4.1 Plan Introduction

Eve is a video game to be developed in Unity, using a pixel art style and a 2.5D rendering system. We are creating this game because we all have an interest in pursuing game development and would like to continue to work on our skills for the future. We will develop this game in Unity, and the pixel art will be done in Promotion NG, with more detailed art done in Procreate and music created in Ableton. We plan to have the Software Development Plan Document complete by Week 7, Software Design Description Document by Week 10, etc. (*fill in with actual milestones)

4.1.1 Project Deliverables

- Software Development Plan Document (Week 07)
- Software Design Description Document (Week 10-12)
- Critical Design Review (Week 12-14)
- User's Manual Final Updates(Week 12)
- Final Product Delivery (Week 16)
- (*fill in with specific things that need to be done)

4.2 Project Resources

4.2.1 Hardware Resources (Development)

- CPU: X64 architecture with SSE2 instruction set support OR Apple M1 or above (Apple silicon-based processors).
- Graphics API: DX10, DX11, and DX12-capable GPUs OR Metal-capable Intel and AMD GPUs.
- Hard Drive Space: ~5-10GB
- Display: 1920 X 1080
- RAM: 8 GB

4.2.2 Software Resources (Development)

- Operating System: Windows & Mac
- Operating Systems Version: Windows 7 (SP1+), Windows 10 and Windows 11, 64-bit versions only OR Mojave 10.14+ (Intel processors), Big Sur 11.0 (Apple silicon-based processors).
- Compiler/Engine: Unity
- Art Software/Graphics: Promotion NG & Procreate
- Audio Creation: Ableton
- Audio Engine: Fmod

4.2.3 Hardware Resources (Execution)

- CPU: X64 architecture with SSE2 instruction set support OR Apple M1 or above (Apple silicon-based processors).
- Graphics API: DX10, DX11, and DX12-capable GPUs OR Metal-capable Intel and AMD GPUs.
- Hard Drive Space: ~1-5GB

- Display: 1920 X 1080
- RAM: 6-8 GB

4.2.4 Software Resources (Execution)

- Operating System: Windows & Mac
- Operating Systems Version: Windows 7 (SP1+), Windows 10 and Windows 11, 64-bit versions only OR Mojave 10.14+ (Intel processors), Big Sur 11.0 (Apple silicon-based processors).

4.3 Project Organization

- Unity Project Management
    - Oversees the entire game development process, coordinating teams, timelines, resources, and ensuring the project stays on track and within scope.
    - Role: CJ
- Scripting for Story
    - Focuses on writing scripts that drive the narrative elements of the game, handling dialogues, branching storylines, and cutscene triggers to enhance the storytelling experience.
    - Role: Gray
- General Scripting
    - Implements core gameplay features and logic, handling tasks such as player controls, NPC behaviors, and game mechanics using C# and Unity's scripting API.
    - Role: Gray and CJ
- Interactive Art
    - Creates and integrates interactive visual assets, such as animations, particle systems, and effects, that respond dynamically to player inputs and game states.
    - Role Brisa
- Scripting for visual design
    - Writes scripts that control visual elements like shaders, lighting, post-processing, and special effects, enhancing the overall aesthetic and feel of the game.
    - Role: Brisa
- Mechanics
    - Designs and scripts the core game systems and mechanics, such as farming, managing the day/night cycle, movement, and more.
    - Role: Matt
- Interaction UI
    - Develops the user interface (UI) elements, focusing on intuitive design and scripting to handle input, feedback, and interaction between the player and the game systems.
    - Role: Matt
- General game play tools
    - Builds reusable tools and systems for the development team, such as level editors, asset managers, and debugging tools, to streamline the game creation process.
    - Role Matt

4.4 Project Schedule

4.4.1 PERT/GANTT Chart

- Link to Chart
    - https://docs.google.com/spreadsheets/d/1h-3aPss_imEJ1yIpYtgKFFn24p-yDgOpw7Lwemw5ZWo/edit?usp=sharing

4.4.2 Task/Resource Table

- UNLESS SPECIFIED hardware needed will be the same as what is listed in section 5.5.1

TASK	HARDWARE/SOFTWARE REQUIRED
Initial Unity 3D setup	- Software - Unity
Setup Tilemap	- Software - Unity
Create Layers/Tags for Collisions and interaction	- Software - Unity
Camera	- Software - Unity
Player Controller	- Software - Unity
Initial Game Manager	- Software - Unity
Farming Scripts	- Software - Unity
Difficulty Scaling	- Software - Unity
Task Manager	- Software - Unity
Music Manager	- Software - Unity - Fmod
Cut Scenes	- Software - Unity
Initial Pixel Draft	- Software - Promotion NG
Pixel Tilemap Draft	- Software - Promotion NG
Setting up 2.5D effects	- Software - Unity

Sound Effects	- Software - Ableton
Farming Sounds	- Software - Ableton
Music	- Software - Ableton
Main Theme	- Software - Ableton

5.1 Introduction

EVE shall be a video game designed in Unity using a 2.5D rendering system, which Unity natively supports. No outside libraries are intended for use, excluding the ones already supported by Unity and the ones that are readily accessible in its provided list of libraries. In terms of rendering, most gameplay will be in a pixel artstyle, with character closer-up interactions containing prerendered art of the individual. Components in terms of how the game should be organized would be separated into a player character manager, a world manager (for keeping track of time, physics, world interaction, etc.), and a plot manager (which would keep track of any character interactions and plot advancements to direct events in the world).

5.2 CSC Component Breakdown

5.2.1 Unity

5.2.1.1 Unity Engine
5.2.1.2 Graphical User Interface
5.2.1.3 Game Manager
5.2.1.4 Game Mechanics
5.2.1.5 Inventory Management
5.2.1.6 Time Management
5.2.1.7 Farming Tools
5.2.1.8 Quests and Objectives

5.2.2 Pro Motion NG

Will be used to create various assets and background images for the game, including most in-game portrayals of player and non-player characters, items, background images, menu and GUI elements, and interaction animations.

5.2.3 Procreate

Will be used to create more detailed art depictions of the various player characters and non-player characters in a traditional art style, as well as interaction UI and concept art

5.2.4 Ableton

Will be used to create various background music tracks, which will be played constantly throughout the game and will dynamically change based on situations detailed by the Unity engine

5.2.5 Fmod

Will be used to generate various in-game audio cues and sounds to give players feedback based upon their actions, or telegraph certain events based on situations detailed by the Unity engine

5.3 Functional Requirements by CSC

5.3.1 Unity

5.3.1.1 Unity- Engine

5.3.1.1 The Unity subsystem shall be used as the primary game engine for rendering environments.
5.3.1.2 The Unity subsystem shall handle physics-based interactions such as collisions, gravity, and movement.
5.3.1.3 The Unity subsystem shall provide support for scripting in C# to control player interactions, game mechanics, and UI elements.
5.3.1.4 The Unity subsystem will include built-in performance profiling tools to monitor framerate, memory usage, and processing overhead.
5.3.1.5 The Unity subsystem shall integrate with third-party libraries and plugins, including Fmod for audio and Pro Motion NG for art assets.
5.3.1.6 The Unity subsystem shall manage lighting and shadow rendering for both daytime and nighttime gameplay environments.
5.3.1.7 The Unity subsystem will support particle systems for rendering weather effects like rain, snow, and wind.
5.3.1.8 The Unity subsystem shall allow the import and management of pixel art assets created in Pro Motion NG and Procreate.

5.3.1.2 Graphical User Interface

The GUI is intended to be the forefront of the project, given that graphical feedback of a game system is imperative. This subsystem will manage the windows for a start menu, pause menu, quit and play buttons, responsive input requirements, and an options menu

5.3.1.2.1 The GUI subsystem shall display a window for the main application

5.3.1.2.2 The GUI subsystem shall display a window to provide details of operation (a "help" window)

5.3.1.2.3 The GUI subsystem shall display a window which will allow users to change input settings or make graphical adjustments (an "options" window)

5.3.1.2.4 The GUI subsystem shall include a button which will quit the application

5.3.1.2.5 The GUI subsystem shall include a menu bar that will pause all physics and player interaction when pressed, and display the aforementioned windows in 5.3.1.2 and 5.3.1.3, and the button in 5.3.1.4 (a "pause" menu)

5.3.1.2.6 The GUI subsystem shall react to mouse clicks on displayed buttons

5.3.1.2.7 The GUI subsystem shall react to keyboard inputs in conjunction with moving displayed game components

5.3.1.2.8 The GUI subsystem shall contain a button to start the main application

5.3.1.2.9 The GUI subsystem shall contain a window which will display the menu bar mentioned in 5.3.1.2.10, as well as the button in 5.3.1.1.8 (a "start menu")

5.3.1.2.11 The GUI subsystem shall contain a sprite which will be rendered in reaction to the buttons pressed as in 5.3.1.7 ("player movement")

5.3.1.2.12 The GUI subsystem shall display a menu which will display various sprites which are contained in the player's inventory

5.3.1.3 Game Manager

5.3.1.3.1 The GM subsystem shall contain references to all keyboard and mouse inputs in relation to how they would influence the GUI

5.3.1.3.2 The E key shall open the inventory by default

5.3.1.3.3 The Spacebar shall perform the jump action by default

5.3.1.3.4 ….

5.3.1.4 Game Mechanics

5.3.1.4.1 The game mechanics subsystem shall support planting various crops.
5.3.1.4.2 The game mechanics subsystem shall include a crop growth cycle based on in-game time, where crops progress through different growth stages.
5.3.1.4.3 The game mechanics subsystem will allow crops to wither if not harvested within a specified period.
5.3.1.4.4 The game mechanics subsystem shall support crop harvesting when they reach maturity.
5.3.1.4.5 The game mechanics subsystem shall support tilling the soil before planting crops.
5.3.1.4.6 The game mechanics subsystem shall include weather conditions, affecting crops and player actions (e.g., rainwater crops automatically).
5.3.1.4.7 The game mechanics subsystem shall support selling crops and animal products for in-game currency.

5.3.1.5 Inventory Management

5.3.1.5.1 The inventory subsystem shall allow the player to store seeds, crops, tools, and other items.
5.3.1.5.2 The inventory subsystem shall limit the number of items the player can carry at one time based on defined slots.
5.3.1.5.3 The inventory subsystem shall support the transfer of items between the player and storage containers such as chests or barns.
5.3.1.5.4 The inventory subsystem shall display the current item selection for use by the player.
5.3.1.5.5 The inventory subsystem shall allow players to stack identical items to save space.
5.3.1.5.6 The inventory subsystem shall categorize items for easy access (e.g., crops, seeds, tools).
5.3.1.5.7 The inventory subsystem shall allow for quick access to tools via a toolbar or hotkey system.
5.3.1.5.8 The inventory subsystem shall support a system for crafting new items using resources stored in the inventory.

5.3.1.6 Time Management

5.3.1.6.1 The time management subsystem shall simulate in-game time, progressing through day and night cycles.

5.3.1.6.2 The time management subsystem shall include seasons that change based on the passage of in-game days.

5.3.1.6.3 The time management subsystem shall include a calendar system where players can track upcoming events (e.g., festivals).

5.3.1.6.4 The time management subsystem shall include an alarm system where players can be notified of significant in-game events.

5.3.1.6.5 The time management subsystem shall include a system where in-game tasks (e.g., watering crops) must be completed within specific time windows.

5.3.1.7 Farming Tools

5.3.1.7.1 The tool subsystem shall allow the player to equip and use farming tools such as a hoe, watering can, scythe, and axe.
5.3.1.7.2 The tool subsystem shall allow tools to be upgraded using in-game materials.
5.3.1.7.3 The tool subsystem shall allow the use of tools to affect the environment (e.g., cutting trees, breaking rocks).
5.3.1.7.4 The tool subsystem shall include a repair mechanic for tools after they degrade with use.
5.3.1.7.5 The tool subsystem shall include a system for crafting new tools using resources collected by the player.
5.3.1.7.6 The tool subsystem will include a fishing tool, allowing the player to catch fish from bodies of water.
5.3.1.7.7 The tool subsystem shall include a mechanic for fertilizing crops to improve yield.

5.3.1.8 Quests and Objectives

5.3.1.8.1 The quest subsystem shall include a main storyline quest where players work to meet a quota provided by their employer.

5.3.1.8.2 The quest subsystem shall include side quests.

5.3.1.8.3 The quest subsystem shall reward players with in-game currency or special items for completing quests.

5.3.1.8.4 The quest subsystem shall include repeatable daily tasks such as watering crops and collecting resources.

5.3.1.8.5 The quest subsystem shall allow players to track their current objectives via an in-game journal.

5.3.2 Pro Motion NG - Pixel Art Software

5.3.2.1 The Pro Motion NG subsystem will be used for creating pixel art assets, including sprites, tiles, and UI elements.
5.3.2.2 The Pro Motion NG subsystem shall support exporting pixel art in compatible formats for Unity import.
5.3.2.3 The Pro Motion NG subsystem shall support creating multi-layered animations for player characters and NPCs.
5.3.2.4 The Pro Motion NG subsystem shall allow batch editing of pixel art assets to ensure consistency across game assets.
5.3.2.5 The Pro Motion NG subsystem shall support color palette management, allowing the creation of variations of the same sprite assets.

5.3.3 Procreate - Art Software

5.3.3.1 The Procreate subsystem will be used for creating concept art, illustrations, and hand-drawn textures.
5.3.3.2 The Procreate subsystem shall support exporting high-resolution textures for use in Unity.
5.3.3.3 The Procreate subsystem shall be used to create character portraits, used in player-NPC dialogue and interaction.
5.3.3.4 The Procreate subsystem shall allow the creation of UI art elements such as buttons, inventory icons, and menu backgrounds.
5.3.3.5 The Procreate subsystem shall support layer management and transparency for easy integration with Unity's sprite system.

5.3.4 Ableton - Music Software

5.3.4.1 The Ableton subsystem will be used for composing and producing background music for the game.
5.3.4.2 The Ableton subsystem will allow the creation of a dynamic soundtrack, adapting to different game conditions such as seasons, weather, or events (e.g., festival music, rainy day music).
5.3.4.3 The Ableton subsystem shall support the creation of looping ambient tracks for various farm locations (e.g., fields, barns, town).
5.3.4.4 The Ableton subsystem will support MIDI integration for composing adaptive music layers that can transition smoothly between game states.
5.3.4.5 The Ableton subsystem will allow the export of music tracks in formats compatible with the Fmod

audio engine for integration into the Unity engine.
5.3.4.6 The Ableton subsystem shall include sound effects creation, allowing for the design of distinct in-game sounds (e.g., harvesting, weather).
5.3.4.7 The Ableton subsystem shall allow collaboration between sound designers through the use of project files that can be shared and edited.

5.3.5 Fmod - Audio Engine

5.3.5.1 The Fmod subsystem will be used for managing all in-game audio, including music, sound effects, and ambient sounds.
5.3.5.2 The Fmod subsystem shall support dynamic audio mixing, adjusting volume levels based on game events or player actions.
5.3.5.3 The Fmod subsystem shall handle the implementation of adaptive music created in Ableton, ensuring smooth transitions between different music states.
5.3.5.4 The Fmod subsystem will support spatial audio for 3D sound effects, enhancing immersion by simulating the distance and direction of sounds.
5.3.5.5 The Fmod subsystem shall manage a library of sound effects triggered by player actions (e.g., using tools, and harvesting crops).
5.3.5.6 The Fmod subsystem shall include a real-time parameter control system, allowing game state changes (e.g., time of day, weather) to influence the soundscape.
5.3.5.7 The Fmod subsystem shall allow for the integration of environmental audio effects, such as wind, rain, and wildlife, which change dynamically with player location.
5.3.5.8 The Fmod subsystem will support multi-channel output for surround sound setups, enhancing the player’s auditory experience.

5.4 Performance Requirements

5.4.1 Cross-Platform Integration

The game will be playable on both 32-bit and 64-bit Windows and Mac computers.

5.4.2 Preloaded Scenes

This game will load scenes before gameplay to reduce lag.

5.4.3 Light Weight

This game will be limited to 50GB to reduce the storage space required to play.

5.4.4 Minimal Loading Times

This game shall not have an unmoving loading screen longer than 10 seconds.

5.4.5 Multiple Saves

This game will have the capacity to hold multiple saves from players without compromising the integrity of the performance quality.

5.4.6 High Visual Quality

This game will be optimized visually for differing screen sizes and refresh rates.

5.5 Project Environment Requirements

5.5.1 Development Environment Requirement

Following are the hardware requirements for Eve:

- CPU: X64 architecture with SSE2 instruction set support OR Apple M1 or above (Apple silicon-based processors).
- Graphics API: DX10, DX11, and DX12-capable GPUs OR Metal-capable Intel and AMD GPUs.
- Hard Drive Space: ~5-10GB
- Display: 1920 X 1080
- RAM: 8 GB
-

Following are the software requirements for Eve:

- Operating System: Windows & Mac
- Operating Systems Version: Windows 7 (SP1+), Windows 10 and Windows 11, 64-bit versions only OR Mojave 10.14+ (Intel processors), Big Sur 11.0 (Apple silicon-based processors).
- Compiler/Engine: Unity
- Art Software/Graphics: Promotion NG & Procreate
- Audio Creation: Ableton
- Audio Engine: Fmod

5.5.2 Execution Environment Requirement

Following are the hardware requirements for Eve:

- CPU: X64 architecture with SSE2 instruction set support OR Apple M1 or above (Apple silicon-based processors).
- Graphics API: DX10, DX11, and DX12-capable GPUs OR Metal-capable Intel and AMD GPUs.
- Hard Drive Space: ~1-5GB
- Display: 1920 X 1080
- RAM: 6-8 GB

Following are the software requirements for Eve:

- Operating System: Windows & Mac
- Operating Systems Version: Windows 7 (SP1+), Windows 10 and Windows 11, 64-bit versions only OR Mojave 10.14+ (Intel processors), Big Sur 11.0 (Apple silicon-based processors).

6.1. Introduction

This document presents the architecture and detailed design for the software system supporting *Eve*, a 2.5D pixel-art video game developed in Unity. *Eve* combines farming simulation and adventure mechanics, where players engage in activities like crop management, resource gathering, a turn-based combat system, and quest completion, all while navigating through a dynamic, visually immersive world. The game utilizes a variety of software tools and libraries, including Unity for core game mechanics, Pro Motion NG and Procreate for art assets, Ableton for music production, and Fmod for audio integration. The focus of this document is to detail the architecture, major software components, and interactions between these elements to support the functional requirements and deliver a cohesive gaming experience.

6.1.1 System Objectives

The objective of *Eve* is to deliver an engaging farming simulation and adventure experience that combines intuitive gameplay mechanics with rich audiovisual elements. The game is designed to offer players a variety of activities, including crop cultivation, inventory and resource management, and quest completion. In addition, *Eve* will now include a robust turn-based combat system, which will be integrated into the world. The user interface will be simple and accessible, enabling players of all experience levels to navigate menus, interact with in-game elements, and track their progress easily. Additionally, *Eve* aims to provide a visually cohesive world, enhanced by adaptive music and spatial audio, to create an immersive environment that responds dynamically to player actions and in-game events.

6.1.2 Hardware, Software, and Human Interfaces

6.1.2.1 Hardware Interfaces

- *CPU and GPU*: The system requires an x64 CPU with SSE2 instruction set support or an Apple M1 (or higher) for Mac. The GPU must support DirectX (DX10, DX11, DX12) on Windows or Metal-capable graphics for Mac. These components ensure the necessary processing power and rendering capability for Unity's 2.5D graphics and game physics.
- *Display*: The minimum supported resolution is 1920x1080, which is essential for displaying the detailed pixel art graphics created in Pro Motion NG and Procreate.
- *Input Devices*: The game will support keyboard and mouse inputs for player movement, menu navigation, and in-game actions. Keyboard hotkeys will allow quick access to inventory and tools, while mouse clicks will handle menu selections and other interactions.

6.1.2.2 Software Interfaces

- *Unity Engine (Version X)*: The core of *Eve*'s game development, Unity provides the base for rendering, physics, and input management. It interfaces with Pro Motion NG for pixel art, Procreate for detailed illustrations, and Fmod for audio. Unity uses C# for scripting game mechanics and user interactions.
- *Pro Motion NG (Version X)*: Used for creating pixel art assets, Pro Motion NG exports sprite sheets and animations in a format compatible with Unity. These assets are integrated directly into Unity's asset pipeline.
- *Procreate (Version X)*: Produces high-resolution character portraits, concept art, and UI elements. Procreate exports images in a format Unity can import, supporting layers and transparency for enhanced integration with the game's 2.5D environment.
- *Ableton (Version X)*: Used for composing and exporting background music, which is imported into Fmod. The music adapts dynamically based on in-game situations, such as weather or event changes.
- *Fmod (Version X)*: An audio engine that manages adaptive soundscapes and spatial audio. Fmod receives input from Unity and Ableton, adjusting music layers and sound effects based on player actions and game states. It integrates directly with Unity for real-time audio adjustments.

6.1.2.3 Human Interfaces

- *Graphical User Interface (GUI)*: Designed to be intuitive, the GUI provides menus for starting the game, accessing settings, and navigating inventory. A toolbar at the bottom of the screen displays tools and quick access items, and an in-game calendar tracks events and tasks. The GUI adapts to keyboard and mouse inputs, with visual feedback for each action.
- *User Controls*: The player will interact with the game through keyboard controls for movement, actions, and inventory access. A mouse-driven interface manages menu navigation and item selection.

6.2 Architectural Design

6.2.1 Major Software Components

The major software subsystems of *Eve* ensure that all functional requirements from section 5.2 are fully supported:

6.2.1.1 Unity Engine Subsystem
Core to *Eve*, Unity handles rendering, physics, and C# scripting, supporting gameplay features like movement, interaction, and environmental effects. It integrates third-party tools like Fmod and Pro Motion NG.

6.2.1.2 Graphical User Interface (GUI) Subsystem
The GUI manages all player interactions through menus and in-game controls, ensuring intuitive access to gameplay options, inventory, and settings.

6.2.1.3 Game Manager Subsystem
Oversees game state and input handling, coordinating player commands, global settings, and key gameplay functions.

6.2.1.4 Game Mechanics Subsystem
Manages core gameplay features such as farming cycles, weather impacts, and in-game economic activities, creating an engaging farming experience.

6.2.1.5 Inventory Management Subsystem
Organizes items, crafting, and hotkey access, allowing efficient item storage and use.

6.2.1.6 Time Management Subsystem
Controls day-night cycles, seasons, and an event calendar, coordinating time-sensitive gameplay elements.

6.2.1.7 Farming Tools Subsystem
Provides mechanics for using, upgrading, and repairing essential farming tools, supporting resource management.

6.2.1.8 Quests and Objectives Subsystem
Tracks main and side quests, rewarding players with in-game items and currency, with a journal system for progress.

6.2.1.9 Pro Motion NG Subsystem
Used for pixel art assets, including sprites, UI elements, and animations, ensuring visual consistency.

6.2.1.10 Procreate Subsystem
Produces detailed art assets and character portraits, enhancing interaction visuals.

6.2.1.11 Ableton Subsystem
Creates adaptive background music, supporting dynamic audio changes based on in-game events.

6.2.1.12 Fmod Subsystem
Manages in-game audio and spatial sound effects, creating an immersive auditory environment.

6.2.2 Major Software Interactions

The *Eve* application relies on integrated communication between major subsystems to deliver a seamless user experience. Below is an overview of these interactions:

6.2.2.1 Unity Engine with GUI, Game Manager, and Game Mechanics
The Unity Engine serves as the foundation, supporting interactions across GUI, Game Manager, and Game Mechanics. It renders graphics, processes user inputs, and handles physics-based interactions. The Game Manager communicates with Unity to process player commands (e.g., movement) and triggers gameplay elements within Game Mechanics, such as farming and inventory management.

6.2.2.2 Game Manager with GUI, Inventory Management, and Time Management
The Game Manager links core gameplay mechanics with the user interface. It uses the GUI subsystem to display player stats, inventory items, and time-based events. The Time Management subsystem provides in-game clock data to the Game Manager, which updates the GUI and synchronizes time-sensitive gameplay elements (e.g., day-night cycles).

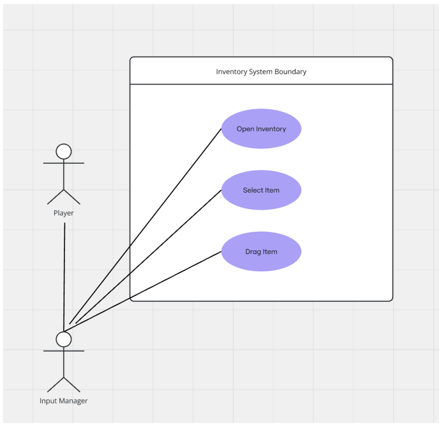
6.2.2.3 Inventory Management with GUI and Farming Tools
Inventory Management allows players to access and use tools or items through the GUI, directly affecting the Farming Tools subsystem. For example, a player selecting a hoe from the inventory initiates interactions with the Farming Tools subsystem, allowing for actions like tilling soil.

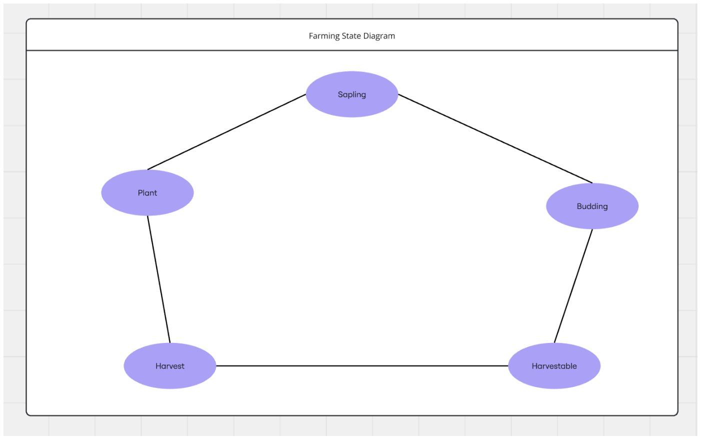
6.2.2.4 Fmod with Ableton and Unity
The Fmod audio engine processes sound effects and dynamic music created in Ableton. Unity communicates with Fmod to adjust audio layers and spatial sound in real-time based on player location and in-game events, such as weather or seasonal changes.

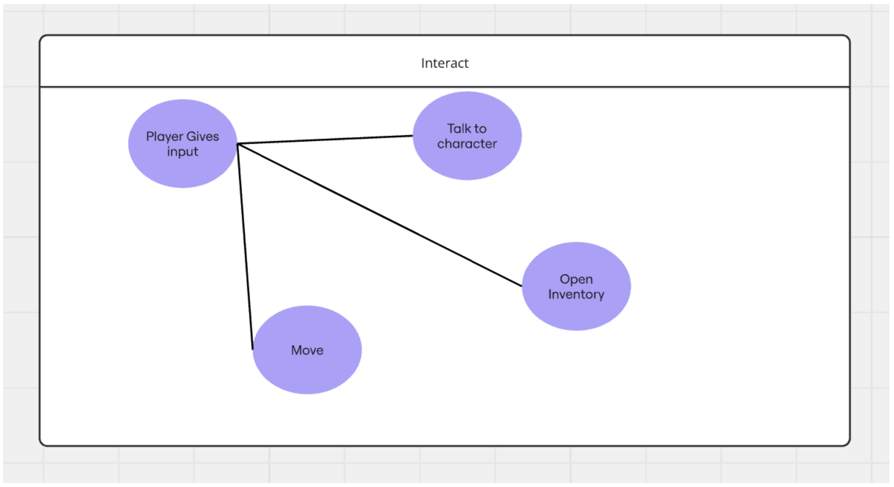
6.2.2.5 Pro Motion NG, Procreate, and Unity
Pro Motion NG and Procreate generate visual assets (sprites, backgrounds, etc.), which Unity imports and renders. These assets interact with Unity's rendering engine to display in-game visuals and animations that adapt to player actions or environmental changes.

These subsystem interactions ensure efficient data flow, responsive gameplay, and a cohesive audiovisual experience, with Unity acting as the central hub connecting the various components.

6.2.3 Architectural Design Diagrams







6.3. CSC and CSU Descriptions

The CSC components for this project include the UI, the interactable components, and the world environment. The CSU components for the UI include the overarching menu class, followed by object instances of various submenus. The CSU components of the interactable components are the abstract parent classes for interactable components, such as plants, computers, and NPCs, followed by the subclasses for specific behaviors of specific object instances. Finally, the CSU components of the world environment include the player character, the lighting and graphics systems, and plot/game progress trackers.

6.3.1 Class Descriptions for the Project

6.3.1.1 Menu Class

Purpose:
The Menu class manages the main interface, serving as the parent for all submenus in the UI. It defines the structure and functionality required to navigate the application's main menu and its components.

- Fields:
  - title (String): Stores the name of the menu displayed to the user.
  - menuItems (Array of MenuItem objects): Holds the list of selectable items in the menu.
  - isVisible (Boolean): Determines if the menu is currently displayed on the screen.
- Methods:
  - show(): Displays the menu on the screen.
  - hide(): Hides the menu.
  - selectItem(int index): Selects a menu item based on its index and executes associated actions.

6.3.1.2 Submenu Class

Purpose:
The Submenu class represents each submenu within the UI. It inherits from the Menu class and adds specific functionality for different submenus.

- Fields:
  - parentMenu (Menu object): Reference to the main menu from which this submenu was accessed.
  - options (Array of Option objects): Contains the selectable options specific to this submenu.
- Methods:
  - navigateBack(): Returns to the parent menu.
  - renderOptions(): Displays the options available in this submenu.

6.3.1.3 Interactable Class (Abstract)

Purpose:
The Interactable class serves as the base for all objects in the world that the player can interact with, such as plants, computers, and NPCs. It provides a template for subclassing specific types of interactables with distinct behaviors.

- Fields:
  - position (Coordinates): The location of the interactable object within the world.
  - isActive (Boolean): Determines if the object is currently interactable by the player.
- Methods:
  - interact(): Abstract method that must be implemented by subclasses to define the interaction behavior.

6.3.1.4 Plant Class

Purpose:
The Plant class is a subclass of Interactable, representing plants in the world that players can interact with.

- Fields:

    - type (String): Indicates the species or type of plant.
    - growthStage (Integer): The current growth stage of the plant.
- Methods:
    - water(): Increases the growth stage of the plant by one.
    - inspect(): Provides information on the plant's type and growth stage.

6.3.1.5 Computer Class

Purpose:
The Computer class is a subclass of Interactable that represents computers within the world environment. Players can access computers for specific functions in the game.

- Fields:
    - status (String): The current operational status of the computer (e.g., “Online”, “Offline”).
    - accessLevel (Integer): The required level for the player to interact with the computer.
- Methods:
    - login(): Initiates the login sequence if the player meets the access level requirements.
    - shutdown(): Powers down the computer.

6.3.1.6 NPC Class

Purpose:
The NPC class is a subclass of Interactable and represents non-playable characters that the player can engage with.

- Fields:
    - name (String): The name of the NPC.
    - dialogue (Array of String): Stores a series of dialog options that the NPC can use in interaction with the player.
- Methods:
    - talk(): Initiates dialogue with the player.
    - giveItem(): Provides an item to the player if certain conditions are met.

6.3.1.7 PlayerCharacter Class

Purpose:
The PlayerCharacter class represents the player within the world environment, tracking attributes and progress through the game.

- Fields:
    - health (Integer): Tracks the health status of the player.
    - inventory (Array of Item objects): Holds items collected by the player.
    - position (Coordinates): Current position of the player in the game world.
- Methods:
    - moveTo(Coordinates destination): Moves the player to a new location.

    - takeDamage(int amount): Reduces the player’s health by a specified amount.
    - addItem(Item item): Adds an item to the player's inventory.

6.3.1.8 LightingSystem Class

Purpose:
The LightingSystem class manages lighting within the game world, creating ambiance and affecting visibility.

- Fields:
    - currentLightLevel (Integer): Current light intensity in the game world.
    - timeOfDay (String): Tracks the time of day to adjust lighting accordingly.
- Methods:
    - adjustLighting(): Modifies light levels based on the time of day and game events.
    - toggleNightMode(): Switches lighting to a darker setting.

6.3.1.9 GraphicsSystem Class

Purpose:
The GraphicsSystem class oversees the graphical rendering of the world environment and UI.

- Fields:
    - resolution (Dimensions): Stores the screen resolution.
    - renderQueue (Array of Renderable objects): Contains objects scheduled for rendering.
- Methods:
    - renderScene(): Draws the scene on the screen.
    - updateResolution(Dimensions newResolution): Updates the screen resolution.

6.3.1.10 PlotTracker Class

Purpose:
The PlotTracker class keeps track of the player’s progress in the game’s storyline, managing completed quests and game objectives.

- Fields:
    - quests (Array of Quest objects): Lists all active and completed quests.
    - currentObjective (Objective): Tracks the current goal for the player.
- Methods:
    - updateObjective(Objective newObjective): Sets a new objective for the player.
    - completeQuest(Quest quest): Marks a quest as completed in the plot tracker.

6.3.2 Detailed Interface Descriptions

6.3.2.1 Menu-to-World Interface

Purpose:
This interface handles data transfer and control flow between the UI's menu system and the world environment, enabling the player to start or pause gameplay and adjust world settings from the UI.

- **Data Fields Transmitted:**
  - selectedOption (String): The specific option chosen by the player in the menu.
  - settingsData (Settings object): Holds any settings adjustments made within the UI, such as volume or brightness.
- **Control Actions:**
  - initiateGame(): Signals the world environment to begin or resume gameplay.
  - applySettings(): Passes adjusted settings to the world environment for immediate effect.

6.3.2.2 World-to-Interactable Interface

Purpose:
This interface provides the mechanism for the world environment to communicate with interactable objects (e.g., plants, computers, NPCs) as players explore and engage with the world.

- **Data Fields Transmitted:**
  - playerLocation (Coordinates): The current location of the player, used to determine nearby interactable objects.
  - interactionState (Boolean): Indicates whether the player is actively interacting with an object.
- **Control Actions:**
  - triggerInteraction(Interactable interactableObject): Notifies the interactable object that the player has initiated an interaction.
  - terminateInteraction(): Ends the interaction when the player steps away from the object.

6.3.2.3 UI-to-PlayerCharacter Interface

Purpose:
The UI-to-PlayerCharacter interface allows player-related actions (e.g., movement, inventory management) to be controlled directly through the UI, providing players with direct influence over their character in the world.

- **Data Fields Transmitted:**
  - inventoryCommand (String): Specifies the inventory action selected by the player, such as adding or removing items.
  - movementCommand (Direction): Indicates the player's desired movement direction.
- **Control Actions:**

    - updateInventory(Item item): Updates the player's inventory with additions or removals based on player input.
    - movePlayer(Direction direction): Commands the player character to move in the specified direction.

6.3.2.4 World-to-GraphicsSystem Interface

Purpose:
This interface enables the world environment to send visual data to the graphics system, which is responsible for rendering scenes on the screen.

- **Data Fields Transmitted:**
    - sceneData (Scene object): Contains the layout and visual details of the current environment scene.
    - lightingSettings (LightingData): Determines lighting and shading based on the time of day or events.
- **Control Actions:**
    - renderScene(Scene sceneData): Sends the current scene to be rendered in real time.
    - adjustLighting(LightingData lightingSettings): Alters the lighting in the scene for realistic effects.

6.3.2.5 PlayerCharacter-to-PlotTracker Interface

Purpose:
The PlayerCharacter-to-PlotTracker interface enables the tracking of player progress within the game's storyline, updating quest status and objectives based on player actions.

- **Data Fields Transmitted:**
    - completedQuests (Array of Quest objects): Lists quests that the player has completed.
    - currentObjective (Objective): The active objective that the player is currently pursuing.
- **Control Actions:**
    - updateProgress(Quest quest): Updates the plot tracker with newly completed quests.
    - setObjective(Objective newObjective): Changes the current objective based on player progress.

6.3.2.6 GraphicsSystem-to-UI Interface

Purpose:
This interface allows the graphics system to inform the UI of visual settings changes, such as resolution adjustments or mode switches, which can be reflected back in the UI settings.

- **Data Fields Transmitted:**
    - resolutionData (Dimensions): The current screen resolution as used in the graphical display.

    - renderMode (String): Specifies the rendering mode (e.g., high-performance, energy-saving).
- **Control Actions:**
    - displaySettings(Dimensions resolutionData): Updates the settings menu with the current resolution.
    - adjustMode(String renderMode): Reflects the graphics system's mode in the UI.

6.3.2.7 PlotTracker-to-UI Interface

Purpose:
The PlotTracker-to-UI interface provides the means for the player to track their progress and objectives through the UI, ensuring easy access to quest statuses and active objectives.

- **Data Fields Transmitted:**
    - activeQuests (Array of Quest objects): List of current quests the player can track in the UI.
    - objectiveDetails (String): Describes the player's immediate objective.
- **Control Actions:**
    - showQuestStatus(): Displays a summary of active and completed quests in the UI.
    - updateObjectiveDisplay(String objectiveDetails): Presents the latest objective details in the UI.

6.3.3 Detailed Data Structure Descriptions

6.3.3.1 MessagePacket Structure

Purpose:
The MessagePacket data structure is used to transmit information between the UI and the world environment, particularly for player inputs and settings adjustments. It encapsulates message details in a structured format to ensure efficient and clear communication across components.

- **Fields:**
    - messageType (String): Specifies the type of message being sent, such as "command" or "update."
    - payload (Map<String, Object>): Contains the data associated with the message, including commands or settings values.
    - timestamp (DateTime): Records the time at which the message was created for synchronization purposes.
- **Operations:**
    - encode(): Converts the message packet into a format suitable for transmission.
    - decode(): Interprets an incoming message packet back into the original data fields.

6.3.3.2 QuestLog Structure

Purpose:
The QuestLog structure tracks the player's active and completed quests, allowing the plot tracker to retrieve and update quest status for storyline progression. It is organized as a queue to keep quests in the order they are received and completed.

- **Fields:**
    - activeQuests (Queue of Quest objects): Stores the quests the player has not yet completed, with new quests added to the end of the queue.
    - completedQuests (List of Quest objects): Holds quests that the player has finished.
- **Operations:**
    - addQuest(Quest newQuest): Adds a new quest to the activeQuests queue.
    - completeQuest(): Removes the quest from activeQuests and adds it to completedQuests.
    - getActiveQuest(): Retrieves the current quest at the front of the activeQuests queue for display.

6.3.3.3 Inventory Structure

Purpose:
The Inventory data structure maintains a list of items collected by the player character, organizing them in a stack format for items that are used sequentially (e.g., potions, ammo) or in a list format for uniquely stored items.

- **Fields:**
    - stackableItems (Map<String, Stack of Item objects>): Contains items that the player may collect in multiples, organized by item type.
    - uniqueItems (List of Item objects): Holds one-of-a-kind items that the player possesses.
- **Operations:**
    - addItem(Item item): Adds an item to the appropriate collection (stackable or unique).
    - removeItem(Item item): Removes an item from the inventory, either by popping from the stack or removing from the list.
    - getItemDetails(Item item): Retrieves details of an item for display or use.

6.3.3.4 SettingsData Structure

Purpose:
The SettingsData structure organizes the application's adjustable settings, such as audio, video, and gameplay preferences. It is passed between the UI and the world environment, allowing for immediate implementation of setting adjustments.

- **Fields:**
    - audioSettings (Map<String, Integer>): Stores audio-related settings, such as volume levels for music, effects, and dialogue.
    - videoSettings (Map<String, Object>): Contains graphical settings, including resolution and display mode.

    - gameplaySettings (Map<String, Boolean>): Tracks gameplay-specific preferences like difficulty and hints.
- **Operations:**
    - applyChanges(): Sends updated settings to the relevant subsystems.
    - resetToDefault(): Resets all settings fields to their default values.

6.3.3.5 RenderQueue Structure

Purpose:
The RenderQueue structure in the graphics system is used to manage the sequence of objects to be rendered in each frame. It follows a priority queue model, where objects are rendered based on their priority level, ensuring key visual elements appear in the correct order.

- **Fields:**
    - queue (PriorityQueue of Renderable objects): Contains renderable objects ordered by priority, with the highest-priority objects rendered first.
- **Operations:**
    - enqueue(Renderable object): Adds an object to the render queue based on its priority level.
    - dequeue(): Removes and returns the highest-priority object for rendering.
    - clearQueue(): Empties the queue, typically used when changing scenes.

6.3.3.6 DialogueTree Structure

Purpose:
The DialogueTree structure organizes the conversation flow between the player and NPCs, allowing for structured dialogue with branching paths based on player choices. It is implemented as a tree, where each node represents a dialogue option with branches for possible responses.

- **Fields:**
    - dialogueNodes (Map<Integer, DialogueNode>): Stores each dialogue option and its potential responses.
    - currentNode (DialogueNode): The current position in the conversation, tracking the player's choice and response path.
- **Operations:**
    - advanceToNextNode(int choice): Moves the conversation to the next node based on the player's selection.
    - resetDialogue(): Resets the dialogue to the beginning for future interactions.
    - getAvailableChoices(): Retrieves the list of possible responses available at the current node.

6.3.3.7 SceneData Structure

Purpose:
The SceneData structure provides a comprehensive representation of the current game scene, containing

all elements necessary for rendering by the graphics system. It includes environmental, lighting, and object data.

- **Fields:**
  - environmentDetails (EnvironmentData): Contains data related to the scene's setting, including background textures and terrain.
  - lightingSettings (LightingData): Specifies light sources, colors, and intensities.
  - objectList (List of Renderable objects): A collection of all objects present in the scene that must be rendered.
- **Operations:**
  - updateScene(): Refreshes the scene data to reflect any changes.
  - getRenderableObjects(): Retrieves the list of objects to be drawn on screen for the current frame.

6.3.1.11 CombatSystem Class

Purpose:
The CombatSystem class manages turn-based combat encounters, including player actions, enemy behavior, and deck-based mechanics.

- **Fields:**
  - playerDeck (Deck): The deck of cards the player uses in combat.
  - enemy (Enemy): The current enemy engaged in combat.
  - turnOrder (Queue of Combatant): A queue determining the turn order of combatants (player and enemies).
  - battleState (String): The current state of the combat (e.g., "Player Turn", "Enemy Turn", "Victory", "Defeat").
- **Methods:**
  - startBattle(Enemy enemy): Initializes a new battle with the given enemy.
  - playCard(Card card): Executes the effect of a selected cards
  - processEnemyTurn(): Handles enemy actions during their turn.

6.3.1.12 Deck Class

Purpose:
The Deck class represents the collection of cards the player can draw from in combat.

- **Fields:**
  - cards (List of Card): The current list of available cards in the deck.
  - discardPile (List of Card): Cards that have been played and are set aside until reshuffled. drawPile (List of Card): The pile from which cards are drawn at the start of turns.
- **Methods:**
  - shuffle(): Randomizes the order of the draw pile.
  - drawCard(): Draws a single card from the draw pile.

- discardCard(Card card): Moves a played card to the discard pile.
- resetDeck(): Shuffles the discard pile back into the draw pile when empty.

6.3.1.13 Card Class (Abstract)

Purpose:
The Card class serves as the base for all cards in the game, defining shared attributes and behaviors.

- **Fields:**
  - name (String): The name of the card.
  - description (String): A short description of the card's effect.
  - energyCost (Integer): The amount of energy required to play the card.
  - rarity (String): The card's rarity level (e.g., Common, Rare, Legendary).
- **Methods:**
  - play(Combatant target): Abstract method to execute the card's effect on a given target.

6.3.1.14 AttackCard Class (Subclass of Card)

Purpose:
Represents attack-based cards that deal damage to enemies.

- **Fields:**
  - damage (Integer): The amount of damage the attack deals.
- **Methods:**
  - play(Combatant target): Reduces the target's health by the card's damage value.

6.3.1.15 DefenseCard Class (Subclass of Card)

Purpose:
Represents cards that provide defensive effects, such as armor or shields.

- **Fields:**
  - blockAmount (Integer): The amount of damage blocked.
- **Methods:**
  - play(Combatant target): Increases the target's shield by the block amount.

6.3.1.16 UtilityCard Class (Subclass of Card)

Purpose:
Represents special effect cards that manipulate the deck, status effects, or battle conditions.

- **Fields:**
  - effect (String): Describes the special effect applied when played.
- **Methods:**
  - play(Combatant target): Executes the special effect.

6.3.1.17 Combatant Class (Abstract)

Purpose:
Represents any entity that participates in combat, including the player and enemies.

- **Fields:**
  - health (Integer): The current health of the combatant.
  - maxHealth (Integer): The maximum health the combatant can have.
  - shield (Integer): Temporary shield that absorbs damage.
  - statusEffects (List of StatusEffect): Active status effects on the combatant.
- **Methods:**
  - takeDamage(int amount): Reduces health, accounting for shields if present.
  - applyStatus(StatusEffect effect): Applies a status effect to the combatant.
  - clearStatusEffects(): Removes all active status effects.

6.3.1.18 PlayerCharacter (Extended for Combat)

Additional Fields:

- energy (Integer): The amount of energy available to play cards each turn.
  combatDeck (Deck): The current deck used for battles.

Additional Methods:

- gainEnergy(int amount): Increases energy for playing cards.
- useCard(Card card): Plays a card from hand and deducts energy cost.

6.3.1.19 Enemy Class (Subclass of Combatant)

Purpose:
Represents enemies that the player can fight in combat.

- **Fields:**
    - attackPattern (List of EnemyAction): A sequence of actions the enemy performs each turn.
    - currentAction (EnemyAction): The action the enemy will take on its turn.
- **Methods:**
    - determineNextAction(): Selects the next action based on attack pattern.
    - executeAction(): Performs the current action against the player.

6.3.1.20 EnemyAction Class

Purpose:
Represents a specific action an enemy can take in combat.

- **Fields:**
    - type (String): The type of action (e.g., "Attack", "Defend", "Buff").
    - value (Integer): The effect value (e.g., damage dealt, shield gained).
- **Methods:**
    - execute(Combatant target): Applies the action's effect to the target.

6.3.1.21 StatusEffect Class

Purpose:
Represents temporary effects applied to combatants, such as buffs or debuffs.

- **Fields:**
    - effectType (String): The type of effect (e.g., "Poison", "Strength Buff").
    - duration (Integer): The number of turns the effect remains active.
- **Methods:**
    - applyEffect(Combatant target): Applies the effect at the start of the target's turn.
    - decrementDuration(): Reduces the duration each turn.

6.3.1.22 Item Class

Purpose:
Represents items that can modify the player's deck or affect combat.

- **Fields:**
    - name (String): The name of the item.
    - effect (String): A description of its impact.

- **Methods:**
    - use(): Activates the item’s effect.

6.3.1.23 DeckModifierItem Class (Subclass of Item)

Purpose:
Represents items that modify the player’s deck by adding or removing cards.

- **Fields:**
    - cardEffect (String): The modification applied (e.g., “Add Card”, “Remove Card”).
    - targetCard (Card): The specific card affected.
- **Methods:**
    - applyToDeck(Deck deck): Alters the player’s deck based on the item’s effect.

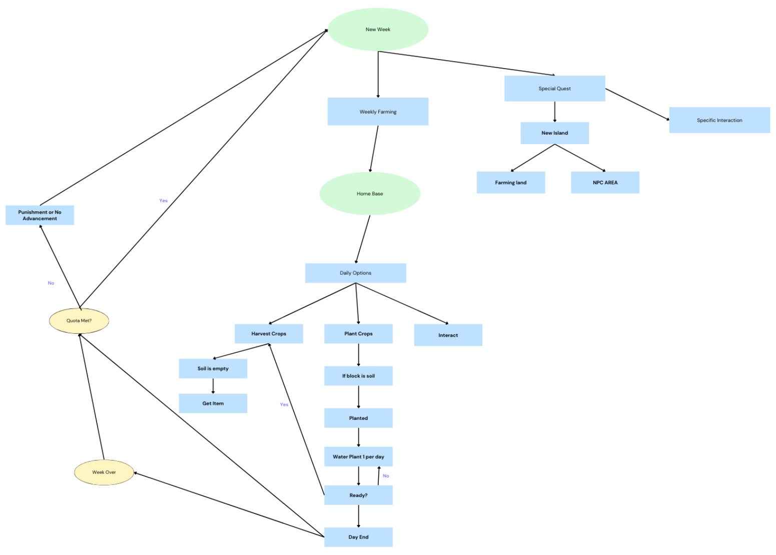
6.3.1.24 PotionItem Class (Subclass of Item)

Purpose:
Represents consumable items that provide temporary buffs in combat.

- **Fields:**
    - statusEffect (StatusEffect): The buff or debuff applied.
- **Methods:**
    - consume(PlayerCharacter player): Applies the effect to the player.

6.3.4 Detailed Design Diagrams

Game flow diagram



6.4 Database Design and Description

There is no database in this project