Last Ember:
A 3D Open-World Survival with Dynamic Environment

Muhammad Atif Rafique
01-135221-069

Taimur Tariq
01-135221-052

Supervisor: Mr. Adnan Jelani
Co-Supervisor: Dr. Adil Khan

Bachelor of Science in Computer Science

Department of Computer Science
Bahria University, Islamabad

December 2025

C e r t i f i c a t e

We accept the work contained in the report titled **"Last Ember: A 3D Open-World Survival with Dynamic Environment"**, written by Mr. **Taimur Tariq** and Mr. **Muhammad Atif Rafique** as a confirmation to the required standard for the partial fulfillment of the degree of Bachelor of Science in Computer Science.

Approved by:

Supervisor: Mr. Adnan Jelani

Co-Supervisor: Dr. Adil Khan

Internal Examiner:

External Examiner:

Project Coordinator: Dr. Kabeer Ahmed Bhatti (Fyp Coordinator IT Department)

Head of the Department: Dr. Muhammad Khurram Ehsan (Head Of Department Computer Science)

December 2025 .

Abstract

The survival games have become so popular in recent years, yet most of the current ones lack any long-term interaction, have repetitive AI, and have stagnating scenery. Games like Minecraft, The Forest, and Rust have deep foundations but are usually lacking dynamic environmental components, adaptive AI activity, and a significant advancement framework. Such flaws create the repetitive nature of the gameplay that lowers the replay value and engagement especially in individuals who want more in-depth survival systems and a sense of any form of progression.

In order to overcome these shortcomings, the game was created as a survival game that can be played on low to mid range hardware without compromising game depth. Examples of features introduced in the game include adaptive AI, seasonal environments, and a balanced resource system that all contribute to increasing long-term engagement of a player. The mix of strategy, exploration, and environmental feeling will enable Last Ember to not only cross the stagnation experienced by other survival titles but also be accessible to many people.

The idea behind the game is to build a high performance game which is also innovative in its gameplay systems. Since it is written in Unity and C#, AI behaviors are modeled as finite state machine and NavMesh pathfindings, so that the wildlife and aggressive animals can respond to the player action in an unpredictable manner. The seasonal changes, the weather that is not predicted (rain, mist, and others) and the statistics of being alive (health, hunger, thirst, stamina) make the players vary their strategies during the game. Others that have a time of year restrictions are farming, building bases on top of one another and crafting, which provides the game with a top-down experience of survival. A user-friendly interface is also desirable to encourage accessibility and quests and exploration provide some direction and narration.

The outcome is a survival game which is differentiated by offering dynamic and changing challenges. In contrast to the fixed survival worlds, Last Ember presents dynamic fauna, waves of enemies and an environment that gets more challenging as the game progresses. The systems that are created can support both short term tactical decision making and long term strategic planning with increased replayability. The game can be played on relatively low-end systems without any issues due to optimization and still provides a detailed experience of an open world, which proves not only its technical effectiveness but also an appealing game.

It is a new step in the survival genre that will solve old-time problems of predictable AI and a lack of diversity. The project has a balance between the realism and the entertainment through innovative mechanics, adaptive systems and performance orientation. Not only does it give its players a great one of a kind and replayable survival game, but it is also a great study in the field of AI design, optimization, and the creation of immersive games.

Acknowledgments

We would also like to give our sincere thanks to everyone who helped to complete our Final Year Project, Last Ember: A 3D Open-World Survival with Dynamic Environment successfully. This project represents several months of creativity, education, and struggle, and, without the help and encouragement of numerous people, it could not have happened.

To begin with, we would like to offer our utmost gratitude to our faculty advisors, Mr. Adnan Jelani and Dr. Adil Khan, who have been guiding us, giving a lot of encouragement, and helpful recommendations in the process of preparing this project. Their recommendations enhanced the technical excellence of the project and motivated us to look at problems creatively and persistently.

We are also expressly thankful to the students who have attended our testing and provided their feedback. Your testing and recommendations proved very instrumental in the development of the user interface of the game, the balance of the gameplay and the experience of the player. The experience with actual users allowed us to develop a game that is immersive and dynamic, as well as accessible and enjoyable to players.

Finally, we are very grateful to our families for their constant support, patience, and encouragement throughout this journey. Your faith in our abilities and your understanding during long hours of coding, designing, and testing have been invaluable.

This project shows the dedication, creativity, and teamwork that went into bringing "Last Ember" to life.

Muhammad Atif Rafique & Taimur Tariq
Bahria University
E-8, Islamabad, Pakistan
December 2025

"The moment a player enters your game, they're entering a contract of trust. They expect a world that responds to them, mechanics that make sense, and goals that are worth chasing. Building a game isn't just about code or design—it's about delivering on that promise. When we get it right, we create something people don't just play—they remember."

— Ken Levine, Creative Director of BioShock

Contents

List of Tables

List of Figures

Acronyms and Abbreviations

AI	Artificial Intelligence
3D	Three-Dimensional
C#	C Sharp
NPC	Non-Player Character
UI	User Interface
HUD	Heads Up Display
PvE	Player vs. Environment
PvP	Player vs. Player
FR-x	Functional Requirement
NFR-xx	Non-Functional Requirement
FPS	Frames Per Second
PC	Personal Computer
FSMs	Finite State Machines
LOD	Level of Detail
GC	Garbage Collection

Chapter 1

Introduction

1.1 Introduction

The 3D open-world survival game Last Ember offers players a realistic wilderness adventure. It focuses on essential survival aspects such as resource management, item crafting, wilderness battles, settlement building, quest creation, and nighttime enemies. In this vast environment, players encounter natural challenges and increasing dangers that arise at night.

Survival video games have gained popularity due to their engaging gameplay and detailed mechanics. The survival themes found in games like Minecraft, The Forest, and Rust have set new standards for the industry [1]. The Last Ember project aims to enhance survival game elements by improving artificial intelligence control, optimizing resource allocation, and providing meaningful survival functions along with integrated quest objectives [2].

Through the use of dynamic systems, intelligent gameplay design, and environmental storytelling, Last Ember aims to provide players with an immersive survival experience that raises the bar for the genre.

1.2 Motivation

Continuing worldwide economic downturns have not impeded the gaming industry which experienced a dramatic 65% expansion during the past five years. A vast number of gaming enthusiasts worldwide play games through mobile phones and computers and video game consoles simultaneously. The defined growth pattern results from the industry's delivery of multiple gaming options which satisfies various types of gamers.

Recognizing the opportunities in the quickly thriving sector led Unity Technologies to create the effective Unity Engine which functions as a cross-platform game development tool. Through its simple user interface, the Unity technology enables developers at every level to build extraordinary gaming experiences which boost market expansion.

Survival games remain one of the most enduring genres because they incorporate exploration elements alongside crafting mechanics and base-construction features and resource cycle management. Three games including Minecraft, The Forest, and Rust created essential industry standards through their combination of principle gaming features in open-ended gameplay. Their achievements demonstrate how survival video games continue to generate

popularity because they force players to adjust their strategies in always-changing environments.

Last Ember develops previous foundational elements through its 3D open-world survival design that includes deep exploration and AI encounters and meaningful quests as well as dynamic game systems. The developers used Unity to create Last Ember which focuses on survival mechanics that involve gathering resources, crafting, encountering wildlife and base construction and adds night enemies as a constant menace. Last Ember works to develop a comprehensive survival experience that enhances the evolution of survival game programming.

1.3 Problem Statement

The gaming market has experienced substantial expansion of survival titles due to popular games such as Minecraft, The Forest, and Rust. High-quality survival games tend to require strong hardware systems which prevents users with low or middle-grade devices from accessing those games. The high hardware requirements restrict survival game access to markets where users do not possess suitable gaming equipment.

All games in this genre contain survival elements that create interaction issues while AI behavior is foreseeable and game mechanics repeat frequently. The present survival systems found in gaming releases mainly focus on crafting and stamina and health management without providing substantial character advancement or effective quest system integration leading to unspecified objectives.

Last Ember creates an optimized immersive 3D survival game through its development with Unity and C# toolkit. The game functions smoothly on basic to intermediary system configurations to provide an interactive survival adventure. The game offers four essential characteristics: easy-to-use controls, AI-controlled animal behavior, significant quest objectives and adaptive environmental elements spanning across day-night systems and seasonal changes. This project offers an alternative to the survival genre through accessibility and thrilling game mechanics. It also gives the development team the chance to practice artificial intelligence techniques, optimize systems, and improve user experiences.

1.4 Scope

Our goal with Last Ember was to create an optimal 3D survival experience that doesn't require a high-end systems, so we developed it in Unity and C# for low-to-mid-range compatibility. Throughout the project, we've focused on bridging technical requirements with artistic elements—this includes fine-tuning AI, gameplay systems, environmental interactions, and user experience. We really prioritized player interaction and accountability here, aiming for a level of realism that makes the gameplay fun and interactive.

1.4.1 Included Features

Through various well-made game systems and features, this project ensures players can experience true survival adventures.

- **Player Core Functionalities**
  - Player movement with smooth control mechanics.
  - Health, stamina, hunger, and thirst systems for realistic survival challenges.
  - A sleep functionality will enable stamina recovery and time advancement.
- **World Dynamics**
  - Fully explorable open-world terrain with forests, rivers, and wildlife zones.
  - The game system controls enemy populations according to day-night shifts and changes environmental visibility rates which force specific survival methods.
  - Gamers can observe basic weather conditions which adjust both world visibility and environmental temperatures.
- **Artificial Intelligence (AI)**
  - The system implements artificial intelligence to control wildlife behavior patterns through state machines operating in a range of activities from wandering to fleeing and chasing and attacking.
  - The game features hostile night-time foes that activate after darkness arrives through NavMesh pathfinding technology.
  - Seasonal AI variations and behavior adjustments (optional for future expansion).
- **Environment Interaction**
  - The game lets players collect genuine resources like wood, stone and plants and edible ingredients.
  - Crafting system for tools, weapons, and survival gear.
  - The gameplay provides players access to both farming and hunting activities that support survival through sustainability.
- **Combat and Equipment**
  - Melee and ranged combat systems.
  - The system evaluates weapon survival rate while also handling weapon maintenance and construction.
  - Enemy damage systems with health and attack patterns.

- **Base Building & Shelter System**
  - Base building through the combination of foundation modules with wall systems that have door and defensive capabilities.
  - Craftable storage containers, campfires, and utility stations.
- **Quest System**
  - Basic NPC-driven quest system with task tracking and item delivery.
  - The missions involve exploration through environmental interactions.
- **User Interface & Inventory**
  - Backpack system to store collected materials and items with an organized layout.
  - Quick slot toolbar for easy access to essential tools and weapons during gameplay.
  - Crafting chests that store required materials for crafting specific items or tools.
  - Categorized crafting menu, making it easier for players to navigate different item types (e.g., tools, weapons, structures).
  - Clean and user-friendly interface designed for intuitive interaction, minimal clutter, and better focus during survival scenarios.

1.4.2 Excluded Initially

- Multiplayer/Online functionality
- Advanced or branching quest systems
- Full seasonal cycle and weather-based mechanics (rain, snow)
- VR and AR support
- Advanced NPC dialogue trees and trading systems
- Commercial optimization or publishing readiness

1.5 Tools and Technology

- Unity 3D
- C#
- Blender/Maya (for 3D modelling)
- Substance Painter/Designer (for texturing)
- Version control system (e.g., Git)

1.6 Hardware

- **Processor:** Intel(R) Core(TM) i7-8850H CPU @ 2.60GHz (12 CPUs), ~2.6GHz
- **Memory:** 32GB RAM
- **Graphics Card:** NVIDIA Quadro P2000 – 4GB

Chapter 2

Literature Review

2.1 Introduction

The survival game category has exploded over the past decade because it feeds players' most fundamental human wants. Players begin by taming the basics of hunger and thirst while overcoming physical challenges, and eventually achieve personal success by dominating their environment. Our game Last Ember is a 3D open-world survival game and the enemy environment, that constantly changes!, created with Unity 2022.3.58f1! [**?**].

The lack of development of fundamental gameplay loops is a major problem in this genre. Many current games have static, unchanging environments, predictable Artificial Intelligence (AI) patterns, and frequently require expensive hardware, which restricts player accessibility[1]. By incorporating dynamic systems and giving optimization for low- to mid-range specifications top priority, Last Ember seeks to remedy these flaws.

The chapter discusses the current state of the genre Survival, presenting a critical review on prominent actors in industry. It then describes a technical and theoretical framework through review of relevant studies about Finite State Machines (FSM), NavMesh pathfinding, dynamic modeling of environments, and systems for resource management. The goal of this literature review is to help designers and implementers make decisions about how to build a more realistic, performance-optimized, and dynamically engaging 3D adventure survival game.

2.2 Theoretical Foundations of Survival Game Design

In a survival game, resources management and tension due to growing threats is imperative in the design [1]. Yes they typically do gameplay mechanics, notee by the industry fall under harvesting, crafting and combat systems: (GD Principles) These systems are the ones that make up our game loop. Characters alternate back and forth between the "Expedition" phase in which they explore to gather resources and the "Rest" phase where they craft, cook, and heal at a base [3].

For a survival game to keep players engaged over time, the difficulty must increase in a non-linear way. Just making enemies stronger is not enough; the environment should introduce new obstacles and limit resources in different areas[4]. This idea shapes the design

of Last Ember's seasonal system, which goes beyond the simple day-night cycles seen in early games.

Additionally, the absence of a clear story in many sandbox survival games puts more pressure on the game mechanics to draw in players. This happens through Emergent Narrative, where the tools and abilities given to players allow for many unexpected interactions, creating unique and personal stories for each player [5]. Last Ember mixes this emergent aspect with a clear quest system driven by NPCs, appealing to both sandbox fans and those who enjoy narrative content.

2.3 Analysis of Existing Market Systems

To find specific market gaps and set functional standards, three key survival titles were examined for their gameplay execution and technical implementation.

2.3.1 Case Study: The Forest

The Forest (Endnight Games, 2014) [6] sets a standard for environmental tension and enemy AI quality. Its antagonist AI uses complex Behavior Trees to control entities. This allows them to observe the player, retreat when they get hurt, and work together in tactical moves. This approach goes well beyond simple patrol or chase patterns [2]. The depth of this AI is important for creating mood. It serves as a solid example of the behavioral complexity we want in Last Ember's wildlife and hostile NPCs.

Limitations: Even with its advanced AI, The Forest's [6] world remains mostly static. The environment does not present systemic challenges beyond enemy patrols. This reduces the feeling of long-term survival strategy once players set up a secure base.

2.3.2 Case Study: Rust

Rust (Facepunch Studios, 2013) [7] is known for its strong modular base-building and tiered technology progression. It offers a clear path from basic tools to modern industrial structures. The crafting system and overall design provide useful insights for Last Ember's own structure and building mechanics.

Limitations: Rust [7] trades off deep PvE complexity to focus on PvP. Its non-player character AI is minimal. Also, its high system requirements make it hard for players with low- to mid-range hardware to access, which is an important market segment that Last Ember aims to reach [1].

2.3.3 Case Study: Raft

Raft (Redbeet Interactive, 2018) [8] offers a unique approach to resource management. It shows that interesting resource loops can happen in tightly controlled environments.

Limitations: The main threats in Raft [8] are easy to predict. They usually follow simple attack patterns based on timers. The consistency of the ocean environment means players rarely need to change their survival tactics. This leads to repeated mechanics once a

player learns the basic loop. Last Ember aims to avoid this predictability by using advanced, adaptive AI and dynamic, changing biomes.

2.3.4 Case Study: ARK: Survival Evolved

ARK: Survival Evolved (Studio Wildcard, 2015) [9] is a key example of large-scale, persistent open-world survival that focuses on managing resources and taming creatures. It features a successful tiered technology system and a huge, ever-changing game map. Its complexity supports the idea of blending detailed crafting with constant environmental challenges.

Limitations: ARK: Survival Evolved [9] uses a lot of resources and needs high-end hardware for smooth gameplay because of its complex physics and rendering process. Its AI behavior is mostly limited to basic aggression and pathfinding. It relies on large numbers and big creature models instead of advanced behavior to create threats. This highlights Last Ember's focus on optimization and effective, smart AI.

2.4 Advanced Technical Implementations

The successful development of a high-performance 3D open-world game depends on the careful use of new techniques in AI and rendering.

2.4.1 Intelligent Non-Player Character (NPC) Design

The AI for Last Ember uses a common mix of Finite State Machines and Pathfinding.

Finite State Machines (FSM)

Last Ember uses FSMs to control NPC behavior because they are predictable and require low processing power [2]. An FSM arranges behavior into distinct states like Idle, Flee, Attack, and Hunt. It only changes states when certain conditions are satisfied [10]. Although complex FSMs can be difficult to manage, they are clearly more efficient and simpler to fix than complicated Behavior Trees or more demanding Deep Reinforcement Learning models for the specific behaviors needed from survival game antagonists [2]. The FSM design in Last Ember includes hysteresis, which adds a delay in transition conditions to stop rapid shifts between states. For example, an enemy would not quickly switch between "Chase" and "Idle" at the detection boundary. This design feature helps create smoother, more realistic NPC movement [11].

NavMesh and Pathfinding

For AI entities to move through the complex, adjustable 3D landscape of Last Ember, the Unity Navigation Mesh (NavMesh) system is used [12]. NavMesh pre-calculates the walkable areas of the environment, which allows for fast pathfinding by creating an optimized polygonal map. This is different from older, less efficient systems like A* that rely on grid structures. The mix of FSM and NavMesh helps AI dynamically figure out the best path to a target, avoiding obstacles and following the environment's spatial rules [10].

2.4.2 Dynamic Environmental and Gameplay Modeling

Environmental changes and dynamic interactions are essential for increasing difficulty in Last Ember. This goes beyond static encounters [4]. The project achieved dynamic complexity through the following main systems:

- **Dynamic Resource and Threat Management:** The environment uses dynamic systems to manage world resources and threats. This approach ensures ongoing challenges beyond just resource depletion [4].
    - **Procedural Resource Generation:** Key survival resources, such as trees, sticks, stone, and animals, randomly appear and regenerate across the map. This design prevents static farming routes and encourages continuous exploration.
    - **Night-Cycle Threat Escalation:** Hostile entities, like zombies, only spawn during the night cycle. This rule limits safe player exploration and increases difficulty, forcing players to prioritize securing a defensive position before dark.
- **Interactive Farming Mechanic:** Interactive Farming Mechanic: The project includes a functional farming loop where crops grow based on direct player actions like watering. This system provides a simple, sustainable food source that connects directly to the player's resource management cycle.
- **Action-Based Stat Modeling:** The system uses mathematical models to calculate how quickly player status effects like Health, Stamina, and Hunger decay and replenish based on real-time effort and inputs [3].
    - **Energy Consumption:** Stamina and Hunger decay rates are linked to intense player activities, such as running, gathering resources, and combat. This relationship ensures that every player action has a measurable impact on short-term survival.
    - **Stat Calculation:** The character's Health, Stamina, and Hunger are calculated by tracking their physical output. This management of these stats is key to the gameplay loop [5].

2.5 Performance Optimization Techniques

The main requirement for the project was to run efficiently on low to mid-range hardware. This led to the careful use of several performance optimization patterns [1]. These techniques aimed to balance the workload between the CPU and GPU. This approach helped maintain a stable frame rate and reduced memory usage.

2.5.1 GPU Workload Management: Distance-Based Culling (LOD Implementation)

For managing the GPU workload when rendering the vast open world, the project used a very efficient Distance-Based Culling strategy as a simple Level of Detail (LOD) approach

[1]. This technique was achieved entirely by configuring built-in Unity parameters without needing custom scripting. This maximized engine efficiency:

- **Camera Clipping Plane:**The camera's far clipping plane was adjusted to create a clear limit on the visible geometry. It removed all objects beyond this set distance.
- **Terrain Draw Distance:** The draw distances for terrain detail and trees/grass were set to match this clipping plane.

This combined approach makes sure that the rendering pipeline only processes the needed objects and terrain details close to the player. By greatly cutting down the overall number of visible draw calls at any time, this method reduces the GPU workload and keeps performance stable, all while maintaining good visual quality in the player's immediate area [13, 12].

2.5.2 CPU and Logic Efficiency Patterns

To reduce the CPU time used by game logic each frame, several key programming and design patterns were used to distribute the workload and cut down on unnecessary calculations.

Asynchronous Task Management via Coroutines

To mitigate frequent **Garbage Collection (GC) spikes** and prevent frame rate stuttering caused by executing high-load logic simultaneously, the project utilized Coroutines (`IEnumerator` / `StartCoroutine`). This approach enables computationally intensive tasks, such as managing the persistent state via the `SaveManager`, updating the complex `FishingSystem`, or dynamically spawning resources, to be executed asynchronously or distributed over several sequential frames. This practice ensures that per-frame processing remains low and consistent, significantly enhancing runtime stability.

Targeted Physics Optimization

Physics calculations were optimized to ensure rapid and efficient interaction checks. Instead of querying the entire scene, systems like the `PlacementSystem` utilize the **Physics.Raycast** function in conjunction with Layer Masks (e.g., `LayerMask.GetMask("Ground")`). This targeted approach restricts the raycasting checks exclusively to specific, relevant layers, preventing the physics engine from processing irrelevant geometry. This methodology significantly reduces the computational overhead associated with broad-spectrum physics queries, thereby ensuring fast, optimized interactions.

Dynamic Component Culling

Performance was enhanced through a mechanism of **Dynamic Component Culling**, where non-essential scripts and UI systems are disabled when their functionality is not required. For example, input managers such as the `MovementManager` and `SelectionManager` are deactivated while the player is interacting with the inventory or crafting UI. This technique prevents unnecessary execution cycles during each frame, which substantially reduces overall CPU utilization and overhead.

Singleton Design Pattern for System Access

Key persistent manager classes (e.g., `PlacementSystem`, `InventorySystem`, `SelectionManager`) were implemented using the **Singleton pattern**. By centralizing access to these managers, the development avoids the performance bottleneck caused by repeatedly searching the scene hierarchy for objects at runtime (using costly operations like `GameObject.Find()`). This design promotes superior code organization while simultaneously ensuring fast and efficient access to crucial game logic.

2.6 Comparative Analysis Framework

The following table summarizes the key features and strategic positioning of *Last Ember* relative to established genre leaders.

Table 2.1. Comparative Study of Survival Games

Feature	**The Forest [6]**	**Rust [7]**	**Raft [8]**	**ARK: Survival Evolved [9]**	**Last Ember**
Core Focus	Horror / Narrative	PvP / Territory	Collection / Building	Taming / Tiered Tech	**PvE / Dynamic Survival**
AI Implementation	Advanced Behavior Trees	Basic Patrol Loops	Simple Timer-based	Basic Aggression	**FSM with Hysteresis**
Environment Dynamics	Static Day/Night	Static Biomes	Static Ocean	Static Biomes	**Dynamic Resource & Threat**
Progression Model	Story-based	Tech Tree / Blueprints	Research Table	Linear Tech	**Quest-driven & Tiered**
Performance Target	Mid-High Range	High Range	Low-Mid Range	Very High Range	**Optimized for Low-Mid**
Optimization Focus	Standard LOD	Renderer/Phy	Simple Geometry	Complex Renderer	**Distance-Based Culling**

2.7 Conclusion

The review of academic literature and existing commercial titles shows that the survival game genre is popular. However, there is still a significant gap in the market for a game that combines advanced, dynamic systems with high performance accessibility [1]. The analysis of

industry leaders, such as The Forest, Rust, and Raft, reveals key limitations. These include predictable AI, unchanging environments, and system requirements that limit access for low- and mid-range hardware.

The theoretical framework set out in this chapter supports the technical and design choices for Last Ember as a complete solution to these identified issues:

1. **Ensuring Adaptive and Complex Gameplay:** The design effectively addresses the issue of unchanging worlds and expected threats.
    - By adopting the efficient Finite State Machine (FSM) architecture [2] and using the technical foundation of the NavMesh system [10, 12], the project guarantees complex but efficient enemy behavior. This directly avoids the simple timer-based threats seen in benchmark titles [11].
    - Dynamic complexity comes from using Procedural Resource Generation and Night-Cycle Threat Escalation [4]. It works alongside an Action-Based Stat Modeling system [3]. This setup creates a clear, ongoing connection between player effort and survival status.
2. **Prioritizing Performance and Accessibility:** A varied method for improving performance meets the main project goal of supporting low-to-mid-range hardware [1].
    - GPU Efficiency: GPU load is managed through Distance-Based Culling, which is a configured LOD implementation. This method greatly reduces draw calls without needing custom scripting [1].
    - CPU Efficiency: CPU use is kept low through modern coding methods. These include Coroutines for managing tasks asynchronously, Targeted Physics Optimization with Layer Masks, and the Singleton Design Pattern for quick system access. This approach ensures better runtime stability and reduced overhead.

This detailed literature review confirms that the design and implementation choices for Last Ember are logically sound, well-placed in the market, and strongly built to provide an engaging, high-performing, and easily accessible 3D survival experience.

Chapter 3

Requirement Specifications

3.1 Requirement Specifications

Clearly defining and communicating the functional and non-functional needs of a software project or system is essential for its successful implementation. In the case of **Last Ember**, requirement specifications serve as the foundation for development, ensuring that the survival gameplay mechanics, AI systems, resource dynamics, and user experiences are implemented cohesively. These requirements guide development, maintain stakeholder alignment, and ensure quality assurance through testing and feedback cycles.

3.2 Existing Games

Most existing 3D survival games in the market face recurring issues such as poor AI behavior, clunky inventory systems, shallow crafting mechanics, lack of dynamic world changes, and limited player interaction depth. Games like The Forest, Rust, and Last Day on Earth each contribute useful mechanics but fall short in delivering a fully immersive and evolving survival experience. These issues lead to repetitive gameplay, weak engagement, and reduced replayability.

3.2.1 Limitations or Shortcomings of the Existing Games

- **Simplistic or Predictable AI Behavior:** Enemy and wildlife interactions in many games are based on scripted behavior, reducing tension and immersion.
- **Static World Environment:** Lack of real seasonal changes, weather patterns, and environmental dynamics causes the world to feel unchanging over time.
- **Shallow Resource Management and Crafting Systems:** Resource gathering and crafting often lack meaningful depth, reducing their relevance to gameplay progression.
- **Unresponsive Controls and UI:** Clunky menus and unrefined control schemes make survival mechanics like inventory management, crafting, and building feel sluggish.

- **Weak Progression Through Survival Mechanics:** Health, hunger, stamina, and sleep systems are often underutilized, failing to create real strategic depth or risk-reward decisions.

3.3 Proposed System

Last Ember is designed to overcome these limitations by delivering a rich, immersive survival experience. It aims to integrate dynamic environmental factors, intelligent AI, and deep resource systems into a cohesive and evolving open-world game.

3.3.1 The Proposed System Overcomes Existing System Limitations

- **Dynamic AI Behavior:** Enemy and wildlife interactions feature state-machine-based behaviors that adapt to player decisions and environmental changes.
- **Seasonal and Environmental Systems:** Seasons influence farming, animal activity, and weather patterns, forcing players to adjust survival strategies.
- **Deep Crafting and Inventory Management:** Players gather, combine, and refine materials into usable tools, structures, and consumables using an intuitive crafting system and inventory.
- **Smooth and Natural Controls:** Controls and UI are optimized for smooth interactions, enabling natural movement, combat, base-building, and tool management.
- **Meaningful Survival Mechanics:** Status bars (health, hunger, thirst, stamina) interact with food, rest, and environment. These systems impact player efficiency and unlock strategic gameplay.

3.4 Functional Requirements

3.4.1 FR-1: Player Mechanics

FR-1.1	Player can move, sprint, jump, and crouch within the 3D environment.
FR-1.2	Players can swim and drown based on stamina and breath.
FR-1.3	Player can consume food and drink to restore stats.
FR-1.4	Custom cursor enables interaction with world objects.

3.4.2 FR-2: Environment Interaction

FR-2.1	Players can interact with terrain, trees (chop), water, and wildlife.
FR-2.2	Campfires can be built, lit, and used for cooking.
FR-2.3	Trees and vegetation regrow over time.
FR-2.4	Day/night and weather systems dynamically affect visibility, temperature, and gameplay.

3.4.3 FR-3: Inventory & Crafting

FR-3.1	Drag-and-drop inventory system with quick slots.
FR-3.2	Crafting system enables the creation of items using gathered materials.
FR-3.3	Storage system allows item deposit and retrieval.
FR-3.4	Players can equip weapons and tools.

3.4.4 FR-4: AI & Combat

FR-4.1	Enemy NPCs and wild animals have dynamic AI behavior.
FR-4.2	Players can fight, take damage, die, and respawn.
FR-4.3	Stealth and alert systems influence enemy behavior.

3.4.5 FR-5: Base Building & Farming

FR-5.1	Players can build base structures using gathered resources.
FR-5.2	Farming system allows planting, growth, harvesting.
FR-5.3	Cooking system turns raw food into edible meals.

3.4.6 FR-6: Game Systems

FR-6.1	Players can save and load their progress.
FR-6.2	Main menu provides game options, load/save access.
FR-6.3	Quest system offers missions and objectives.
FR-6.4	In-game shop allows item purchase.

3.5 Non-Functional Requirements

NFR-01	Accessibility: The game should be available for play 24/7 on supported platforms.
NFR-02	Compatibility: The game must support various resolutions and hardware specs (optimized for mid-range PCs).
NFR-03	Security: Save data should be protected from corruption.
NFR-04	Performance: Game must load within 15 seconds and run at minimum 30 FPS on target hardware.
NFR-05	Load/Stress Testing: Game should support large environments, with dynamic entities and multiple AI agents active concurrently.
NFR-06	Usability: Players should be able to learn the controls and core gameplay within 10–20 minutes.
NFR-07	Scalability: Game systems should allow for future additions like multiplayer, new quests, or expanded maps without significant rework.
NFR-08	Sound Design: Environmental and interaction-based sound effects enhance immersion and provide feedback cues.

3.6 Use Cases

Table 3.1. Use Case: Player Movement and Navigation

Use Case ID	UC-1.1
Use Case Name	Player Movement and Navigation
Actor(s)	Player
Pre-Conditions	Game is running, player is spawned, MovementManager initialized, CharacterController attached
Priority	High
Basic Flow	1. Player presses WASD keys for movement. 2. MovementManager checks if movement is allowed. 3. PlayerMovement calculates movement vector. 4. CharacterController applies movement with physics. 5. Player uses mouse for camera look. 6. MouseMovement calculates rotation with smoothing. 7. Camera rotation is applied. 8. Player presses Space for jumping. 9. Ground check is performed. 10. Jump velocity is applied if grounded.
Actor Actions	Player controls movement with keyboard and mouse input
System Response	System updates player position, applies physics, handles camera rotation, manages jump mechanics
Alternative Course of Action	Swimming mode when in water, underwater oxygen mechanics, movement disabled when UI open

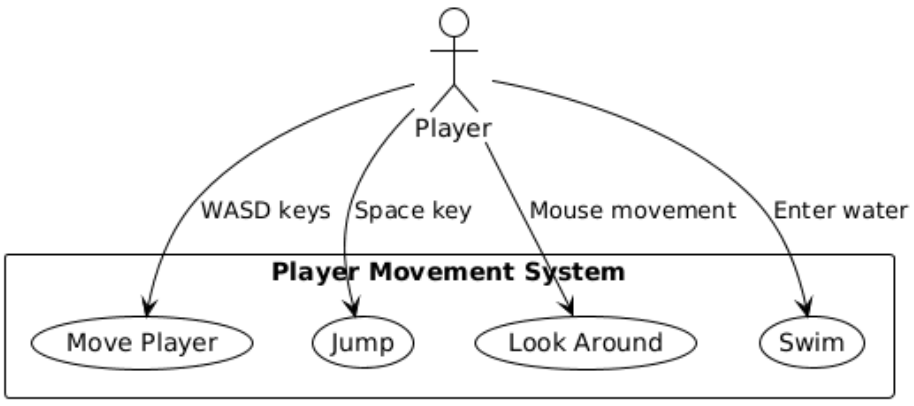

Figure 3.1. Player Movement and Navigation Use Case Diagram

Table 3.2. Use Case: Item Collection and Pickup

Use Case ID	UC-1.2
Use Case Name	Item Collection and Pickup
Actor(s)	Player
Pre-Conditions	Player within 10-unit range of interactable item, inventory system available
Priority	High
Basic Flow	1. Player moves mouse to target item. 2. SelectionManager performs raycast detection. 3. InteractableObject validates player range. 4. Interaction prompt is displayed. 5. Player clicks left mouse button. 6. InventorySystem checks available slots. 7. Item is added to inventory. 8. Item is removed from world. 9. Pickup sound is played. 10. Pickup notification is shown.
Actor Actions	Player targets and clicks on items to collect them
System Response	System detects items, validates range, adds to inventory, removes from world, plays feedback
Alternative Course of Action	Inventory full (pickup denied), out of range (interaction disabled), item not selected (no action)

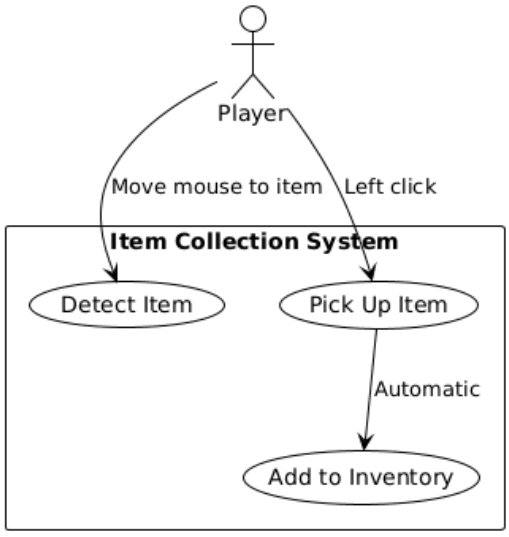

Figure 3.2. Item Collection and Pickup Use Case Diagram

Table 3.3. Use Case: Inventory Management

Use Case ID	UC-1.3
Use Case Name	Inventory Management
Actor(s)	Player
Pre-Conditions	InventorySystem initialized, player has items or inventory slots available
Priority	High
Basic Flow	1. Player presses 'I' key. 2. InventorySystem opens UI. 3. Movement and selection are disabled. 4. Cursor is unlocked and made visible. 5. Inventory contents are displayed. 6. Player browses items and quick slots. 7. Player selects quick slot (1-7). 8. Item is equipped to selected slot. 9. Player presses 'I' to close. 10. UI is closed and controls restored.
Actor Actions	Player manages inventory, assigns quick slots, views item information
System Response	System displays inventory, handles item stacking, manages quick slots, updates UI
Alternative Course of Action	Item stacking (up to 3), item info display on hover, quick slot management

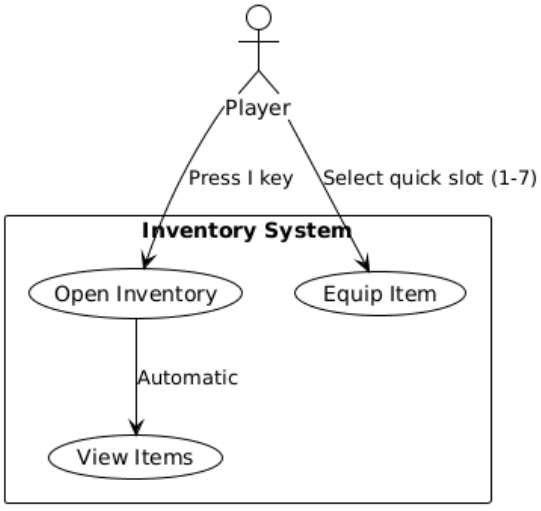

Figure 3.3. Inventory Management Use Case Diagram

Table 3.4. Use Case: Crafting System

Use Case ID	UC-1.4
Use Case Name	Crafting System
Actor(s)	Player
Pre-Conditions	Player has required materials, CraftingSystem available, blueprint exists
Priority	High
Basic Flow	1. Player presses 'C' key. 2. CraftingSystem opens UI. 3. Player selects category (Tools/Survival/Refine/Construction). 4. Available blueprints are displayed. 5. System checks material requirements. 6. Player clicks craft button. 7. Crafting sound is played. 8. Materials are consumed from inventory. 9. 1-second delay for crafting animation. 10. Crafted item sound is played. 11. Items are added to inventory.
Actor Actions	Player selects blueprints, crafts items from materials
System Response	System validates materials, consumes resources, produces items, plays audio feedback
Alternative Course of Action	Insufficient materials (craft disabled), no inventory space (crafting blocked)

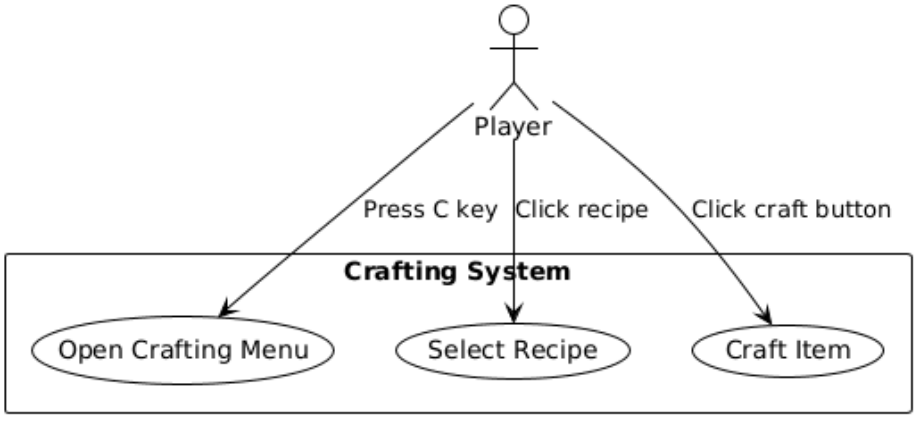

Figure 3.4. Crafting System Use Case Diagram

Table 3.5. Use Case: Building and Construction

Use Case ID	UC-1.5
Use Case Name	Building and Construction
Actor(s)	Player
Pre-Conditions	Player has construction items, ConstructionManager available, PlacementSystem initialized
Priority	Medium
Basic Flow	1. Player enters construction mode. 2. ConstructionManager enables placement mode. 3. Placement preview is activated. 4. Player selects buildable item from inventory. 5. Preview model follows mouse cursor. 6. Grid snapping is applied. 7. Collision detection validates placement. 8. Player clicks to place item. 9. Item is removed from inventory. 10. Buildable object is instantiated.
Actor Actions	Player selects construction items and places them in world
System Response	System shows preview, validates placement, applies grid snapping, creates objects
Alternative Course of Action	Invalid placement (red preview), collision detection blocks placement

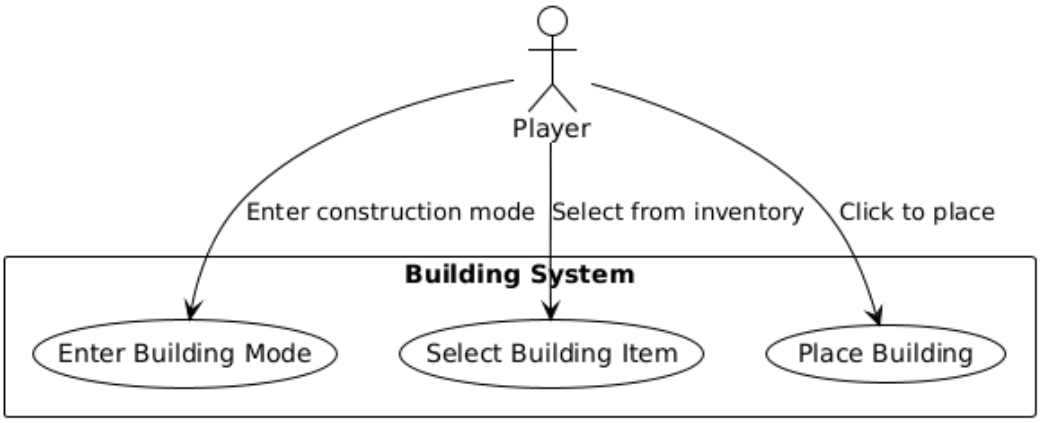

Figure 3.5. Building and Construction Use Case Diagram

Table 3.6. Use Case: Tree Chopping and Resource Gathering

Use Case ID	UC-1.6
Use Case Name	Tree Chopping and Resource Gathering
Actor(s)	Player
Pre-Conditions	Player has axe equipped, tree within interaction range, tree has health points
Priority	High
Basic Flow	1. Player targets tree with mouse. 2. SelectionManager detects tree. 3. Chop interaction prompt is shown. 4. Player clicks with axe equipped. 5. Swing animation is triggered. 6. Swing sound is played. 7. Tree receives damage. 8. Tree health is reduced. 9. If health reaches zero: logs dropped, added to inventory, stump created. 10. Regrowth timer starts.
Actor Actions	Player chops trees with axe to gather wood resources
System Response	System applies damage, reduces tree health, drops logs, manages tree regrowth
Alternative Course of Action	No axe (chopping disabled), tree regrowth over time, multiple hits required

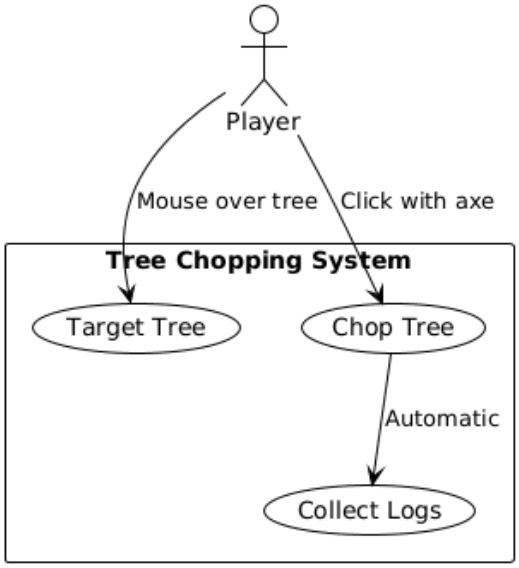

Figure 3.6. Tree Chopping and Resource Gathering Use Case Diagram

Table 3.7. Use Case: Fishing System

Use Case ID	UC-1.7
Use Case Name	Fishing System
Actor(s)	Player
Pre-Conditions	Player has fishing rod equipped, near valid water source (Lake/River/Ocean)
Priority	Medium
Basic Flow	1. Player equips fishing rod. 2. Player approaches water source. 3. Player casts fishing line. 4. 3-second wait period begins. 5. Fish detection calculation occurs. 6. Fish type determined by water source. 7. If fish bites: struggle begins, player pulls, minigame starts. 8. Player plays skill-based minigame. 9. Success/failure determined. 10. Fish added to inventory if successful.
Actor Actions	Player casts line, times fish pull, plays fishing minigame
System Response	System calculates fish probability, manages minigame, determines catch success
Alternative Course of Action	Fish escapes (failed minigame), no bite (random chance), different fish types by location

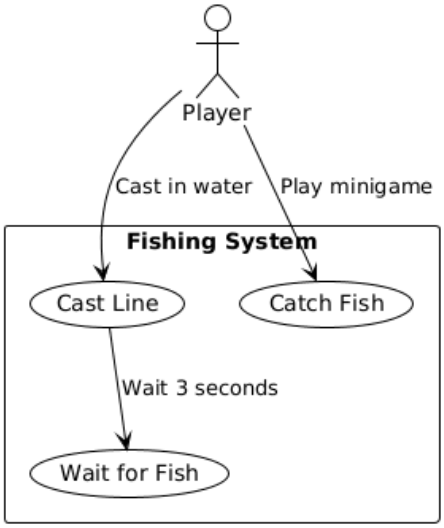

Figure 3.7. Fishing System Use Case Diagram

Table 3.8. Use Case: Cooking and Campfire Management

Use Case ID	UC-1.8
Use Case Name	Cooking and Campfire Management
Actor(s)	Player
Pre-Conditions	Player near campfire, has raw food items, campfire has fuel
Priority	Medium
Basic Flow	1. Player approaches campfire. 2. SelectionManager shows interaction prompt. 3. Player clicks on campfire. 4. CampfireUIManager opens cooking interface. 5. Available food items displayed. 6. Player selects food to cook. 7. Food added to campfire cooking queue. 8. Raw food removed from inventory. 9. Cooking process begins. 10. Progress monitored and displayed. 11. When cooking completes: cooked food available. 12. Player takes cooked food, effects applied.
Actor Actions	Player manages campfire, selects food to cook, takes cooked items
System Response	System manages cooking process, applies food effects, tracks cooking progress
Alternative Course of Action	No fuel (cooking blocked), multiple foods cooking, fuel management required

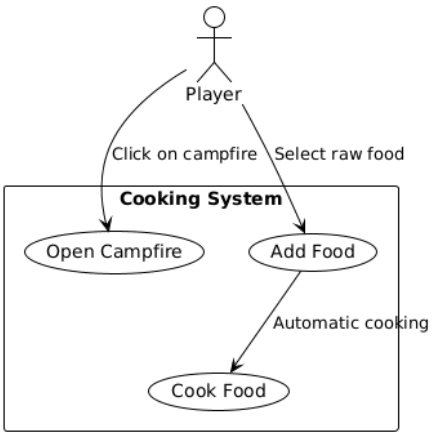

Figure 3.8. Cooking and Campfire Management Use Case Diagram

Table 3.9. Use Case: Equipment System

Use Case ID	UC-1.9
Use Case Name	Equipment System
Actor(s)	Player
Pre-Conditions	Player has items in inventory, EquipSystem initialized, quick slots available
Priority	High
Basic Flow	1. Player assigns items to quick slots. 2. Player presses number key (1-7). 3. EquipSystem selects quick slot. 4. Item equipped to slot. 5. 3D model shown in player's hand. 6. Item capabilities activated. 7. Player uses equipped item. 8. Tool animation triggered. 9. Item effects applied. 10. Swing cooldown prevents spam.
Actor Actions	Player manages equipment, assigns quick slots, uses tools and weapons
System Response	System equips items, shows 3D models, activates capabilities, manages cooldowns
Alternative Course of Action	Empty slot (no effect), tool restrictions, animation cooldown

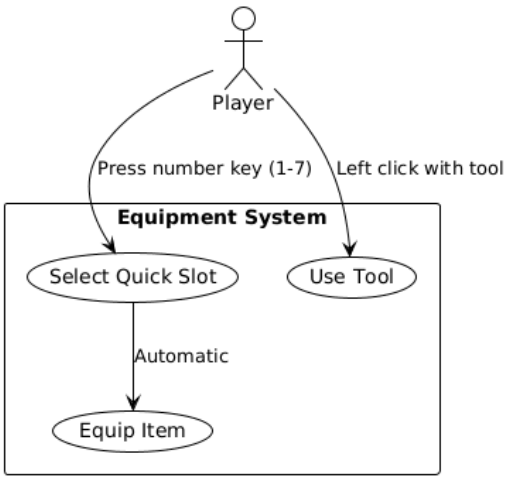

Figure 3.9. Equipment System Use Case Diagram

Table 3.10. Use Case: Quest Management

Use Case ID	UC-1.10
Use Case Name	Quest Management
Actor(s)	Player
Pre-Conditions	QuestManager initialized, NPCs have quests available, player can interact
Priority	Medium
Basic Flow	1. Player presses 'Q' key. 2. QuestManager opens quest menu. 3. Active and completed quests displayed. 4. Quest tracker shows progress. 5. Player interacts with quest giver. 6. Quest added to active list. 7. Quest automatically tracked. 8. Progress monitored in real-time. 9. When requirements met: validation occurs. 10. Required items consumed, rewards given. 11. Quest moves to completed list.
Actor Actions	Player manages quests, tracks progress, completes objectives
System Response	System tracks quest progress, validates completion, applies rewards
Alternative Course of Action	Multiple active quests, manual tracking/untracking, checkpoint quests

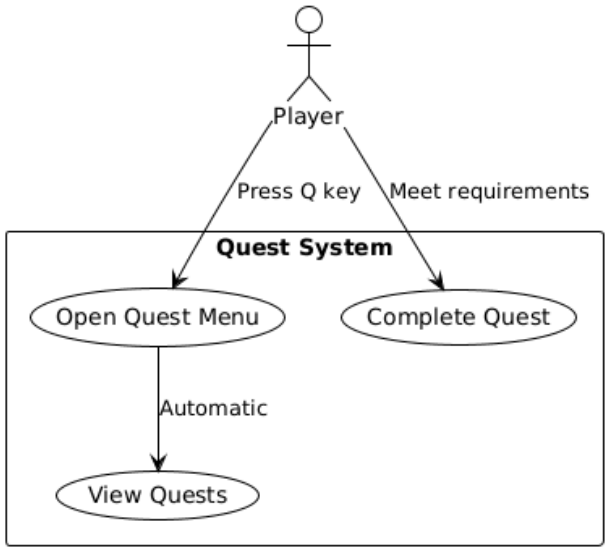

Figure 3.10. Quest Management Use Case Diagram

Table 3.11. Use Case: NPC Interaction and Dialogue

Use Case ID	UC-1.11
Use Case Name	NPC Interaction and Dialogue
Actor(s)	Player
Pre-Conditions	Player within interaction range of NPC, NPC not in conversation, DialogSystem available
Priority	Medium
Basic Flow	1. Player approaches NPC. 2. SelectionManager detects NPC in range. 3. Interaction prompt displayed. 4. Player clicks to start conversation. 5. DialogSystem activates dialogue UI. 6. Dialogue text displayed. 7. Player continues through dialogue options. 8. Based on NPC type: quest options, shop interface, or general dialogue. 9. Player ends conversation. 10. Dialogue UI closed, interaction state reset.
Actor Actions	Player initiates conversations, navigates dialogue, accesses quests and shops
System Response	System manages dialogue flow, provides quest/shop access, handles conversation state
Alternative Course of Action	Quest giving, shop access, dialogue branches with different responses

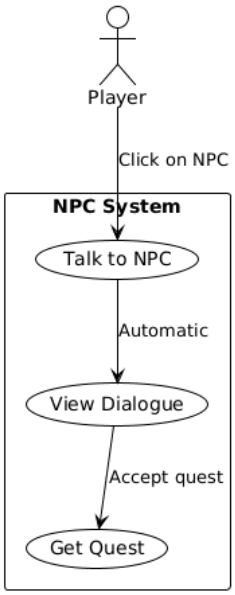

Figure 3.11. NPC Interaction and Dialogue Use Case Diagram

Table 3.12. Use Case: Save and Load Game

Use Case ID	UC-1.12
Use Case Name	Save and Load Game
Actor(s)	Player
Pre-Conditions	Game running, SaveManager initialized, player data available
Priority	High
Basic Flow	1. Player initiates save. 2. SaveManager collects player data (stats, position, inventory). 3. Environment data gathered (items, trees, animals, storage). 4. Data serialized (JSON or Binary). 5. Save file written to disk. 6. Player initiates load. 7. Loading screen displayed. 8. Save file read and deserialized. 9. Player data restored. 10. Environment state reconstructed. 11. Game continues from saved state.
Actor Actions	Player saves and loads game progress in multiple slots
System Response	System preserves/restores complete game state, manages save files
Alternative Course of Action	Multiple save slots, JSON/Binary formats, corrupted save handling

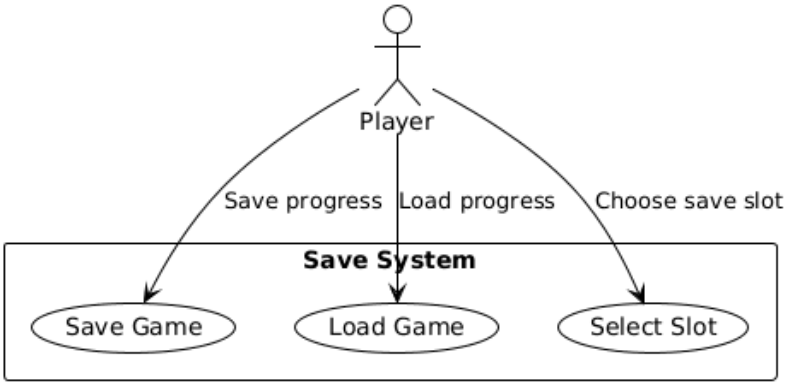

Figure 3.12. Save and Load Game Use Case Diagram

Table 3.13. Use Case: Weather and Day-Night Cycle

Use Case ID	UC-1.13
Use Case Name	Weather and Day-Night Cycle
Actor(s)	Player (Observer)
Pre-Conditions	Game world loaded, environmental systems initialized, time progression active
Priority	Low
Basic Flow	1. DayNightSystem continuously updates time. 2. Current hour calculated (24-second day cycle). 3. Skybox materials updated based on time. 4. Lighting system adjusts directional light. 5. Time UI displays current hour. 6. At midnight: TimeManager triggers next day. 7. WeatherSystem generates random weather. 8. Season-based rain probability calculated. 9. Weather effects applied if rain occurs. 10. Visual effects updated.
Actor Actions	Player observes environmental changes
System Response	System manages time progression, weather generation, visual effects
Alternative Course of Action	Rain effects, seasonal variation, weather transitions

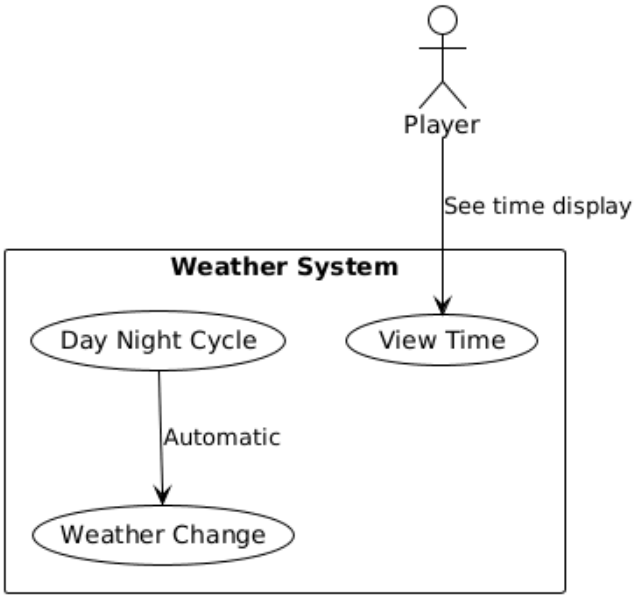

Figure 3.13. Weather and Day-Night Cycle Use Case Diagram

Table 3.14. Use Case: Storage Box Management

Use Case ID	UC-1.14
Use Case Name	Storage Box Management
Actor(s)	Player
Pre-Conditions	Player near storage box, storage box placed in world, StorageManager available
Priority	Medium
Basic Flow	1. Player approaches storage box. 2. SelectionManager detects storage box. 3. "Open" interaction prompt shown. 4. Player clicks on storage box. 5. StorageManager opens storage UI. 6. Storage contents displayed. 7. Player drags items between inventories. 8. Item transfers validated. 9. Storage state updated. 10. Player closes storage UI. 11. Storage data saved.
Actor Actions	Player manages additional storage space through placeable storage boxes
System Response	System handles item transfers, validates capacity, persists storage data
Alternative Course of Action	Capacity limits, item validation, multiple storage boxes

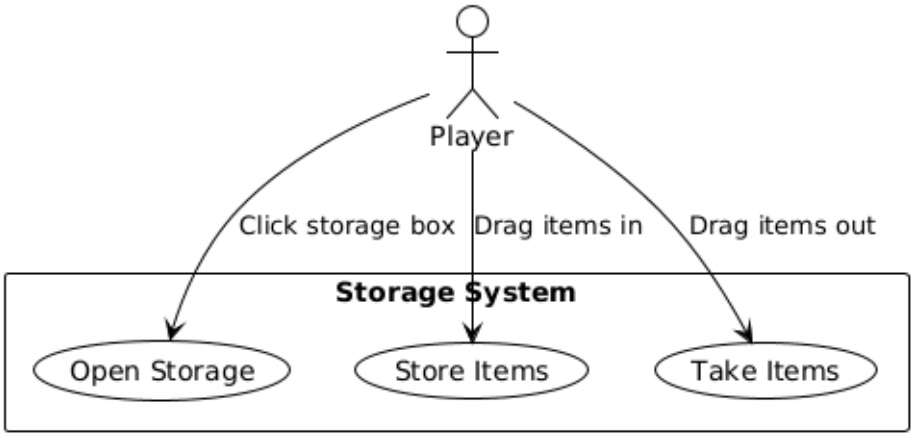

Figure 3.14. Storage Box Management Use Case Diagram

Table 3.15. Use Case: Shop and Trading System

Use Case ID	UC-1.15
Use Case Name	Shop and Trading System
Actor(s)	Player
Pre-Conditions	Player near shop keeper NPC, has currency or items to trade, shop keeper available
Priority	Medium
Basic Flow	1. Player approaches shop keeper. 2. SelectionManager shows "Talk" prompt. 3. Player clicks to start conversation. 4. ShopKeeper opens trading interface. 5. Available items and prices displayed. 6. Player's current coins shown. 7. For buying: player selects item, price validated, inventory checked. 8. Transaction completed, coins deducted, item added. 9. For selling: player selects item, sell price calculated. 10. Item removed from inventory, coins added. 11. Currency display updated. 12. Player closes shop interface.
Actor Actions	Player buys and sells items with NPCs using game currency
System Response	System manages transactions, validates funds/space, updates currency and inventory
Alternative Course of Action	Insufficient funds (buying blocked), no inventory space, different item values

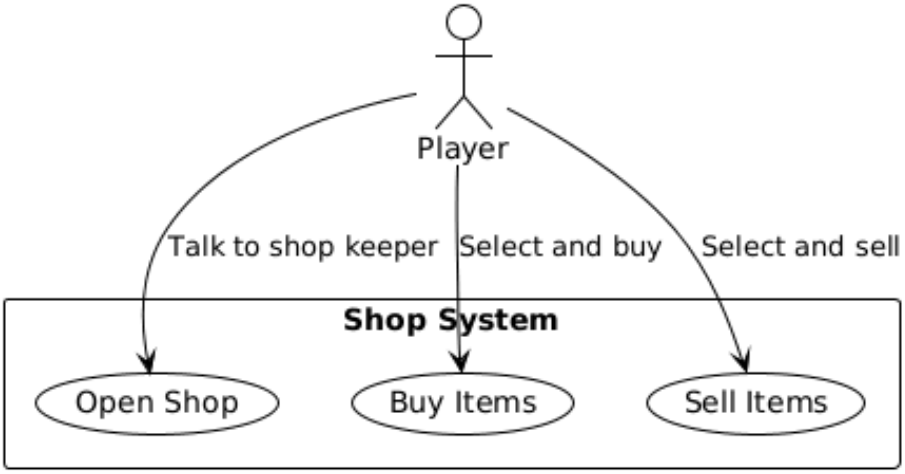

Figure 3.15. Shop and Trading System Use Case Diagram

3.7 System Sequence Diagrams

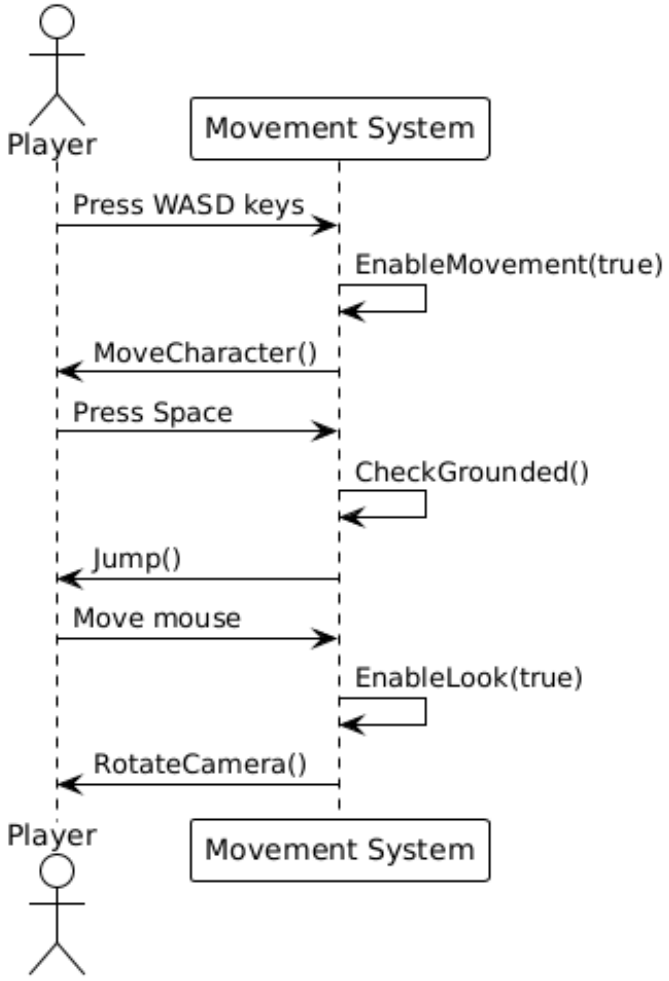

Figure 3.16. System Sequence Diagram - Player Movement

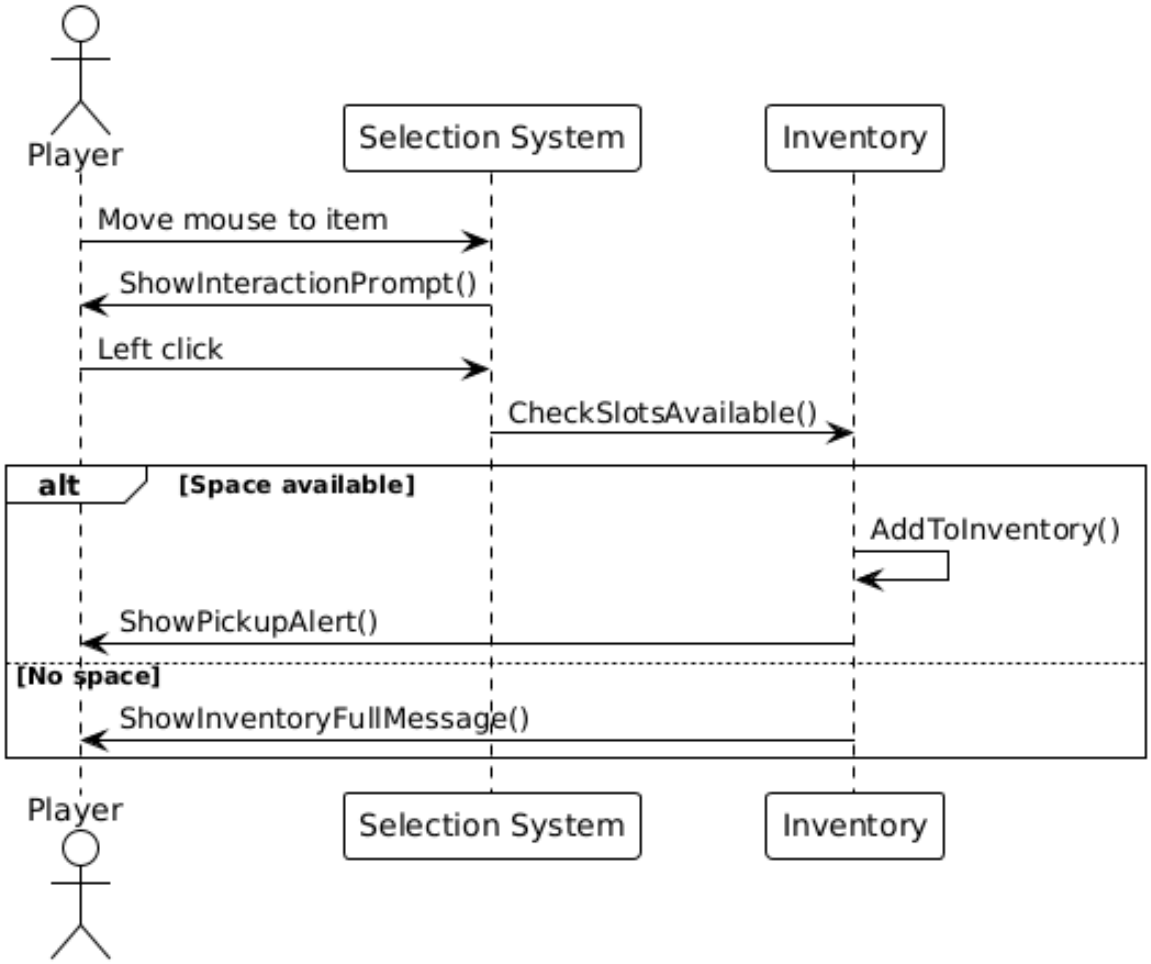

Figure 3.17. System Sequence Diagram - Item Collection

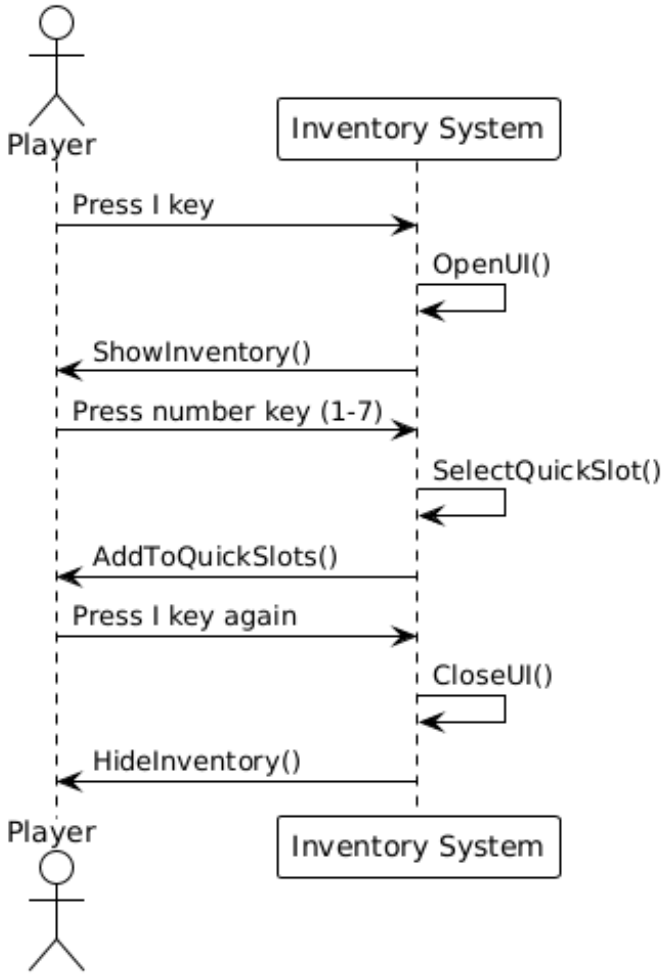

Figure 3.18. System Sequence Diagram - Inventory Management

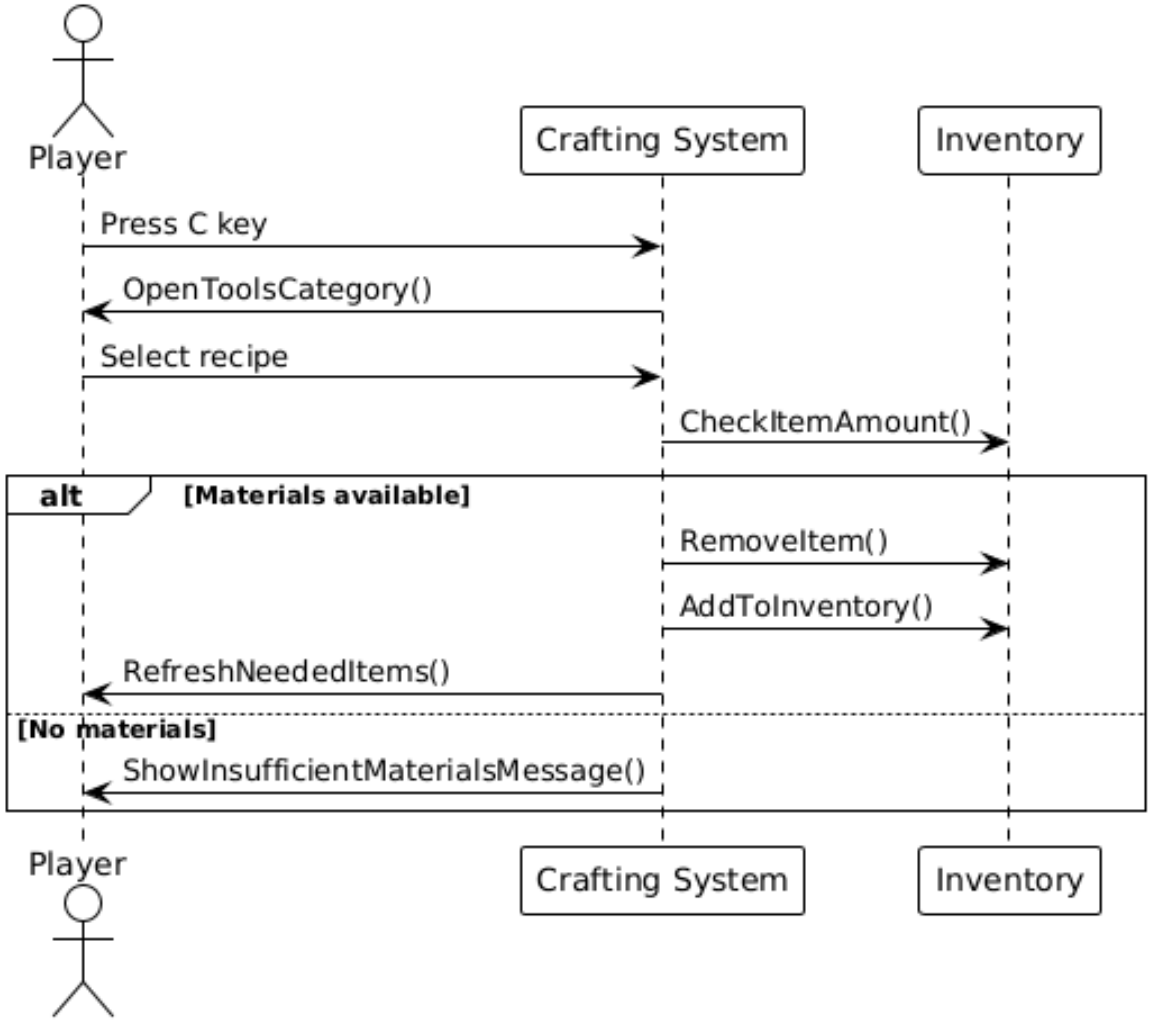

Figure 3.19. System Sequence Diagram - Crafting

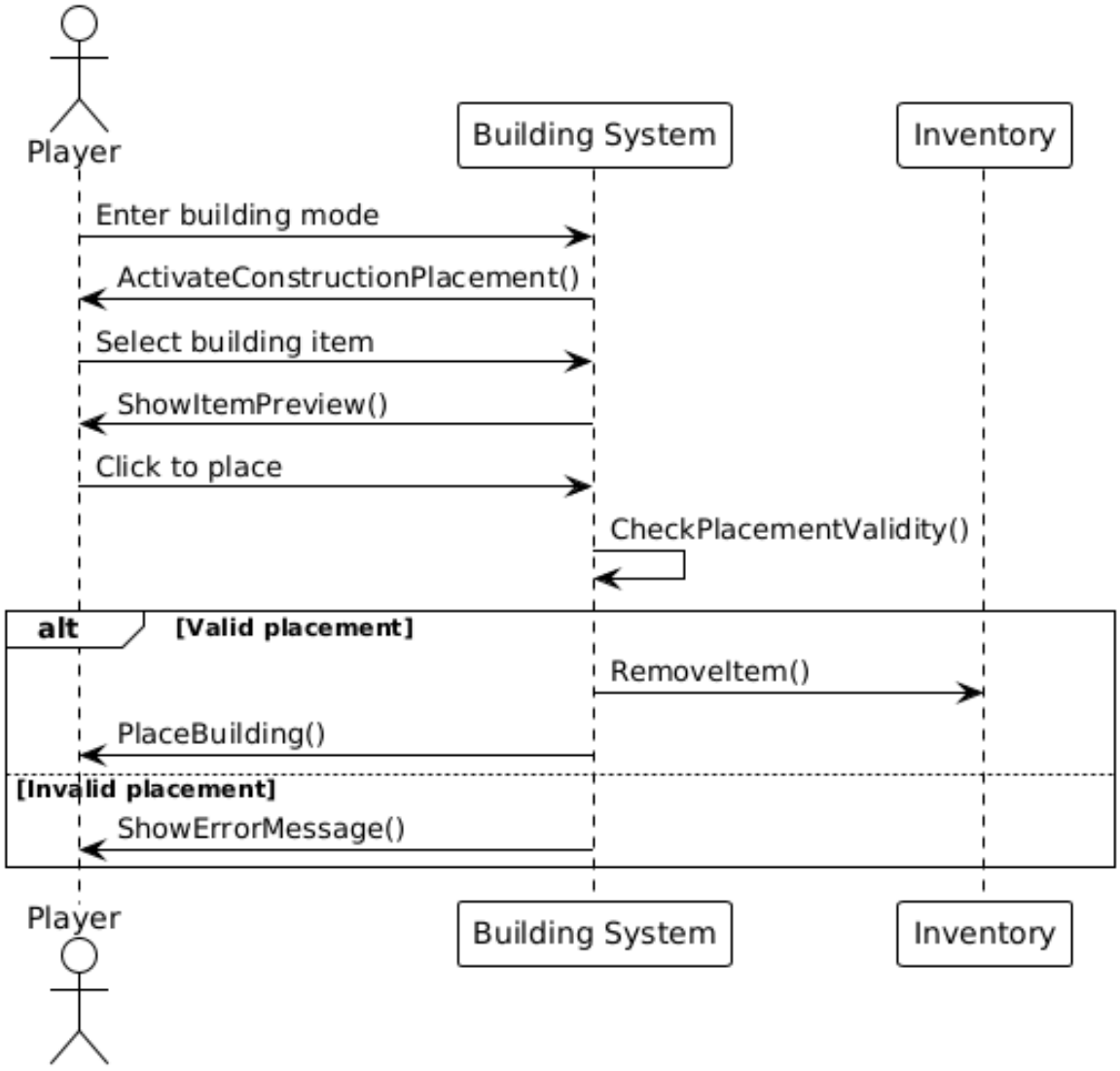

Figure 3.20. System Sequence Diagram - Building

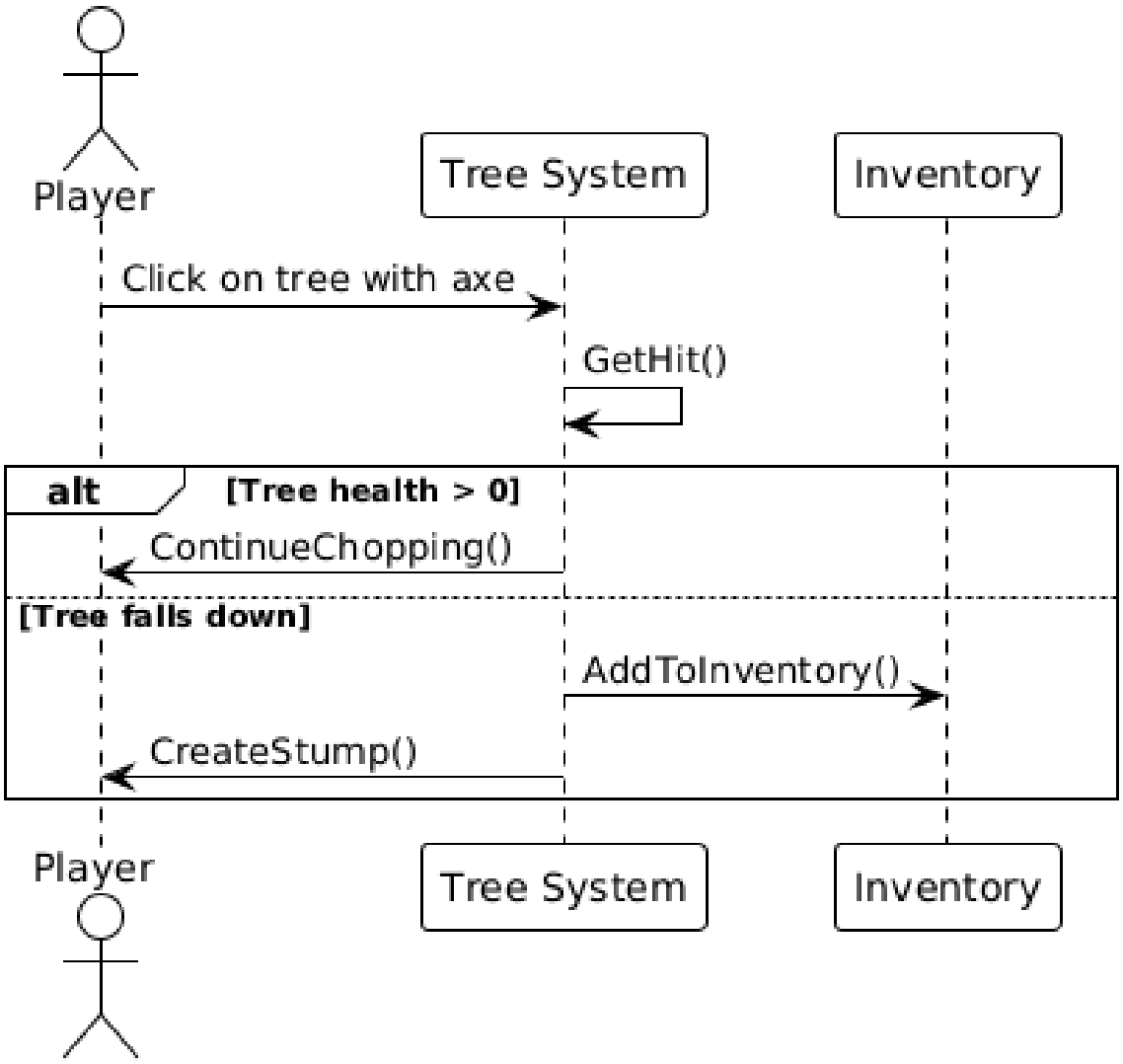

Figure 3.21. System Sequence Diagram - Tree Chopping

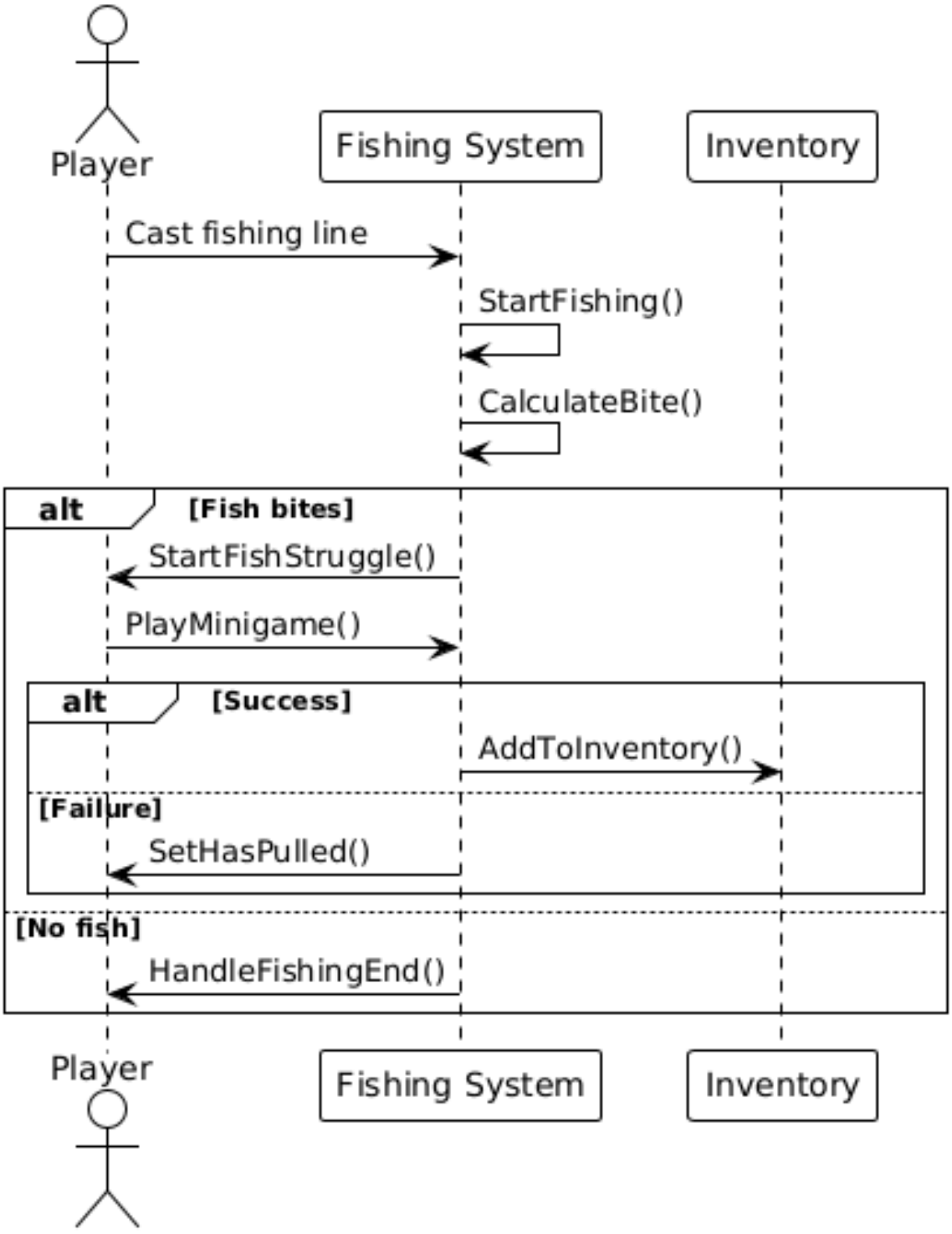

Figure 3.22. System Sequence Diagram - Fishing

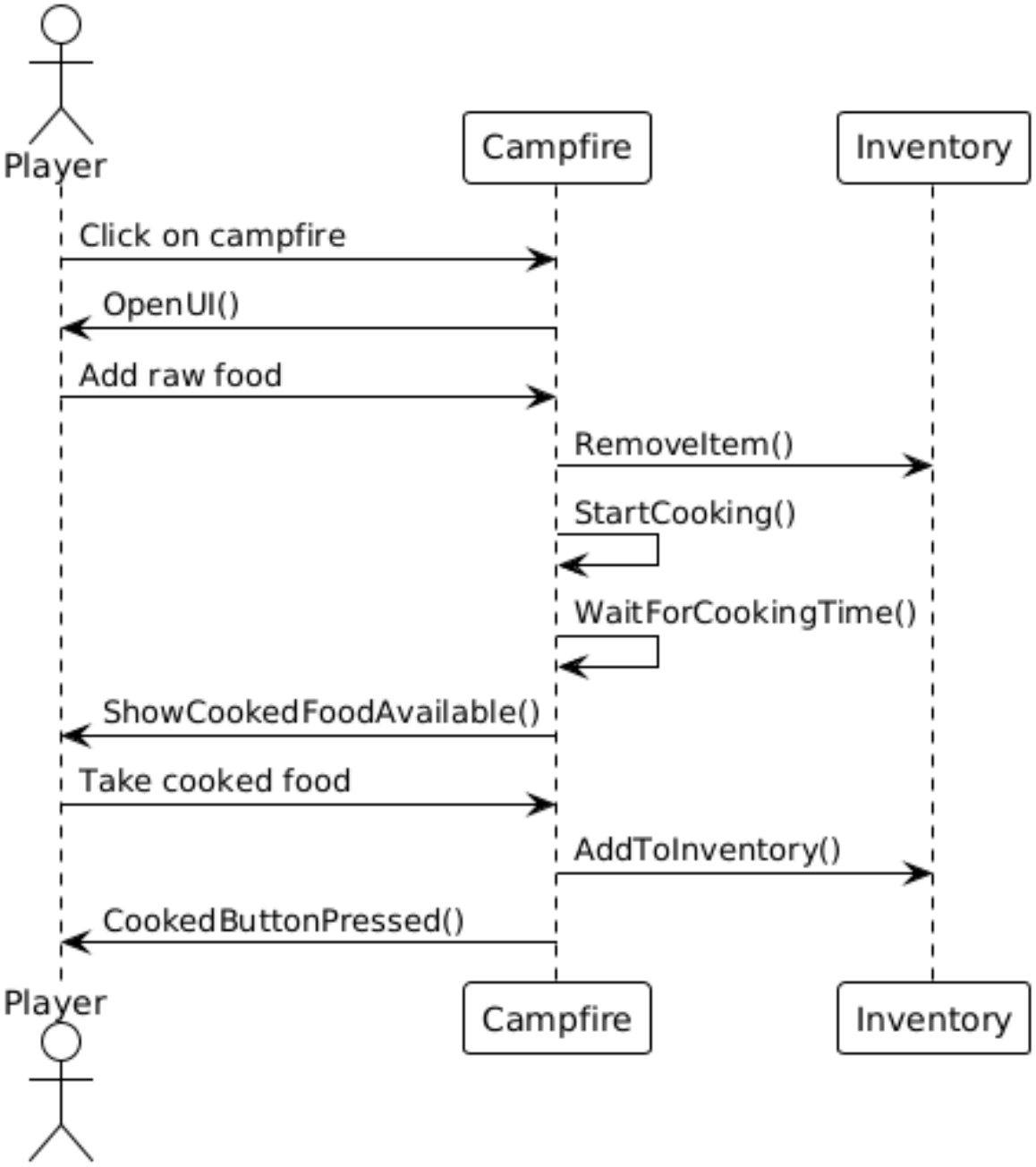

Figure 3.23. System Sequence Diagram - Cooking

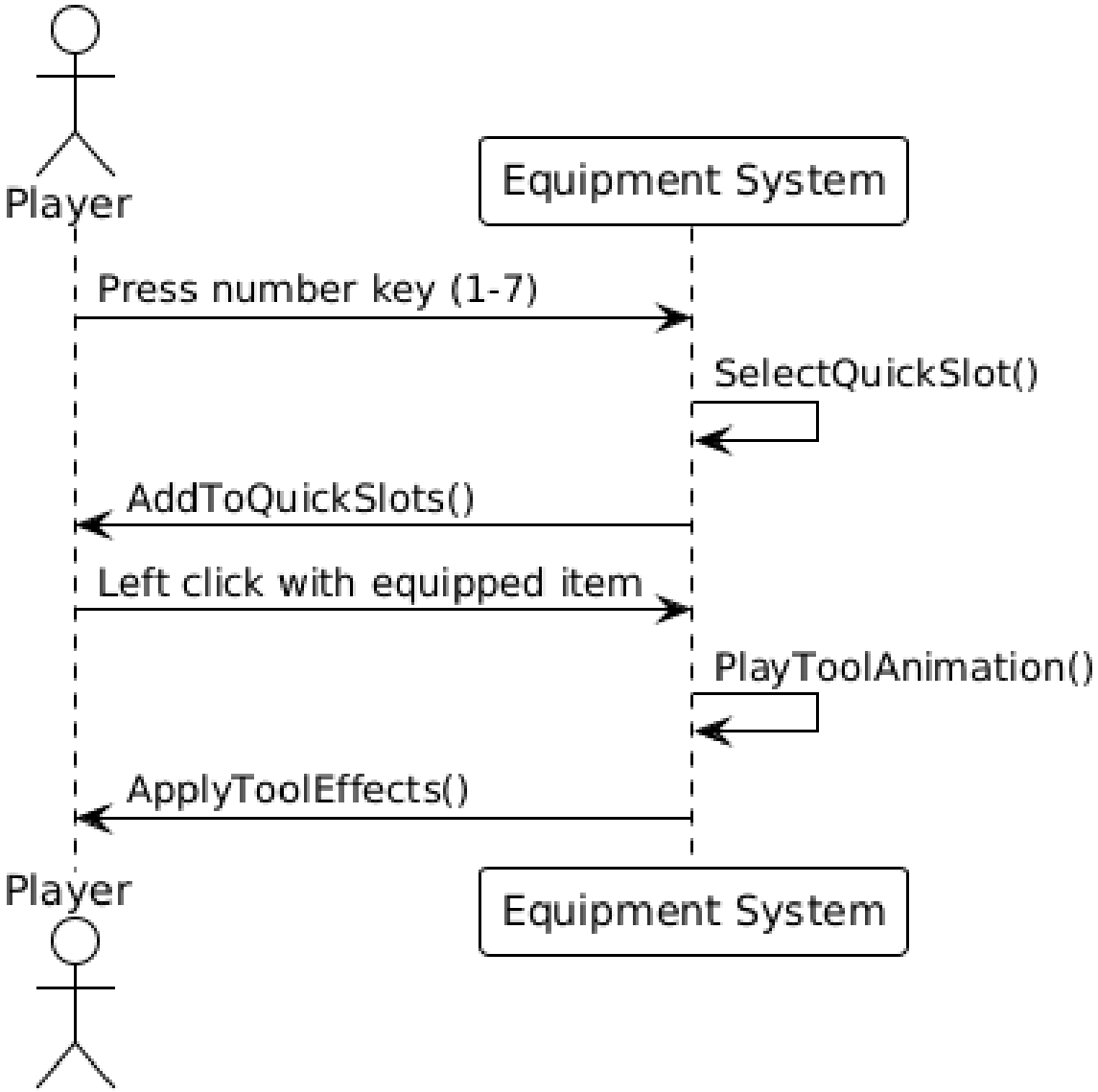

Figure 3.24. System Sequence Diagram - Equipment

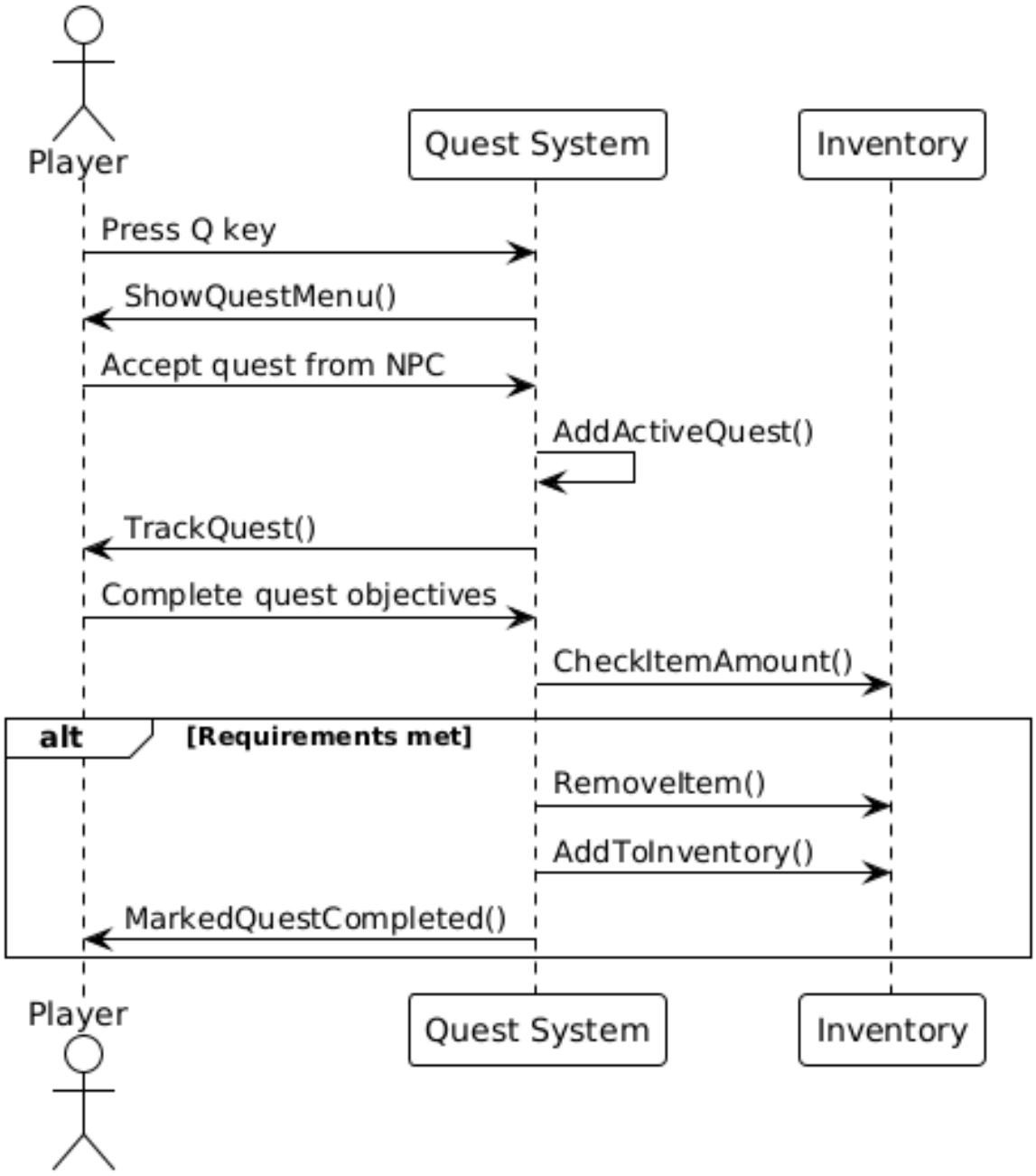

Figure 3.25. System Sequence Diagram - Quest

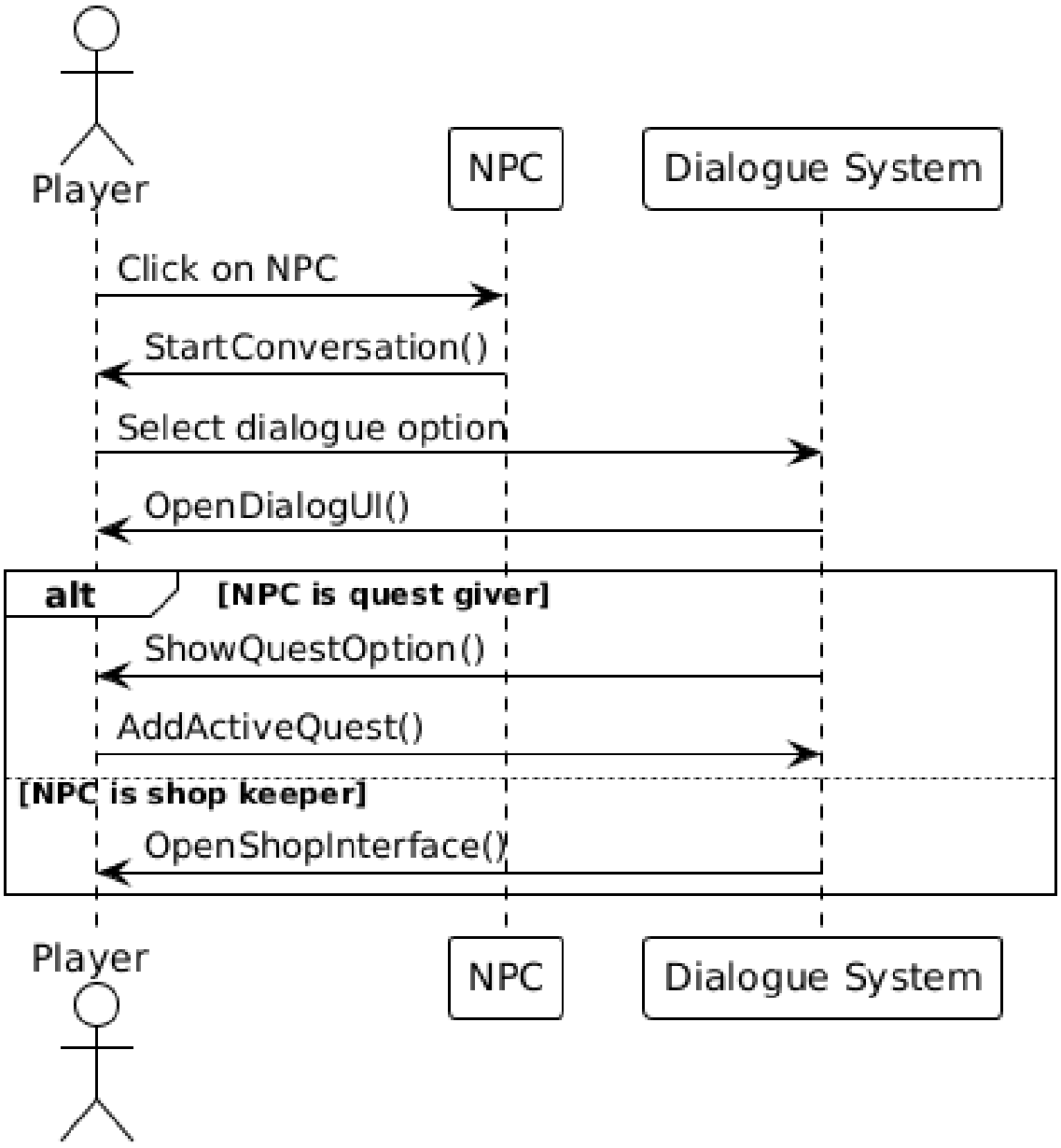

Figure 3.26. System Sequence Diagram - NPC Interaction

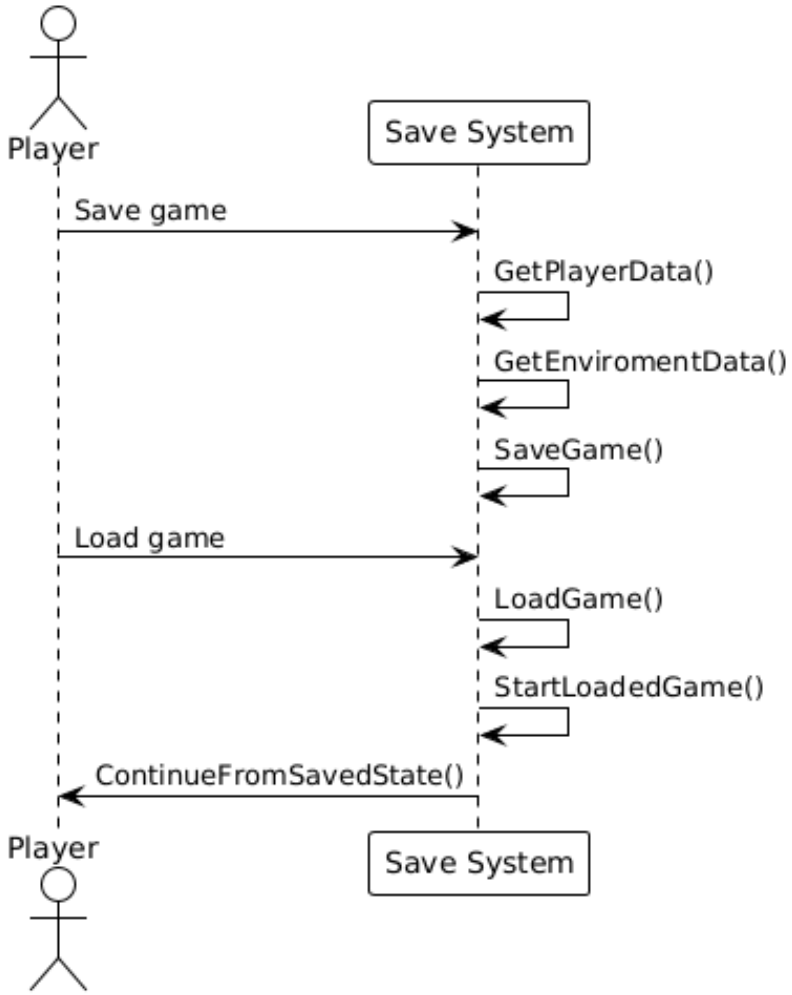

Figure 3.27. System Sequence Diagram - Save Load

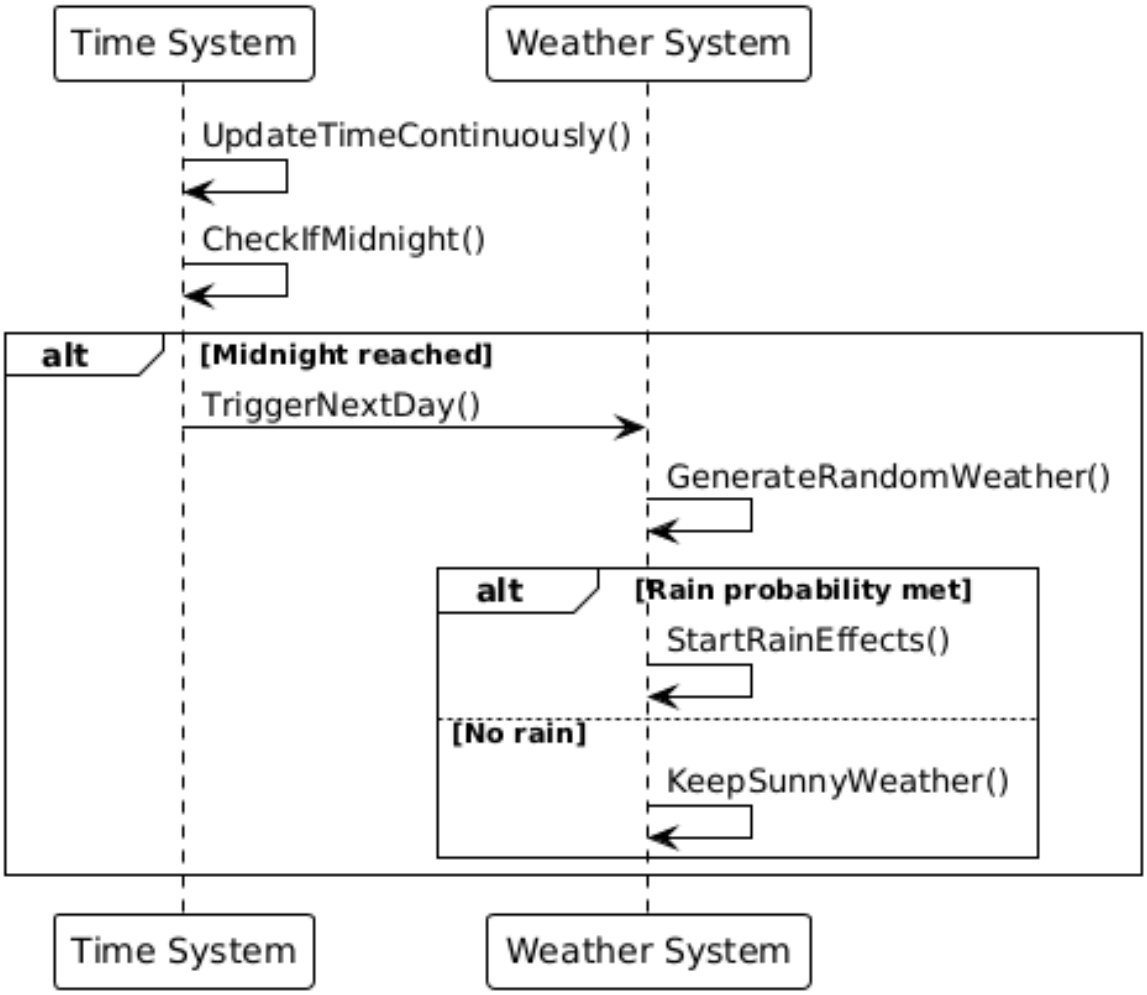

Figure 3.28. System Sequence Diagram - Weather

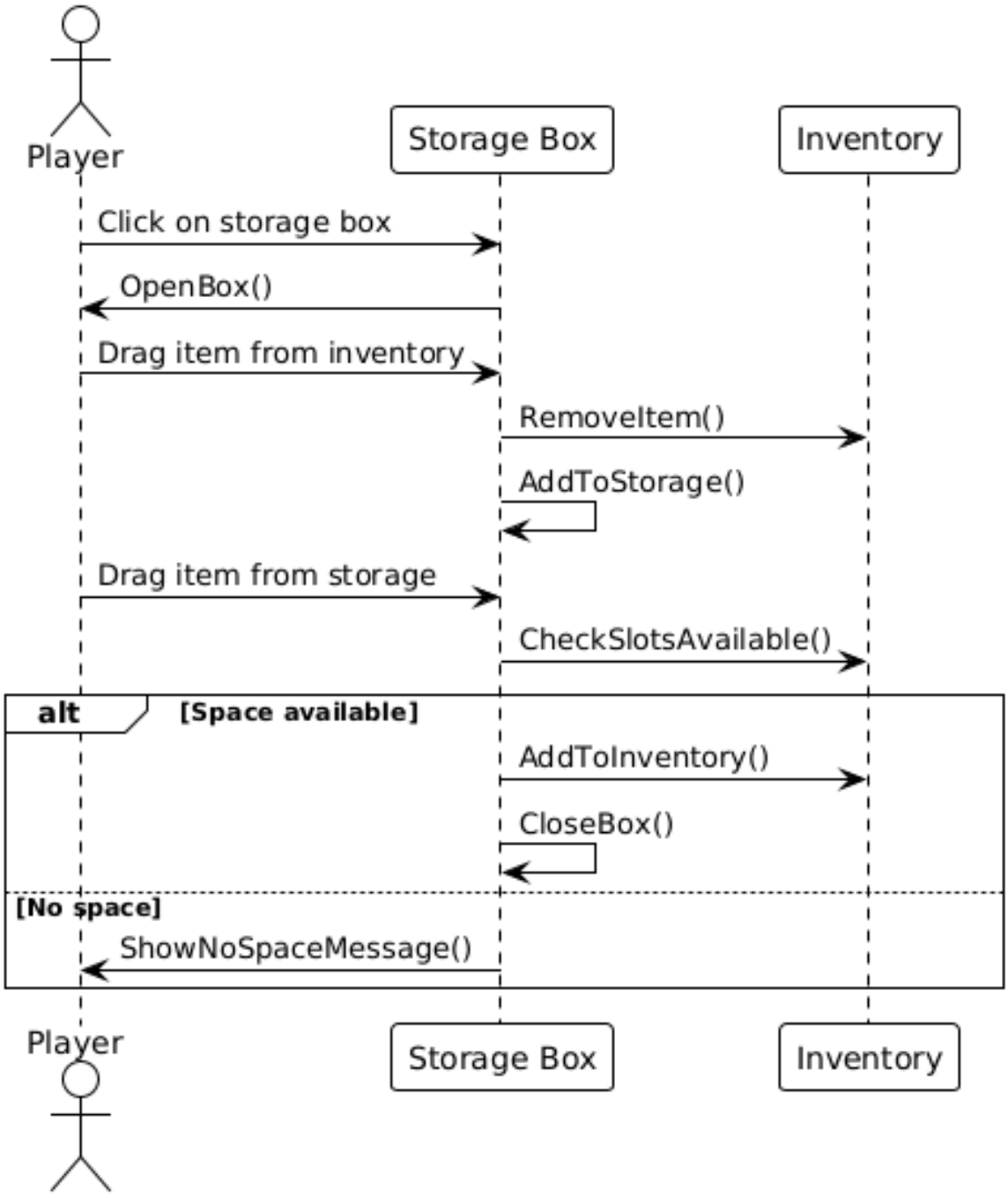

Figure 3.29. System Sequence Diagram - Storage

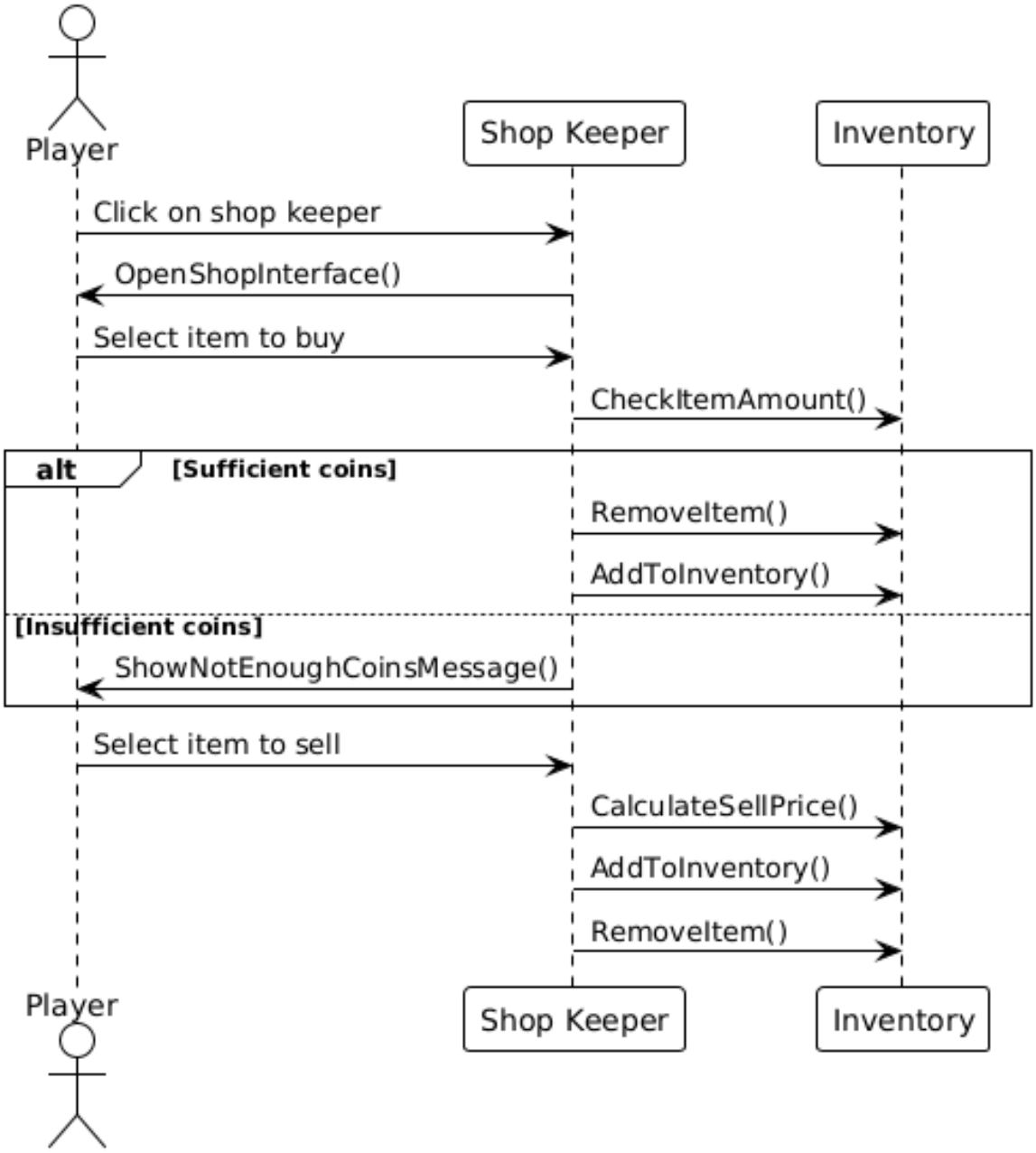

Figure 3.30. System Sequence Diagram - Shop

3.8 Complete System Activity Diagram

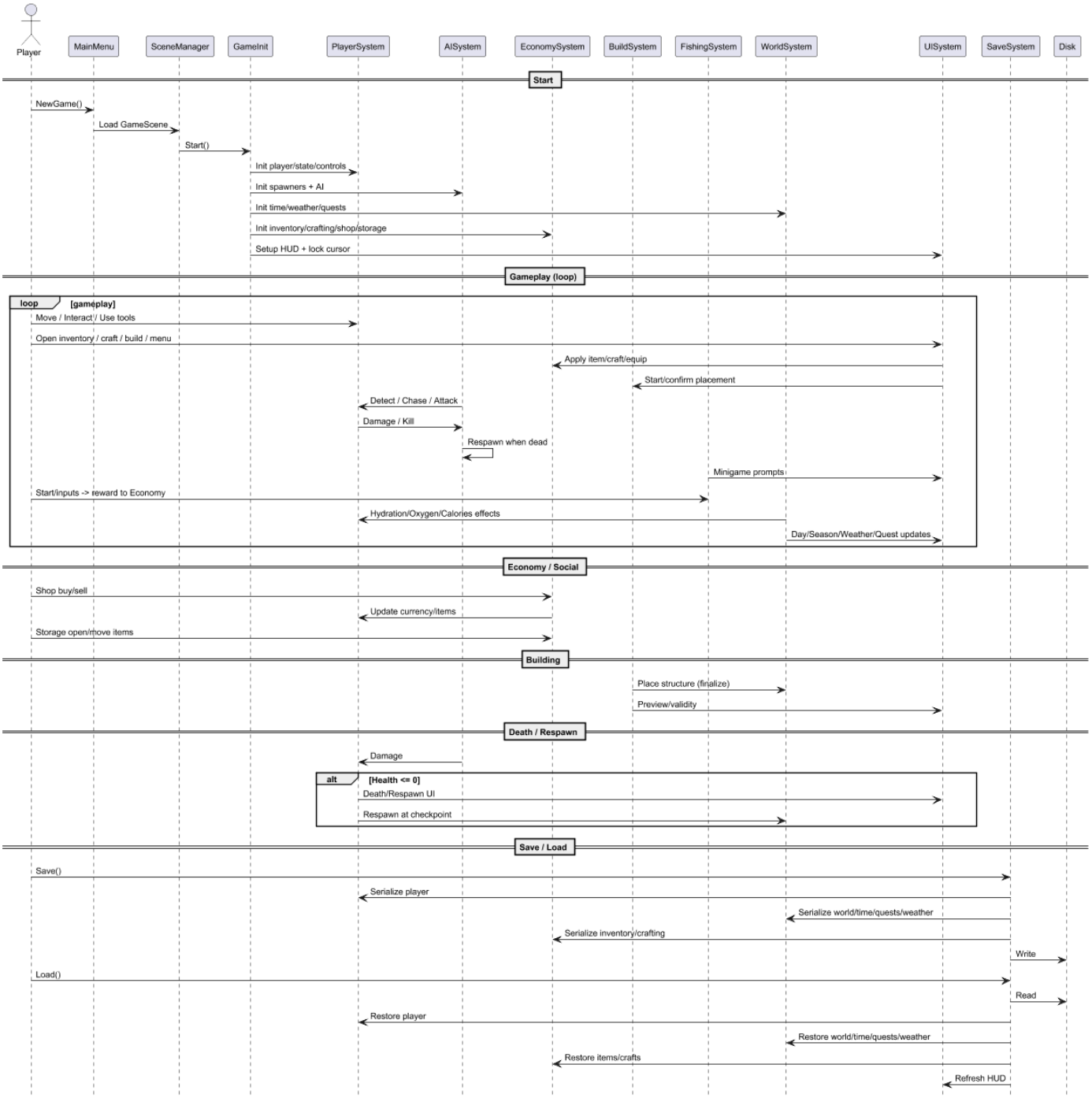

Figure 3.31. Complete System Activity Diagram

3.9 Activity Diagram

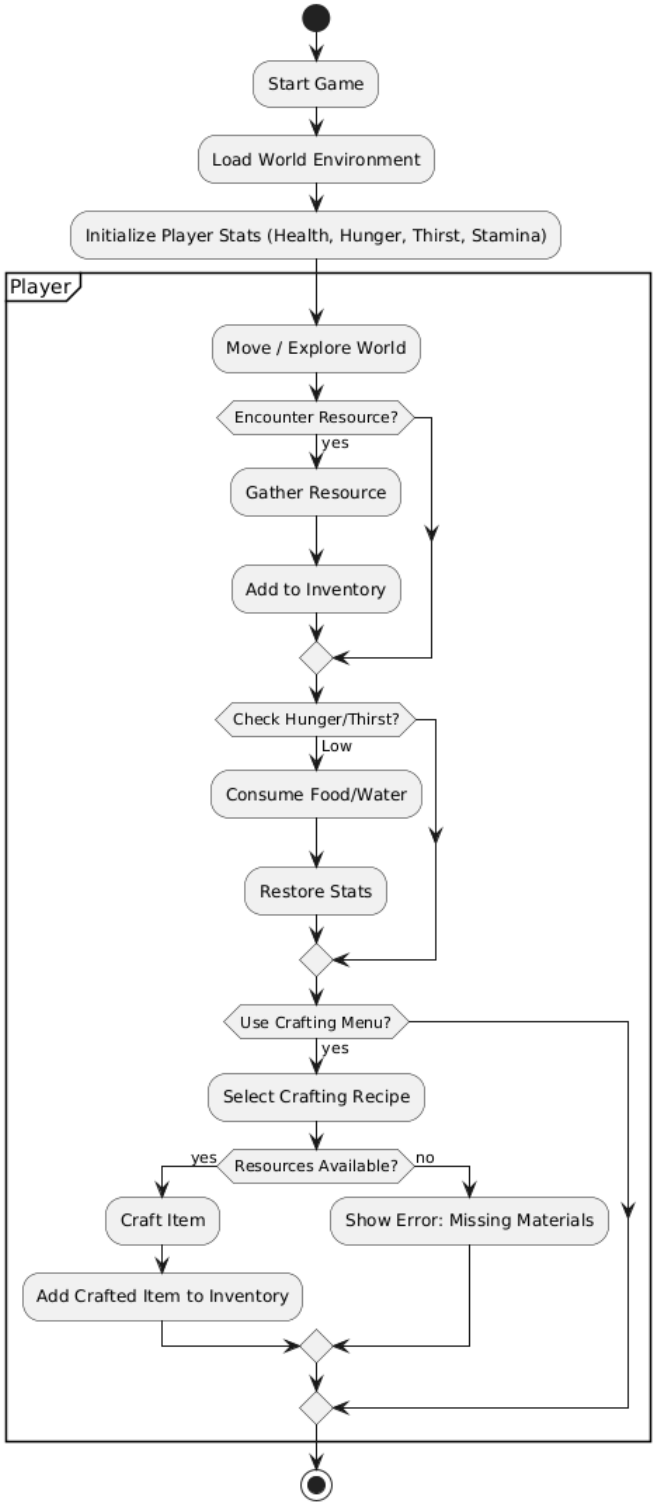

Figure 3.32. Activity Diagram of Last Ember - Part 1

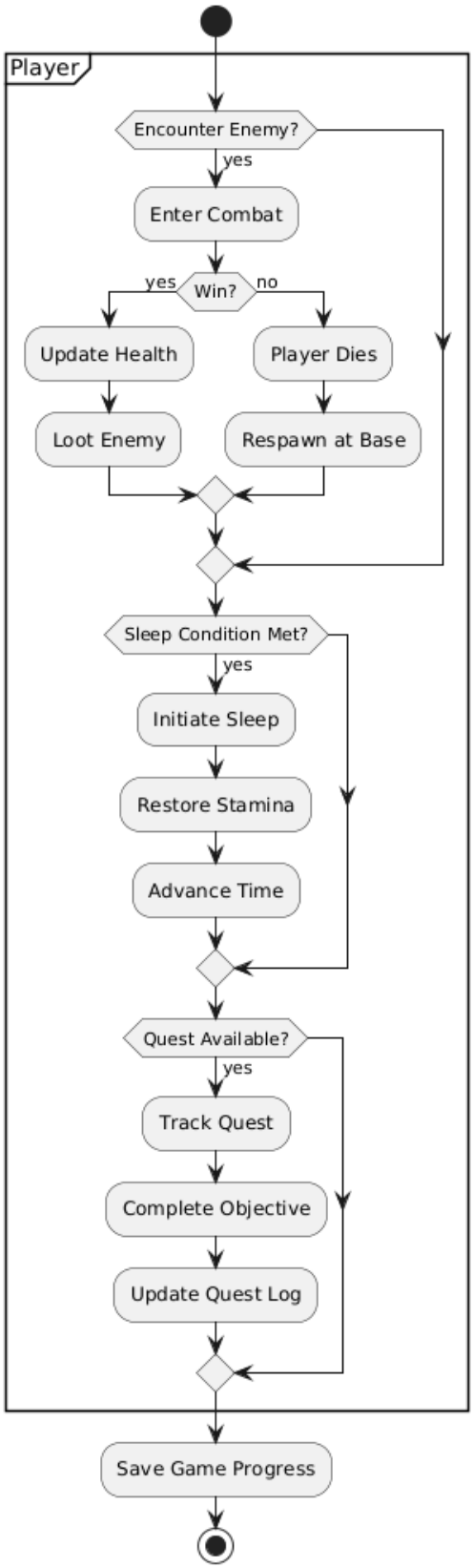

Figure 3.33. Activity Diagram of Last Ember - Part 2

Chapter 4

Design and Methodology

System design is the phase that involves the organization of the architecture, modules, and components of the system according to the requirements specified. It serves as a guide to follow since it converts the needs of the user into a structured model. The stage plays an important role in the Software Development Life Cycle (SDLC) because it removes the barrier between requirements and development.

This chapter explains how we have designed our 3D survival game in Unity and how the various game components are organized and how they interrelate with one another. The design encompasses the layout of game objects, scripts, and subsystems like Player Controller, Environment, Inventory System, resource management, Enemy AI, and User Interface.

In order to describe this structure, we draw a hierarchy diagram that indicates how different objects and components are arranged in Unity. This diagram helps to visualize the relationships between parents and children and between such elements as the player, the terrain, the game camera, the lighting, the environment assets, and the UI components.

Modularity, scalability, and optimization of performance are provided by the system design to guarantee easy playability and less complex updates to the system in the future. With industry standards, the design ensures that the game is friendly and has immersive gameplay, yet is efficient.

4.1 System Architecture

Last Ember is organized as follows:

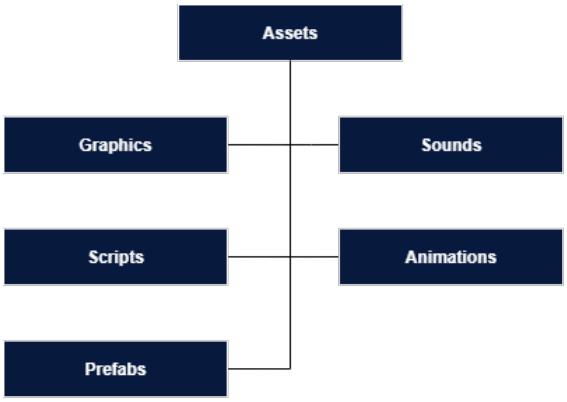

Figure 4.1. Basic project hierarchy

- **Project Folder:** This is the parent folder of the game, which has the subfolders. Graphics, Sounds, Scripts, and Animations, and Materials.
- **Graphics:** The folder will include the graphics used in the game, e.g., background, characters, enemies, food, weapons, etc.
- **Sounds:** The folder contains the sounds of the game, like walking, cutting trees, picking up items, etc.
- **Scripts:** The folder has the source code script files that are used in the game.
- **Animations:** The folder consists of the component files that produce the movement of the character and the movement constraints.
- **Prefab:** The prefab folder stores a GameObject complete with all its components, property values, and child GameObjects as a reusable asset [12].

Once the game is activated and running, the sound will be activated instantly for the game, and the player will have the option to start a new game, continue a saved game, or quit playing the game. When the player clicks on the exit, the application will close. Otherwise, the game will start.

The initial functions that will be activated include the following:

- **Show map:** It enables the gamer to see the map and move around as the character.
- **Handle collision (playing):** The player may stand on the ground or hit objects.
- **Show defeat:** The player loses if he is starved or thirsty or his health is low.
- **Show inventory:** Picked-up items are displayed in inventory and in quick slots.

- **Show character details:** The user is able to see health, food, and water in the status bar.
- **Craft items:** The player has access to collect items like sticks, wood and stone to create weapons, shelter, etc.
- **Shop system:** The gamer is able to sell or purchase goods at the village shop.

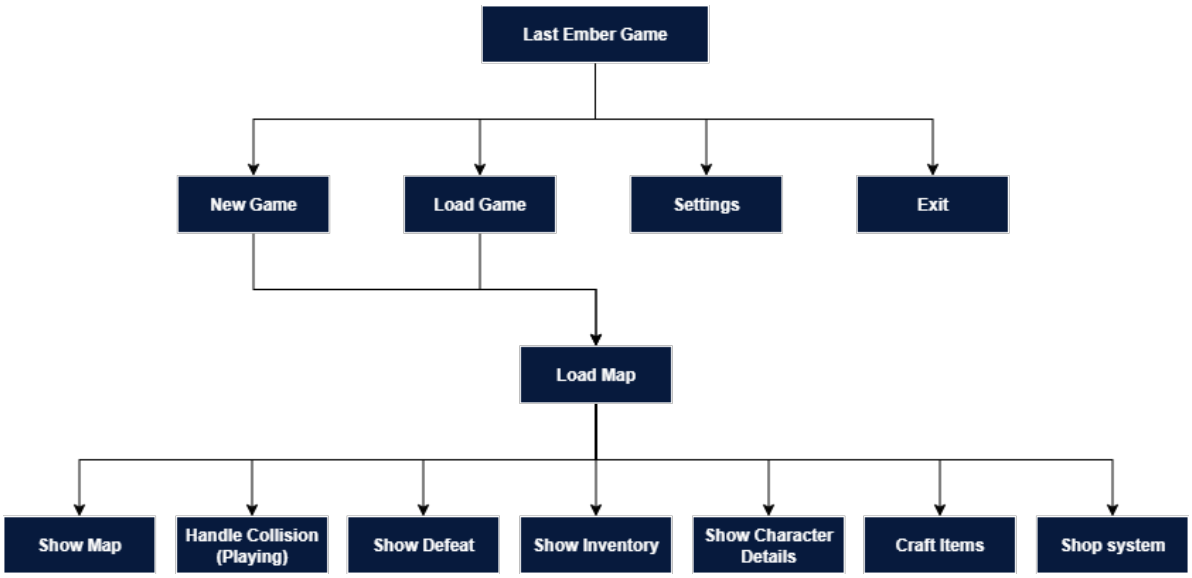

Figure 4.2. Hierarchical diagram of game functions.

4.2 Data design

Game development data design is the process of structuring, organizing, and controlling all the data that defines the functionality, content, and player experience of a game. It provides a link between the design ideas behind the game and how they are implemented technically, and it keeps and ensures that the information stored and needed is stored and remains accessible efficiently, at scale, and in a maintainable way.

Properly designed data structures enable developers and designers to maintain complicated systems within the game, like player progress, levels, objects, enemies, and interactions, without recreating fundamental mechanisms. Separating game data and the underlying code thus makes the game easier to modify, extend, and balance in development.

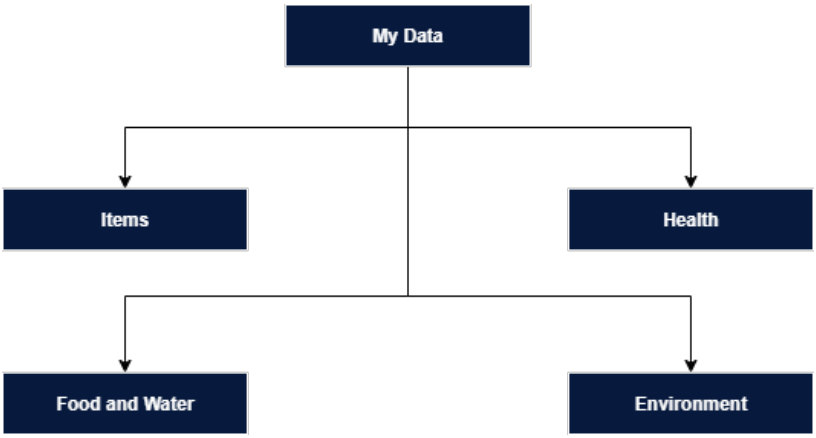

Figure 4.3. Game Data hierarchy

All the game-related data is stored in MyData. It structures the information in both the static design that defines the game, and the dynamic state which evolves as the game proceeds. The key branches are:

- **Items:** This branch holds all the item definitions present in the game (e.g., item ID, item name, and item attributes, like damage, weight or nourishment). These definitions are used by the gameplay systems during the spawning of items or showing item details. Items in this list are referred to by their IDs as player inventories and loot tables.
- **Player State:** This branch keeps track of the player during his or her run time.
  - **Health:** The health of the player is the present and maximum health and any modifier or effects that affect the health.
  - **Vitals (Food and Water):** The survival needs of the player, which are hunger and thirst (or other meters). These values wear out, and are replenished through the consumption of the right items in inventory.
- **Environment:** This branch is the information regarding the configuration and conditions of the game world, including the biome settings, weather profiles, and the parameters of the day/night cycle. It determines the appearance and behavior of the world and has the ability to manipulate the systems of a game, such as visibility, movement, and availability of resources.

This hierarchy provides a clean separation of the data that never changes (Items) and the data that changes each gameplay (Player State and World State) and the configuration that sets the overall experience is provided in the environment.

4.3 Storage design

- **Save Game Function:** This enables the current progress of the players to be recorded hence they do not lose data when they start the next gaming session.

- **Load Game Function:** Recalls a game that has been saved before and allows the player to restart the game at the exact spot and condition he/she had saved.

Three game data saving and loading techniques are applied in this project, including: Using PlayerPref, Json, Binary.

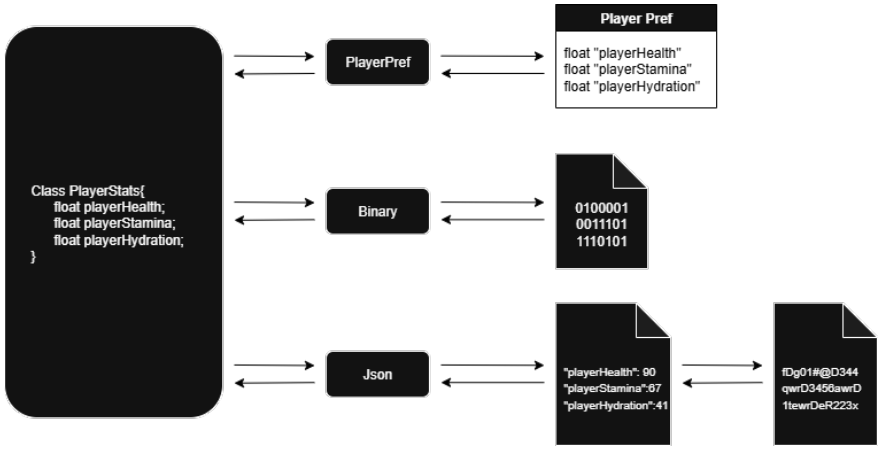

Figure 4.4. Methods for save and load game

- **PlayerPrefs:** Simple and simple to operate, should be used in storing small key-value information like game settings. However, it does not provide much security and is only suitable with the basic information.
- **JSON:** It is easy to edit, understandable by humans and flexible enough to represent a complex structure such as an object. On a negative note, it performs worse and is also prone to tampering.
- **Binary:** It is the most optimized and secure as well as efficient when it comes to handling a lot of data. It is however hard to read and edit on its own.
- **Combinations:** A hybrid solution is possible. PlayerPrefs simple data, JSON-structured and easily handled information, and Binary big data that needs to be fast and secure.

4.4 Design Constraints

Design constraints refer to constraints, rules, or conditions related to the design of any given system, and they will be very important to consider in our Last Ember game.

- **Screen Responsiveness:** PC is the target platform of our game, so our game will have to support various screen resolutions and aspect ratios. The game should be able to scale to the different monitor sizes.
- **Scalability:** When our game becomes bigger, it must be in a position to add more levels, features, and content that can retain current players and attract new players.

One of the major constraints is the design being scalable in the first place so that an expansion in the future will be easily joined with no interfering with the performance or stability.

- **Content Quality:** The content in our game should be of a certain quality in order to provide the player with an entertaining and enjoyable experience. This limitation affects the design and development cycle, where there must be consistency in the level design, visual assets, audio and overall gameplay to ensure there is immersion and that players are satisfied with it.

4.5 Design Methodology

Last Ember was developed using an iterative and incremental approach to software development. This method was selected to handle the complexity of the open world environment by dividing the project into manageable cycles, each of which are referred to as an iteration. Figure 4.5 indicates that every iteration was a full software development cycle, which included a basis, design, development, and test period.

The system was developed in functional stages as opposed to attempting all of it at once. The initial cycles were dedicated to founding the foundation of mechanics, like the movement of players, and terrain creation, followed by cyclic additions of more complicated systems (AI behaviour, inventory control, quest-tracking) to the game. The major strength of this methodology was that it was flexible. It enabled the development team to evaluate the performance of the game and playability on completion of each cycle. As a result, it was possible to add, edit, or delete functionalities directly through testing feedback that ensured the end product was optimized to the mid-range hardware and provides a balance user experience.

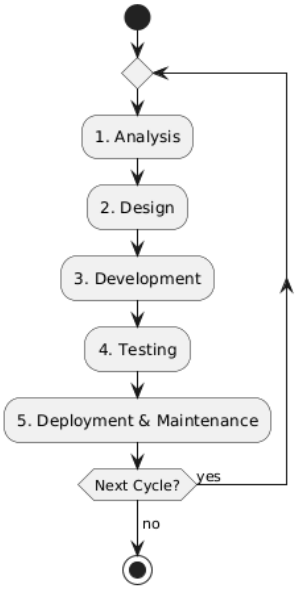

Figure 4.5. Design Methodology

4.6 GUI Design

- **Main Menu Screen:** The Main Menu of the game is the place where the adventure begins: there is the option to start a new game or a saved game or to change the settings of the game.

Figure 4.6. Main Menu Screen

- **Load Game Screen:** The Load Game screen shows all the progress your have saved and indicates the date and time when you saved the file. This will enable you to jump so easily back into your game and resume at the point you paused.

Figure 4.7. Load Game Screen

- **Game Settings Screen:** The Settings menu allows you to change the master volume, music, and sound effects so that you can change the audio of the game to your taste.

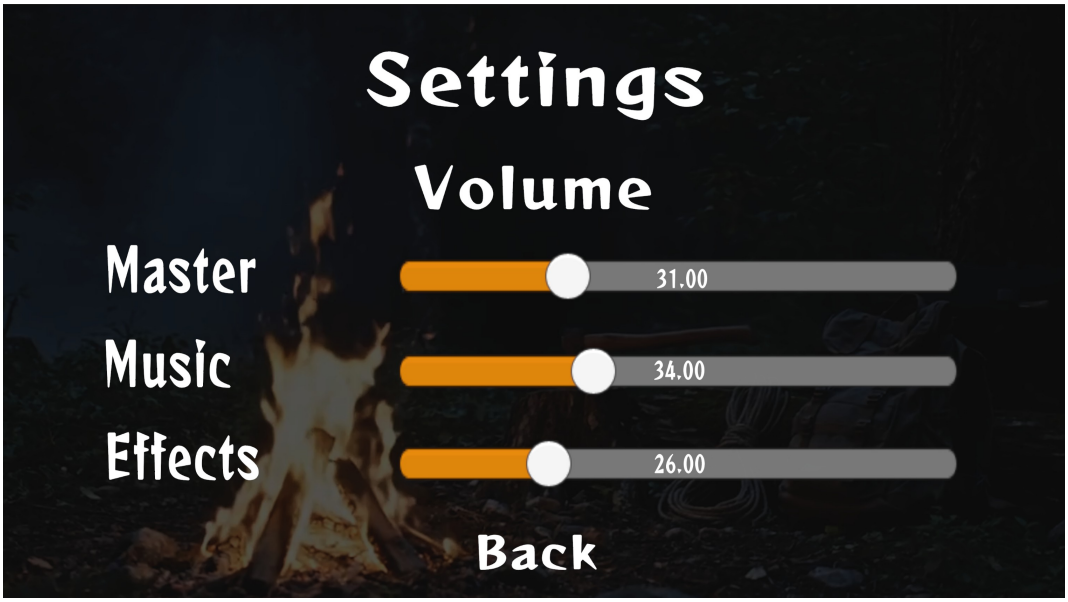

Figure 4.8. Game Settings Screen

- **Inventory Interface:** You may see and manage all of the items you have collected in the Inventory. You are able to see what you have and have an idea of the number of coins you have.

Figure 4.9. Inventory interface

- **Crafting Interface:** The crafting interface allows the ability to make new objects using what you have in your inventory. You are able to search recipes in various categories, including Tools, Basic Survival, and Construction, in order to make new gear and structures.

Figure 4.10. Crafting interface

- **Construction Material Crafting Interface:** The Construction Material Crafting interface gives one a special area to construct buildings. It displays to you the detailed resources needed by each building element, e.g., a foundation or a wall, so that you can easily plan building.

Figure 4.11. Construction Material Crafting interface

- **Tools Crafting Interface:** The Tools menu, which is accessed through the crafting interface, enumerates the tools that you can make and the materials required to make them. It also includes recipes of some of the basic survival supplies, like an axe or a torch, and many more, so you can live and hunt more effectively.

Figure 4.12. Tools Crafting interface

- **Quick Slots Interface:** Quick slots are slots at the bottom of the screen that have numbers that can be used to access your most used slots instantly. With tools or resources, you can access them after each other without the need to open your inventory.

Figure 4.13. Quick Slots interface

- **Campfire Interface:** One of the survival functions is the campfire that enables you cook food. The food and fuel can be inserted in the specific holes to make cooked meals.

Figure 4.14. Campfire interface

- **Quest Interface:** By clicking on this screen, one starts to talk with a Non-Player Character (NPC) to obtain or advance a quest. It presents a conversation and leaves you a choice to go on, which engages you into the story and activities of the game.

Figure 4.15. Quest interface

- **Quest Tracker Interface:** Quest Tracker assists in controlling your active missions, displaying your ongoing quests and their goals along with any possible rewards. It reminds you of what you are doing and gives you the broad view of your progress.

Figure 4.16. Quest Tracker Interface

- **Player Info HUD interface:** This heads up display provides critical details of the status of your character. It shows health (red), stamina (blue), a third stat and the instant date and time of the game.

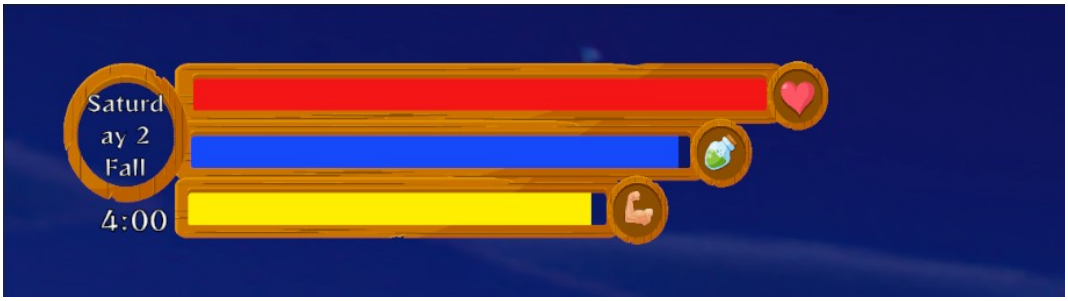

Figure 4.17. Player Info HUD interface

Chapter 5

System Implementation

System implementation consists of all that follows a planned system or software concept in turning it into a reality by writing the code and delivering the parts that allow it to run.

5.1 Technical Architecture

A three tier architecture is described in our LastEmber survival game group. We created the app using the Unity Engine and coded in C# and used a component-based design. Native systems of Unity are involved with rendering, physics, audio and asset management. To get a graphical picture, refer to the figure below.

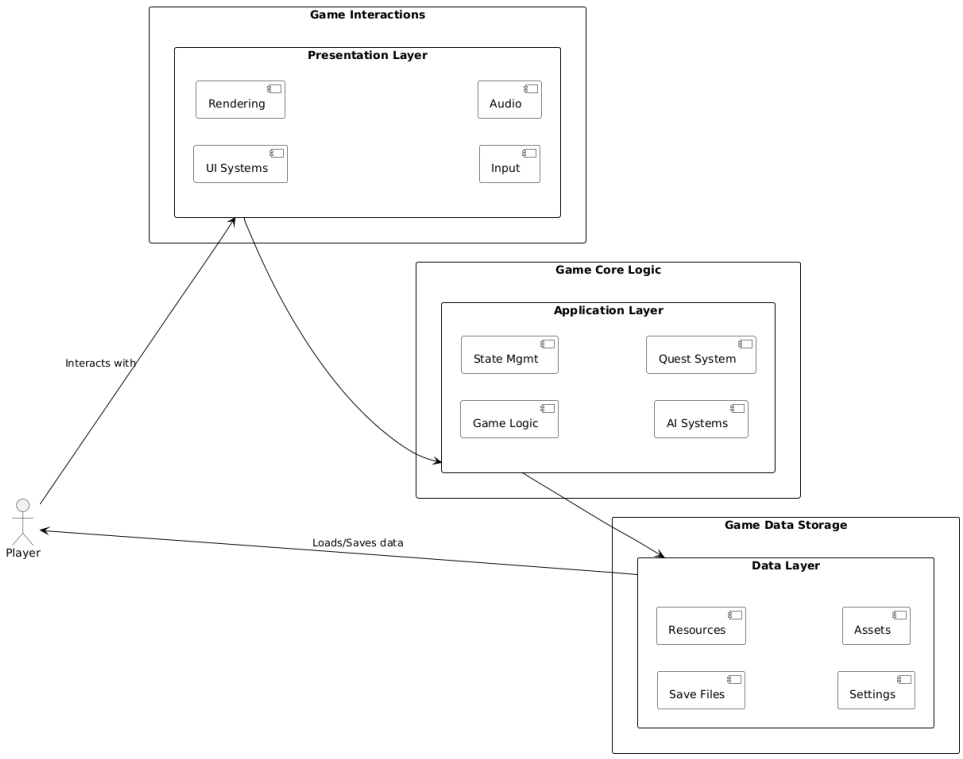

Figure 5.1. High-Level System Architecture Diagram

5.2 System Internal Components

Our LastEmber survival game consists of various internal components, including:

Table 5.1. LastEmber Game System Components

System Category	Key Components / Description
Game Management Systems	Core game logic, state management, and initialization.
Player Systems	Player movement, health, hunger, hydration, and inventory management.
Environment Systems	Terrain, weather, day/night cycle, and resource management.
AI Systems	Animal behavior, NPC interactions, and enemy AI.
Quest Systems	Quest tracking, completion logic, and reward distribution.
Crafting Systems	Item creation, recipe management, and resource processing.
Save/Load System	Game state persistence and data management.
Sound Systems	Sound effects, music, and audio management.
UI Systems	Menu management, inventory interface, and user interactions.

5.3 Functionality of the Components

- **Game Management Systems:** The GameInitializer, then, is more or less the primary thing that establishes the entire game, including player state and ensuring that all the systems are co-ordinated. It performs multiple verifications that all of it has been properly defined initially and relies on the singleton pattern such that everybody can simply get its instance in any other part of the project.
- **Player Systems:** The PlayerState deals with the player statistics such as health, calories and hydration. Movement of the character is managed by PlayerMovement which also controls the camera. The InventorySystem deals with storage and piling of items and speedy-slot
- **Environment Systems:** The world state is handled by the EnvironmentManager, which deals with objects like trees, animals, and objects placed by the user. Environmental conditions are managed by the WeatherSystem. DayNightSystem is a time and lighting flow management.

- **AI System:** Animal AI uses state machines of idle behavior, walk behavior, chase behavior, and attack behavior. Conversation with NPCs and the assigning of quests takes place in NPC systems. Animal AI controls the behavioral pattern in wildlife.
- **Quest System:** QuestManager follows quests, both in progress and achieved, and quest goals, as well as rewarding. It offers quest tracking and completion in real-time.
- **Crafting System:** The CraftingSystem takes control of recipes of items, resource demands and the creation of items. It interconnects with the inventory system to use materials and manufacture of crafted items.
- **Save/Load System:** The SaveManager is, thus, simply the tool that processes the saving of the state of the game, both with the help of both binary and JSON files. It manages all the player data, the environment configuration, and the progress on a number of various save slots.

5.4 Tools and Technologies

Tools and Technologies used in our LastEmber survival game are shown below:

Table 5.2. Project Tools and Technology Stack with Core Technologies

Name	Version	Rationale
Tools		
Unity Engine	2022.3.58f1	Game Development Platform
Visual Studio	2022	Code Editor and IDE
Overleaf	–	Documentation
Canva	–	Presentation
Github	–	Version Control
Core Technologies		
C# Language	–	Programming Language

5.4.1 Explanation

This subsection explains the rationale for selecting the above tools and technologies.

- **Unity Engine:** The core platform that we use to develop the game is Unity Engine 2022.3.58f1. The Unity provides us with a set of tools we need to develop 3D game development, including rendering, physics, audio, and asset management systems [14].
- **Visual Studio:** Visual Studio 2022 has been utilized by us as the integrated development environment in C programming. It offers advance level of debugging, and effortless integration with Unity [15].

- **C#:** C# is the main programming language that we have used. C# has the capabilities of object-oriented programming, garbage collection and it has very high performance in game development [15].
- **Overleaf:** Overleaf has been our main tool of writing down all the documentation. It is an online LaTeX editor which allows us to work together in real time and write our reports like professionals. It does all the citations, maintains everything in a tidy manner, and makes working in a team extremely easy [16].
- **Canva:** Our presentations were done on Canva. It is a web-based designing tool, which allows us to create visually attractive slides, and the customization features made it simple to compile our project presentations effectively [17].
- **GitHub:** We have deployed our project on GitHub as a collaboration platform and version control system. It made us effective with Git repositories in order to manage our codebase, track changes, and work as a team [18].

Chapter 6

System Testing

Table 6.1. Test Case: Player Movement and Navigation

Test Case ID	TC-1.1
Related Use Case	UC-1.1 Player Movement and Navigation
Pre-Conditions	Game running, player spawned, MovementManager and CharacterController initialized
Test Steps	1. Launch game and spawn player. 2. Press WASD keys individually and in combination. 3. Move mouse left/right and up/down. 4. Press Space while grounded.
Expected Result	1. Player moves smoothly in all directions. 2. Camera rotates according to mouse movement. 3. Player jumps only when grounded. 4. Movement and jump disabled when UI open.
Actual Result	Player movement and camera rotation worked smoothly in all directions. Jumping only occurred when grounded. No movement when UI was open. All controls responsive and accurate.
Status	Pass

Table 6.2. Test Case: Item Collection and Pickup

Test Case ID	TC-1.2
Related Use Case	UC-1.2 Item Collection and Pickup
Pre-Conditions	Player spawned, interactable items present in environment
Test Steps	1. Move player near an item. 2. Press interaction key (E). 3. Attempt pickup when inventory full.
Expected Result	1. Item detected and highlighted. 2. Item added to inventory when collected. 3. Error message shown if inventory full.
Actual Result	Items were correctly detected and highlighted when the player approached. Pressing the interaction key successfully added items to the inventory. When the inventory was full, an appropriate error message appeared, and the item remained in the environment without being collected
Status	Pass

Table 6.3. Test Case: Inventory Management

Test Case ID	TC-1.3
Related Use Case	UC-1.3 Inventory Management
Pre-Conditions	Player has items collected
Test Steps	1. Open inventory menu. 2. Move items between slots. 3. Equip and unequip items. 4. Drop item from inventory.
Expected Result	1. Inventory opens correctly. 2. Items can be rearranged. 3. Equipped items function properly. 4. Dropped items appear in world.
Actual Result	Inventory opened without delay. Items rearranged and equipped successfully. Dropped items appeared in the world and could be recollected. All interactions stable.
Status	Pass

Table 6.4. Test Case: Crafting System

Test Case ID	TC-1.4
Related Use Case	UC-1.4 Crafting System
Pre-Conditions	Crafting UI enabled, player has required materials
Test Steps	1. Open crafting menu. 2. Select a valid recipe. 3. Attempt crafting with missing materials. 4. Attempt crafting with full inventory.
Expected Result	1. Crafting UI opens. 2. Crafting succeeds when requirements met. 3. Error message shown when requirements not met. 4. Error message shown when inventory full.
Actual Result	Crafting UI displayed correctly. Valid recipes produced expected items. Missing materials and full inventory triggered proper warning messages. No glitches observed.
Status	Pass

Table 6.5. Test Case: Base Building System

Test Case ID	TC-1.5
Related Use Case	UC-1.5 Base Building System
Pre-Conditions	Player has building materials, building mode enabled
Test Steps	1. Enter building mode. 2. Place a wall on valid ground. 3. Attempt placement on invalid terrain. 4. Attach multiple pieces together.
Expected Result	1. Building mode activates correctly. 2. Structures snap and place on valid terrain. 3. Error feedback when terrain invalid. 4. Structures connect seamlessly.
Actual Result	Building mode activated smoothly, allowing accurate placement of structures on valid terrain. Walls and other pieces snapped correctly into place, with clear error feedback shown when attempting to build on invalid terrain. Connected structures aligned seamlessly without gaps or clipping.
Status	Pass

Table 6.6. Test Case: Combat System (Melee and Ranged)

Test Case ID	TC-1.6
Related Use Case	UC-1.6 Combat System (Melee and Ranged)
Pre-Conditions	Player equipped with weapon, enemies present
Test Steps	1. Attack enemy with melee weapon. 2. Kill enemy.
Expected Result	1. Damage applied to enemies. 2. Enemy dies and drops loot.
Actual Result	Melee attacks successfully registered on enemies, applying appropriate damage values. Enemy health decreased as expected, and upon death, enemies dropped loot items correctly. Combat animations and hit feedback worked smoothly.
Status	Pass

Table 6.7. Test Case: Fishing System

Test Case ID	TC-1.7
Related Use Case	UC-1.7 Fishing System
Pre-Conditions	Player has fishing rod, near water source
Test Steps	1. Equip fishing rod. 2. Cast line into water. 3. Wait for bite notification. 4. Complete fishing minigame. 5. Attempt fishing with no bait.
Expected Result	1. Fishing animation and cast succeed. 2. Bite notification appears when fish detected. 3. Successful minigame rewards fish. 4. Failed minigame results in lost fish. 5. Fishing fails when no bait equipped.
Actual Result	Fishing rod equipped correctly. Line cast animation and minigame functioned as expected. Bite detection worked. Fishing failed properly without bait. All results matched expectations.
Status	Pass

Table 6.8. Test Case: Farming System

Test Case ID	TC-1.8
Related Use Case	UC-1.8 Farming System
Pre-Conditions	Player has seeds, tools, farm plot
Test Steps	1. Till soil with tool. 2. Plant seeds in farm plot. 3. Water crops daily. 4. Attempt growth in different seasons. 5. Harvest when ready.
Expected Result	1. Soil properly prepared. 2. Seeds planted successfully. 3. Crops grow when watered and season allows. 4. Crops fail when season unsuitable. 5. Harvest adds produce to inventory.
Actual Result	Soil was tilled and prepared correctly, seeds planted without issue, and crops grew steadily when watered in suitable seasons. Crops failed to grow or withered during unsuitable seasons. Harvesting worked as intended, adding produce to the player's inventory.
Status	Pass

Table 6.9. Test Case: Wildlife AI System

Test Case ID	TC-1.9
Related Use Case	UC-1.9 Wildlife AI System
Pre-Conditions	Wildlife spawned in environment
Test Steps	1. Observe neutral wildlife. 2. Attack aggressive wildlife.
Expected Result	1. Neutral wildlife ignores player until provoked. 2. Aggressive animals attack when approached.
Actual Result	Neutral wildlife remained passive until attacked, aggressive animals engaged when the player approached too closely
Status	Pass

Table 6.10. Test Case: Enemy AI System (Nighttime Foes)

Test Case ID	TC-1.10
Related Use Case	UC-1.10 Enemy AI System (Nighttime Foes)
Pre-Conditions	Game time is night, enemies spawned
Test Steps	1. Wait until night cycle begins. 2. Encounter enemy waves. 3. Observe AI pathfinding and attacks. 4. Defeat or evade enemies.
Expected Result	1. Enemies spawn during night. 2. AI navigates terrain using NavMesh. 3. Enemies attack base/player intelligently. 4. Player can defeat or evade them.
Actual Result	Enemies spawned correctly at night, navigated terrain smoothly using NavMesh, avoided obstacles, and coordinated attacks on the player and base. Player was able to defeat or evade enemies successfully.
Status	Pass

Table 6.11. Test Case: Environmental Effects (Weather and Seasons)

Test Case ID	TC-1.11
Related Use Case	UC-1.11 Environmental Effects (Weather and Seasons)
Pre-Conditions	Dynamic weather and season system enabled
Test Steps	1. Wait for rain event. 2. Observe season transition.
Expected Result	1. Seasonal transitions affect farming and resources. 2. Player must adapt survival strategy.
Actual Result	Rain event successfully triggers, affecting soil moisture and crop growth. Season transitions smoothly.
Status	Pass

Table 6.12. Test Case: Save and Load Game

Test Case ID	TC-1.12
Related Use Case	UC-1.12 Save and Load Game
Pre-Conditions	Game session active, player data available, SaveManager functional
Test Steps	1. Start a game and make progress (collect items, move position, build structure). 2. Open pause menu and select "Save Game". 3. Confirm save completion message. 4. Exit to main menu. 5. Select "Load Game" and choose the same save slot. 6. Verify that progress, inventory, and environment are restored.
Expected Result	1. Save file correctly stores game state. 2. Loading restores player stats, inventory, and environment. 3. Corrupted or missing save files trigger error message gracefully.
Actual Result	Game state saved successfully. Player position, inventory, and structures restored accurately after reloading. No data loss observed.
Status	Pass

Table 6.13. Test Case: Weather and Day-Night Cycle

Test Case ID	TC-1.13
Related Use Case	UC-1.13 Weather and Day-Night Cycle
Pre-Conditions	Environment loaded, DayNightSystem and WeatherSystem initialized
Test Steps	1. Start game and observe time progression. 2. Note lighting and skybox transitions throughout the day. 3. Wait until midnight and observe next day trigger. 4. Enable or fast-forward time to trigger rain or fog. 5. Observe seasonal weather variations.
Expected Result	1. Day-night transitions occur smoothly. 2. Lighting and shadows adjust correctly with time. 3. Weather events (rain) trigger based on probabilities. 4. Environment reacts dynamically to seasonal or weather changes.
Actual Result	Time progressed smoothly with correct lighting and skybox transitions. Midnight triggered next day automatically. Random rain effects displayed as expected.
Status	Pass

Table 6.14. Test Case: Storage Box Management

Test Case ID	TC-1.14
Related Use Case	UC-1.14 Storage Box Management
Pre-Conditions	Player near a placed storage box, StorageManager active
Test Steps	1. Approach a storage box and interact with it. 2. Open the storage interface. 3. Transfer items from inventory to storage. 4. Transfer items back to player inventory. 5. Attempt to exceed storage capacity. 6. Close and reopen storage box.
Expected Result	1. Storage UI opens and displays correct contents. 2. Item transfers succeed if space is available. 3. Storage prevents exceeding maximum capacity. 4. Stored items persist after closing and reopening.
Actual Result	Storage UI opened correctly. Item transfers between inventories worked properly. System enforced capacity limits and saved stored items persistently.
Status	Pass

Table 6.15. Test Case: Shop and Trading System

Test Case ID	TC-1.15
Related Use Case	UC-1.15 Shop and Trading System
Pre-Conditions	Player near shopkeeper NPC with active trading interface and sufficient currency/items
Test Steps	1. Approach NPC and initiate trade. 2. Buy an item with enough coins. 3. Attempt to buy an item with insufficient coins. 4. Sell an item to the NPC. 5. Attempt to sell an unequippable or restricted item.
Expected Result	1. Trading UI opens and displays shop inventory. 2. Purchase succeeds when player has sufficient coins. 3. System prevents transaction when funds or space are insufficient. 4. Selling items increases player's coin count. 5. Validation prevents illegal or restricted sales.
Actual Result	Trading UI opened correctly. Purchases and sales worked as intended. Insufficient funds message displayed properly. No bugs found.
Status	Pass

Chapter 7

Conclusions

In the development and implementation of **Last Ember**, valuable insights and lessons were gained that extend beyond the technical aspects of the project. These lessons contribute not only to improving the design of survival games but also to advancing the understanding of player engagement, dynamic environments, and AI-driven interactions. The project delivered a survival experience that emphasizes adaptability, unpredictability, and immersion, while also highlighting best practices and challenges in creating optimized survival games for diverse hardware capabilities. As a result, **Last Ember** provides a strong foundation for future advancements in survival game design and open-world interactive systems.

7.1 Lessons Learned

7.1.1 Inclusive Design

The project highlighted the importance of creating an accessible survival experience optimized for mid-range systems. By ensuring efficient performance and compatibility, the game remains playable by a wider range of users.

7.1.2 Player-Centric Approach

Focusing on intuitive controls, responsive mechanics, and natural UI navigation proved essential. A user-centered design ensures that survival mechanics are engaging without overwhelming new players.

7.1.3 Security and Data Integrity

Protecting save data and ensuring reliable load mechanisms are critical. Regular testing of persistence systems prevents corruption and supports long-term engagement.

7.1.4 Scalability

The need for scalable systems became evident. From AI spawning logic to seasonal farming cycles, the infrastructure was designed to accommodate future expansions such as multiplayer or extended maps.

7.1.5 Performance Testing

Optimization, stress testing, and frame-rate monitoring are not one-time tasks but continuous processes. Ongoing performance evaluation ensures smooth gameplay, even with multiple AI agents and environmental changes active.

7.1.6 Dynamic Survival Systems

The integration of seasonal changes, adaptive AI, and resource scarcity reinforced the importance of flexible mechanics. Allowing players to adapt to evolving conditions increased replayability and immersion.

7.2 Future Directions

To further enhance **Last Ember**, the following steps can be considered:

7.2.1 Expanded Content

Introduce additional questlines, environments, and crafting recipes, along with interactive world events that provide long-term progression and narrative depth.

7.2.2 AI Enhancements

Explore advanced AI techniques, such as reinforcement learning or ML-Agents integration, to create even more unpredictable wildlife and enemy encounters.

7.2.3 Community and Multiplayer

Incorporate multiplayer co-op or community-driven features, enabling players to collaborate on base-building, resource management, and survival strategies.

7.2.4 Player Analytics and Balancing

Develop monitoring tools to analyze player behavior, helping to fine-tune difficulty scaling, balance resource availability, and improve progression pacing.

Appendix A

User Manual

A.1 Introduction

This user manual provides instructions for playing the LastEmber survival game. The manual covers basic controls, gameplay mechanics, and essential features to help players navigate and enjoy the game experience.

A.2 System Requirements

Before playing LastEmber, ensure your system meets the following requirements:

- **Operating System**: Windows 10 or later
- **Processor**: Intel Core i5 or equivalent
- **Memory**: 8 GB RAM minimum
- **Graphics**: DirectX 11 compatible graphics card
- **Storage**: 5 GB available space
- **Input**: Keyboard and Mouse

A.3 Game Controls

A.3.1 Basic Movement

The following controls are used for character movement and interaction:

- **W, A, S, D**: Move character forward, left, backward, and right
- **Space**: Jump
- **Left Shift** or **Right Shift**: Sprint (hold to run faster)

- **Mouse Movement**: Rotate camera view
- **Left Mouse Button**: Interact with objects, place items, attack
- **Right Mouse Button**: Cancel current action

A.3.2 Menu and System Controls

- **I**: Open/Close inventory
- **C**: Open/Close crafting menu
- **Tab**: Open/Close quest log
- **ESC**: Open pause menu
- **1-6**: Quick slot selection (equip items from quick slots)
- **R** or **Left Click**: Register respawn checkpoint
- + or =: Zoom in map
- **-**: Zoom out map

A.3.3 Placement and Building Controls

- **Left Click**: Place item when in placement/construction mode
- **X**: Cancel item placement or construction
- **Mouse Movement (Left/Right)**: Rotate items during placement (for campfire, storage boxes, and bed only)
- **Mouse Movement**: Position items during placement mode

A.4 Gameplay Mechanics

A.4.1 Survival Systems

The game features several survival mechanics that players must manage:

Health System

- Health decreases when taking damage from enemies or environmental hazards
- Health can be restored by consuming food items
- When health reaches zero, the player respawns at the last registered checkpoint

Calories System

- Calories decrease based on distance traveled and physical activities
- Consume food items to restore calories
- Low calories affect movement speed and health regeneration

Hydration System

- Hydration decreases automatically over time
- Drink water or consume hydrating items to restore hydration
- Low hydration causes health to decrease gradually

Oxygen System

- Oxygen is only relevant when underwater
- Oxygen decreases while submerged
- When oxygen depletes, the player takes damage
- Surface to restore oxygen levels

A.4.2 Inventory Management

Accessing Inventory

Press **I** to open the inventory system. The inventory displays:

- Grid-based item slots
- Item quantities and stack information
- Currency display (coins)
- Quick slots for tools and weapons (6 slots)
- Item pickup notifications

Item Management

- **Drag and Drop**: Move items between inventory slots
- **Left Click**: Select and interact with items
- **Right Click**: Use consumable items (food, water)
- **Quick Slots (1-6)**: Equip items to quick slots for easy access
- **Stack Limit**: Maximum of 3 items per stack
- Items can be equipped, consumed, or placed in the world

A.4.3 Crafting System

Accessing Crafting

Press **C** to open the crafting menu. The crafting system is organized into categories:

- **Tools**: Craft axes, torches, and other tools
- **Survival**: Basic survival items
- **Refine**: Process raw materials (e.g., Log → Plank)
- **Construction**: Building materials (foundations, walls, doors, roofs)

Crafting Process

1. Open crafting menu (**C**)
2. Select a category tab (Tools, Survival, Refine, Construction)
3. Select desired item from recipe list
4. View required materials and quantities
5. Ensure all required materials are in inventory
6. Click craft button to create item
7. Newly crafted item appears in inventory

Example Recipes

- **Axe**: 3 Stone + 3 Stick
- **Plank**: 1 Log (refined)
- **Foundation**: 4 Planks
- **Wall**: 2 Planks
- **Torch**: 2 Stone + 2 Stick

A.4.4 Building and Placement

The game features two building systems: **Placement System** for placeable items and **Construction System** for building structures.

Placement System (Items)

For items like campfires, storage boxes, and beds:

1. Select item from inventory
2. Item enters placement mode

3. Move mouse to position item in the world
4. Green highlight indicates valid placement location
5. Red highlight indicates invalid placement (overlapping or not grounded)
6. For campfire, storage boxes, and bed: Move mouse left/right to rotate
7. Left click to place item
8. Press **X** to cancel placement

Rotating Placeable Items

The following items support rotation during placement:

- Campfire
- Storage Box (Small)
- Storage Box (Big)
- Bed

To rotate: Move mouse left/right while in placement mode. The item rotates around the Y-axis. Rotation is saved when the item is placed.

Construction System (Structures)

For building structures like foundations, walls, doors, and torches:

1. Select construction item from inventory (Foundation, Wall, Door, Roof, Torch)
2. Enter construction mode
3. For **Foundations**: Free-style placement with validation
4. For **Walls** and **Doors**: Snap to ghost positions (preview markers)
5. For **Torches**: Attach to walls by aiming at wall surface
6. Green highlight indicates valid placement
7. Red highlight indicates invalid placement
8. Left click to place structure
9. Press **X** to cancel construction

Ghost System

When placing foundations, the game generates "ghost" markers that show:

- Where walls can be placed (snap points)
- Where roofs can be placed
- Valid connection points for structures

Ghosts appear as semi-transparent preview objects and help ensure proper structure placement.

A.4.5 Quest System

Accessing Quests

- Press **Tab** to open/close quest log
- Quests are received by talking to NPCs in the world
- Quest objectives are displayed in the quest log
- Active quests can be tracked in the quest tracker UI

Quest Types

The game features three types of quests:

- **Item Collection**: Gather a specific amount of items
- **Checkpoint Quests**: Visit specific locations
- **No Requirements**: Instant completion quests

Quest States

- **Not Started**: Quest available but not accepted
- **Active**: Quest accepted and in progress
- **Completed**: Quest finished, rewards given

Completing Quests

1. Talk to NPCs to receive quests
2. Accept or decline quest offers
3. Read quest objectives in quest log (**Tab**)
4. Complete required tasks (gather items, visit locations, etc.)
5. Progress is tracked automatically
6. Return to quest giver (NPC) when requirements are met
7. Receive rewards: Coins and items

A.4.6 Resource Gathering

Chopping Trees

1. Approach a tree
2. Equip an axe or tool
3. Left click to chop
4. Collect wood resources when tree falls

Mining Rocks

1. Approach a rock
2. Equip a pickaxe or tool
3. Left click to mine
4. Collect stone resources

A.4.7 Fishing System

Starting Fishing

1. Equip a fishing rod (from inventory or quick slot)
2. Approach a water body (Lake, River, or Ocean)
3. Cast line into water using left click
4. Wait approximately 3 seconds for a fish to bite
5. System determines fish type based on water source and probability
6. Complete fishing minigame when fish bites
7. Successfully caught fish is added to inventory

Fishing Tips

- Different water sources (Lake, River, Ocean) have different fish types
- Each fish type has a probability value affecting catch chance
- Fishing rods may have durability that decreases with use

A.5 Game Features

A.5.1 Storage System

Storage Boxes

Players can place and use storage boxes to store items:

- Two types: Small Box and Big Box
- Craft or place storage boxes in the world
- Approach storage box and interact (left click) to open
- Drag and drop items between inventory and storage
- Storage contents are saved with the game
- Each storage box maintains its own item list

Using Storage

1. Place a storage box (small or big) in the world
2. Approach the storage box (within 10 units)
3. Left click to interact and open storage UI
4. Drag items from inventory to storage slots
5. Drag items from storage to inventory
6. Close storage UI to continue playing

A.5.2 Campfire Cooking System

Using Campfires

Players can cook food using campfires:

1. Place a campfire in the world (can be rotated during placement)
2. Approach the campfire (within 10 units)
3. Left click to interact and open campfire UI
4. Place fuel in fuel slot (Log or Stick)
5. Place raw food in food slot (e.g., Raw Meat)
6. Click cook button when both slots are filled
7. Wait for cooking timer to complete
8. Return to campfire to collect cooked food

Cooking Details

- Valid fuels: Log, Stick
- Valid foods: Raw Meat (cooks into Cooked Meat)
- Cooking time varies by food type (e.g., Raw Meat takes 10 seconds)
- Fire visual effect appears during cooking
- Cooked food appears in campfire food slot when ready

A.5.3 Trading System

Buying Items

Players can buy items from shopkeepers:

1. Approach a shopkeeper NPC

2. Interact (left click) to open dialog
3. Click "Buy" button to enter buy mode
4. Browse available items and prices
5. Click buy button on desired items (requires sufficient coins)
6. Purchased items are added to inventory

Selling Items

Players can sell items to shopkeepers:

1. Approach a shopkeeper NPC
2. Interact (left click) to open dialog
3. Click "Sell" button to enter sell mode
4. Drag items from inventory to sell slots
5. View total sell value
6. Click sell button to complete transaction
7. Receive coins based on item selling prices

A.5.4 Day and Night Cycle

The game features a dynamic day and night cycle:

- Time progresses automatically (24 game hours = configurable real-time seconds)
- Skybox changes based on time of day
- Lighting adjusts dynamically
- Day counter increments at midnight
- Time is displayed in the game UI

A.5.5 Weather System

Weather conditions affect gameplay:

- Weather changes dynamically
- Some weather conditions may affect visibility
- Weather impacts certain game mechanics

A.5.6 Respawn and Checkpoints

Registering Checkpoints

1. Approach a respawn checkpoint location
2. Press **R** or **Left Click** to register checkpoint
3. Checkpoint is saved as respawn location

Respawn System

- When health reaches zero, player respawns at last registered checkpoint
- Player position, rotation, and stats are restored
- Inventory and equipment are preserved

A.5.7 Saving and Loading

Save System

The game supports two save formats:

- **JSON Format**: Human-readable text file
- **Binary Format**: Encrypted, smaller file size (recommended)

What Gets Saved

The save system preserves:

- **Player Data**:
  - Health, Calories, Hydration, Oxygen
  - Player position and rotation
  - All inventory items
  - Quick slot items
  - Currency (coins)
- **Environment Data**:
  - Trees (chopped/standing state)
  - Animals (which ones exist)
  - Storage boxes and their contents
  - Placed items (campfires, beds, etc.)
  - Built structures (foundations, walls, doors, torches)
  - Picked up items

Saving Game

- Game progress can be saved manually through the menu
- Multiple save slots available
- Save files are encrypted to prevent tampering

Loading Game

1. Open main menu
2. Select "Load Game"
3. Choose desired save slot
4. Confirm to load saved game
5. All saved data is restored to previous state

A.6 Tips for New Players

1. **Manage Survival Stats**: Keep track of health, calories, hydration, and oxygen. Consume food and water regularly to maintain stats.
2. **Gather Resources Early**: Collect wood (from trees), stone (from rocks), and other resources as soon as possible. These are essential for crafting.
3. **Craft Essential Tools**: Start by crafting an Axe (3 Stone + 3 Stick) to efficiently gather wood. Tools make resource gathering much faster.
4. **Build a Base**: Create foundations and walls to build a shelter. Place storage boxes inside to store items safely.
5. **Use Storage Boxes**: Don't carry everything in your inventory. Use storage boxes to organize and store excess items.
6. **Cook Food**: Use campfires to cook raw meat into cooked meat, which provides better nutrition than raw food.
7. **Complete Quests**: Talk to NPCs and complete quests to earn coins and items. Quests provide valuable rewards.
8. **Trade Wisely**: Sell excess items to shopkeepers for coins, then buy items you need. Manage your currency carefully.
9. **Register Checkpoints**: Press R or Left Click at respawn points to save your progress location. This ensures you respawn at a safe location.
10. **Use Quick Slots**: Equip frequently used items (tools, weapons) to quick slots (1-6) for instant access.
11. **Plan Your Buildings**: Use the ghost system when building. Place foundations first, then walls snap to ghost positions automatically.

12. **Explore Carefully**: Be aware of enemies and environmental hazards. Keep health items ready for emergencies.
13. **Save Regularly**: Use the save system to avoid losing progress. Multiple save slots allow you to experiment safely.
14. **Refine Materials**: Use the Refine category in crafting to process raw materials (e.g., Logs into Planks) for building.

References

[1] V. H. Tran and T. K. G. Hoang. Build a 3d survival game. In *Proceedings of the Vietnam-Korea University of ICT*, 2025. Project Report.

[2] G. N. Yannakakis and J. Togelius. *Artificial Intelligence and Games*. Springer, 2018.

[3] M. T. Boyce and M. D. Evans. Using mathematical models in game design: A survival mechanics case. *ResearchGate*.

[4] Game Design Skills. Survival game design (principles, examples, template).

[5] M. G. Olsson. The relationship between survival mechanics and emergent narrative, 2021.

[6] Endnight Games. The forest, 2014. [Video game] Endnight Games Ltd.

[7] Facepunch Studios. Rust, 2013. [Video game] Facepunch Studios.

[8] Redbeet Interactive. Raft, 2018. [Video game] Redbeet Interactive.

[9] Studio Wildcard. Ark: Survival evolved, 2015. [Video game] Studio Wildcard.

[10] S. L. Olughu and O. B. L. Adejumo. Design and implementation of an intelligent gaming agent using a* algorithm and finite state machines. *International Journal of Engineering Research and Technology (IJERT)*, 2020.

[11] J. Everitt. Ai - state machines. Technical report, Research Newcastle University.

[12] Unity Technologies. Unity documentation: Prefabs.

[13] R. Nystrom. Object pool. Game Programming Patterns, 2014.

[14] Unity Technologies. Unity 2022.3.58f1 release notes.

[15] Microsoft. https://code.visualstudio.com/docs/languages/csharp.

[16] Overleaf. https://www.overleaf.com/about/why-latex.

[17] Canva. https://www.canva.com/presentations/.

[18] GitHub. https://docs.github.com/en/get-started/start-your-journey/about-github-and-git.

Last Ember - A 3D Open-World Survival with

ORIGINALITY REPORT

8%	6%	2%	4%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	Submitted to Higher Education Commission Pakistan Student Paper	1%
2	repository.cuilahore.edu.pk Internet Source	1%
3	hal.laas.fr Internet Source	1%
4	Submitted to University of Greenwich Student Paper	1%
5	Submitted to Middlesex University Student Paper	<1%
6	umpir.ump.edu.my Internet Source	<1%
7	Submitted to Universitas Pamulang Student Paper	<1%
8	jbitdoon.edu.in Internet Source	<1%
9	Submitted to HELP UNIVERSITY Student Paper	<1%
10	Submitted to Informatics Education Limited	

	Student Paper	<1%
11	Oakman, Rachel. "Implementation and Evaluation of a CRNA-Led Preoperative Evaluation Checklist to Reduce Cancellations in Adult Elective Surgery at an Ambulatory Surgery Center", Wilmington University (Delaware) Publication	<1%
12	Submitted to University of Bristol Student Paper	<1%
13	Submitted to University of London External System Student Paper	<1%
14	Submitted to University of Southampton Student Paper	<1%
15	drmc.library.adelaide.edu.au Internet Source	<1%
16	Submitted to Ryerson University Student Paper	<1%
17	Submitted to University of Sousse Student Paper	<1%
18	Submitted to University of Limerick Student Paper	<1%
19	Submitted to University of Warwick Student Paper	<1%

20	Deyneha, Mykyta. "Smart Shopping in Retail Using Beacon Technology", Universidade NOVA de Lisboa (Portugal), 2025 Publication	<1 %
21	Submitted to University of Birmingham Student Paper	<1 %
22	Submitted to NCC Education Student Paper	<1 %
23	Submitted to University of Rijeka Student Paper	<1 %
24	scholar.ppu.edu Internet Source	<1 %
25	www.igi-global.com Internet Source	<1 %
26	competitiveanalytics.com Internet Source	<1 %
27	riuma.uma.es Internet Source	<1 %
28	community.chocolatey.org Internet Source	<1 %
29	www.coursehero.com Internet Source	<1 %
30	www.studymode.com Internet Source	<1 %
31	core.ac.uk Internet Source	<1 %

32	www.cs.cmu.edu Internet Source	<1%

Exclude quotes	Off	Exclude matches	Off
Exclude bibliography	Off		

*% detected as AI

AI detection includes the possibility of false positives. Although some text in this submission is likely AI generated, scores below the 20% threshold are not surfaced because they have a higher likelihood of false positives.

Caution: Review required.

It is essential to understand the limitations of AI detection before making decisions about a student's work. We encourage you to learn more about Turnitin's AI detection capabilities before using the tool.

Frequently Asked Questions

How should I interpret Turnitin's AI writing percentage and false positives?
The percentage shown in the AI writing report is the amount of qualifying text within the submission that Turnitin's AI writing detection model determines was either likely AI-generated text from a large-language model or likely AI-generated text that was likely revised using an AI paraphrase tool or word spinner.

False positives (incorrectly flagging human-written text as AI-generated) are a possibility in AI models.

AI detection scores under 20%, which we do not surface in new reports, have a higher likelihood of false positives. To reduce the likelihood of misinterpretation, no score or highlights are attributed and are indicated with an asterisk in the report (*%).

The AI writing percentage should not be the sole basis to determine whether misconduct has occurred. The reviewer/instructor should use the percentage as a means to start a formative conversation with their student and/or use it to examine the submitted assignment in accordance with their school's policies.

What does 'qualifying text' mean?
Our model only processes qualifying text in the form of long-form writing. Long-form writing means individual sentences contained in paragraphs that make up a longer piece of written work, such as an essay, a dissertation, or an article, etc. Qualifying text that has been determined to be likely AI-generated will be highlighted in cyan in the submission, and likely AI-generated and then likely AI-paraphrased will be highlighted purple.

Non-qualifying text, such as bullet points, annotated bibliographies, etc., will not be processed and can create disparity between the submission highlights and the percentage shown.