Politecnico di Torino

Ingegneria Informatica (Computer Engineering) LM-32(DM270)

A.a. 2025/2026

Graduation Session March 2026

Interactive High-Density Agent Simulation in Unreal Engine

An Evaluation of the Mass Entity Framework

Supervisors:
Francesco Strada
Corrado Raffaelli

Candidate:
Martina Plumari

Table of Contents

List of Figures V

1 Introduction 1
1.1 Goal . . . 1
1.2 Target Platform Definition . . . 2
1.3 Thesis Structure . . . 2

2 Entity Component System Architecture 4
2.1 Object-Oriented vs Data-Oriented Design . . . 4
2.2 The Three Pillars of ECS . . . 5
2.2.1 Entities . . . 6
2.2.2 Components . . . 6
2.2.3 Systems . . . 6
2.3 Industry Implementations and Case Studies . . . 7
2.3.1 Modern ECS Engines . . . 7
2.3.2 Hybrid Frameworks . . . 7
2.3.3 Practical Applications . . . 8
2.4 Trade-offs Analysis . . . 8

3 Unreal Engine Mass 10
3.1 MassEntity . . . 10
3.1.1 Entities . . . 11
3.1.2 Memory Organization: Archetypes and Chunks . . . 11
3.1.3 Mass Execution Environment . . . 13
3.1.4 Entity Processing and Interaction . . . 14
3.1.5 Fragments and Tags . . . 14
3.1.6 Traits . . . 17
3.1.7 Processors and Queries . . . 20
3.2 Bridging the ECS and OOP Architectures . . . 27
3.2.1 Data Synchronization Modes . . . 27
3.3 Additional Mass Plugins . . . 28

3.3.1 MassGameplay . . . 29
3.3.2 MassAI . . . 30
3.3.3 MassCrowd . . . 30

4 Large-Scale Entity Representation 32
4.1 Actor-based Representation Limitations . . . 33
4.1.1 The Mass Solution . . . 34
4.2 Mass Level Of Detail . . . 36
4.2.1 Mass LOD Overview . . . 36
4.2.2 Mass LOD Subsystem . . . 36
4.3 Mass Representation . . . 39
4.3.1 Vertex Animated ISM . . . 40
4.3.2 Niagara-Based Representation . . . 41

5 Decision Logic and State Management 43
5.1 State Tree . . . 43
5.1.1 Architecture . . . 44
5.1.2 Selection and Transition Flow . . . 46
5.1.3 State Tree Schemas . . . 49
5.1.4 Data Management . . . 50
5.2 Mass and State Tree . . . 51
5.2.1 Mass State Tree Schema . . . 51
5.2.2 Data Handling and Memory Locality . . . 53
5.2.3 Transition Handling and Logic Flow . . . 55

6 Navigation in the Mass Framework 56
6.1 Overview . . . 56
6.2 NavMesh Navigation . . . 57
6.2.1 The NavMesh Trait . . . 57
6.2.2 The StateTree Task . . . 58
6.2.3 The Path Following Processor . . . 58
6.3 ZoneGraph . . . 59
6.3.1 Zones . . . 59
6.3.2 Lanes . . . 60
6.3.3 Intersections . . . 60
6.3.4 Connections . . . 61
6.3.5 The ZoneGraph Trait . . . 61
6.3.6 The StateTree Task . . . 62
6.3.7 The Path Following Processor . . . 62
6.4 Additional Movement Traits and Processors . . . 63
6.4.1 Steering . . . 63

6.4.2 Smooth Orientation 63
6.4.3 Avoidance 64

7 Conclusions 65
7.1 Production Risks and Tooling Deficits 65
7.2 Scaling Bottlenecks: Navigation and Avoidance 66
7.3 The Paradigm Shift 66
7.4 Final Outlook 67

Bibliography 68

List of Figures

2.1 A comparison of memory access patterns between Object-Oriented and Data-Oriented Design. On the left (OOD), data is grouped by object: fetching a single component loads unrelated data into the cache (the green box), resulting in cache pollution. On the right (DOD/ECS), components of the same type are stored contiguously. This ensures that a single cache line fetch contains multiple relevant data points, maximizing cache hits and linear processing speed. . . 5
2.2 High-density zombie crowd in World War Z, developed using the Swarm Engine. 8
2.3 Pedestrians and traffic simulation in Cities: Skylines II. This game shows current Unity DOTS capabilities. 9

3.1 Memory organization within a Mass Archetype. The diagram shows how a single Entity ID acts as a shared index to access different data fragments (Transform and Velocity) stored in parallel, contiguous arrays. 11
3.2 Logical organization of Archetype-related data in chunks. Different colors of rows correspond to different fragments while the numbers refer to the Entity ID. 12
3.3 The image shows why it is convenient to have chunks sized to fit the cache rather than a single array structure for the whole Archetype. 13
3.4 Entity composition in Mass. This overview shows how individual Traits inject per-entity and shared fragments, along with filtering tags, into an entity's archetype during configuration. 17
3.5 Example of Mass Entity Config Asset defining an Entity via editor. It is possible to create one by clicking on the "Add" button in the content drawer, selecting Data Asset and then choosing Mass Entity Config Asset. 19
3.6 Example of Assorted Fragment trait use. 20
3.7 Entity Queries filter entities by archetype and pass data to Processors. 23

3.8 Example of Mass Agent Component added to an Actor. On the right side the section of the Details Panel where it is possible to define the entity configuration. 27
3.9 Plugin window showing the Mass Plugins listed. While MassEntity plugin is deprecated and now integrated in the engine codebase, MassAI, MassCrowd and MassGameplay are still marked as experimental and can be enabled from this panel. 29

4.1 CPU performance tanking when a grid of actors is instanced in an empty scene. The Game Thread took 44ms to process this frame, hence lowering the FPS. 33
4.2 ISM based representation managed by Mass. As we can see in the top-right corner, this type of representation significantly reduces the computational weight on the Game Thread we noticed in the case of the Actor representation. 35
4.3 Different entity representation based on the distance from the viewer. The red entities are Static Mesh Instances, the yellow ones are LowResSpawnedActors and the white ones are HighResSpawnedActors 40

5.1 An example of StateTree editor visualization. 44
5.2 When clicking on a State it is possible to add and configure Tasks in this section of the detail panel. 45
5.3 A simple evaluator defined in Blueprint that returns the current location of the player. 46
5.4 Example of transition definition. The next state in this case will be different based on the state outcome. 48

6.1 In green the active NavMesh created from the level geometry. . . . 57
6.2 An example of ZoneGraph lanes, with an intersection that merges them. 60

Chapter 1

Introduction

The demand for increasingly immersive and densely populated virtual worlds has pushed modern game engines to their computational limits. In traditional software architecture, simulating thousands of autonomous agents, each requiring physics, logic, rendering, and navigation updates, quickly saturates the CPU, resulting in severe performance bottlenecks. To bypass these limitations, the game development industry is undergoing a structural paradigm shift, moving away from familiar Object-Oriented Programming (OOP) in favor of Data-Oriented Design (DOD).

This thesis explores this architectural transition through the lens of Unreal Engine's Mass framework, a specialized Entity Component System (ECS) designed to handle massive-scale simulations.

1.1 Goal

The primary objective of this work is to dissect and evaluate the Mass framework as a viable solution for large-scale agent simulations in commercial game development. While the theoretical performance benefits of an ECS architecture are well-documented, applying these concepts within an engine historically rooted in object-oriented design presents unique software engineering challenges.

This thesis aims to demystify the Mass ecosystem. By providing a comprehensive technical analysis of how the framework operates under the hood, from its low-level memory allocation to high-level artificial intelligence and rendering, this work seeks to bridge the gap between Epic Games' experimental features and practical, production-ready implementation. Ultimately, it assesses the framework's viability, highlighting both its unprecedented performance capabilities and the steep learning curve required for a development team to adopt it.

1.2 Target Platform Definition

The target used for performance considerations is a laptop with the following hardware specifications:

- CPU: 12th Generation Intel Core i7-12700H
- RAM: 16GB DDR5-4800 MHz
- GPU: NVIDIA GeForce RTX 3050 Ti (4 GB dedicated memory)

1.3 Thesis Structure

To comprehensively analyze the implementation and impact of the Mass framework, this thesis progresses logically from theoretical foundations to practical, systemic execution.

The exploration begins by establishing the theoretical baseline of Entity Component Systems (ECS). The first chapter contrasts traditional Object-Oriented Design with Data-Oriented Design, detailing the three core pillars of the ECS pattern and analyzing the inherent trade-offs of adopting this methodology. Building on this foundation, the second chapter dives directly into Unreal Engine's specific ECS implementation. It examines the core architecture of Mass, illustrating how memory is managed through Archetypes and Chunks, how processors execute logic in parallel, and how the framework successfully bridges the gap between raw data entities and traditional Unreal Engine Actors.

Once the underlying simulation architecture is established, the thesis tackles the inevitable rendering bottlenecks associated with large-scale crowds. The third chapter explores how Mass decouples logical simulation from visual representation, utilizing Level of Detail (LOD) systems, Instanced Static Meshes (ISM), and Vertex Animation Textures to render thousands of characters without overwhelming the graphics pipeline.

With the entities efficiently simulated and rendered, the focus then shifts to their behavior. The fourth chapter delves into artificial intelligence, providing a technical breakdown of how Unreal Engine's hierarchical State Tree system integrates with Mass. This integration allows for complex, node-based decision-making that scales efficiently through signal-driven, asynchronous execution. Following the decision-making process, the fifth chapter explores how these intelligent entities physically traverse their environment. It details the data-oriented wrappers for standard open-world pathfinding via NavMesh and strict spline-based routing through ZoneGraph, concluding with an analysis of low-level steering and crowd avoidance processors.

Finally, the Conclusion synthesizes these findings to evaluate the framework's current state. It grounds the technical achievements in reality by discussing the

practical production risks, the current deficit in editor tooling, and the remaining bottlenecks in navigation. Ultimately, it reflects on the significant shift in development methodology required for a team to successfully leverage Mass in a commercial production pipeline.

Chapter 2

Entity Component System Architecture

The Entity Component System (ECS) is a software architectural pattern specifically engineered to facilitate high-performance simulations. Unlike traditional game development frameworks that rely on deep object hierarchies, ECS is built upon the Data-Oriented Design (DOD) paradigm.

At its core, ECS shifts the focus from what an object is to what data it possesses. By decoupling data from behavior, it enables the construction of complex simulations using granular, reusable modules rather than monolithic, rigid classes.

This chapter first analyzes the Entity-Component-System architectural pattern and how it can offer performance advantages over the traditional object-oriented paradigm. It then examines the trade-offs of this type of design compared to the object-oriented one.

2.1 Object-Oriented vs Data-Oriented Design

To understand why ECS has become the industry standard for mass-scaled simulations, we must compare it to the traditional Object-Oriented Design (OOD).

In OOD, functionality is typically extended through inheritance, making the resulting objects inextricably bound to their ancestors through a hard-coded relationship. As a project grows, this can lead to deep and rigid class hierarchies. For instance, when a Soldier class inherits from a Human class, it does not merely get access to human traits, but becomes physically coupled to the entire memory footprint and logic chain of the parent.

On the other hand, DOD favors composition. In this case, rather than defining what an entity is (a Soldier), we define what it has (Transform, Health, Ammo). This approach allows us to avoid being forced to choose between two unrelated

classes and enables architectural flexibility, since the entity is simply a dynamic collection of data components assigned at runtime.

The primary driver for DOD is data locality. Modern CPU speeds vastly outpace memory access times; consequently, performance is often throttled by memory latency, the delay incurred when fetching data from RAM. In Object-Oriented Design, objects are frequently scattered throughout memory. Accessing them requires pointer-chasing through class hierarchies, leading to frequent cache misses and significant performance penalties in contemporary CPU architectures.

Conversely, in Data-Oriented Design, components of an identical type are stored contiguously in memory. When a system processes these components, the CPU can accurately predict the next data address and pre-load it into the cache. This maximizes cache coherency, facilitating rapid, linear iteration over thousands of entities (Fig. 2.1).

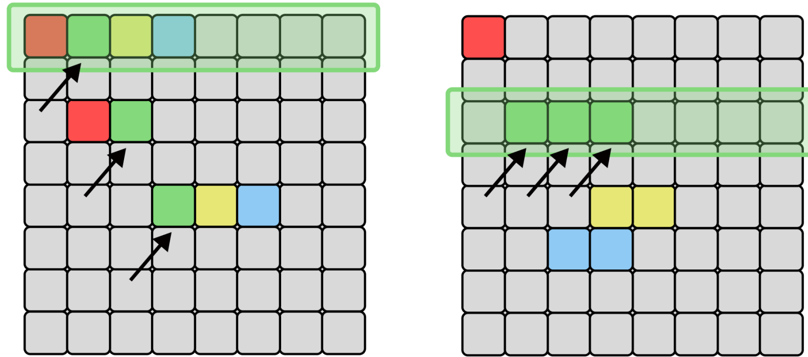

Figure 2.1: A comparison of memory access patterns between Object-Oriented and Data-Oriented Design. On the left (OOD), data is grouped by object: fetching a single component loads unrelated data into the cache (the green box), resulting in cache pollution. On the right (DOD/ECS), components of the same type are stored contiguously. This ensures that a single cache line fetch contains multiple relevant data points, maximizing cache hits and linear processing speed.

2.2 The Three Pillars of ECS

As the name suggests, the ECS pattern abandons the monolithic object approach and instead distributes the simulation workload across three distinct functional

layers: Entities, Components, and Systems.

2.2.1 Entities

In an ECS architecture, an Entity is fundamentally stripped of all intrinsic data and behavior. It is not an object or a class, but rather a lightweight, unique identifier, often implemented simply as an integer. Its sole purpose is to serve as a handle or a database key that conceptually binds a specific set of components together. By reducing the entity to a mere index, the architecture ensures that the creation, destruction, and tracking of simulated actors incur virtually zero memory overhead.

2.2.2 Components

Components are the granular data containers of the simulation. Unlike object-oriented properties that are often entangled with getter and setter methods, components strictly encapsulate state and contain absolutely no logic. They are designed as plain old data (POD) structures. Because they are completely devoid of functional overhead, components of the same type can be packed tightly into dense, contiguous memory arrays. This strict data alignment is the foundation of the architecture's memory throughput optimization, ensuring that when the CPU requests one component, the adjacent components are simultaneously loaded into the high-speed cache. How these contiguous arrays are managed under the hood, such as through Sparse Sets or Archetype chunks, determines the specific performance profile of the ECS, which will be explored in the context of Unreal Engine in the following chapter.

2.2.3 Systems

The System represents the active logic layer of the architecture. Systems are independent, stateless execution units that operate over filtered collections of components. Rather than calling functions on individual objects, a system utilizes declarative queries to request all entities possessing a specific combination of components. For example, a Movement System could request all entities with both a Transform and a Velocity component. To preserve this strict isolation, systems do not communicate via direct method calls. Instead, inter-system communication is achieved indirectly: a system might append a specific "Tag" component to an entity to flag it for processing by another system, or utilize deferred command buffers and event queues to safely broadcast state changes. Because systems operate on dense data arrays and explicitly declare their read and write dependencies, they are inherently suited for multi-threaded execution. This allows the engine's task

scheduler to safely distribute workloads across multiple CPU cores with minimal synchronization overhead or risk of race conditions.

2.3 Industry Implementations and Case Studies

The transition toward Data-Oriented Design is reflected in the evolution of modern game engines. These implementations generally fall into two categories: pure ECS architectures and hybrid frameworks that integrate DOD principles into existing object-oriented environments.

2.3.1 Modern ECS Engines

A prominent example of a pure ECS architecture is Bevy, an open-source engine written in Rust. In Bevy, the ECS is not an auxiliary feature but the core foundation. Every element of the engine, from the renderer to the UI, is implemented as a system operating on components. This provides a high degree of modularity and ensures that data locality is maintained across the entire software stack.

In the proprietary space, Saber Interactive's Swarm Engine represents a specialized application of ECS. Developed to power the massive NPC counts in *World War Z* (Fig. 2.2) and later utilized for the Tyranid hordes in *Warhammer 40,000: Space Marine 2*, the engine utilizes ECS to manage thousands of autonomous agents simultaneously. By utilizing a custom architecture that prioritizes high-throughput data processing over traditional object-oriented hierarchies, the engine facilitates complex collective behaviors.

2.3.2 Hybrid Frameworks

Many established engines have opted for a hybrid approach to maintain backward compatibility while offering DOD performance. Unity's Data-Oriented Technology Stack (DOTS) is a prime example, providing a specialized Entities package that works alongside its traditional C# scripting API. This allows developers to isolate performance-critical logic, such as large-scale physics or crowd simulation, within an ECS workflow while using traditional tools for higher-level game logic.

Similarly, Unreal Engine's Mass framework provides a data-oriented calculation layer designed for massive-scale simulations. While Unreal remains fundamentally rooted in the `UObject` ecosystem, Mass allows for the processing of Entities that exist outside the heavy `AActor` lifecycle. This architecture enables the engine to simulate urban environments with thousands of pedestrians or complex traffic systems, bridging the gap between high-fidelity object-oriented features and high-throughput data processing.

Figure 2.2: High-density zombie crowd in World War Z, developed using the Swarm Engine.

2.3.3 Practical Applications

The efficacy of these architectures is best observed in titles where the simulation complexity is the primary gameplay driver. Beyond *World War Z*, games such as *Cities: Skylines II* (Fig. 2.3) leverage ECS to track the individual economic and navigational states of tens of thousands of citizens in real-time.[6] Even the Bedrock edition of *Minecraft* transitioned to an internal ECS to manage its vast array of world entities, demonstrating that the paradigm shift is not merely for niche simulations but is essential for the scalability of modern, cross-platform interactive media.

2.4 Trade-offs Analysis

While the Entity Component System provides the theoretical foundation and raw performance necessary for frameworks like Unreal Engine's Mass, adopting this architecture introduces specific structural trade-offs that must be carefully managed during development.

One of the primary challenges is an increase in initial architectural complexity. Unlike the intuitive, object-based approach of traditional OOP, where a class naturally groups its data and related behaviors, ECS requires a rigorous, upfront design

Figure 2.3: Pedestrians and traffic simulation in Cities: Skylines II. This game shows current Unity DOTS capabilities.

of component interfaces and system boundaries. Developers must meticulously plan how data will be accessed and mutated before writing the logic.

Furthermore, the radical decoupling of data from logic can introduce its own form of decoupling overhead. If component granularity is not managed carefully, developers may experience what is called component bloat. This occurs when state is fragmented into overly trivial data structures, resulting in repetitive memory fetching and complicating the architecture rather than streamlining it.

Finally, the most significant barrier to adoption is the required mentality shift. Transitioning from an object-centric mindset to a purely Data-Oriented paradigm demands a fundamental restructuring of how developers conceptualize program flow, state management, and memory access. This cognitive shift takes considerable time and training. The steep learning curve associated with DOD may not be strategically feasible for all production timelines or team structures, making ECS a powerful but demanding architectural choice.

Chapter 3

Unreal Engine Mass

While the Entity Component System is a theoretical pattern, Mass is Unreal Engine's specific framework for data-oriented calculations and high-performance simulations. It does not replace the traditional Actor-based system; instead, it provides a parallel infrastructure capable of processing a large number of entities with minimal CPU overhead.

In this chapter we are going to explore the declination of the ECS pattern in the Mass System via the core plugin MassEntity and the features available through the addition of standalone plugins offered by Epic Games.

3.1 MassEntity

At the heart of Mass system lies MassEntity, the core module responsible for the framework's low-level heavy lifting. It manages the execution pipeline, entity lifecycle and the specialized memory layouts required for high-performance computing. Initially introduced as a plugin, MassEntity was integrated into the core engine as of UE5.5, with the standalone plugin version now deprecated.

Within MassEntity, we find specialized implementation of the three pillars of the ECS pattern:

- **Entities:** Unique identifiers that represent individual simulated objects.
- **Fragments:** Logic-less data structures that serve the role of Components, storing the entity's data.
- **Processors:** Highly optimized logic units that act as Systems, operating on specific sets of Fragments.

In the following sections, we will present an overview of the software architecture

of Mass Entity, focusing on data structures, memory organization and logic execution to illustrate how these core ECS concepts are implemented within framework. Following this high-level overview, we will dive deeper in the analysis of each element, providing practical code examples to demonstrate their implementation and interaction.

3.1.1 Entities

In the Mass framework, an Entity is a lightweight, unique identifier rather than a data-carrying object. It serves as a hook to associate various fragments within the Mass database.

When an entity is created, the `MassEntitySubsystem` assigns it an index and a serial number used to prevent dangling references if the entity is destroyed. This abstraction allows the engine to treat entities as transient, high-performance entries in a table.

3.1.2 Memory Organization: Archetypes and Chunks

In Mass Framework, an Archetype (Fig. 3.1) is the logical grouping of entities sharing an identical composition of Fragments and Tags (`FMassArchetypeData`). As entities dynamically gain or lose fragments at runtime, they are migrated between Archetypes. This structural consistency allows the engine to bypass the pointer-chasing common in Object-Oriented Programming (OOP) in favor of linear memory access.

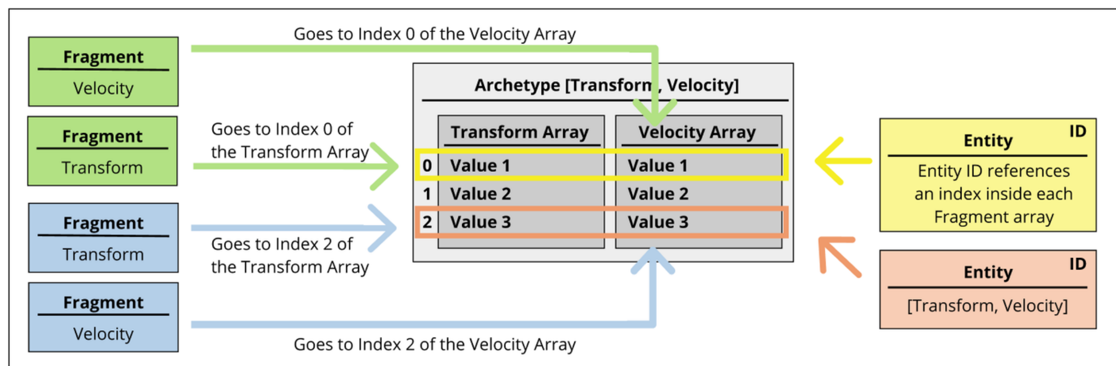

Figure 3.1: Memory organization within a Mass Archetype. The diagram shows how a single Entity ID acts as a shared index to access different data fragments (Transform and Velocity) stored in parallel, contiguous arrays.

To optimize for CPU cache coherency, Mass organizes Archetype data into Chunks (`FMassArchetypeChunk`). A Chunk is a fixed-size memory block that

serves as the fundamental unit of both memory allocation and parallel execution. Each Archetype holds an array of Chunks with fragment data and each chunk contains fragments referring to a subset of entities belonging to the archetype.

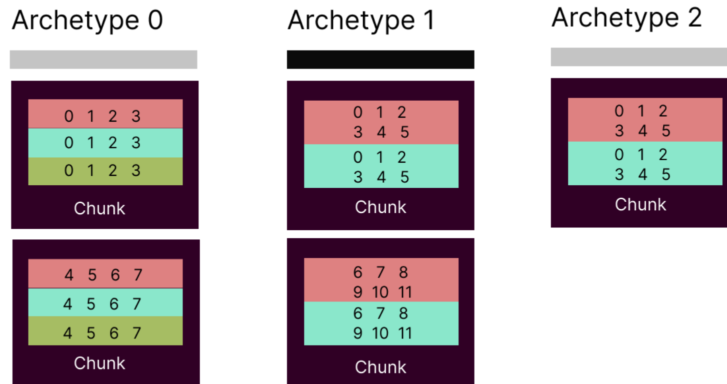

Figure 3.2: Logical organization of Archetype-related data in chunks. Different colors of rows correspond to different fragments while the numbers refer to the Entity ID.

To facilitate rapid filtering and query matching, the archetype structure incorporates a bitset representation of its composition. Each bit in this mask corresponds to a specific fragment or tag type. If a bit is set, the archetype contains that component. This bitmasking approach allows the `MassEntitySubsystem` to perform constant-time validation when determines if an archetype satisfies the requirements of a processor's `FMassEntityQuery`.

The default size of the memory block is defined in UMassEntitySettings and it is equal to 128 KB. This size has been established based on current generation caches and could change in future versions of the engine.

Rather than storing all fragments of a single type for an entire Archetype in one massive array, Mass partitions the Archetypes into these 128 KB blocks. Within each chunk, data is organized as struct of arrays, assigning a block of memory to each type of Fragment and Tag instances.

By sizing chunks to fit within modern L2 or L3 caches, Mass ensures that when a processor requests multiple fragments for an entity, the entire row of data is likely already in the cache.

Furthermore, Chunks facilitate safe multi-threading. the MassEntitySubsystem can distribute individual chunks across different CPU cores via the Task Graph, ensuring that no two threads attempt to write to the same memory block simultaneously.

Because the ChunkMemorySize is a fixed constant, the entity density is inversely

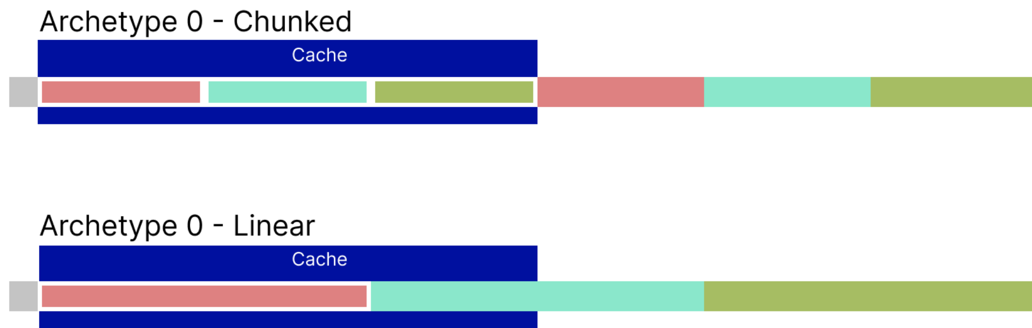

Figure 3.3: The image shows why it is convenient to have chunks sized to fit the cache rather than a single array structure for the whole Archetype.

proportional to the total size of the fragments. Having slim entities means that we can fit more entities in the chunk, improving cache efficiency by reducing the management overhead. When designing fragments, developers should aim for granularity. Large monolithic fragments reduce the number of entities per chunk, potentially dropping the density to a point where the overhead of chunk-level iteration negates the performance gains of the ECS architecture.

3.1.3 Mass Execution Environment

To coordinate the complex relationship between archetypes, chunks, and entities, Mass utilizes a centralized management layer consisting of the `UMassEntitySubsystem` and the `FMassEntityManager`.

The `FMassEntityManager` acts as the low-level data authority. It maintains the global registry of all active archetypes and provides the logic for structural changes, such as adding or removing fragments and archetype migration. By centralizing these operations, the manager ensures that memory remains contiguous and that entity handles (`FMassEntityHandle`) remain valid throughout their lifecycle.

Surrounding the manager is the `UMassEntitySubsystem`, a `UWorldSubsystem`[1]. that provides a stable API for the rest of the engine. It facilitates the baking of `FMassEntityTemplates` and manages the lifecycle of the simulation relative to the `UWorld`. Together, these components serve as the backbone of the framework, translating high-level commands into the low-level memory operations that define the Mass Framework's performance characteristics.

[1]A World Subsystem in Unreal Engine is a singleton-like class that is automatically managed by the engine and tied directly to the lifecycle of a `UWorld` instance.

3.1.4 Entity Processing and Interaction

To maintain the integrity of the linear memory chunks during multi-threaded execution, Mass employs a deferred-mutation pattern via the Mass Command Buffer. Structural changes, such as spawning entities, destroying them, or altering their archetype through the addition or removal of fragments, are not executed immediately by processors. Instead, these requests are queued as Commands. At designated synchronization points in the frame, the `FMassEntityManager` flushes these buffers, performing the memory-intensive migrations in a batch. This prevents race conditions and ensures that pointers to fragment data remains valid throughout the duration of a processing task.

While processors typically operate on entire batches of entities, individual entity interaction is facilitated through the Entity View (`FMassEntityView`). Because fragments are stored in raw memory buffers, they cannot be accessed through standard pointer dereferencing from an entity handle. The Entity View acts as a transient window into a specific entity's data, resolving the `FMassEntityHandle` into a structured set of pointers for its constituent fragments. This provides a type-safe interface for logic that must operate outside of the standard batch-processing pipeline without sacrificing the performance benefits of the underlying ECS architecture.

3.1.5 Fragments and Tags

Fragments constitute the atomic data units of the Mass Framework. Unlike traditional class properties, fragments are implemented as lightweight `USTRUCTs` inheriting from `FMassFragment` and contain no logic, serving strictly as data containers. By decoupling data from behavior, Mass allows for highly flexible entity composition and optimal memory alignment.

While generic fragments are unique to each entity, the framework provides specialized fragment types to optimize memory footprint and group-level data management.

Entity Fragments

The standard fragment type stored in rows within a chunk. Each entity has its own instance of this data (e.g. `FTransformFragment`), allowing for unique per-entity state.

```
1
2 USTRUCT()
3 struct FSimpleMovementFragment : public FMassFragment
4 {
5   GENERATED_BODY()
```

```
6     FVector InitialTarget;
7     float Velocity;
8  };
```

The design of individual fragments should strictly adhere to the principle of locality, where data is grouped according to its specific access pattern or functional aspect. For instance, a `FSimpleMovementFragment`, such as the one in the code example above, should ideally only contain variables relevant to spatial translation, such as `InitialTarget` and `Velocity`.

Integrating unrelated variables into a single monolithic fragment introduces several architectural inefficiencies:

1. **Cache Line Pollution:** CPUs fetch data from memory in fixed-size lines. If a movement processor iterates through a fragment that contains many variables to access velocity data, the CPU is forced to load the adjacent, unrelated variables into the cache. This results in polluted cache lines where a significant portion of the high-speed memory is occupied by data that will not be necessary for execution, effectively wasting memory bandwidth.

2. **Decreased Entity Density:** Because `ChunkMemorySize` is a fixed constant, the number of entities that can coexist within a single chunk is inversely proportional to the total size of its fragments. Increasing the memory footprint of a fragment directly reduces entity density, necessitating more frequent chunk-switching and increasing the management overhead per entity.

3. **Parallelization Constraints:** The Mass Framework achieves high performance through concurrent execution, where different processors can operate on different fragments of the same entity simultaneously. If disparate variables, such as `Health` and `Velocity`, are defined in a single fragment, a processor modifying health will effectively lock the entire fragment. This creates a data dependency that prevents a movement processor from accessing the velocity data in parallel, forcing sequential execution or requiring complex synchronization overhead that negates the benefits of the ECS architecture.

4. **Structural Change Overhead:** Entities in Mass are dynamic, and adding or removing fragments triggers a migration to a new Archetype. This process involves a memory copy of the entity's existing fragments. Monolithic fragments significantly increase the volume of data that must be shuffled during these structural changes, even if the modification only involves a single, small variable.

Chunk Fragments

These fragments (`FMassChunkFragment`) are stored once per chunk rather than once per entity. They are ideal for data that is identical for every entity within a specific memory block, such as Level-of-Detail (LOD) information or spatial bounds. By updating a single chunk fragment, a processor can effectively update the state of hundreds of entities simultaneously.

Shared Fragments

Shared fragments (`FMassSharedFragment`) represent data that is common across multiple chunks or even multiple archetypes. They are stored in a central registry and referenced via a handle, making them highly efficient for constant data, such as a shared mesh reference or a global movement configuration, that rarely changes. Here is an example of shared fragment that contains the Player's position:

```
1
2 USTRUCT(BlueprintType)
3 struct FCurrentPlayerPositionFragment : public
      FMassSharedFragment
4 {
5      GENERATED_BODY()
6
7      UPROPERTY(EditAnywhere, BlueprintReadWrite)
8      FVector PlayerPosition;
9 };
```

Tags

Implemented via `FMassTag`, tags are zero-sized fragments that carry no data, having no member properties. They exist solely as bits within the archetype's `FMassTagBitSet`. They are used for high-speed categorization and filtering, allowing processors to include or exclude entities based on boolean-like markers without increasing memory consumption.

To define a custom tag, a developer simply inherits from `FMassTag`, ensuring the struct remains empty:

```
1
2 USTRUCT()
3 struct FDeadTag : public FMassTag
4 {
5      GENERATED_BODY()
6 };
```

3.1.6 Traits

While fragments and tags provide the low-level data structures for the ECS, traits serve as the high-level configuration layer that bridges the gap between the Unreal Editor and the ECS backend. A trait acts as a high-level recipe that specifies which fragments, tags and shared data should be attached to an entity. This abstraction allows designers to compose complex agents behavior, such as crowd movement, avoidance or visualization, without directly interacting with the underlying implementation.

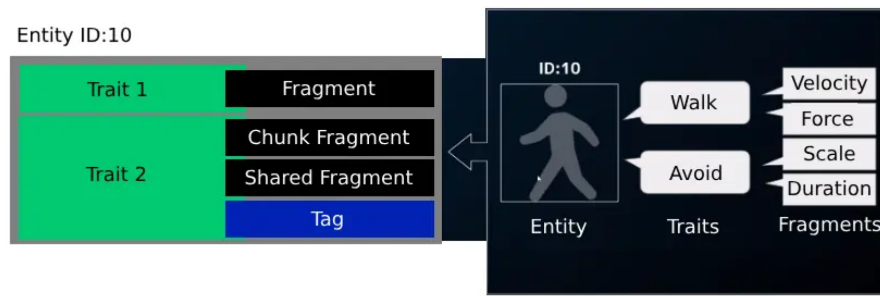

Figure 3.4: Entity composition in Mass. This overview shows how individual Traits inject per-entity and shared fragments, along with filtering tags, into an entity's archetype during configuration.

A trait is implemented in C++ by inheriting from `UMassEntityTrait` and overriding the `BuildTemplate` function:

```
1
2  UCLASS(meta = (DisplayName = "NPC Horde Movement"))
3  class NPCHORDE_API UNPCMovementTrait : public
       UMassEntityTraitBase
4  {
5      GENERATED_BODY()
6
7  protected:
8      virtual void BuildTemplate(FMassEntityTemplateBuildContext&
       BuildContext, const UWorld& World) const override
9      {
10         FMassEntityManager& EntityManager =
11                 UE::Mass::Utils::GetEntityManagerChecked(World)
       ;
```

```
12
13          // Ensuring another trait provides this fragment
14          BuildContext.RequireFragment<FTransformFragment>();
15
16          // Adding a Fragment
17          BuildContext.AddFragment<FSimpleMovementFragment>();
18
19          // Adding a Tag
20          BuildContext.AddTag<FMovingTag>();
21
22          // Creating and adding a shared fragment
23          const FConstSharedStruct RandomMovementFragment =
        EntityManager.GetOrCreateConstSharedFragment(RandomMovement)
        ;
24          BuildContext.AddConstSharedFragment(
        RandomMovementFragment);
25      }
26
27      UPROPERTY(Category = "Movement", EditAnywhere, meta = (
        EditInline))
28      FMassMovementParameters RandomMovement;
29  };
```

When the MassEntitySubsystem processes a trait, it executes the BuildTemplate function. During this phase, the trait contributes with its components to the entity template adding necessary fragments and tags to the archetype definition.

Following this construction phase, traits can override ValidateTemplate to provide custom verification logic. ValidateTemplate is a function invoked for all traits once the template composition is complete. It allows native traits to audit the final BuildContext, log configuration errors, or perform last-minute adjustments to ensure structural integrity. By separating construction from validation, the framework ensures that any missing dependencies or invalid parameters are caught in the Editor before the simulation begins, rather than causing runtime failures in the high-performance memory chunks. Here follows an example of template validation function from the MassLODTrait.h file:

```
1       bool UMassLODCollectorTrait::ValidateTemplate(const
        FMassEntityTemplateBuildContext& BuildContext, const UWorld&
         World, FAdditionalTraitRequirements& OutTraitRequirements)
        const
2   {
3     // if bTestCollectorProcessor is set to true, we require
        MassLODCollectorProcessor to be enabled.
```

```
4     if (bTestCollectorProcessor && !ValidateRequiredProcessor(
       this, UMassLODCollectorProcessor::StaticClass()))
5     {
6       return false;
7     }
8
9     return Super::ValidateTemplate(BuildContext, World,
       OutTraitRequirements);
10  }
```

Traits are primarily utilized during the construction of an `FMassEntityTemplate`, which is authored via a `UMassEntityConfigAsset`. This specialized Data Asset acts as a high-level container where designers can aggregate various traits to define an entity's composition and behavior. They often contain parameters that can be edited in the editor tab which are then used to populate the initial values of the fragments.

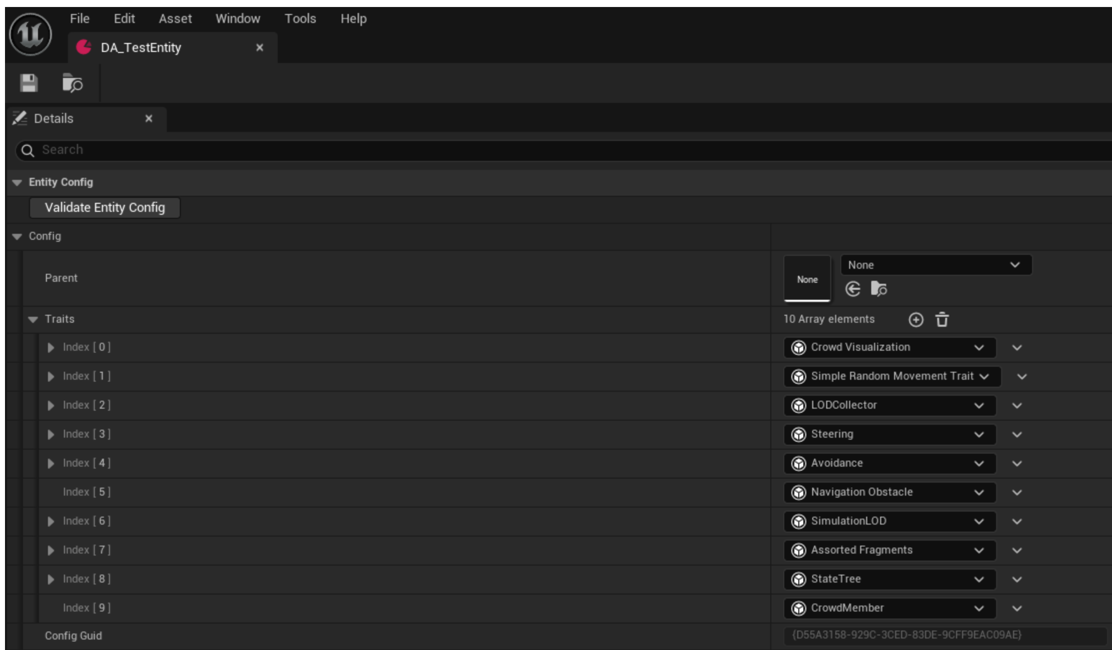

Figure 3.5: Example of Mass Entity Config Asset defining an Entity via editor. It is possible to create one by clicking on the "Add" button in the content drawer, selecting Data Asset and then choosing Mass Entity Config Asset.

While custom C++ traits offer the most control, the Mass Framework provides versatile built-in traits to streamline the development workflow. A notable example

is the Assorted Fragments Trait (`UMassAssortedFragmentsTrait`). This trait utilizes an array of `FInstancedStructs`, allowing designers to add fragments directly within the Unreal Editor.

Figure 3.6: Example of Assorted Fragment trait use.

Unlike standard Actor Components, traits do not exist at runtime once the entity has been spawned but are Editor-time construction tools. Once the simulation begins, the trait's responsibility ends, and the entity exists purely as a collection of fragments and tags. This ensures that the high-level flexibility of the Unreal Editor does not compromise the efficiency of the simulation.

3.1.7 Processors and Queries

The System element of the ECS pattern is represented in Unreal Engine by Processors.

Processors are stateless classes (inheriting from `UMassProcessor`) that house the simulation logic. They are designed to be data-agnostic, meaning they do not operate on individual entities through isolated function calls, but rather on contiguous fragments provided by the Entity Manager. By utilizing one or more user-defined Queries, processors filter the global entity population into optimized batches, allowing for high-throughput execution across matching memory chunks.

By default, any class deriving from `UMassProcessor` is automatically registered with the Mass Framework. To ensure synchronization with the rest of the Unreal Engine ecosystem (such as Physics, Animation, and Networking), the Mass Framework organizes processors into Processing Phases (Table 3.1). Each phase is mapped to a specific `ETickingGroup`, which dictates when the Mass logic will execute relative to the rest of the engine's frame logic.

Processors are automatically added to the `PrePhysics` phase, but developers can redistribute them to optimize parallelization or satisfy data dependencies.

In their constructor, processors define the execution topology by specifying their processing phase and establishing relative execution rules (e.g., `ExecuteBefore` or `ExecuteAfter`). Furthermore, they utilize **Execution Flags** to restrict logic to specific environment types, such as dedicated servers, local clients, or standalone instances.

Table 3.1: Unreal Engine 5 Mass Processing Phases and Tick Groups

Processing Phase	Unreal Tick Group	Description
PrePhysics	`TG_PrePhysics`	The default phase. Used for logic that sets up movement or state before the physics engine runs.
StartPhysics	`TG_StartPhysics`	A specialized phase that marks the beginning of the physics simulation.
DuringPhysics	`TG_DuringPhysics`	Logic that runs concurrently with the physics simulation, ideal for calculations that don't depend on current-frame physics results.
EndPhysics	`TG_EndPhysics`	A phase used to finalize physics-related calculations.
PostPhysics	`TG_PostPhysics`	Used for logic that depends on physics results, such as updating transforms based on collision or rigid body movement.
FrameEnd	`TG_LastDemotable`	A catch-all phase for cleanup or low-priority logic that can be deferred to the end of the frame.

```
1   UMyProcessor::UMyProcessor() :
2       MyQuery(*this) // Link the query to this processor
3   {
4       // Enables automatic discovery and registration by the
    MassEntitySubsystem
5       bAutoRegisterWithProcessingPhases = true;
6       // Assigns execution to a specific engine tick group
7       ProcessingPhase = EMassProcessingPhase::PrePhysics;
8       // Organizes the processor within a specific functional
    simulation block
9       ExecutionOrder.ExecuteInGroup = UE::Mass::
    ProcessorGroupNames::Movement;
10      // Establishes an execution dependency on another
    processor
11      ExecutionOrder.ExecuteAfter.Add(TEXT("
    MyMovementProcessor"));
12      // Limits execution to specific network roles (Client
    or Standalone)
```

```
13          ExecutionFlags = (int32)(EProcessorExecutionFlags::
        Client | EProcessorExecutionFlags::Standalone);
14          // Disables multithreading to allow safe access to Game
         Thread-only APIs
15          bRequiresGameThreadExecution = true;
16      }
```

While the Mass Framework executes processors in parallel across multiple worker threads by default, they can be constrained to a single-threaded execution on the Game Thread by setting `bRequiresGameThreadExecution` to `true`. This configuration is essential for tasks that must interact with non-thread-safe Unreal Engine systems, such as spawning Actors or modifying UI components, which require a stable execution environment on the main thread.

Users can aggregate processors into logical blocks called Processor Groups. Groups allow for a modular organization of the simulation logic. For example, a Movement group might contain separate processors for path following, steering, and avoidance. By assigning a processor to a group via `ExecuteInGroup`, developers can manage high-level simulation goals rather than individual class dependencies. The exact sequence of logic across these groups is enforced through the `ExecuteBefore` and `ExecuteAfter` constraints. These rules allow the framework to build a dependency graph during initialization. A Visualization Processor should always be set to `ExecuteAfter` the movement processor, to ensure that entities are rendered at their updated location.

Even if it may appear that a processor iterates over every entity simultaneously, the execution is actually performed in batches based on the memory organization of Chunks and Archetypes we previously analyzed. The standard execution flow follows these steps:

1. **Configuration:** in the `ConfigureQueries` function the processor defines its data requirements using one or more query.

2. **Filtering:** the query matches the requirements against the existing archetypes. Entire chunks are skipped if they do not meet the criteria.

3. **Iteration:** the processor calls `ForEachEntityChunk`. For every matching chunk, a lambda function is executed.

4. **Contextual Access:** within the lambda, the processor uses the Mass Execution Context to access fragment views (arrays of data) and perform calculations.

Mass Entity Queries

The `FMassEntityQuery` is the mechanism used to filter and fetch data based on the presence or absence of fragments and tags. Queries are highly optimized to perform bitwise comparisons against archetype bitsets, ensuring that the processor only touches relevant memory.

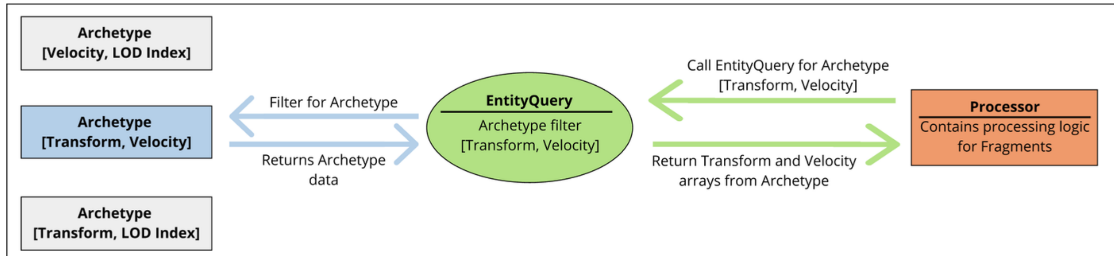

Figure 3.7: Entity Queries filter entities by archetype and pass data to Processors.

To maximize parallelization and cache efficiency, queries define precisely how they intend to interact with data. These rules are categorized into Access and Presence requirements. Access requirements define the lock type on the data:

- **ReadOnly:** the processor will only read the data, allowing other read-only processors to access it simultaneously.
- **ReadWrite:** the processor intends to modify the data, requiring exclusive access to prevent race conditions.

Presence requirements define the filtering rules:

- **All:** it mandates that an entity possess every specified fragment and tag in the query. It acts as a strict logical `AND` filter. This is the default presence requirement.
- **Any:** it mandates that an entity posses at least one of the specified fragment and tags. It acts as a logical `OR` filter.
- **None:** it excludes archetypes that contain these components. This is ideal for state changes, for example a Movement processor that needs to ignore entities with a `DeadTag`.
- **Optional:** it accesses the data if it exists, but does not filter out the entity if it is missing.

To apply these filters and access rules, processors must override the `ConfigureQueries` function. This is where the `FMassEntityQuery` is structured before the execution phase begins.

```
1  void UCheckPlayerProximityProcessor::ConfigureQueries(const
      TSharedRef<FMassEntityManager>& EntityManager)
2  {
3    EntityQuery.AddRequirement<FTransformFragment>(
      EMassFragmentAccess::ReadWrite);
4    EntityQuery.AddRequirement<FMassMoveTargetFragment>(
      EMassFragmentAccess::ReadWrite);
5    EntityQuery.AddRequirement<FNPCEntityState>(
      EMassFragmentAccess::ReadWrite);
6    EntityQuery.AddRequirement<FMassRepresentationLODFragment>(
      EMassFragmentAccess::ReadOnly);
7    EntityQuery.AddSharedRequirement<
      FCurrentPlayerPositionFragment>(EMassFragmentAccess::
      ReadWrite);
8    EntityQuery.AddTagRequirement<FDeadTag>(EMassFragmentPresence
      ::None);
9
10     //EntityQuery.RegisterWithProcessor(*this);
11 }
```

Queries must be registered with the processor to be tracked by the execution pipeline. While the most common pattern is to initialize the `FMassEntityQuery` in the processor's constructor, passing `*this` as the owner, it is also possible to register it explicitly within the `ConfigureQueries` function. The syntax to use for the second case is the one commented in the last line of the previous code example.

To process the filtered entities, the `Execute` function calls the query's `ForEachEntityChunk` method. This function takes the execution context and a lambda expression, which defines the logic to be applied to each chunk matching the query's requirements.

```
1
2  void MyProcessor::Execute(FMassEntityManager& EntityManager,
      FMassExecutionContext& Context)
3  {
4    EntityQuery.ForEachEntityChunk(Context,
5                    [this](FMassExecutionContext& Context)
6      {
7              // Obtain view for the fragments
8        TArrayView<FTransformFragment> TransformList = Context.
      GetMutableFragmentView<FTransformFragment>();
9        TArrayView<FSimpleMovementFragment> SimpleMovementList =
      Context.GetMutableFragmentView<FSimpleMovementFragment>();
10
11             // Obtain the shared parameters from the shared
      fragment
```

```
12          const FSimpleRandomMovementParameters MovementParams =
        Context.GetConstSharedFragment<
        FSimpleRandomMovementParameters>();
13
14          for (int32 EntityIndex = 0; EntityIndex < Context.
        GetNumEntities(); EntityIndex++)
15          {
16              FTransform& Transform = TransformList[EntityIndex].
        GetMutableTransform();
17              FVector& MoveTarget = SimpleMovementList[EntityIndex].
        InitialTarget;
18
19              // ...Computation using fragment data
20          }
21      });
22  }
```

Table 3.2: Mass Execution Context Access Functions

Requirement	**Function**	**Description**
ReadOnly Fragment	`GetFragmentView`	Returns a `TConstArrayView` containing read-only fragment data.
ReadWrite Fragment	`GetMutableFragmentView`	Returns a `TArrayView` for modifying fragment data.
ReadOnly Shared	`GetConstSharedFragment`	Returns a constant reference to a shared fragment.
ReadWrite Shared	`GetMutableSharedFragment`	Returns a mutable reference to a shared fragment.
ReadOnly Subsystem	`GetSubsystemChecked`	Returns a constant reference to a world subsystem.
ReadWrite Subsystem	`GetMutableSubsystemChecked`	Returns a mutable reference to a world subsystem.

Inside the lambda, the `FMassExecutionContext` provides specific functions to access data based on the requirements defined in `ConfigureQueries` (See Table 3.2).

To scale processing for thousands of entities, the framework provides `ParallelForEachEntityChunk`. This method is functionally similar to the serial `ForEachEntityChunk` but leverages the Unreal Engine Task Graph to distribute

the workload across all available CPU cores. For example, this means that while one thread is calculating movement for a batch of entities in Chunk A, another thread can simultaneously process Chunk B (chunk-level parallelization).

However, shifting to parallel execution introduces strict thread-safety requirements: the logic inside the lambda must be re-entrant and avoid writing to shared variables or class members without synchronization. Furthermore, any structural changes, such as adding fragments or destroying entities, cannot be performed immediately. Instead, these operations must be queued through the `FMassExecutionContext` command buffer, which Mass flushes safely on the main thread once the parallel tasks conclude. This approach ensures maximum hardware utilization while maintaining the integrity of the underlying memory archetypes.

Observer Processor

While standard processors run every frame, Observer Processors (inheriting from `UMassObserverProcessor`) are event-driven. They respond to structural changes in the entity's lifecycle.

Observers are tied to specific operations on fragments or tags:

- **OnAdded**: Triggered when a component is added to an entity.
- **OnRemoved**: Triggered when a component is removed.

This is critical for initialization and cleanup logic. For example, an observer might listen for the addition of a `FTransformFragment` to register the entity with a spatial grid, or listen for the removal of a `FHealthFragment` to trigger a death animation or VFX. By using observers, the system avoids polling and instead reacts only when the change occurs.

In the following code snippet, it is possible to see how this kind of processor is linked to a specific fragment and an operation in its constructor:

```
1 UNPCMovementInitializerProcessor::
      UNPCMovementInitializerProcessor() : EntityQuery(*this)
2 {
3   ObservedType = FMassMoveTargetFragment::StaticStruct();
4   Operation = EMassObservedOperation::Add;
5 }
```

3.2 Bridging the ECS and OOP Architectures

While the Mass framework thrives in a purely data-oriented environment, game development practically requires continuous interaction with Unreal Engine's traditional Object-Oriented paradigm. Characters, player controllers, and complex interactive objects are still fundamentally rooted in the `AActor` and `UObject` ecosystem. To solve this architectural divide, Epic Games provides the `UMassAgentComponent`.

The `UMassAgentComponent` acts as a direct bridge between the standard Actor-based world and the Mass Entity Component System. When this component is attached to an Actor, it effectively grants the object a dual existence: it remains a fully functional Actor in the `UWorld`, but it is also registered as a unique Entity within the UMassEntitySubsystem.

Rather than relying entirely on a separate `UMassEntityConfigAsset` to define the entity, the MassAgent component allows developers to add Mass Traits directly to the Actor via the Unreal Editor's details panel. When the Actor initializes (typically during BeginPlay), the component dynamically bakes an `FMassEntityTemplate` based on these selected traits and spawns the corresponding Entity in the ECS backend, maintaining a continuous link between the two via an `FMassEntityHandle`.

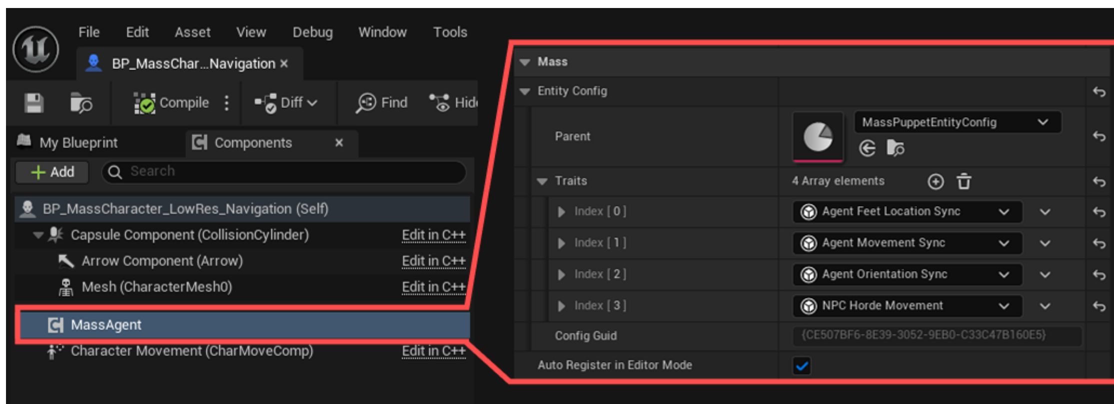

Figure 3.8: Example of Mass Agent Component added to an Actor. On the right side the section of the Details Panel where it is possible to define the entity configuration.

3.2.1 Data Synchronization Modes

Because the Actor and the Entity exist in parallel, data such as world transforms, health, or state variables must be continuously synchronized between the OOP properties and the DOD fragments to prevent state desynchronization. The

MassAgent handles this by allowing developers to define explicit synchronization directions for its attached traits.

This synchronization is typically categorized into three distinct modes, determining which architecture holds the authoritative state:

- **Actor to Mass:** In this mode, the traditional `AActor` is the strict authority. The Actor's components (such as a CharacterMovementComponent or player input) drive the logic. The Mass Entity acts merely as a downstream reflection of the Actor, with specific processors copying data from the UObject properties into the Entity's fragments every frame. This is commonly used for the player character, allowing the ECS to see the player and react to them without Mass actually controlling the player's movement.

- **Mass to Actor:** Here, the ECS simulation is the authority. The highly optimized Mass processors calculate the entity's logic, and the resulting fragment data is pushed outward to update the `AActor`'s properties. This is frequently used for complex AI agents that require the computational speed of Mass for their decision-making and movement, but still need an Actor shell to trigger standard Unreal Engine features like audio, collision events, or complex animation blueprints.

- **Both Ways:** This bidirectional mode allows both architectures to mutate the data depending on the execution phase or specific game events. While this offers the highest flexibility, allowing an agent to be simulated by Mass most of the time, but temporarily overridden by an Actor-based sequence or physics event, it is also the most complex to manage, requiring careful configuration of Processor execution groups to avoid race conditions and frame-delay artifacts.

3.3 Additional Mass Plugins

While MassEntity provides the foundational memory architecture and processing pipeline, Unreal Engine extends its capabilities through a suite of specialized plugins. These modules, developed and tested during the creation of the City Sample technical demo [8], provide the high-level logic required to transform raw entities into complex, autonomous agents. As of Unreal Engine 5.6, these systems remain marked as Experimental, signaling that while they are architecturally robust, their APIs continue to evolve to meet the needs of large-scale open-world simulations, constituting a production risk.

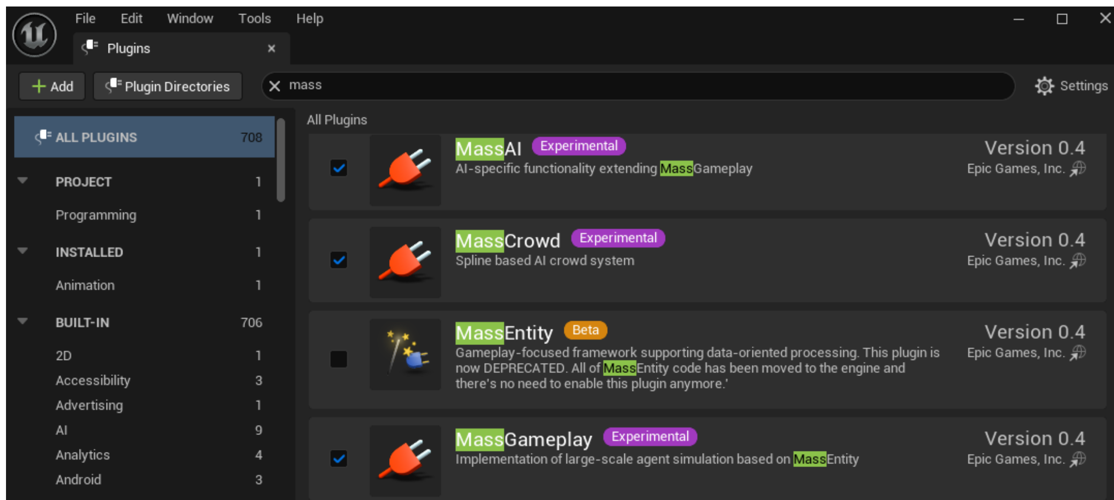

Figure 3.9: Plugin window showing the Mass Plugins listed. While MassEntity plugin is deprecated and now integrated in the engine codebase, MassAI, MassCrowd and MassGameplay are still marked as experimental and can be enabled from this panel.

3.3.1 MassGameplay

The MassGameplay plugin acts as the primary bridge between the data-oriented ECS and the traditional Actor-based environment of Unreal Engine. It provides the essential gateway for world interaction, representation, and lifecycle management.

- **Mass Representation Subsystem:** responsible for the visual lifecycle of entities. It utilizes an intelligent Actor Pooling system to switch between high-fidelity Actors for nearby agents and lightweight Instanced Static Meshes (ISM) for distant crowds, maintaining visual density without the overhead of thousands of ticking Actors.
- **Mass LOD Subsystem:** a high-performance spatial query system that calculates the Level of Detail (High, Medium, Low, Off) for every entity. This LOD state is then utilized by other subsystems (Representation, Replication, and Movement) to scale computational costs based on viewer proximity.
- **Mass Replication Subsystem:** implements a specialized one-way (Server-to-Client) replication pipeline. By operating directly on memory fragments rather than Actor properties, it enables the synchronization of massive entity populations across the network.
- **Mass Signals and StateTree:** these subsystem integrate hierarchical state machines with the ECS. Mass Signals provides a lightweight notification system

to wake up entities for processing, while the StateTree Subsystem allows agents to execute complex, multi-state behaviors driven by a unified, signal-based logic flow.

- **Mass SmartObject Subsystem:** links entities to the SmartObject database. This allows agents to query, reserve, and interact with environmental objects (e.g., chairs, doors, or vending machines) via spatial partitioning, enabling rich environmental interactions at scale.

3.3.2 MassAI

The MassAI plugin provides the structural framework for agent intelligence by defining standardized data interfaces and execution pipelines. Architecturally, it does not implement specific behaviors. Instead, it provides the abstractions and execution schemas required to bridge the Mass ECS with decision-making systems like StateTree.

- **MassAIBehavior:** defines the foundational slots for AI logic. It provides the fragments and processors necessary to manage an entity's internal thought state, acting as the host for StateTree execution and goal-oriented transitions.
- **MassAIDebug:** offers a diagnostic layer that peeks into the abstract containers. It provides visualization for internal AI states and diagnostic tools that are essential for troubleshooting the non-deterministic behaviors of large-scale agent populations.
- **MassAIReplication:** specialized for the AI domain, this module ensures that the results of the AI's decision-making are synchronized across the network. It prioritizes the replication of high-level state changes, utilizing the LOD system to ensure that bandwidth is only spent on AI logic that is currently relevant to the client.

3.3.3 MassCrowd

While the previous plugins provide the generic tools for AI and gameplay, MassCrowd is the specialized layer designed for the high-density simulation of pedestrian populations. It was the primary vehicle for the crowd logic seen in the City Sample, focusing on the transition from individual agent navigation to collective flow. The plugin's high-level responsibilities include:

- **Crowd Visualization and Animation:** it provides specialized processors that allow the crowd to have varied, vertex-animated appearances that are synchronized with their movement speed, enabling thousands of unique animations without the skeletal mesh overhead.

- **Navigation Specialization (ZoneGraph Integration):** MassCrowd implements the specific logic required for agents to follow ZoneGraphs, the lightweight, lane-based navigation paths that replace traditional NavMesh. It handles lane-switching and intersection logic, allowing entities to respect the rules of a structured urban environment.
- **Proximity and Avoidance Tuning:** building upon the base avoidance systems, this plugin includes specific tuning for pedestrian dynamics. It manages how agents wait at crosswalks or navigate around obstacles, ensuring that high-density groups do not result in clumping or collision-based artifacts.

Chapter 4

Large-Scale Entity Representation

The true power of a Data-Oriented simulation is often measured by its visual density, the ability to render thousands of individual agents without compromising the frame rate or visual fidelity. However, as the entity count scales up, the primary bottleneck shifts from CPU logic to GPU throughput and draw calls.

In traditional game developement, the visual presence of an object is inextricably linked to its logical container, in Unreal Engine this container is typically the `AActor`. In the Mass Framework this link is severed. Mass Representation serves as the managing layer that translates abstract entity data into high-performance visual primitives, primarily batching thousands of individual entity transforms into a single Instanced Static Mesh draw call.

This chapter explores this architectural shift by treating representation as a dynamic state rather than a static component property. We begin by diagnosing the inherent limitations of the Actor-centric model, specifically analyzing how `UObject` overhead and individual draw calls create a performance ceiling that traditional architectures cannot breach.

Once these bottlenecks are established, the discourse moves to the Mass LOD system, which serves as the simulation's brain for visibility and performance throttling, orchestrating agent fidelity based on spatial relevance. This leads into a technical examination of standard representation strategies, where we analyze the mechanics of Instanced Static Meshes (ISM) and the AnimToTexture pipeline, the dual frameworks that enable high-fidelity crowd visualization at scale. Finally, we explore alternative and non-standard approaches, such as Niagara-based representation.

4.1 Actor-based Representation Limitations

In the standard Unreal Engine workflow, the `AActor` serves as a monolithic container that inextricably links an object's logical state with its visual presence. While this is ideal for individual player characters, it introduces some critical bottlenecks when scaling to thousands of agents.

The `UObject` system comes with a computational overhead: each actor requires dedicated memory for reflection metadata, property tracking and integration with the garbage collection system. Simply maintaining the memory footprint of thousands empty actors can saturate the system resources even before simulation logic is processed.

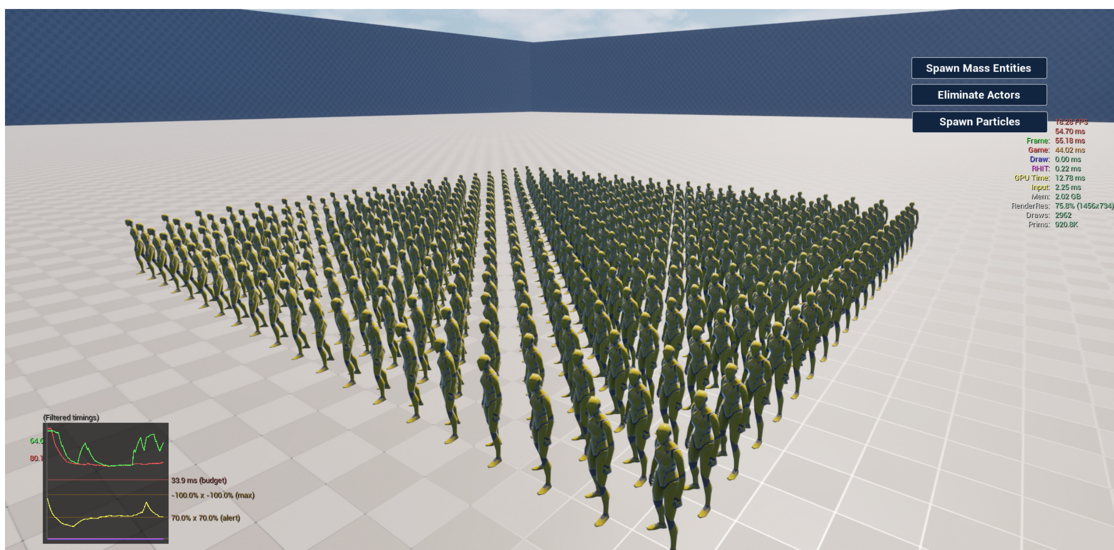

Figure 4.1: CPU performance tanking when a grid of actors is instanced in an empty scene. The Game Thread took 44ms to process this frame, hence lowering the FPS.

This overhead extends into the rendering pipeline, creating a draw call ceiling. Because each Actor's `UStaticMeshComponent` typically generates its own unique render commands, the CPU's command buffer becomes a critical bottleneck. The CPU spends so much time preparing and submitting tens of thousands of individual instructions that it cannot keep pace with the GPU, leaving the graphics hardware idle while the CPU-side rendering thread is saturated.

Finally, the architecture suffers from sequential scene updates driven by the hierarchical `USceneComponent` system. Every frame, the engine must recursively calculate world-space transforms, update bounding volumes, and perform individual visibility culling checks for every agent on the Game Thread. When the number

of entities scales, this transform overhead alone can exceed the entire 16.6ms per frame budget, making real-time performance impossible within the traditional Actor-centric paradigm.

4.1.1 The Mass Solution

Mass Representation addresses the limitations of the Actor model by severing the link between an entity's logical existence and its visual manifestation. In this paradigm, the entity remains a lean data construct, while its physical presence in the world is managed by a dedicated translation layer. This architectural shift replaces the static one-to-one relationship of Actors with a dynamic visual proxy system, where the method of rendering is merely a transient state determined by the entity's current relevance to the observer.

The efficiency of this solution is rooted in the way the Mass Representation Processor interacts with the entity's existing spatial data. Instead of each agent being responsible for its own rendering, the processor iterates over the `FTransformFragment` data in a high-performance, cache-aligned stream. This allows the system to gather the transforms of thousands of entities simultaneously and push them into a centralized rendering backend.

The primary tool for this large-scale visualization is the Instanced Static Mesh (ISM), or more specifically, the Hierarchical Instanced Static Mesh (HISM). By batching these gathered transforms into a single component, Mass collapses what would traditionally be thousands of individual draw calls into a handful of GPU-optimized operations. This is critical because every draw call represents a significant communication overhead from the CPU to the GPU. In a non-instanced scenario 30000 entities, each with a mesh and unique material attributes, would generate upwards of 60000 draw calls, overwhelming the CPU's command buffer and leading to a complete collapse in frame rate.

Through the Mass Representation Subsystem, these entities are not merely rendered, but managed through a transactional phase. Instead of individual updates, the system pushes contiguous arrays of transforms and Per-Instance Custom Data (PICD), such as animation indices or color variations, to the GPU buffers in a single, streamlined operation. This system does not apply a uniform representation to all entities; instead, it utilizes a multi-tiered Level of Detail (LOD) logic to determine whether an entity should be rendered as a high-fidelity Actor, a high-performance HISM instance, or culled entirely based on its spatial relevance to the viewer.

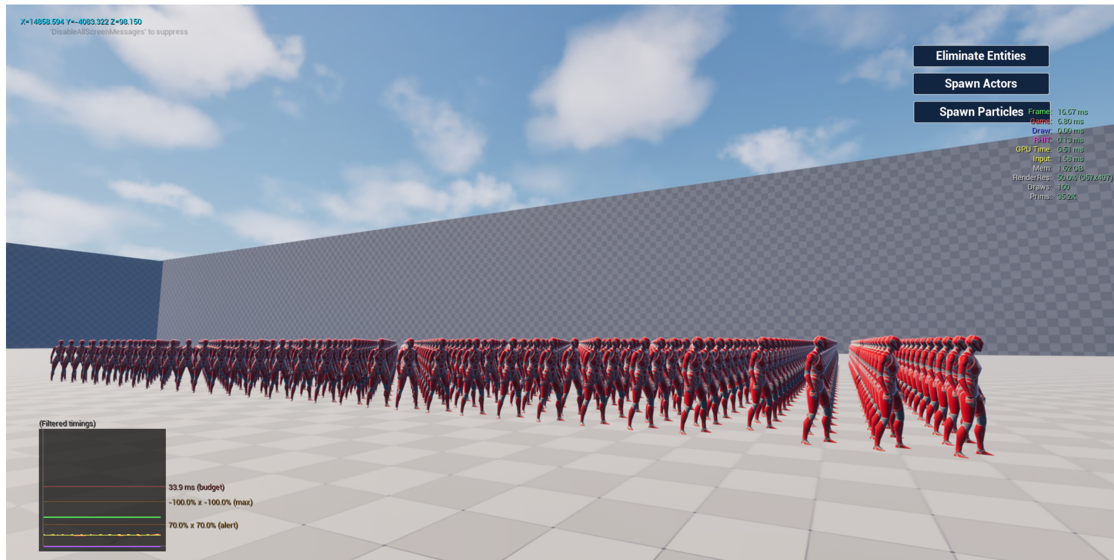

Figure 4.2: ISM based representation managed by Mass. As we can see in the top-right corner, this type of representation significantly reduces the computational weight on the Game Thread we noticed in the case of the Actor representation.

Processor Registration

A critical practical consideration when implementing this architecture is processor registration. In a default Unreal Engine 5.6 project, the specific processors responsible for translating entity data into visual proxies and managing spatial relevance are not always active out-of-the-box.[11]

For the representation and LOD pipelines to function, their respective processors must be explicitly told to run during the simulation tick. This is achieved by enabling `bAutoRegisterWithProcessingPhases` either through the Editor's Project Settings or directly within the `DefaultMass.ini` configuration file. Specifically, the framework requires the activation of the core Representation Processor, the Visualization LOD Processor, and the LOD Collector Processor:

```
1 [/Script/MassRepresentation.MassRepresentationProcessor]
2 bAutoRegisterWithProcessingPhases=True
3
4 [/Script/MassRepresentation.MassVisualizationLODProcessor]
5 bAutoRegisterWithProcessingPhases=True
6
7 [/Script/MassLOD.MassLODCollectorProcessor]
8 bAutoRegisterWithProcessingPhases=True
```

Without these explicit registrations, entities may simulate correctly in the ECS memory space, but the framework will fail to gather viewer data or push spatial transforms to the GPU buffers, resulting in an invisible simulation.

4.2 Mass Level Of Detail

The Mass LOD system serves as the central decision-making engine for entity relevance. Unlike traditional LOD systems that only swap mesh resolutions, Mass uses LOD to drive a multi-system load balancing strategy. By categorizing entities into four distinct states (High, Medium, Low, and Off) the framework can dynamically throttle everything from rendering fidelity to the frequency of C++ logic updates.

4.2.1 Mass LOD Overview

The LOD Subsystem does not act alone, it serves three primary clients that use its spatial calculations to optimize different hardware bottlenecks:

- **Mass Visualization LOD**: manages the visual proxy swapping discussed in the previous section. It determines if an entity should be an Actor, an ISM, or culled entirely based on distance and camera frustum visibility
- **Mass Simulation LOD**: implements time throttling and variable ticking. It allows the CPU to skip update frames for distant entities. For example, we could be updating a distant agent's logic 5 times per second instead of 60.
- **Mass Replication LOD**: optimizes network bandwidth by reducing the frequency of data synchronization for entities far from the player

4.2.2 Mass LOD Subsystem

The `UMassLODSubsystem` acts as the global state provider for spatial relevance, serving as the synchronization point between the Game Thread's camera data and the Mass ECS simulation. In a high-density simulation, having tens of thousands of entities independently query the `APlayerController` for camera coordinates would create a relevant CPU bottleneck. The subsystem solves this by centralizing viewer management, ensuring that the heavy lifting of gathering engine-side world state is performed only once per frame.

Technically, the subsystem is hooked into the `PrePhysics` phase of the `UMassSimulationSubsystem`. At the start of this phase, it iterates through all active controllers, camera components, and World Partition streaming sources to populate a internal temporary array of `FViewerInfo` structures. Each entry in this

array serves as a lightweight snapshot of an observer's perspective, encapsulating the following properties:

- **Location and Rotation**: the world-space origin and orientation of the viewer
- **Field of View (FOV) and Aspect Ratio**: critical parameters used to derive the frustum planes for visibility culling

This centralized array acts as the source of truth for all downstream LOD clients. Rather than entities pulling data from the world, the `UMassLODCollectorProcessor` pushes the subsystem's cached viewer data into the entity fragments. To determine how an entity archetype should interact with this viewer data, Mass provides two primary collector traits that define the policy for relevance calculation:

- `UMassDistanceLODCollectorTrait`: this trait is the more computationally lean of the two. It adds the `FMassCollectDistanceLODViewerInfoTag`, triggering the `UMassLODDistanceCollectorProcessor`. This processor calculates relevance based purely on radial proximity to the nearest viewer, ignoring the viewer's orientation. This is ideal for systemic logic, such as background AI, that must remain active even if the agent is behind the camera.
- `UMassLODCollectorTrait`: the standard collector trait adds the `FMassCollectLODViewerInfoTag`, triggering the `UMassLODCollectorProcessor`. This version utilizes the full frustum data provided by the subsystem to perform a dot-product based visibility test. It is the primary choice for visualization, allowing the system to demote entities to an Off state the moment they exit the camera's view.

Rather than entities pulling data from the world, these processors push the subsystem's cached viewer data into the entity fragments. This is facilitated by the `TMassLODCollector` processor, which performs high-speed comparisons between the global `FViewerInfo` array and the local `FTransformFragment` of each entity.

The output of this process is the population of the `FMassViewerInfoFragment`. By storing the `ClosestViewerDistanceSq` and `ClosestDistanceToFrustum` directly on the entity, Mass allows other systems, such as the Simulation and Representation processors, to perform constant-time checks on an entity's relevance. This architecture effectively decouples the complexity of the world's viewing conditions from the per-entity logic, ensuring that even with a large number of agents, the overhead of determining visibility remains a linear, cache-friendly operation.

The final state of an entity's relevance is represented by a set of mutually exclusive tags defined in `MassLODFragments.h`:

- `FMassHighLODTag`
- `FMassMediumLODTag`
- `FMassLowLODTag`
- `FMassOffLODTag`

These tags are updated by the collector processors and serve as the primary mechanism for structural filtering. By including one of these tags in a `FMassEntityQuery` requirement, a processor can be restricted to execute only on entities within a specific proximity or visibility range, effectively pruning thousands of distant agents from the execution set at the archetype level. In the following example, the `EntityQuery` will only contain entities with the `FMassHighLODTag`:

```
1 void UMyProcessor::ConfigureQueries(const TSharedRef<
     FMassEntityManager>& EntityManager)
2 {
3   EntityQuery.AddTagRequirement<FMassMediumLODTag>(
     EMassFragmentPresence::None);
4   EntityQuery.AddTagRequirement<FMassLowLODTag>(
     EMassFragmentPresence::None);
5   EntityQuery.AddTagRequirement<FMassOffLODTag>(
     EMassFragmentPresence::None);
6 }
```

Beyond archetype filtering, the system allows for more granular, logic-based branching within a single processor. The `FMassRepresentationLODFragment` contains a `LOD` property of the `EMassLOD` enumeration type. This allows a single query, which might include all entities from High to Low LOD, to differentiate behavior through standard conditional checks. For example, a movement processor might update an agent's avoidance steering at `EMassLOD::High`, but switch to a simplified linear interpolation at `EMassLOD::Low`. This dual-layered approach, combining tag-based query filtering for performance with enum-based branching for behavioral variety, provides the developer with total control over the simulation's performance over fidelity.

```
1 void UMyProcessor::Execute(FMassEntityManager& EntityManager,
     FMassExecutionContext& Context)
2 {
3    EntityQuery.ForEachEntityChunk(Context, ([&](
     FMassExecutionContext& Context)
4    {
5        const TConstArrayView<FMassRepresentationLODFragment>
     LODRepresentationList =
```

```
6                       Context.GetFragmentView<
    FMassRepresentationLODFragment>();
7
8           const int32 NumEntities = Context.GetNumEntities();
9           for (int32 i = 0; i < NumEntities; ++i)
10          {
11              const EMassLOD::Type CurrentLOD =
    LODRepresentationList[i].LOD;
12
13              if (CurrentLOD == EMassLOD::High)
14              {
15                  // Execute high-fidelity simulation logic
16                  // e.g., Full obstacle avoidance steering
17              }
18              else if (CurrentLOD == EMassLOD::Low)
19              {
20                  // Execute simplified simulation logic
21                  // e.g., Simple forward velocity
22              }
23          }
24      }));
25  }
```

4.3 Mass Representation

The `UMassRepresentationProcessor` is the execution engine that synchronizes an entity's logical state with its visual manifestation. It operates by evaluating the results stored in the `FMassViewerInfoFragment` and the current LOD tags to determine the appropriate `EMassRepresentationType`. This enum acts as a state machine for the entity's visual proxy, supporting four primary modes:

- **HighResSpawnedActor**: the entity is represented by a full, high-fidelity `AActor`. This is reserved for the highest LOD (typically `EMassLOD::High`), enabling complex skeletal mesh animations, physics interactions, and full component functionality.

- **LowResSpawnedActor**: a lightweight version of the Actor representation. This is often used for entities that are visible but not in the immediate foreground. It may swap the complex skeletal mesh for a simpler one or disable expensive components, such as complex AI or ticking components, while maintaining the `AActor` wrapper for basic interaction or sound.

- **StaticMeshInstance**: the entity is rendered via the HISM (Hierarchical Instanced Static Mesh) system. This is the ideal representation for performance, allowing thousands of agents to be drawn in a few GPU batches. In this state, the entity no longer exists as an `AActor` in the world; it is merely a transform and a set of custom data in a GPU buffer.
- **None**: The entity is completely culled. The mesh is removed from the render pipeline entirely, though the underlying Mass simulation continues to run.

The transition between these states is managed through the `FMassRepresentationFragment`, which tracks both the `CurrentRepresentation` and `PrevRepresentation`. This allows the system to handle the "setup and teardown" of visual assets, such as calling the `UMassActorSpawnerSubsystem` to spawn an Actor or clearing an index from a HISM, only when a state change occurs, preventing expensive per-frame logic.

Figure 4.3: Different entity representation based on the distance from the viewer. The red entities are Static Mesh Instances, the yellow ones are LowResSpawnedActors and the white ones are HighResSpawnedActors

4.3.1 Vertex Animated ISM

For large crowds of entities, traditional skeletal mesh skinning (calculating bone offsets on the CPU) is a primary bottleneck. A way to bypass this is found by

leveraging AnimToTexture (Vertex Animation Texture).

In this workflow, skeletal animation data is baked into a 2D HDR texture where rows represent bones and columns represent time frames. This allows the engine to bypass the skeletal mesh pipeline entirely. During the `StaticMeshInstance` representation phase, the HISM vertex shader samples this texture to offset vertices directly on the GPU. This transforms the computational cost of an animated character into that of a static mesh with a specialized material. To ensure the crowd does not appear synchronized, it is possible to push Per-Instance Custom Data (PICD) via Mass to the HISM. In the standard AnimToTexture shader at scale, the two critical floats are typically:

- **Animation State**: This tells the shader which row of the baked texture to read, allowing to select the animation among the baked ones. This is often a quantized float derived from the `FMassCrowdAnimationFragment`.
- **Normalized Animation Phase**: Instead of a global time offset, Mass frequently calculates and pushes the actual playback position (as a value going from 0.0 to 1.0) of the animation loop. This is calculated on the CPU by taking the entity's internal clock, adding a random seed, and modulo-ing it by the animation duration.

Since UE 5.4, VAT supports Nanite-enabled static meshes. While Nanite provides excellent geometric scaling, it should be used judiciously: for low-poly crowd agents, the overhead of Nanite's cluster management may outweigh the benefits of its level-of-detail scaling.

The practical application of this vertex-driven animation pipeline is most notably demonstrated in Unreal Engine's City Sample project, where the `UMassCrowdAnimationProcessor` serves as a specialized extension of the core Mass framework to orchestrate the movement of thousands of simulated pedestrians.

4.3.2 Niagara-Based Representation

While Hierarchical Instanced Static Meshes (HISMs) are the standard for Mass visualization, the framework is architecturally decoupled enough to allow for a total shift in the rendering pipeline. By setting aside the standard representation processors, a developer can treat Mass entities as a raw data source for Niagara Particle Systems.

This approach represents the massive tier of simulation, moving from a CPU-side push model to a GPU-side pull model. In this setup, a single Niagara Emitter samples the Mass ECS fragments (Transform, Velocity, Animation State) directly from GPU memory to drive tens of thousands of particles that look and behave like complex skeletal agents.

Despite the clear performance benefits for very large scale crowds (100,000+ entities), the current state of Niagara integration is non-trivial. Because the standard Unreal Engine 5.6 toolset lacks a native bridge between Mass and Niagara, a developer must bridge the gap manually. This requires:

- **Custom C++ Middleware**: creating a specialized Niagara Data Interface (NDI) to map ECS memory chunks to Niagara parameters
- **Manual Lifecycle Management**: Synchronizing the birth and death of Mass entities with the spawning and reaping of Niagara particles
- **Shader Complexity**: Re-implementing the VAT (Vertex Animation Texture) logic within the Niagara material graph to ensure visual consistency with the HISM-based agents

The primary precedent for this hybrid architecture is found in Unreal Engine's City Sample (The Matrix Awakens Demo). To populate the vast streets of the city without overwhelming the CPU, the developers utilized a tiered representation strategy.

While closer pedestrians use Actors and HISMs, the background crowd, the thousands of agents visible in the distance or from high-rise perspectives, is rendered entirely through a specialized Niagara-based system. This implementation serves as a proof-of-concept, demonstrating that while the integration is technically demanding and requires bespoke C++ extensions, it is the key to breaking through the performance limitations of traditional mesh instancing.

Ultimately, the introduction of the `NiagaraDataChannelSubsystem` in Unreal Engine 5.7 represents a promising shift in the framework's maturity; by standardizing the communication between the ECS and the VFX pipeline, it transforms what was once a complex optimization into a viable pathway for future large-scale crowd simulations. The `UNiagaraDataChannelSubsystem` acts as a centralized broker that allows the Mass ECS to publish data to a named channel, which any Niagara System can then consume as a data interface. In this architecture, a specialized Mass Processor iterates through entity chunks and writes fragment data (such as `FTransformFragment` or custom animation states) into the Data Channel's producer buffer via the subsystem. The Niagara GPU emitter then binds to this same channel, allowing its simulation stages to read the buffered Mass data at the start of the frame. This effectively replaces the need for a low-level C++ memory pull with a thread-safe, engine-managed stream, significantly simplifying the synchronization of particle lifecycles with Mass entity states.

Chapter 5

Decision Logic and State Management

While previous chapters focused on the spatial and visual aspects of the simulation, this chapter examines the logic layer that drives agent behavior. A significant challenge in ECS-based architectures is implementing complex, branching behavior without sacrificing the performance gains of data-orientation. In a system where data is stripped of its traditional object-oriented context, the mechanism for decision-making must be as lean and cache-coherent as the data it manipulates.

Throughout this project, the implementation of agent intelligence evolved into a modular, hierarchical system utilizing the Unreal Engine State Tree. The goal of this chapter is to detail how the State Tree functions as a high-performance brain within the Mass ecosystem. It provides a technical analysis of how the `UMassStateTreeSchema` decouples stateful logic from the `UObject` lifecycle, allowing designers to utilize a node-based interface for complex AI while maintaining the simulation's ability to scale. By analyzing the transition from a traditional polling model to a reactive, signal-driven architecture, this chapter demonstrates how the State Tree coordinates high-level intent while delegating computational muscle to specialized Mass Processors.

5.1 State Tree

This section provides a technical overview of the State Tree asset and its execution model, analyzing first its funcionalities outside the framework's integration context.

While often associated with AI, the State Tree is a general-purpose, hierarchical state machine (HSM) designed to handle any stateful logic within Unreal Engine. Its architecture is specifically optimized for high-performance scenarios, such as the Mass Framework, by decoupling static logic from runtime instance data.

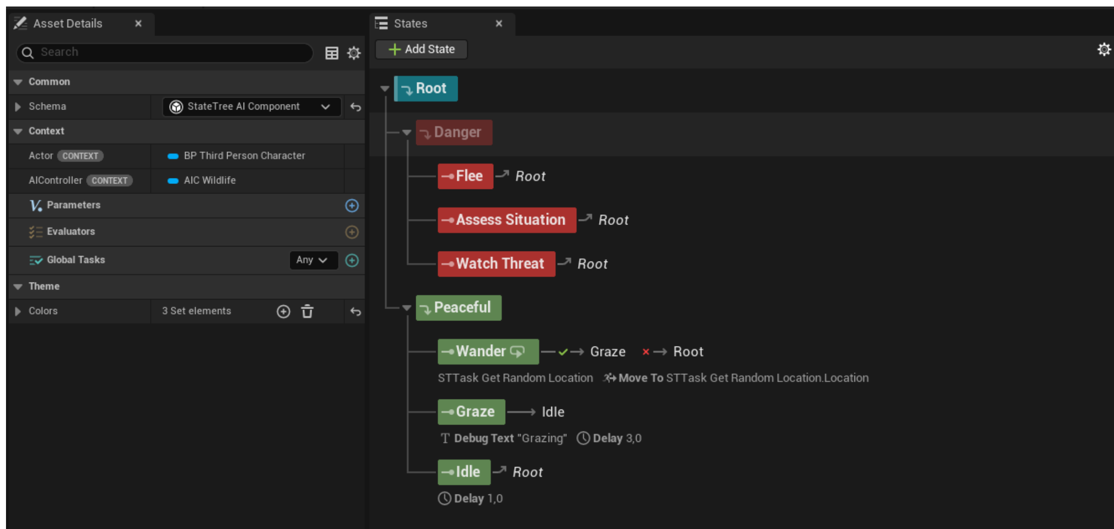

Figure 5.1: An example of StateTree editor visualization.

5.1.1 Architecture

At its core, a State Tree is a hierarchical organization of states where logic is encapsulated into modular, reusable blocks. Unlike traditional Finite State Machines (FSMs), where transitions can become a "spaghetti" of connections, the State Tree uses its hierarchy to organize data and define fallback behaviors.

States

At the apex of the hierarchy is the Root State, which serves as the mandatory entry point for the tree's execution flow. It is the first state evaluated during initialization and remains implicitly active as the ultimate ancestor of any selected leaf.

Other than the Root State, State Tree consists of two primary types of states: intermediate states and leaf states. Intermediate states act as organizational nodes or decision points. They do not contain actions, but instead define how to select a child state. Leaf states are the terminal nodes of a branch. When a leaf state is selected, the state itself and all its parent states in the hierarchy become active.

Tasks

States contain tasks, the functional building blocks of a State Tree. Tasks contain the actual logic to be executed when a state is active. Each state can host multiple tasks and when a state is activated all of them run concurrently. Typically, intermediate states host configuration tasks, such as setting gameplay tags, while leaf state host behavior tasks, such as MoveTo or PlayAnimation.

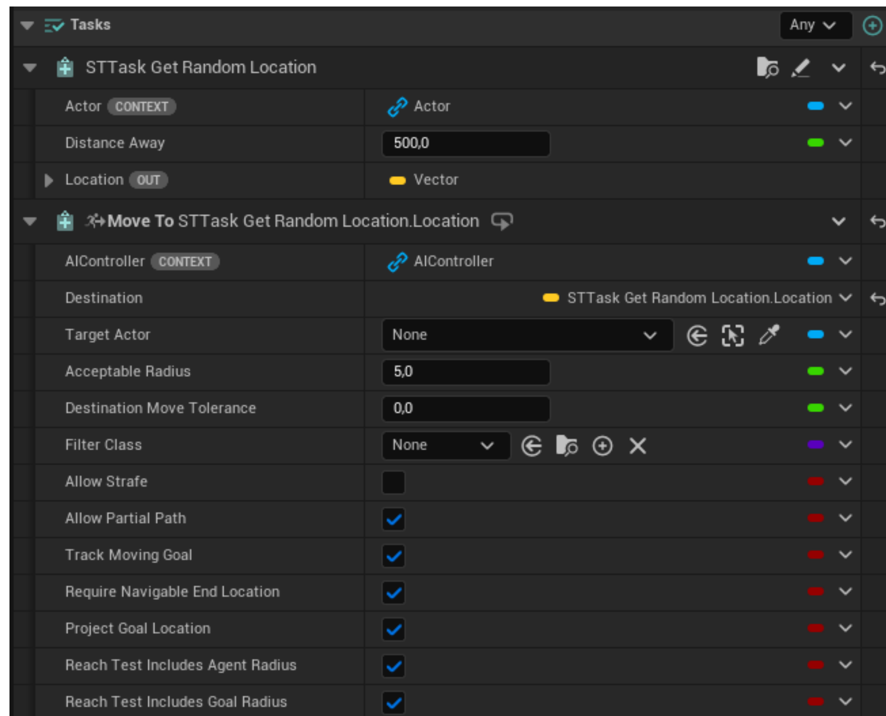

Figure 5.2: When clicking on a State it is possible to add and configure Tasks in this section of the detail panel.

Other than regular tasks that can be defined in relation to states, State Tree gives the developer the possibility to define global tasks. These run for the entire duration of the tree's execution, providing a persistent layer for event listening or data exposure. They are ideal for persistent logic, such as managing a high-level state timer or providing a continuous stream of data from an external subsystem that every state in the tree might need to access.

State Tree architecture also gives the possibility to use Evaluators: specialized, stateless nodes designed to pre-process information before the selection phase begins. Unlike tasks, which execute behavior, evaluators run at the start of the tree's tick to query the environment, perform complex calculations, and expose the results as output properties. By centralizing data preparation into these nodes, the State Tree ensures that conditions and tasks have access to up-to-date information (such as the distance to the nearest threat or a weighted utility score) without needing to redundantly recalculate that data across multiple states.

In Unreal Engine 5.6 Tasks and Evaluators can be defined in blueprint by extending respectively `StateTreeTaskBlueprintBase` and `StateTreeEvaluatorBlueprintBase`.

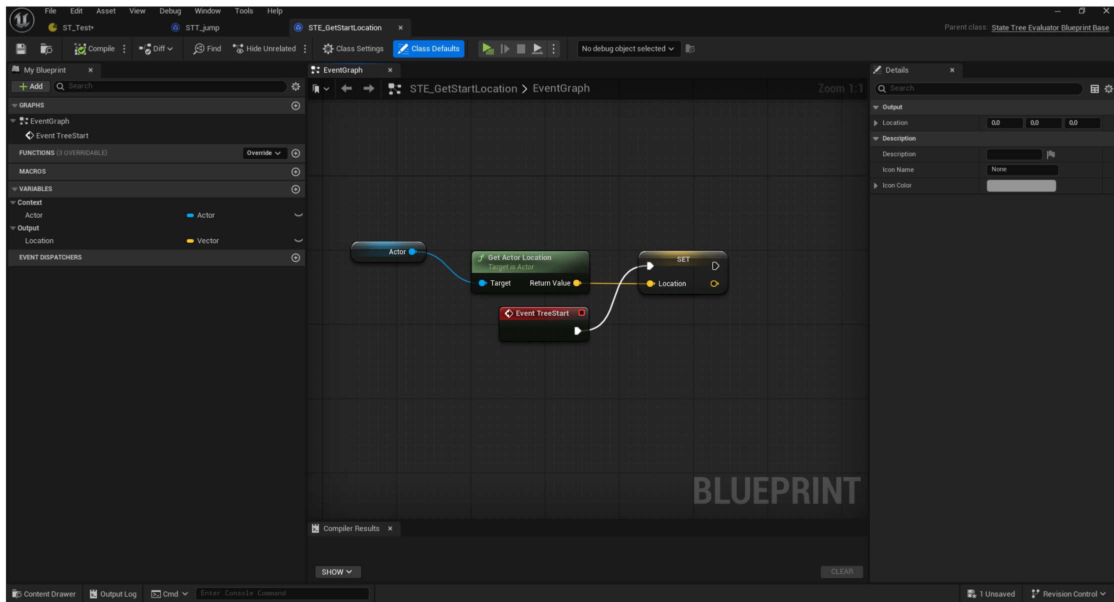

Figure 5.3: A simple evaluator defined in Blueprint that returns the current location of the player.

5.1.2 Selection and Transition Flow

The execution of a State Tree is defined by a continuous cycle of State Persistence and Evaluation Phases. Unlike traditional state machines that might transition blindly into a connected node, the State Tree utilizes a Selection-First logic. This means the system must validate the entire path from the root down to a specific leaf before any state changes are officially committed.

The evaluation process begins whenever the tree is initialized or an active transition is triggered. At this point, the `FStateTreeExecutionContext` enters a pre-emptive evaluation pass to determine the next valid branch.

Enter Conditions

Before any state can be considered for activation, it must first satisfy its enter conditions. These act as logical gates: if a state's conditions evaluate to false, that entire branch of the hierarchy is instantly discarded. This pruning allows the system to bypass invalid behaviors with minimal computational overhead.

Selection Behavior

Once a state passes its initial conditions, the context determines how to tunnel deeper into the hierarchy. In Unreal Engine 5.6, this selection logic offers a suite of

strategies for different architectural needs:

- **None and Try Enter:** serve as the most direct behaviors, typically used for leaf states or states where the focus is on the state's own tasks rather than its descendants.
- **Try Select Children in Order:** provides a predictable, priority-based approach, evaluating children from top to bottom until the first valid candidate is found.
- **Try Select Children at Random:** introduces stochastic variety, picking a child based on either uniform or weighted probabilities. Ideal for breaking the repetition in ambient behaviors.
- **Try Select Children with Highest Utility and Try Select Children at Random Weighted by Utility:** they offer the most dynamic, data-driven approaches. These behaviors use a scoring system to evaluate potential child states, either selecting the absolute best performer or using those scores to influence a weighted random choice.
- **Try Follow Transitions:** allows a state to act as a reusable logic bundle, where the selection of a child state is determined by the transition rules defined within that state itself.

This evaluation is recursive. The system continues to tunnel deeper into the hierarchy, applying these selection rules at every level until it successfully identifies a leaf state. The selection is only finalized once this terminal node is reached. If the evaluation encounters a dead end, where no child states satisfy their conditions, the selection fails, and the system typically falls back to the Root State to restart the search, ensuring the entity never remains in an undefined or stuck logical state.

Transition Lifecycle

Transitions define the specific conditional logic that drives the system to pivot from its current active branch to a new destination. Unlike the transitions in Finite State Machines, transitions in a State Tree are governed by the same hierarchical principles as the states themselves.

A defining characteristic of this execution model is its hierarchical evaluation order. When a transition is triggered, the engine does not perform a broad search across the entire asset. Instead, it evaluates potential transitions starting from the current leaf state and goes upward through each parent toward the Root. This type of flow ensures that the behavior closest to the current action is given the first opportunity to determine the next state.

In its current state, the triggers available for defining the transitions have been refined to support more granular control over the entity's lifecycle:

- **On State Succeeded / Failed / Completed:** These are outcome-based triggers. While "Succeeded" and "Failed" handle specific logic, like a pathfinding task failing, "Completed" acts as a catch-all for whenever the tasks in a state finish their work, regardless of the result.
- **On Event:** This provides a reactive interface for the State Tree. By utilizing Gameplay Tags or custom Data Structs as payloads, external systems can "ping" the tree, allowing it to respond dynamically to world changes without waiting for a task to finish.
- **On Tick:** This trigger allows for continuous polling of conditions. While more computationally expensive than event-driven logic, it is essential for behaviors that must react to environmental variables that do not send discrete events.
- **On Delegate:** This specialized trigger allows the State Tree to bind directly to C++ or Blueprint delegates. It is particularly useful for decoupling the tree from external systems, allowing a state change to be fired the moment a specific delegate is broadcasted.

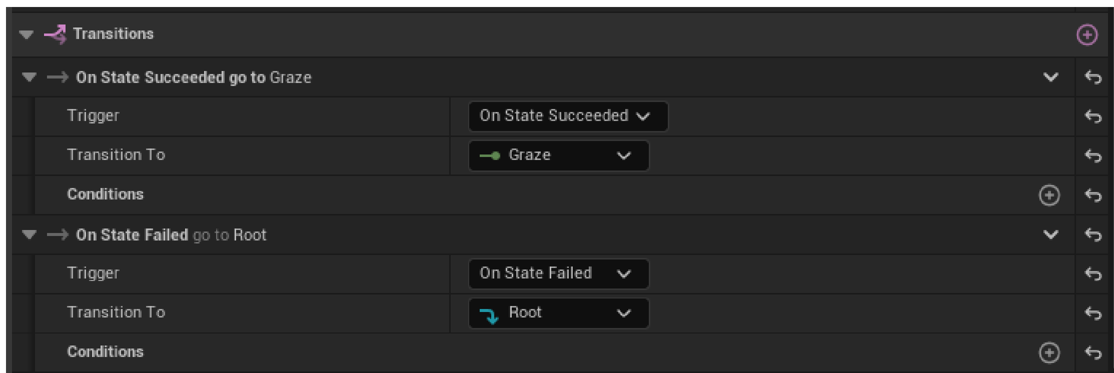

Figure 5.4: Example of transition definition. The next state in this case will be different based on the state outcome.

Beyond the transitions explicitly defined by the developer, the State Tree incorporates Implicit Transitions to ensure the execution flow never enters an invalid state. These act as a built-in safety net for the tree's lifecycle. For instance, upon the initial start of the tree, the system performs an implicit transition to the Root State to begin the first evaluation. During runtime, if a leaf state completes but provides no explicit transition for its outcome, the system automatically goes up to the parent to check for a completion transition there.

If this search reaches the top of the hierarchy without finding a valid path, either because no transitions were configured or because all potential target states failed their Enter Conditions, the system triggers a final fallback by transitioning back to the Root State. This recursive search for a valid state ensures that the entity remains logically active, though it highlights a critical design principle: developers should structure their state hierarchies so that a valid selection is always available, preventing the entity from perpetually rebounding to the root in a logic loop.

To manage the complexity of these overlapping rules, the State Tree utilizes a Priority system that can modify the standard evaluation flow. While the default behavior favors the leaf, a developer can assign a higher priority to a transition in a parent state. This allows for global interrupts, critical logic that must take precedence over any local behavior. A classic example is a "Damage Taken" event hosted in a high-level parent state; by setting this to a higher priority, the system can ensure that a "Flee" behavior immediately overrides a specific leaf, regardless of where the entity was in its local task lifecycle.

A critical aspect of the State Tree execution model is that all tasks within an active state and its active parents run concurrently. However, the lifecycle of the state is often dictated by the first task to reach a conclusion. When any single task signals a "Succeeded" or "Failed" state, it immediately triggers the transition evaluation process for the entire tree. This means that a state is effectively as long as its fastest-completing task. For instance, if a state contains a long-running "Patrol" task and a shorter "Wait" task, the completion of the "Wait" task will initiate a transition, potentially interrupting the "Patrol" logic before it finishes. This behavior requires developers to be intentional about task composition, ensuring that auxiliary tasks do not accidentally signal completion and prematurely terminate the primary behavior of the state.

5.1.3 State Tree Schemas

A State Tree Schema defines the specific use case for a tree by establishing the execution environment and the data contract required for it to run. By selecting a schema, the developer informs the engine of what external data will be available and which types of nodes can be used to build the logic.

In practice, the schema serves three primary functions: it specifies the Context Data, the external objects or structs (such as an `AActor`, `AIController`, or `FMassEntityHandle`), that must be provided by the owner for the tree to tick; it restricts which Tasks, Evaluators, and Conditions are visible in the editor; it defines how the tree is ticked and who owns the instance data.

Unreal Engine 5.6 provides several built-in schemas tailored for different systems:

- **State Tree Component Schema:** The default choice for actor-based logic, assuming a single actor owner and a standard `UStateTreeComponent`.

- **State Tree AI Component Schema:** A specialized version of the above that guarantees access to an AAIController, making it the standard for traditional NPC behavior.

- **Mass Behavior:** the critical schema for high-performance ECS logic. It allows the State Tree to run on Mass Entities rather than Actors, providing direct access to Mass Fragments and allowing the `MassStateTreeProcessor` to manage execution across thousands of entities.

- **Camera Director:** A highly specialized schema used for camera logic and transitions. It provides a context tailored for managing camera states, blending, and framing.

5.1.4 Data Management

In contrast to the decoupled Blackboard architecture used in traditional Behavior Trees, where data is stored in a general-purpose key-value map, the State Tree employs a strictly typed Property Binding system. This approach creates a direct pipeline between data providers and data consumers, significantly reducing the memory overhead and string-key lookups that can weight on large-scale AI systems.

The foundation of the State Tree's data awareness is Context Data. While most tasks need data to function, they do not own that data. Instead, the State Tree Schema defines a set of required external objects or structs that must be provided at runtime.

In practice, developers define the data flow within C++ structs or Blueprint classes by using specific metadata or visibility settings. When you create a State Tree Task or Evaluator, the engine inspects the properties to determine how they can be linked:

- **Inputs:** In the State Tree editor, the Value UI for these properties is hidden, forcing the developer to bind them to a source. They are assumed to be valid by the time the task's `EnterState` is called.

- **Outputs:** Once a task performs a calculation (e.g., a "FindTarget" task identifying an enemy), it stores the result in an output property. This data then becomes available for any subsequent tasks or child states in the hierarchy.

- **Context:** These properties automatically attempt to bind to the Context Data defined in the Schema. For instance, if your Task has a context property of type `AActor*`, the State Tree will automatically pipe in the owner of the tree without any manual wiring.

One of the most powerful features for high-performance systems is the Property Reference (`FStateTreePropertyRef`). In many state machines, data must be copied from the source into the task's local instance, modified, and then copied back out. When scaled to thousands of entities, this constant cycle becomes a significant CPU bottleneck. State Tree property references allow a binding to return a direct pointer to the original data source rather than a copy. It is important to note that this feature is exclusively available via C++. By using the `FStateTreePropertyRef<T>` type in a Task or Evaluator's definition, developers can bypass the standard property copying mechanism.

5.2 Mass and State Tree

5.2.1 Mass State Tree Schema

In a standard Unreal Engine environment, state logic is typically bound to an `AActor` via a component. However, the Mass Framework necessitates a complete decoupling of logic from the `UObject` lifecycle. This is achieved through the `UMassStateTreeSchema`.

The most fundamental shift when moving to Mass is the abandonment of `UStateTreeTaskBlueprintBase` in favor of `FMassStateTreeTaskBase` and `FMassStateTreeEvaluatorBase`. Because Mass aims for high entity density, it cannot afford the memory overhead or the pointer-chasing inherent in ticking thousands of `UObject` instances. Instead, logic is encapsulated in `USTRUCTs`. These structs are not ticked by the engine's standard actor-ticking loop; rather, they are executed by the `UMassStateTreeProcessor`. This processor utilizes a `FMassStateTreeExecutionContext`, which acts as a lightweight bridge, providing the Task with access to the entity's fragments and the global subsystems without the need for a persistent actor owner.

In a Mass context, the `Tick` function of a task is effectively a data-transformation step. It receives the execution context, performs a calculation based on the provided fragment data, and returns a status. This is executed within the Mass Task Graph, allowing for massive parallelization that traditional state machines cannot achieve.

The Tick function

In a standard `UStateTreeComponent`, the tree is ticked every frame by the actor. In Mass, the `UMassStateTreeProcessor` is responsible for executing the logic, and its efficiency comes from not ticking every entity every frame.

Instead of how would you normally expect it to work, the `Tick` function in a Mass Task acts as a response to a wake-up call: when a task returns `EStateTreeRunStatus::Running`, the entity is essentially frozen in that state.

The processor will stop looking at that specific entity in subsequent frames to save CPU cycles. The entity only wakes up when a signal is sent to it via the `UMassSignalSubsystem`. This could be a Delay Signal (e.g., set by the task itself to wake up in 2 seconds) or an external signal (like a "Target Perceived" event). When the signal is received, the `UMassStateTreeProcessor` marks that entity as dirty, gathers it into a chunk with other signaled entities, and then calls the Tick() function. To learn more about this behavior, inspect the file `MassStateTreeProcessors.cpp`.

The following implementation exemplifies how this asynchronous communication works. This processor performs spatial queries and only dispatches the `UE::Mass::Signals::StateTreeActivate` signal when a specific gameplay threshold is met.

```
1  // Inside the Execute function of a Mass Processor
2  EntityView.ForEachEntityChunk([&](FMassExecutionContext&
       Context)
3  {
4       const int32 NumEntities = Context.GetNumEntities();
5       const TArrayView<FNPCEntityStateFragment> StateList =
       Context.GetMutableFragmentView<FNPCEntityStateFragment>();
6       const TArrayView<FTransformFragment> TransformList =
       Context.GetFragmentView<FTransformFragment>();
7
8       for (int32 i = 0; i < NumEntities; ++i)
9       {
10           const float DistanceSqr = FVector::DistSquared(
       TransformList[i].GetTransform().GetLocation(),
       PlayerLocation);
11
12           if (DistanceSqr < 16900.f)
13           {
14               if (StateList[i].State != ENPCStates::Attacking)
15               {
16                   StateList[i].State = ENPCStates::Attacking;
17                   SignalSubsystem.SignalEntity(UE::Mass::Signals
       ::StateTreeActivate, Context.GetEntity(i));
18               }
19           }
20           else if (DistanceSqr > 1000000.f)
21           {
22               if (StateList[i].State != ENPCStates::Roaming)
23               {
24                   StateList[i].State = ENPCStates::Roaming;
25                   SignalSubsystem.SignalEntity(UE::Mass::Signals
       ::StateTreeActivate, Context.GetEntity(i));
26               }
27           }
```

```
28       }
29  });
```

5.2.2 Data Handling and Memory Locality

As previously stated, Mass performance relies on Cache Coherency, meaning data must be stored contiguously. To maintain this, State Trees in Mass distinguish strictly between where data lives and how it is accessed.

Understanding the residency of data is critical for a stable simulation. In the context of Mass, Fragments should be used to store data that must survive across different states. Instance data, defined via `FInstanceDataType` struct, lives within the `FMassStateTreeInstanceFragment`. It is used for variables that only matter for the duration of a specific state.

A practical example of this distinction is found in the FMassClaimSmartObject-Task, which uses the following instance data:

```
1  USTRUCT()
2  struct FMassClaimSmartObjectTaskInstanceData
3  {
4    GENERATED_BODY()
5
6    /** Result of the candidates search request (Input) */
7    UPROPERTY(VisibleAnywhere, Category = Input, meta = (
      BaseStruct = "/Script/MassSmartObjects.
      MassSmartObjectCandidateSlots"))
8    FStateTreeStructRef CandidateSlots;
9
10   UPROPERTY(VisibleAnywhere, Category = Output)
11   FSmartObjectClaimHandle ClaimedSlot;
12 };
```

To bridge the gap between the State Tree and the Mass memory layout, we use `TStateTreeExternalDataHandle<T>`. In practice, this follows a linking pattern: during the `Link` phase (initialization), the task requests access to specific fragments or subsystems. The State Tree Linker resolves these requests into handles. At runtime, the task uses these handles to retrieve direct pointers to the fragment data within the current chunk, ensuring zero-copy overhead.

A clear implementation of this pattern can be found in the `FMassClaimSmartObjectTask`:

```
1  //MassClaimSmartObjectTask.h
2  USTRUCT(meta = (DisplayName = "Claim SmartObject"))
3  struct FMassClaimSmartObjectTask : public
      FMassStateTreeTaskBase
4  {
5    GENERATED_BODY()
6
7    using FInstanceDataType =
      FMassClaimSmartObjectTaskInstanceData;
8
9    MASSAIBEHAVIOR_API FMassClaimSmartObjectTask();
10
11 protected:
12     MASSAIBEHAVIOR_API virtual bool Link(FStateTreeLinker&
      Linker) override;
13     //...
14
15     //External data handles definition
16   TStateTreeExternalDataHandle<FMassSmartObjectUserFragment>
      SmartObjectUserHandle;
17   TStateTreeExternalDataHandle<USmartObjectSubsystem>
      SmartObjectSubsystemHandle;
18   TStateTreeExternalDataHandle<UMassSignalSubsystem>
      MassSignalSubsystemHandle;
19 };
20
21
22 //MassClaimSmartObjectTask.cpp
23 bool FMassClaimSmartObjectTask::Link(FStateTreeLinker& Linker)
24 {
25   Linker.LinkExternalData(SmartObjectUserHandle);
26   Linker.LinkExternalData(SmartObjectSubsystemHandle);
27   Linker.LinkExternalData(MassSignalSubsystemHandle);
28
29   return true;
30 }
```

By requiring tasks to explicitly link their data, the Mass State Tree achieves two major goals:

- **Performance:** It bypasses the standard Unreal `GetComponentByClass` or `GetSubsystem` calls during the simulation loop, which are too slow for thousands of entities.

- **Safety:** If a required fragment (like `FMassSmartObjectUserFragment`) is

missing from the entity's archetype, the Link phase will fail early, preventing null-pointer crashes during the high-speed simulation tick.

5.2.3 Transition Handling and Logic Flow

Transitions in Mass are outcome-driven. Because the `UMassStateTreeProcessor` only updates entities when necessary, the status returned by a task's `Tick` or `EnterState` is the primary driver of the simulation's flow.

A common pitfall is attempting to perform complex polling logic inside a task. In Mass, if a task is waiting for something to happen (like reaching a destination), it should return `EStateTreeRunStatus::Running`. If the goal is reached, it returns Succeeded. This immediate return signals the State Tree to evaluate transitions instantly, moving the entity to the next state without waiting for the next frame's tick. This ensures that the entity's decision logic is as reactive as possible while keeping the action logic separate.

In fact, A critical design pattern in Mass-integrated State Trees is the strict separation between Decision Logic and Execution Logic. Tasks should be treated as lightweight state mutators: their role is to reconfigure fragment values or toggle gameplay tags to signal a change in intent. The computational heavy lifting, such as pathfollowing, avoidance, or complex sensory filtering, should be delegated to specialized Mass Processors. This ensures that the State Tree remains a reactive brain that only consumes CPU cycles when a high-level transition occurs, while the Mass Task Graph provides the muscle for continuous, high-performance simulation.

Chapter 6

Navigation in the Mass Framework

Navigating a massive number of agents through a complex 3D environment is traditionally one of the most CPU-intensive tasks in game development. In Unreal Engine's standard Object-Oriented paradigm, navigation relies heavily on `AController` and `UPathFollowingComponent`. While robust, this approach requires constant pointer-chasing and individual tick updates, creating a hard ceiling on the number of autonomous agents the engine can simulate simultaneously.

Within the Mass framework, navigation is completely reimagined to fit the Entity Component System (ECS) architecture. This chapter explores the navigation paradigms available in Mass, detailing the data-oriented implementations of Navigation Mesh (NavMesh) and ZoneGraph, demonstrating how to build hybrid navigation systems, and examining the low-level steering and avoidance processors.

6.1 Overview

Mass does not entirely reinvent the mathematical wheel of pathfinding. Instead, it optimizes how path requests are batched, stored, and executed. The MassNavigation and MassAI plugins provide the structural logic for moving entities, decoupling the intent to move from the physical execution of movement.

Regardless of the chosen pathfinding method, the ultimate goal of the Mass navigation pipeline is to populate and continuously update an entity's `FMassMoveTargetFragment`. This universal fragment acts as the movement intent, storing the immediate destination vector, desired speed, and distance to the goal. Processors downstream read this fragment without needing to know whether the target was generated by a complex NavMesh query or a simple spline calculation.

6.2 NavMesh Navigation

For open-world environments, unstructured terrain, and dynamic spatial queries, Mass provides a data-oriented wrapper around the standard Unreal Engine Recast Navigation Mesh. Because querying the underlying NavMesh octree is a heavy, object-oriented operation, it cannot be safely executed within a high-speed, parallelized processor loop. To bridge this architectural divide, Mass strictly separates the initial path calculation from the continuous path execution, offloading the heavy synchronous queries to the AI's StateTree while the ECS processors handle the frame-by-frame locomotion.

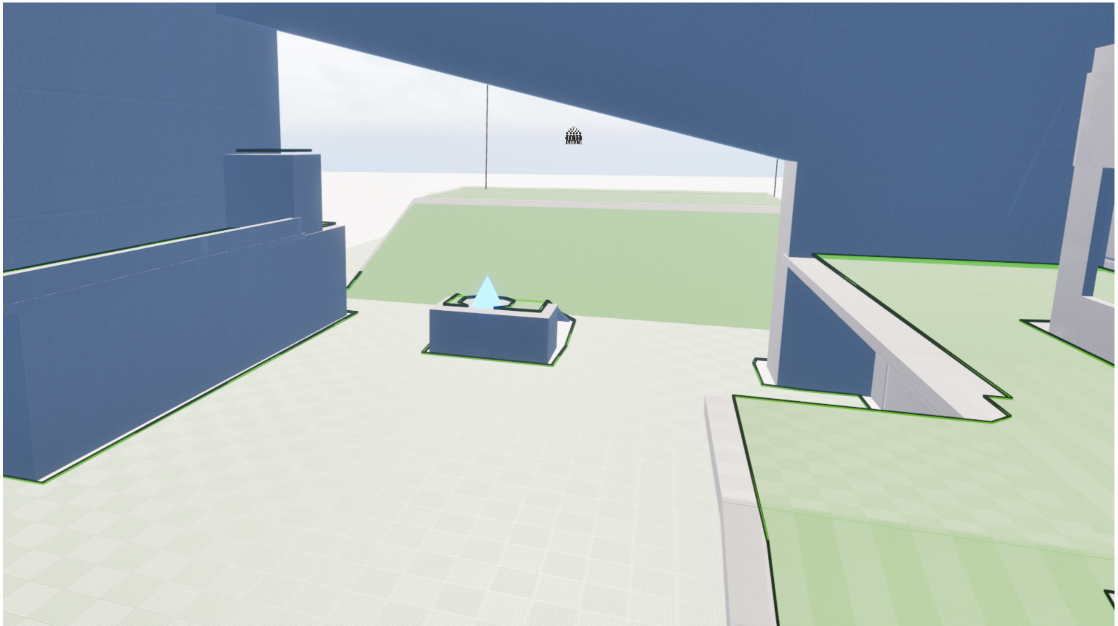

Figure 6.1: In green the active NavMesh created from the level geometry.

6.2.1 The NavMesh Trait

To utilize NavMesh routing, an entity's archetype must be configured with a specific movement trait via the editor, such as `UMassNavMeshNavigationTrait`. This trait acts as the structural blueprint, ensuring the entity is initialized with the required memory containers. Most notably, it appends two critical fragments to manage the path data. The first is the `FMassNavMeshCachedPathFragment`, which holds the long-term references to the standard Unreal navigation path and corridor. The second is the `FMassNavMeshShortPathFragment`, a lightweight data structure designed to store a static array of up to eight immediate spatial waypoints and

tangent directions. By splitting the data, the architecture allows high-frequency processors to calculate immediate movement along the short path without parsing the entire macroscopic navigation corridor every frame.

6.2.2 The StateTree Task

To realize this architectural separation in practice, the entity's StateTree relies on a specific execution node: `FMassNavMeshPathFollowTask`. Rather than offloading the pathfinding to a processor, this task handles the entire lifecycle of the NavMesh request and data translation through three distinct phases:

- **Synchronous Query:** Upon entering the state, the task interfaces directly with `UNavigationSystemV1`. It performs a synchronous pathfinding query using the agent's specific radius and location, securing the navigational data immediately rather than waiting for an asynchronous callback.
- **Data Translation and Caching:** Once a successful path is returned, the task first stores the heavy `FNavPathSharedPtr` and builds a detailed navigation corridor within the long-term `FMassNavMeshCachedPathFragment`. Second, it extracts only the immediate upcoming waypoints and populates the lightweight `FMassNavMeshShortPathFragment`. Finally, it calculates the desired speed and pushes it to the `FMassMoveTargetFragment`.
- **Wait State and Buffer Refresh:** The task returns a `Running` status, effectively putting the entity's StateTree to sleep and halting further behavioral evaluation. Rather than polling the path progress every frame, the task relies on Mass's signal subsystem. The continuous monitoring is delegated to the `UMassNavMeshPathFollowProcessor`. When this processor exhausts the short path buffer, it broadcasts a completion signal (`FollowPointPathDone`). This signal awakens the StateTree, triggering the task's Tick function. If the overall route is incomplete, the task calls `UpdateShortPath` to fetch the next chunk of waypoints from the cached corridor, refilling the buffer and returning the StateTree to sleep until the destination is finally reached.

6.2.3 The Path Following Processor

Once the path data is securely written to the chunk memory, the continuous logic of traversing the corridor is handed off to the `UMassNavMeshPathFollowProcessor`. Concurrently, while the StateTree task sleeps and manages the high-level waypoint buffer, this processor takes over the low-level execution. It does not apply physical velocity directly but instead, it acts as a geometric interpolator.

During its parallel execution phase, the processor reads the `FMassNavMeshShortPathFragment` and calculates the entity's true progress by projecting its physical location onto the closest path segment. Using the entity's desired speed and the frame's delta time, it calculates a look-ahead target along the immediate path corridor. It then pushes this exact spatial coordinate and tangent vector into the `FMassMoveTargetFragment` frame-by-frame for the steering processors to follow.

By continuously dragging this target point just ahead of the entity, the processor decouples the complexity of the NavMesh corridor from the actual physical locomotion. Furthermore, when the processor detects that the entity has reached the final waypoint of the short path within a predefined tolerance, it integrates with the `UMassSignalSubsystem` to broadcast a completion signal (e.g., `UE::Mass::Signals::FollowPointPathDone`). As established, this signal wakes the sleeping StateTree task, prompting the entity to either refresh the buffer or evaluate its next behavioral state without requiring expensive, per-frame polling.

6.3 ZoneGraph

While NavMesh is highly versatile for open-world traversal, computing routes across thousands of interconnected polygons is computationally expensive. For simulations like the City Sample, where thousands of pedestrians and vehicles follow strict traffic laws, Epic Games introduced ZoneGraph.

ZoneGraph is a spline-based routing system completely detached from the NavMesh. Instead of a walkable surface area, ZoneGraph defines explicit lanes, intersections, and connections. It operates on a conceptual level similar to a rail network, built upon four foundational structural elements that we will now analyse.

6.3.1 Zones

Zones are the high-level containers that define the overall trajectory of a route. In the Unreal Editor, a Zone is the primary authoring unit, typically represented by a spline curve. However, it does not exist as an Actor at runtime. During the environment baking process, these splines are compiled into flattened data structures (such as `FZoneGraphStorage`). A Zone does not dictate specific agent movement. Instead, it provides the geometric foundation for grouping multiple parallel tracks. This grouping helps Mass spatial partitioning systems efficiently manage entity Level of Detail (LOD) logic based on the agent's current sector.

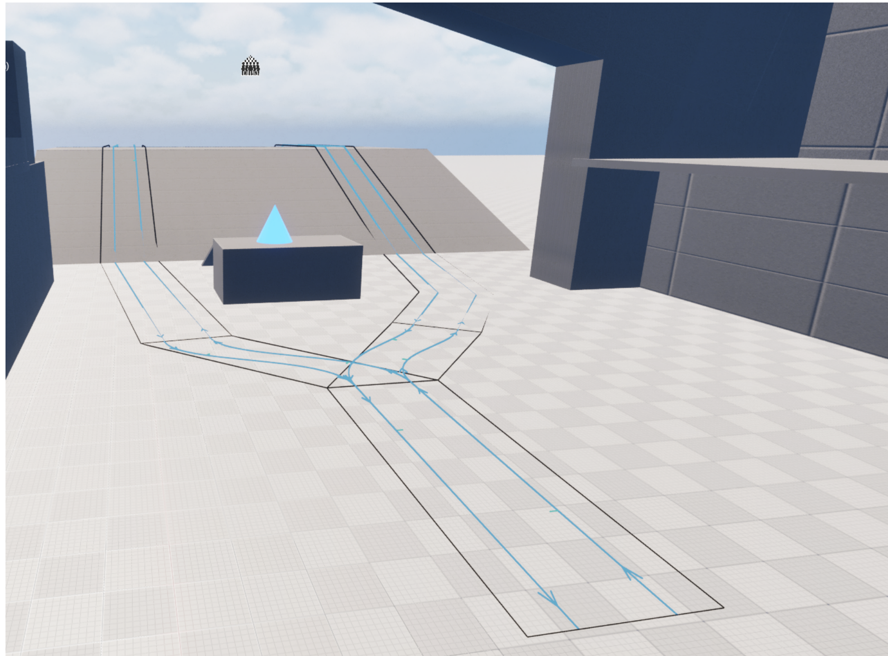

Figure 6.2: An example of ZoneGraph lanes, with an intersection that merges them.

6.3.2 Lanes

Lanes are the specific, functional tracks within a Zone that dictate the exact path an entity follows. A single Zone can contain multiple parallel lanes, each with distinct semantic profiles, tags, and traversal rules (e.g., separating a forward vehicle lane from a pedestrian sidewalk). At runtime, an entity's position on a lane is tracked using lightweight data fragments, specifically the `FMassZoneGraphLaneLocationFragment`. This fragment stores the current lane index and the distance along its curve. By reducing movement to a 1D distance value along a track, Mass processors can advance the entity's position through simple scalar addition, bypassing the need for continuous 3D surface projections or complex path smoothing.

6.3.3 Intersections

Intersections act as connective nodes that link individual lanes together to form a cohesive network. Rather than relying on real-time spatial queries to discover the next path, intersections provide entities with a predefined set of linked lane indices. They act as logical checkpoints where the Mass framework can handle branching routing (like turning at a crossroad) and StateTree tasks can evaluate conditional

flow (like yielding at a red light) through highly efficient array lookups.

6.3.4 Connections

Connections are the explicit data links that define how individual lanes relate to one another. While intersections serve as the major routing hubs, connections represent the directed, point-to-point transitions, such as moving forward into a new spline segment or shifting laterally to an adjacent lane. Within the ECS architecture, these relationships are stored as simple arrays of linked lane indices (e.g., `FZoneGraphLinkedLane`). By pre-calculating these valid transitions during the environment baking process, the framework allows entities to execute complex maneuvers, like overtaking a slower vehicle or merging into traffic, through instantaneous memory lookups. This entirely eliminates the need for expensive physics sweeps or dynamic overlap checks when an agent needs to change its current track.

A critical advantage of this entire system lies in its Data-Oriented Design. Unlike standard Unreal Engine splines that are typically constructed from multiple object-oriented Actors, ZoneGraph data, including all lanes and intersection links, is baked into flat, contiguous memory arrays. This structural alignment allows the navigation data to be queried in constant time, making it exceptionally fast for ECS processors to read during parallel execution. Ultimately, because the path is explicitly constrained by the lane's mathematical geometry and intersection rules, the complex burden of spatial pathfinding is reduced to a highly optimized, one-dimensional graph-traversal problem.

With this structural foundation established, we can examine how the Mass framework executes agent navigation across this network.

6.3.5 The ZoneGraph Trait

To navigate this structured network, an entity must be configured with a specialized trait, such as `UMassZoneGraphNavigationTrait`. This trait strips away the heavy polygon-tracking data used in open-world navigation and injects fragments optimized for spline traversal. The two most critical are the `FMassZoneGraphLaneLocationFragment`, which stores the entity's current `FZoneGraphLaneHandle` and its exact scalar distance along that curve, and the `FMassZoneGraphShortPathFragment`, which acts as the memory buffer holding the immediate geometric data for the lane the entity is currently traversing.

6.3.6 The StateTree Task

The physical progression of an entity along a lane is governed by the StateTree via an execution node: `FMassZoneGraphPathFollowTask`. Unlike NavMesh pathfinding, which calculates a full route to a distant destination, this specific ZoneGraph task is highly localized, operating purely on a lane-by-lane basis to maximize performance. The task manages the movement request through a strict lifecycle:

- **Localized Intent Verification:** Upon entering the state, the task does not perform a city-wide A* search. Instead, it receives a `FMassZoneGraphTargetLocation`. It first verifies that the target lane matches the entity's current `FMassZoneGraphLaneLocationFragment`. If they do not match, the task immediately fails, ensuring entities do not attempt to traverse unconnected space.
- **Data Translation and Caching:** Once verified, the task builds a `FZoneGraphShortPathRequest`. It extracts the target distance, anticipated exit links, and end-of-path intent, using the `UZoneGraphSubsystem` to translate this intent into actionable geometry. It populates the entity's `FMassZoneGraphShortPathFragment` and calculates the desired speed based on the lane's inherent speed limits, pushing this data into the `FMassMoveTargetFragment`.
- **Event-Driven Buffer Refresh:** The task returns a `Running` status, effectively putting the StateTree to sleep. It relies entirely on the underlying processors to navigate the spline. When the processor exhausts the immediate path buffer, it broadcasts a completion signal. This awakens the StateTree, triggering the task's `Tick` function to refresh the short path buffer for the next segment of the lane.

6.3.7 The Path Following Processor

With the StateTree asleep and the lane data cached, the `UMassZoneGraphPathFollowProcessor` assumes control of the continuous execution. Because the entity is locked to a predefined track, this processor completely bypasses the complex 3D geometric projections required by NavMesh.

During its parallel execution phase, the processor reads the `FMassZoneGraphLaneLocationFragment` and simply advances the entity's 1D scalar distance along the curve based on its speed and the frame's delta time. It evaluates the spline's geometry at this new distance to extract the spatial coordinate and forward tangent, pushing them directly into the `FMassMoveTargetFragment` for the steering processors to utilize.

Crucially, this processor is designed to handle intersection transitions seamlessly within the high-speed ECS loop. When an entity reaches the end of its current lane, the processor checks the `FMassZoneGraphShortPathFragment`. If a valid next lane has already been assigned by the StateTree, the processor automatically snaps the entity's location fragment to the new `FZoneGraphLaneHandle`, resets its distance, and fires a `CurrentLaneChanged` signal without ever pausing movement. It is only when the entity reaches the absolute final destination of its queued path that the processor integrates with the `UMassSignalSubsystem` to broadcast a `FollowPointPathDone` signal. Just as in the NavMesh implementation, this final signal awakens the sleeping StateTree task to evaluate its next major behavioral state.

6.4 Additional Movement Traits and Processors

Once the StateTree and navigation subsystems have populated the `FMassMoveTargetFragment`, the actual physical translation of the entity is handled by a suite of Movement Processors, configured via Entity Traits.

6.4.1 Steering

To enable an entity to physically follow its navigational intent, it must be assigned the `UMassSteeringTrait`. This trait introduces the fragments required to calculate reactive velocities, which are subsequently processed by the steering systems.

The `UMassSteerToMoveTargetProcessor` processor reads the `FMassMoveTargetFragment` and bridges the gap between navigational intent and physical simulation. It calculates the optimal steering direction and speed required to reach the look-ahead target. Crucially, it does not modify the entity's velocity directly. Instead, it acts as a proportional controller: it compares its newly calculated steering velocity against the entity's current trajectory and outputs the corrective vector into the `FMassForceFragment`, ensuring the entity respects its maximum speed and acceleration parameters while turning.

6.4.2 Smooth Orientation

Snapping an entity's rotation instantly to match its movement direction looks unnatural for visible characters. Rather than storing rotational velocity data, the `UMassSmoothOrientationProcessor` processor calculates a desired heading by dynamically blending the entity's physical velocity vector with the path's geometric forward tangent. For high-resolution entities, it applies an exponential smoothing mathematical function to interpolate the entity's current transform towards this blended heading over time, providing fluid, natural turns without the overhead of

an Animation Blueprint. Crucially, to maximize ECS performance, the processor executes a separate query for low-resolution entities (those marked with an OffLOD tag) and instantly snaps their rotation to the target, bypassing the smoothing calculations entirely when the agent is too far away for the player to notice.

In order to enable this processor to work, we must add the `UMassSmoothOrientationTrait` to the entity configuration.

6.4.3 Avoidance

Calculating a navigational intent is only half the problem; entities must also actively avoid static boundaries and dynamic agents. In the Mass framework, avoidance is applied as an adjustment phase. Executing strictly after the steering calculations, the avoidance processors read the environment and neighboring entities, generating repulsive vectors that are added directly to the entity's `FMassForceFragment`.

To handle the vastly different dynamics of active traversal versus idle crowds, Mass splits this logic into two specialized processors.

Moving Avoidance

When an entity is actively traversing a path, the `UMassMovingAvoidanceProcessor` applies a force-based model to steer clear of both static geometry and dynamic agents. To bypass the $O(n^2)$ performance bottleneck typical of dense crowd simulations, the processor queries a spatial hash grid to evaluate only a strict maximum of nearby neighbors, generating predictive repulsive forces based on their relative trajectories and time-to-impact.

Standing Avoidance

When agents reach their destination and transition into a standing state, applying rigid repulsive forces often leads to severe visual jittering or continuous vibrating as dense crowds push against one another in a confined space.

To solve this, Mass utilizes the `UMassStandingAvoidanceProcessor`, which introduces a "Ghost" mechanic. Instead of applying avoidance forces directly to the physical agent, the processor simulates an invisible `FMassGhostLocationFragment`. The repulsive forces from nearby standing agents or moving obstacles are applied strictly to this Ghost location, shoving it around the environment. The underlying steering processor then gently pulls the physical agent toward its displaced Ghost.

Chapter 7

Conclusions

Unreal Engine's Mass framework represents a definitive leap forward for large-scale simulations. As demonstrated throughout this thesis, by fundamentally restructuring how data is stored and processed, shifting from an Actor-centric Object-Oriented paradigm to a Data-Oriented Design, Mass effectively shatters the performance ceilings that have historically limited crowd density and agent autonomy. It stands as the most viable, native tool for creating massive agent simulations within the Unreal Engine ecosystem.

However, viability in theory does not seamlessly translate to readiness in production. While the foundational MassEntity architecture is robust and integrated into the core engine, utilizing the framework to build complex, interactive worlds currently requires a massive software engineering investment to overcome significant integration, documentation, and scalability hurdles.

7.1 Production Risks and Tooling Deficits

At its current stage of maturity, one of the most pressing limitations of the Mass framework is its lack of accessible Editor integration. Traditional game development workflows heavily rely on empowering non-technical staff, such as game designers, level artists, and animators, to iterate on entity behavior via intuitive interfaces like Blueprints and Actor Details panels. Mass, by contrast, is distinctly code-heavy and structurally abstract. To implement Mass in a production environment, a studio must dedicate consistent engineering time to develop custom tooling, bridging the gap between the C++ data-oriented backend and the editor-facing front end. Without this custom infrastructure, designer iteration is heavily bottlenecked.

Furthermore, the higher-level plugins that make Mass usable for gameplay, namely MassGameplay, MassAI, and MassCrowd, remain firmly labeled as Experimental features by Epic Games as of Unreal 5.7. For a commercial studio, adopting

experimental technology for core gameplay loops constitutes a severe production risk. The APIs governing these systems are subject to undocumented, breaking changes in future engine updates, potentially requiring costly and time-consuming code refactors mid-development.

7.2 Scaling Bottlenecks: Navigation and Avoidance

While Mass excels at iterating over contiguous memory, integrating it with standard Unreal Engine systems can reintroduce the very bottlenecks the framework was designed to bypass. This is most evident in complex spatial pathfinding.

If a project aims to simulate tens of thousands of free-roaming agents, the default NavMesh integration simply does not scale. The traditional Recast Navigation Mesh is an inherently Object-Oriented construct. Querying a path requires synchronous A* searches across complex polygonal networks, resulting in heavy pointer-chasing and inevitable cache misses. While the Mass State Tree attempts to hide this latency by caching short paths and putting entities to sleep, the initial generation of the path remains a massive CPU burden. When thousands of agents request NavMesh paths simultaneously, the Game Thread will stall.

Similarly, while the built-in Mass avoidance systems utilize spatial hashing to improve performance, calculating dynamic repulsive forces across a massive, dense crowd remains a computationally heavy operation.

To truly reach the upper limits of entity density, a development team must either:

- **Develop Custom Navigation Solutions:** Completely replace the NavMesh dependency with a custom, data-oriented spatial grid or flow-field navigation system tailored specifically for ECS processing.
- **Implement Aggressive Throttling:** If sticking to the built-in NavMesh and avoidance, engineers must design smart, localized time-slicing systems. This involves strictly limiting the number of concurrent pathfinding requests per frame and heavily relying on the UMassLODSubsystem to disable avoidance and navigation queries for entities outside the player's immediate proximity.

7.3 The Paradigm Shift

Beyond the technical challenges, integrating Mass into a production pipeline involves a significant shift in development methodology. Transitioning from familiar Object-Oriented Programming (OOP) principles to Data-Oriented Design (DOD) presents

a notable learning curve. Developers accustomed to relying on inheritance and standard object lifecycles will need to adapt their workflows to prioritize memory layouts, cache coherency, and data transformations. Onboarding a team to this new architecture requires time and careful code-review practices to ensure that traditional OOP patterns, like monolithic data structures or heavy pointer use, do not inadvertently compromise the performance benefits of the ECS environment.

7.4 Final Outlook

Despite these profound implementation challenges, Unreal Engine Mass remains a highly promising technology. For studios with the engineering bandwidth to build custom tooling and navigate its experimental edges, it offers unparalleled performance. Furthermore, the principles and infrastructure of MassEntity are not limited strictly to crowd simulations; the framework can be leveraged as a generic optimization tool for any system requiring the high-speed processing of massive datasets, such as large-scale traffic management or backend systemic calculations. As the framework matures and Epic Games standardizes the APIs in future engine releases, Mass is poised to become the definitive standard for high-performance simulation in interactive media.

Bibliography

References for Chapter 2

[1] Davide Amato. *A very simple Entity-Component System implementation in C++*. 2025. URL: https://www.youtube.com/watch?v=fAvpt6Zj3s4.

[2] Elizabeth Baumel. *Understanding data-oriented design for entity component systems - Unity at GDC 2019*. 2019. URL: https://www.youtube.com/watch?v=0_Byw9UMn9g.

[3] Richard Johnson. *Entity-Component System Design Patterns: Definitive Reference for Developers and Engineers*. HiTeX Press, 2025.

[4] Mark Jordan. *Entities, components and systems*. Medium. Nov. 20, 2018. URL: https://medium.com/ingeniouslysimple/entities-components-and-systems-89c31464240d.

[5] Sander Mertens. *Entity Component System FAQ*. 2018. URL: https://github.com/SanderMertens/ecs-faq.

[6] Damien Morello. *Tapping the Entity Component System for Cities: Skylines II | Unite 2024*. 2024. URL: https://www.youtube.com/watch?v=nEkIyWhvq3o.

[7] Vittorio Romeo. «Analysis of entity encoding techniques, design and implementation of a multithreaded compile-time Entity-Component-System C++14 library». Bachelor's Thesis. Messina: University of Messina, 2016. URL: https://www.researchgate.net/publication/305730566.

References for Chapter 3

[8] Epic Games. *City Sample*. 2021. URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/city-sample-project-unreal-engine-demonstration.

[9] Epic Games. *MassEntity Overview*. 2021. URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/overview-of-mass-entity-in-unreal-engine.

[10] Epic Games. *MassGameplay Overview*. 2025. URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/overview-of-mass-gameplay-in-unreal-engine.

[11] James Keeling. *Your First 60 Minutes With Mass*. Epic Games. 2025. URL: https://dev.epicgames.com/community/learning/tutorials/6vG6/unreal-engine-your-first-60-minutes-with-mass.

[12] Karl Mavko and Alvaro Jover. *Community Mass Sample*. 2025. URL: https://github.com/Megafunk/MassSample.

[13] Vladimir Nepor. *Mass Framework for crowds and traffic simulation in Unreal Engine*. 2023. URL: https://vrealmatic.com/unreal-engine/mass.

[14] X157. *Mass Entity*. 2025. URL: https://x157.github.io/UE5/Mass/.

References for Chapter 4

[15] X157. *Mass LOD*. 2025. URL: https://x157.github.io/UE5/Mass/LOD.

References for Chapter 5

[16] Epic Games. *StateTree Overview*. 2025. URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/overview-of-state-tree-in-unreal-engine.

[17] hzFishy. *State Tree*. 2025. URL: https://notes.hzfishy.fr/Unreal-Engine/AI/State-Tree/State-Tree.

[18] James Keeling. *Your First 60 Minutes With Mass*. Epic Games. 2024. URL: https://dev.epicgames.com/community/learning/tutorials/lwnR/unreal-engine-your-first-60-minutes-with-statetree.

[19] Mikko Mononen. *State Tree Deep Dive | Unreal Fest 2024*. 2024. URL: https://www.youtube.com/watch?v=YEmq4kcblj4.

References for Chapter 6

[20] Epic Games. *Basic Navigation*. 2025. URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/basic-navigation-in-unreal-engine.

[21] Epic Games. *Mass Avoidance Overview*. 2025. URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/mass-avoidance-overview-in-unreal-engine.

[22] James Keeling. *ZoneGraph Quick Start Guide*. Epic Games. 2022. URL: https://dev.epicgames.com/community/learning/tutorials/qz6r/unreal-engine-zonegraph-quick-start-guide.

[23] Marius Muja. *FLANN - Fast Library for Approximate Nearest Neighbors*. 2012. URL: https://github.com/flann-lib/flann.

[24] Gianmarco Picarella, Joel Brieger, and Nikos Giakoumoglou. *Crowd Simulation Project Report*. 2024. URL: https://github.com/gianmarcopicarella/crowd-simulation/blob/main/Assignment_Deliverables/Crowd_Simulation_Game_Report.pdf.

[25] X157. *Mass Navigation*. 2025. URL: https://x157.github.io/UE5/Mass/Navigation.