Computer Games Laboratory (TUM)

WS 2025/26

# A Game Played in Reverse (GPR)

Project Notebook — Milestones 1–2

Team *Last Minute*

Ayoub Mabkhout, Samuel Saruba


**Contacts:** ayoub.mabkhout@mytum.de, samuel.saruba@mytum.de

## Contents

# I Milestone 1: Formal Game Proposal

## 1 Game Description

### 1.1 Hook & Core Concept

The idea for our game originated from years of playing round-based RPGs and watching damage numbers spiral out of control. Stats stack, combos chain, status effects multiply—suddenly you're dealing thousands of damage per hit. The exact numbers stop mattering. It becomes about watching numbers go up, triggering that same dopamine response as a slot machine. A lot of games do this on purpose, and it's exciting, but also... kind of shallow.

But what if we flip that formula? *Literally*.

Instead of dealing damage, you reverse the damage, effectively healing enemies. Status effects apply in reverse. Combos unravel backwards. Instead of killing enemies, you resurrect them back to 100% health. But you have to be precise. Overshoot by even 1 HP and you've broken the rules: Nobody can have more than 100% health. The timeline collapses and the loop restarts.

It's not about making the number bigger anymore. It's about hitting the only right number that preserves the timeline. Suddenly, mechanics that have been the foundation of every RPG for decades become a precision puzzle.

That's *GPR*—a Game Played in Reverse. The abbreviation GPR intentionally mirrors "RPG" read backwards.

### 1.2 Setting & Narrative Framework

The setting is a gritty steampunk factory: think Ekko from *League of Legends*, a mix of industrial machinery and impossible technology. It's the perfect visual language for a game that blends fantasy RPG tropes with sci-fi time manipulation.

The story starts at the end. You're standing in a ruined lab, a destroyed mech smoking in front of you, its pilot slumped dead in the cockpit. You're bloodied, exhausted, your memory fragmented. What happened here? Who are you? What's this artifact in your hand?

The artifact activates. Like a reverse VHS tape, we see the events that unfolded before: the hero presses a button on the artifact, moving backwards towards the mech that's still lying dormant. Combat starts, but instead of choosing the attack you're about to do, you're choosing the attacks you already made, un-doing combat move by move, resurrecting the mech pilot turn by turn. If you can bring each enemy back to exactly 100% health, you'll trace the path backwards through this factory. From advanced mechs down to sewer rats, from your strongest gear down to your weakest, descending layer by layer until you reach the beginning to figure out what happened.

But beware. Failing to bring an enemy back to 100% health, overshooting their health, or the enemy doing the same to you creates a timeline paradox. The loop collapses and restarts at the end.

### 1.3 Theme: Recursive Time Loop

Our idea incorporates the theme in multiple interconnected ways. Obviously, there's the narrative loop: each run literally starts at the end and rewinds to the beginning. But it goes deeper.

The combat itself is recursive. When you play a *Vulnerability* card that doubles your previous attack, which itself triggered another effect, which modified something else, you're creating cascading chains of cause and effect that ripple backwards through time. Actions in the present alter actions in the past. That's recursion.

Then there's our *Resolution Engine* (detailed in the Technical Achievement section), which calculates every possible valid path through a combat encounter. The most elegant way to build that? Recursively. Each action branches into new game states, each evaluated recursively until you hit a win condition, a loss condition, or realize there's no valid path. The engine's architecture mirrors the game's theme—timelines branching and collapsing, evaluated and re-evaluated until the paradox resolves.

### 1.4 Core Loop & Roguelike Structure

Each run is a self-contained roguelike descent from top to bottom, from the end of the story to its beginning. Similar to a reverse *Slay the Spire*, there's no exploration between encounters. In between combat, you're forced to "un-loot" chests, losing weapons and abilities, downgrading your stats. It's reverse progression. Winning a fight means moving deeper into the past, closer to the start.

Your health persists across encounters, which creates interesting resource management. The final encounter requires you to restore yourself back to exactly 100% health while solving the combat puzzle. It's the true final challenge.

When you create a paradox, the loop collapses. There is no meta-progression and no permanent upgrades between runs. Each attempt is fresh.

### 1.5 Combat System

Combat looks and feels like classic turn-based RPGs—think *Pokémon*, old *Final Fantasy*, *Chrono Trigger*. We are deliberately evoking that familiarity before twisting it.

You face an enemy. You choose an action:

- **Attacks** raise enemy HP (tied to your weapon)
- **Defensive moves** like *Heal* lower your HP, or *Shield* retroactively reduces incoming damage (tied to armor)
- **Skills** do everything else: buffs, debuffs, retroactive effect multipliers like *Vulnerability* (tied to rings)

After you choose, a reverse animation plays: attacks "un-happen," damage rewinds. Then the enemy acts.

Each enemy has a unique moveset, but their actions fall into the same three categories. Similar to *Slay the Spire*, enemies follow predetermined patterns rather than adaptive AI (e.g., Attack $\rightarrow$ Defend $\rightarrow$ Skill $\rightarrow$ Repeat). This isn't a reaction-based game; it's a puzzle game. You need to figure out the pattern, then solve the math of getting them to exactly 100% HP given their sequence of moves and your limited actions.

To keep the *Resolution Engine* computationally feasible, damage is deterministic and all actions have limited uses. You might only have 3 *Heals*, 2 *Shield* charges, 5 swings of

your sword.

Combat ends with:

- **Victory:** Enemy reaches exactly 100% HP
- **Timeline Collapse:** Enemy exceeds 100% HP (you overshot)
- **Timeline Collapse:** You exceed 100% HP (enemy overshot you)

### 1.6 Player Stats & Gear System

The player has several key stats and gear pieces:

- **Weapon + Strength:** Determines base damage (e.g., Axe(3) + Strength(2) = 5 base damage)
- **Armor + Defense:** Determines received damage reduction (e.g., Metal Armor(2) + Defense(1) = -3 damage reduction to any attack)
- **Ring:** Provides the player with a set of skills
- **Memory:** Determines how many turns of enemy actions you can see ahead
- **Time Buffer:** A stat giving you leeway to not having to bring an enemy back to exactly 100%, providing some overshoot tolerance (e.g., Time Buffer(5) means even if you overshoot an enemy's health by 5, you still win)

### 1.7 Progression & The Reverse Difficulty Curve

One of the main design challenges is the reverse difficulty curve that naturally emerges when you start a game at the end, where games are traditionally hardest. As the player loses stats and downgrades weapons, we had to come up with design principles to ensure a positive learning and difficulty curve.

We're solving this with two main design pillars:

1. **Memory degradation:** You start with a high *Memory* stat, seeing many enemy turns ahead. As you descend into the distant past, your memory "fades." You see fewer future moves. Early fights are more predictable (easier to plan), later fights require more pattern recognition and risk-taking. Lore-wise, this is explained by memory still somewhat functioning in the recent past but not as well in the distant past.
2. **Complexity vs. power:** weaker gear is more complex. Early in the run (late in the timeline), you might have a massive axe that does 30 damage—simple, powerful, boring. Later (earlier in the timeline), you're stuck with a poisoned dagger that does 5 damage over 3 turns with conditional triggers. Less raw power, more intricate puzzle-solving.

Together, these systems create a learning curve that eases players in with simpler math problems, then gradually introduces status effects, damage over time, retroactive modifiers, and cascading combos as they get more comfortable with the reverse combat system.

### 1.8 Resolution Engine & Player Feedback

Every turn, the *Resolution Engine* calculates all possible paths to victory and displays that number in the UI. It's transparency: there are 7 ways to win from here" or 0 possible solutions, you're stuck." This isn't just for the player; it's also a design tool for testing actions and balancing encounters.

The engine itself is a technical challenge we're solving recursively (see Technical Achievement section for details). Eventually, we plan to use it for procedural encounter genera-

tion, creating fights with specific “solution counts” tied to difficulty.

### 1.9 UI & Controls

The game is designed for PC with keyboard and mouse controls. The UI provides all necessary information at a glance:

- **Health bars** for player and enemy (hover for details about resistances, weaknesses, special abilities, and lore)
- **Resolution Engine display** showing the number of valid paths to victory
- **Action list** showing your available moves with base effects (hover for details, but only base values, not accounting for retroactive modifiers from previous/future actions)
- **Enemy action timeline** displaying upcoming moves (number of visible turns determined by your *Memory* stat)
- **Stats and gear panel** showing your current equipment (hover for full descriptions)

### 1.10 Engine, Art, Audio

We’re building in *Unity* as a 2D pixel art game, aiming for that SNES-era aesthetic—old *Final Fantasy*, *Chrono Trigger*’s future levels. Since neither of us are experienced pixel artists or animators, we’re leaning on *Unity Asset Store* resources, community assets (with permission), and generative AI.

For audio, we’re aiming to blend SNES-style chiptune sounds with analog synth textures, contrasting the warm, mechanical hum of steampunk machinery with the cold precision of early digital music. We’ll also try to experiment with reverse samples to reinforce the theme of time running backward. The goal is to create something that feels both nostalgic and unsettling at the same time, matching the game’s core tension between familiar RPG combat and its bizarre reverse logic.

### 1.11 Mockups and Sketches

Figure 1: Mood concept: The player confronts a fallen mech in a dark steampunk lab


Figure 2: Gameplay mockup showing the combat UI

Figure 3: Concept art of the sewer boss fight: the player faces a giant rat bare-handed.

Figure 4: Concept art of the mysterious time reversal artifact

## 2 Technical Achievement

### 2.1 Problem Statement

Playing a turn-based RPG in reverse introduces fundamentally different constraints than traditional forward combat. In a normal fight, overshooting an enemy's HP (dealing more damage than remaining HP) has no retroactive consequence: the enemy simply dies and any excess damage is ignored. In a reverse timeline, however, overshooting creates an impossible history. It would imply the enemy started with more HP than the defined maximum or that previous attacks inexplicably dealt less damage than they should have. Both outcomes break causal consistency.

Therefore this project is not a straightforward swap of damage-to-heal mechanics. Instead, each encounter becomes a constrained puzzle: the player must apply a sequence of retroactive moves that yields a forward-consistent history from the encounter's end state back to the canonical start state. The technical challenge is to design an engine

that detects and prevents impossible states (overshoot and paradox), guarantees at least one valid reverse history for generated encounters, and provides quantitative measures of timeline stability for gameplay and difficulty tuning.

### 2.2 Approach

The core technical component is a **recursive resolution engine** that treats a single encounter as a bounded-space constraint-satisfaction problem. Key design decisions and simplifications to keep the solver tractable are:

- **Bounded horizon:** encounters have a fixed number of turns $N$ (typical $N \in [6, 10]$).
- **Windowed, integer moves:** each move is encoded as a small integer vector that affects a contiguous window ending at the current reverse cursor. This keeps the action space discrete and amenable to linearization.
- **Deterministic opponent behavior:** enemy damage/healing patterns are precomputed (optionally a single deterministic trigger). We avoid reactive adversaries that inject combinatorial branching.
- **Charge & budget caps:** every move has limited charges and a temporal-energy budget constrains total use, which drastically reduces branching.

**Data structures**

- **State node:** stores the current reverse cursor $\tau$, the applied delta vector $\Delta[0 \dots N-1]$, remaining charges, and remaining budget.
- **Tree / DAG:** nodes are connected to children generated by placing one legal move at the current cursor; because we canonicalize move order, equivalent permutations collapse and the graph often becomes a DAG.
- **Transposition table (memo):** hash partial states by a compressed signature (remaining-charge multiset + a small set of window sums) to prune duplicate subtrees.

**Algorithmic approach** We will implement a depth-limited DFS with strong forward-checking and dominance pruning. The engine will:

1. Take the current end-state (enemy HP at 0) and set the reverse cursor $\tau = N$.
2. At each node, enumerate legal moves (subject to charges and budget), but only in a canonical, non-decreasing index order to avoid permutation blow-up.
3. For each candidate move, perform a *cheap feasibility check*: update the delta vector and run a fast forward simulation to detect immediate paradoxes (overshoot, impossible early death). If the check fails, prune the branch.
4. Maintain prefix upper-bounds for HP at earlier turns; if adding a move would violate any prefix bound, prune early (forward-checking).
5. Use memoization to avoid re-exploring equivalent states.
6. When a node's forward simulation yields a valid start state (i.e. $\mathrm{HP}[0] = E_{\mathrm{max}}$ with no paradox), record a solution. Continue until a cap of $K$ solutions or until budget/time limits are reached.

Below is a clear pseudocode listing for the resolution engine. Replace the verbatim block below with a proper algorithm environment if you add the appropriate package to the preamble.

**Algorithm 1** Recursive Resolution Engine (DFS with pruning)

```
procedure Solve(node, depth)
    if depth > MAX_DEPTH then
        return
    end if
    if ForwardSimIsValid(node) then
        if StartStateMatches(node) then
            RecordSolution(node)
            if SolutionsFound ≥ SOLUTION_CAP then
                return
            end if
        end if
    end if
    legal_moves ← EnumerateMoves(node)          ▷ canonical, non-decreasing order
    for all move in legal_moves do
        if not HasCharges(node, move) then
            continue
        end if
        if BudgetExhausted(node, move) then
            continue
        end if
        node2 ← ApplyMove(node, move)
        if ViolatesPrefixBounds(node2) then
            continue
        end if
        if MemoHit(node2) then
            continue
        end if
        if ForwardSimDetectsParadox(node2) then
            continue
        end if
        StoreMemo(node2)
        Solve(node2, depth+1)
        if SolutionsFound ≥ SOLUTION_CAP then
            return
        end if
    end for
end procedure
```

**Procedural Encounter Generation** We have many procedural options available for encounter generation. The prevailing workflow at present is a forward-simulation + discovery loop: generate a forward encounter, compute its terminal end-state, and then run the resolution engine to enumerate valid reverse paths back to the canonical start. The generator iterates parameters until the encounter meets design targets for solution count and timeline stability. All heavy validation and enumeration runs are performed offline in the content-generation pipeline (not at runtime). This ensures shipped encounters are provably solvable and annotated with a stability score used for difficulty tagging and hinting.

### 2.3 Risks and Mitigations

Risk 1 — Combinatorial explosion (wide tree). Mitigation: the branching factor scales with the number of distinct moves, so limit the move set and keep primitives simple; impose charge/budget caps and a maximum search depth; canonicalize move ordering and memoize equivalent states.

Risk 2 — Very few solutions (underconstrained encounters). Mitigation: parametrically loosen the win condition, e.g., if the forward simulation overkills the opponent by $m$ HP, allow an $m$-HP overshoot margin instead of forcing an exact 100% start-state as a win condition;

Risk 3 — Deep recursion / closed loops. Mitigation: bound encounter length $N$ and cap recursion depth; limit action points for indibidual abilities or implement an energy/mana budget; add closed-loop detection if time allows.

## 3 “Big Idea” Bull’s-Eye


Figure 5: Big Idea Bull’s-Eye (twist = Backwards Causality, tech = Recursive Resolution Engine and Procedural Encounter Generation).

### 3.1 Core Idea (Inner Ring)

GPR spells RPG in reverse. The game is an RPG with the core twist of being played in reverse. Attacks undo damage to your opponent and status effects apply retroactively. The goal is to undo all the damage to the opponent without overshooting 100% HP to avoid a time paradox.

### 3.2 Technical Innovation (Outer Ring)

Procedural generation of encounters with checks to guarantee the existence of one or several solutions (valid timelines). Recursive Resolution Engine that performs tree search on branching timeline possibilities to detect valid timelines and timeline collapse scenar-

ios.

# 4 Development Schedule

## 4.1 Layered Development Plan

### 4.1.1 Functional Minimum

*Goal: Prove the core concept works with a playable vertical slice.*

**Gameplay:**

- Core game loop: player turn → enemy turn → state check
- 1 handcrafted combat encounter with fixed enemy pattern
- Player and enemy each have 3 basic action types: *Attack*, *Heal*, *Vulnerability*
- Simple enemy AI: predetermined 3-move loop (e.g., Attack → Defend → Skill)
- Fixed player stats and gear (no progression yet)
- Modular system architecture: actions, gear, and enemy types implemented as data-driven components (enabling easy addition of new content in later milestones)
- Win condition: bring enemy to exactly 100% HP
- Lose condition: enemy or player HP exceeds 100% (timeline collapse)
- Basic Resolution Engine: DFS-based solver that counts valid solution paths

**UI, Art & Audio:**

- Text-based UI displaying: HP bars, available actions with charges, next enemy move, solution path count
- Placeholder sprites

### 4.1.2 Low Target

*Goal: Feature-complete vertical slice demonstrating all core mechanics.*

**Gameplay:**

- 3 handcrafted encounters with unique enemies of increasing complexity
- 2 weapons, 2 armor types, 2 rings (each providing unique actions)
- Linear roguelike structure: progress through encounters, downgrade stats/gear between fights
- Optimized Resolution Engine with memoization and pruning

**UI, Art & Audio:**

- Main menu (start game, quit)
- Simple reverse animations for combat actions
- Basic pixel art for player sprite, enemy types, and environment
- Minimal functional UI: health bars, action buttons with tooltips, enemy move preview

### 4.1.3 Desired Target

*Goal: Polished, content-complete game suitable for public playtest.*

**Gameplay:**

- Procedural encounter generation using Resolution Engine and ILP-assisted constructive seeding to create balanced puzzles with target solution counts
- 5 unique enemy types with distinct actions and properties
- 3 weapons, 3 armor types, 3 rings (each with unique actions)

- Final boss challenge: bring both enemy and player to exactly 100% HP

**UI, Art & Audio:**

- Polished UI: real-time solution counter, hover tooltips showing base effects, enemy action timeline
- Smooth reverse animations with VHS-style rewind effects
- Pixel art for all enemy types, player character, and environments
- Story framing: intro and ending sequences establishing narrative context
- Background music
- Sound effects for all main actions and events (attack, heal, paradox, etc.)

#### 4.1.4 High Target

*Goal: Portfolio-quality submission with extended content and replayability.*

**Gameplay:**

- Non-linear progression with branching paths (Slay the Spire-style map)
- Player choice between gear upgrades/downgrades at nodes
- 7 unique enemy types including mini-bosses (multi-enemy encounters, time based, etc.)
- 4 weapons, 4 armor types, 4 rings (each with unique actions)
- Tutorial sequence teaching mechanics organically through first encounters

**UI, Art & Audio:**

- Enhanced visual effects: particle systems, screen shake, timeline distortion visuals for key moments (paradox, perfect solution)
- Custom pixel art for UI elements
- Expanded lore: item descriptions, enemy information, environmental storytelling
- Expanded soundtrack (3-5 tracks)

#### 4.1.5 Extras

*Post-semester content for potential commercial release.*

**Gameplay:**

- Multiple playable characters with unique stat distributions and starting gear
- Expanded enemy roster with additional bosses
- Greater gear variety and complexity
- Fleshed-out story: dialogue windows and cutscenes between major encounters

**UI, Art & Audio:**

- Fully custom pixel art for all game elements (player, enemies, environments, UI)
- Advanced Resolution Engine visualization (e.g., animated tree/graph showing branching solution paths)
- Full soundtrack

### 4.2 Timeline and Responsibilities

Table 1: Planned timeline (owners & estimated hours).

| Week | Milestone | Owner | Task (Est. h) |
|---|---|---|---|
| W0 (Nov 4-5) | M1 | All | Finalize proposal document (10h) |
| | | All | Prepare presentation slides (4h) |
| | | All | Create UI mockups (2h) |
| | | *— Functional Minimum —* | |
| W1 (Nov 6-12) | M2 | All | Set up Unity project and version control (3h) |
| | | Ayoub | Implement game state system and turn manager (5h) |
| | | Samuel | Implement modular action system (Attack, Heal, Vulnerability) (8h) |
| | | Samuel | Design first combat encounter (enemy pattern, action balance) (4h) |
| | | Samuel | Implement fixed player stats and gear data structures (4h) |
| | | Ayoub | Implement basic Resolution Engine (DFS-based solver, path counting) (8h) |
| | | Ayoub | Implement win/lose conditions (paradox detection) (4h) |
| | | All | Create text-based UI (HP bars, action buttons with charges, enemy move display, path count) (4h) |
| | | All | Create placeholder sprites (4h) |
| | | All | Integration testing and bug fixing (6h) |
| | | All | Prepare prototype report and slides (8h) |
| | | *— Low Target —* | |
| W2-3 (Nov 13-26) | | All | Design 2 additional encounters with increasing complexity (4h) |
| | | Ayoub | Implement main menu (start game, quit) (4h) |
| | | All | Implement modular gear system: 2 weapons, 2 armor, 2 rings (each with unique actions) (12h) |
| | | Ayoub | Implement linear roguelike structure (encounter progression, gear downgrade between fights) (8h) |
| | | Ayoub | Optimize Resolution Engine (memoization and pruning) (10h) |
| | | All | Source and integrate basic pixel art (player sprite, 3 enemy types, environment) (6h) |
| | | Samuel | Implement simple reverse animations for combat actions (5h) |

*(continued from previous page)*

| Week | Milestone | Owner | Task (Est. h) |
|---|---|---|---|
| | | All | Update UI to minimal functional version (health bars, action buttons with tooltips, enemy move preview) (6h) |
| | | All | Integration testing and balancing (8h) |
| W4-5 (Nov 27-Dec 3) | M3 | All | Polish Low Target features (6h) |
| | | All | Prepare interim demo, report, and slides (10h) |
| | | *— Desired Target —* | |
| W6-9 (Dec 4-Jan 7) | M4 | Ayoub | Implement procedural encounter generation framework (14h) |
| | | Ayoub | Implement ILP-assisted constructive seeding for balanced puzzles (14h) |
| | | All | Design and implement 2 additional enemy types with distinct actions/properties (10h) |
| | | All | Expand gear variety: 3 weapons, 3 armor, 3 rings (each with unique actions) (10h) |
| | | Samuel | Design and implement final boss (player + enemy to 100% HP) (8h) |
| | | Samuel | Polish UI (real-time solution counter, hover tooltips with base effects, enemy action timeline) (10h) |
| | | Samuel | Implement smooth reverse animations with VHS-style rewind effects (8h) |
| | | All | Source and integrate pixel art for all enemy types, player character, environments (10h) |
| | | All | Write story framing: intro and ending sequences (6h) |
| | | Samuel | Compose and implement background music (10h) |
| | | Samuel | Create and implement sound effects for main actions and events (attack, heal, paradox, etc.) (6h) |
| | | All | Alpha testing, bug fixing, and encounter balancing (14h) |
| | | All | Prepare alpha documentation, report, and slides (10h) |
| W10-11 (Jan 8-21) | M5 | All | Conduct playtesting sessions with external players (8h) |
| | | All | Analyze feedback and identify pain points (4h) |

*(continued from previous page)*

| Week | Milestone | Owner | Task (Est. h) |
|---|---|---|---|
| | | All | Fix critical bugs identified in playtesting (8h) |
| | | All | Balance adjustments based on feedback (6h) |
| | | All | Prepare playtesting report and slides (8h) |
| W12-13 (Jan 22-Feb 4) | M6 | All | Final polish pass on all systems (8h) |
| | | Samuel | Record and edit gameplay video (8h) |
| | | All | Final bug fixing and stability testing (8h) |
| | | All | Write final documentation (12h) |
| | | All | Prepare final presentation slides (6h) |
| | | All | Compile and test final build (4h) |
| | | All | Prepare for Demo Day presentation (4h) |

**Total estimated workload**

Ayoub: 304 h — core systems, Resolution Engine, optimization, procedural generation, ILP seeding.

Samuel: 308 h — game design, encounters, UI/UX, animations, audio, narrative, content integration.

Joint tasks: 237 h of shared work — setup, modular systems, integration, testing, documentation, presentations (included in both individual totals above).

**Milestone mapping:**

- **M1 (Nov 5):** Formal Game Proposal
- **M2 (Nov 12):** Prototype - *Functional Minimum* complete (1 encounter, basic engine working)
- **M3 (Dec 3):** Interim Demo - *Low Target* complete (3 encounters, gear system, optimized engine)
- **M4 (Jan 7):** Alpha Release - *Desired Target* complete (procedural generation, full content, audio/visuals)
- **M5 (Jan 21):** Playtesting - External feedback and iteration
- **M6 (Feb 4):** Final Release - Polished, shippable build

**Note:** High Target features (non-linear progression, 7 enemy types, tutorial, expanded lore) are stretch goals. If development proceeds ahead of schedule, these will be prioritized during W10-13.

## 5 Assessment

### 5.1 Audience

Core audience: players who enjoy puzzle-RPG hybrids and roguelike systems (fans of Slay the Spire, puzzle-RPGs, and systems-driven indie games). Secondary: students and designers interested in provable mechanics and tool-assisted design.

### 5.2 What's Most Cool

- The reversal of familiar RPG mechanics into a precise puzzle: attacks heal, combos unravel, and numbers matter in a new way.
- The Resolution Engine: a visible, provable solver that both informs design and becomes a player-facing UI element ("X ways to win").
- Reverse roguelike structure and memory-degradation: a novel difficulty curve where simpler late-game math transitions into complex earlier puzzles.

### 5.3 Success Criteria

- Solver guarantees: all shipped encounters have at least one valid reverse history (verified offline); target $\geq 60\%$ of encounters have multiple ($\geq 2$) solutions.
- Mechanic comprehension: $\geq 75\%$ can explain the reverse goal after 15 minutes of play.
- Achieving a satisfactory move set with a balanced experience.

# II Milestone 2: Prototype

## 6 Prototype

### 6.1 Prototype Design and Construction

The purpose of this milestone was to begin constructing a working prototype that captures the core structure of **A Game Played in Reverse (GPR)**. Namely, turn-based reverse combat governed by retroactive effects and recursive causality. Our immediate goal was not yet to produce a playable experience, but to implement and validate the lowest-level logic blocks that will later enable the *Resolution Engine* to function.

The prototype currently exists in two partially overlapping forms:

- **Python CLI Prototype:** A command-line implementation focusing on combat logic, action definitions, and state transitions. It includes implemented actions for *Attack*, *Defend*, *Heal*, and *Vulnerability*, with both *Defend* and *Vulnerability* designed to apply retroactive effects across multiple turns.
- **Unity Game Shell:** A lightweight 2D framework implementing the basic round system and data-driven actions for testing in a visual environment. This version shares similar logic to the CLI but with a modular component-based architecture suitable for future integration.

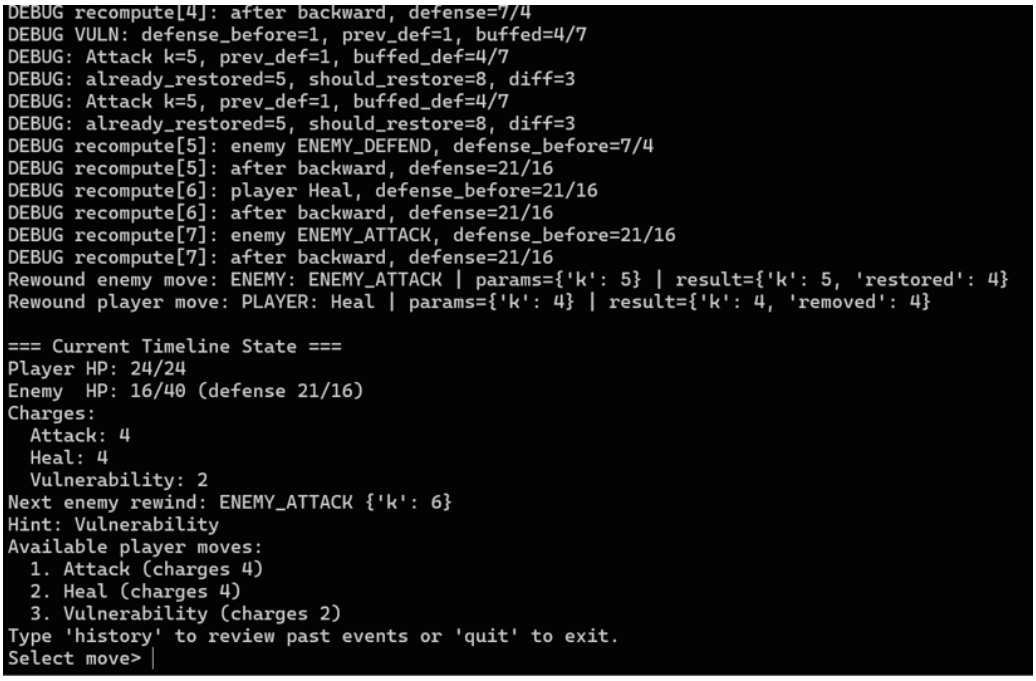


Figure 6: Command-line prototype implementing preliminary combat logic. (Not yet ready for demonstration.)

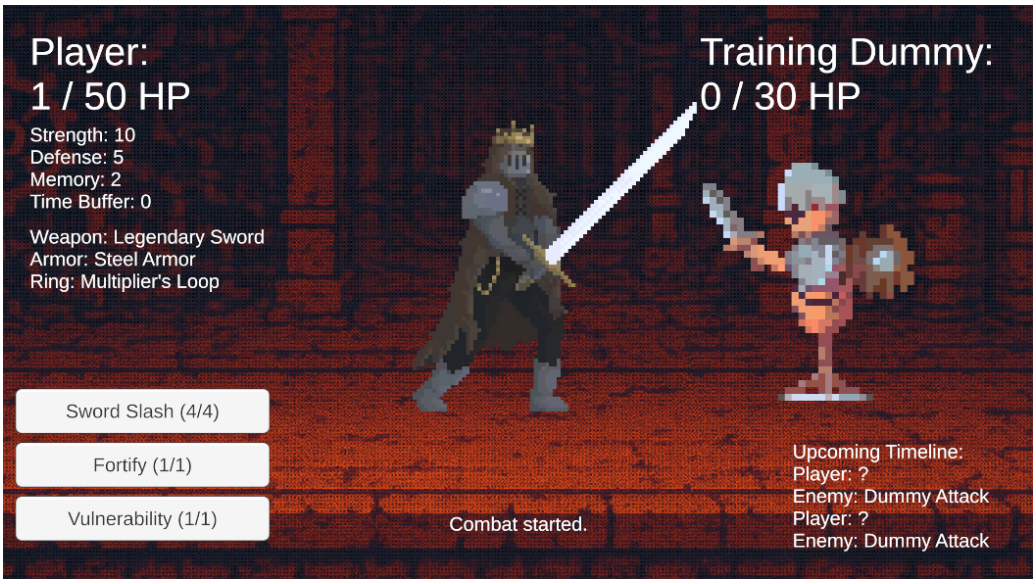


Figure 7: Unity prototype implementing a minimal combat loop with placeholder assets.

#### Implemented Logic and Current State

The CLI prototype now supports:

- Turn-by-turn battle flow for player and enemy.
- Deterministic damage and defense calculations.
- Action definitions with limited uses and charge tracking.
- Partial implementation of retroactive mechanics:
  - **Vulnerability** applies to the target for the next three turns (in forward time), which in the game's reverse logic corresponds to the previous three turns. It doubles incoming damage during that window.
  - **Defend** increases defense values across all previous turns ("future turns" from the in-game perspective), effectively recalculating prior outcomes.

Although the recursive propagation of these retroactive effects is not yet fully correct, the system correctly triggers timeline recalculation when they occur. The combat model can already record turn histories and re-evaluate partial state changes, which will directly feed into the upcoming Resolution Engine.

At this stage, both the CLI and Unity implementations are functionally similar in scope, though the Python version is more focused on correctness and debugging, while the Unity prototype serves as an early sandbox for UI and architectural design.

### 6.2 Prototyping Strategy and Goals

The strategy for this milestone was to reach the *Resolution Engine* as early as possible. While that target has not yet been reached, progress toward it was fast and structured. By building the battle logic independently first, we established the foundation required for the solver to operate on concrete, testable game states.

Key goals for this phase were:

- Define and test the complete move set and their effects.
- Ensure deterministic, reproducible combat states to serve as inputs for solver testing.

- Prepare the sandbox system around which the recursive resolution system will be developped and against which it will be tested.

The Python prototype in particular was chosen because it allows fast iteration and debugging of recursive logic without overhead from Unity's rendering and event systems. Once the recursive solver and procedural encounter generator are complete and validated in Python, they will be reimplemented (not directly ported) in Unity with refactored data structures designed for real-time use.

### 6.3 Next Steps

The next phase of work will focus on:

- **Implementing the recursive Resolution Engine** within the Python prototype. This component will enumerate valid reverse timelines and detect paradoxes arising from overshoot or conflicting state transitions.
- **Procedural encounter generation**, leveraging the solver to produce encounters with specific target properties such as number of valid solutions or timeline stability.
- **Visual and thematic consistency** in Unity, by identifying or generating pixel-art assets with a consistent steampunk style—either through curated asset packs or generative AI tools—to improve presentation before alpha development.
- **Reintegration of systems**, bridging the logic and data models between the Python and Unity prototypes once the solver is complete.

### 6.4 Summary

While not yet playable, the current prototype validates the technical foundation of the game. The core turn logic, retroactive mechanics, and timeline recalculation system have been structured in a way that prepares for the Resolution Engine and procedural content generation in the next milestone. The upcoming focus will be on completing the recursive solver and establishing a unified framework between the analytical (Python) and interactive (Unity) prototypes.

# III Milestone 3: Interim Report

## 7 Interim Report

### 7.1 System Architecture

#### 7.1.1 Building the Foundation

When we started implementation after Milestone 2, our first priority wasn't to jump straight into creating encounters or flashy features. Instead, we spent the initial week focused on building a solid foundation: setup work we'd thank ourselves for later when scaling encounters, enemies, and gear. We knew from the start that modularity was critical. With only two people and limited time, we needed a structure where adding new content wouldn't require rewriting core systems.

The decision to use Unity's ScriptableObject system for all our game data turned out to be one of our best early calls. Every action, piece of gear, and enemy type is defined as a ScriptableObject asset rather than hard-coded in scripts. This means adding a new weapon or designing a new enemy encounter can be done without touching any code, just by creating a new asset in the Unity editor, filling in some fields, and referencing existing action behaviors. The only coding required is when implementing entirely new action behaviors.

#### 7.1.2 3-Layer Architecture

Our system is organized into three distinct layers, each with clear responsibilities. The **Core layer** contains all persistent game entities: the Player class with its stats and equipped gear, Enemy instances built from EnemyData assets, and the Gear system (Weapon, Armor, Ring). These classes persist across encounters and represent the ”what” of the game.

The **Combat layer** handles turn-based runtime and state management. TurnManager orchestrates the combat loop, while CombatState provides immutable snapshots of combat at any given moment. This layer represents the ”how”: the system that makes combat actually run.

The **UI layer** is purely presentation. CombatUI, ActionButtonsUI, and various hover tooltip handlers listen to events fired by the Combat layer and update what the player sees. This separation means we can completely redesign the UI without touching combat logic, or vice versa.

The bridge between these layers is the EncounterManager, which instantiates the Player and Enemy from data assets, constructs the initial CombatState, and hands everything off to TurnManager. The Combat layer then fires events (OnStateChanged, OnCombatLog, OnCombatEnded) that the UI layer subscribes to.

#### 7.1.3 UI, Animations & Art Pipeline

Neither of us are artists, which became immediately apparent when we started looking for visual assets. For now we've been pulling from free pixel art packs on itch.io, which has given us placeholder sprites for UI, the player character, enemies, and environments.

We also experimented with PixelLab, an AI-powered sprite generator that creates pixel

art from text prompts. The results are... mixed. Sometimes you get a surprisingly solid sprite on the first try. Other times, asking for something specific like "a sword being held in both hands" just doesn't work (probably due to limited training data on specific requests). Quality varies wildly between generations.

PixelLab offers a paid tier with a plugin for the pixel-art editor Aseprite that includes the ability to create custom animations, which we're considering. Especially since I've now started learning Aseprite myself. I've been modifying existing assets—turning an arrow attack overlay into a mechanical sting animation for the Clockwork Beetle, adjusting color palettes, and experimenting with simple frame-by-frame animations. I'll try creating some assets from scratch, but realistically, with just two people, AI collaboration seems necessary, at least as a starting point.

## 7.2 Combat System Implementation

### 7.2.1 Turn Manager & The Turn Count Problem

The initial implementation of TurnManager was straightforward: player acts, enemy responds, check win/loss conditions, repeat. This worked fine in our heads and on paper. But the moment we started running actual combat with dummy actions like Attack and Heal, edge cases immediately surfaced that we hadn't considered.

The first was a simple oversight: we needed a loss condition for when HP goes negative due to healing. In reverse time, healing yourself reduces your HP (you're undoing the healing you received in forward time). If you heal too much, you drop below zero, which should trigger a timeline paradox. We hadn't implemented that check initially because we were so focused on the "overshoot above 100%" paradox that we forgot about the symmetric failure state.

The bigger headache was the turn counter. In a traditional RPG, turn counting is trivial: increment after each full round of player + enemy actions. But in reverse time, the first turn from the player's perspective is actually the *final* turn of the forward-time encounter—the killing blow. That turn only includes the player's action. The enemy doesn't act because they're already dead. Then on turn 2 (from the player's reverse perspective), the enemy acts first, followed by the player. This asymmetry means the turn number increments *after* the player acts but *before* the enemy responds—the opposite of most turn-based games.

For the UI, we embraced the weirdness and display "Turn: ? - X" at the top of the screen, with the question mark representing the unknown final turn number and X being the internal forward turn count starting from 0, which is also what the code uses internally.

### 7.2.2 Forward Simulation: The Right Choice

The most critical design decision we made was how to actually implement action logic. We identified two main approaches early on.

**Option 1: Reverse Logic Directly.** Implement each action with reverse-time behavior. A 10-damage attack would heal for +10 HP. A buff that increases damage would retroactively *reduce* damage. This seemed simple at first—just flip the signs and invert the effects, right? But the more we thought about complex actions, the messier it got.

Consider the Duplicate action, which from the player's perspective retroactively doubles the effect of their previous action. If you Attack for 10, then Duplicate, the Attack should have been worth 20 all along. To implement this with pure reverse logic, you'd need to track what the previous action was, what damage it dealt, then reverse-calculate the new

damage value. Simple enough for one action. But once you start stacking skills: Duplicate on Duplicate, conditional triggers, multi-turn status effects—you're maintaining a complex history of past actions and damage values, wrestling with awkward reverse logic to make sense of them. Which led us to a realization: if we're already tracking action history, why not just replay it forward?

**Option 2: Forward Simulation.** The player and enemy choose actions in reverse time, but under the hood, the system runs a forward-time simulation of the entire encounter after every action. We simulate what would happen if time played forward from the current state, applying all previously chosen actions in chronological order (the reverse order of how the player chose them). Keeping track of multipliers, stacks, and flags becomes trivial—it's just normal RPG combat logic. We compare the simulated end state to the canonical end state (enemy at 0 HP, player at their predetermined ending HP), calculate the delta, and apply that delta to the current reverse-time state.

We chose Option 2. It's conceptually elegant: we're literally "shifting the timeline" to stay consistent with forward causality. More importantly, it's dramatically easier to implement. Actions just do what they normally do in a traditional RPG. Attacks reduce HP. Buffs set multiplier flags. Duplicate sets a 2x multiplier that gets consumed by the next action in the simulation. We use a lightweight, mutable ForwardSimState to run the simulation, then throw it away and apply the calculated delta to the immutable CombatState.

The main downside is performance—we're re-simulating the entire encounter after every action. If we base the Resolution Engine off this approach (which we haven't implemented yet), that means forward-simulating the full fight for every possible action in every possible game state. However, with limited action charges and deterministic enemy patterns, the computational cost remains negligible. And the upside is huge: adding new actions became trivial once this system was in place. Even complex systems like stat-based defense scaling fell naturally into this framework. You implement the forward-time logic once, and the reverse-time behavior emerges automatically from the delta calculation.

### 7.3 Encounter Design

For the interim demo, we designed two consecutive encounters that serve as the first practical showcase of the backwards oriented combat system. Since the project at this stage focuses on implementing and validating core mechanics rather than building complex enemy behavior, both encounters remain intentionally simple. Their purpose is to verify that turn sequencing, retroactive effects, and deterministic state updates function correctly under controlled conditions.

Because the current build does not yet include any map flow or inter encounter structure, the two fights execute back to back. As a result, we made the design choice that the player's HP persists between encounters. This persistence is meant to establish the long term structure the full game will eventually rely on, where health management is one of the few resources that span multiple fights. Although the present encounters do not yet stress this mechanic in any meaningful way, the persistence rule is already in place in order to ensure consistency with the intended final system.

The two encounters included in this milestone are:

- **Clockwork Beetle**: a low complexity enemy used to validate the basic turn loop, deterministic damage rules, and limited use actions.
- **Cinderplate Armadillo**: a slightly more demanding encounter that tests the interaction between multiple retroactive effects and showcase that recalculations remain stable across a longer timeline.

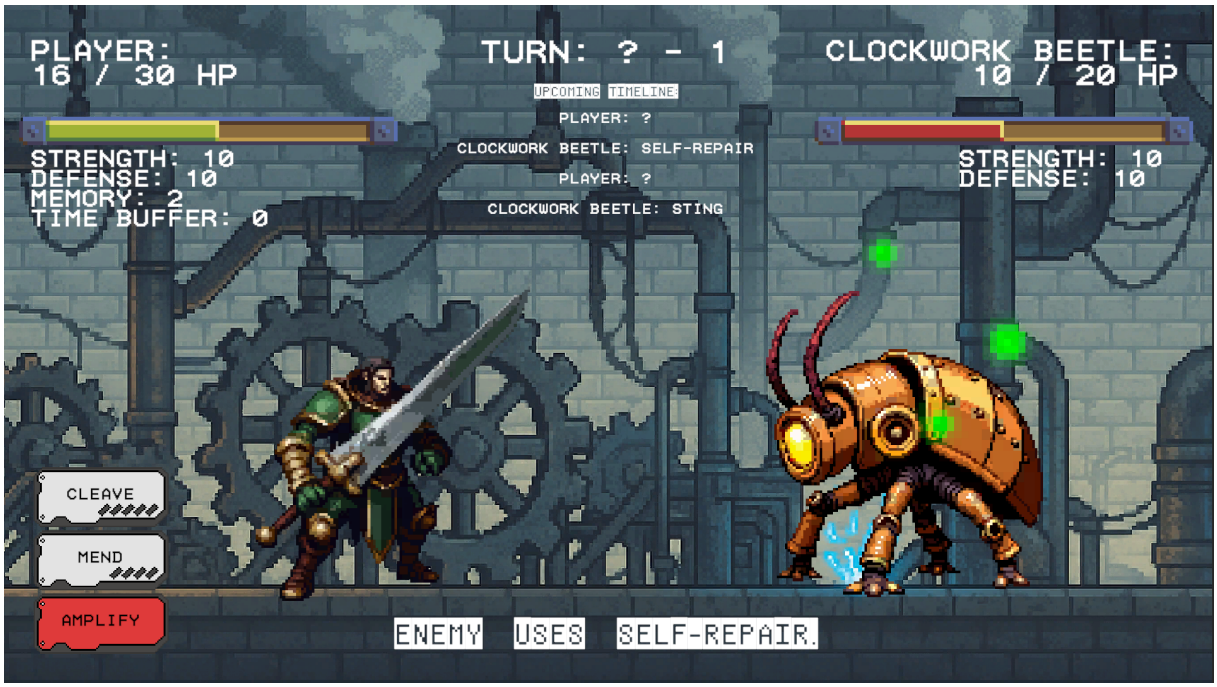


Figure 8: The screenshot shows the current prototype state: the Player and the Clockwork Beetle are instantiated from their ScriptableObject data, the immutable CombatState is rendered through the UI (HP bars, stats, charges, and enemy-timeline preview), and the TurnManager has just processed the Beetle's Self-Repair action. All buttons and values come directly from the modular action/gear/enemy architecture.

Both encounters follow fixed, predictable patterns as specified in the Low Target scope, allowing players to reason about upcoming actions using the Memory stat and to focus on the central mechanic of reconstructing a valid reverse timeline.

### 7.3.1 Clockwork Beetle

A mechanical insect that serves as the simplest enemy in the current build.

Figure 9: Clockwork Beetle sprite.

The Clockwork Beetle is intentionally designed as a basic introductory encounter. Its purpose is not to challenge the player with complex patterns but to provide a controlled environment for testing the core mechanics of reverse combat, deterministic turn resolution, and limited use actions.

The beetle has a maxHP of 20 and starts dead at 0. The Player starts at 11 out of 30

maxHP

The enemy has only two abilities, which it alternates in a fixed loop:

- **Sting**: deals 5 damage to the player.
- **SelfRepair**: heals itself by 5 HP.

The player has access to three actions:

- **Cleave**: deals 10 damage to the enemy.
- **Mend**: heals the player by 10.
- **Amplify**: has one charge and triples the effect of the next action.

The Clockwork Beetle has base strength and defense values of 10, which are the identity values meaning the damage deal and received is unchanged from those defined in their respective abilities.

The Clockwork Beetle encounter has multiple valid solutions, offering players flexibility in how they approach the puzzle. The **fastest solution** is straightforward: use Cleave twice in succession (Turn 0 and Turn 1) to restore the Beetle to full HP before it heals itself: Cleave $\rightarrow$ Sting$\rightarrow$ Cleave This is the most efficient path, requiring only two turns but it raises player HP to 16.

However, players can also opt for **HP-minimizing strategies** that extend the encounter across more turns, for example: Mend $\rightarrow$ Sting $\rightarrow$ Cleave $\rightarrow$ Self-Repair $\rightarrow$ Cleave $\rightarrow$ Sting $\rightarrow$ Mend $\rightarrow$Self-Repair $\rightarrow$ Cleave wins the encounter while minimizing player HP to 1.

This approach trades turn efficiency for preserving player HP, demonstrating that the puzzle allows for different optimization goals—minimize turns, minimize HP loss, or balance both.

Any use of amplify (after cleave or mend) leads to an immediate HP over- or undershoot, respectively, and loses the encounter.

The encounter serves as a low-stakes sandbox where players can experiment with the core actions (Cleave, Mend, Amplify) and internalize the reverse-time mechanics. The primary design goal at this stage is to confirm that the underlying combat logic behaves consistently across different action sequences, rather than to test complex decision-making. Players can freely explore the solution space without severe punishment, building intuition for how forward simulation translates player choices into reverse-time outcomes.

### 7.3.2 Cinderplate Armadillo

A heavy, steam powered armadillo whose reinforced plating grows stronger every turn.

The Cinderplate Armadillo represents a more interesting test of the reverse combat logic. It has a maxHP of 35.

It has only one action available to it:

- **FortifyDefense**: increases its defense by 5.

Even though the Armadillo has no offensive capabilities to restore the player's HP or create a loss condition through overshoot, the encounter remains non trivial. This comes from the fact that FortifyDefense is a retroactive effect. When the enemy uses it, the defense stat shown on the next visible turn does not change. A Cleave executed immediately afterwards still deals the usual 10 damage. However, FortifyDefense increases the defense value from that turn forward in the forward timeline. As a result, all previously

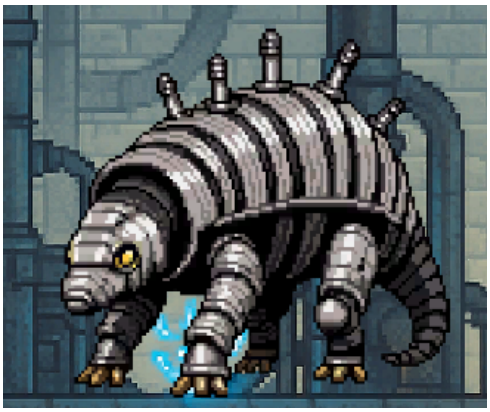

Figure 10: Cinderplate Armadillo sprite.

applied Cleave actions are recalculated with higher defense values. The further back in the timeline those Cleaves occurred, the more they are reduced.

This encounter requires the player to use Amplify at some point, which makes it a useful test of whether the backward simulation handles both defensive retroactivity and amplified moves in a consistent manner. Manual verification confirms that the current system correctly rewinds and reapplies all state changes.

One valid sequence is the following:

$$\text{Cleave} \rightarrow \text{Fortify} \rightarrow \text{Cleave} \rightarrow \text{Fortify} \rightarrow \text{Cleave} \rightarrow \text{Fortify} \rightarrow \text{Cleave} \rightarrow \text{Fortify} \rightarrow \text{Cleave} \rightarrow \text{Fortify} \rightarrow \text{Amplify} \rightarrow$$

When evaluated in reverse, each Cleave operates under a different effective defense value. Earlier Cleaves are weakened the most, while the latest Cleave carries the strongest effect. This creates a layered structure where the player must navigate the gradually increasing defense and choose a moment to apply Amplify in order to reach exactly 100 percent HP.

Although the encounter still contains only a single class of solutions in its current form (any permutation of 5 Cleave and 1 Amplify), it serves as a more complete stress test of the retroactive simulation rules and ensures generality for future, more complex encounters.

### 7.4 Current Progress

We have completed our **Functional Minimum** and the majority of our **Low Target** objectives. All the groundwork is done. The core three-layer architecture is stable, the forward simulation system works reliably, and we have two fully playable encounters that validate the reverse-time combat mechanics. Due to our ScriptableObjects approach adding new enemies, gear and actions is straightforward for future milestones.

For **Milestone 4 (Alpha Release)**, our priorities are:

- Implement Resolution Engine & UI display showing solution count
- Design and implement at least 3 additional encounters with unique actions
- Add between-encounter structure (map flow, gear selection, "unlooting", story beats, etc.)
- Add game juice: (idle) animations, hit feedback, background motion to make it feel alive
- Replace placeholder sprites with cohesive pixel art
- Add background music and sound effects

- Integrate story framing (intro sequence)

Besides the resolution engine, the hard technical work is behind us. Most of what remains is content creation and making the game feel alive. If we maintain pace, we should hit our Desired Target for Alpha.

# IV Milestone 4: Alpha Release

## 8 Alpha Release

### 8.1 Progress Overview

The progress between the interim milestone and the alpha has been substantial. The game has reached a playable and somewhat polished state, although not all of our target goals have been reached.

#### 8.1.1 Alpha Goal Breakdown

Below we revisit the Alpha goals listed in the interim report and summarize what was achieved, what changed in priorities, and what remains.

**Implement Resolution Engine & UI display showing solution count: Not implemented.** While previously framed as a top priority, we deliberately de-prioritized this feature for Alpha. We observed that early player experience is dominated by what is immediately visible: readability, clarity of feedback, onboarding, and overall polish. In contrast, the resolution engine's output (solution counts and solver-derived feedback) is inherently harder to interpret without additional UI explanation and player mental models. For the Alpha, we therefore chose to focus on a fully playable, handcrafted encounter set and a strong presentation layer, while keeping the resolution engine as a major post-Alpha deliverable. This component remains core to the long-term vision and was already described as the project's key technical achievement in the original proposal.

**Design and implement at least 3 additional encounters with unique actions: Largely achieved.** Three encounters are currently in a fully playable state: the Beast Maker (the "final boss" fought at the start of the run), the Cinderplate Armadillo, and the Rot Rat (a plague-infested enemy fought bare-handed at the end of the run). In addition, two further encounters are implemented as work-in-progress previews. Their scenes, enemies, and basic combat logic are functional, but their animations are still incomplete. All encounters still require balancing and tweaking, but we hope to gather focused playtesting feedback to fine-tune difficulty, pacing, and clarity before the next milestone.

**Add between-encounter structure (map flow, gear selection, "unlooting", story beats, etc.): Complete.** The game now features a full node-based progression map (i.e., a *Slay the Spire*-style encounter map presentation, adapted here as a linear route with dedicated loot nodes). Progression is structured from the top of the tower down to the sewer level at the bottom, with in-between loot rooms where the player "unloots" gear and abilities downgrade accordingly, consistent with the reverse-progression core loop described in the proposal.

**Add game juice (idle animations, hit feedback, background motion): Substantial progress (ongoing).** Combat actions now have clearer animations, the background is animated, and certain abilities trigger camera motion to emphasize impact and timing. The game is noticeably closer to a "finished-feeling" play experience. However, hit feedback and ability-specific clarity (especially in complex turns) still needs further iteration.

**Replace placeholder sprites with cohesive pixel art: Complete (and ongoing with content expansion).** Placeholder art has been largely replaced by cohesive pixel art across the game. This was enabled by an AI-assisted sprite and animation pipeline (PixelLab.ai), which allowed a two-person team to reach a consistent visual baseline despite limited artistic talent.

**Add background music and sound effects: Mostly incomplete.** Partially achieved. Five music tracks are currently integrated into the build: one main menu theme, three encounter-specific tracks, and one overworld track. No sound effects have been implemented yet, which limits moment-to-moment combat feedback and perceived responsiveness.

**Integrate story framing (intro sequence): Partial.** Partial. While the narrative remains deliberately minimal and indirect, the Alpha build now includes an introductory text, enemy descriptions, item descriptions, and main menu framing that establish tone, world logic, and thematic context. Storytelling is primarily environmental and systemic rather than explicit, reinforcing the reverse-progression concept without locking the project into a rigid canonical narrative.

**Additional major progress (not explicitly listed as an Alpha goal).** A large amount of effort went into **communicating the mechanics clearly**. It appears, in hindsight, that should have been listed as an explicit milestone objective. A complete tutorial was implemented in the first level to explain both the basic loop and the less intuitive reverse-time implications as clearly as possible without excessive verbosity. Additionally, hovering over abilities (player and enemy) now displays explanations in both forward and backward time, including HP outcome implications. Hover tooltips were also added for individual stats. This has substantially improved transparency for a concept that is inherently unintuitive on first contact.

### 8.2 Implementation Details

This subsection expands on the major Alpha changes from an implementation perspective. Not every system is described exhaustively. The focus is on additions that most directly improved playability and clarity.

#### 8.2.1 Run Structure and Progression Map

The Alpha build introduces a complete between-encounter structure through a node-based encounter map in the style popularized by *Slay the Spire*. In our case, the route is intentionally linear to keep pacing predictable and to support tutorialization, but the presentation and mental model remain the same. The player starts at the top of the tower and descends toward the sewers, with dedicated loot rooms between combat nodes.

The loot rooms implement the project's reversal loop. Instead of acquiring stronger gear as the run progresses, the player *unloots* equipment in a controlled and readable sequence. This also creates natural breakpoints where the player can reassess their build between fights.

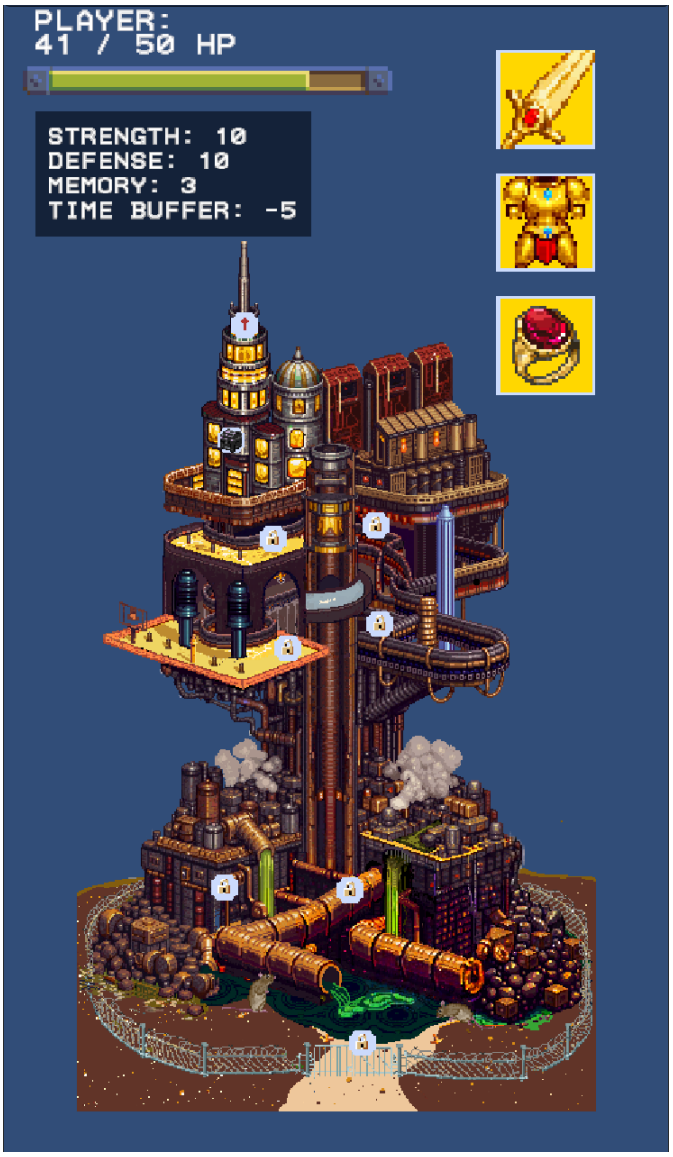


Figure 11: Node-based progression map with linear descent from the tower into the sewer level and dedicated loot rooms between encounters.

### 8.2.2 Gear System and Loot Rooms

A complete gear system was added and integrated into ability selection. Gear pieces define both stat modifiers and the player's available actions, which makes build identity readable and easy to communicate through UI.

The gear progression follows the intended reversal. Early "future" gear is stronger and simpler. Lower level gear is weaker but introduces more complex interactions. Loot rooms handle unlooting as a structured downgrade process, affecting both equipment and, where relevant, associated stats.


Figure 12: A loot room where the player unloots gear and transitions to a downgraded loadout between encounters.

### 8.2.3 Mechanic Communication, Timeline UI, and Player Transparency

A large portion of Alpha effort went into making the game understandable without external explanation. The first level now includes a tutorial that covers the core mechanics, including the unintuitive implications of reverse-time actions. To reduce ambiguity during combat, we also implemented more transparency UI: ability hover tooltips explain effects in both forward and reverse time, and stat hover tooltips clarify derived values.

A key usability addition is the **Memory** stat. It controls how many enemy turns the player can see in the timeline preview. Planning depends heavily on knowing the enemy pattern, since the player is reconstructing a consistent timeline rather than reacting to surprises. The Memory stat is also justified in-world. The player's memory of the past is incomplete at first, then becomes clearer as enemy actions "remind" them of what happened and reveal more of the recent past.

This set of changes improved puzzle readability substantially. It also enabled encounter tuning around partial information rather than assuming full pattern knowledge.


Figure 13: A single combat screenshot illustrating three clarity features: tutorial text guidance, timeline preview (limited by the Memory stat), and a hover tooltip describing an ability in both forward and reverse time.

### 8.2.4 Time Buffer and Timeline Stability

We also added a **Time Buffer** stat. It functions as a stability margin that allows the player to temporarily overshoot the enemy's max HP without collapsing the timeline immediately. Conceptually, it represents a limited "elasticity" in the timeline where short-lived inconsistencies can be corrected later.

In the current implementation, overshooting by an amount within the Time Buffer does not count as a win. It only prevents the timeline from collapsing. This keeps the win condition strict while still allowing the player to explore sequences that might later become consistent if the enemy reduces its HP back under max HP through reverse-acting healing or retroactive fortification effects. Whether we later treat an overshoot within the Time Buffer as an immediate victory is still open and will depend on playtesting and balancing.

## 8.3 Playability Evaluation and Further Plans

At the Alpha milestone, the project has reached a consistently playable state with a complete run structure and just enough content variety to communicate the intended experience. The core loop is now coherent end-to-end: the player progresses through a structured sequence of encounters, unloots gear between fights, and solves deterministic combat puzzles by reconstructing a valid reverse timeline. Visual cohesion and basic animation work further support the feeling that this is a game rather than a technical demo.

At the same time, the Alpha also made clear that reverse-time combat is not only mechanically unusual, but also cognitively fragile. Even when the player understands the concept, outcomes can still feel surprising because causality is inverted and effects propagate retroactively. The most important improvement relative to previous milestones is therefore not purely mechanical but communicative. The tutorial encounter now explains the foundational rules, and the UI exposes far more state and intent through timeline preview,

hover tooltips for abilities in both forward and reverse interpretation, and stat explanations. This brings the experience closer to a solvable puzzle system rather than a sequence of opaque recalculations.

The absence of the Resolution Engine is the main gap relative to the original Alpha target list. This was a deliberate tradeoff. For the Alpha, handcrafted encounters are sufficient to deliver a complete and testable experience, while the resolution engine's benefits are less visible to a first-time player without substantial additional UI scaffolding. It remains a core planned feature for post-Alpha development. With limited ability charges, deterministic enemy patterns, and a bounded number of turns, the solution space is expected to remain tractable rather than exploding combinatorially. This should also avoid issues such as recursion stack overflow from cyclical patterns. In addition, the newly introduced Time Buffer stat is intended to increase timeline stability by allowing temporary overshoots without immediate collapse, which should expand the number of solvable timelines in difficult encounters.

In the next development phase, work will focus on three areas:

- **Resolution Engine and stability-driven encounter tuning.** Implement the solver and integrate a UI representation that communicates results in a player-meaningful way. This enables procedurally generated encounter parameters (e.g., HP and stat ranges) while keeping encounter mechanics hand designed, with the goal of maximizing solvability given the player's persistent HP and current gear.
- **Content completion and encounter breadth.** Finalize the two work-in-progress encounters and expand the roster with additional unique enemy actions. Short, localized tutorial prompts will be added in later encounters to introduce enemy-specific mechanics without repeating the full onboarding sequence.
- **Feedback, audio, and polish.** Expand audio feedback through sound effects and ambient sound layers, and further refine the existing music integration, including more distinct tracks for individual encounters as well as dedicated music for the ending sequence.

Overall, the Alpha milestone validates that the project's core idea can support a complete playable loop, but it also highlights that the long-term success of the concept depends on continued investment in transparency and feedback. The next milestone therefore prioritizes systems that increase player trust in outcomes, alongside continued content expansion and audiovisual polish.