KAM

Project Information

Project Type: Graduation Project
Engine: Unity 6
Genre: Turn-Based Roguelite RPG
Perspective: Isometric
Theme: Turkish Mythology
Role: Game Designer / Systems Designer / Gameplay Programmer
Core Tools: Unity, C#, ScriptableObjects, UGUI, TextMeshPro, New Input System, Cinemachine, NavMesh

---

My Role

For this project, I worked across both design and implementation. I was responsible for designing and developing the core gameplay systems that support the main loop of the game.

My work included:

- Isometric camera and player movement
- Interaction system for chests, portals, bonfires, pickups, and skill tree objects
- Inventory, quick slots, item pickups, and collectible logic
- Turn-based combat system
- Action Point system
- Timed-hit / QTE combat mechanic
- Enemy roaming, detection, and combat behavior
- Enemy skill system
- Skill tree and permanent progression
- Oba hub and Spirit Realm run loop
- Bonfire save, death, and respawn flow
- Persistent world state for chests, bosses, pickups, and shortcuts
- Combat UI, HUD, tooltips, damage popups, and combat log
- Audio management for scene music, combat music, UI sounds, and result feedback
- Scene transition fade system

Design Goal

The main design goal was to create a small but expandable RPG framework that could support a roguelite structure while still feeling readable and manageable as a graduation project.

Instead of building isolated mechanics, I focused on connecting systems into a complete gameplay loop. The player needed to explore, encounter enemies, make tactical combat decisions, collect rewards, save progress, die, respawn, and return stronger through permanent upgrades.

From a technical perspective, the goal was to avoid hardcoded one-off features. I wanted the project to be easy to expand with new items, enemies, skills, interactable objects, and combat actions without rewriting the core systems.

Core Gameplay Loop

The game is structured around the relationship between the Oba hub and the Spirit Realm.

The Oba works as the player's persistent safe area. From there, the player can enter the Spirit Realm, fight enemies, collect temporary and permanent resources, and return through progression or death.

The core loop is:

Oba Hub → Enter Spirit Realm → Explore → Fight → Collect Loot → Save at Bonfire → Unlock Skills → Die or Return → Start Stronger

This loop helped define the structure of the game's systems. Combat, inventory, saving, progression, and scene transitions were all designed to support this flow.

System Architecture

I separated the project into multiple gameplay layers:

Game Flow: GameManager, scene transitions, hub/run state
Combat: TurnManager, CombatUnit, combat actions, AP, status effects
Progression: PlayerProgression, SkillEffectApplier, skill tree data
Inventory: InventoryManager, item data, quick slots, pickups, spirit fragments
Interaction: InteractionDetector, interactable objects, prompt UI
Save System: SaveManager, bonfire data, persistent world state
UI: Combat HUD, inventory UI, skill tree UI, death UI, pause/main menu
Audio: AudioManager, combat music director, UI SFX, combat result SFX
Enemy Logic: Roaming, detection, state machine, enemy combat skills

This separation made it easier to test individual features and prevented systems such as UI, combat, inventory, and saving from directly owning each other's logic.

Isometric Camera and Player Movement

I implemented an isometric camera system with smooth follow behavior. The camera follows a target transform using an offset and smoothing, keeping the view stable and readable during exploration.

Player movement was built separately from camera behavior. The player handles movement and rotation, while the camera only follows the target. This separation allowed combat state changes, exploration movement, and visual presentation to remain independent.

The character animation system was also separated from the movement logic. Movement scripts produce speed and direction values, while the Animator reacts through parameters such as `IsMoving`, `Speed`, `InCombat`, `Attack`, `Hit`, and `Death`. This allowed the animation clips and character model to change without rewriting gameplay code.

Interaction System

I built an interaction system that allows the player to interact with world objects through a shared input flow. Objects such as chests, bonfires, portals, pickups, lore objects, and the skill tree altar all use the same general interaction structure.

The player carries an interaction detector. When the player approaches an interactable object, the UI prompt appears and updates based on the target object. Pressing the interaction key triggers the object's own behavior.

This approach made the system scalable. Instead of writing separate input logic for every object type, each interactable object defines what happens when it is used.

Inventory, Quick Slots, and Collectibles

The inventory system handles item collection, stacking, quick slot assignment, item usage, and spirit fragment tracking.

Item information is stored as data, while the player's current inventory exists at runtime. This means item icons, descriptions, effects, and pickup behavior can be adjusted without changing the core inventory code.

Quick slots work as shortcuts to inventory items. Using an item from a quick slot still consumes it from the inventory, keeping one source of truth for item ownership.

I also integrated plant consumables, chest rewards, pickups, and spirit fragments into the inventory structure. This allowed exploration rewards and combat resources to connect naturally with the UI and progression systems.

Turn-Based Combat System

The combat system is built around a shared `CombatUnit` model for both the player and enemies. This shared structure handles health, damage, healing, status effects, death checks, and combat feedback.

Combat flow is managed by a central turn manager. When combat starts, the system gathers active units, calculates turn order, starts turns, listens for actions, and determines victory or defeat.

By keeping turn flow centralized, combat result logic only fires once. This prevents duplicate UI updates, repeated audio triggers, and inconsistent win/lose behavior.

The combat system was designed to support multiple layers of feedback, including health bars, turn order portraits, damage popups, enemy intents, combat log messages, and audio cues.

Action Point System

To make combat decisions more tactical, I implemented an Action Point system. Player actions such as basic attacks and skills use AP, while unavailable actions are disabled in the UI when the player does not have enough AP.

A key design decision was that failed actions should not spend AP or end the turn. The system checks whether an action can be executed before consuming resources.

Defend works as a strategic action that allows the player to preserve or bank AP, creating opportunities to use stronger skills on a later turn. This gives combat a stronger sense of planning instead of making every turn feel identical.

Timed-Hit / QTE Combat Mechanic

I added a timed-hit mechanic to make combat more active. During certain attacks, the player is asked to press the correct input within a timing window. The result can be a miss, success, or perfect input, which then affects the damage calculation.

The timed-hit system is intentionally separated from the combat action itself. It does not directly own the damage logic; instead, it returns a result that the combat system can use. This keeps input timing, UI display, and damage calculation modular.

This system also connects with skill tree upgrades, allowing progression bonuses to affect timed-hit results through a central skill effect layer.

Enemy Roaming, Detection, and Combat Behavior

Enemies use a state machine structure to separate exploration behavior from combat behavior. In exploration, enemies can roam, patrol, detect the player, become alert, and then transition into combat.

Detection uses settings such as vision distance, detection angle, and combat engage range. This gives enemies a more readable behavior pattern while keeping combat initiation under control.

I also implemented a first-pass enemy skill system. Instead of only using basic attacks, enemies can choose special actions based on their profile and combat situation. These actions can include debuffs, poison, healing, special attacks, or other behavior types.

This made enemies more distinct and helped the combat system support more varied encounters.

Skill Tree and Progression

The skill tree allows the player to spend spirit fragments on permanent upgrades. These upgrades can affect combat, exploration, timed-hit behavior, defensive bonuses, and future skill unlocks.

A major technical decision was to avoid hardcoding skill effects directly into every combat script. Instead, I used a central effect query system. Combat and exploration systems can ask whether a specific skill bonus is active, without needing to know the full structure of the skill tree.

This made the progression system easier to expand and reduced the risk of scattering upgrade logic across unrelated scripts.

Oba Hub and Spirit Realm Run Structure

The project uses a hub-and-run structure. The Oba acts as the persistent home area, while the Spirit Realm acts as the main danger zone for exploration and combat.

The GameManager tracks whether the player is in the hub or in a run. This allows portals, respawn logic, scene loading, and run results to behave differently depending on the current state.

This structure helped support the roguelite identity of the project. The player repeatedly enters a dangerous space, collects resources, risks death, and returns to a permanent hub where long-term progression happens.

Bonfire Save, Death, and Respawn

I implemented a bonfire save system inspired by Souls-like checkpoint logic. When the player rests at a bonfire, the game records the bonfire ID, scene name, spawn position, player progression, inventory data, and persistent scene state.

When the player dies, the game respawns them at the last saved bonfire and applies a spirit fragment penalty. The player’s state is restored, normal enemies and non-persistent loot can reset, and persistent world objects retain their saved state.

The bonfire is not just a UI object. It is an interactable save point connected to the SaveManager and the respawn flow.

Persistent World State

To make the world remember important changes, I implemented persistent state saving for objects such as opened chests, collected pickups, defeated bosses, and opened shortcuts.

These objects are saved using unique IDs rather than scene position. This means that when the scene reloads, the game can correctly restore whether a chest has already been opened or whether a shortcut should remain unlocked.

This was important for supporting a roguelite structure where some elements reset after death or rest, while others remain permanent.

Combat UI and Player Feedback

I worked on the combat UI and HUD systems to make turn-based information readable. The UI includes action buttons, skill submenu, AP display, health panels, turn order, damage popups, status icons, enemy intents, and a combat log.

The UI does not own the gameplay logic. Instead, it listens to combat, inventory, AP, and progression events. When data changes, the UI updates visually.

This event-driven approach made the UI more flexible and prevented combat logic from becoming dependent on specific UI objects.

Audio and Scene Transitions

I implemented a central audio structure for menu music, Oba music, Spirit Realm music, temporary combat music, UI sounds, and combat result sound effects.

When combat starts, the audio system switches to combat music. When combat ends in victory, the victory sound plays and the scene music returns. When combat ends in defeat, the lose sound plays and the death/respawn flow takes over.

I also added a scene transition fade system. Instead of instantly loading scenes, the game fades out, loads the target scene, waits for initialization, and fades back in. This is used for menu transitions, portal travel, bonfire respawn, and death recovery.

The fade system also helps hide awkward teleport or spawn frames during scene reloads.

Technical Decisions

Several architectural decisions shaped the project:

ScriptableObject-based data
Items, skills, combat actions, loot tables, and configuration values are handled as data where possible. This makes balancing and iteration easier inside the Unity Inspector.

Event-driven UI
UI systems listen to gameplay events instead of controlling gameplay directly. This keeps visual feedback separate from core logic.

Manager classes for global systems
Game flow, saving, audio, and turn management are handled through central managers so that scene changes and global state transitions remain controlled.

State machines for enemy behavior
Enemy roaming, alert, and combat behavior are separated into clear states, making AI behavior easier to debug and expand.

ID-based save data
Persistent world objects are saved by unique IDs, allowing the game to restore object states correctly after scene reloads.

Modular combat structure
Combat units, combat actions, AP, timed hits, status effects, and skill progression were designed as connected but separate systems.