

The Beaver Game

System Design Document

Version 1.0
2024-02-29

CAN Game Studio

Contents

1 Introduction	4
1.1 Purpose of the SDD	4
2 General Overview and Design Guidelines/Approach	4
2.1 General Overview	4
2.2 Assumptions/Constraints/Risks	4
2.2.1 Assumptions	4
2.2.2 Constraints	5
2.2.3 Risks	5
2.3 Alignment with an Enterprise Architecture Framework	5
3 Design Considerations	6
3.1 Goals and Guidelines	6
3.2 Development Methods and Contingencies	6
3.3 Architectural Strategies	8
3.4 Performance Engineering	9
4 System Architecture and Architecture Design	9
4.1 Logical View	9
4.2 Hardware Architecture	9
4.2.1 Security Hardware Architecture	10
4.2.2 Performance Hardware Architecture	10
4.3 Software Architecture	10
4.3.1 Security Software Architecture	13
4.3.2 Performance Software Architecture	13
4.4 Information Architecture	14
4.4.1 Records Management	14
4.5 Internal Communications Architecture	15
4.6 Security Architecture	15
4.7 Performance	15
4.8 System Architecture Diagram	15
5 System Design	15
5.1 Business Requirements	15
5.2 Database Design	15
5.2.1 Data Objects and Resultant Data Structures	15
5.2.2 File and Database Structures	16
5.3 Data Conversion	16
5.4 User Machine-Readable Interface	16
5.4.1 Inputs	16
5.4.2 Outputs	17
5.5 User Interface Design	17
5.5.1 Section 508 Compliance	17

6 Operational Scenarios	17
7 Detailed Design	20
7.1 Hardware Detailed Design	20
7.2 Software Detailed Design	20
7.3 Security Detailed Design	28
7.4 Performance Detailed Design	28
7.5 Internal Communications Detailed Design	28
8 System Integrity Controls	28
9 External Interfaces	28
9.1 Interface Architecture	28
9.2 Interface Detailed Design	28
A Record of Changes	29
B Acronyms	30
C Glossary	31
D Referenced Documents	32
E Approvals	33
F Additional Appendices	34
F.1 Software Architecture Diagrams	34
F.2 Requirements Traceability Matrix	39
F.3 Data Dictionary	40
G Database Design Document	41
G.1 Assumptions/Constraints/Risks	41
G.1.1 Assumptions	41
G.1.2 Constraint	41
G.1.3 Risks	41
G.2 Design Decisions	41
G.2.1 Key Factors Influencing Design	41
G.2.2 Functional Design Decisions	41
G.2.3 Database Management System Decisions	41
G.2.4 Security and Privacy Design Decisions	41
G.2.5 Performance and Maintenance Design Decisions	41
G.3 Data Software Objects and Resultant Data Structures	41
G.3.1 Database Management System Files	41
G.4 Roles and Responsibilities	42
G.4.1 System Information	42
G.4.2 Database Management System Configuration	42
G.4.3 Database Support Software	42
G.4.4 Security and Privacy	42

G.5 Performance Monitoring and Database Efficiency	42
G.5.1 Operational Implications	42
G.5.2 Data Transfer Requirements	42
G.5.3 Data Formats	42
G.5.4 Backup and Recovery	42
H Interface Control Document	43
H.1 Interface Overview	43
H.2 Interface Controls	43
H.3 Functional Allocation	43
H.3.1 Data Transfer	43
H.3.2 Transactions	43
H.3.3 Security and Integrity	43
H.4 Detailed Interface Requirements	43
H.4.1 Requirements for [Given Interface Name]	43
H.4.2 Assumptions	43
H.4.3 Technical Interface Requirements	43
H.4.4 General Processing Steps	43
H.4.5 Interface Processing Time Requirements	43
H.4.6 Message Format (or Record Layout) and Required Protocols	44
H.4.7 Security Requirements	44
H.4.8 Requirements for "Given Interface Name"	44
H.5 Quality Assurance	44

List of Figures

1 Class Diagram	34
2 DFD Layer 0	35
3 DFD Layer 1: Structure Upgrade Process	36
4 DFD Layer 1: Combat Process	37
5 System Architecture Diagram	38

List of Tables

1 Record of Changes	29
2 Acronyms	30
3 Glossary	31
4 Referenced Documents	32
5 Approvals	33
6 Requirements Traceability Matrix	39

1 Introduction

This System Design Document (SDD) will define the technical design for the Beaver Game.

The Beaver Game adds an interesting twist to the established pixel simulation RPG video game genre. Playing as a beaver, the user will harvest trees, build dams, face predators, and more while watching the health of the game environment come to life.

1.1 Purpose of the SDD

This document serves as a comprehensive guide to the architectural and design needs of this project, serving as the blueprint and communication tool needed to stay organized. Another purpose that this document has is to be a reference to mitigate development complications.

2 General Overview and Design Guidelines/Approach

This section describes the principles and strategies to be used as guidelines when designing and implementing the system.

2.1 General Overview

The Beaver game is a 2D top-down pixel RPG video game. The system utilizes the Godot game engine, a free and open-source game engine with extensive 2D game development tools. The system will be developed using Godot version 4.2.1, the latest stable release as of December 12th, 2023.

2.2 Assumptions/Constraints/Risks

2.2.1 Assumptions

This project assumes that there exists an end-user base interested in a game of this nature. It also assumes that said users have access to the required computing devices to play the game and a medium level of technical literacy and confidence with simple gameplay mechanics. The latter assumptions are reasonable to expectations from a user base with a previous interest in RGP video games.

This project assumes to have access to the specified hardware outlined in the document, including RAM, the CPU, and the GPU, as the game's performance is dependent on the users' hardware specifications. Variations of the hardware can impact the quality of the graphics and the gameplay experience.

This project assumes compatibility with specific operations systems as indicated in system requirements. The game's performance is dependent on the operating system and changes in a user system may affect functionality. If this occurs updates to the game may be required.

This game assumes compatibility with a specified version of the Godot game engine. Changes in the engine may affect game behavior and require an update or patch to ensure compatibility.

This game assumes that users will have a certain level of technical proficiency. The user interface will be designed for users with varying skill levels to provide clear instructions and support.

2.2.2 Constraints

This section will describe the constraints and limitations that must be adhered to during development.

- Hardware or software environment: This will be optimized to run smoothly on the outlined specifications. The design will focus on efficiency and compatibility to ensure optimal performance.
- End-user environment: Adherence to Godot outlined specifications mentioned in Software Requirements Specification. The user interface and overall user experience will be constructed to meet the expectations and capabilities defined in the Godot specifications.
- Availability or volatility of resources: The continued availability of Godot and the Aseprite plugin. Regular updates will be necessary to ensure the code remains functional after Godot updates.
- Interoperability requirements: The game will be on PC only, running on Windows and MacOS. The system will be designed to operate on Windows and MacOS to ensure cross-platform compatibility as outlined in the requirements.
- Data repository and distribution requirements: References in Section 4.4.
- Memory or other capacity limitations: Ensure the game does not exceed the memory limitations of targeted devices.
- Performance requirements: Described in Section 3.4.

2.2.3 Risks

The risks posed come in the form of dependencies, that being Aseprite and the Godot game engine. Even though the amount of risks is low and unlikely to occur, it is still important to keep them in mind for a worst-case scenario. The following are the dependence-based risks and the ways to work around and mitigate them:

- Aseprite Plugin: Currently, there is a plugin for Godot that allows for an easy way to port Aseprite art and animation files into Godot, making it easier to quickly separate and organize groups of animations and sprites. However, there is a chance that the plugin may not work or become unavailable. If that occurs, all animations and sprites would have to be converted to PNG images and manually sorted and organized.
- Godot: During development, the creators of Godot may make a new update available for the game engine. This poses a risk of certain functions and snippets of code becoming unsupported or broken. To mitigate this risk, development of the game will be done solely on version 4.2.1

2.3 Alignment with an Enterprise Architecture Framework

Not Applicable.

3 Design Considerations

3.1 Goals and Guidelines

Goals

- **Resemble Stardew Valley.** This project seeks to replicate the basic functionality, quality, and charming mood of the simulation RGP video game Stardew Valley. This will be done through the implementation of chopping trees and harvesting wood, a combat system, dynamic player health, an in-game timekeeping system, and player progress stats. The graphics and sound effects will be developed with aesthetic similarities to Stardew Valley. The game will take a top-down perspective and pixel art style and the music will loop with an upbeat 8-bit style, where sound effects will play with actions such as chopping wood. The Beaver Game will share similar UI interfaces for the home screen, inventory, settings, and status bars to Stardew Valley.

The twist or feature that extends the Beaver Game from Stardew Valley is that the main player is a beaver who changes the quality of the landscape by controlling water flow. The former goal of mimicking Stardew Valley is vital to providing a strong basis for the Beaver Game. The gameplay in Stardew Valley has a proven audience and entertainment aspect. Stardew Valley also provides intuitive, streamlined UI interfaces and controls. This goal will aid in ensuring that the twist of the Beaver Game is successful through dimensional, accessible gameplay.

- **Compact Memory Usage.** The Beaver Game is to be hosted locally through an exported Godot executable file. This file must have a conservative memory usage expectation. Thoughtful memory usage is essential for exposing the Beaver Game to a larger audience with a wide range of hardware access.
- **Optimized Performance.** The Beaver Game will emphasize speed and smooth gameplay, which ensures minimized lag and load time.

Guidelines

- **Coding Standards.** The Beaver Game will be developed with the best practices in version control and code documentation. Naming conventions, commenting, and formatting will be standardized across the code. This guideline will ensure maintainable, scalable, and readable source code.
- **Iterative Development and Player Feedback.** Iterative development will ensure issues with gameplay, bugs, and feature design issues are known and combated early on. Continuous player feedback aims to enhance user experience. A user-driven development strategy ensures that the final product aligns with users' expectations.

3.2 Development Methods and Contingencies

Development Methods

Godot-Specific Tools and Best Practices: The Beaver Game will be developed using the Godot game engine. Godot is a free and open-source game engine that has vast 2D game development features which this game will leverage fully. The Beaver Game will utilize the GDScript language, Godot code editor, and Godot best practices.

Object Oriented Programming (OOP): Godot takes an object-oriented programming (OOP) based approach to game assets to reduce high coupling between game components. The Node class is the basic building block of Godot game components. Nodes ensure that complex systems such as physics, audio, and graphic rendering are held in separate component classes. A user-defined Node, such as the one that defines the main player, can inherit any number of preexisting Godot Nodes.

Agile Development with Scrum: The Agile software development methodology, specifically with the Scrum framework, is an iterative software development and project management approach. This process will be used to structure and accomplish project objectives and aid the development team's ability to react to change, improve the product, and manage strict deadlines. This approach ensures collaboration, clear communication, and alignment with expectations.

Prototyping: Prototyping is an essential aspect of the game development process, allowing for the exploration and validation of design concepts and gameplay mechanics early in the project lifecycle. In Godot, rapid prototyping is facilitated by the engine's flexible scene system and scripting capabilities, enabling quick iterations and adjustments based on playtest feedback. Prototyping serves as a critical tool in assessing the feasibility of game features, fine-tuning user experience, and identifying potential issues before committing to full-scale development. This practice aligns with the principles of Agile development, emphasizing the value of working software and user feedback as primary measures of progress.

Contingencies

Glitches: During the implementation of components-based architecture glitches are a possibility that should be anticipated. To address this contingency a testing strategy should be implemented to identify glitches when they occur. These tests will be implemented often to ensure that glitches are caught quickly. The other part of this contingency is a rollback plan to revert the game to a previous stable version in the case of a severe glitch. A git version control will be maintained and updated often, allowing the ability to revert previous working states if necessary.

Scope Creep: To manage scope effectively, a change control process will be implemented. All proposed changes to the game's scope will undergo careful evaluation, considering their impact on existing components and the project timeline. Regular reviews of the project scope will be conducted, ensuring alignment with the initial design goals.

Platform Limitations: This will involve a flexible design approach that considers the different platforms. The team will test the game on various devices to identify and address potential issues. If significant platform-related challenges are encountered, alternative plans may involve the development of platform-specific components to ensure the game works correctly on all platforms.

3.3 Architectural Strategies

1. **Use of the Godot Engine Decision:** Utilize the Godot Engine as the primary development platform for the game.

Reasoning: Godot offers a robust set of tools and features specifically designed for game development, including a visual editor, a built-in physics engine, and support for both 2D and 3D graphics. Leveraging Godot allows for rapid development, cross-platform compatibility, and streamlined deployment of the final product.

Alternatives Considered: Other game engines, such as Unity or Unreal Engine, were evaluated. However, Godot was chosen for its lightweight nature, ease of use, and strong community support. Other engines considered were the Unreal and Unity engines. Godot was chosen for the lightweight 2D development features and open-source licensing features.

2. **Use of the GdScript Language:**

Reasoning: The choice to develop using Godot Engine-specific language GdScript was chosen for many reasons. GdScript is specifically designed for use with the Godot game engine, offering seamless integration and efficient performance. GdScript provides concise syntax, built-in functionality for game development tasks, and rapid iteration.

Alternatives Considered: C# was considered as an alternative language for development within the Godot Engine. C# is a widely used language with a large community. GdScript was chosen due to its native integration with Godot.

3. **Use of the Aseprite Decision:**

Reasoning: Aseprite is specifically designed for creating pixel art and animations. It offers dedicated features for pixel design that align with the requirements for the game. Aseprite supports seamless integration with the Godot Engine, streamlining the workflow between graphics creation and game development.

Alternatives Considered: Other Art programs were considered such as Photoshop and Procreate. However, Aseprite was chosen for its ease of use and library to integrate with Godot. Other art programs were considered but Aseprite was chosen for its 2D animation and art support.

4. **Components Based Architecture:**

Reasoning: The adoption of a components-based architecture aims to achieve modular design and code reusability. This allows for breaking complex systems into smaller, manageable components. It promotes easier unit testing as one can isolate components. Components can be reused across different parts of the system, promoting efficiency in development and reducing redundancy.

Alternatives Considered: Other architectural patterns, such as the monolithic architecture, were considered. However, a components-based architecture was chosen due to its modularity, code reusability, and ease of testing.

5. **Error detection and recovery:**

Reasoning: The game will implement mechanisms to detect errors related to runtime issues. In the event of an error, the game will incorporate recovery strategies. This may involve

providing error messages and saving the game to prevent data loss. For critical errors, the game may prompt the user to restart.

Alternatives Considered: One alternative considered was to rely on the default error handling provided by the underlying programming language. However, this approach may lead to a poor user experience with abrupt crashes and potential data loss.

6. Communication mechanisms:

Reasoning: Message bus and direct message calls were chosen for the game due to efficient communication. They allow real-time communication between the parts of the game. These mechanisms are also built into Godot to allow ease of use.

Alternatives Considered: Other methods such as the Observer Pattern were considered but were not chosen due to the tendency for memory leaks and complexity. Message buses and direct message calls are the most efficient communication mechanisms for the game.

3.4 Performance Engineering

1. **Requirement:** Reasonable load times are required.

System Design: To address this requirement, the design of the system focuses on optimizing the loading process. This includes utilizing appropriate data structures for quick retrieval and efficient asset loading to ensure quick load times.

2. **Requirement:** A fluid and constant frame rate to provide a seamless graphics system.

System Design: To achieve a fluid and constant frame rate, the system design focuses on efficient rendering techniques. For example, managing the spawn rates of certain entities or assets while the player is playing the game to not put a strain on the game's performance.

3. **Requirement:** Mindful memory usage to be considerate of end users' system requirements.

System Design: The system design focuses on efficient asset compression and selective inclusion of assets. To ensure that the exported game is less than 500 MB.

4 System Architecture and Architecture Design

This section outlines the system architecture design of the Beaver Game. All supporting Figures can be found in Appendix [F.1](#)

4.1 Logical View

A UML class diagram (Figure [1](#)), which demonstrates the game's high-level modules and their relationships, and data flow diagrams (Figures [2](#), [3](#), & [4](#)) are seen in Appendix [F.1](#)

4.2 Hardware Architecture

The game is a centralized processing system that will run locally on the user's computer. The player's device hosts the game and handles all processing tasks. Section [4.2.2](#) discusses the required hardware for said system.

4.2.1 Security Hardware Architecture

Not Applicable.

4.2.2 Performance Hardware Architecture

The recommended computing power required for users to run an exported Godot game, the Beaver game's chosen development engine, is as follows:

- CPU: Windows: x86_64 CPU with SSE4.2 instructions, with 4 physical cores or more (ex. Intel Core i5-6600K, AMD Ryzen 5 1600).
macOS: x86_64 or ARM CPU (Apple Silicon) (ex. Intel Core i5-8500, Apple M1).
- GPU: For basic graphical compatibility, a dedicated graphics card supporting OpenGL 4.6 is required (ex. NVIDIA GeForce GTX 650 or AMD Radeon HD 7750).
- RAM: A minimum of 4 GB.
- Storage: A minimum of 150 MB of available storage to be used for the executable, project files, and cache.
- Operating system: Windows 10, macOS 10.15

These specifications ensure that the game runs smoothly and provides a good user experience on both Windows and macOS platforms.

4.3 Software Architecture

Software and Tools

Godot Engine:

- **Function:** Game development platform which provides tools for 2D game creation.
- **Target Hardware:** User PC (Windows or MacOS).
- **Location:** Locally installed on developer machine.
- **Version:** Godot Engine, Version 4.2.1
- **Licenses:** MIT License (open source).

GDScript:

- **Function:** The high-level, object-oriented programming language developed for Godot. This is used for game logic scripting.
- **Compiler/Interpreter:** Integrated within the Godot Engine.
- **Location:** Scripts are located in the game project directory.

JSON/Dictionary Objects:

- **Function:** Data storage structures for the game states, player progress, and configurations.
- **Location:** Local storage on the user's device within the game's directory.

Computer Software Components (CSCs) Figure 1 is a class diagram that depicts the classes of the Beaver Game. In this diagram, each class is defined by a reference number and a name. For example, the Tree class is labeled as 7. The following is a description of each class in order of their numerical labels.

Classes

1. Castor (Player)

- **Type:** Game Entity
- **Purpose:** To represent the playable character in the game world.
- **Function:** Allows the player to interact with the game world, gather trees, build/upgrade structures, fight adversary NPCs, and communicate with friendly NPCs.

2. River Map

- **Type:** Game World Component
- **Purpose:** To represent the dynamically changing riverscape that the playable character improves over gameplay.
- **Function:** Provides the home base of the playable character and hosts the river game entity and the buildable structures. Hosts some tree game entities.

3. Forest Map

- **Type:** Game World Component
- **Purpose:** To represent the forest in which the playable character must gather resources and face predators/ adversary NPCs.
- **Function:** Serves as the primary location for resource gathering.

4. Friendly NPC

- **Type:** Game Entity
- **Purpose:** To provide tutorial information to the playable character.
- **Function:** Enhance the narrative and gameplay through interactions and instructional support.

5. Adversary NPC

- **Type:** Game Entity
- **Purpose:** To challenge the player through conflict.
- **Function:** Acts as an enemy obstacle to the act of harvesting wood.

6. Inventory

- **Type:** Management System
- **Purpose:** To track and manage wood supply collected by the playable character.
- **Function:** Allows the user to store, use, and track wood collected.

7. Tree

- **Type:** Resource
- **Purpose:** To serve as the primary natural resource for upgrading structures.
- **Function:** Can be harvested by the player for wood and allocated to building structures.

8. Structures

- **Type:** Constructible Object
- **Purpose:** To provide the player with the ability to build two upgradable structures within the game world.
- **Function:** Impact the environment health based on the level of upgrade.

9. Lodge

- **Type:** Subclass of Structures
- **Purpose:** To serve as the player's main base of operations.
- **Function:** Offers spawn point, passive wood collection, and impacts environment health.

10. Dam

- **Type:** Subclass of Structures
- **Purpose:** To create a challenge by having a continuously degrading quality.
- **Function:** Impact the water retention status of the river.

11. River

- **Type:** Game World Entity
- **Purpose:** To provide a dynamic and interactive waterway within the game world.
- **Function:** Affects game progression by having a water retention status determined by structures which impacts environment health.

12. Environment Health

- **Type:** Status
- **Purpose:** To drive gameplay.
- **Function:** Provide a metric to trigger the river map visual environment change and mark game completion. To be determined by the upgrade status of structures and water retention value.

Class Relationships

- **Castor (Player) and the Game World**

The basis for the Beaver Game lies in the relationship between the player, represented by the Castor (Player) class (1), and the game world, encapsulated by various environmental and structural classes such as River Map (2), Forest Map (3), Structures (8), and Environment Health (12). Each world-defining class is designed to fulfill specific roles within the game's ecosystem.

The Castor (Player) class interacts with the landscapes of the River and Forest Map classes. These map classes act as subcomponents of the game's world, providing resources and challenges and defining the finite world map. The player interacts with these components by harvesting materials from the Tree class (7) for building or modifying the environment by constructing the two Structures subclasses. The Forest map is required to be interacted with to collect a sufficient amount of wood. This interaction between the maps and the player is crucial for the gameplay, as it directly influences the Environment Health class, reflecting the game's core mechanics around conservation and ecosystem management.

- **Castor (Player) and NPCs**

Friendly NPC and Adversary NPC classes serve as agents within the game world, offering assistance and challenges respectively. These classes interact with both the player and environmental classes. For example, an Adversary NPC might interrupt the player's collection of wood in the forest map, directly impacting the player's ability to upgrade their structures. If the player is to be killed¹ by Adversary NPC they lose all the wood in their inventory and respawn at the home lodge, raising the stakes for collecting wood.

- **Castor (Player) and Inventory**

The Inventory class functions as a management system for the wood collected by the player. All wood collected will be tracked in this system unless allocated to a structure or lost during combat.

Data Flow and Process Level Decomposition The high-level flow of data in the Beaver Systems is demonstrated at various levels in Figures 2, 3, and 4 through the use of Data Flow Diagrams. These diagrams communicate the big picture of how data flows through the system.

4.3.1 Security Software Architecture

Not Applicable.

4.3.2 Performance Software Architecture

Below the software components supporting the performance and reliability of the system are discussed. These components will be integrated throughout each class mentioned in Section 4.3. These performance-optimized components are provided features of the Godot engine.

¹Killed being defined by the player's health status reaching zero.

Software Components

- **Physics Engine:** A physics engine is a software component that simulates the physical behaviors of game objects. Godot's built-in physics engine handles all collision and interaction logic for static and dynamic objects.
- **Game Logic Implementations:** The scripts controlling the logic of the classes mentioned in Section 4.3 will implement conservative memory allocation and garbage collection strategies. Scripts with relationships or dependencies to one another will use Godot's signals to emit information, when applicable. This allows object interactions without requiring objects to reference one another, reducing coupling.

Single Points of Failure (SPOF) The Save/Load system can become an SPOF if not properly implemented. To mitigate the associated issues of a Save/Load system such as data corruption and inability to load game saves, a data validation strategy will take place after each manual save.

Data Validation Process: Whenever a player initiates a manual save, the game executes a data validation routine. This routine checks the integrity and consistency of the game data to be saved. It ensures that all game state information is correct and that there are no errors or corruptions that could compromise the save file.

4.4 Information Architecture

In the Beaver Game data is stored and manipulated locally using the Godot Dictionary data structure and JSON files. The types of data stored for each game save are related to player stats, structure level status, environment health status, and time spent in the game. This data is the marker of player progress. This data with other supporting data such as player location on the map, resource inventory, and current map rendering can be used to create an image of the current game state or a save file. Accessing, creating, and managing save files will be supported by the Save/Load systems of the Beaver Game. The data related to the Beaver game is stored locally on the user's computer and does not access data across a network. This game does not store nor access personally identifiable information (PII), individually identifiable information (IIF), or personal health information (PHI).

4.4.1 Records Management

The game collects data on player progress (player stats, resources gathered, structures built/upgraded) and game settings (audio levels, graphics settings) across a finite number of save files. This data is supplied by the player through their interactions with the game interface. Data is collected and stored electronically, without any paper or manual input methods involved

Manual/Electronic Inputs Upon entering the game system, all electronic inputs from the player, such as game progress, are processed and stored within the game's data management system. Instead of utilizing a traditional relational database, the game employs a series of structured JSON files for data storage. This approach allows for a flexible and easily accessible format for storing and retrieving game data, which includes but is not limited to game settings, saved game states, and in-game level achievements.

This file-based approach to data management enables the game to operate efficiently without the need for a conventional database management system. Additionally, it simplifies data backup and transfer processes.

Master Files The master files of the Beaver Game contain detailed records of:

Player Progress: Player health and strength status, resources gathered, and current status of built or upgraded structures. Game Settings: Custom configurations set by the player relating to audio volume. This data is maintained to facilitate game functionality, such as loading saved games, and applying settings across game sessions.

4.5 Internal Communications Architecture

Not Applicable.

4.6 Security Architecture

Not Applicable.

4.7 Performance

Not Applicable.

4.8 System Architecture Diagram

The system architecture diagram (Figure 5) can be seen in Appendix F.1.

5 System Design

5.1 Business Requirements

Not Applicable.

5.2 Database Design

5.2.1 Data Objects and Resultant Data Structures

Player Progress:

- Data Structure: JSON object
- Data Elements:
 - Type: Object
 - Length: Variable
 - Source: User input and game events
 - Validation Rules: Ensure valid data types and range for numeric values

- Maintenance: CRUD operations allowed
- Data Stores: Local storage on the user’s device
- Outputs: Used to render player statistics and progress

Game Settings:

- Data Structure: JSON object
- Data Elements:
 - Type: Object
 - Length: Variable
 - Source: User configuration
 - Validation Rules: Validate against predefined options
 - Maintenance: CRUD operations allowed
 - Data Stores: Local storage on the user’s device
 - Outputs: Used to apply user-specific settings

5.2.2 File and Database Structures

Database Management System Files Not Applicable.

Non-Database Management System Files Not Applicable.

5.3 Data Conversion

Not Applicable.

5.4 User Machine-Readable Interface

5.4.1 Inputs

Game Controls:

- Input Media: Keyboard, Mouse.
- Mapping to Data Flows: Direct user interactions.
- Input Screens: In-game controls and menus.
- Data Element Definitions: Movement, action commands.
- Edit Criteria: Ensure valid commands, and range checks for numerical inputs.

5.4.2 Outputs

Game Display:

- Output Media: Computer Monitor.
- Mapping to Data Flows: Rendering of game elements.
- Output Screens: Game interface and HUD.
- Report Contents: Visual representation of player progress, environment status.
- Report Distribution: Real-time rendering to the user screen.
- Access Restrictions: Ensure privacy and security considerations.

5.5 User Interface Design

At the time of this Software System Design document creation the the detailed user interface designs, including layouts of main menus, in-game screens, and Heads-Up Display elements, have not been finalized. As the development progresses, the design team will collaborate to create visually engaging and user-friendly interfaces that align with the overall vision of the game.

5.5.1 Section 508 Compliance

The game interface is designed with some consideration for accessibility. However, they are not comprehensive in their support. The following accessibility considerations have been implemented:

- Language Support: The game will only be available in English.
- Visual Accessibility: Efforts to ensure that visual elements, such as text and icons, are clear and legible will be made. Color choices have not been tested for color blindness accessibility.
- Auditory Accessibility: Sound effects and music are not essential for gameplay. Players who are deaf or hard of hearing can still fully engage with the game.

6 Operational Scenarios

The following are the core operational scenarios that define the functionality of the system.

1) Starting the Game:

- Objective: To initiate a new game session.
- Precondition: The game is open to the main menu.
- Trigger: The player selects "New Game" from the main menu.
- Steps:
 1. Player chooses the "New Game" button.
 2. Game initializes the game world with beginning stats.

3. Player is in the game world, ready to begin playing the Beaver Game.

- Frequency: Every time a player starts a new game session.

2) Saving and exiting the game:

- Objective: To exit the game and maintain a save file for the achieved game progress.
- Precondition: The player is in active gameplay.
- Trigger: The player selects "Exit Game" from in-game UI.
- Steps:
 1. Player opens in the game UI menu.
 2. Player selects "Exit Game" button.
 3. Player is prompted to "Save and Exit Game", "Continue", or "Exit Game Without Saving".
 4. If player select "Save and Exit Game" or "Exit Game Without Saving", they are prompted with a redundancy check to confirm the action. If the player selects "Continue", the prompt will exit the in-game UI window without changing anything.
 5. Game state is saved if "Save and Exit Game" is selected. The game state is exited without saving the progress made during the session if "Exit Game Without Saving" is selected. Both options exit the game to the main menu.
- Frequency: Every time a player wishes to end a game session.

3) Loading a saved game:

- Objective: To resume the game from a specified game save.
- Precondition: The game is open to the main menu.
- Trigger: The player selects "Load Game" from the main menu.
- Steps:
 1. Player chooses the "Load Game" button.
 2. A selection menu is opened with previously saved games. If there are no previous saves, the "Load Game" button will raise a flag with this information and remain on the main menu.
 3. Player selects a save file to play on.
 4. Game initializes the game world with saved stats.
 5. Player is in the game world, ready to begin playing the Beaver Game.
- Frequency: Every time a player starts a game session from a previous save.

4) Harvest wood from trees:

- Objective: To gather material required for structure upgrades.

- Precondition: The game is running a loaded or new game save.
- Trigger: The player decides to cut down a tree.
- Steps:
 1. Player navigates to a tree within either the River Map or the Forest Map.
 2. Player presses the action key for cutting down a tree on their keyboard.
 3. Player waits for the action to be performed while holding the actions keyboard input. This waiting period is to be determined by the player's strength status.
 4. Upon completion, the wood is added to the player's inventory.
- Frequency: Every time a player cuts down a tree.

5) Building and Upgrading a Dam or Lodge:

- Objective: To progress gameplay by improving the river's water retention levels and ultimately the environment health stat.
- Precondition: The game is running a loaded or new game save.
- Trigger: The player selects "Build" or "Upgrade" after interacting with a structure.
- Steps:
 1. Player navigates to the site of the dam or the lodge in the River Map.
 2. Player is met by a menu that describes the structure's current level and requirements for the next upgrade. If the structure is a Lodge on or past level two, the player can collect passive wood once a day from this interaction.
 3. If the player has enough wood pieces, the construction or upgrade can be performed.
 4. The structure is instantaneously upgraded. The upgrade impacts the environment's health status and may trigger a visual environmental health change the following day.
- Frequency: Every time a player wishes to build/upgrade a structure. This occurs six times in the game as there are three upgrades per structure.

6) Engaging in Combat with Predators:

- Objective: To add a challenging element to the resource collection aspect of gameplay.
- Precondition: The player is located in the Forest Map.
- Trigger: The player is in the trigger proximity radius of a predator.
- Steps:
 1. Player navigates within the proximity radius of an idle predator.
 2. Predator becomes alert and activates attack mode.
 3. If the player navigates out of the predator proximity radius, the predator will leave attack mode after some time if this condition remains true.


4. If player chooses to attack the predator, the player and the predator engage in attacking each other. The player can perform the "beaver tail slap" attack by pressing the specified attack key input.
 5. The combat continues until the predator or player's health points reach zero.
 6. If the player wins, the predator is defeated and the player's strength status is improved.
 7. If the player's health reaches zero, the game proceeds according to its loss condition . The player respawns at the Lodge with zero wood remaining in their inventory.
- Frequency: Every time a player is in the Forest Map and is near a predator.

7 Detailed Design

7.1 Hardware Detailed Design

Not Applicable.

7.2 Software Detailed Design

This section outlines a detailed design for the key software services and classes in the Beaver Game, based on the provided class descriptions of Figure .

Player

Classification: Game Entity, Playable Character

Definition: Represents the player's character, enabling interaction with the game world.

Requirements:

- Must be able to move fluidly.
- Must be able to gather resources.
- Can allocate resources to building and upgrading structures.
- Can engage in combat with adversary NPCs and interact with friendly NPCs.

Internal Data Structures:

- int health: Health of the player from 100 - 0.
- int strength: Strength of the player from 0 - 100.
- int capacity: Maximum storage capacity.

Constraints: Gameplay is limited by the player's health and inventory capacity.

Composition: The player is composed of subsystems of movement controls, resource management, combat systems, and interaction mechanics.

Users/Interactions: The player interacts with the game world through user inputs such as keyboard or mouse.

Processing:

- `move()`: Controls the player's movement within the game world.
- `attack()`: Initiates combat actions against adversary NPCs.
- `chop_tree()`: Initiates the action of chopping down a tree to gather resources.
- `harvest_wood()`: Manages the process of gathering wood resources from trees and allocating them to the player's inventory.

River Map

Classification: Game World Map

Definition: The River Map serves as one of the primary environments within the game, featuring a dynamic river ecosystem that plays a critical role in the player progression systems.

Requirements:

- The graphics are dynamic based on the Environment Health status. When a stat level is achieved the land shall appear more fertile.

Constraints: The River map hosts the game progression elements, so it must run smoothly and provide entertaining gameplay to the end user.

Composition: The River Map is composed of the Dam, the Lodge, the River, Trees, and other visual elements such as grass.

Users/Interactions: The Player interacts with the River Map through various activities such as resource gathering, building structures (e.g., dams), and navigating the terrain.

Processing:

- `change_landscape()`: This function enables the game to simulate dynamic environmental changes within the River Map.

Forest Map

Classification: Game World Map

Definition: The Forest Map serves as the second environment map within the game, featuring abundant harvesting material for the player and offering the challenge of the combat system.

Requirements:

- Must feature a high density of trees, serving as the main source of wood for the player.
- Must be populated with Adversary NPCs.

Constraints: The abundance of resources and NPC adversaries must be balanced to ensure a challenging yet rewarding experience for the player.

Composition: The Forest Map is composed of Trees and Adversary NPCs.

Users/Interactions: The Player interacts with the Forest Map through various activities, including resource collection and combat.

Friendly NPC

Classification: Game Entity

Definition: Friendly NPCs are non-hostile characters within the game that provide assistance and tutorial information.

Requirements:

- Must be distinguishable from adversary NPCs.
- Should offer various narrative and tutorial dialogue interactions.

Internal Data Structures:

- int type: 1, 2, or 3. Identifies which of the three Friendly NPCs the NPC is.
- Dictionary dialogue: Stores dialogue options based on the player's progress or NPC's type.

Constraints: Interaction with Friendly NPCs should be designed to avoid disrupting the flow of gameplay.

Composition: Friendly NPCs are composed of a set of predetermined dialogues and a function for initiating interaction.

Users/Interactions: Players interact with Friendly NPCs to gain narrative or tutorial information.

Processing:

- `get_dialogue()`: Retrieves the appropriate dialogue options to the player based on the NPC's type and the player's status in the game.

Adversary NPC

Classification: Game Entity

Definition: Adversary NPC, portrayed as natural predators of the beaver, are used to enhance the complexity of gameplay. They introduce a layer of challenge to the act of resource collection. Their habitat, the Forest Map, is where most trees will be harvested by the player.

Requirements:

- Predators must enter attack mode if the player enters their defined proximity radius.
- Predators must continue to attack the player until the predator dies, the player dies, or the player has left the proximity radius for a specified amount of time.

Internal Data Structures:

- `int type`: 1 or 2. The predators can be a Cyote or a Fox.
- `int health`: Health of the predator from 100 - 0.
- `int strength`: Aids in determining the attack damage to the player.

Constraints: Must be located in the Forest Map.

Composition: Predators are composed of an AI-based combat and stalking systems.

Users/Interactions: The Player will partake in combat with the Adversary NPCs.

Processing:

- `attack()`: Defines the logic for initiating and conducting an attack on the player, including damage calculation and applying the effects of the attack.
- `stalk()`: Engages a behavior pattern where the predator actively seeks out the player within a certain range, simulating a hunting behavior.

Inventory

Classification: Inventory Management System

Definition: A subsystem designed to manage and store the player's collected resources.

Requirements:

- Must dynamically update to reflect the current number of resources held by the player.
- Ensure the capacity constraints are enforced.
- Must reset if the player dies during combat.
- Must be saved in game saved data.

Internal Data Structures:

- int capacity: Describes the number of wood pieces that can be carried.

Constraints: Capacity is fixed and predefined and must be enforced by the system.

Users/Interactions: Inventory is used by the player for resource management.

Processing:

- update(): Updates the inventory based on allocation, collection, or loss of wood.

Tree

Classification: Game Entity

Definition: Trees are the most critical environment entity in the game as they are required to upgrade structures.

Requirements:

- Must spawn a finite number of trees each in-game day.
- The spawn rate for trees is higher in the Forest Map.

Internal Data Structures:

- int health: metric for determining cut down time.

Constraints: The game must balance the spawn rate of trees to prevent the resource from being too sparse or too abundant.

Composition: Trees are composed of various attributes such as type, health, and location.

Users/Interactions: The player interacts with trees to harvest them.

Processing:

- `harvest()`: Mechanism for harvesting trees and deallocating their memory in the world map.
- `spawn(map_location)`: Spawn rate of trees determined by which map they are in.

Structures

Classification: Game Entity

Definition: Structures within the game serve as fundamental components that support player progression, environmental interaction, and gameplay mechanics. Each structure has unique functionalities, upgrade requirements, and roles within the game's ecosystem.

Requirements:

- Structures must provide specific functionalities or benefits to the player or the game environment.
- Structures should have upgrade paths that allow for increased efficiency in water retention or additional capabilities.

Internal Data Structures:

- `int level`: determined by upgrades. Is 1, 2, or 3.

Dam

Classification: Sub-entity of Structures

Definition: Main mechanism for improving water retention level and overall Environment Health status.

Requirements:

- Dam must be eroded by water flow.
- Dam level stat must impact Environment Health status.

Internal Data Structures:

- `int dam_level`: determined by upgrades. Is 1, 2, or 3.
- `int health`: From 0-100. Degrades over time and can impact supplies needed to upgrade the dam or maintain the dam level.

Constraints: Can only be upgraded to a maximum level of 3, where each upgrade requires an increasing amount of resources.

Composition: The Dam is composed of an upgrade system, an erosion mechanism, and a water retention calculator.

Users/Interactions: The Player interacts with the Dam to perform upgrades

Processing:

- `retain_water()`: increases the water retention status in a more continuous way.
- `erode()`: Erode the dam (increase the amount of wood until the next upgrade).

Lodge

Classification: Sub-entity of Structures

Definition: The player's home base.

Requirements:

- Act as the spawn point after dying in combat.
- Generate passive wood during upgrade levels 2 and 3.
- Lodge level stat must impact Environment Health status.

Internal Data Structures:

- `int lodge.level`: determined by upgrades. Is 1, 2, or 3.

Constraints: Can only be upgraded to a maximum level of 3, where each upgrade requires an increasing amount of resources.

Composition: The lodge consists of a spawn point system, upgrade system, and passive wood generation mechanism.

Users/Interactions: The Player interacts with the Lodge to perform upgrades, spawn, and collect passive wood. The Lodge is present in the River Map.

Processing:

- `give_passive_wood()`: Generate passive wood for the player based on the timekeeping cycle.

River

Classification: Game World Entity

Definition: The River entity represents a dynamic waterway within the game.

Requirements:

- Must have a calculable water retention status based on structure levels.
- Must have a dynamic flowing visual depiction.

Internal Data Structures:

- `int retention_level`: Represent the current water level of the river from 0 - 100.

Constraints: Water level and flow rate must stay within predefined limits to prevent unrealistic scenarios.

Composition: The River is composed of water physics simulations and visual shaders to dynamically simulate a realistic and interactive river within the game world.

Users/Interactions: The Player can swim through this feature. The River visually increases in size in the River Map. River status impacts the Environment Health Status.

Processing:

- `flows()`: Flowing water function which can be used to calculate dam degradation.

Environment Health

Classification: Status Indicator

Definition: The Environment Health status represents the overall health of the game ecosystem. It is affected by the player's progress on structure building and serves as a key indicator of game progression. When the Environment Health status is maximized the player has successfully restored the health of the landscape and has won the game.

Requirements:

- Must be reflected visually by changing the assets (e.g. grass, trees) of the River Map into more lush variations of themselves.
- Should increase in relationship to the levels of the dam and lodge.
- Must be visually in the in-game UI stats menu.

Internal Data Structures:

- `int environment_health`: Represents the health of the environment from 0 - 100.

Constraints: The value of `environment_health` cannot exceed 100 or fall below 0.

Composition: Dependent on the status of Dam, Lodge, and River entities.

Users/Interactions: The Player impacts this status through gameplay. The River Map reflects the value of this status visually.

Processing:

- `set_health(int dam_level, int lodge_level, int rention_level)`: Calculates the value of environment_health.

7.3 Security Detailed Design

Not Applicable.

7.4 Performance Detailed Design

Not Applicable.

7.5 Internal Communications Detailed Design

Not Applicable.

8 System Integrity Controls

Not Applicable.

9 External Interfaces

Not Applicable.

9.1 Interface Architecture

Not Applicable.

9.2 Interface Detailed Design

Not Applicable.

A Record of Changes

Table 1: Record of Changes

Version Number	Date	Author/Owner	Description of Change
1.0	02/29/2024	CAN Game Studio	Initial draft.

B Acronyms

Table 2: Acronyms

Acronym	Literal Translation
AI	Artificial Intelligence
CPU	Central Processing Unit
FPS	Frames Per Second
GPU	Graphics Processing Unit
HUD	Heads Up Display
RAM	Random Access Memory
RPG	Role-Playing Game

C Glossary

Table 3: Glossary

Term	Acronym	Definition
Artificial Intelligence	AI	The simulation of human intelligence in machines that are programmed to think like humans and mimic their actions.
Central Processing Unit	CPU	The primary component of a computer that performs most of the processing inside a computer.
Frames Per Second	FPS	A measure of how many unique consecutive images a computer graphics system can produce in one second.
Graphics Processing Unit	GPU	A specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display device.
Heads Up Display	HUD	A Display that presents data without requiring users to look away from their usual viewpoints
Random Access Memory	RAM	A type of memory that is volatile and accessed randomly.
Role-Playing Game	RPG	A genre of video game where players control the actions of characters immersed in a well-defined world.
Sprite	N/A	A two-dimensional image or animation that is integrated into a larger scene or game environment.
Top-down	N/A	A game that offers an elevated viewpoint above the action.

D Referenced Documents

Table 4: Referenced Documents

Document Name	Document Location and/or URL	Issuance Date
Initial Project Idea Submission: Conception of the Beaver Game	https://github.com/natalieswork/SeniorProjectDocuments/blob/main/Senior_Project_Topic_Idea.pdf	12/06/2023
Team Formation Proposal I	https://github.com/natalieswork/SeniorProjectDocuments/blob/main/Team_Formation_I.pdf	12/12/2023
Team Formation Proposal II	https://github.com/natalieswork/SeniorProjectDocuments/blob/main/Team_Formation_II.pdf	01/25/2024
Project Proposal (Summary Sheet) for the Beaver Game	https://github.com/natalieswork/SeniorProjectDocuments/blob/main/Summary_Sheet.pdf	02/01/2024
Software Requirements Specification (SRS) for the Beaver Game	https://github.com/natalieswork/SeniorProjectDocuments/blob/main/SRS.pdf	02/08/2024

E Approvals

The undersigned acknowledge that they have reviewed the SDD and agree with the information presented within this document. Changes to this SDD will be coordinated with, and approved by, the undersigned, or their designated representatives.

Table 5: Approvals

Document Approved By	Date Approved
x _____ Name: Carrie Wasieloski, Audio & Environment Specialist - CAN Game Studio	Date: _____
x _____ Name: Austin Counterman, Art & Animation Expert - CAN Game Studio	Date: _____
x _____ Name: Natalie Lee, Lead Engineer - CAN Game Studio	Date: _____
x _____ Name: George Dimitoglou, Ph.D., Stakeholder - Hood College	Date: _____

F Additional Appendices

F.1 Software Architecture Diagrams

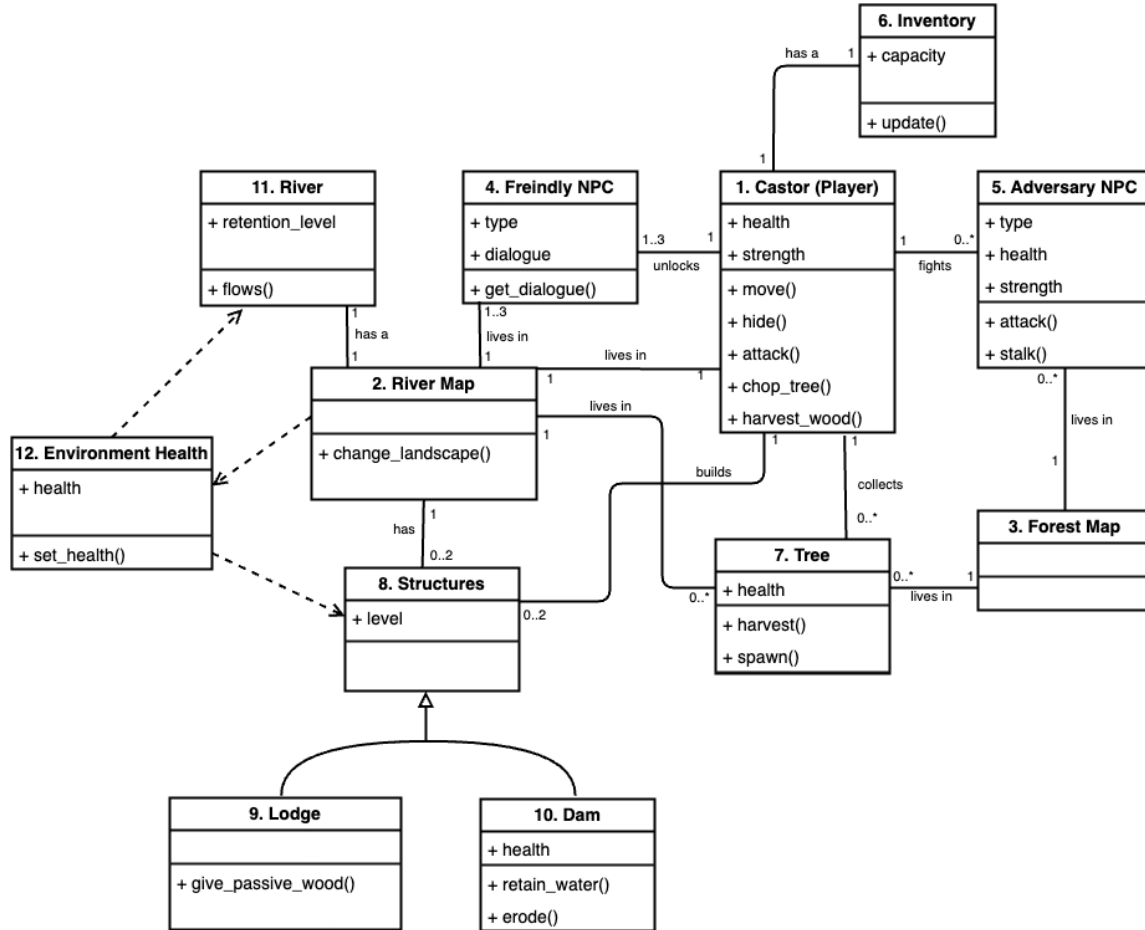


Figure 1: UML Class Diagram of the Beaver Game System. This diagram depicts major game logic components.

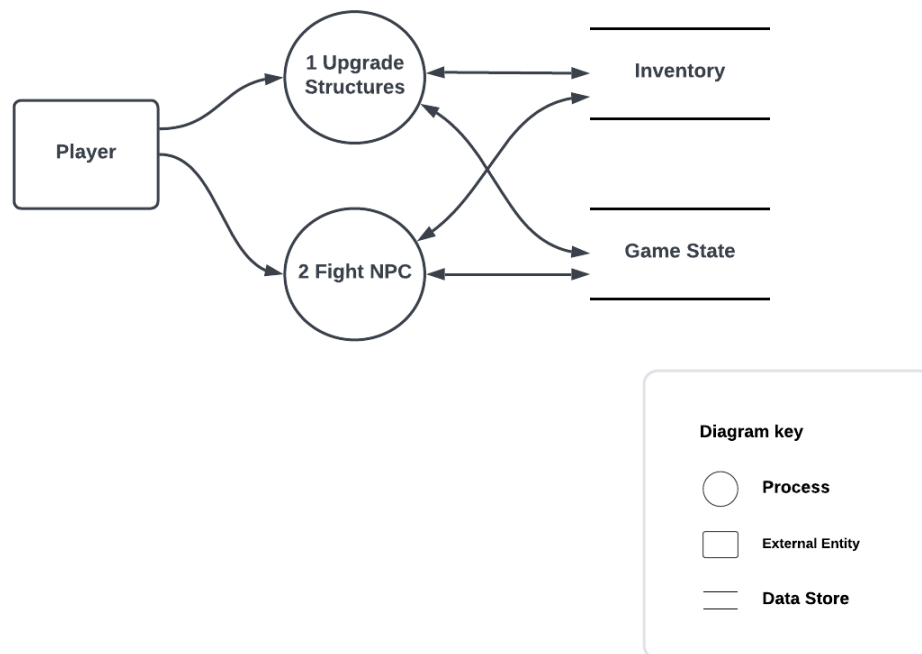


Figure 2: Data flow diagram at layer 0 or the context layer.

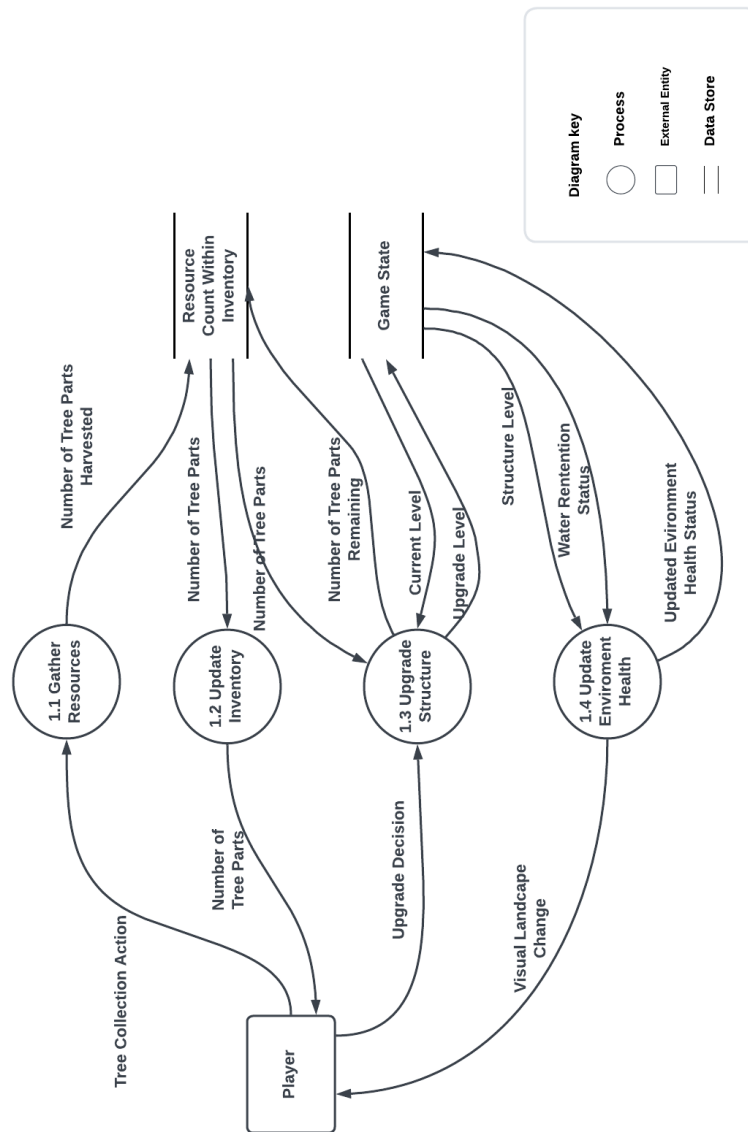


Figure 3: Data flow diagram at layer 1 describing the upgrade structure process.

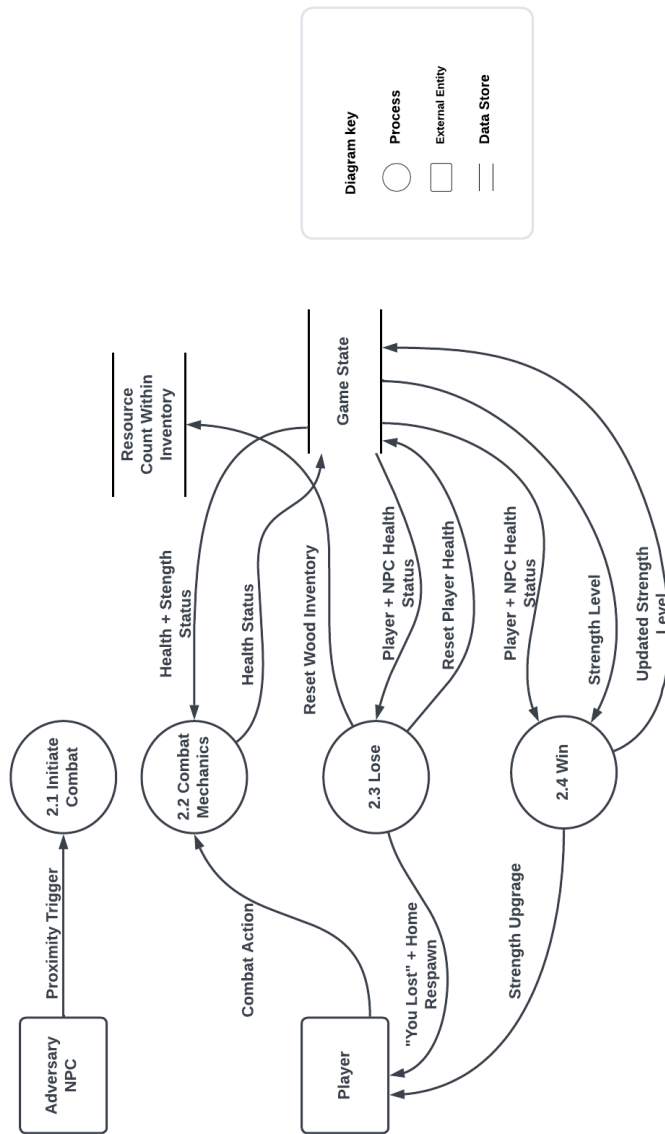


Figure 4: Data flow diagram at layer 1 describing the combat process.

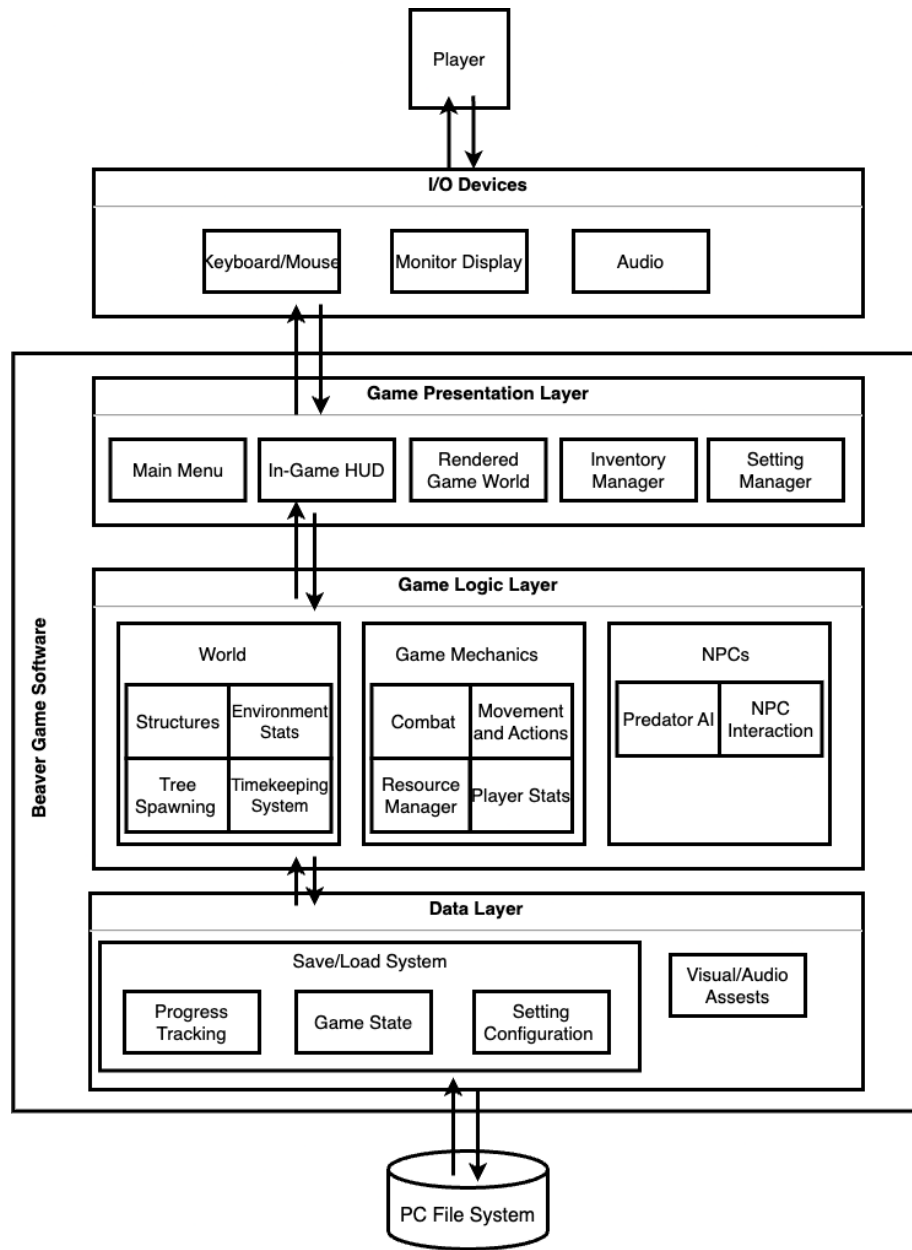


Figure 5: System Architecture Diagram: Presentation, Logic, and Data Layers Interaction.

F.2 Requirements Traceability Matrix

Table 6: Requirements Traceability Matrix

Requirement ID	Requirement Description	System Design Document Reference
FR1	This system shall allow the user to move the PC using keyboard input.	Section 5.5
FR2	This system shall provide fluid mechanics for walking, swimming, cutting down trees, and combat.	Section 4.3
FR3	The system will allow the player to hold an inventory of wood pieces.	Section 4.3
FR4	The system will allow the player to allocate the wood to dam or lodge structures.	Section 4.3
FR5	This system includes the PC stats of health and strength, which are improved through in-game milestones (ex. cutting down 20 trees, and upgrading a dam).	Section 4.3
FR6	The PC shall pass out if the health stat reaches zero during predator attacks. If the player passes, then the player will re-spawn at home base with lost inventory.	Section 4.3
FR7	If the health stat is improved, then the player has increased capacity to take predator attacks and has increased walking speed.	Section 4.3
FR8	If the health and strength stats are improved, then the amount of time it takes to cut down trees, inventory capacity, and attack damage is increased.	Section 4.3
FR9	If faced with a predator, the PC can interact with a bush to hide.	Section 4.3
FR10	The system shall provide a main menu where users can load their saved game.	Section 4.4
FR11	If the user is actively playing the game, a menu screen shall allow users to view inventory, game and character stats, configure audio settings, and the option to exit to the main menu without saving.	Section 7.2
FR12	If the in-game menu is open, then timekeeping is paused.	Section 7.2
FR13	A save station shall allow users to save game progress, load game progress, and/or exit to the main menu.	Section 7.2
FR14	The system has a main world map where the player resides and a forest map where players can be faced with combat and more resources.	Section 4.3
FR15	The graphics are dynamic based on the environment health stat. When a stat level is achieved, the land shall appear more fertile.	Section 4.3
FR16	The game clock should simulate a real-life day where 24 in-game hours is equal to 1 in-game day.	

Requirements Traceability Matrix (Continued)

Requirement ID	Requirement Description	System Design Document Reference
FR17	The clock facilitates the implementation of the chopping wood feature, where upgrades to strength reduce the time required to chop wood.	Section 7.2
FR18	The game shall allow players to observe the passage of time through an in-game clock display.	Section 7.2
FR19	Dam structures are to be built with a predetermined amount of wood pieces.	Section 4.3
FR20	The dam system will have a health status impacted by the flow of water.	Section 4.3
FR21	An upgraded dam structure increases water retention status and has more health status points.	Section 4.3
FR22	The water retention function will impact the overall world health status.	Section 4.3
FR23	Water will continuously impact dam health stat.	Section 4.3
FR24	A HUD provides the player with their health status.	Section 4.4
FR25	If a player is faced with a predator, the player shall be able to hide from or attack predators.	Section 6
FR26	Predators will be able to take idle walking status, attack mode, and search/stalking mode.	Section 6
FR27	The NPC predators will have in-range PC detection to trigger attack mode.	Section 6
NFR1	Reasonable load times are required.	Section 3.4
NFR2	A fluid and constant frame rate to provide a seamless graphics system.	Section 3.4
NFR3	Mindful memory usage to be considerate of end users' system requirements.	Section 4.4
NFR4	A UI and control mechanics that are intuitive and clear make movement, combat, and other game functions easy to use.	Section 7.2
NFR5	The source code must be organized and thoughtfully designed where adding additional features is not increasingly laborious.	Section 3.4

F.3 Data Dictionary

Not Applicable.

G Database Design Document

G.1 Assumptions/Constraints/Risks

Not Applicable.

G.1.1 Assumptions

Not Applicable.

G.1.2 Constraint

Not Applicable.

G.1.3 Risks

Not Applicable.

G.2 Design Decisions

Not Applicable.

G.2.1 Key Factors Influencing Design

Not Applicable.

G.2.2 Functional Design Decisions

Not Applicable.

G.2.3 Database Management System Decisions

Not Applicable.

G.2.4 Security and Privacy Design Decisions

Not Applicable.

G.2.5 Performance and Maintenance Design Decisions

Not Applicable.

G.3 Data Software Objects and Resultant Data Structures

Not Applicable.

G.3.1 Database Management System Files

Not Applicable.

G.4 Roles and Responsibilities

Not Applicable.

G.4.1 System Information

Not Applicable.

G.4.2 Database Management System Configuration

Not Applicable.

G.4.3 Database Support Software

Not Applicable.

G.4.4 Security and Privacy

Not Applicable.

G.5 Performance Monitoring and Database Efficiency

Not Applicable.

G.5.1 Operational Implications

Not Applicable.

G.5.2 Data Transfer Requirements

Not Applicable.

G.5.3 Data Formats

Not Applicable.

G.5.4 Backup and Recovery

Not Applicable.

H Interface Control Document

H.1 Interface Overview

Not Applicable.

H.2 Interface Controls

Not Applicable.

H.3 Functional Allocation

Not Applicable.

H.3.1 Data Transfer

Not Applicable.

H.3.2 Transactions

Not Applicable.

H.3.3 Security and Integrity

Not Applicable.

H.4 Detailed Interface Requirements

Not Applicable.

H.4.1 Requirements for ;Given Interface Name;

Not Applicable.

H.4.2 Assumptions

Not Applicable.

H.4.3 Technical Interface Requirements

Not Applicable.

H.4.4 General Processing Steps

Not Applicable.

H.4.5 Interface Processing Time Requirements

Not Applicable.

H.4.6 Message Format (or Record Layout) and Required Protocols

Not Applicable.

File Layout Not Applicable.

Data Assembly Characteristics Not Applicable.

Field/Element Definition Not Applicable.

Interface Initiation Not Applicable.

Flow Control Not Applicable.

H.4.7 Security Requirements

Not Applicable.

H.4.8 Requirements for "Given Interface Name"

Not Applicable.

H.5 Quality Assurance

Not Applicable.