Feature Specification - Core Game Architecture

Overview

This document outlines the implementation requirements for the Core Game Architecture of Kuroro: Beast Tactics. This module serves as the foundation for the entire game, managing game state, progression, and communication between all other systems.

Goals

- Create a robust state management system to handle game flow
- Implement a turn-based system that enforces game rules
- Establish communication pathways between all game subsystems
- Store and manage persistent game data
- Provide interfaces for other systems to interact with game state

Technical Requirements

1. Game State Manager

1.1 Game States

Implement a state machine with the following states:

- GameSetup: Initial game setup, team selection, map generation
- TurnStart: Beginning of a new turn (weather events, start-of-turn effects)
- PlayerInput: Players select movement for their Beasts
- HazardRolls: Roll for Beasts in hazardous biomes
- TurnOrderDetermination: Roll for Beast movement order
- TurnExecution: Execute Beast movements in order
- TurnEnd: Process end-of-turn effects
- GameOver: Handle victory conditions and end game

1.2 Turn Counter

- Track the current turn number (starts at 0 for setup, increments to 1 for first actual turn)
- Provide access to turn information for weather duration, passive ability triggers, etc.

1.3 Player Management

- Track current active player
- Store player colors (Red, Blue, Green, White)
- Track Beast ownership
- Handle alternating choices during setup

1.4 State Transitions

- Implement verification for state transitions
- Enforce completion of required actions before state changes
- Implement coroutines for timed transitions between states

2. Event System

2.1 Event Types

Define events for key game occurrences:

- OnTurnBegin/OnTurnEnd
- OnBeastMove
- OnCombat
- OnBeastDestroyed
- OnBeastLevelUp
- OnBiomeRevealed
- OnWeatherChange
- OnShardCollected
- OnBiomeTransformed

2.2 Event Subscription Management

- Implement registration/unregistration of event listeners
- Handle event priorities and execution order
- Support event parameter passing for data exchange
- Manage garbage collection to prevent memory leaks

3. Data Persistence

3.1 Save/Load System

- Define serializable data structures for game state
- Create saving/loading interface for game progress
- Support both auto-save and manual save
- Implement error handling for corrupted saves

3.2 Game Configuration

- Create settings manager for game parameters
- Support adjustable values for:
 - Biome distribution rates
 - Weather event frequency
 - o Shard spawn chances
 - Debug/testing options

4. Subsystem Communication

4.1 System Interfaces

Define interfaces for all major subsystems:

- IMovementSystem
- ICombatSystem
- IFogOfWarSystem
- IBiomeSystem
- IWeatherSystem
- IElementalSystem
- IBeastEvolutionSystem

4.2 Service Locator

- Implement service locator pattern for subsystem access
- Provide central access point for subsystem queries
- Handle initialization order and dependencies

4.3 Command Queue

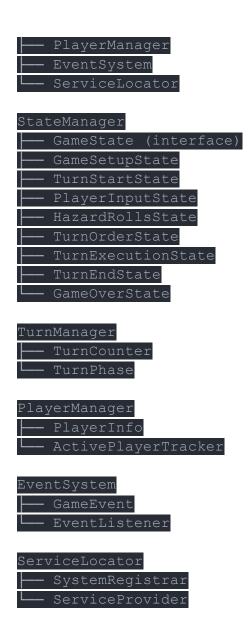
- Create command queue for sequential execution of game actions
- Support timed delays between commands for visual feedback
- Implement command validation
- Enable command bundling for simultaneous effects

Implementation Details

Class Structure

Copy





Key Methods and Properties

GameManager

- Initialize(): Bootstrap game systems
- ChangeState (GameState newState): Transition to new game state
- GetSubsystem<T>(): Get reference to a game subsystem
- SaveGame()/LoadGame(): Handle game persistence
- CurrentTurn: Get current turn number
- CurrentPlayer: Get current active player
- IsGameInProgress: Check if game has started

StateManager

- EnterState(): Initialize state
- UpdateState(): Handle logic while in state
- ExitState(): Clean up before leaving state
- CanTransitionTo(GameState nextState): Validate if transition is allowed

EventSystem

- RegisterListener(GameEventType eventType, Action<object> callback): Add event listener
- UnregisterListener(GameEventType eventType, Action<object> callback): Remove event listener
- TriggerEvent(GameEventType eventType, object data): Trigger event

Dependencies

- Unity Event System (for UI interaction)
- JSON utilities (for serialization)
- Dependency Injection system (optional)

Integration Points

Map Generation System

- The GameManager calls MapGenerator during GameSetupState
- MapGenerator reports completion for state transition

UI System

- UI elements subscribe to events from EventSystem
- UI components query GameManager for game state

Player Input System

- Input system interacts with GameManager during PlayerInputState
- Input validation rules retrieved from GameManager

Technical Constraints

- Utilize ScriptableObjects for configurable values
- Ensure single responsibility principle for all classes
- Implement proper error handling and debugging
- Components should be testable in isolation

Avoid direct GameObject references; use ID-based lookup instead

Acceptance Criteria

- 1. Game progresses through all states correctly without errors
- Events are triggered at appropriate times and all subscribers receive notifications
- 3. Game state can be saved/loaded correctly at various points
- 4. All subsystems can communicate through defined interfaces
- 5. Turn flow matches the rulebook progression
- 6. Game correctly enforces rules about turn order and actions
- 7. System is extensible for future feature additions

Development Timeline

1. State machine implementation: 1 days

2. Event system: 1 day

3. Game manager and core services: 1 days

4. Persistence system: 1 day5. Integration and testing: 1 days

Total estimated time: 5 days

Notes for Implementation

- Begin with implementing the state machine framework
- Use extensive logging during development for state transitions
- Build visualization tools for state and event debugging
- Consider using the Command pattern for turn actions to simplify undo functionality
- Implement dependency injection to make testing easier
- Create dummy implementations of dependent systems for isolated testing

Resources

- Reference the LUA code for turn flow implementation details
- State machine pattern examples in Unity
- Event system implementation best practices
- Service locator pattern tutorials