Wave-Based Encounter System

A Data-Driven, Modular Combat Orchestration Framework for Action Roguelites and Dungeon Crawlers

---

Project Context: Open Veins (Action Roguelite Prototype)
Author: Guilherme Martins (www.guifamar.com)
Engine / Tools: Unreal Engine (Blueprints Ecosystem)
Document Version: 1.1 (Production Ready)

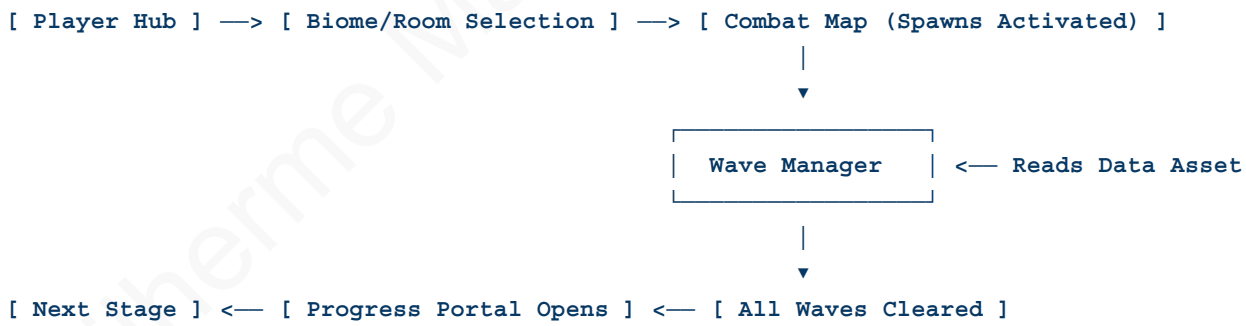
1. Executive Summary & Industry Context

The **Wave-Based Encounter System** is a highly decoupled, production-grade combat orchestration framework designed specifically for session-based progression models, such as action roguelites (e.g., *Hades*) and modern dungeon crawlers. The core objective of this framework is to handle complex entity lifetime loops, including asynchronous timed spawning, continuous aggregate state tracking, and reactive progression gating, entirely driven by isolated structural assets.

By moving room and encounter rules completely out of hardcoded node topologies and injecting them at runtime via engine data objects, design pipelines achieve maximum velocity. Level designers can establish varying encounter lengths, modify monster composition, adjust pacing intervals, and control gated rewards across different biomes without opening a single Blueprint graph.

Industry Scalability Note: This architecture was built and validated during development for Open Veins, ensuring seamless integration with underlying systems like gameplay tag dictionaries, global progression maps, and loose actor communication hooks.

Architectural Flowchart: Macro Progression Loop



2. High-Level Design & Player Experience Loop

2.1 Core Pillars

- **Code-Free Iteration:** Complete isolation of numerical tuning, monster scaling, and spawn positions inside distinct data containers.
- **Deterministic Progression:** Explicit gating ensuring zero traversal ambiguity; room progression actors state-lock

doors/portals until live array registers absolute depletion.
- **Biomic Adaptability:** Native support for distinct pacing personalities, enabling immediate reconfiguration from low-density grunt waves to asymmetric boss arenas.

2.2 Step-by-Step Runtime Experience

1. **Entry & Initialization:** The player moves into an uncleared combat zone. The local traversal asset (Portal) defaults to an inactive, non-traversable state with disabled interaction channels and zero particle emission.
2. **Staggered Ingress:** The system reads the local index configuration and executes sequential entity instantiation from specific physical anchors based on pre-defined delay offsets.
3. **Engaged Combat State:** The player engages the entities. The orchestrator tracks living instances reactively, without relying on persistent performance-heavy polling tick frames.
4. **Wave Transition & Validation:** Upon the elimination of the active wave's final entity, the orchestrator evaluates structural arrays. If a subsequent wave index is found, it queues the next wave block following a localized cooldown period.
5. **Encounter Completion:** When all wave indexes are exhausted, and all arrays return zero active elements, the orchestrator triggers progression dispatchers, updating the Portal to an open state.

3. Data-Driven Architecture & Struct Hierarchy

To eliminate systemic code modification during gameplay balance sweeps, the design maps variables to a nested hierarchy of native Unreal Engine Structs compiled inside a **Primary Data Asset (PDA)**.

Structure Name	Field Key	Type Definition	Functional Responsibility
F_BiomeConfig	BiomeTag	GameplayTag	Global unique key mapping for progression tracking and unlocks.
	MapSequence	Array<Name>	Ordered list of level asset identifiers defining the biome's seed.
	BossMap	Name	Target level name dedicated to the final boss climax encounter.
	Waves	Array<F_WaveConfig>	Sequential container containing individual wave configurations.
F_WaveConfig	NPCsToSpawn	Array<F_NPCSpawnInfo>	Collection of specific individual entity descriptors allocated to

Structure Name	Field Key	Type Definition	Functional Responsibility
			the wave.
F_NPCSpawnInfo	NPCClass	TSubclassOf<Actor>	Class-type configuration lookup defining the blueprint template to instance.
	SpawnDelay	Float	Time pacing value (seconds) executed before processing the next array element.
	SpawnPointTag	Name	Actor search identifier tag used to pair the entity with world transforms.

Critical Engineering Safe-Guard: The NPCClass field MUST explicitly use a class reference metadata type (TSubclassOf<Actor>) instead of a direct object reference. Object reference structures in static assets force heavy default memory serialization overhead and break the engine's dynamic SpawnActorFromClass subroutines, resulting in total runtime failure.

4. System Runtime Architecture & Actor Relations

The framework operates live using a highly decoupled triad pattern, preventing hard dependencies through asynchronous event registration:

- **BP_WaveManager (Actor):** The central engine authority. It interprets structural datasets, executes loop recursions, hosts tracker arrays, and broadcasts completion states.
- **BP_NPC_Base (Character/Actor):** The base combat actor. It completely lacks references to the manager; it simply exposes a native uncoupled execution hook (Event Dispatcher) signaling death.
- **BP_Portal (Actor):** The spatial blocker. It remains in a static, un-clickable state until receiving direct instructions to re-route interaction channels.

4.1 Internal State Management (BP_WaveManager)

Variable Name	Type Definition	Internal Functional Mechanics
MapWaveData	PDA_BiomeConfig (Object Reference)	Target runtime source asset containing the complete blueprint execution structural map.
CurrentWaveIndex	Integer	Tracked index pointing to the active wave segment inside the

Variable Name	Type Definition	Internal Functional Mechanics
		primary data layer.
CurrentNPCIndex	Integer	Tracked loop position marking the next entity object configuration scheduled for spawning.
ActiveNPCs	Array<BP_NPC_Base>	Live collection storing hard references to all spawned, surviving enemies in the scene.
WaveSpawnFinished	Boolean	State flag indicating if the current wave array loop has concluded structural instantiation.

5. Logic Flows & Core Blueprints Implementations

5.1 Guard Validation Pipeline (BeginPlay)

Upon level instantiation, the BP_WaveManager passes its primary asset through a strict validation check before allocating memory registers. This prevents null runtime exceptions if the actor is accidentally left inside non-combat levels like transition hubs.

```
[BeginPlay] ──> [ IsValid (MapWaveData) ] ──(Valid)──────> [ Call: SpawnNextNPC ]
                          |
                      (Invalid)
                          └──────> [ Log: "Encounter Silent Shutdown: No Data Assigned" ]
```

5.2 Timed Asynchronous Spawning (The Spawning Loop)

Entity instantiation relies on a timed, recursive indexing sequence. Instead of using complex standard For-Each blocks that trigger instantly within a single frame, the system schedules operations on a dedicated node timeline using sequence trackers.

Execution Graph Logic (SpawnNextNPC Function):

```
1. Check: Is [CurrentNPCIndex] < [Length of NPCsToSpawn Array]?
   ├── YES:
   │    ├── Get spatial transform from world actor matching [SpawnPointTag].
   │    ├── Execute [SpawnActorFromClass] using configured [NPCClass].
   │    ├── Push spawned object reference into the [ActiveNPCs] Array.
   │    ├── Execute [Bind Event to OnDied] linking the entity to our custom death listener.
   │    ├── Increment [CurrentNPCIndex] by 1.
   │    └── Pipe workflow into a [Delay] node using [SpawnDelay] float data.
   │         └── Upon Delay expiry ──> Re-route execution back to step 1 (Recursion).
   │
   └── NO:
```

```
└── Set [WaveSpawnFinished] = True. (Loop breaks naturally).
```

5.3 Reactive Lifecycle Tracking (Death & Array Removal)

To avoid polling overhead, the manager handles death events reactively. By using direct object references passed through the death event signature, the system remains immune to indexing errors caused by simultaneous multi-entity deaths.

```
[BP_NPC_Base Dies] ──> [ Broadcast: OnDied(Self) ] ──> [ Event Received by WaveManager ]
                                                                  |
                                                                  ▼
[ Evaluate: ActiveNPCs.Length == 0 ] <── [ Remove Item: (Self Reference) from ActiveNPCs ]
```

5.4 Gated Verification & Wave Advancement

When the ActiveNPCs array length reaches zero, the system conducts a dual-gated verification check. This prevents the progression portal from triggering prematurely while later enemies in the current wave are still waiting to spawn.

```
On ActiveNPCs.Length == 0:
  └── IF [WaveSpawnFinished] == True?
      ├── YES (All entities scheduled are dead):
      │    └── Evaluate: Is [CurrentWaveIndex + 1] < [Length of Waves Array]?
      │         ├── YES (More waves remain):
      │         │    ├── Increment [CurrentWaveIndex] by 1.
      │         │    ├── Reset [CurrentNPCIndex] to 0.
      │         │    ├── Set [WaveSpawnFinished] = False.
      │         │    └── Trigger: [Delay (Inter-wave Warmup)] ──> [SpawnNextNPC].
      │         │
      │         └── NO (Absolute Room Cleared):
      │              └── Trigger Event: [UnlockEncounterCompletion].
      │
      └── NO (Array is empty, but more enemies are still spawning):
           └── Do Nothing. (Spawning loop continues processing naturally).
```

5.5 Progression Portal Activation Pipeline

Upon receiving the UnlockEncounterCompletion signal, the BP_Portal object coordinates its activation sequence across multiple subsystems. This ensures rendering consistency and prevents gameplay bugs like missing particle displays on subsequent visits.

1. Set [IsPortalActive] = True
2. Call [Set Visibility] on Niagara/Cascade Component ──> True
3. Call [Activate (Reset = True)] on Particle Component
   (*Crucial step to force re-evaluation of non-looping emitter bursts)
4. Call [Set Collision Enabled] on static mesh/blocker bounds ──> Query and Physics

6. Production Use-Case Implementation Example

The following dataset demonstrates a two-wave combat room configuration inside the framework, highlighting how encounter difficulty curves are shaped entirely via raw data injection:

Wave Index	Target Class (NPCClass)	Spawn Offset (Delay)	World Anchor Tag	Design Intention / Pacing Notes
Wave 0 (Introductory)	BP_NPC_MeleeGrunt	1.5s	SpawnAnchor_Left	Staggered front flank deployment to establish player spatial awareness.
	BP_NPC_MeleeGrunt	1.5s	SpawnAnchor_Right	
	BP_NPC_MeleeGrunt	1.5s	SpawnAnchor_Center	
	BP_NPC_RangedMage	3.0s	SpawnAnchor_RearElevated	Delayed backline artillery placement to pressure defensive positions.
Wave 1 (Climax Overrun)	BP_NPC_ArmoredBrute	1.0s	SpawnAnchor_Center	Immediate high-threat priority target dropped directly into focus.
	BP_NPC_FastStalker	1.0s	SpawnAnchor_Left	Rapid flankers introduced simultaneously to force player repositioning.
	BP_NPC_FastStalker	2.0s	SpawnAnchor_Right	

7. Defensive Architecture & Stability Engineering

To ensure high stability across long player sessions, the framework incorporates several robust, defensive design patterns:

- **Simultaneous Death Slicing:** Traditional loop iterations using integer array indices frequently skip records or read out of bounds when multiple enemies are destroyed on the exact same tick. By abandoning indices and leveraging direct self-references through event parameters, the Remove Item node safely updates array lengths under any condition.
- **Null Pointer Suppression:** If an assembly scene contains an active orchestrator but lacks assigned data parameters, standard runtime routines throw heavy loop errors. The introduction of upstream validation gates ensures the system shuts down gracefully without interrupting gameplay.

- **Decoupled Communication:** The manager completely avoids expensive, performance-heavy nodes like *Get All Actors of Class* or heavy direct casting. Communications are handled exclusively via isolated listeners and tags, keeping memory overhead minimal.

8. Architectural Growth & Industrial Extensions

This system serves as a scalable foundation designed to support several advanced gameplay features over time:

1. **Dynamic Scalability Modifiers:** By exposing an array transformation layer before the initialization process, game systems can inject modifiers that dynamically adjust encounter size and threat metrics based on persistent session data like player clear speeds or current run depth.
2. **Global State Dispatchers:** Exposing decoupled global event hooks such as OnWaveStarted and OnWaveEnded allows developers to easily connect contextual UI elements, dynamic combat music cues, and environment lighting shifts without modifying the underlying spawning code.
3. **Weighted Spawning Groups:** Replacing strict tag fields with weighted array structures enables random selection from groups of target points, introducing unexpected combat angles and improving room replayability.