readonly Vector2 R_BASIS = new Vector2(Mathf.Sqrt(3f) / 2f * HEX_SIZE, 1.5f
* HEX_SIZE);

    // Converts the logical hex coordinates to a 3D world position (typically on the XZ plane in Unity).
    // yOffset allows for vertical positioning, useful for layered effects or "stair" visuals.
    public Vector3 ToWorldPosition(float yOffset = 0f)
    {
        float x = Q_BASIS.x * q + R_BASIS.x * r;
        float z = Q_BASIS.y * q + R_BASIS.y * r; // Unity's Z-axis is typically depth, while Y is up.
        return new Vector3(x, yOffset, z);
    }

    // Returns an enumerable collection of all six direct neighbors of this hex.
    // This is crucial for movement, range calculations, and pathfinding.
    public IEnumerable<Hex> GetNeighbors()
    {
        yield return new Hex(q + 1, r);      // East
        yield return new Hex(q, r + 1);      // Northeast
        yield return new Hex(q - 1, r + 1); // Northwest
        yield return new Hex(q - 1, r);      // West
        yield return new Hex(q, r - 1);      // Southwest
        yield return new Hex(q + 1, r - 1); // Southeast
    }

    // Overrides the default Equals method to compare Hex structs by their q and r coordinates.
    // Essential for using Hex as keys in Dictionaries or elements in HashSets.
    public override bool Equals(object obj)
    {
        if (obj is Hex other)
        {
            return q == other.q && r == other.r;
        }
        return false;
    }

    // Overrides the default GetHashCode method for efficient storage and retrieval in hash-based
collections.
    public override int GetHashCode()
    {
        return (q * 31) + r; // A simple prime-number based hash for structs.
    }

    // Overloads the equality (==) and inequality (!=) operators for easier comparison of Hex structs.
    public static bool operator ==(Hex a, Hex b) => a.Equals(b);
    public static bool operator!=(Hex a, Hex b) =>!(a == b);
}
```

3.2. Procedural Generation of the 3D Hex Board

The "Turf War" game board is specified to have 36 hexes.[1] This can be represented as a hexagonal grid with a certain "radius" from a central hex (e.g., a radius of 3 from the center results in 37 hexes, including the center). Each individual hex on the board will be represented by a

HexTile MonoBehaviour script attached to a 3D model. This script will store its logical Hex coordinate, references to its visual components, and potentially its current "block" type as defined in the game design.[1]

C#

```csharp
// Scripts/HexGrid/HexTile.cs
using UnityEngine;

public class HexTile : MonoBehaviour
{
    // exposes the private backing field of the auto-property in the Inspector.
    public Hex HexCoordinates { get; private set; } // The logical coordinates of this tile on the hex grid
    public MeshRenderer TileRenderer { get; private set; } // Reference to the tile's visual MeshRenderer component

    // Initializes the HexTile with its logical coordinates, world position, and parent transform.
    public void Initialize(Hex hex, Vector3 worldPos, Transform parentTransform)
    {
        HexCoordinates = hex; // Assign the logical hex coordinates
        transform.position = worldPos; // Set the tile's position in the Unity world
        transform.SetParent(parentTransform); // Organize the tile under a parent GameObject in the Hierarchy
        name = $"Hex ({hex.q}, {hex.r})"; // Name the GameObject for clarity in the Hierarchy
        TileRenderer = GetComponent<MeshRenderer>(); // Get the MeshRenderer component attached to this GameObject
    }

    // Method to visually highlight or unhighlight the tile, e.g., on mouse hover or selection.
    public void Highlight(bool enable)
    {
        if (TileRenderer!= null)
        {
            // Example: Change the material color to yellow when highlighted, or back to white.
            // In a real game, this might involve enabling/disabling an outline, a glow effect, etc.
            TileRenderer.material.color = enable? Color.yellow : Color.white;
        }
    }

    // Future methods would include updating the tile's block type, visual appearance based on block,
    // or tracking ownership for turf capture.
}
```

The HexGridGenerator script is responsible for creating the entire board at runtime, utilizing the Hex struct and the HexTile prefab.

C#

```
// Scripts/HexGrid/HexGridGenerator.cs
using UnityEngine;
using System.Collections.Generic;

public class HexGridGenerator : MonoBehaviour
{

    private GameObject hexTilePrefab; // Assign your HexTile prefab (a 3D model with HexTile
script) here in the Inspector
    private int gridRadius = 3; // Defines the size of the hexagonal map. A radius of 3 creates a 37-hex
grid (center + 3 rings).
    private float tileYOffsetStep = 0.1f; // Controls the height difference between "stair" rows [1]

    // A dictionary to quickly access HexTile objects by their logical Hex coordinates.
    // This allows for efficient lookup of tiles based on their grid position.
    public Dictionary<Hex, HexTile> hexTiles = new Dictionary<Hex, HexTile>();

    void Start()
    {
        GenerateGrid();
    }

    // [ContextMenu] attribute allows calling this method directly from the Inspector's context menu.
    // Useful for testing grid generation during development.
    [ContextMenu("Generate Grid")]
    public void GenerateGrid()
    {
        // Clear any existing tiles before generating a new grid.
        // This is important for regenerating the board in editor or at runtime.
        // DestroyImmediate is used in editor mode for instant cleanup.
        foreach (Transform child in transform)
        {
            DestroyImmediate(child.gameObject);
        }
        hexTiles.Clear(); // Clear the dictionary as well

        // Loop through axial coordinates to generate hexes in a radial pattern based on gridRadius.
        for (int q = -gridRadius; q <= gridRadius; q++)
        {
            // Calculate the valid range for 'r' coordinate for the current 'q' to form a hexagonal shape.
            int r1 = Mathf.Max(-gridRadius, -q - gridRadius);
            int r2 = Mathf.Min(gridRadius, -q + gridRadius);
            for (int r = r1; r <= r2; r++)
            {
                Hex hex = new Hex(q, r); // Create a new Hex struct for the current coordinates

                // Calculate Y offset for the "stair" effect.[1]
                // A simple approach: Y increases with the Manhattan distance from the center (0,0,0).
                float yOffset = (Mathf.Abs(hex.q) + Mathf.Abs(hex.r) + Mathf.Abs(hex.s)) / 2 *
tileYOffsetStep;
                Vector3 worldPos = hex.ToWorldPosition(yOffset); // Convert hex coordinates to
world position with Y offset

                // Instantiate the visual hex tile prefab at the calculated world position.
                // It's parented to this GameObject for organization in the Hierarchy.
                GameObject tileGO = Instantiate(hexTilePrefab, worldPos,
Quaternion.identity, transform);
                HexTile hexTile = tileGO.GetComponent<HexTile>(); // Get the HexTile script
component

                if (hexTile!= null)
                {
                    hexTile.Initialize(hex, worldPos, transform); // Initialize the HexTile script
                    hexTiles.Add(hex, hexTile); // Add the tile to the dictionary for easy lookup
                }
            }
        }
        Debug.Log($"Generated {hexTiles.Count} hex tiles.");
    }

    // Retrieves a HexTile object given its logical Hex coordinates.
    public HexTile GetHexTile(Hex hex)
    {
        hexTiles.TryGetValue(hex, out HexTile tile); // TryGetValue is safer than direct access if key might not exist
        return tile;
    }

    // Returns all generated HexTiles.
    public IEnumerable<HexTile> GetAllHexTiles()
    {
        return hexTiles.Values;
    }

    // Placeholder method for distributing different "block" types randomly on the board.
    // This would involve assigning different materials, colors, or child GameObjects to each HexTile.
    public void DistributeBlocksRandomly()
    {
        // Example: Iterate through all generated hex tiles and assign a random block type.
        // This would involve a list of block prefabs or materials.
        Debug.Log("Randomly distributing blocks on the board.");
        foreach (HexTile tile in hexTiles.Values)
        {
            // For a simple visual, you could change the tile's material or color.
            // For complex blocks, you might instantiate a BlockPrefab on the tile.
            // tile.SetBlockType(RandomBlockType()); // Assuming a method to set block type
        }
    }
}
```

While "Turf War" specifies a board "randomly generated from specific building blocks" [1], purely random generation can lead to undesirable outcomes such as unwinnable games, instant victories, or scenarios where winning is impossible.[29] Such issues can severely impact game balance and player enjoyment. Procedural generation, when guided by algorithms, offers a solution by allowing for controlled randomness, ensuring that generated boards are varied and unique while remaining playable and fair.[29] The game's unique "stair" visual effect [1] necessitates precise Y-axis positioning based on hex coordinates, a requirement directly supported by the

Hex.ToWorldPosition method and the HexGridGenerator's logic. The implementation of this controlled randomness is a fundamental principle for any game relying on generated content, ensuring that the game remains engaging and strategically sound rather than frustrating or trivial.

3.3. Player Interaction: Selecting Hex Tiles

To enable players to select hex tiles on the game board, Unity's Raycasting system is employed. This involves casting an invisible ray from the camera, through the mouse cursor's position (or a touch input position on mobile devices), into the 3D world. The system then detects if this ray intersects with a Collider component attached to a HexTile GameObject.[30]

C#

```csharp
// Scripts/Input/HexInputHandler.cs
using UnityEngine;

public class HexInputHandler : MonoBehaviour
{
    private Camera mainCamera; // Assign your main camera here in the Inspector
    private LayerMask hexTileLayer; // Create a new Layer in Unity (e.g., "HexTile") and assign it to your hexTilePrefab.
                                    // Then, select this layer mask in the Inspector for this script.

    // C# event that other systems can subscribe to, to be notified when a hex tile is clicked.
    // This promotes loose coupling between the input system and game logic.
    public static event System.Action<HexTile> OnHexTileClicked;

    void Update()
    {
        // Check for left mouse button click.
        if (Input.GetMouseButtonDown(0))
        {
            HandleClick(Input.mousePosition);
        }
        // For mobile touch input, you would check Input.touchCount > 0 and Input.GetTouch(0).phase ==
TouchPhase.Began.
        // The HandleClick method can then be reused for touch positions.
    }

    // Handles the actual raycasting logic based on a screen position (mouse or touch).
    private void HandleClick(Vector3 screenClickPosition)
    {
        // Creates a ray originating from the camera and passing through the given screen point.
        Ray ray = mainCamera.ScreenPointToRay(screenClickPosition);
        RaycastHit hit; // A struct to store information about what the ray hit.

        // Performs the raycast.
        // 'ray.origin': starting point of the ray.
        // 'ray.direction': direction of the ray.
        // 'out hit': outputs the RaycastHit information.
        // '100f': max distance the ray will travel.
        // 'hexTileLayer': ensures the ray only detects colliders on the specified layer, improving
performance and accuracy.
        if (Physics.Raycast(ray, out hit, 100f, hexTileLayer))
        {
            // Attempt to get the HexTile component from the GameObject that was hit.
            HexTile clickedTile = hit.collider.GetComponent<HexTile>();
            if (clickedTile!= null)
            {
                Debug.Log($"Clicked on Hex Tile: {clickedTile.HexCoordinates.q},
{clickedTile.HexCoordinates.r}");
                OnHexTileClicked?.Invoke(clickedTile); // Invoke the event, passing the clicked
HexTile.
                clickedTile.Highlight(true); // Example: Visually highlight the clicked tile.
            }
        }
    }
}
```

The game's initial description does not specify a target platform, implying a general "game" experience rather than solely PC or mobile. "Turf War" is a "virtual Tactical Card game" [1], a genre that often lends itself to multi-platform releases, including PC and mobile tablets. While directly utilizing

Input.GetMouseButtonDown is suitable for PC, mobile platforms require checking Input.GetTouch for touch inputs.[30] By abstracting input handling into a dedicated

HexInputHandler script and employing C# events (OnHexTileClicked), the core game logic becomes independent of the specific input method. This means the game logic only needs to know *that* a tile was clicked, not *how* it was clicked. This design aligns with modular programming principles.[13] Designing input systems with such abstraction from the outset significantly simplifies the process of porting the game to different platforms (PC, mobile, console) without necessitating extensive rewrites of core gameplay logic. This consideration is key for maximizing audience reach, especially for a card game that could thrive on mobile devices.

4. Bringing Game Elements to Life: Cards, Tokens & Leaders

This section delves into the dynamic aspects of "Turf War," focusing on how game elements like cards and tokens are created, played, and interact within the game world.

4.1. Instantiating Cards and Tokens at Runtime

In Unity, Prefabs serve as essential reusable GameObject assets, acting as blueprints or templates from which new instances can be created dynamically at runtime.[32] For "Turf War," this means creating Prefabs for the visual representations of cards and tokens. Instead of manually placing every card or token in the scene, they will be instantiated from their respective Prefabs using

GameObject.Instantiate() whenever needed, such as when a player draws a card or deploys a token onto the board.[32]

C#

```
// Scripts/Gameplay/CardManager.cs (Simplified example for core functionality)
```

```
using UnityEngine;
using System.Collections.Generic;
using TMPro; // Required for TextMeshProUGUI if used in UI

public class CardManager : MonoBehaviour
{
    private GameObject cardPrefab; // Assign your generic Card visual prefab (e.g., a UI Panel or 3D
Plane with CardDisplay script)
    private Transform handTransform; // The UI parent transform where cards in hand will be placed
(e.g., a Horizontal Layout Group)

    // A list to keep track of the visual CardDisplay instances currently in the player's hand.
    public List<CardDisplay> playerHand = new List<CardDisplay>();

    // Example method to simulate drawing a card from a deck.
    // It takes a CardDataSO (the data for the card) and creates its visual representation.
    public void DrawCard(CardDataSO cardData)
    {
        if (cardData == null)
        {
            Debug.LogWarning("Attempted to draw a null card data.");
            return;
        }

        // Instantiate the visual card prefab as a child of the handTransform.
        GameObject newCardGO = Instantiate(cardPrefab, handTransform);
        // Get the CardDisplay component from the newly created GameObject.
        CardDisplay cardDisplay = newCardGO.GetComponent<CardDisplay>();

        if (cardDisplay!= null)
        {
            // Initialize the CardDisplay with the ScriptableObject data, which will update its visuals.
            cardDisplay.Initialize(cardData);
            playerHand.Add(cardDisplay); // Add the visual card to the player's hand list.
            // Additional logic here would arrange cards visually within the hand (e.g., using a Horizontal
Layout Group or custom spacing).
            Debug.Log($"Player drew: {cardData.DisplayName}");
        }
        else
        {
            Debug.LogError("CardPrefab is missing a CardDisplay component.");
        }
    }

    // Example method to deploy a token onto the game board.
    // It takes a TokenDataSO (the data for the token) and the target board position.
    public void DeployToken(TokenDataSO tokenData, Vector3 boardPosition)
    {
        if (tokenData == null |

| tokenData.Prefab == null)
        {
            Debug.LogWarning("Attempted to deploy a null token data or token prefab.");
            return;
        }

        // Instantiate the token's visual prefab at the specified board position.
        GameObject newTokenGO = Instantiate(tokenData.Prefab, boardPosition,
Quaternion.identity);
        // If tokens have their own controller script (e.g., TokenController), initialize it here.
        TokenController tokenController =
newTokenGO.GetComponent<TokenController>();
        if (tokenController!= null)
        {
            // Assuming the token is deployed to a specific Hex, you'd pass its Hex coordinates here.
            // For this example, we'll use a placeholder Hex.
            tokenController.Initialize(tokenData, new Hex( (int)boardPosition.x,
(int)boardPosition.z) );
        }
        Debug.Log($"Deployed token: {tokenData.DisplayName} at {boardPosition}");
    }
}
```

The game design for "Turf War" outlines distinct visual patterns for "Deploy cards" (pastel colors) and "Event cards" (vibrant colors), along with different behaviors.[1] Instead of creating a separate prefab for every single card, a more efficient approach involves using a generic

CardPrefab combined with a CardDisplay script. This script dynamically updates the card's visuals—including artwork, colors, and text—based on the CardDataSO it receives. This method aligns strongly with modular design principles and a data-driven approach, which were established in Section 2. This strategy offers significant optimization for asset management and memory usage, particularly crucial for a "card collection" game [1] that is intended to feature "new cards every day".[34] By decoupling the card's

data (CardDataSO) from its *visual presentation* (CardDisplay script on a CardPrefab), the system achieves immense flexibility. New cards can be introduced simply by creating new ScriptableObject assets, eliminating the need for new prefabs or code modifications. This flexibility is fundamental for supporting a "live service game" model [8], where frequent content updates are a core component of player engagement and retention.

4.2. Implementing Card Play Logic (Deploy & Event Cards)

The CardDisplay script, attached to the cardPrefab, is responsible for rendering the CardDataSO information and managing player interactions, such as clicking to play a card.

C#

```csharp
// Scripts/Gameplay/CardDisplay.cs
using UnityEngine;
using UnityEngine.UI; // For UI elements like Image
using TMPro; // For TextMeshProUGUI, recommended for better text rendering in UI

public class CardDisplay : MonoBehaviour
{
    // References to UI elements on the card prefab, assigned in the Inspector.
    private Image artworkImage;
    private TextMeshProUGUI nameText;
    private TextMeshProUGUI costText;
    private TextMeshProUGUI descriptionText;
    private Image cardFrameImage; // Used to change color based on card type

    // The CardDataSO instance that this visual card represents.
    public CardDataSO CardData { get; private set; }

    // C# event: Notifies other systems when this card is clicked to be played.
    public static event System.Action<CardDisplay> OnCardPlayed;

    // Initializes the card's visual display with data from a CardDataSO.
    public void Initialize(CardDataSO data)
    {
        CardData = data; // Assign the data
        UpdateUI(); // Update the visual elements based on this data
    }

    // Updates the UI elements of the card based on its assigned CardDataSO.
    private void UI()
    {
        if (CardData == null) return; // Ensure data is assigned

        artworkImage.sprite = CardData.Artwork; // Set the card's artwork
        nameText.text = CardData.DisplayName; // Set the card's name
        costText.text = CardData.ManaCost.ToString(); // Set the card's mana cost
        descriptionText.text = CardData.Description; // Set the card's description

        // Apply visual style based on card type.[1]
        // Deploy cards use pastel colors, Event cards use vibrant colors.
        if (CardData.Type == CardDataSO.CardType.Deploy)
        {
            cardFrameImage.color = new Color(0.8f, 0.9f, 1.0f); // Example: Pastel blue
        }
        else // Event card
        {
            cardFrameImage.color = new Color(1.0f, 0.6f, 0.2f); // Example: Vibrant orange
        }
    }

    // This method is called when the player clicks on the card.
    // It would be hooked up to a Unity UI Button component's OnClick() event.
    public void OnClickPlayCard()
    {
        // Basic check: Only allow playing if it's currently the player's turn.
        if (GameManager.Instance.CurrentState == GameManager.GameState.PlayerTurn)
        {
            OnCardPlayed?.Invoke(this); // Invoke the event, notifying listeners that this card was played.
        }
        else
        {
            Debug.Log("It's not your turn to play cards!");
        }
    }
}
```

The PlayerController (or a dedicated TurnManager) will subscribe to the OnCardPlayed event and manage the execution of the card's effects.

C#

```
// Scripts/Gameplay/PlayerController.cs (Simplified example for demonstration)
using UnityEngine;

public class PlayerController : MonoBehaviour
{
    private CardManager cardManager; // Reference to the CardManager in the scene
    private HexGridGenerator hexGrid; // Reference to the HexGridGenerator in the scene

    private CardDisplay selectedCardToPlay; // Stores the card the player has selected to play

    // Called when the script instance is enabled. Used to subscribe to events.
    void OnEnable()
    {
        CardDisplay.OnCardPlayed += HandleCardPlayed; // Subscribe to card play events
        HexInputHandler.OnHexTileClicked += HandleHexTileClicked; // Subscribe to hex tile click events
        GameManager.OnGameStateChanged += OnGameStateChanged; // Subscribe to game state changes
    }

    // Called when the script instance is disabled. Used to unsubscribe from events to prevent memory leaks.
    void OnDisable()
    {
        CardDisplay.OnCardPlayed -= HandleCardPlayed;
        HexInputHandler.OnHexTileClicked -= HandleHexTileClicked;
        GameManager.OnGameStateChanged -= OnGameStateChanged;
    }

    // Reacts to changes in the game state.
    private void OnGameStateChanged(GameManager.GameState newState)
    {
        // Logic to enable or disable player input and highlight UI elements based on the current turn.
        if (newState == GameManager.GameState.PlayerTurn)
        {
            Debug.Log("It's your turn! Make your move.");
            // Example: Enable player input scripts, highlight playable cards in hand.
        }
        else
        {
            // Example: Disable player input scripts, dim cards in hand.
        }
    }

    // Handles a card being "played" (clicked by the player).
    private void HandleCardPlayed(CardDisplay card)
    {
        // Ensure a card can only be played during the player's turn.
        if (GameManager.Instance.CurrentState!= GameManager.GameState.PlayerTurn)
        {
            Debug.LogWarning("Cannot play card outside of player turn.");
            return;
        }

        selectedCardToPlay = card; // Store the selected card
        Debug.Log($"Player selected card to play: {card.CardData.DisplayName}. Now select a target hex (if applicable).");

        // If it's a Deploy card, the player needs to select a hex on the board.
        if (card.CardData.Type == CardDataSO.CardType.Deploy)
        {
            // Future logic: Highlight valid deployment hexes on the board for the player.
        }
        else if (card.CardData.Type == CardDataSO.CardType.Event)
        {
            // Event cards might not require a specific target hex, or their targeting is determined by their ability.
            ExecuteCardEffect(card.CardData, null); // Execute the effect immediately, passing null for no target.
            selectedCardToPlay = null; // Clear the selected card after playing.
        }
    }

    // Handles a hex tile being clicked by the player.
    private void HandleHexTileClicked(HexTile tile)
    {
        // Check if a Deploy card is currently selected and waiting for a target hex.
        if (selectedCardToPlay!= null && selectedCardToPlay.CardData.Type == CardDataSO.CardType.Deploy)
        {
            // Future logic: Validate if the clicked tile is a valid deployment spot (e.g., empty, not a "denied" area).
            // For this prototype, assume any clicked tile is valid for demonstration.
            ExecuteCardEffect(selectedCardToPlay.CardData, tile); // Execute the deploy card's effect on the chosen tile.
            cardManager.playerHand.Remove(selectedCardToPlay); // Remove the card from the player's hand list.
            Destroy(selectedCardToPlay.gameObject); // Destroy the visual card GameObject from the scene.
            selectedCardToPlay = null; // Clear the selected card.
        }
        else
        {
            Debug.Log("No deploy card selected or invalid click for current action.");
        }
    }

    // Executes the effects defined by a CardDataSO.
    // This method would contain the core game logic for applying card effects.
    private void ExecuteCardEffect(CardDataSO cardData, HexTile targetTile)
    {
        Debug.Log($"Executing effect for {cardData.DisplayName}");

        if (cardData.Type == CardDataSO.CardType.Deploy)
        {
            if (cardData.TokenToDeploy!= null && targetTile!= null)
            {
                // Deploy the token's prefab at the target tile's position.
                cardManager.DeployToken(cardData.TokenToDeploy,
targetTile.transform.position);
                // Future logic: Assign the deployed token to the HexTile, handle existing token removal.[1]
            }
        }
        else if (cardData.Type == CardDataSO.CardType.Event)
        {
            // Iterate through all abilities defined on the Event card.
            foreach (var ability in cardData.Abilities)
            {
                // Trigger specific ability logic based on the ability.Effect type.
                Debug.Log($"  - Triggering ability: {ability.DisplayName} ({ability.Effect})");
                // This would typically involve a more complex 'AbilityExecutor' system
                // that interprets the AbilityDataSO and applies its effects to game objects/state.
            }
        }
        // After any card is played, check hand size and discard if > 5.[1]
        // For Deploy cards, enforce "only one deploy card can be played per turn".[1]
        // This logic would reside in the TurnManager or PlayerController.
    }
}
```

"Turf War" involves a complex interplay between cards, tokens, leaders, and the game board, with cards having "game changing maneuvers" and "effects that can alter the game".[1] Directly calling methods between all these interconnected systems would create a tightly coupled and difficult-to-manage codebase. An event-driven architecture, utilizing C# events, offers a superior solution by allowing systems to communicate without needing intimate knowledge of each other's internal implementation details.[13] For example, the

CardDisplay script simply announces that a card has been played via an event; the PlayerController (or a TurnManager) then listens for this event and reacts accordingly. This approach is particularly beneficial for managing the "reaction window" [1], where multiple game systems might need to respond to an opponent's action simultaneously. For a game with many interconnected systems and complex rules like "Turf War," an event-driven architecture is paramount. It promotes modularity, simplifies debugging by isolating issues, and facilitates the easier expansion of new card effects or game mechanics without requiring significant modifications to existing code.

4.3. Managing Token Interactions and Abilities

The game design specifies that bigger tokens can be placed over smaller ones or empty blocks, effectively removing the existing token.[1] This core game logic for token placement and removal would typically reside within the

PlayerController or a dedicated BoardController when a Deploy card is played. Each instantiated token GameObject will have a TokenController script attached to it, responsible for managing its individual state, visual representation, and responding to various abilities.

C#

```
// Scripts/Gameplay/TokenController.cs
using UnityEngine;

public class TokenController : MonoBehaviour
{
    // exposes the private backing field of the auto-property in the Inspector.
    public TokenDataSO TokenData { get; private set; } // The data for this specific token instance
    public Hex HexPosition { get; private set; } // The current logical hex coordinates of this token
```

```
// Initializes the token with its data and position.
public void Initialize(TokenDataSO data, Hex hexCoords)
{
    TokenData = data; // Assign the ScriptableObject data
    HexPosition = hexCoords; // Set the token's logical position on the grid

    // Apply visual properties from TokenDataSO.
    // Example: Scale the token based on its level, making higher-level tokens visually larger.
    transform.localScale = Vector3.one * (1f + (int)TokenData.Level * 0.2f);
    // You might also set its material, color, or other visual aspects here.
}

// Applies a specified effect to this token. This method would be called by an AbilityExecutor.
public void ApplyEffect(AbilityDataSO.AbilityEffectType effectType, int value, float duration)
{
    Debug.Log($"Token {TokenData.DisplayName} at {HexPosition} applying effect: {effectType}");
    switch (effectType)
    {
        case AbilityDataSO.AbilityEffectType.Remove:
            Destroy(gameObject); // Remove the token from the game
            Debug.Log($"Token {TokenData.DisplayName} was removed.");
            // Additional logic: Notify BoardController to update tile state.
            break;
        case AbilityDataSO.AbilityEffectType.Knockback:
            // Implement movement logic based on 'value' (number of spots to move).
            Debug.Log($"Token {TokenData.DisplayName} was knocked back by {value} spots.");
            break;
        case AbilityDataSO.AbilityEffectType.Immunity:
            // Apply an immunity status for a 'duration'. This would involve a status effect system.
            Debug.Log($"Token {TokenData.DisplayName} gained immunity for {duration} seconds.");
            break;
        case AbilityDataSO.AbilityEffectType.UpgradeToken:
            // Logic to upgrade this token to a higher level.
            // This would involve changing its TokenDataSO or modifying its stats directly.
            Debug.Log($"Token {TokenData.DisplayName} was upgraded.");
            break;
        //... handle other AbilityEffectType cases as defined in AbilityDataSO
        default:
                Debug.LogWarning($"Effect type {effectType} not yet implemented for tokens.");
                break;
        }
    }
}
```

The "Turf War" game design includes specific rules for tokens: four levels exist, with larger tokens capable of removing smaller ones upon placement. Additionally, Level 3 tokens are designated for deploying special units, and the Level 4 token is a unique "Alpha" unit.[1] The

TokenDataSO already captures these properties, but the TokenController is where the actual *behavior* associated with these properties is implemented. The mechanic of "removing that token" when a larger one is placed [1] represents a core game rule that must be robustly managed within the

BoardController or PlayerController following a Deploy action. Furthermore, the concept that "Level 3 tokens are special, used to deploy special units with effects that can alter the game" [1] suggests that these tokens may possess their own unique

AbilityDataSOs or trigger specific game logic upon deployment or interaction. The token system in "Turf War" is more than a simple unit placement mechanism; it is a dynamic interaction layer that contributes significantly to the game's strategic depth. Properly implementing token levels and their associated effects is crucial for achieving the intended strategic complexity. This also highlights the eventual need for a robust AbilityExecutor system capable of interpreting and applying effects defined in AbilityDataSO to TokenControllers or HexTiles.

4.4. Developing Leader Passive and Active Skills

At the beginning of a game, players select their leader and alpha token.[1] This selection process would typically involve the instantiation of a

LeaderController GameObject in the scene, which is then assigned the chosen LeaderDataSO.

C#

```
// Scripts/Gameplay/LeaderController.cs
using UnityEngine;

public class LeaderController : MonoBehaviour
{
    // exposes the private backing field of the auto-property in the Inspector.
    public LeaderDataSO LeaderData { get; private set; } // The data for this leader

    // Initializes the LeaderController with its associated LeaderDataSO.
    public void Initialize(LeaderDataSO data)
    {
        LeaderData = data; // Assign the leader data
        Debug.Log($"Leader selected: {LeaderData.DisplayName}. Passive Ability:
{LeaderData.PassiveDescription}");
        // Immediately apply the passive ability if it's a constant effect,
        // or subscribe to relevant game events (e.g., OnTurnStart) if it's a triggered passive.
        ApplyPassiveAbility();
    }

    // Applies the leader's passive ability.
    private void ApplyPassiveAbility()
    {
        if (LeaderData.PassiveAbility!= null)
        {
            Debug.Log($"Applying passive ability: {LeaderData.PassiveAbility.DisplayName}");
            // Logic to apply the passive effect.
            // For example, if the common ability is "deploy 1 rank 1 token for free" [1],
            // this might involve modifying player's resources, adding a special action,
            // or subscribing to a deployment event to grant a free token.
            // This would likely involve calling into a central AbilityExecutor system.
        }
        else
        {
            Debug.LogWarning($"Leader {LeaderData.DisplayName} has no passive ability assigned.");
        }
    }
    // If leaders have active skills that players can manually trigger,
    // this method would contain the logic for using that skill.
    public void UseActiveSkill()
    {
        // Logic to check if the skill is currently available (e.g., cooldown, mana cost).
        // Then, trigger the associated ability defined in LeaderData.PassiveAbility (if it's an active skill).
        Debug.Log($"Leader {LeaderData.DisplayName} used active skill.");
        // This would also likely interact with the AbilityExecutor system.
    }
}
```

The "Leader" is a distinct game element in "Turf War" that grants a "passive ability".[1] This passive ability is chosen before the game begins [1], signifying a core strategic decision that helps define a player's playstyle for the duration of the match. For instance, the common ability to "deploy 1 rank 1 token for free" [1] suggests a potential resource advantage or an early game tempo boost. The game design also notes that "Unique Abilities not decided yet" [1], which indicates a clear future design space. This future content can be seamlessly integrated using the existing

AbilityDataSO system. Leaders introduce a significant layer of strategic depth and contribute to player identity within the game. Their abilities, whether passive or active, can profoundly influence gameplay outcomes. Designing these abilities as distinct AbilityDataSO assets allows for straightforward iteration, balancing adjustments, and future expansion, thereby supporting the game's evolution within a live service model.

5. Orchestrating the Turn-Based Gameplay Loop

This section integrates all individual game elements and systems to create the complete turn-based gameplay experience, encompassing game setup, player actions, the unique reaction window, and game conclusion.

5.1. Game Initialization and Setup Phase

The GameManager's HandleGameSetup() method, as introduced in Section 2.3, is responsible for orchestrating the initial sequence of events to prepare the game for play. This sequence includes several critical steps:

1. **Random Map Generation:** The HexGridGenerator.GenerateGrid() method is invoked to create the hexagonal board, followed by DistributeBlocksRandomly() to populate the hexes with various "blocks" as specified in the game design.[1]
2. **Deck Shuffling:** Player decks are initialized from their collection of CardDataSO assets and then shuffled to ensure randomness.[1]
3. **Draw Starting Hand:** Each player draws an initial hand of 4 cards [1], a process managed by the
   CardManager.
4. **Coin Flip for First Player:** A random determination is made to decide which player takes the first turn.[1]
5. **Leader and Alpha Selection:** Players select their leader and alpha token.[1] For a prototype, this might be a pre-selected default or a simple UI choice.
6. **Secret Turf Assignment:** Three random turfs are secretly assigned to each player.[1] These turfs can be specific
   HexTiles marked with a player ID, with their ownership hidden from the opponent.

C#

```csharp
//... inside GameManager.cs (continued from Section 2.3)

  // Handles the initial setup sequence of the game.
  private IEnumerator HandleGameSetup()
  {
      Debug.Log("Game State: Setting Up Game...");

      // 1. Generate Hex Grid and Distribute Blocks [1]
      // Ensure hexGridGenerator is assigned in the Inspector.
      hexGridGenerator.GenerateGrid();
      hexGridGenerator.DistributeBlocksRandomly();
      yield return new WaitForSeconds(1f); // Simulate some loading time or animation for visual effect

      // 2. Initialize Player Decks (simplified for prototype)
```

```
        // In a full game, this would involve loading specific cards from a player's collection.
        Player1Deck = new List<CardDataSO>(allAvailableCards); // Example: populate with all
cards available in the game
        Player2Deck = new List<CardDataSO>(allAvailableCards);
        ShuffleDeck(Player1Deck); // Custom helper method to shuffle the deck
        ShuffleDeck(Player2Deck);
        Debug.Log("Decks shuffled.");
        yield return new WaitForSeconds(0.5f);

        // 3. Draw Starting Hand [1]
        for (int i = 0; i < 4; i++)
        {
            // The CardManager's DrawCard method handles the visual instantiation and adding to hand.
            // DrawCardFromDeck is a helper method to get a card from the conceptual deck list.
            cardManager.DrawCard(DrawCardFromDeck(Player1Deck));
            cardManager.DrawCard(DrawCardFromDeck(Player2Deck)); // Draw for Player 2 as
well
        }
        Debug.Log("Starting hands drawn.");
        yield return new WaitForSeconds(0.5f);

        // 4. Random Coin Flip for First Player [1]
        isPlayer1Turn = (Random.Range(0, 2) == 0); // Randomly decide if Player 1 or Player 2 goes
first
        Debug.Log($"Coin flip: Player {(isPlayer1Turn? "1" : "2")} goes first.");
        yield return new WaitForSeconds(0.5f);

        // 5. Leader and Alpha Selection (Simplified: assume pre-selected or basic UI interaction) [1]
        // For a prototype, you might assign default leaders/alphas. In a full game, this would be a player
choice UI.
        // Ensure player1Leader and player1TokenAlpha GameObjects exist in the scene and are
referenced in GameManager.
        player1Leader.Initialize(defaultLeaderSO);
        // Place alpha token on a starting hex (e.g., center hex or a specific spawn point).
        // Hex(0,0) is an example; a proper spawn point system would be needed.
        player1TokenAlpha.Initialize(defaultAlphaTokenSO, new Hex(0, 0));
        //... similar initialization for Player 2's leader and alpha token.
        Debug.Log("Leaders and Alphas selected.");
        yield return new WaitForSeconds(0.5f);
        // 6. Secret Turf Assignment (Simplified: assign random hexes as turfs) [1]
        // These turfs are assigned secretly and their ownership is not immediately revealed to the
opponent.
        AssignRandomTurfs(player1Turfs, 3);
        AssignRandomTurfs(player2Turfs, 3);
        Debug.Log("Turfs assigned secretly.");
        yield return new WaitForSeconds(0.5f);

        Debug.Log("Game Setup Complete. Starting First Turn.");
        // Transition the game state to the first player's turn.
        UpdateGameState(isPlayer1Turn? GameState.PlayerTurn :
GameState.OpponentTurn);
    }

    // Helper method to shuffle a list using the Fisher-Yates shuffle algorithm.
    // This ensures a truly random order for the deck.
    private void ShuffleDeck(List<CardDataSO> deck)
    {
        for (int i = deck.Count - 1; i > 0; i--)
        {
            int rnd = Random.Range(0, i + 1); // Get a random index from 0 to i
            CardDataSO temp = deck[i]; // Swap current element with the random element
            deck[i] = deck[rnd];
            deck[rnd] = temp;
        }
    }

    // Helper method to draw a card from the top of the given deck list.
    private CardDataSO DrawCardFromDeck(List<CardDataSO> deck)
    {
        if (deck.Count == 0)
        {
            Debug.LogWarning("Deck is empty, cannot draw card.");
            return null; // Return null if the deck is empty to prevent errors.
        }
        CardDataSO card = deck; // Get the card at the top (first element)
        deck.RemoveAt(0); // Remove the drawn card from the deck
        return card;
    }
    // Helper method to assign a specified number of random turfs from all available hexes.
    private void AssignRandomTurfs(List<Hex> turfList, int count)
    {
        // This is a basic random assignment. In a production game,
        // you would need more sophisticated logic to ensure turfs are strategically placed,
        // distinct, and not on top of starting tokens or other critical areas.
        List<Hex> allHexes = new List<Hex>(hexGridGenerator.hexTiles.Keys); // Get all
available hex coordinates from the generator
        for (int i = 0; i < count; i++)
        {
            if (allHexes.Count == 0)
            {
                Debug.LogWarning("Not enough unique hexes to assign turfs.");
                break; // Stop if no more unique hexes are available
            }
            int randomIndex = Random.Range(0, allHexes.Count); // Pick a random hex
            turfList.Add(allHexes[randomIndex]); // Add it to the player's turf list
            allHexes.RemoveAt(randomIndex); // Remove it from the pool to prevent duplicate
assignments
        }
    }
```

The game design explicitly states that "Turf War" will "reveal to each player 3 blocks secretly" and "Game assigns 3 random turfs to each player secretly".[1] In a multiplayer PVP game, maintaining "secret" information, such as an opponent's hand or hidden objectives, while simultaneously ensuring game state synchronization and preventing cheating, presents a significant technical challenge. The standard solution for this is an authoritative server model.[14] In this model, the server holds the true, complete game state, and clients only receive the information they are permitted to see. For random elements like deck shuffling or the assignment of secret objectives, the random seed can be generated by the server and then shared with clients. This allows clients to deterministically reproduce the random outcomes, but the server remains the ultimate arbiter of truth, validating all actions and states.[35] The "secret" aspect of turfs and blocks immediately highlights a crucial future multiplayer networking requirement. While a prototype can simulate this locally, a full PVP game demands a robust client-server architecture with an authoritative server to prevent cheating and ensure a fair and consistent player experience.

5.2. Player Turn Flow and Action Management

During a player's turn in "Turf War," they are granted specific actions: drawing one card, playing exactly one deploy card, utilizing available leader skills, and playing as many event cards as desired.[1] The

GameManager transitions the game to GameState.PlayerTurn, at which point a PlayerController (or a dedicated TurnManager) enables player input and activates relevant UI elements for the active player. To enforce the game's rules, variables are necessary to track actions taken per turn, such as hasPlayedDeployCardThisTurn or deployCardsPlayedCount.

C#

```csharp
// Scripts/Gameplay/TurnManager.cs (Manages the flow of turns and associated timers)
using UnityEngine;
using System.Collections; // Required for Coroutines

public class TurnManager : MonoBehaviour
{
    // Flags and timers for turn management.
    private bool hasPlayedDeployCardThisTurn = false; // Tracks if the player has used their one deploy card this turn [1]
    private float turnTimeLimit = 60f; // Each turn is 1 to 2 minutes long [1]
    private float currentTurnTimer; // Current time remaining in the turn
    private Coroutine turnTimerCoroutine; // Reference to the running turn timer coroutine

    // References to other managers (assigned in Inspector or found via GameManager)
    private CardManager cardManager;
    private GameManager gameManager; // Direct reference for clarity in prototype

    void Start()
    {
        // Subscribe to game state changes to react when a turn begins or ends.
        GameManager.OnGameStateChanged += OnGameStateChanged;
    }

    void OnDestroy()
    {
        // Unsubscribe from events to prevent memory leaks when the GameObject is destroyed.
        GameManager.OnGameStateChanged -= OnGameStateChanged;
    }

    // Reacts to changes in the overall game state.
    private void OnGameStateChanged(GameManager.GameState newState)
    {
        if (newState == GameManager.GameState.PlayerTurn)
        {
            StartPlayerTurn(); // Begin the player's turn sequence
        }
        else if (newState == GameManager.GameState.OpponentTurn)
        {
            StartOpponentTurn(); // Begin the opponent's turn sequence
        }
        // If the game transitions out of an active turn state (e.g., to ReactionPhase or GameEnd),
        // stop any running turn timers.
        if (newState!= GameManager.GameState.PlayerTurn && newState!=
GameManager.GameState.OpponentTurn && turnTimerCoroutine!= null)
        {
            StopCoroutine(turnTimerCoroutine);
        }
    }

    // Initiates the sequence of events for the active player's turn.
    private void StartPlayerTurn()
    {
        Debug.Log("Player's turn begins!");
        hasPlayedDeployCardThisTurn = false; // Reset deploy card usage for the new turn
        currentTurnTimer = turnTimeLimit; // Reset the turn timer
        turnTimerCoroutine = StartCoroutine(CountdownTurnTimer()); // Start the countdown

        // Player draws 1 card at the start of their turn.[1]
        // This assumes GameManager has a public method to draw a card from the current player's deck.
        // cardManager.DrawCard(gameManager.DrawCardFromDeck(gameManager.Player1Deck)); // Example for Player 1
    }

    // Coroutine to handle the countdown timer for the current turn.
    private IEnumerator CountdownTurnTimer()
    {
        while (currentTurnTimer > 0)
        {
            currentTurnTimer -= Time.deltaTime; // Decrement timer by time passed since last frame
            // Update UI timer display (this would be handled by a UI script listening to this value)
            // (See Section 6.3 for UI timer implementation)
            yield return null; // Wait for the next frame
        }
        Debug.Log("Turn time expired! Ending turn.");
        EndTurn(); // Automatically end the turn if time runs out
    }

    // Method called when the player attempts to play a Deploy card.
    public void PlayDeployCard(CardDisplay card)
    {
        if (!hasPlayedDeployCardThisTurn) // Check if a deploy card has already been played this turn
        {
            // Logic to process the deploy card.
            // This would involve calling PlayerController.ExecuteCardEffect() for the deploy action.
            hasPlayedDeployCardThisTurn = true; // Mark that a deploy card has been played
            Debug.Log($"Played deploy card: {card.CardData.DisplayName}");
        }
        else
        {
            Debug.LogWarning("Already played a deploy card this turn. Only one allowed per turn.");
        }
    }

    // Method called when the player attempts to play an Event card.
    public void PlayEventCard(CardDisplay card)
    {
        // Logic to process the event card. Event cards can be played multiple times per turn.[1]
        // This would involve calling PlayerController.ExecuteCardEffect() for the event action.
        Debug.Log($"Played event card: {card.CardData.DisplayName}");
    }

    // Method called when the player attempts to use their Leader's skill.
    public void UseLeaderSkill()
    {
        // Logic to use the leader skill. This might involve cooldowns or resource costs.
        // This would involve calling LeaderController.UseActiveSkill() or similar.
        Debug.Log("Leader skill used.");
    }

    // Method to manually end the current player's turn.
    public void EndTurn()
    {
        if (turnTimerCoroutine!= null)
        {
            StopCoroutine(turnTimerCoroutine); // Stop the turn timer if it's running
        }
        Debug.Log("Player ended turn.");
        // Check hand size and discard cards if hand limit (5) is exceeded.[1]
        // Transition the game state to the opponent's turn.
        gameManager.UpdateGameState(GameManager.GameState.OpponentTurn);
    }

    // Initiates the sequence of events for the opponent's turn (AI or remote player).
    private void StartOpponentTurn()
    {
        Debug.Log("Opponent's turn begins!");
        // For a prototype, this can be a simple delay to simulate opponent actions.
        // In a full game, this would involve AI logic or waiting for network input from another player.
        StartCoroutine(SimulateOpponentTurn());
    }

    // Coroutine to simulate the opponent's turn.
    private IEnumerator SimulateOpponentTurn()
    {
        yield return new WaitForSeconds(3f); // Simulate opponent thinking/acting time
        Debug.Log("Opponent's turn ends.");
        // After opponent's actions, transition back to the player's turn.
        gameManager.UpdateGameState(GameManager.GameState.PlayerTurn);
    }
}
```

The "Turf War" rules explicitly define distinct actions a player can take during their turn: "Play 1 deploy card," "Use leader available skills," and "Play as many event cards as u want".[1] These are specific constraints on player actions that must be rigorously enforced by the game logic. The Finite State Machine (FSM) implemented in the

GameManager ensures that actions are only permitted during the correct game phase. Additionally, specific flags, such as hasPlayedDeployCardThisTurn, are necessary to track and enforce per-turn constraints. The inclusion of a turn timer, ranging from 1 to 2 minutes [1], is a critical component for pacing the game and preventing players from stalling. This timer necessitates the use of Coroutines for non-blocking updates, ensuring the game remains responsive. Implementing these action constraints and turn timers is vital for creating fair and engaging turn-based gameplay. It clearly defines the strategic choices available to players and ensures that the game progresses smoothly, preventing frustrating scenarios where turns might drag on indefinitely.

5.3. Implementing the Reactive "Reaction Window"

A unique mechanic in "Turf War" is the "reaction window": "Once each interaction will pause the turn for 10 seconds for the other player to respond, other player can either dismiss or wait the 10 seconds to auto dismiss".[1] This system introduces a real-time element into the turn-based gameplay. When the active player performs an "interaction" (e.g., playing an Event card or using an ability), the

GameManager transitions to GameState.ReactionPhase. A dedicated UI panel will then appear for the non-active player, displaying a countdown timer and options to react.

C#

```
// Scripts/Gameplay/ReactionHandler.cs (Manages the reaction phase UI and logic)
using UnityEngine;
using System.Collections; // Required for Coroutines
using UnityEngine.UI; // For UI elements like GameObject, Text, Button
using TMPro; // For TextMeshProUGUI for better text rendering

public class ReactionHandler : MonoBehaviour
{
    // References to UI elements for the reaction window, assigned in the Inspector.
    private GameObject reactionWindowPanel; // The UI Panel GameObject to show/hide
    private TextMeshProUGUI reactionTimerText; // Text element to display the countdown
    private Button dismissButton; // Button for the opponent to manually dismiss the reaction
window

    private float reactionTimeLimit = 10f; // The fixed duration for the reaction window [1]
    private Coroutine reactionTimerCoroutine; // Reference to the running countdown coroutine

    void Start()
    {
        // Subscribe to game state changes to activate/deactivate the reaction window.
        GameManager.OnGameStateChanged += OnGameStateChanged;
        // Add a listener to the dismiss button to call DismissReaction when clicked.
        dismissButton.onClick.AddListener(DismissReaction);
        // Initially hide the reaction window panel.
        reactionWindowPanel.SetActive(false);
    }

    void OnDestroy()
    {
        // Unsubscribe from events to prevent memory leaks.
        GameManager.OnGameStateChanged -= OnGameStateChanged;
        dismissButton.onClick.RemoveListener(DismissReaction);
    }

    // Callback method for GameManager.OnGameStateChanged event.
    private void OnGameStateChanged(GameManager.GameState newState)
    {
        if (newState == GameManager.GameState.ReactionPhase)
        {
            StartReactionPhase(); // Activate the reaction phase
        }
        else
        {
            EndReactionPhase(); // Deactivate the reaction phase (hide UI)
        }
    }

    // Initiates the reaction phase, showing the UI and starting the timer.
    private void StartReactionPhase()
    {
        Debug.Log("Reaction phase started for opponent.");
        reactionWindowPanel.SetActive(true); // Show the UI panel [42, 43]
        reactionTimerCoroutine = StartCoroutine(CountdownReactionTimer()); // Start the
countdown coroutine
        // At this point, the opponent would be enabled to play reactive event cards.
        // This would involve enabling their hand UI for event cards and disabling other inputs.
    }

    // Coroutine to handle the countdown for the reaction window.
    private IEnumerator CountdownReactionTimer()
    {
        float timer = reactionTimeLimit; // Initialize timer with the limit
        while (timer > 0)
        {
            timer -= Time.deltaTime; // Decrement timer by the time passed since last frame
            // Update the UI text to display the remaining time, rounded up to the nearest whole second.
            reactionTimerText.text = Mathf.CeilToInt(timer).ToString();
            yield return null; // Wait for the next frame
        }
        Debug.Log("Reaction time expired. Auto-dismissing.");
        DismissReaction(); // Automatically dismiss the window if the timer runs out
    }

    // Method to dismiss the reaction window, either manually or automatically.
    public void DismissReaction()
    {
        if (reactionTimerCoroutine!= null)
        {
            StopCoroutine(reactionTimerCoroutine); // Stop the countdown coroutine if it's still running
        }
        Debug.Log("Reaction dismissed.");
        // Transition the game state back to the main turn flow.
        // GameManager would need to determine the correct next state (e.g., back to PlayerTurn or OpponentTurn).

GameManager.Instance.UpdateGameState(GameManager.GameState.PlayerTurn); // Example: back to player turn
    }

    // Deactivates the reaction phase, hiding the UI.
    private void EndReactionPhase()
    {
        reactionWindowPanel.SetActive(false); // Hide the UI panel
    }
}
```

The "10 seconds for the other player to respond" [1] represents a unique blend of real-time and turn-based mechanics, demanding precise timing and immediate feedback. Coroutines are an ideal solution for managing such time-based actions, as they operate asynchronously without blocking the main game thread, ensuring responsiveness.[36] User interface elements, including the timer text and panel visibility, must update in real-time to reflect the changing state.[40] Furthermore, well-designed audio cues are crucial for player awareness and conveying urgency within this time-sensitive phase.[46] A ticking clock sound can build tension, a distinct "popup" sound [49] can signal the window's appearance, and a "whoosh" sound [51] can accompany its dismissal. This "reaction window" adds a dynamic layer of interaction to the turn-based core of the game. Its effectiveness hinges on seamless UI updates and clear audio-visual feedback, ensuring the game feels responsive and engaging even during periods of waiting for an opponent's response.

5.4. Defining Game End Conditions and Scoring

The game concludes when the time limit expires (15-20 minutes). At this point, the player who has captured more of their 3 secretly assigned turfs wins. In the event of a tie in assigned turfs, the player with more total turfs captured across the board is declared the winner.[1] A global game timer, distinct from the individual turn timers, is necessary to track the overall match duration.

C#

```
// Scripts/Gameplay/GameTimer.cs (Manages the overall game countdown)
using UnityEngine;
using System.Collections; // Required for Coroutines
using TMPro; // For TextMeshProUGUI for UI text display

public class GameTimer : MonoBehaviour
{
    private float gameTimeLimitMinutes = 15f; // The total duration of the game in minutes [1]
    private TextMeshProUGUI gameTimerText; // The UI Text element to display the remaining
game time

    private float timeRemaining; // Tracks the time left in the game
    private bool timerRunning = false; // Flag to control if the timer is actively counting down
    private Coroutine gameCountdownCoroutine; // Reference to the running game countdown
coroutine

    void Start()
    {
        timeRemaining = gameTimeLimitMinutes * 60f; // Convert minutes to seconds
        // Subscribe to game state changes to start/stop the timer appropriately.
        GameManager.OnGameStateChanged += OnGameStateChanged;
        UpdateTimerDisplay(timeRemaining); // Initial display of the timer
    }

    void OnDestroy()
    {
        // Unsubscribe from events to prevent memory leaks.
        GameManager.OnGameStateChanged -= OnGameStateChanged;
    }
    // Callback method for GameManager.OnGameStateChanged event.
    private void OnGameStateChanged(GameManager.GameState newState)
    {
        // The game timer should only run during active player turns.
        if (newState == GameManager.GameState.PlayerTurn |

| newState == GameManager.GameState.OpponentTurn)
        {
            if (!timerRunning) // Start timer only if it's not already running
            {
                timerRunning = true;
                if (gameTimerText!= null) gameTimerText.gameObject.SetActive(true); // Show
timer UI
                gameCountdownCoroutine = StartCoroutine(CountdownGameTimer()); // Start
the countdown coroutine
            }
        }
        else
        {
            // Stop the timer if the game transitions out of an active turn state.
            timerRunning = false;
            if (gameCountdownCoroutine!= null)
            {
                StopCoroutine(gameCountdownCoroutine);
            }
            if (gameTimerText!= null) gameTimerText.gameObject.SetActive(false); // Hide
timer UI
        }
    }

    // Coroutine to handle the main game countdown.
    private IEnumerator CountdownGameTimer()
    {
        while (timeRemaining > 0 && timerRunning)
        {
            timeRemaining -= Time.deltaTime; // Decrement time by the time passed since last
frame
            UpdateTimerDisplay(timeRemaining); // Update the UI display
            yield return null; // Wait for the next frame
        }
        // If the timer runs out and it was actively running, trigger the game end.
        if (timeRemaining <= 0 && timerRunning)
        {
            timeRemaining = 0; // Ensure timer doesn't go negative on display
            UpdateTimerDisplay(timeRemaining); // Final update to show 00:00
            Debug.Log("Game time limit reached!");

GameManager.Instance.UpdateGameState(GameManager.GameState.GameEnd); //
Transition to GameEnd state
        }
    }

    // Formats the time into a "MM:SS" string and updates the UI text.
    private void UpdateTimerDisplay(float timeToDisplay)
    {
        // Ensure time doesn't go negative for display purposes.
        if (timeToDisplay < 0) timeToDisplay = 0;

        float minutes = Mathf.FloorToInt(timeToDisplay / 60); // Calculate whole minutes
        float seconds = Mathf.FloorToInt(timeToDisplay % 60); // Calculate remaining seconds
        // Format the string to ensure two digits for both minutes and seconds (e.g., 05:30).[40, 41, 44,
45]
        gameTimerText.text = string.Format("{0:00}:{1:00}", minutes, seconds);
    }
}
```

The scoring logic would typically reside within the GameManager or a dedicated ScoreManager script, triggered when the GameManager transitions to GameState.GameEnd.

C#

```
//... inside GameManager.cs (continued from Section 2.3 and 5.1)

    // Handles the logic for ending the game, calculating scores, and declaring a winner.
    private void CalculateAndDeclareWinner()
```

```
  {
      Debug.Log("Game Over! Calculating Scores...");

      // These calculations assume that HexTile objects have an 'OwnerID' property
      // that is updated throughout the game whenever a player captures a turf or deploys a token.
      // The 'player1Turfs' and 'player2Turfs' lists would contain the Hex coordinates of their secretly
assigned turfs.

      int player1AssignedTurfsCaptured =
CalculateCapturedAssignedTurfs(player1Turfs, PlayerID.Player1);
      int player2AssignedTurfsCaptured =
CalculateCapturedAssignedTurfs(player2Turfs, PlayerID.Player2);

      int player1TotalTurfs = CalculateTotalTurfs(PlayerID.Player1);
      int player2TotalTurfs = CalculateTotalTurfs(PlayerID.Player2);

      string winnerMessage = "It's a Draw!"; // Default message

      // First, compare the number of captured assigned turfs (primary win condition).[1]
      if (player1AssignedTurfsCaptured > player2AssignedTurfsCaptured)
      {
          winnerMessage = "Player 1 Wins!";
      }
      else if (player2AssignedTurfsCaptured > player1AssignedTurfsCaptured)
      {
          winnerMessage = "Player 2 Wins!";
      }
      else // If assigned turfs are tied, then compare the total number of turfs owned (tie-breaker).[1]
      {
          if (player1TotalTurfs > player2TotalTurfs)
          {
              winnerMessage = "Player 1 Wins (more total turfs)!";
          }
          else if (player2TotalTurfs > player1TotalTurfs)
          {
              winnerMessage = "Player 2 Wins (more total turfs)!";
          }
      }
```

```
        Debug.Log(winnerMessage);
        // At this point, you would typically activate an end-game UI panel to display the winner
        // and offer options like "Play Again" or "Return to Main Menu" (see Section 6.1).
    }

    // Placeholder method to calculate how many of a player's secretly assigned turfs they have
captured.
    private int CalculateCapturedAssignedTurfs(List<Hex> assignedTurfs, PlayerID player)
    {
        int count = 0;
        foreach (Hex turfHex in assignedTurfs)
        {
            HexTile tile = hexGridGenerator.GetHexTile(turfHex);
            // This is where you would check the actual ownership of the tile.
            // Assuming HexTile has an 'OwnerID' property that gets updated when a player controls it.
            // if (tile!= null && tile.OwnerID == player)
            // {
            //     count++;
            // }
        }
        return count;
    }

    // Placeholder method to calculate the total number of turfs a player controls on the board.
    private int CalculateTotalTurfs(PlayerID player)
    {
        int count = 0;
        // Iterate through all hex tiles on the board.
        foreach (HexTile tile in hexGridGenerator.GetAllHexTiles()) // Assuming
HexGridGenerator has a method to get all tiles.
        {
            // Check if the tile is owned by the specified player.
            // if (tile.OwnerID == player)
            // {
            //     count++;
            // }
        }
        return count;
    }
```

The game's design mandates clear win/loss conditions based on turf capture and a time limit.[1] Implementing these dynamic game end conditions and providing immediate player feedback are crucial for player satisfaction and replayability, reinforcing the game's competitive nature. Players need to understand why they won or lost, and this clarity motivates them for future matches. The integration of a global game timer, distinct from individual turn timers, ensures that matches adhere to the specified duration, contributing to a consistent and predictable game experience.

6. User Interface (UI), Visuals & Audio Integration

This section addresses the crucial aspects of user experience, visual presentation, and auditory feedback, bringing "Turf War" to life for the player.

6.1. Designing and Implementing the Game UI

The user interface (UI) and user experience (UX) are critical for player engagement, especially in a tactical card game with numerous interactive elements and a unique "reaction window".[1] An intuitive UI guides players through complex mechanics, provides clear feedback, and enhances immersion.[52] The UI should feature key elements such as the player's hand, board overlays (for highlights and targeting), turn timers, game timers, and various pop-up windows (e.g., for card effects, reaction phase, end-game results).[1]

Unity's UI system, built around Canvas, RectTransform, and various UI components (Image, Text, Button, Layout Groups), provides the tools for this. TextMeshPro is highly recommended over the legacy Text component for superior text rendering quality.

Basic UI show/hide logic is straightforward:

C#

```
// Example: Scripts/UI/UIPanelController.cs
using UnityEngine;

public class UIPanelController : MonoBehaviour
{
    private GameObject panelToControl; // Assign the UI Panel GameObject here

    // Method to show the panel.
    public void ShowPanel()
    {
        if (panelToControl!= null)
        {
            panelToControl.SetActive(true); // Activates the GameObject, making it visible [42, 43]
            Debug.Log($"{panelToControl.name} is now visible.");
        }
    }

    // Method to hide the panel.
    public void HidePanel()
    {
        if (panelToControl!= null)
        {
            panelToControl.SetActive(false); // Deactivates the GameObject, hiding it [42, 43]
            Debug.Log($"{panelToControl.name} is now hidden.");
        }
    }

    // Optional: Toggle panel visibility.
    public void TogglePanel()
    {
        if (panelToControl!= null)
        {
            panelToControl.SetActive(!panelToControl.activeSelf);
            Debug.Log($"{panelToControl.name} visibility toggled to {panelToControl.activeSelf}.");
        }
    }
}
```

This UIPanelController script can be attached to any GameObject, and its ShowPanel, HidePanel, or TogglePanel methods can be called by other scripts or hooked up to UI Buttons. For instance, the ReactionHandler (Section 5.3) would call ShowPanel() on its reactionWindowPanel when the reaction phase begins.

Intuitive UI is crucial for player engagement, particularly in a complex card game with numerous elements and a unique reaction window. A well-designed UI provides clear visual hierarchy, immediate feedback on player actions, and easily understandable information, all of which are essential for guiding players and maintaining their interest.[52] This approach ensures that players can focus on strategic decision-making rather than struggling with the interface.

6.2. Crafting Game Visuals and Art Pipeline

The visual presentation of "Turf War" plays a significant role in its appeal. The game aims for an isometric map of a town with seamlessly merging tiles, and distinct card art styles.[1]

- **Isometric Map Design:** Isometric projection creates a pseudo-3D illusion on a 2D plane, characterized by fixed 120° angles between axes and no perspective distortion.[54] For "Turf War," this means designing hex tiles and environmental elements that align with an isometric grid.[54] Objects positioned higher on the screen should appear farther away to simulate depth.[54]
- **Modular Environment Assets:** To achieve the seamless merging of tiles and efficient level creation, modular design principles are applied to environment assets.[59] This involves creating small, reusable components (like different types of hex blocks or decorative elements) that can be easily combined to construct varied environments. This approach increases efficiency in level design, allows for asset reuse, and supports scalability for expansive game worlds.[11]
- **Seamless Tileable Textures:** For the hexagonal board, using seamless tileable textures is essential to avoid visible seams between individual hex tiles.[60] This often involves careful texture creation (e.g., in image editing software) to ensure edges blend perfectly when repeated.[60]
- **Card Art Styles:** "Turf War" specifies two distinct card art styles: "Deploy cards" with a pastel color pattern to convey less urgency, and "Event cards" with more vibrant colors to emphasize urgency.[1] Digital painting techniques can be employed to achieve these styles.[64] Pastel colors are typically soft, light versions

of hues, often created by mixing white into base colors, yielding a calm, dreamy vibe.[64] Vibrant colors, conversely, use strong contrasts and pure hues to create emphasis and pop.[65]

Consistent art direction, modular assets, and optimized textures are vital for both visual appeal and game performance. For a game with a unique "stair" board and distinct card aesthetics, maintaining visual consistency across all elements is crucial for a cohesive player experience.[1] Modular assets reduce development time and memory usage by allowing reuse of components.[11] Texture optimization, such as using appropriate resolutions and compression, further enhances performance by reducing load times and memory footprint.[78] The "stair" effect on the board, for instance, requires careful 3D modeling and rendering techniques to ensure smooth transitions between elevated hexes.[80] This meticulous attention to visual detail and optimization ensures that "Turf War" not only looks appealing but also runs smoothly across target platforms.

6.3. Integrating Game Audio

Sound design is a powerful, often underestimated, component of game development, capable of shaping player emotions, providing critical feedback, and enhancing immersion.[82]

- **Sound Design Principles:** Effective game audio involves understanding the game's narrative and genre, creating dynamic and interactive audio that reacts to player actions and environmental changes, and focusing on mixing and mastering for different platforms.[82] Sound effects should have variations to prevent repetition and "ear fatigue".[85]
- **Audio for Game Mechanics:**
    - **Card Plays:** When a card is played, distinct sounds can provide satisfying feedback. This might involve sounds of shuffling, dealing, or a unique "snap" or "whoosh" for playing a card.[82] Different card types (Deploy vs. Event) could have subtle variations in their play sounds to reinforce their visual and mechanical differences.
    - **Token Deploys:** Deploying a token could be accompanied by a "thud" or a magical "spawn" sound, possibly varying by token level.[82]
    - **Abilities:** Each ability effect (e.g., Draw, Remove, Negate, Knock back) should have a unique, recognizable audio cue that clarifies its impact.[82] For example,

    a "negate" could have a sharp, dissonant sound, while a "draw" could be a light, positive chime.

- **UI Sound Effects:** User interface sounds are crucial for guiding players and providing immediate feedback for interactions.[89]
    - **Button Clicks & Menu Navigation:** Crisp, consistent sounds for button presses, menu navigation, and panel transitions enhance the tactile feel of the UI.[92]
    - **Reaction Window Popups & Timers:** The "reaction window" (Section 5.3) benefits greatly from specific UI audio. A distinct "popup" sound [49] when it appears, and a subtle "ticking clock" sound as the timer counts down, can create a sense of urgency and alert the player to the time-sensitive decision.[46] A "dismiss" sound (e.g., a "whoosh" [51] or a soft "click") can confirm the action.
- **Background Music and Ambient Sounds:** Music sets the overall tone and atmosphere, while ambient sounds (e.g., town noises, subtle environmental effects) enhance the sense of place on the isometric map.[47] Dynamic music that changes with game state (e.g., more intense during reaction phase, calmer during planning) can heighten emotional engagement.[47]

The strategic use of audio feedback is pivotal for player engagement. Well-designed audio cues enhance immersion, provide critical feedback for player actions, and create a sense of urgency, particularly for the reaction window and other turn-based actions.[93] For instance, distinct sounds for card plays, token deployments, and ability activations confirm player input and provide immediate understanding of game events. The ticking sound of the reaction timer, combined with a unique UI sound for its appearance, clearly communicates the limited time for response, driving player decision-making. This multi-layered audio design contributes significantly to the game's overall responsiveness and player satisfaction.

7. Advanced Considerations & Future Roadmap

As "Turf War" evolves beyond a prototype, several advanced considerations become critical for its long-term success, particularly concerning its Player vs. Player (PVP) nature and potential as a live service game.

7.1. Multiplayer Networking Architecture

For a competitive PVP tactical card game like "Turf War," a robust multiplayer networking architecture is paramount. The primary concern in such games is preventing cheating and ensuring a fair play experience.

The most effective model for this is the **authoritative server model**.[14] In this architecture, the server is the sole authority on all game logic and state. Clients send their inputs (e.g., "play card X on hex Y") to the server, which then processes these actions, updates the official game state, and sends the results back to all connected clients.[14] The clients act as "dumb clients," primarily responsible for rendering the game state received from the server and collecting player input.[14] This prevents hacked clients from manipulating game variables like health, position, or card effects, as the server always holds the true state.[14] Designing for multiplayer from the ground up, with this client-server separation, is crucial, as attempting to retrofit networking onto a single-player architecture often necessitates extensive rewrites.[15]

For turn-based games, state synchronization is managed differently than in real-time games. Instead of continuous updates, the server typically sends a complete game state or a diff (difference) to clients after each significant action or turn transition.[35] For elements involving randomness, such as deck shuffling or drawing secret objective cards, the server can generate a random seed at the start of the game or turn. This seed is then shared with clients, allowing them to deterministically reproduce the random outcomes, while the server still validates the results.[35] This approach ensures that secret information, like an opponent's hand or hidden turfs [1], is only known by the server, and revealed to clients only when appropriate.[35]

Unity provides its own networking solutions (e.g., Netcode for GameObjects, Unity Transport Package), which can be explored for implementing the client-server communication. Alternatively, a custom backend server in a separate language could be developed, though this introduces the challenge of maintaining synchronization between server-side game logic and client-side game logic.[103] For "Turf War," an authoritative server is crucial for competitive play, preventing cheating, and ensuring a fair experience. The management of state synchronization in a turn-based context involves careful handling of secret information and deterministic random outcomes, all validated by the server.

7.2. Monetization and Live Service Strategy

"Turf War" is designed as a virtual tactical card game that includes elements like card collection, deck building, and guilds.[1] These features align well with the

Free-to-Play (F2P) model, which is prevalent in digital card games and mobile gaming.[104] In an F2P model, the game is free to download and play, generating revenue through optional purchases.

Key monetization strategies for "Turf War" could include:

- **In-App Purchases (IAPs):** These are digital goods or benefits players can buy within the game.[104] For a card game, IAPs typically involve:
  - **Card Packs/Boosters:** The primary source of new cards for collection and deck building.[34]
  - **Cosmetic Items:** Skins for leaders, tokens, or card backs, allowing players to customize their experience without affecting gameplay balance.[104]
  - **In-game Currency:** Bundles of virtual currency that can be spent on card packs, cosmetics, or other in-game items.[104]
  - **Consumables/Boosts:** Temporary benefits like XP boosts or extra turns in PVE modes, if applicable.[104]
- **Battle Pass System:** A popular monetization strategy that offers a series of tiers with rewards earned by completing challenges.[104] Battle Passes typically have a free track (basic rewards for all players) and a premium track (more valuable and exclusive rewards requiring purchase).[104] They are time-limited, creating urgency and encouraging consistent play.[104]
- **Live Service Content Updates:** As a live service game, "Turf War" would receive a continuous stream of new content after its initial release.[7] This includes:
  - **Regular Content Updates:** New cards, leaders, game modes (PVE challenges, new PVP formats), and seasonal events.[110]
  - **Balance Changes:** Adjustments to card stats or rules to maintain a healthy meta and keep gameplay fresh.[8]
  - **Community Engagement:** Fostering a strong community through social features like guilds [1], leaderboards [111], and in-game chat. Guilds provide a sense of belonging and encourage collaboration, driving player engagement and retention.[111]

A well-balanced F2P model with In-App Purchases, a Battle Pass system, and continuous content updates is essential for fostering long-term engagement and generating sustainable revenue for a live service game. The goal is to offer real value to players without resorting to "pay-to-win" mechanics, which can alienate the free player base.[104] Regularly updating the game with new content, features, and events keeps players interested and coming back.[110] This continuous evolution, combined with strong community building strategies, transforms the game into a dynamic platform that retains players over extended periods, maximizing its customer lifetime value.[104]

Conclusions & Recommendations

Developing "Turf War" in Unity, as outlined in this report, provides a robust framework for a turn-based tactical card game with PVP elements and aspirations for a live service model. The foundational steps, from proper Unity project setup to the implementation of core data systems using ScriptableObjects and a Finite State Machine for game state management, are critical for long-term project health. The detailed approach to hexagonal board generation, dynamic instantiation of game elements, and event-driven card play logic ensures a flexible and extensible core. Furthermore, the emphasis on UI/UX, consistent visual art styles, and strategic audio integration is designed to create an engaging and immersive player experience.

For a junior developer embarking on this journey, the following recommendations are paramount:

1. **Prioritize Modular Design and Data Separation:** Continuously strive to separate game logic from data and presentation. ScriptableObjects are your best friends for this, enabling designers to iterate on content without code changes. This practice will save immense time and effort as the game grows and evolves, especially if it transitions into a live service model.
2. **Embrace State Machines:** For turn-based games, a well-defined Finite State Machine is indispensable. It provides clarity, enforces rules, and simplifies debugging by ensuring that game logic only executes when it's supposed to.
3. **Design for Multiplayer from Day One:** Since "Turf War" is PVP, plan for an authoritative server model and robust state synchronization. Retrofitting multiplayer is significantly more complex and time-consuming than building with it in mind from the start.
4. **Iterate on Player Feedback:** Even in early stages, get feedback on core mechanics and UI. For a live service game, player feedback is a continuous loop

that drives development and ensures the game remains relevant and engaging.

5. **Focus on Audio-Visual Feedback:** For a tactical game with a unique "reaction window," clear and immediate audio and visual cues are essential. They enhance player understanding, create urgency, and deepen immersion.
6. **Start Simple, Expand Incrementally:** The provided steps cover a broad scope. For a junior developer, it is recommended to build a "vertical slice" first—a fully functional but minimal version of the core gameplay loop (e.g., one card type, one token, basic turn flow)—before expanding to all card types, abilities, and advanced features. This iterative approach allows for validation of core mechanics early in development.

By adhering to these principles and diligently following the step-by-step guidance, a junior developer can successfully build a robust prototype of "Turf War" and gain invaluable experience in developing complex, scalable, and engaging interactive experiences in Unity.

Works cited

1. Turf war .docx.pdf
2. Start Your Creative Projects and Download the Unity Hub, accessed July 12, 2025, https://unity.com/download
3. Project setup processes - Unity Learn, accessed July 12, 2025, https://learn.unity.com/tutorial/project-setup-processes
4. Best Game Engines for Beginner Game Developers in 2024 - Game Design Skills, accessed July 12, 2025, https://gamedesignskills.com/game-development/video-game-engines/
5. Unity vs Unreal Engine: Which One to Choose in 2025 - RocketBrush Studio, accessed July 12, 2025, https://rocketbrush.com/blog/unity-vs-unreal-engine-which-one-should-you-choose-in-2024
6. Unreal Engine vs Unity 3D Games Development: What to Choose? - AIS Technolabs, accessed July 12, 2025, https://www.aistechnolabs.com/blog/unreal-engine-vs-unity-games-development
7. Part I: Best Practices for Live Service Game Campaigns - AMT Lab @ CMU, accessed July 12, 2025, https://amt-lab.org/blog/2025/3/best-practices-for-live-service-game-campaigns
8. Live Service Games: A Strategic Playbook for Marketing Teams - InAppStory, accessed July 12, 2025, https://inappstory.com/blog/live-service-games
9. Live service game - Wikipedia, accessed July 12, 2025, https://en.wikipedia.org/wiki/Live_service_game
10. 10 Top Tips for Video Game Sound Effect Design, accessed July 12, 2025,

https://sound.krotosaudio.com/video-game-sound-effects-tips/
11. Modular Design - Gaming Glossary - Lark, accessed July 12, 2025, https://www.larksuite.com/en_us/topics/gaming-glossary/modular-design
12. Enhancing Game Development with Modular Assets - Tripo AI, accessed July 12, 2025, https://www.tripo3d.ai/blog/collect/enhancing-game-development-with-modular-assets-9bjb_g-kpvc
13. Architect your code for efficient changes and debugging with ScriptableObjects | Unity, accessed July 12, 2025, https://unity.com/how-to/architect-game-code-scriptable-objects
14. Fast-Paced Multiplayer (Part I): Client-Server Game Architecture - Gabriel Gambetta, accessed July 12, 2025, https://www.gabrielgambetta.com/client-server-game-architecture.html
15. Best practices for transitioning from single to multiplayer P2P in a turn-based card game : r/godot - Reddit, accessed July 12, 2025, https://www.reddit.com/r/godot/comments/1bjvx05/best_practices_for_transitioning_from_single_to/
16. Robust Networking in Multiplayer Games - Caltech Computer Science, accessed July 12, 2025, https://courses.cms.caltech.edu/cs145/2011/robustgames.pdf
17. Separate Game Data and Logic with ScriptableObjects - Unity, accessed July 12, 2025, https://unity.com/how-to/separate-game-data-logic-scriptable-objects
18. ScriptableObject - Unity - Manual, accessed July 12, 2025, https://docs.unity3d.com/6000.1/Documentation/Manual/class-ScriptableObject.html
19. Turn Based Strategies - basics and engine choices - Game Development - Idle Forums, accessed July 12, 2025, https://www.idlethumbs.net/forums/topic/10162-turn-based-strategies-basics-and-engine-choices/
20. I can't believe how much simpler using a finite state machine is : r/Unity3D - Reddit, accessed July 12, 2025, https://www.reddit.com/r/Unity3D/comments/1apx5od/i_cant_believe_how_much_simpler_using_a_finite/
21. State Machines in Unity (how and when to use them) - Game Dev Beginner, accessed July 12, 2025, https://gamedevbeginner.com/state-machines-in-unity-how-and-when-to-use-them/
22. Creating a game manager in Unity - Brian Moakley @ Jezner, accessed July 12, 2025, https://www.jezner.com/2025/02/01/creating-a-game-manager-in-unity/
23. Start menu and Game Manager - Unity Learn, accessed July 12, 2025, https://learn.unity.com/course/tanks-make-a-battle-game-for-web-and-mobile/tutorial/start-menu-and-game-manager?version=6.0
24. Understanding the Singleton Pattern in C# and Unity | by Code_With_K | Medium, accessed July 12, 2025, https://medium.com/@Code_With_K/understanding-the-singleton-pattern-in-c-and-unity-f5abd1ab80bb

25. Everything you need to know about Singleton in C# and Unity - DEV Community, accessed July 12, 2025, https://dev.to/devsdaddy/everything-you-need-to-know-about-singleton-in-c-and-unity-n40
26. Unity Hex Map Tutorials - Catlike Coding, accessed July 12, 2025, https://catlikecoding.com/unity/tutorials/hex-map/
27. imurashka/HexagonalLib: Hexagonal lib for .NET - GitHub, accessed July 12, 2025, https://github.com/imurashka/HexagonalLib
28. Hexagonal Grids - Red Blob Games, accessed July 12, 2025, https://www.redblobgames.com/grids/hexagons/
29. Hex Game - Randomly Building Game Boards - Mike Lynch, accessed July 12, 2025, https://mikelynchgames.com/game-development/hex-game-randomly-building-game-boards/
30. Handle a Tap or Click on a GameObject in Unity 3D with this simple script | by Riley Bolen, accessed July 12, 2025, https://medium.com/@RileyBolen/handle-a-tap-or-click-on-a-gameobject-in-unity-3d-with-this-simple-script-c470a07f0970
31. Click to Move System Using Raycasting in Unity | by Chris Hilton - Medium, accessed July 12, 2025, https://christopherhilton88.medium.com/click-to-move-system-using-raycasting-in-unity-b544abd7326e
32. Instantiating Prefabs at runtime - Unity - Manual, accessed July 12, 2025, https://docs.unity3d.com/530/Documentation/Manual/InstantiatingPrefabs.html
33. A Comprehensive Guide to Generating Entity Prefabs at Runtime in Unity ECS, accessed July 12, 2025, https://dev.to/rk042/a-comprehensive-guide-to-generating-entity-prefabs-at-runtime-in-unity-ecs-16o4
34. Cards, Universe & Everything 4+ - App Store, accessed July 12, 2025, https://apps.apple.com/us/app/cards-universe-everything/id1484798863
35. Networking of a turn-based game - Hacker News, accessed July 12, 2025, https://news.ycombinator.com/item?id=30352710
36. Beginning Game Development: Implementing Timers and Delays with Coroutines - Medium, accessed July 12, 2025, https://medium.com/@lemapp09/beginning-game-development-implementing-timers-and-delays-with-coroutines-5a93d16d173e
37. Understanding Unity Coroutines: A Timer System Example and Best Practices, accessed July 12, 2025, https://dunkingdoggames.com/understanding-unity-coroutines-a-timer-system-example-and-best-practices/
38. Coroutines in Unity: The Power of Asynchronous Programming | by Metehan Demir, accessed July 12, 2025, https://medium.com/@mthndmr16/coroutines-in-unity-the-power-of-asynchronous-programming-3ad534286e3d
39. Using coroutines in Unity - LogRocket Blog, accessed July 12, 2025,

https://blog.logrocket.com/using-coroutines-unity/
40. How to make a countdown timer in Unity - Wayline, accessed July 12, 2025, https://www.wayline.io/blog/how-to-make-a-countdown-timer-in-unity
41. How to make a countdown timer in Unity (in minutes + seconds) - Game Dev Beginner, accessed July 12, 2025, https://gamedevbeginner.com/how-to-make-countdown-timer-in-unity-minutes-seconds/
42. Ep. 14 Show and hide panel - unity tutorial - YouTube, accessed July 12, 2025, https://www.youtube.com/watch?v=e0feEWLRSYI
43. Hiding a UI panel from script in unity - Stack Overflow, accessed July 12, 2025, https://stackoverflow.com/questions/28773818/hiding-a-ui-panel-from-script-in-unity
44. Scripting API: UI.Text.text - Unity - Manual, accessed July 12, 2025, https://docs.unity3d.com/2017.3/Documentation/ScriptReference/UI.Text-text.html
45. How to update text in Unity UI - Stack Overflow, accessed July 12, 2025, https://stackoverflow.com/questions/52749704/how-to-update-text-in-unity-ui
46. Advanced Windows Game Development Techniques - Number Analytics, accessed July 12, 2025, https://www.numberanalytics.com/blog/advanced-windows-game-development-techniques
47. The Role of Sound Design in Immersive Gaming Experiences | Pingle Studio, accessed July 12, 2025, https://pinglestudio.com/blog/the-role-of-sound-design-in-immersive-gaming-experiences
48. A Deep Dive into the Importance of Sound Effects in Game Design - MoldStud, accessed July 12, 2025, https://moldstud.com/articles/p-a-deep-dive-into-the-importance-of-sound-effects-in-game-design
49. Ui Popup Sound Effects - Pond5, accessed July 12, 2025, https://www.pond5.com/search?kw=ui-popup&media=sfx&pp=13
50. Free ui sounds - Uppbeat, accessed July 12, 2025, https://uppbeat.io/sfx/category/digital-and-ui/ui
51. The Art of Audio Feedback in Games - Number Analytics, accessed July 12, 2025, https://www.numberanalytics.com/blog/audio-feedback-in-games
52. Video game UI designer - job description with missions, studies, salary, accessed July 12, 2025, https://gamingcampus.com/careers/ui-designer.html
53. UI/UX Designer Job Description for 2025 - Simplilearn.com, accessed July 12, 2025, https://www.simplilearn.com/tutorials/ui-ux-career-resources/ui-ux-designer-job-description
54. How to Create an Isometric Game: A Detailed Guide - 300Mind, accessed July 12, 2025, https://300mind.studio/blog/how-to-create-an-isometric-game/
55. Pleasing isometric hexagons - Game Development Stack Exchange, accessed July 12, 2025,

https://gamedev.stackexchange.com/questions/5199/pleasing-isometric-hexagons
56. How to create Isometric view in Unity - Chidre's Tech Tutorials, accessed July 12, 2025, https://www.chidrestechtutorials.com/gaming/unity/creating-isometric-view.html
57. How to Draw Isometric Art Using a Hex Grid | Illustrator Tutorial - YouTube, accessed July 12, 2025, https://www.youtube.com/watch?v=WdIM3nMCQ2s
58. How to make ISOMETRIC game environments | Tutorial 2025 - YouTube, accessed July 12, 2025, https://www.youtube.com/watch?v=Tpy8124TErc
59. Best Practices for 3D Modular Design Workflow : r/gamedev - Reddit, accessed July 12, 2025, https://www.reddit.com/r/gamedev/comments/5bblpf/best_practices_for_3d_modular_design_workflow/
60. How can I design good continuous (seamless) tiles? - Game Development Stack Exchange, accessed July 12, 2025, https://gamedev.stackexchange.com/questions/26152/how-can-i-design-good-continuous-seamless-tiles
61. How to Make Seamless Textures For Your Materials #gamedev #tutorial #godot - YouTube, accessed July 12, 2025, https://www.youtube.com/watch?v=nt9frkuOZSg
62. Seamless Textures - Architextures, accessed July 12, 2025, https://architextures.org/textures
63. Isometric Wall Texture Pack by Screaming Brain Studios - Itch.io, accessed July 12, 2025, https://screamingbrainstudios.itch.io/isometric-wall-texture-pack
64. 20+ Best Pastel Color Palettes for 2025 - Venngage, accessed July 12, 2025, https://venngage.com/blog/pastel-color-palettes/
65. Color in digital painting. The only guide you'll ever need - mleczny mlecz, accessed July 12, 2025, https://www.mlecznymlecz.com/color-in-digital-painting/
66. How to Improve Vibrancy and Contrast in Washed-Out Digital Paintings in Just a Few Minutes: A Quick Art Hack for Digital Artists! - Strike A Spark, accessed July 12, 2025, https://strikeaspark.wordpress.com/2018/05/15/how-to-improve-vibrancy-and-contrast-in-washed-out-digital-paintings-in-just-a-few-minutes-a-quick-art-hack-for-digital-artists/
67. [technique] How does this artist make their work so vibrant while using not particularly vibrant colors? : r/ArtistLounge - Reddit, accessed July 12, 2025, https://www.reddit.com/r/ArtistLounge/comments/1k73x0e/technique_how_does_this_artist_make_their_work_so/
68. Try These 3 Game Changing Techniques to Unlock Hidden Colors! Pastel Painting Tutorial, accessed July 12, 2025, https://www.youtube.com/watch?v=jPXC9LC9nQ4
69. Color Palette Challenge - Pinterest, accessed July 12, 2025, https://www.pinterest.com/pin/698058010978313399/
70. Vibrant Self-Portrait in Bold Digital Art Style - Easy-Peasy.AI, accessed July 12, 2025,

https://easy-peasy.ai/ai-image-generator/images/vibrant-self-portrait-painting-expressive-facial-features
71. Game Color Palette - Pinterest, accessed July 12, 2025, https://www.pinterest.com/ideas/game-color-palette/935289433631/
72. Struggling to choose a colour palette for my game, either pastel type colours or a more realistic approach? thoughts? : r/IndieDev - Reddit, accessed July 12, 2025, https://www.reddit.com/r/IndieDev/comments/1ip7gv2/struggling_to_choose_a_colour_palette_for_my_game/
73. Best 2D Video Game Art Styles: Pixel Art to Isometric and Realistic - RocketBrush Studio, accessed July 12, 2025, https://rocketbrush.com/blog/best-2d-video-game-art-styles-from-pixel-art-to-isometric-and-realistic-games
74. Procreate Color Palettes - Bardot Brush, accessed July 12, 2025, https://bardotbrush.com/procreate-color-palettes/
75. What size should your assets be? | HD 2D GAME ART - YouTube, accessed July 12, 2025, https://www.youtube.com/watch?v=MrPoCGHM80E
76. Pastel Painting Made Easy: Learn Texture, Color, Confidence - Realism Today, accessed July 12, 2025, https://realismtoday.com/pastel-painting-made-easy-learn-texture-color-confidence/
77. Digital Painting Tutorial: How To Color Soft Pastel Portrait | Color Therapy Adult Coloring, accessed July 12, 2025, https://www.youtube.com/watch?v=9tUHagKrIVU
78. Optimizing Performance In Unity 3D Game Development — Tips & Tricks - Medium, accessed July 12, 2025, https://medium.com/@eleanor-charlotte/optimizing-performance-in-unity-3d-game-development-tips-tricks-ce13aab37b97
79. Configuring your Unity project for stronger performance, accessed July 12, 2025, https://unity.com/how-to/project-configuration-and-assets
80. Hex Strategy Map: Render - Nick Chavez, accessed July 12, 2025, https://nicolaschavez.com/projects/hex-map-render/
81. Place a stair - Intergraph Smart 3D - Help - Hexagon, accessed July 12, 2025, https://docs.hexagonppm.com/r/en-US/Intergraph-Smart-3D-Structure/13/54565
82. Tips for Sound Design for Games - University of Silicon Valley, accessed July 12, 2025, https://usv.edu/blog/tips-for-sound-design-for-games/
83. Basics of Sound Design for Video Games - Dan Frost, accessed July 12, 2025, https://frost.ics.uci.edu/ics62/BasicsofSoundDesignforVideoGames-MichaelCullen.pdf
84. Mastering Game Sound Design Principles - Number Analytics, accessed July 12, 2025, https://www.numberanalytics.com/blog/mastering-game-sound-design-principles
85. Lessons learned in audio feedback for game and app design | by Fernando Lins | Medium, accessed July 12, 2025, https://medium.com/@fernando1lins/lessons-learned-in-audio-feedback-for-ga

me-and-app-design-e4818c9b72fd
86. Sound effects system design - Game Development Stack Exchange, accessed July 12, 2025, https://gamedev.stackexchange.com/questions/47152/sound-effects-system-design
87. How to make playing card noises sound more satisfying and full, accessed July 12, 2025, https://sound.stackexchange.com/questions/17889/how-to-make-playing-card-noises-sound-more-satisfying-and-full
88. All Cards - SoundInGames.com - Sound Design in Games, accessed July 12, 2025, https://soundingames.dei.uc.pt/index.php/All_Cards
89. www.numberanalytics.com, accessed July 12, 2025, https://www.numberanalytics.com/blog/ultimate-guide-ui-audio-design-game-audio#:~:text=UI%20audio%20design%20is%20the,character%20interactions%2C%20and%20narrative%20progression.
90. Mastering UI Audio Design - Number Analytics, accessed July 12, 2025, https://www.numberanalytics.com/blog/ultimate-guide-ui-audio-design-game-audio
91. UI Sound Design - UI Design Handbook - Design+Code, accessed July 12, 2025, https://designcode.io/ui-design-handbook-ui-sound-design/
92. Designing Game Menu/Interface Sounds? - Sound Design Stack Exchange, accessed July 12, 2025, https://sound.stackexchange.com/questions/11604/designing-game-menu-interface-sounds
93. How can I implement a real-time game loop in a text-based game?, accessed July 12, 2025, https://gamedev.stackexchange.com/questions/92533/how-can-i-implement-a-real-time-game-loop-in-a-text-based-game
94. Sound Design for Video Games: Best Practices and Techniques - Mello Studio, accessed July 12, 2025, https://www.mellostudio.com/en/sound-design-for-video-games/
95. Sound Cues in Gaming: What Does It Mean? - Onlyfarms.gg, accessed July 12, 2025, https://onlyfarms.gg/wiki/general/sound-cues-in-gaming
96. Beginner's game-audio questions | VI-CONTROL, accessed July 12, 2025, https://vi-control.net/community/threads/beginners-game-audio-questions.155126/
97. Game Audio Explained - Your Guide To Making Sound For Games | A Sound Effect, accessed July 12, 2025, https://www.asoundeffect.com/gameaudioexplained/
98. Audio Middleware & How To Use It — Game Audio Learning Portal, accessed July 12, 2025, https://www.gameaudiolearning.com/knowledgebase/audio-middleware-and-how-to-use-it
99. TAKING AUDIO FEEDBACK - League of Gamemakers, accessed July 12, 2025, https://www.leagueofgamemakers.com/taking-audio-feedback/

100. These are your 50 favourite strategy games of all time | Rock Paper Shotgun, accessed July 12, 2025, https://www.rockpapershotgun.com/these-are-your-50-favourite-strategy-games-of-all-time
101. "Not a clear-cut answer" whether Final Fantasy 17 will be turn-based, says Naoki Yoshida, following Clair Obscur's success - Eurogamer, accessed July 12, 2025, https://www.eurogamer.net/not-a-clear-cut-answer-whether-final-fantasy-17-will-be-turn-based-says-naoki-yoshida-following-clair-obscurs-success
102. www.construct.net, accessed July 12, 2025, https://www.construct.net/en/forum/construct-3/how-do-i-8/sync-host-peer-multiplayer-184326#:~:text=Syncing%20refers%20to%20updating%20synced,then%20synced%20back%20to%20peers.
103. Client Server architecture for a multiplayer card game : r/godot - Reddit, accessed July 12, 2025, https://www.reddit.com/r/godot/comments/9yrhxi/client_server_architecture_for_a_multiplayer_card/
104. Successful Game Monetisation Strategies for Free-to-Play Games - The Game Marketer, accessed July 12, 2025, https://thegamemarketer.com/insight-posts/successful-game-monetisation-strategies-for-free-to-play-games
105. How do successful mobile games balance free-to-play features with monetization? - Quora, accessed July 12, 2025, https://www.quora.com/How-do-successful-mobile-games-balance-free-to-play-features-with-monetization
106. Preventing and Managing In-App Purchases | Supercell Support Portal, accessed July 12, 2025, https://support.supercell.com/clash-of-clans/en/articles/preventing-and-managing-in-app-purchases.html
107. The Evolution of Battle Pass, Event Pass, and Season Pass Systems - Gamigion, accessed July 12, 2025, https://www.gamigion.com/the-evolution-of-battle-pass-event-pass-and-season-pass-systems/
108. Game Monetization For Battle Passes - Meegle, accessed July 12, 2025, https://www.meegle.com/en_us/topics/game-monetization/game-monetization-for-battle-passes
109. Battle Pass - Balancy Documentation, accessed July 12, 2025, https://en.docs.balancy.dev/game_templates/battle_pass/
110. The Variety of Design With Live Service Games - Game Wisdom, accessed July 12, 2025, https://game-wisdom.com/critical/live-service-games
111. Social Features in Game Design - Number Analytics, accessed July 12, 2025, https://www.numberanalytics.com/blog/ultimate-guide-to-social-features-in-game-design
112. Creating a Winning Strategy: How Gamification Drives Player Engagement - Optimove, accessed July 12, 2025,

https://www.optimove.com/blog/gamification-strategies-to-drive-player-engagement
113. Card Games in the Library - Ideas & Inspiration from Demco, accessed July 12, 2025, https://ideas.demco.com/blog/card-games-in-the-library/
114. Card Game Development: A Guide - Mind Studios Games, accessed July 12, 2025, https://games.themindstudios.com/post/card-game-development/
115. Running a Successful Live Service Game: Live Outside of Game Updates - - Routledge, accessed July 12, 2025, https://www.routledge.com/Running-a-Successful-Live-Service-Game-Live-Outside-of-Game-Updates/Vasiuk/p/book/9781032718200