CSCI 265 — Decompositional Design

Team Badass — All Who Wander

Executive Summary

Document Overview

This document contains the implementation strategy for the All Who Wander project, and will provide written and visual maps of its architecture. Contained within is a high-level overview of this architecture including an overview of any programming paradigms or frameworks used, a mid-level overview of the architecture's various components and how they fit together, and a low level description of each individual component.

The contents of this document will be highly technical — as such, this document's intended use is as a guide for developers. For other descriptions of the project designed for other purposes, see this project's Software Requirements Specification document.

Gameplay Overview

All Who Wander is a topdown sword-and-shield adventure game built in Unity, and designed to replicate the 8-bit experience of its predecessors. The game's central features will be its multifaceted combat system and progression through its world and story. Auxiliary to these mechanics will be interaction with NPCs and in-world objects, a multitude of items with various uses, and a level-grinding system.

Like most games of its nature, All Who Wander will demand progressing from level to level to accomplish objectives, defeating hostile NPCs and collecting treasure along the way. The central objective will be to defeat the boss-creature at the center of each of the world's dungeons.

The combat system in All Who Wander will support the use of multiple weapons (a sword, a spear, a shield, a bow and arrow, and numerous offensive magic spells). These weapons will be collected from throughout the landscape and held within an inventory. Weapons will be equippable to hotkeys such that many attacks are available to the player during combat. Spells, themselves just projectile weapons of a certain type, will be collected at key locations in the game's world, often as a reward for completing an objective. Buffs and other combat-modifying effects may be implemented, time permitting.

In traversing the world from objective to objective, the player will encounter numerous hostile enemies to be fought, puzzles and scenery embedded within the landscape, and NPCs with whom to interact. Side quests such as trading sequences or other ways of rendering assistance to the world's inhabitants may be implemented, time permitting.

Story progression will occur as a result of interaction with specific NPCs or world-objects, or else the defeat of specific hostile creatures, in sequence.

Requirements Summary

Recontextualized in terms of implementable systems, the above specific requirements arise:

- **A tile-based gameworld** built from layer upon layer of square tile sprites.
    - **Layers of impassable tiles** via Unity's built-in collider system.
    - **Layers of obscuring tiles** which block line-of-sight to the enemy AI.
    - **Layers of scenery tiles** to add texture to the world.
    - **Layers of path tiles** to guide the player from region to region.
    - **Offshoots from the main path** wherein to hide world-texturizing side content.
    - **Multiple environments** to characterize the world's various regions.
    - **Multiple dungeons** each containing mini-levels, and a boss-creature.
- **A player control system** which responds to user input, allowing for:
    - **Player movement** in all four cardinal directions.
    - **Use of weapons, spells, and items** registered to hotkeys.
- **Interactable non-playable characters and objects** with various functions, including:
    - **Dialogue** both relevant and irrelevant to the story.
    - **Items given** to the player.
    - **Items taken** from the player.
    - **Story-Progression triggers** upon interaction.
- **Items with various uses, including:**
    - **Healing** the player.
    - **Temporary increases to combat stats** aka "buffs."
    - **Unlocking off-limits areas.**
    - **Other uses.**
- **Treasure chests** scattered throughout the world, containing items.
- **Loot-drops** resulting from defeated enemies, containing items.
- **Enemies with whom to fight,** spawned via a spawning system.
- **A multifaceted combat system with:**
    - **Melee weapons** of various types (sword, spear, shield).
    - **A ranged weapon** (a bow and arrow) with different types of arrow.
    - **Offensive magic spells** which operate like ranged weapons.
    - **Buffs and debuffs** which modify player combat stats.
- **An inventory** with which to hold items collected.
  A slot based hot-key system with which
- **A story** progressed through via interaction with NPCs, in-world objects, the defeating of enemies, and the completion dungeons.
- **A heads up display** (Overlay) which shows important information about the player.

- **A pause screen/main menu** which allows the user to load a game/create a new gave/save a game.

High-Level Implementation

Unity in a Nutshell

To understand the game's architecture fully, a general understanding of Unity's own architecture is required.

Unity is primarily a 3D rendering and physics engine used for game development. Auxiliary to this are a vast collection of tools for handling input, simulating physics upon objects (such as collisions or gravity), dealing with logic in 3D space, creating or importing content, and so on.

Unity maintains an object-model of interconnected in-world entities known as GameObjects, or objects hereafter, organized in a tree according to parent-child relationships. Everything which exists in a Unity game world, from characters and collectable items to lights and special effects, are objects of some kind. Objects are themselves entirely generic — the characterizing properties which differentiate objects and give them their functionality are provided by modular scripts which are attached to them known as components.

To be rendered, built-in components such as Transforms, Filters and Renderers are attached to objects to define their positions, shapes and appearances (respectively). Objects with these components participate in the engine's rendering system, and are drawn every frame.

To participate in physics, built-in components known as Rigid Bodies and Colliders are attached to objects to govern their interactions with forces (such as gravity) and collisions with other objects. Objects with these components participate in the physics system, and various physical interaction algorithms are applied to them each frame.

Many built-in components exist with myriad purposes such an input mapping to callbacks, exotic physics, drawing and animation, and more.

It is with custom components that a Unity game is extended beyond basic functionality. Custom components can be used to attach data-members to objects (such as a player's health or experience), or else to attach the functionality which gives objects their behaviour. Often, a traditional class-based object hierarchy is implemented and attached to game objects. This hierarchy defines everything about the objects in the gameworld one would expect to find, from a player's equipped items to callbacks executed when an object is interacted with by the player.

Once assembled, objects can be saved as “prefabs.” These prefabs are essentially prebuilt objects which can be spawned as needed (much the same way that classes can be used to create objects in code).

Entity Component Systems

All Who Wander will implement an Entity-Component-System (ECS) style architecture. This is a logical choice given Unity’s own component-based architecture. Moreover, ECS architectures are an effective conceptual framework with which to segregate model from controller, thereby promoting unspagettified code and decoupled functionality.

An Entity-Component-System is a programming paradigm which seeks to separate model from controller by prohibiting the affixation of functionality to objects in the model. In an ECS, all objects are generic Entities, defined not by their type and inheritance but rather by packets of data attached to them known as Components. Components contain only data. Entities are acted upon by Systems, which search for and act upon all Entities in the world who possess those components required by that system. In this way, the code required to execute some process, such as combat, is unified in a single location instead of strewn about numerous classes.

There are numerous benefits and some drawbacks to this paradigm, though, these will not be discussed within this document.

It should be noted that while Unity already imposes an Entity-Component system upon games built within it, Systems will require some careful imagining to implement faithfully to the paradigm.

A Standard Topdown

A standard topdown game involves moving the player through a game-world, interacting with objects and NPCs, engaging in combat, using items, and progressing through a story.

Implementationally, this means the core functionality of a topdown game comes down to input handling mapped to player movement and actions, animating the player and other objects, and triggering interactions between objects. Also of importance are creation of the game-world to start with, the population of this world with interactive entities, and story progression.

Concluding this may seem intuitive, but it bears stating because from this it can be seen that a topdown relies on a simple set of technologies layered on top of each other.

Specifically, the main technologies required for a topdown game are:

- **Input handling which maps user input to player actions.** In this case, a dynamic mapping system will be required which lets the user choose which actions to map to which keys.
- **A rendering engine** with which to draw the world.
- **A physics engine** with which to detect interactions (collisions).
- **Animation control** which automatically executes the animation of game-world entities based on their internal state (itself toggled by scripts).
- **A level editor** for building a game-world as a composite from many smaller sprites.
- **A scripting language** with which to compile the functionality and data that define objects.
- **An object model** which defines the game world (either inheritance based or compositional).

Of course, the above relies upon a slew of lower-level technologies. Thankfully, nearly all of these technologies are provided automatically by Unity.

Architectural Overview

Abstract View

The All Who Wander architecture could be summarized as an interconnected model of objects to which data-carrying components and functionality-carrying systems are attached.

Control over the player will be mediated by built-in input handling components. Numerous types of interaction between objects will be mediated by hit-boxes which trigger callbacks on systems. Animation will be controlled by scripts attached to objects, triggered by interactions, and then handled automatically by Unity's animation system. The game world itself will be assembled by hand using Unity's built-in level editor. The world will be populated by hand-assembled objects saved as prefabs, and spawned on demand.

Other mechanics, such as story progression among others, will likely rely upon entirely hand built components and systems.

Within the raw code, native GameObjects will take the place of Entities to which Components and Systems are attached. Components of all types will define each object, its properties, it's state, and so on. Systems will be called every frame or else in response to events, reading from and writing back to Components as needed.

Concrete View

Below is a summary of the game's actual, concrete functionality. For brevity, what follows does not discuss the component-structure of the various entities at play — rather, the overall functionality in terms of *systems* is described.

The game's architecture begins with input handling which maps player input to player control scripts: the Player-Movement and Player-Action Systems. These systems are responsible for moving the player through the game's world, and triggering interactions between the player and other objects.

Interactions between objects are mediated by Unity's collision system, which automatically fires a callback in response to a collision between objects. Since players expect interactions to take place at the strike of a key, the Player-Action system's role is to briefly enable and then disable so-called "interaction colliders" on key strikes. Each weapon, spell, and the player itself has its own interaction collider. These colliders automatically trigger callbacks on systems when triggered by Unity's physics system.

Items, Non-Living Objects, and NPCs all map their interaction colliders to their own interaction systems — the Player-Item-Interaction system, the Player-Object-Interaction system, and the Player-NPC-Interaction system respectively.

Combat occurs when input is mapped to an attack action. A collider is briefly enabled for the given weapon. Should an enemy enter that weapon's collider while enabled, the Player-Attack system is invoked. This process occurs in reverse for enemy attacks upon the player (using the Enemy-Attack system). Enemy actions are controlled by the Enemy-AI system, called every frame. When triggered, the attack systems gather the combat stats related to the player and the enemy, and crunch the requisite mathematics to deal damage to one or the other.

Two systems fire when enemies are killed: the Loot-System and the Leveling System. These systems spawn items for the player to collect, and increment the player's base stats.

Some entities (enemies, NPCs, objects) may have story-triggers attached to them as components. The above systems will be configured to recognize these triggers and forward the interaction to a Story-Progression-System. This system changes the dialogue trees of NPCs or else modulates the landscape as the story progresses.

Other systems with various functions will exist, and they will be discussed as needed throughout the rest of the document. Of note is the UI-Control-System: this system consumes input and modulates visual UI-elements in response to it. This system will have various other functionalities as well.

Summary View

A complete list of the entities, components, and systems required can be found in the Low-Level Implementation section of this document. Speaking generally however, All Who Wander will implement the following systems:

- **The Player Movement System,** called in response to input to move the player and toggle movement animations.
- **The Player Action System,** called in response to input to enable a weapon's hitbox and toggle attack animations.
- **The Player Attack System,** called when an enemy enters a weapon's hitbox in order to deal damage to that enemy.
- **The Enemy AI System,** called each frame to govern enemy behaviour including movement, and toggling of the enemy's attack-collider.
- **The Enemy Attack System,** called when the player enters an enemy's attack-collider in order to deal damage to the player.
- **The Loot Drop System,** called when an enemy is destroyed in order to spawn items to be collected by the player.
- **The Item Pickup System,** called when the player interacts with an item in order to add it to their inventory.
- **The Player-Object Interaction System,** called when the player interacts with a gameworld object. This system may have multiple effects.
- **The Player-NPC Interaction System,** called when the player interacts with an NPC. This system may have multiple effects.
- **The Weapon-Object Interaction System,** which works the same as the Player-Object interaction system, except that interactions are triggered when a specific weapon is used on an object.
- **The Story Progression System,** called by other systems in reaction to various interactions.
- **The Spawning System,** called when the player moves in order to spawn new enemies and treasure.
- **The Leveling System,** called whenever the player defeats an enemy in order to increment their level and combat stats.
- **The UI Control System,** called in response to user input when the game is paused. Various uses, discussed in full within its own section below.

The above list is non-exhaustive, and will likely change in the future. A map of these systems and the data they transceive is shown below. Some systems may not be shown.

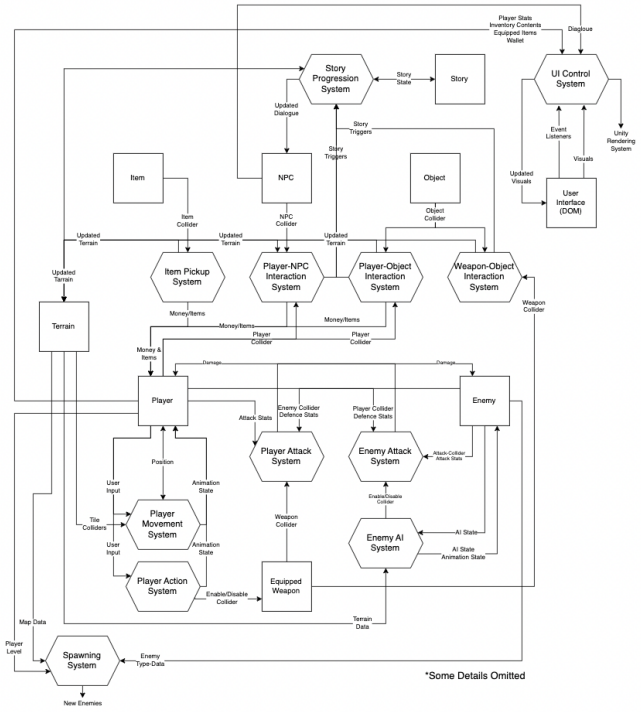


Figure 1 Architectural Overview Diagram

Mid-Level Implementation

Player Movement

Player movement is mediated by a single system — the Player-Movement-System — which reads data from the player and from the terrain.

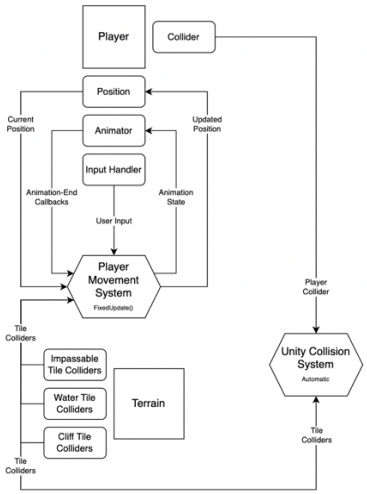


Figure 2 The Player Movement System

The Player-Movement-System is called in response to OnMove() events, triggered by player input. These events correspond to directional arrows on the keyboard, and the joystick on a game controller.

The Player-Movement-System consumes a player's current position and current animation state, and then directly modifies these values. The updated values are then written back to the player's Position and Animation-State components.

In order to prevent the player from walking through impassable tiles, the Player-Movement system performs a ray casting operation to check whether the position it intends to move to is impassable. With a ray casting operation, a call to Unity's collision system is made to check if the desired position falls within the collider of an impassable tile. If this happens, no movement occurs. In the interest of more natural movement, independent ray casts are performed in both the *x* and *y* directions.

Should climbing and swimming mechanics be implemented, further ray casting will be performed to check whether movement will place the player into water or onto a cliff. If a ray cast determines that motion will place the player in the water from the ground, the player's animation state is changed to "swimming", and they are animated accordingly. The same is true for climbing.

Interactions

An "interaction" refers to a collision between the player and some other entity such as an NPC or a gameworld object. Interactions are mediated by a different system for each type of interaction, specifically: Player-Object, Player-NPC, Player-Item, and Weapon-Object interactions.

All of these interactions proceed along similar lines, with some minor deviations. One such interaction is pictured below. For the diagrams depicting each, see the appropriate sections in the Systems subsection of the Low-Level Implementation section near the bottom of this document.

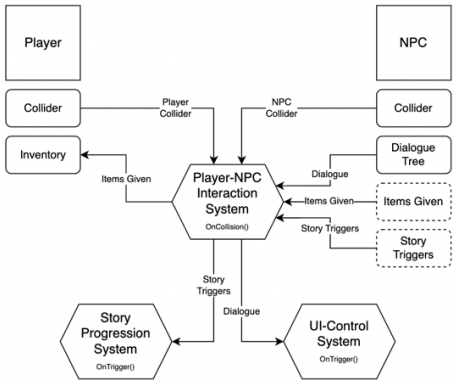


Figure 3 Player-NPC Interaction System

Players have two colliders attached to them: a main collider used by the combat system, and a second collider used only in interactions. This second collider is disabled by default, and is enabled briefly when the player strikes the “interact” key.

To detect an interaction between the player and some other object, the player’s interaction collider is briefly enabled by striking the “interact” key. If an NPC or world object is within the interaction collider, the appropriate system is called. Some world objects demand interacting with them via a weapon instead of the generic interaction collider. In this case, a weapon’s collider takes the place of the generic collider.

Interaction systems may make further calls to other systems. Interacting with an NPC will forward dialogue to the UI control system. Some interactions may trigger story progression.

Item Collection

Item collection is mediated by the Item-Pickup system, which reads data from the player and from the items being interacted with. This system works very similarly to the interaction systems outlined above.

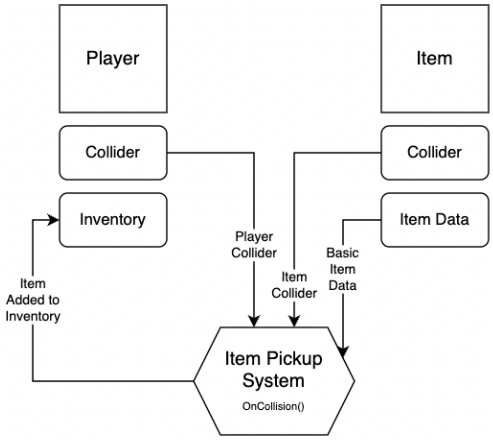


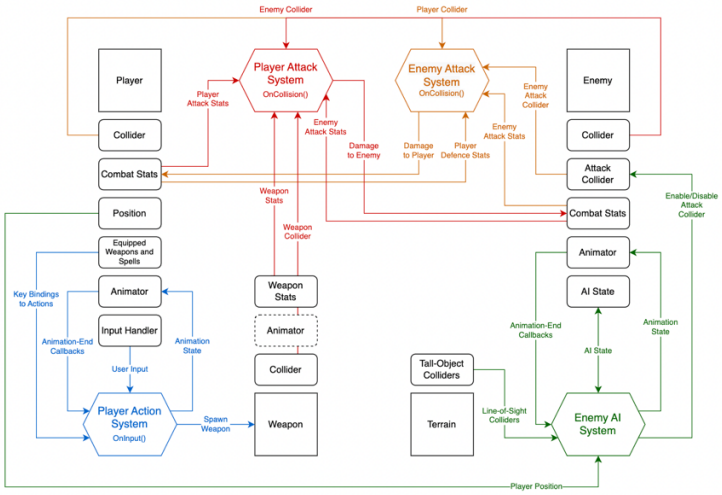
Figure 4 Item Pickup System

Items are entities strewn across the landscape, or else dropped by enemies when they are defeated. Each item has a collider, enabled at all times. When the player collides with an item, the item is “collected.”

To collect an item, the item itself is removed from the game world. Information about that item is saved, and transferred to the player’s Inventory component. This component tracks the items held by the player. The item may then be used by the player by assigning it to an action.

Combat

Combat is mediated by four interconnected systems reading from the player, enemy, and terrain entities. These systems are the Player-Action, Player-Attack, Enemy-AI, and Enemy-Attack systems.



Each of these systems will be discussed in fuller detail within the Low-Level Implementation section of this document.

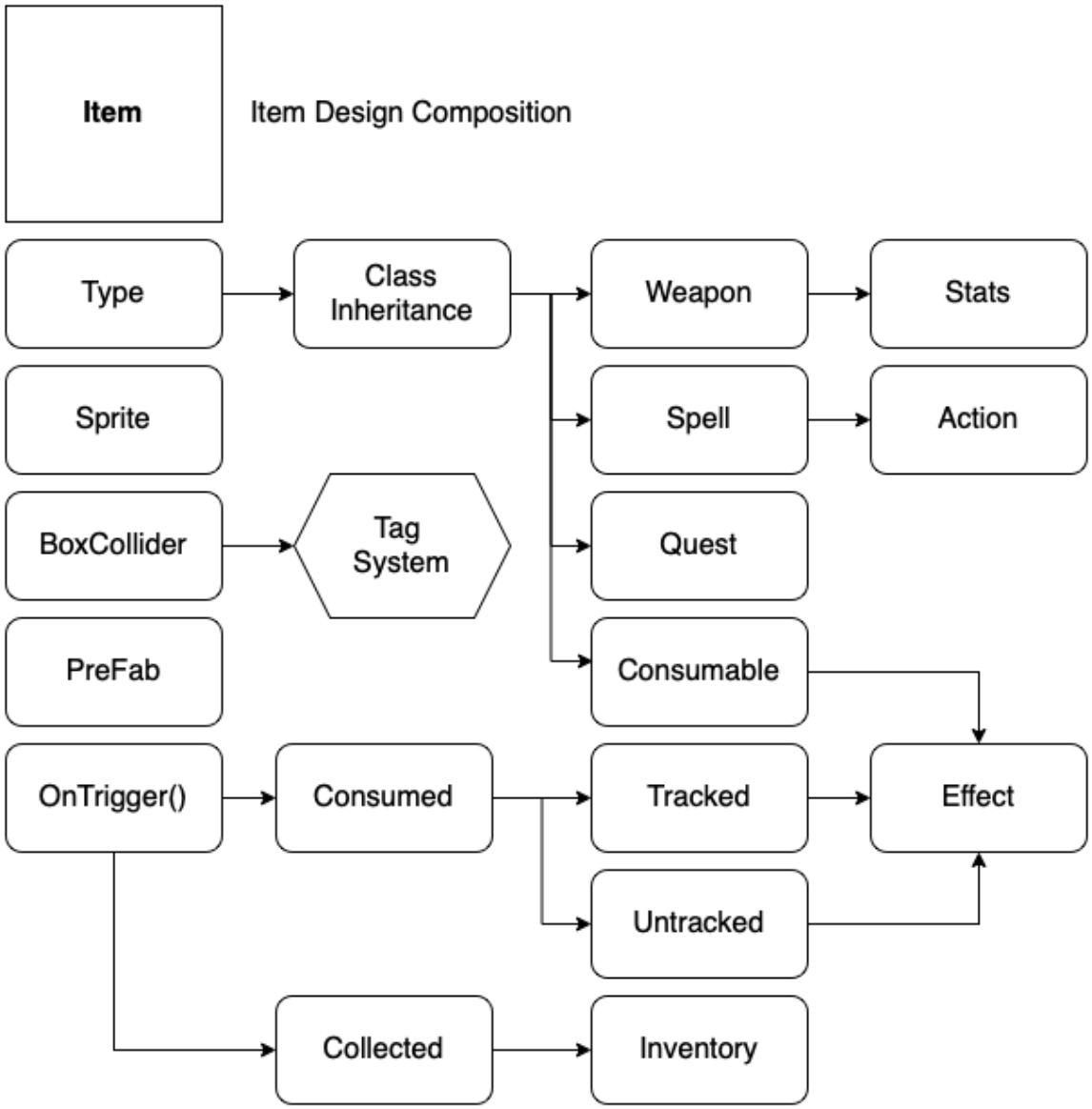
At its most simple, combat occurs when the player uses a weapon and an enemy steps into the collider attached to that weapon (or the reverse). When this happens, an Attack-System is invoked which deals damage to the subject of the attack. The “combat system”, therefore, emerges from the spawning of weapon colliders in response to player control input, and enemy AI input.

Player attacks are initiated by the Player-Action-System. This system consumes user input and maps it to actions. These actions trigger attack animations, and spawn weapon colliders. If an enemy enters into a weapon collider, the Player-Attack-System is triggered. All combat logic is embedded in this system. This system consumes the player’s offensive combat stats, the enemy’s defensive combat stats and HP, and the enemy’s animation state. This system computes the amount of damage to be assigned and then writes the enemy’s new HP back out to the enemy. This system also triggers recoil or death animations upon the enemy. Should an enemy be killed as a result of an attack, this system calls the Leveling-System, Loot-Drop-System, and Story-Progression-System as needed.
Enemy attacks work much same. Enemy weapon colliders are spawned by the Enemy-AI system, which also controls enemy movement and enemy animation state. These colliders trigger the Enemy-Attack-System, which assigns damage and calls other systems as needed.

The Enemy-AI-System is a simple finite-state based AI which toggles an enemy from states such as “patrol”, “attack”, “retreat”, and so on. This AI makes choices based on the enemy’s current HP, whether or not the player is in view, and so on.

Items

Items are represented by in game objects that contain a variance of components - both native to unity, and user created - that define the functionality of the item, and how it interacts within the larger game system and game object.
Separating items from other in game objects and interactive elements is determined by the behaviour of the item within a few key systems; namely the player inventory system, loot system, and combat system.



Items will either directly interact with these systems and exchange relevant data or will be represented within these entities after player interaction.
These three key relationships are further explored and expanded on below.
Items as a concept are defined through the use of class inheritance, component scripts, sprite rendering, and object creation. They are maintained within a database, which is used by other systems to create items as needed.

PreFab System

Items and objects within the game are created during runtime by Unities native PreFab system. (This includes enemies, items, and interactive game world entities)

Items are defined within the PreFab system and that system is called upon by triggers and methods within the game.

When an interaction in the game passes data to the PreFab system it create an instance of the required item entities based on the criteria pre set within the PreFab system.

Master Item Database

The Master Item Database or MID is the main storage of all items created for All Who Wander.

It is pulled on by other systems and databases during game startup so that any items added between versions is updated to the respective areas.

It maintains a list of all items, the other sub databases they belong to (such as the Loot system and its subsequent merchant system) and any other areas that items will need to be updated.

the loot system uses the Master Item Database and the PreFab system to determine what ineligible for loot and what isn't.

Loot Drops

Loot drops, the spawning of items left behind by enemies, is mediated by the Loot-Drop-System.

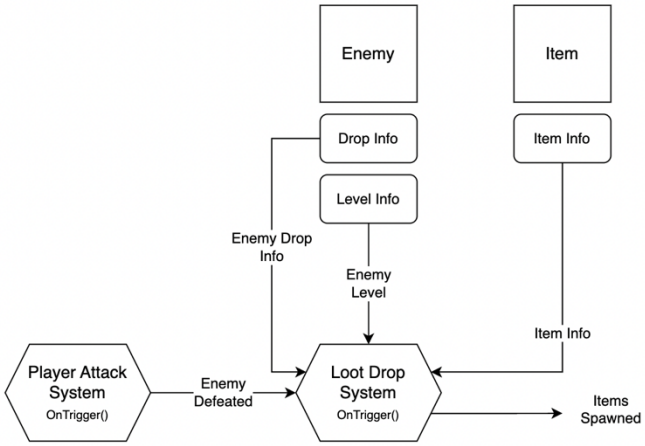


Figure 5 Loot Drop System

This system is called by the Player-Attack-System whenever an enemy's HP is reduced to or below zero during combat. The Loot-Drop-System consumes information about the enemy's preferred drops (different enemies are configured to drop different items) in order to decide which items to drop. The quality or "grade" of the items to drop is determined by the slain enemy's level.

Based off of this information, the items dropped are selected randomly from the possible choices.

The Loot-Drop system then drops a special "bag" item which, when interacted which, opens a UI component which prompts the player to select what, if anything from the drop, they wish to collect.

Leveling

Player leveling is mediated by the Leveling-System. This system is triggered whenever an enemy is slain by the player. This system reads data from the slain enemy and from the player, and it increments a player's level (and base combat stats) with each call.

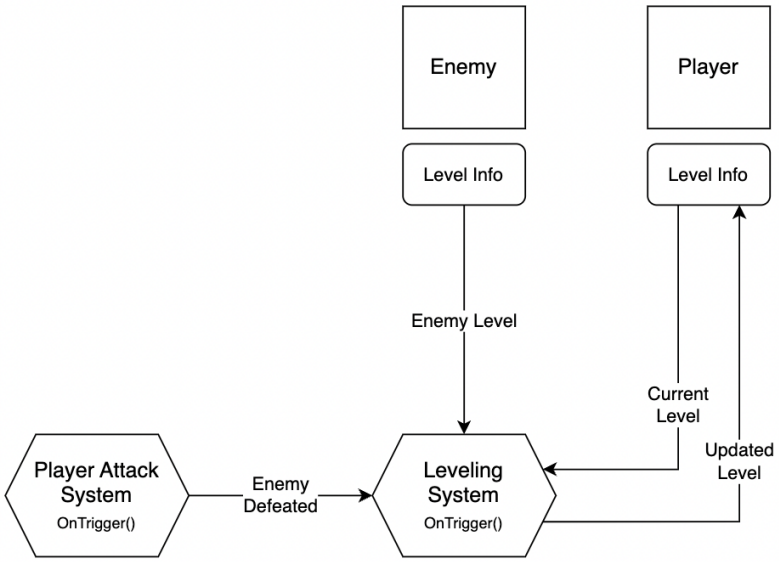


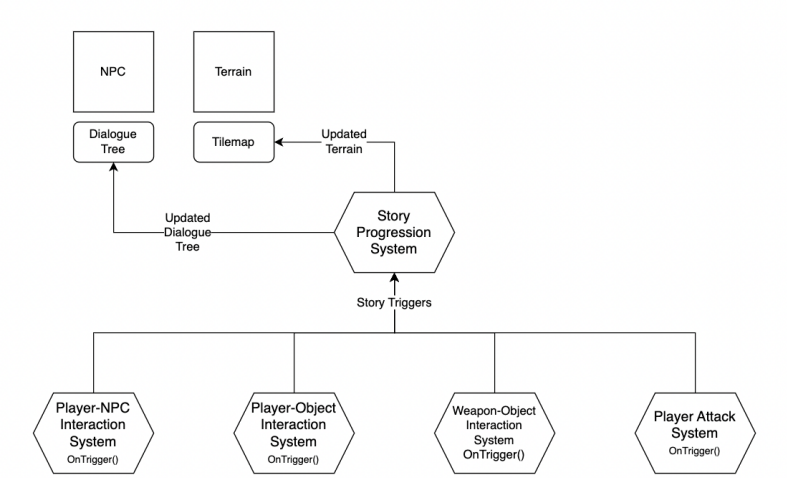
Figure 6 Leveling System

This system is called by the Player-Attack-System whenever an enemy's HP is reduced to or below zero during combat. This system consumes information about an enemy's own level to compute a desired amount of EXP to award to the player. Each time the player's accumulated EXP exceeds a threshold value, the player's level is increased. With each increase in level, the player's base combat stats are increased.
The threshold for an increase in level itself increases with each additional level. As well, enemies with higher levels yield greater amount to a player's EXP when defeated. Needless to say, enemies with higher levels have higher combat stats, and are therefore more difficult to defeat.

Spawning Enemies and Treasure

Story

Story progression is mediated by the Story-Progression-System, which itself is called in response to interactions or combat. This system consumes data from a variety of sources, though this data is always in the form of a uniform "story trigger." This system can have a variety of effects.



By and large, the story-progression system is used to modify the dialogue of key characters. It can also be used to modify the landscape in response to events.

The various interaction systems, as well as the Player-Attack-System, will contain logic to detect any story triggers embedded on the objects they act upon. These story triggers and any other salient information about the interactions (such as which object was interacted with, which NPC said what, and so on) are passed from the originating system to the Story-Progression-System.

Using this information, the Story-Progression-System what, if anything, to modify about the game world in response to the event.

The exact format for story triggers has not yet been decided. They may be simple integer flags, enums, strings, or objects in their own right.

User Interface and Rendering

This diagram represents the interactions that occur between aspects of the UI system in the game. Entities are representations of game concepts and the components attached to them provide data about the entity. Functionality is facilitated by the systems that gather information from the components.

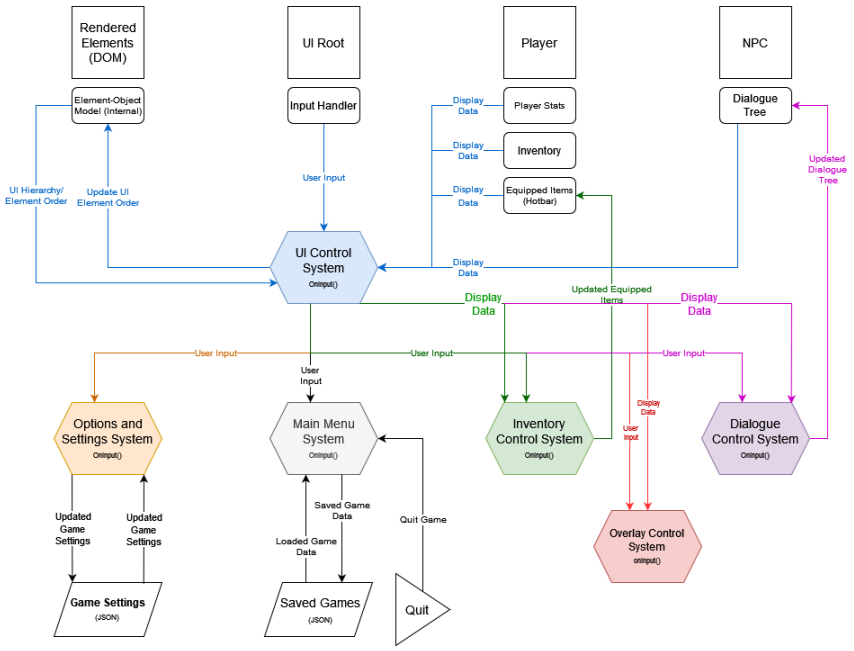


Figure 2 User Interface Systems

Designing the Gameworld

Gameworld Design Overview

The tile-based world of All Who Wander features several regions which are interconnected by traversable paths. Players can explore “off-path” areas through swimming and climbing. In addition, the world includes region-specific dungeons and other in-world objects such as chests, or shrines which endow the player with rewards or story progression.

Tile Maps

Unity’s tile map system will be utilized to sketch and design the many regions within the vast world of All Who Wander. A tile map is made up of a grid overlay and several other cooperating parts. With the use of tiles, brush tools, and defined rules for tile behavior, the entire system allows for easy painting of levels. It comes with support for moving tiles, randomized tile placements, and more.

Tile maps in Unity present themselves as objects which can be added to the hierarchy. There is only one grid in the Scene, but there can be multiple tile maps thanks to the one-to-many relationship that the level editor uses. The tile maps rely on the Scene's layout, which is created by the grid.

Layering and Collision

Due to the tile map system's versatility, several aspects of the game world will be arranged onto different layers. For impassable tiles such as rocks or trees, the Tile map Collider2D component comes in handy. For Tiles on a Tile map component on the same GameObject, Tile map Collider2D generates Collider shapes. The Tile map Collider 2D modifies the Collider forms during *LateUpdate* when Tiles from the Tile map component are added or removed.

Low-Level Implementation

Entities

Everything which exists in the game world is an Entity. These entities are all represented by a sprite, and have a collider attached to them. Entities themselves are entirely generic. What defines an entity — its behaviour, the data attached to it, how its rendered, and so on — are the components attached to it.

In All Who Wander, there exist eight types of Entity: the Player, Enemies, NPCs, Objects, Items, Terrain, the User Interface, and the Story.

Many of these entities share characteristics — all are renderable, and all participate in physics by way of collisions. The player and enemies both have combat-stat components used in combat, while the enemy and NPC entities both have AI which move them around the world.

Player

The player entity is the game object which represents the player within the game. This entity is controlled by the player through input from the keyboard or a game controller. This entity engages in combat, moves around the world at the behest of the user, and interacts with most other entities in the game.

This entity contains the following components:

Animator	Input Handler	Box Collider	Sprite Renderer	Combat Stats
Animation State	Wallet	Inventory	Equipped Items	Position

The entity interacts with the following systems:

Player-Movement-System	Player-Action-System	Player-Object-Interaction-System

Player-NPC-Interaction-System	Player-NPC-Interaction-System	Item-Pickup-System
Story-Progression-System	UI-Control-System	Inventory-Control-System
Spawning-System	Leveling-System	

Enemy

Enemies are self-controlling entities strewn throughout the gameworld. Enemies engage in combat through the combat system. When enemies are defeated, they trigger the Loot-Drop, Leveling, and Story-Progression systems. Enemies move themselves around the gameworld and execute attacks at the behest of the Enemy-AI system.

This entity contains the following components:

Animator	Sprite Renderer	Box Collider	Combat Stats	Drop-Info
Animation State	AI-State	Attack-Collider		

The entity interacts with the following systems:

Enemy-AI-System	Enemy-Attack-System	Leveling-System
Loot-Drop-System	Spawning-System	Story-Progression-System
Player-Attack-System		

NPC

Non-Player Characters (NPCs) are self-controlling entities strewn throughout the gameworld.

NPCs do not engage in combat. Often, NPCs do participate in the dialogue subsystem of the user interface. When interacted with, an NPC may engage in dialogue, give items to the player, or trigger story events. NPCs move around the gameworld at the behest of the NPC-AI system.

This entity contains the following components:

Animator	Box Collider	Sprite Renderer	AI-State	Items-Given
Animation State	Story Triggers	Dialogue Tree		

This entity interacts with the following systems:

NPC-AI-System Story-Progression-System UI-Dialogue-Subsystem Player-NPC-Interaction-System

Object

Objects are non-living, interactable entities strewn throughout the gameworld. These objects do not move around or act on their own accord, but they can be interacted with (unlike static scenery baked into the tile maps).

Like NPCs, Objects can trigger dialogue (like signposts), give items to the player, or trigger story events. Some objects may only be interacted with by striking them with specific weapons.

This entity contains the following components:

Box Collider Sprite Renderer Dialogue Tree Items-Given Story Triggers Animation State

This entity interacts with the following systems:

Player-Object-Interaction-System Weapon-Object-Interaction-System UI-Dialogue-Subsystem Story-Progression-System

Items

Items are non-living entities strewn throughout the game-world. There are many different types of items — weapons and spells are items, healing or fortifying potions are items, and some items may have entirely unique effects.

Items are spawned from defeated enemies or else collected from the landscape. By walking over an item, the player collects it. This destroys the item, but a figment of it is stored as pure data within the player's inventory. The player may use that item by equipping via the hotkey registration system.

This entity contains the following components:

Box Collider	Sprite Renderer	Item Method Components	Sprite Shaders	Item Data Components

This entity interacts with the following systems:

Item-Pickup-System	Loot-Drop-System	Player-Object-Interaction-System
Player-NPC-Interaction-System	Weapon-Object-Interaction-System	Player-Action-System

Terrain

Terrain are tile maps. Assembled using Unity's tile editor, tile maps are drawn sprite by sprite layered on top of each other. Some terrains are passable, while other terrains cannot be moved through by the player. Some terrains have colliders on them used for purposes other than impassability, such as obscuring the line of sight for the enemy AI.

This entity contains the following components:

Tile Collider	Tile Renderer

This entity interacts with the following systems:

Enemy-AI-System	Story-Progression-System	Player-Movement-System
NPC-AI-System		

Interface

The user interface is represented by the UI Root Entity. Interactions with the current UI are monitored with one component. These aspects are then used by the UI Control System.

- Input intended for the UI is contained within its own Input Handler component, and is passed to the UI Control System.
- The UI Control System takes in the user inputs and along with other data from relevant components to determine the user selected action to perform.

Every user interface within Unity is contained within the canvas. This canvas is a child of the UI-Root object, but should be regarded as the same entity as the UI Root. Interactions between the canvas and the rest of the game are monitored by two processes and facilitated by a single system.

- Element-Object Model, this component contains all of the individual user interfaces in a hierarchy and has event listeners for each (where the event occurring is sent to the UI Control System for processing)
- The UI Control System updates the element visibility based on event listeners obtained from the element object model by updating the order in which UI elements appear in the element object model

Story

The story is an invisible entity and child to the Scene (the root object in the Unity object model).

The story entity maintains a list of boolean flags indicating whether or not certain events have occurred. These events may be interactions with key NPCs, the defeat of certain enemies, and so on. Typically, one event cannot be triggered as being true until the event preceding it has been triggered.

These booleans are used by NPCs and Objects to determine what dialogue is appropriate for a given situation.

This entity contains the following components:

Story State

This entity interacts with the following systems:

Story-Progression-System UI-Dialogue-Subsystem

Systems

Systems are scripts attached to game objects which contain the functionality that make the game run. Each system is responsible for one and only one task (though, this task may be arbitrarily complex). Systems can be executed once per frame, executing functionality regularly. They may also be executed in response to events such as interactions between objects. Some systems may call other systems in response to specific circumstances which arise in their logic.

Systems contain only functionality, and store no internal state of their own. However they are called upon, systems consume entities and components which are attached to them. The data

contained in these components is read and used within the system's logic. The system may write that data back to original source once modified, or it may write out to entirely different targets.

At this stage in development, it would be folly to try to count the number of systems which exist in the game's architecture. As it stands, *somewhere* around a dozen such systems should exist.

Player Movement System

The Player-Movement-System is responsible for translating user input into player movement and animation of the player entity. This system is triggered whenever input is detected. For clarity, let it be noted that the built-in Input-Handler components attached to the player will automatically fire callbacks attached to this system.

This system consumes user input, the player's current position, the player's current animation state, and terrain data. Based on this data, this system decides where to move the player.

To prevent the player from moving through impassable tiles, this system forms a ray casting operation prior to moving the player. In doing this, this system checks to see if there is an impassable tile collider at the location the player is trying to move to. Independent ray casts are made for movement in the *x* and *y* directions.

Assuming motion is valid, this system updates the player's position and animation state.

This system contains the following methods:

OnMove()	This method is called whenever arrow keys or the movement joystick is used. This method performs the logic outlined above.

Player Action System

The Player-Action-System is responsible for translating user input into player actions, such as using a weapon or interacting with another entity. This system is triggered whenever input is detected. For clarity, let it be noted that the built-in Input-Handler components attached to the player will automatically fire callbacks attached to this system.

This system consumes user input, the player's animation state, and the current mapping of actions to keys. If the action mapped to an input is a weapon, this system will spawn an invisible instance of that weapon whose collider can be interacted with, and also trigger an attack animation. If the action mapped to an input is an item, the item's effect-callback will be called.

If the input is the generic interaction key, then the player's generic interaction collider will be enabled briefly.

This system also receives feedback from the player's animation controller component. When attack animations end, a method on this system will automatically disable any colliders attached to the player, and also reset any animation state members to null/zero/false.

This system contains the following methods:

On<SomeKey>()	These methods, as there are many, are automatically called by the Input Handler component on the player. Once called, these methods execute whatever action is mapped to that key. Often, these methods spawn colliders which themselves trigger interactions between entities.
OnAttackEnd()	This method fires whenever an attack animation (on the player) finishes. This method resets any animation toggles to null/zero/false, and disables any colliders attached to the player.

Player Attack System

The Player-Attack-System is responsible for engaging in combat with enemies. This system is triggered whenever an enemy collides with a weapon's collider.

This system consumes the player's combat stats, the enemy's combat stats, and the weapon's combat stats. Based on this numerical data, the amount of damage dealt to the enemy is computed (the specifics of this algorithm has not yet been decided upon). The enemy's HP is reduced by this amount.

Should an enemy's HP be reduced to or below zero, the enemy is slain. This system will then forward the player and enemy entities to the Loot-Drop and Leveling Systems. This system will also check for any story triggers attached to the enemy, and call the Story-Progression-System as needed.

This system contains the following methods:

OnTrigger()	This method is called when an enemy enters a weapon's collider. It performs the above logic.

Enemy AI System

The Enemy AI System is responsible for controlling an enemy's movement, triggering its approach and retreat from the player, and triggering attack animations and colliders.
This system consumes the player's position, the enemy's current AI and animation states, and terrain data in order to make simple decisions about controlling the enemy. This system is essentially a very simple finite-state-machine based AI which is called once per frame in order to update and control an enemy.

The exact nature of this FMS has not yet been decided. What is known is that the AI will perform a line-of-sight check (using ray casting and terrain colliders) to determine if the player is within view of the enemy. If so, the enemy may engage in combat. Otherwise, the enemy will likely remain idle or else idly patrol. Additionally, this AI will invoke as yet undecided upon path finding routines to determine how to move to the player if needed.

This system will directly modify an enemy's position and animation state in accordance with whatever decisions it makes. Additionally, this system will also directly enable an enemy's attack collider as needed. Upon completion of an enemy's attack animation, a method on this system will be called to disable the enemy's attack collider.

This system contains the following methods:

FixedUpdate()	This method is called once per frame to execute the AI logic, outlined above.
OnAttackEnd()	This method fires whenever an attack animation (on the enemy) finishes. This method resets any animation toggles to null/zero/false, and disables the enemy's attack collider.

Enemy Attack System

This system works identically to the Player-Attack-System, except it makes a call to the user interface in order to signal the end of the game should the player's HP be reduced to or below zero.

NPC AI System

This system is responsible for controlling an NPC's motion. This system is called every frame.

Since NPCs do not engage in combat, this system will be limited to controlling simple back and forth, aimless motion. This system will consume an NPC's animation and AI states. This system will then choose a random location to move to. Each frame, the NPC's position will be updated until it reaches its destination, and the process is repeated. Of course, this system will also handle an NPC's animation state ("idle" or "walking") as needed.

This system contains the following methods:

FixedUpdate()	This method is called once per frame to execute the AI logic, outlined above.

Player-Object Interaction System

This system is responsible for executing player-object interaction logic. This system will be called whenever a player's interaction collider is spawned in proximity to an interactable object.

This system consumes an object's story triggers, dialogue tree, and interaction-effect callbacks (stored as components on the object). Any story triggers present on the object will be passed to a call to the Story-Progression-System. Any dialogue will be passed to the UI-Dialogue-System.

Any other custom interaction effects will be called, potentially triggered other systems.

This system contains the following methods:

OnTrigger()	This method is called whenever a player's interaction collider is spawned in proximity to an interactable object. It executes the logic outlined above.

Player-NPC Interaction System

This system is responsible to executing Player-NPC interaction logic. It operates identically to the Player-Object interaction system, outlined above.

Weapon-Object Interaction System

This system is responsible to executing Weapon-Object interaction logic. It operates identically to the Player-Object interaction system outlined above, except that it is triggered when a specific weapon is used in proximity to an Object.

Spawning System

The spawning system dictates where/when/how enemies are spawned within the world. Through tracing the main camera vector position within the gameObject and the calculated use of enemy spawner objects the world of All Who Wander will be populated with a variance of challenging enemies.

Spawners are objects within the gameObject with no rendering or collision box that determine when and where enemies of a specific type are instantiated within the games scene.

Spawners begin the creation of enemies when the main camera object is moved within a specified distance of their static position within the games (X, Y) coordinate plane. Additionally, to save memory enemies that have moved out of a specified distance of the main camera destroy themselves, without the call to and on death functionality.

Loot Drop System

The Loot Drop System is a subset of the Item system that is in turn modified and manipulated by the Combat and Level Systems.

Loot helps determine player power as well as player reward and thus is a delicate system where enough needs to be given to create a feeling of reward without giving so much that the player feels overpowered.

Loot is handled by a RNG that is passed either player level (with regards to chests) or enemy level (with regards to loot drops). These levels modify the mathematics handled by and RNG system that determines what items are rewarded to the player upon interaction with a chest object or the defeating of an enemy. This RNG range determines what items are dropped from the Loot Item Database which is a subset of the Master Item Database maintained by the item system. Depending on the difficulty of the enemy or dungeon this may be called on multiple times to reward multiple items for loot.

High difficulty objects (either Boss enemies, or chests at the end of dungeon) will have their levels skewed to offer higher rewards.

Leveling System

This system is responsible for scaling the players combat experience, as well as determining where the player is within the Story Progression System.

Through stat increases (or an overall increase to player power in combat) , Dungeon/Area progression, as well as loot availability the Levelling system directly influences a players survivability, challenge level, and overall effectiveness within the game.

Some items will be unusable until the player reaches a specific level, and those items may be required for continuation along the Story Progression System.

Story Progression System

This system is responsible for modifying the game-world in response to key story events. It may update an NPC's dialogue tree, modify the landscape in some way, or have other effects. This system is called by other systems, usually in response to an interaction between the player and some object or NPC.

This system consumes story triggers from any number of different entities. Since this system can be triggered by Player-Object interactions, Player-NPC interactions, and more, it has multiple methods attached designed to handle the specifics of each type of trigger. When triggered by a Player-Object interaction, for example, the object in question will be supplied to this system for further processing.

The exact nature of this system has yet to be decided. Whether story triggers are callbacks with specific effects baked in, or if this system must infer the effects of a trigger based on other data, remains to be seen.

This system contains the following methods:

OnPlayerObjectInteraction()	This method is called in response to Player-Object interaction when that object contains a story trigger.
OnPlayerNPCInteraction()	This method is called in response to Player-NPC interaction when that NPC contains a story trigger.

OnWeaponObjectInteraction()	This method is called in response to Weapon-Object interaction when that object contains a story trigger.
OnEnemySlain()	This method is called by the Player-Attack-System when an enemy is slain, and that enemy contains a story trigger.

UI Control System

Based on display data, event listeners, and the UI element order, all obtained from relevant components, this system delegates data to be handled by the correct system.

- If a display value is to be updated, the respective system will handle this update, the UI Control System will determine precisely which system should perform the action and what data that system will require to do so
- If a new UI is to be displayed, then the UI Control System will handle this by updating the element-object models ordering to display the correct UI

Subservient to the UI Control System are a handful of subsystems. They are:

- **Options and Settings System:** Facilitates changes made to the game settings through the settings user interface. The game settings are stored in a JSON file, initially being populated by default values. The UI Control System sends the user input to this system to be used for displaying and updating the game settings.
    - When the settings screen is displayed then the options and settings are displayed as specified in the game settings file
    - If a game option/setting is changed, then the change is written to the file, once the user submits the changes their input will be noticed, and the game options/settings will be displayed with the new updated values
- **Main Menu System:** Facilitates actions contained within the start game screen, like saving a game, loading a game, settings, quitting. Receives user input from the UI Control System and performs the required action.
    - If the user selects quit the game immediately stops running
    - Saving a game will save a snapshot of the entire game at that instance to a JSON file
    - Loading a game will pull a snapshot from the saved games file and resume the game from the point in which the saved game specifies
    - Opening the settings menu while in the start menu will shift the system being used to the options and settings system
- **Inventory Control System:** Responsible for displaying the player character inventory. Receives display data and user input from the UI control system. As the hot bar is a subset of the

inventory and is graphical in nature it is updated by this system.

- **Dialogue Control System:** This system is responsible for the display of any dialogue within the game. It receives display data and user input and in turn displays a text prompt displaying whichever text was passed to it. It updates the dialogue tree component contained within the NPC entity once it has displayed to the user.

- **Overlay Control System:** Displays important data to the user while the game is actively running. Takes in display data, and user input from the UI Control System and makes the appropriate changes if necessary. The hotbar is also displayed within the game overlay so the user can see their currently equipped weapons.

Components

Tag

The tag variable is a native Unity variable that is called on by various built in methods from unity as well as from user defined methods. Tags allow components to check for Boolean like states to determine if methods are appropriate to execute.
An example of this is if the player attacks it deals damage only to objects with the Enemy tag rather than all objects.

Input Handler

This built-in component contains protocols for registering input from the user and mapping that input to callbacks. Specifically, when this components a “move” event (for example), it will automatically search for an “OnMove” method attached to any other component on the same entity as the one owning the input handler.

Unity’s input handling system is very robust. A single Input Handler is sufficient to map input from keyboards and game controllers simultaneously, whilst also being highly configurable from inside the Unity editor.

Animator

This built-in component automatically controls sprite animations. This component must be configured for each type of entity from within the Unity editor. Sequences of sprites or “animations” are loaded in and tagged individually. Each animation can be assigned to listen for changes to some member on some component within the entity — its “animation state.” When this member is assigned the appropriate value, the animation triggered.

Individual animations can be assigned callbacks which are executed when the animation ends. These callbacks are often used to reset animation state to null/zero/false values, cleaning up the animation-state component.

Position

Position is handled with Unities Vector class, which stores and handles an objects dynamic location within a plane of (x,y) coordinates

Collider

This built-in component is used by Unity's physics system to detect collisions. It comes in numerous shapes, including custom shapes. Colliders can be set to "trigger" mode. When a trigger collider is collided with, physics are ignored and instead a callback registered to collider is executed. This functionality forms the basis of the various interaction systems.

Interaction Collider

The interaction collider is a secondary collider attached to the player, enabled briefly whenever the "interact" key is pressed. This is a trigger collider which calls interaction systems.

Weapon Collider

The weapon collider is the primary collider attached to weapons. It deserves special mentioning because its usage varies from other entities in the game. Items in a player's "equipped items" components are "virtual" in that they don't actually exist as part of the scene. When the player uses a weapon, an actual weapon object is spawned and placed in the scene for a brief moment. The weapon is destroyed again when the attack animation is completed.

This weapon's collider is set as a trigger collider during this brief period. If an enemy collides with it, the Player-Attack-System is triggered.

Attack Collider

The attack collider is a secondary collider attached to enemies, enabled briefly when the enemy AI triggers an attack. This is a trigger collider which calls the Enemy-Attack-System when the player collides with it.

Player Stats

The player stats component contains player character information. Health points, Mana, Stamina, and Experience Points are stored here. This data is sent to the UI Control System to be used for display purposes.

Inventory

The player entity's inventory is what contains the items that the player character possesses. This data is sent to the UI Control system for display purposes.

Equipped Items (Hotbar)

Contains the currently equipped items for a player character. The hotbar items are a subset of the entire inventory of items. The items that are currently contained within this component are sent to the UI Control System which can then use the data and further delegate to other systems.

Wallet

This custom component stores how much money (and potentially other numeric collectables) held by the player. This is used by the Merchant System.

Combat Stats

This custom component is held by the player and by enemy entities. It stores a variety of numerical data used in combat, such as the entity's current HP and maximum HP, experience level, base attack and defence stats, movement speed, and so on.

Item Data

This component contains boilerplate/administrative data about items including their type, rarity, value in currency, and so on.

Item Callbacks

Since items can have a great deal of different effects — too many to hard code into a singular entity — an item's effects are custom written into closures attached as components to individual items. These closures are called when the item is used by the player.

Enemy Info

This component contains boilerplate/administrative data about an enemy including its type.

Drop Stats

This component contains data about what type of items an enemy is likely to drop when defeated, including the relatively likelihood than any one item is dropped.

Interaction Callbacks

Since objects and NPCs can have unique effects when interacted with by the player. While most interactions are cookie-cutter and can be handled by systems in question, some custom interaction events may be required. These components take the place of closures, and are called if present on an object or NPC during an interaction. They may have any number of effects.

Story State

This component stores the current state of the story. Its exact nature has yet to be determined, but this component will likely store a sequence of boolean values each corresponding to an event in the story. When interacting with an NPC, the dialogue system may reference the story's state in order to determine which unit of dialogue to display.

Story Triggers

Story triggers are components stored on objects and NPCs, used to signal a story progression event in response to interaction with the object or NPC.

Dialogue Tree

In terms of UI, the information required from an NPC is its dialogue. The dialogue tree contains the dialogue for all NPCs and an order in which these dialogues should follow. This data is then sent to the UI Control System. It is then delegated further to the Dialogue Control System. The dialogue control system also updates the dialogue tree with the current trajectory of dialogue.

Region Stats

UI Input Handler

Input data is collected from the input handler and is sent to the UI Control System that uses it. The user input and information obtained from the element-object model component allow the UI Control System to perform a specified action. For a button, toggle, slider, the position of the mouse click relative to the screen will be sent to the UI Control System.

Element Object Model

This component contains the 'templates' for all the UI elements present in the game. Internally Unity keeps a hierarchy of these elements. UI Elements may have or be buttons, toggles and other interactable

components, the element object model will listen for interaction events that occur and relay these to the UI Control System.

The order in which the UI elements are displayed, or if they are visible at all, is controlled by the order in which they appear in the Element Object Model. This order is changed by the UI control system determining which UI elements are viewable.