Politecnico di Torino

Politecnico di Torino

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

A.a. 2025/2026

Graduation Session March 2026

Managing large hordes of NPCs in Unreal Engine 5

An analysis of Mass ECS system

Supervisors:

Francesco Strada

Corrado Raffaelli

Candidate:

Martina Plumari

Acknowledgements

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Tristique senectus et netus et malesuada fames ac turpis. Pretium nibh ipsum consequat nisl vel pretium lectus quam. Urna molestie at elementum eu facilisis sed odio morbi quis. Sed egestas egestas fringilla phasellus faucibus scelerisque eleifend. At in tellus integer feugiat. Mauris rhoncus aenean vel elit scelerisque mauris pellentesque pulvinar. Commodo sed egestas egestas fringilla. Nunc lobortis mattis aliquam faucibus purus in massa. Facilisis magna etiam tempor orci eu lobortis elementum nibh. Elementum curabitur vitae nunc sed velit dignissim. Neque volutpat ac tincidunt vitae. Massa id neque aliquam vestibulum morbi blandit cursus risus at. Porta non pulvinar neque laoreet suspendisse interdum consectetur. Turpis in eu mi bibendum. Ut tristique et egestas quis ipsum suspendisse. Integer quis auctor elit sed vulputate mi sit amet. Viverra nam libero justo laoreet sit amet cursus.

Table of Contents

List of Figures

Chapter 1

Introduzione

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Porttitor eget dolor morbi non arcu risus quis varius. Libero id faucibus nisl tincidunt. Neque laoreet suspendisse interdum consectetur libero id faucibus nisl tincidunt. Scelerisque in dictum non consectetur a erat. Leo a diam sollicitudin tempor id eu. Sodales ut eu sem integer vitae justo eget magna fermentum. A cras semper auctor neque vitae. Cursus euismod quis viverra nibh cras pulvinar. Mi tempus imperdiet nulla malesuada pellentesque elit eget gravida cum. Dictum at tempor commodo ullamcorper a lacus vestibulum sed. Ultricies mi eget mauris pharetra. In pellentesque massa placerat duis ultricies lacus. Fringilla phasellus faucibus scelerisque eleifend donec pretium vulputate.

Aliquam sem fringilla ut morbi tincidunt augue interdum. Risus at ultrices mi tempus imperdiet nulla malesuada pellentesque. Semper quis lectus nulla at volutpat. Nullam non nisi est sit amet facilisis. Eget velit aliquet sagittis id consectetur purus ut faucibus pulvinar. Ultricies tristique nulla aliquet enim tortor at auctor urna nunc. Pharetra diam sit amet nisl suscipit adipiscing bibendum est ultricies. Amet facilisis magna etiam tempor. Nunc non blandit massa enim nec dui nunc mattis. At ultrices mi tempus imperdiet nulla malesuada pellentesque. Cursus metus aliquam eleifend mi in nulla posuere. In eu mi bibendum neque egestas congue quisque. Augue eget arcu dictum varius duis at consectetur.

1.1 Goal

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Tristique senectus et netus et malesuada fames ac turpis. Pretium nibh ipsum consequat nisl vel pretium lectus quam. Urna molestie at elementum eu facilisis sed odio morbi quis. Sed egestas egestas fringilla

phasellus faucibus scelerisque eleifend. At in tellus integer feugiat. Mauris rhoncus aenean vel elit scelerisque mauris pellentesque pulvinar. Commodo sed egestas egestas fringilla. Nunc lobortis mattis aliquam faucibus purus in massa. Facilisis magna etiam tempor orci eu lobortis elementum nibh. Elementum curabitur vitae nunc sed velit dignissim. Neque volutpat ac tincidunt vitae. Massa id neque aliquam vestibulum morbi blandit cursus risus at. Porta non pulvinar neque laoreet suspendisse interdum consectetur. Turpis in eu mi bibendum. Ut tristique et egestas quis ipsum suspendisse. Integer quis auctor elit sed vulputate mi sit amet. Viverra nam libero justo laoreet sit amet cursus.

1.2 Struttura della tesi

Lorem ipsum dolor sit amet:

- onsectetur adipiscing elit,
- sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.
- Tristique senectus et netus et malesuada fames ac turpis.

Chapter 2

Entity Component System

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 analyses the Entity–Component–System architectural pattern and the reasons it can offer performance advantages over the traditional object-oriented paradigm. It then examins 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 delaay 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. On the other hand, 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.

2.2 The Three Pillars of ECS

As the name suggests, the ECS pattern consists of three main functional layers:

- Entity: it is a unique identifier serving as a handle that associates a set of components. It contains no intrinsic data or behavior. Its sole purpose is to provide a unique index to which stateful data can be mapped;
- Component: granular data container that encapsulate state but contain no logic. As "plain old data" structures, components are stored in dense arrays to optimize memory throughput;
- System: it represents the logic layer of the architecture. Systems are independent units that operate over filtered collections of components by means of declarative queries. Because systems operate on dense arrays with explicit data dependencies, they are inherently suited for multi-threaded execution, allowing the engine to distribute workloads across CPU cores with minimal synchronization overhead.

2.2.1 Trade-offs

While ECS provides the performance requisite for systems like UE Mass, it introduces specific architectural trade-offs:

- Architectural Complexity: initial planning requires a more rigorous design of component interfaces and system boundaries compared to the more intuitive Object based approach of OOD;
- Decoupling Overhead: the radical separation of data can lead to "component bloat" if the granularity is not managed carefully, potentially resulting in repetitive or trivial data structures.

- Mentality Shift: transitioning from an object-centric logic to Data-Oriented paradigm requires a fundamental shift in how developers conceptualize program flow and state management. This shift takes time, and the steep learning curve may not be strategically feasible for all production timelines or team structures.

Chapter 3

Ureal 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 additional features available through the addition of standalone plugins offerd 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. MassEntity was firstly introduced as a plugin, but has been moved into the core engine as of UE5.5 and is now deprecated as standalone.

In the following sections we analyse the data structures, memory organization as well as the logic execution of the Mass framework.

3.1.1 Data Structures: Fragments and Tags

MassEntity redefines the standard ECS Component into more specialized structures to optimize memory and filtering.

- Fragments: they are the primary data structure in MassEntity. A Fragment represents an atomic unit of data (e.g. Transform, Velocity, LOD index). In alignment with the ECS paradigm, Fragments are logic-less and exist only to store state.

- Shared Fragments:
- Chunk Fragments: unlike standard Fragments associated with a specific Entity, a ChunkFragment is associated with a Memory Chunk. These are used to store data shared by all entities within that chunk, such as communal Level of Detail (LOD) calculation, reducing redundant data processing.
- Tags: these are trivial Fragments that contain no data. Their presence or absence on an Entity serves as a high-performance filter for the queries.

3.1.2 Memory Organization: Archetypes and Chunks

The performance of MassEntity is largely a result of its Archetype-based storage system, which organizes data to maximize CPU cache efficiency.

- Entities and Composition: an Entity in Mass is a unique ID associated with a collection of Fragments. Mirroring the composition pattern, an Entity's makeup is dynamic and Fragments can be added or removed at runtime to change the Entity's behavior or state.
- Archetypes: when multiple Entities share an identical Fragment composition, they are grouped into an Archetype. If one Entity has '[Transform, Velocity]' and another has '[Transform, Velocity, Health]', they belong to two different Archetypes. This categorization allows the engine to predict exactly how data is laid out in memory.
- Memory Chunks: within an Archetype, Entities are stored in contiguous memory blocks called Chunks, optimized for current cache sizes. By grouping similarly composed entities together, Mass ensures that the CPU can retrieve and process batches of data in a single linear pass, significantly reducing the time spent waiting for RAM fetches.

3.1.3 Logic Execution: Processors and Queries

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

Processors are stateless classes that house the simulation logic. They are designed to be data-agnostic, meaning they do not operate on individual entities one by one, but rather on streams of data.

They use EntityQueries to define their data requirements (e.g. "Provide all entities that have a Velocity Fragment but do not have and Enemy Tag"). The query fetches the data in batches from the Chunks, allowing the Processor to execute

logic across thousands of entities with maximum efficiency and high potential for parallelization.

Accordingly to the memory organization articulated in the previous section, processors cover a chunk of entities in an archetype. The execution flow of Mass Processors follows this steps:

1. Each processor starts by configuring an entity query which adds requirements that will be used for filtering entities. Requirements can be put on tags, fragments, shared fragments, chunk fragments and even subsystems.

2. Processors then batch the updates for the chunk of entities by calling ForEachEntityChunk. This is where Chunk Fragments are use, since a single chunk fragment can be used across all the entities in the chunk.

3. The MassEntityQuery matches the requirements with the archetypes and can filter out chunks of matching archetypes based on chunk fragment filters. Requirements are usually put on the presence of a given tag or fragment in the archetype, but they can be also used to filter based on the absence of them.

4. After filtering, the MassEntityQuery triggers a function on each chunk of entities, and the individual entities of the chunk can be accessed via the FMassExecutionContext. Most of the plugin code makes use of lambda expressions when executing ForEachentityChunk.

3.2 Additional Mass Plugins

On top of the MassEntity core, Epic has developed several plugins that enable useful processors and traits for given known tasks. All these plugins in Unreal Engine 5.6 are marked as experimental, meaning that they could be subject to changes in the next future and should be used with care for projects ment for shipping.

These plugins are the result of the work done on the CitySample tech demo, where Mass was used to create massive crowd and vehicle simulations.

From here follows a brief exposition of the features provided by each of them in UE 5.6.

3.2.1 MassGameplay

Provides gameplay integration such as spawning entities in the world, replication support, level-of-detail (LOD) management, movement basics and visualization of entities spatially. It supports representing Mass entities as Actors if needed.

The plugin contains the following subsystems that build directly on top of MassEntity functionalities:

- Mass Representation Subsystem: responsible for managing the different visual aspects of the Mass Entities. It can recycle and pool spawned Actors automatically

- Mass Spawner Subsystem: spawns and manages the Entities based on the MassSpawner objects and procedural calls

- Mass LOD Subsystem: calculates the Level Of Detail (LOD) necessary for each Mass Entity. It outputs four LOD values: High, Medium, Low and Off

- Mass Replication Subsystem: replicates entities over the network in a client-server manner. It performs one-way replication from the server to its clients. This subsystem is considered experimental for Unreal Engine 5.1 and requires a C++ implementation to replicate custom values

- Mass StateTree Subsystem: used to integrate the StateTree system, a general-purpose hierarchical state machine, with Mass Entity. It allows us to configure StateTrees for each entity, and update each entity's StateTree based on signals sent from other Mass systems

- Mass Signals Subsystem: provides a way to send a named signal to tell an entity that it has some processing to do. This subsystem is currently used in Mass Statetree to wake it up when it has to perform some processing

- Mass Movement Subsystem: defines a simple movement model for Mass agents. The Fragments and Processors are set up so that other Traits can modify the velocity or force directly. For example, when steering the agent, the system may set the initial steering force and avoidance can alter it to avoid collisions during movement

- Mass SmartObject Subsystem: used to integrate the SmartObject system with MassEntity. SmartObjects are objects placed in a level that AI Agnets and Players can interact with: they are part of a global database and use a splatial partitioning structure. At a high level, Smart Objects represent a set of activities in the lecel that can be used through a reservation system. Mass SmartObject subsystem provides the requires Traits, Fragments and Processors to perform Smart Object queries and execute simple behaviors on agents represented by Mass Entities.

3.2.2 MassAI

This plugin adds AI-specific logic such as behavior tree integration via StateTree, navigation via ZoneGraph, agent avoidance and AI replication in order to enable autonomous agents.

It is composed by the following modules:

- MassAIBehavior: implements core AI logic including behavior execution, State Trees integration, decision making processes, behavior fragments and processors managing AI state and actions
- MassAIDebug: provides debugging tools and integrations, such as gameplay debugger support, visualization of AI internal states, and diagnostics specific to MassAI entities to assist development and troubleshooting
- MassAIReplication: manages network replication of MassAI entities, handling entity state synchronization between server and clients with relevancy and LOD systems to optimize bandwidth usage for large AI populations
- MassAITestSuite: contains automated tests and validation tools ensuring the correctness and stability of MassAI functionalities during development and continuous integration

3.2.3 MassCrowd

Specializes in crowd simulation with custom visualization and navigation processors, built on top of previous plugins. This was used in projects like City Sample and The Matrix Awakens demo.

The module contains processors that handle crowd movement, avoidance and visualization tailored for large groups of Mass Entities.