Bryce Fox
CS3100 - Project 6
December 5, 2025

1. Introduction

This design document outlines the architecture of the Five Nights at Rowdy Raider's Mob Spawn Manager, a Multiset abstraction serving as the core enemy container for a dungeon crawler RPG engine. The system models a mob spawning table using a `HashTable<string, int>`, where each string is a mob identifier and the integer tracks its active quantity.

The game context is a survival cycle in which players face escalating waves of enemies each hour, culminating in boss encounters such as "Rowdy Raider" and his evolved form, "Golden Rowdy Raider." To win, the player must endure five full cycles of waves and bosses.

The Hash Table supports this progression by efficiently tracking populations of key mobs (e.g., Raider Wolf Pups, Tunnel Bugs, WPAFB Alien Escapees, and the Spirit of Orville & Wilbur Wright). It maintains dynamic counts against shifting population limits and uses mob names as lookup keys for associated statistics, ensuring a clear separation between spawn control and entity data.

2. Design Philosophy

Prioritized Qualities
The Mob Spawn Manager prioritizes efficiency and extensibility. Efficiency is critical because the spawn system is polled constantly within the game loop. Using a Hash Table ensures average constant time lookups and updates, preventing lag during high stress scenarios such as the Golden Rowdy Raider's multi enemy spawns.

Extensibility supports content updates and modding. Since mobs are stored as dynamic string keys mapped to counts, new enemies can be added or modified without altering the class itself. This design allows developers or modders to introduce features like secret holiday bosses (Easter's Rowdy Rabbit-Wolf Hybrid) with minimal effort.

Intended Client
The primary client is the game's Wave Manager System, which polls the Mob Spawn Manager to determine current mob counts. By abstracting storage and memory details, the manager provides simple, reliable answers that drive game logic while keeping complexity hidden.

3. Core Operations

1. Spawn Mob
    - Behavior: Adds a new enemy by incrementing its count; if absent, creates a new entry with count = 1.
    - Complexity: O(1) average.
    - Edge Cases: Possible integer overflow, though unlikely.
    - Support: Hash Table enables direct key lookup and fast updates.
2. Kill Mob
    - Behavior: Decrements the count for a defeated enemy.
    - Complexity: O(1) average.
    - Edge Cases: Prevents negative counts by validating before decrement.
    - Support: Constant time retrieval ensures smooth performance during rapid events.
3. Get Quantity
    - Behavior: Returns the current population of a given mob type.
    - Complexity: O(1) average.
    - Edge Cases: Nonexistent keys return 0 instead of error.
    - Support: Primary lookup for spawn limits; remains efficient regardless of table size.
4. Reset Table
    - Behavior: Clears all entries or resets counts to zero, used at gameplay transitions.
    - Complexity: O(N).
    - Edge Cases: Safe to call on an empty table.
    - Support: Iterates through all buckets; acceptable since it runs outside active gameplay.

4. Set Operations

1. Merge Waves `[union_with()]`
    - Gameplay: Combines two spawn tables (e.g., during "Golden Hour"), summing mob counts without clearing the current state.
    - Manipulation: Iterates through the incoming table; adds counts to existing entries or creates new ones.
    - Complexity: O(N), where N = number of unique mobs in the incoming set.
    - Edge Cases: Possible integer overflow if very large waves are merged; capped at max value.
2. Stay In Memory Check `[intersection_with()]`
    - Gameplay: Identifies mobs common to consecutive waves so their assets remain loaded, improving efficiency.
    - Manipulation: Iterates through the smaller table; checks for matches in the larger one and records shared mobs.
    - Complexity: O(N), leveraging constant time lookups for fast background execution.
    - Edge Cases: Empty intersection yields an empty set, signaling no assets can be reused.

5. Extension Feature

This feature allows the player to bypass a non boss wave instantly by escaping through the Wright State tunnel system. It introduces a private counter escape_charges, initialized to 3. When triggered, the system checks two conditions:

- Charges remain `(escape_charges > 0)`
- Current round is not a boss `(boss_active == false)`

  There should also be a small chance (~10%) that a tunnel bug squad appears during escape, keeping the player from feeling completely safe.

This can be explained with the following pseudocode:

```
// tunnel escape button availability
if (escape_charges > 0 && !boss_active) {
    escapeButtonVisible = true;
} else {
    escapeButtonVisible = false;
}
```

And

```
// tunnel escape button pressed
if (escapeButtonVisible && escapeButtonToggled) {
    tunnelEscape();
}
```

And

```
// on tunnel escape, jump to next round but roll for a rat flood
void tunnelEscape() {
    clearHashTable();
    jumpToNextRound();
    RNG = roll();
    if (RNG > 0 && RNG < some_magic_number) {
        spawnTunnelRatBatallion(); // spawn 1 million tunnel rats!
    }
}
```

6. UML Diagram / Abstraction Boundary

Mob_Spawn Manager
- spawnTable: HashTable<string, int>
- escape_charges: unsigned int

Mob_Spawn Manager
- max_capacity: unsigned int
+ MobSpawnManager()
+ spawn_mob(mobName: string): void
+ kill_mob(mobName: string): void
+ get_quantity(mobName: string): int
+ reset_table(): void
+ union_with(other: MobSpawnManager): void
+ intersection_with(other: MobSpawnManager): MobSpawnManager
+ trigger_escape(is_boss_active: bool): bool

Where private members are represented with (-) and public members are represented with (+).

Private Members (Internal Data):
The spawnTable is kept private to enforce encapsulation, ensuring that the internal memory layout and hashing logic cannot be corrupted by external classes. Similarly, escape_charges and max_capacity are private to maintain strict invariants. For example, ensuring that escape charges strictly remain between 0 and 3. By hiding these variables, the design prevents the Game Director from accidentally setting values that would break the game's difficulty balance (such as setting negative charges or lowering the capacity limit).

Public Interface (External Behaviors):
The member functions are public to define the interface for the client (the Game Director). These methods allow the main game loop to manipulate the spawn table. This includes spawning mobs, checking counts, or triggering escapes all in a safe, controlled manner. This ensures that every modification to the game state allows the Mob-Spawn Manager to run its necessary validation logic before updating the internal Hash Table.

7. Trade-Off Analysis

Alternative Considered: Sequence (Dynamic List)
I compared my chosen Hash Table architecture against a `Sequence<String>` that would store each mob instance individually. In that alternative design, the container would look like {"Tunnel Bug", "Tunnel Bug", "Raider Wolf Pup"} rather than the Hash Table's memory-efficient approach of {"Tunnel Bug": 2, "Raider Wolf Pup": 1}.

While implementing a Sequence is significantly simpler than managing a hashed container, I chose against it due to its linear time complexity. To perform the get_quantity() operation in a Sequence, the

system would be forced to iterate through the entire list and increment a counter for every matching string found. In contrast, the Hash Table instantly locates the input key and returns the stored integer value. Given that the Game Director polls this Multiset at some high frequency, the Sequence's lookup time would likely cause performance degradation during intense mob waves where fast decisions are crucial. For this job, the Hash Table ensures stable performance.

Design Comparison:

Feature	Hash Table	Sequence
Internal Data	Map<string,int>	List
Concept	Inventory	Roster
Spawn Logic	Find Hash, Increment Integer	Append String to End of List
Count Logic	Get Integer (Instant)	Iterate & Count (Slow)

8. Alternative Design Sketch

If the Mob-Spawn Manager were implemented using a Sequence (such as a Dynamic List) instead of a Hash Table, the internal architecture would fundamentally change.
In this alternative design, the class would hold a list of strings representing every single active entity individually.

- Storage: Instead of storing {"Tunnel Bug": 3}, the memory would look like {"Tunnel Bug", "Tunnel Bug", "Tunnel Bug"}.
- Spawn Operation: This would simplify to a basic append operation. Adding a mob would just place the string at the end of the list, which is actually faster than hashing.
- Get Quantity Operation: This would change from a lookup to a calculation. The class would not "know" the count instantly; it would have to iterate through the entire list and count the occurrences of the specific name every time the Game Director requested a status update.

9. Evaluation Plan

To validate correctness, efficiency, and extensibility, the following tests would be implemented:

1. Unit Testing
    - Lifecycle: Spawn a mob, confirm count increments, kill one, confirm decrement.
    - Boundary: Ensure `kill_mob()` on zero population does not produce negative values.
    - Extension: Trigger `trigger_escape()` three times to clear the table; verify the fourth attempt is denied.
2. Stress Testing

    - Load: Populate with 10 mob types and 100 entities.
    - Measurement: Record `get_quantity()` execution time.
    - Pass Criteria: Lookup must remain constant time regardless of table size; linear scaling indicates failure.
3. Extensibility Verification
    - Input: Add a new mob key (e.g., "Easter_Rowdy_Rabbit_Wolf").
    - Pass Criteria: System creates a new entry without errors or recompile, confirming easy content updates.

10. Conclusion / Reflection

The Mob-Spawn Manager successfully prioritizes efficiency and extensibility. By employing a Hash Table, the system ensures performance during heavy-load gameplay moments, such as the "Golden Rowdy Raider" ultra-boss fight, preventing the lag associated with linear searches. The primary trade-off accepted was higher memory overhead (unused buckets) in exchange for speed. Future iterations would address the "5-Day" campaign structure by implementing Serialization to save/load progress.

- Abstraction: The Mob-Spawn Manager presents a simple, high-level interface (e.g. spawn_mob, trigger_escape) to the Game Director. The driver sends orders without needing to understand the hidden math or memory management.
- Encapsulation: By marking data members like escape_charges and spawnTable as private, the design enforces safety restrictions. External areas cannot accidentally corrupt the memory or set bad values.
- Composition: The Manager utilizes composition by owning a Hash Table. This allows the class to enforce on the data structure specific rules such as blocking escape attempts when necessary.

References

[1] CppReference, "std::unordered_map," cppreference.com, 2023. [Online]. Available: Link
[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to Algorithms, 4th ed. Cambridge, MA: MIT Press, 2022, ch. 11.