

Schaep Simon

Simulating massive amounts of AI agents in video games



Supervisor: Vanden Abeele Alex

Coach: Verspecht Marijn

Graduation Work 2023-2024

Digital Arts and Entertainment

Howest.be

CONTENTS

Abstract & Key words	4
Preface.....	5
Introduction.....	6
Literature Study / Theoretical Framework.....	7
1. Data oriented programming.....	7
2. Unreal Engine Mass	8
3. Spatial partitioning	10
4. Multithreading	10
5. Rendering and animations	11
5.1. Niagara.....	11
5.2. Animation Budget Allocator	12
5.3. Animation Sharing	12
5.4. Vertex Animations.....	12
6. GPU programming	13
6.1. Compute Shaders	13
6.2. CUDA.....	13
6.3. Opencl.....	14
6.4. Unreal Engine	14
case study	15
1. Experiment setup	15
1.1. Introduction	15
1.2. Requirements.....	15
1.3. Measurements	16
2. Experiment Execution	17
2.1. Simple battle Simulator	17
2.2. Mass battle Simulator	19
2.3. Rendering/LODs	28
2.4. Multithreading	29
2.5. Spatial Partitioning	30
2.6. Animation Sharing	33
3. Results.....	35
Discussion	36
1. Frame Time Over Unit Count.....	37

2. Game Thread Frame Time Composition	44
3. Summary.....	47
Conclusion	48
Future work	49
Critical Reflection	50
List of Figures	51
Figures	51
Code Snippets	51
Charts	52
References	53
Acknowledgements.....	56
Appendices	57

ABSTRACT & KEY WORDS

This paper covers the topic of optimizing the 3D simulation of massive amounts of independent AI agents in a video game made with a modern game engine. It includes a literature study on different aspects of optimization, of which a significant part is dedicated to the topic of data oriented programming. Following that, there is a case study on how to specifically optimize a battle simulator game in Unreal Engine 5, by utilizing the Mass system. We also explain how to combine techniques such as multithreading and spatial partitioning with Mass, to achieve better performance.

PREFACE

I have always been amazed by big battles in games, I would often just look at the battles in games like Planetside 2 even when I'm supposed to play in them as well. I am also a big fan of the Total War games, because they can have huge battles as well.

Ultimate Epic Battle Simulator 2 was also a big inspiration for this graduation work. In that game, you can have millions of units in a single battle, so even achieving a small fraction of that amount in my own game would be amazing.

This paper was made during a single semester, while also working on another big project. Most of the case study was made in three weeks at the end of the semester.

This paper is meant for people like me 1 year ago, with basic knowledge of C++ programming and game development, and an interest in how to simulate big battles in games. Since I wanted to develop a game with big battles, but didn't know where to start. I hope it will help some people develop some really cool games.

INTRODUCTION

You do not often encounter huge battles with thousands of units in a video game. This is because it is not simply a matter of increasing the amount of units, it is all about having the battle still run at a playable framerate.

With this study, we try to figure out how to optimize big simulations of agents in video games. We will look at the most used techniques and implement some in our own battle simulator game, which is then compared to a battle simulator made without optimizations, to see which techniques had which impact on performance.

The study assumes we are optimizing a battle simulator, but most optimization techniques can be applied to any game that has a large amount of agents in a simulation.

The main pattern that will be used is data oriented programming and how to use other techniques together with it, to achieve better performance.

The research question that we will try to answer in this research paper is the following:

How to simulate massive amounts of independent AI* agents in real-time, in a 3D game made with a modern game engine?

** When we refer to AI in this paper, we mean intelligent behavior of NPCs in a game, this can include things like pathfinding. We do not discuss the topic of machine learning in this study [1].*

To answer this question, we form a hypothesis by doing a rough estimation of how many agents we will be able to simulate at a playable framerate (30, for the purpose of this study). This is done by looking at some other games with large amounts of agents, like for example older games in the Total War franchise being able to get over 20 000 units in a battle [2], [3], and Ultimate Epic Battle Simulator 2 getting millions [4]. We arrive at the following hypothesis:

By utilizing Unreal Engine's Mass system, combined with other techniques, we can simulate more than 20 000 agents fighting each other at 30 fps.

If this hypothesis turns out to be true, the answer to the research question will be: "By utilizing Unreal Engine's Mass system, combined with other techniques."

To define these other techniques better, we construct additional hypotheses.

1. **Using Unreal Engine's Mass system will improve overall performance.**
Data oriented programming should reduce CPU cache misses, and therefore improve performance.
2. **Multithreading certain processes within Unreal Engine's Mass system will improve overall performance.**
Running processes parallel to each other, should lower the time needed to complete them all.
3. **Using octree spatial partitioning will reduce the frame time needed for target acquisition.**
An octree will likely reduce the frame time needed for finding a target, since agents won't need to calculate distances to all agents in the world.
4. **Using animation sharing will reduce animation frame time.**
Sharing animations should mean that less individual animations need to be calculated, resulting in better performance

LITERATURE STUDY / THEORETICAL FRAMEWORK

The optimization of the simulation of large amounts of agents in a video game is a complex topic because there are many different factors relating to performance. There are many ways to optimize depending on the needs of your game. Some are more complicated and difficult to implement than others.

We will do our best to explain the most commonly applied methods.

1. DATA ORIENTED PROGRAMMING

Data oriented programming (sometimes called data oriented design) is a very complex topic, so we will just explain the basics here.

However, before we can explain data oriented programming, we have to understand CPU cache memory.

The CPU has caches, which are used to speed up the process of accessing data from the RAM. A cache stores copies of frequently used data from the RAM. When the CPU requests data, it first looks into the caches, because it is faster to access memory from there. If it is not in a cache, the CPU will look in the RAM, which is a slower process [5].

Data oriented programming is a programming approach that is different to object oriented programming. When you do object oriented programming, you create classes, each having data and functionality relating to the object. A unit in a battle simulator game would for example have data to store its position, movement speed, damage... It would also have functions that operate on that data and do things like movement, attacking...

The issue is that this does not make efficient use of the CPU cache.

For example, when you want the unit to attack, the CPU might load the unit object into cache memory, which means there is lots of data that is not related to attacking, which fills up the cache. This means there is less room for data that would actually be used.

In data oriented programming, we separate the data from the functionality. We will for example group all the attacking data together, do the same with movement, and do the same with all the other data. This way, when the CPU is moving the units, it can load as much movement data as possible into the cache memory, without irrelevant data filling it up [6], [7].

In game engines like Unity and Unreal Engine, data oriented programming is applied by using ECS (Entity Component System) [8], [9].

ECS is an approach where there are entities, represented by a unique ID, that are associated with components. There are also components that belong to an entity and hold the data needed for certain functionality. Lastly, there are systems, which execute the functionality. They operate on entities that have specific components, and manipulate the data [10], [11].

An example of this is a movement system, which evaluates all entities that have a position and movement component. The position component just holds a position variable, and the movement component holds movement speed and direction variables. The system then goes over all position components and updates their position based on the direction and movement speed of the movement components with the same entity ID.

2. UNREAL ENGINE MASS

Unreal Engine has implemented their own ECS system, called Mass, with the introduction of Unreal Engine 5.0 [12]. This system uses the same concepts as ECS, but has named things a bit different. Entities are still called entities, components are called fragments, and systems are called processors [9].

There are some other concepts that they implemented, like traits, which are used to define what fragments an entity has. For example a movement trait might add a transform fragment and a movement fragment. They allow for easier configuration of entities.

Another concept they implemented are archetypes, which group entities with the same fragment types together, so that processors can operate on these archetypes. You can see this in Figure 1.

They also added tags, which are fragments without data, used to identify entities easier. An entity might have the “dead” tag to indicate it is dead. The movement processor will then not operate on archetypes with the “dead” tag. Processors will use EntityQuery objects to filter which archetypes should be included when processing, as shown in Figure 2. [9], [13], [14]

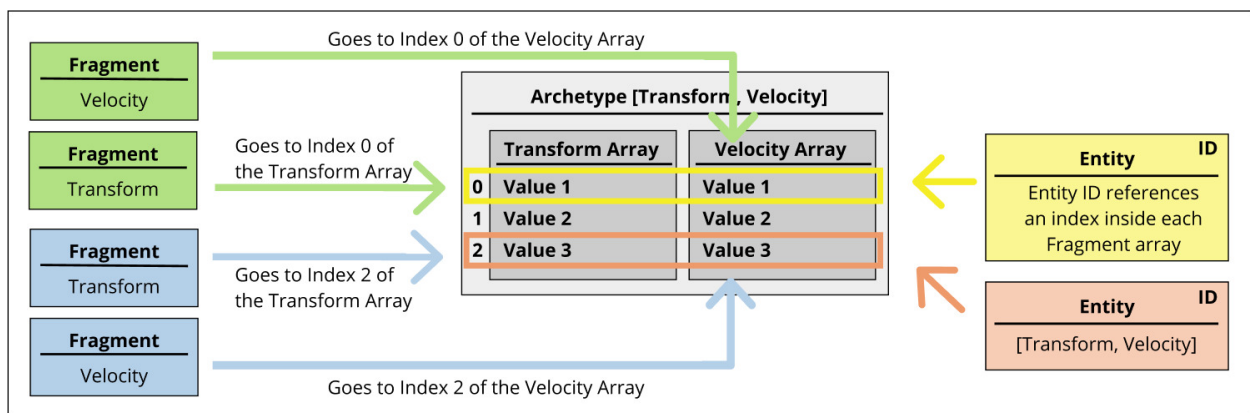


Figure 1: MassEntity archetype graph, it shows how archetypes are formed - MassEntity documentation (Epic Games)[9]

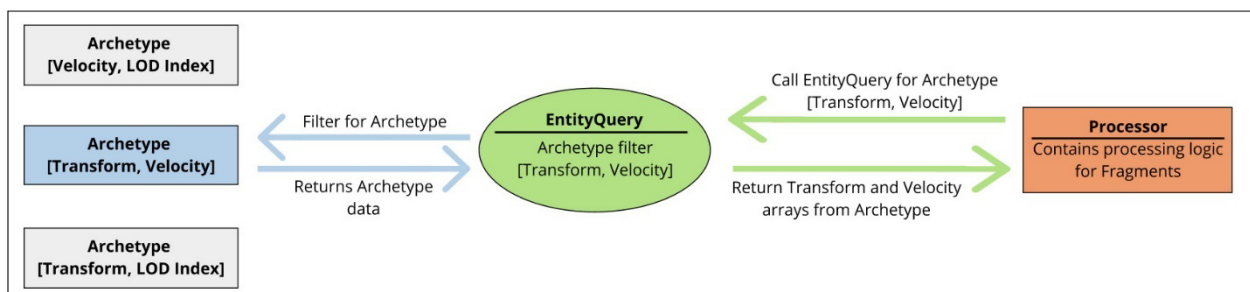


Figure 2: MassEntity processor graph, it shows how processors use EntityQueries to interact with archetypes - MassEntity documentation (Epic Games)[9]

Mass also includes lots of in-built fragments and tags that can help you set up basic functionality very quickly. It includes things like steering and avoidance, and also visualization. It is also made to work well with zonegraphs, which can be used to direct crowds. This was used extensively in the City Sample demo project, released with Unreal Engine 5.0. [14]

The built-in visualization for Mass is quite complex. It supports two different representation methods: instanced static meshes, and actors.

The method that is used, is decided by how far away the entity is from camera, by using LODs.

Instanced static meshes are the most performant, it will create an instance of a specified static mesh at the location of every entity.

Actor representation is heavier on performance since there will be an actor spawned for every entity and there is an extra step in communicating from Mass to the actor. However, actors allow for much more options in how you can render. The most obvious way of using an actor is to add a skeletal mesh to the actor to allow bone animations to be played. But you could also add logic to the actor to for example change material when you are hit.

You can even take it a step further and put gameplay logic on the actor and communicate back to Mass to inform the entity of any changes that have to be made. [13], [15], [16], [17]

3. SPATIAL PARTITIONING

In almost any type of simulation with agents, you will at some point need to query the world for other nearby agents. A good example is if you want agents to avoid each other. Using the simplest approach, you could go over all agents, and compare their distances, to know if you should move away from them. Doing this for one agent would be fine, but doing this for every agent will result in an algorithm with a complexity of $O(n^2)$. With 10 agents, that would be 100 distance calculations, but with 20 agents, it would already be 400 calculations. This can result in huge performance issues when there are large numbers of agents in the simulation.

Luckily, there are spatial partitioning data structures that can help to optimize this process, since they organize their data based on location. This allows us to use algorithms to find the closest position, without having to calculate all distances. There are many different types of spatial partitioning structures, but they are usually either flat or hierarchical.

Flat structures are the easiest to understand and implement. They divide the world in a grid and every position in the data structure is put in the according cell. If you then want to find other nearby positions, you just have to look through that cell and the neighboring ones. If the positions are moving, you will of course have a certain cost for updating which cell each position is in.

Hierarchical structures are more complex since they also take the density of the positions into account. There will for example be more cells in areas where there are more positions. This is usually more optimal when the positions aren't distributed evenly and there are lots of empty spaces. The structure is more complicated to construct and update, but allows for faster evaluation of the data in many cases. [18], [19], [20], [21]

4. MULTITHREADING

One of the more obvious ways of optimization is utilizing multiple threads to divide the work that one thread would otherwise have to do. Usually it makes the most sense to only multithread calculations that take the most time, since multithreading can be the cause of bugs and other issues.

Unreal Engine provides classes that you can use to implement multithreading. For example you can create an `FRunnable` object to create a thread that executes a certain task. Or you can use `ParallelFor` to execute a for loop on multiple threads. Another example is using `FCriticalSection` to make sure that multiple threads can't access the same piece of code at the same time. As you can see, you don't need to use the standard C++ implementations, you can use Unreal's instead [22].

The engine runs a few processes on a different thread by default, such as rendering. But most game logic such as the game tick, runs on a single thread, so there is a lot of room for optimization there [23], [24].

An important thing to be aware of is that some functionality is required to run on the game thread, such as changing the location of actors. This can limit the potential benefits of multithreading.

5. RENDERING AND ANIMATIONS

When working in a 3D game engine, rendering can be a potential bottleneck, especially when rendering a large amount of separate objects [13], [24].

Along with rendering, animations can also take a lot of CPU frame time. Especially since skeletal animations are most commonly used, and require the CPU to evaluate the bones every frame [25].

The research presented here was mostly focused on how to optimize rendering and animations in Unreal Engine 5 specifically.

5.1. NIAGARA

Niagara can be used to render more than just visual effects. Using smart particles, you can render particles as if they were actors, and even add basic gameplay functionality to them. [25]



Figure 3: Screenshot of an army rendered with Niagara - Simulating Large Crowds In Niagara | Unreal Engine (Epic Games) [25]

5.2. ANIMATION BUDGET ALLOCATOR

Unreal Engine has multiple systems to help optimize animations, one of which is the animation budget allocator. This system will dynamically lower tick rate of skeletal meshes that are further away from the camera, depending on the total budget it has. This means you can easily hard-limit the CPU time that is spent on updating animations. It is very easy to implement as it only involves switching skeletal meshes with budgeted skeletal meshes, and setting up the budget you want to give.

An important negative is that if you would exceed the budget too much, you might notice lower animation frame rates even on meshes that are closer.

This system is perfect for when you have more than average, but still relatively small amounts of skeletal meshes. [26]

5.3. ANIMATION SHARING

Animation sharing is another system Unreal Engine has available to help optimize skeletal mesh animations. Instead of each skeletal mesh evaluating their own animation individually, they can share it with each other. This works by evaluating one animation instance for every state required, and then each skeletal mesh component requests animation data from the animation sharing manager, which is also based on which state the mesh owner is in.

This system requires a bit of work to set up correctly, especially if you want it to seem as if each mesh has their own animation or you want to use animations that trigger on certain events, like attacking [27].

5.4. VERTEX ANIMATIONS

Vertex animations are an alternative to skeletal animations, the biggest advantage they have is that they run almost completely on the GPU.

Vertex animations store animation data in a texture, so the shader reads the texture to know where each vertex of the mesh needs to be at which frame in the animation.

A big disadvantage is that you lose a lot of cool things you can easily do with skeletal animations, like blending. It is still possible to blend between different animations, but you need separate textures for every possible blending scenario.

There is also extra labor required to create vertex animations. You usually first create a skeletal animation, and then afterwards use a tool to bake the animation to a texture. After which you also need to make sure the material correctly uses it.

Unreal Engine has a plugin called AnimToTexture, that allows you to bake animation sequences to textures. [25], [28]

6. GPU PROGRAMMING

Whenever the CPU is your biggest bottleneck and the GPU is still relatively free, it can be a good idea to delegate workload from the CPU to the GPU. The GPU is also able to do certain operations much more efficiently than the CPU, which can be another reason to use one of the following techniques.

However, doing calculations on the GPU can add a lot of complexity to your code and should only be done if you need insane performance optimizations. For example in Ultimate Epic Battle Simulator 2, where there are millions of units in a battle, they achieved this by running most of their calculations on the GPU.



Figure 4: Screenshot of a battle with 10 million units in Ultimate Epic Battle Simulator 2 - UEBS 2 The Making Of Pt1 (BrilliantGameStudios)[4]

6.1. COMPUTE SHADERS

Compute shaders are shaders that instead of rendering, are used for computations. They work very similar to normal shaders, except that their output is not used for rendering.

Because they run on the GPU, they can do some operations much more efficient than the CPU. They also run on a separate thread, so they won't block the thread from which you call them. [29], [30], [31], [32], [33]

6.2. CUDA

CUDA is an API developed by Nvidia, to allow for programming on the GPU. It has the same purpose as compute shaders, but instead of writing a shader, you can use a programming language like C, C++ or Fortran. This often makes it easier or more intuitive to use than compute shaders.

Since this is developed by Nvidia, it will only run on Nvidia GPUs. [34], [35], [36], [37]

6.3. OPENCL

OpenCL is very similar to CUDA, but can run on both Nvidia and AMD GPUs since it is developed by Khronos Group. It is a more general parallel programming API, but its main use is GPU programming. [29], [30], [31], [38], [39], [40]

6.4. UNREAL ENGINE

Unreal Engine has native support for using compute shaders, though it is not an extensively explored topic. There are also plugins available for using CUDA and OpenCL, but there is also not a lot of information available about them. [41], [42], [43], [44]

CASE STUDY

1. EXPERIMENT SETUP

1.1. INTRODUCTION

In this case study, we create a battle simulator game, using Unreal Engine's Mass system, and implement different techniques to try to increase performance to allow for the simulation of massive amounts of agents at a playable fps. Playable fps is defined as 30 fps for the purpose of this paper.

We will then compare the performance of the battle simulator that uses Mass, with a simple battle simulator game that is created using usual methods. This is to verify what the impact of using Mass is on performance.

We will also compare the different additions to the Mass battle simulator to see which ones improved what aspects.

Some snippets of code will be included here, for the full source code, please refer to the GitHub repository:

<https://github.com/SimonSchaep/Research-UE5-Mass/tree/main/UnrealProjects>

Code will be written with the standard Unreal Engine coding standards in mind:

<https://docs.unrealengine.com/5.3/en-US/epic-cplusplus-coding-standard-for-unreal-engine/>

1.2. REQUIREMENTS

The simple and Mass battle simulator both have the following requirements:

- Two teams of units face each other.
- When the battle starts, the units run towards the closest enemy unit.
- When they are within a small distance (attack range), they will stop moving and start attacking.
- Attacking means that after a repeating delay, the unit deals damage to its target, reducing their health.
- When the target reaches zero health, it dies.
- The units have to avoid other units.
- The movement utilizes pathfinding to avoid big obstacles and navigate through the world.
- 3D models are used to represent the units, they are of standard video game quality.
- The following animations play at the correct time:
 - Idle
 - Moving
 - Attacking
 - Dying

To ensure the quality of the assets that are used we will be using models from the game Paragon [45]. This was a AAA game developed by Epic Games, which means the quality is at least industry standard and it should be straightforward to use the assets in Unreal Engine. All assets are available on the Unreal Engine Marketplace for free, we will be using the paragon minion models specifically: <https://www.unrealengine.com/marketplace/en-US/product/paragon-minions>

For environmental assets we will be using the free Landscape Pro 2.0 package from the Unreal Engine Marketplace: <https://www.unrealengine.com/marketplace/en-US/product/landscape-pro-auto-generated-material>

1.3. MEASUREMENTS

We will measure performance of the simple battle simulator, the battle simulator made with Mass, and also every time we add a major optimization feature (multithreading, spatial partitioning, animation sharing).

Eight different unit counts will be measured, starting at 100 vs 100, then doubling until we reach 12800 vs 12800.

With these multiple tests we will be able to see how frame time scales with agent count.

Performance will be measured by frame time, which will be recorded using Unreal Insights, in a development build of the game. This is not a shipping build, so there will still be a certain performance cost because debugging data is included, however, this is required to be able to properly profile [24], [46].

We will record from the start of the battle (after spawning in the units), until one side has lost all their units.

Meanwhile the camera will stay static, to eliminate any variance that could be caused by culling, LODs...

We will use the frame at 10 seconds into the battle for our comparisons. Unless that frame has some irregular frame time spike, then we take the next frame. We do this because frame times will differ a lot over time, and we want to accurately compare between all implementations.

The goal is not to have super precise measurements with exact numbers. Instead, we want to get a general idea of where we should optimize and which techniques have a noticeable impact on which aspects of the game.

There will only be one device used for measurements, an Omen Gaming Laptop with the following specs:

CPU: AMD Ryzen 7 5800H 3.20 GHz

RAM: 16 GB

GPU: NVIDIA GeForce RTX 3060 Laptop GPU

2. EXPERIMENT EXECUTION

2.1. SIMPLE BATTLE SIMULATOR

For the simple battle simulator, we will follow the normal pipeline for creating a game in Unreal Engine.

We create a unit manager as a component of the game mode. This manager spawns a specified number of actors within a specified area. It also holds two arrays of pointers to those spawned actors, so these can later be queried when looking for nearby units. There are two arrays, so that the actors can easily be filtered based on which army they belong to. This ensures that the units don't target allies, and reduces the amount they have to look through.

The unit is an actor that inherits from ACharacter, since that will allow us to use the default character movement component, which has in-built RVO (Reciprocal Velocity Obstacles) avoidance.

Almost all functionality will be implemented in separate components, since that allows reusability in case we would implement different types of units. It will also be easier to translate to Mass, since each component can be replaced by a fragment and a processor when simplified.

2.1.1. COMPONENT OVERVIEW

TARGET ACQUISITION COMPONENT

Holds the ID of the army this unit belongs to, so it can filter the arrays of units from the unit manager

Every Tick, get all alive units from the unit manager.

Find and store the one that is the closest to this unit, like in Code snippet 1: Acquiring the closest target.

```
ClosestTargetDistanceSqr = FLT_MAX;
float DistanceSqr;
for (AActor* Actor : UnitManager->GetAllEnemyUnits(TeamId))
{
    DistanceSqr = FVector::DistSquared(GetOwner()->GetActorLocation(), Actor-
    >GetActorLocation());
    if (DistanceSqr < ClosestTargetDistanceSqr)
    {
        ClosestTarget = Actor;
        ClosestTargetDistanceSqr = DistanceSqr;
    }
}

if (ClosestTargetDistanceSqr == FLT_MAX)
{
    //No target was found
    ClosestTargetDistanceSqr = 0;
    ClosestTarget = nullptr;
}
```

Code snippet 1: Acquiring the closest target

MOVE COMPONENT

Every Tick, read the current target from the target acquisition component.

If we are outside the stop range, move towards the target. If we use the AIcontroller MoveTo functions, it will follow a path on the navigation mesh.

Otherwise, stop all movement.

HEALTH COMPONENT

Holds the current health amount.

Has a function that subtracts health, and broadcasts a delegate event when health reaches zero or below. This delegate is used to disable tick on all the components of the actor.

After death, we leave the actor alive, since it looks cool and won't bring our performance lower than it was at the start of the battle. If you are in a scenario where you spawn in more units as the battle progresses, it is probably better to destroy the actor sometime after death.

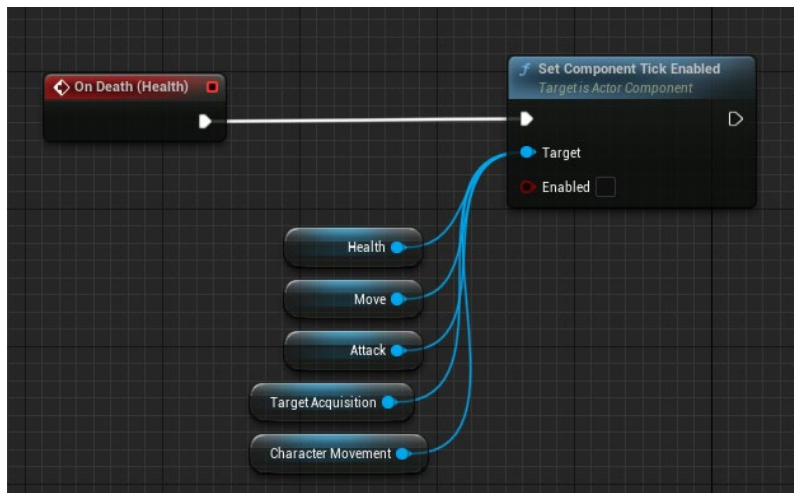


Figure 5: Disabling component tick in the actor blueprint

ATTACK COMPONENT

Holds data regarding attacking (damage, attack interval, attack interval timer).

Every Tick, read the current target from the target acquisition component.

If the target is within attack range, count down the timer.

If the timer reaches zero or below, deal damage to the target and reset the timer.

2.1.2. ANIMATIONS

We still have to set up animations. For that, we add our Paragon minion model to the skeletal mesh component in the unit blueprint and we create a simple animation blueprint with four different states:

idle, moving, attacking and dead.

We assign the correct animations to each state and we get our current state from an animation instance object that we define in code. This class will evaluate the actor that the skeletal mesh is a part of, and by using the components, will determine the state that the unit is in.

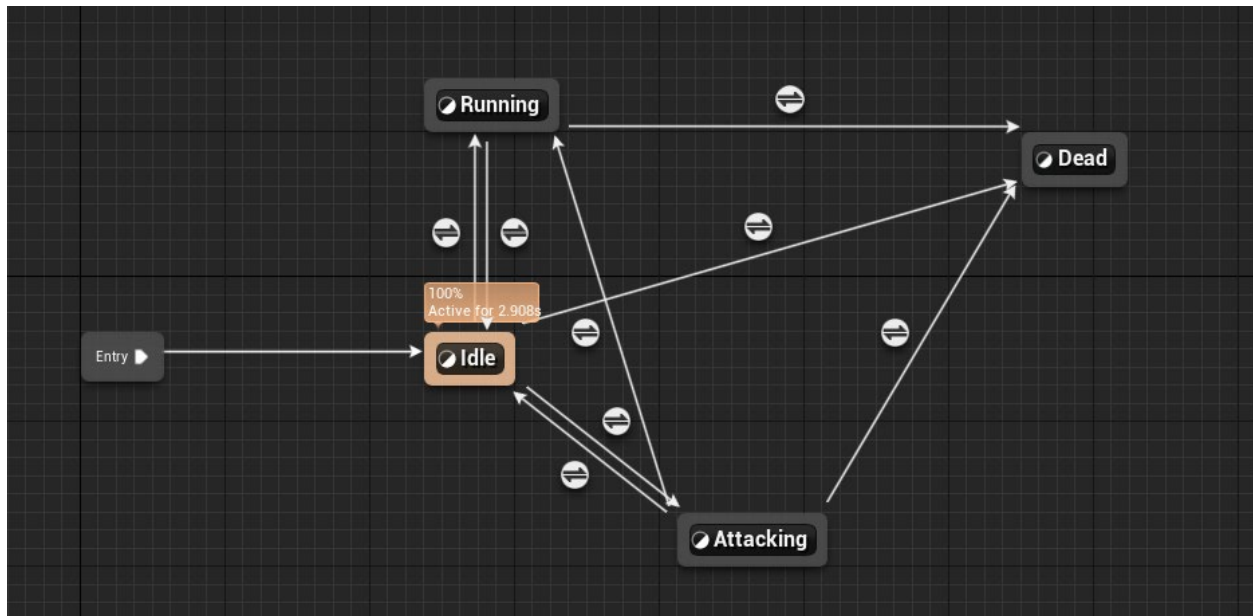


Figure 6: Simple battle simulator animation state graph

2.1.3. PROJECT SETTINGS

We don't change many project settings, except for changing the global illumination method from lumen to none. The anti-aliasing method is kept at the default Temporal Super Resolution, because other methods look noticeably worse in this project.

We leave the shadow map method to virtual shadow maps.

2.2. MASS BATTLE SIMULATOR

For the battle simulator made with Mass we have to first include the required plugins, these include MassEntity (for all base Mass functionality) and MassAI, MassGameplay and MassCrowd (which all include some pre-built functionality as well)

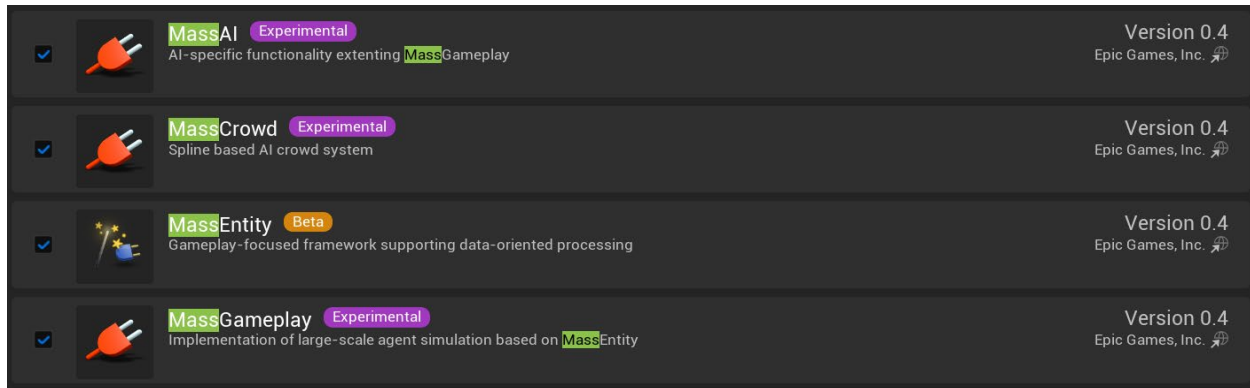


Figure 7: Plugins required for using Mass in Unreal Engine 5

Next, we will create our fragments and processors. Fragments are very simple, they just hold data.

Setting up a processor usually requires the following three functions: the constructor, ConfigureQueries and Execute.

In the constructor, we define the initial value of some variables, like whether the processor will automatically register itself at the start of the game, or if we want to do that at another time through our own code. We can also configure during which tick phase we want our processor to execute, and there other options as well. ConfigureQueries is used for configuring which archetypes the processor will operate on, you can define which fragments and/or tags the archetype needs to have or not have.

In the Execute function, you can program the operations that the processor needs to perform on each entity chunk. Chunks are entities grouped by archetype.

A good example of these functions is shown in Code snippet 6: DestroyEntitiesProcessor's three main functions.

2.2.1. FUNCTIONALITY

SPAWNING

Spawning Mass entities can be done in two ways by default:

Adding a mass agent component to an actor, and spawning in the actor.

Or using the mass spawner actor to spawn in entities based on an EntityConfig data asset. This mass spawner also uses a spawn data generator to generate spawn points.

We could create our own spawner as well if we want custom spawning behavior, but for our purposes, the default mass spawner will do.

The EntityConfig asset can be used to define which traits, so in turn, which fragments will be part of the spawned entities.


▼ Spawn	
Count	100
▼ Entity Types	1 Array element
▼ Index [0]	2 members
Entity Config	 <div>EC_SimpleUnit</div>
Proportion	1.0
▼ Spawn Data Generators	1 Array element
▼ Index [0]	2 members
▼ Generator Instance	EQS SpawnData Generator
▶ Query	
Proportion	1.0
Auto Spawn on Begin Play	<input checked="" type="checkbox"/>
Spawning Count Scale	1.0

Figure 8: Example of a mass spawner configuration

▼ Entity Config	
Validate Entity Config	
▼ Config	
Parent	None
▼ Traits	3 Array elements
▶ Index [0]	Move Trait
▼ Index [1]	Attack Trait
▼ Attack Parameters	
Range	200.0
Attack Delay	1.0
Damage	20.0
Animation Attack Delay	0.0
▶ Index [2]	Mass Movable Visualization Trait
Config Guid	{924A7A5A-4C23-0FAD-4CB6-508F1BC7AB38}

Figure 9: Example of an EntityConfig data asset configuration

TARGET ACQUISITION

For the target acquisition, we first need to be able to associate every entity with a certain army. For that, we add an army ID fragment, which just holds an integer value that represents the army this entity belongs to. You can see this in Code snippet 2: FArmyIdFragment definition. This is a simple example of how to define a fragment.

```
USTRUCT()
struct BATTLESIMULATORMASS_API FArmyIdFragment : public FMassFragment
{
    GENERATED_BODY()

    int ArmyId = 0;
};
```

Code snippet 2: FArmyIdFragment definition. This is a simple example of how to define a fragment

Ideally, we should be able to configure a spawner to spawn units with a specific army ID. This is not possible with the default spawner – data generator setup. To solve this issue, we create our own spawn data generator, based on the default EQS spawn points generator. Here we just add an extra value, the army ID, to the spawn data. Then we also create a copy of the default UMassSpawnLocationProcessor that sets up all spawned entities with the correct spawn transforms, we again just add the army ID.

Next, we need to be able to get a list of all possible targets. For that, we will be using a World Subsystem, which is similar to a singleton in that it can easily be accessed from any class. It is created right before the world initializes, and destroyed after the world cleans up.

This subsystem will hold an array of arrays of mass entity handle, similar to the simple battle simulator. A mass entity handle is a struct that can be used together with the entity manager to operate on a specific entity, for example getting a specific fragment or tag.

We then create a target acquisition processor, where we go over every entity and for each one, we go over all entity arrays that are not at the index of our own army ID, to find the one that is closest. We then update our target acquisition fragment with the target we found. The code can be seen below.

```
EntityQuery.ForEachEntityChunk(EntityManager, Context, ([&](FMassExecutionContext& Context)
{
    const TArrayView<FArmyIdFragment> ArmyIdList =
    Context.GetMutableFragmentView<FArmyIdFragment>();
    const TArrayView<FUnitTargetAcquisitionFragment> TargetAcquisitionList =
    Context.GetMutableFragmentView<FUnitTargetAcquisitionFragment>();
    const TConstArrayView<FTransformFragment> TransformList =
    Context.GetFragmentView<FTransformFragment>();

    for (int32 EntityIndex = 0; EntityIndex < Context.GetNumEntities();
    ++EntityIndex)
    {
        //Acquire target
        float& ClosestDistanceSqr =
        TargetAcquisitionList[EntityIndex].ClosestTargetDistanceSqr;
        ClosestDistanceSqr = FLT_MAX;

        //Loop over all entities in TargetAcquisitionSubsystem
```

```

auto& EntitiesArrays = TargetAcquisitionSubsystem-
>GetPossibleTargetEntities();
for (int32 ArrayIndex{}; ArrayIndex < EntitiesArrays.Num();
++ArrayIndex)
{
    //Skip those with the same army id
    if (ArrayIndex == ArmyIdList[EntityIndex].ArmyId)
    {
        continue;
    }

    //Find closest entity
    for (const FMassEntityHandle& Handle :
EntitiesArrays[ArrayIndex])
    {
        if (!EntityManager.IsEntityValid(Handle)) continue;

        auto HandleTransform =
EntityManager.GetFragmentDataStruct(Handle,
FTransformFragment::StaticStruct()).Get<FTransformF
ragment>();
        float DistanceSqr =
FVector::DistSquared(TransformList[EntityIndex].Get
Transform().GetLocation(),
HandleTransform.GetTransform().GetLocation());
        if (DistanceSqr < ClosestDistanceSqr)
        {
            TargetAcquisitionList[EntityIndex].CurrentTar
get = Handle;
            ClosestDistanceSqr = DistanceSqr;
        }
    }
}

//No target was found
if (ClosestDistanceSqr == FLT_MAX)
{
    ClosestDistanceSqr = 0;
    TargetAcquisitionList[EntityIndex].CurrentTarget =
EntityManager.InvalidEntity;
}
}

});

```

Code snippet 3: TargetAcquisitionProcessor Execute functionality. A more complex example of a processor

MOVEMENT

For movement, we will use the Steering, Avoidance, Movement and SmoothRotation traits that are included in the Mass plugin. These will provide the basic functionality required to get the units moving.

We still have to set the steering target and do pathfinding ourselves. To do so, we will create a movement processor and a navigation processor.

The navigation processor will go over every entity, and find a path on the navigation mesh towards the target from the target acquisition fragment. It uses the `UNavigationSystemV1::FindPathSync` function to do so, as seen in Code snippet 4.

The movement processor will check if the unit is within stop range, and if so, will stop the unit by setting the steering target to its own position. With this, there is still one issue, the avoidance will still be active. To solve this, we set the velocity and force values of the according fragments to zero every frame. It would be more ideal to have our own avoidance system where we can just disable it, but that is out of scope of this case study.

```
//Configure path query params
FNavAgentProperties NavAgentProperties{};
FPathFindingQuery NavParams{};
NavParams.NavData = NavigationSystem->MainNavData;
NavParams.QueryFilter = NavParams.NavData->GetDefaultQueryFilter();
NavParams.StartLocation = Transform.GetLocation();
NavParams.EndLocation = TargetEntityLocation;

//Find path
FPathFindingResult Result = NavigationSystem->FindPathSync(NavAgentProperties,
NavParams, EPathFindingMode::Regular);

if (Result.IsSuccessful() && Result.Path->GetPathPoints().Num() >= 2)
{
    const FVector& Target = Result.Path->GetPathPoints()[1]; //Take point after
    starting point of path
    MoveTarget.Center = Target;
}
else //We are outside navmesh, try to get back
{
    FNavLocation NavLocation{};
    NavigationSystem->ProjectPointToNavigation(Transform.GetLocation(),
NavLocation, FVector(1000, 1000, 1000));
    MoveTarget.Center = NavLocation.Location;
}
```

Code snippet 4: Navigation mesh pathfinding in the navigation processor Execute function

ATTACKING

For our attacking functionality, we need two fragments to hold data, an Attack and a Health fragment. The attack fragment will hold the attack delay timer, while the health fragment will keep track of the current health. For our constant data, like damage and attack delay, we will use a shared fragment. This fragment will be shared between all entities of the same archetype, allowing for more efficient memory usage. It is a good practice to make use of shared fragments wherever possible, to reduce memory usage. The definition of this shared fragment can be found below.

```
USTRUCT()
struct BATTLESIMULATORMASS_API FUnitAttackParameters : public FMassSharedFragment
{
    GENERATED_BODY()

    UPROPERTY(EditAnywhere)
    float Range = 200.f;

    UPROPERTY(EditAnywhere)
    float AttackDelay = 1.f;

    UPROPERTY(EditAnywhere)
    float Damage = 20.f;

    UPROPERTY(EditAnywhere)
    float AnimationAttackDelay;
};
```

Code snippet 5: Attack parameters shared fragment definition

The attack processor will simply go over all entities, and count down their attack delay timer. Once this timer is below or equal to zero, the health fragment of the target entity will lose health equal to the damage of the shared fragment. If the health falls below 0, it will simply add a dying tag to the entity, which will exclude it from most processors and allow a death processor to take over.

DYING

Dying will be handled in two separate stages: dying and dead, which will be indicated by the appropriate tag. Dying entities are still in the process of dying, they need to play their animation, but no longer be doing any functionality like movement. They also need to be excluded from the target acquisition subsystem. The dying fragment will use a timer to indicate how long it takes until it is dead. The processor will go over every entity and count down the timer, once done, it will spawn a new entity, according to an EntityConfig asset. This entity will just have a transform and visualization.

We will not immediately destroy the now dead entity, as a good practice is to instead mark it as dead by using a tag, and then using DestroyEntities() to destroy multiple entities at once. You can find the code for this processor below.

```
UDestroyEntitiesProcessor::UDestroyEntitiesProcessor()
    :EntityQuery{ *this }
{
    bAutoRegisterWithProcessingPhases = true;
    ProcessingPhase = EMassProcessingPhase::FrameEnd;
    ExecutionFlags = int32(EProcessorExecutionFlags::All);
}

void UDestroyEntitiesProcessor::ConfigureQueries()
{
    EntityQuery.AddTagRequirement<FDeadTag>(EMassFragmentPresence::All);
}

void UDestroyEntitiesProcessor::Execute(FMassEntityManager& EntityManager,
FMassExecutionContext& Context)
{
    EntityQuery.ForEachEntityChunk(EntityManager, Context,
    ([&](FMassExecutionContext& Context)
    {
        EntityManager.Defer().DestroyEntities(Context.GetEntities());
    }));
}
```

Code snippet 6: DestroyEntitiesProcessor's three main functions

2.2.2. VISUALIZATION

We will visualize the units by adding the `MassMovableVisualizationTrait` to the `EntityConfig` asset. This trait requires us to configure which actors or static meshes will represent the entity, along with at what distance actors or instanced static meshes will be shown. The static meshes will be just one extracted frame from the idle animation, and the actor will be a newly created actor blueprint.

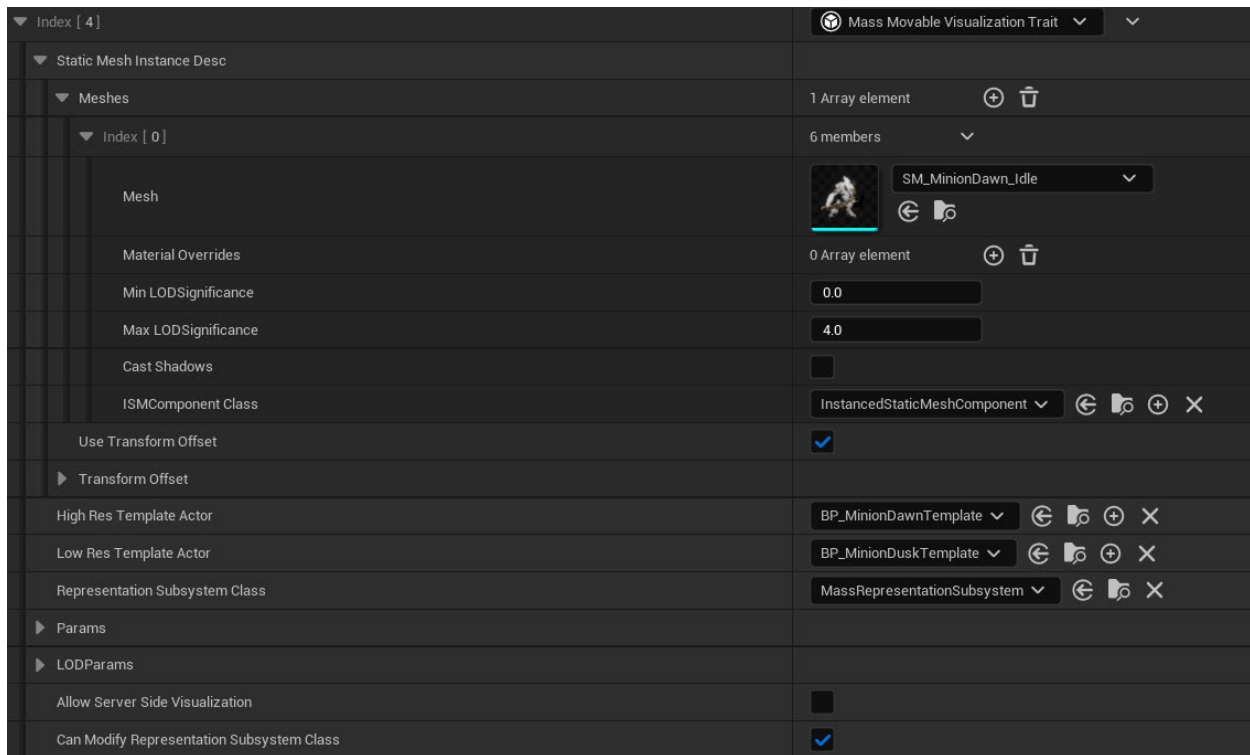


Figure 10: Visualization trait configuration

This actor needs to have a mass agent component to be able to work properly. In this component, we can assign a sync trait. There are some sync traits that come with the plugin, but these assume that our actor inherits from `ACharacter` and have a character movement component. This is not the case for us, so we will create our own sync trait and translator. The translator is actually just a processor that sets the transform of the actors associated with an entity.

We also need to add the `LODCollectorTrait`, to make sure that the correct LOD tags are added to, and removed from the entity. This way, the visualization processor knows when to show what type of visualization.

2.2.3. ANIMATION

We will not implement vertex animations, so whenever the instanced static meshes are shown, they will not be animated. However, we will use skeletal meshes in the actors that are used for visualization up close. This means we add a skeletal mesh component to our actor, assign the correct model and create an animation blueprint. The animation blueprint is exactly the same as the one used in the simple battle simulator, but the animation instance class that is used to gather the state of the actor will be different.

We need to access the mass agent component, get the associated entity handle, and get a fragment that holds our animation state. This fragment's state will be set by various processors, like the movement and attack processors.

References used for creating the battle simulator with Mass: [9], [13], [14], [15], [16], [17], [47], [48], [49], [50], [51], [52]

2.3. RENDERING/LODS

We soon noticed that our draw calls exceeded insane amounts, and the draw thread got overloaded way sooner than expected. This is because we originally used models from the Paragon Greystone pack, which were intended to be models for a main player character. This meant it had 18 different materials applied resulting in an insane amount of draw calls per unit. After replacing this with the Paragon minions models, the draw calls went back to a more normal amount.

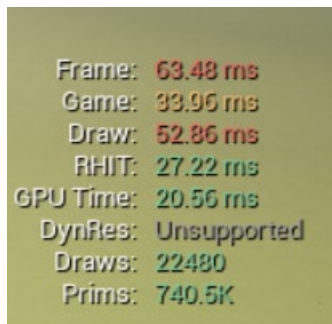


Figure 11: Draw call amount with old model

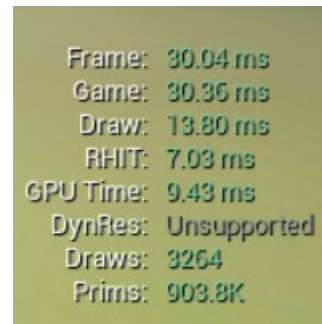


Figure 12: Draw call amount with new model

However, the primitive count was still too high and could become a bottleneck. To solve/remedy this, we generated new LODs for the models. Going from 5000 vertices to only 250 at far distances. Generating LODs in Unreal Engine is luckily very easy, you just configure an LOD data asset by specifying the percentage of triangles that you wish to keep at which LOD levels, and click regenerate.

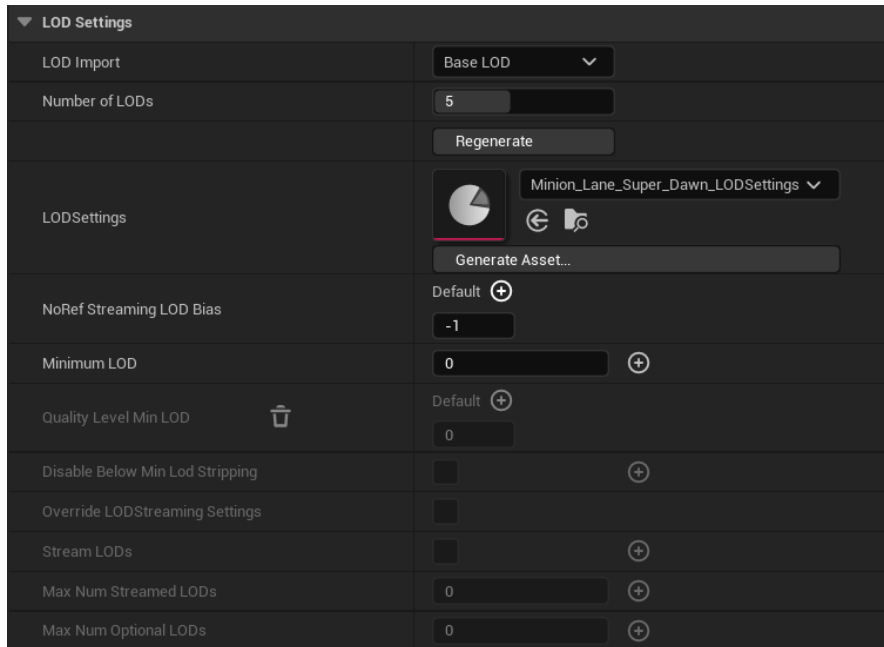


Figure 13: LOD Settings after generating an LOD settings asset

2.4. MULTITHREADING

As of Unreal Engine 5.1, Mass already does some multithreading by default [53]. Every processor will execute on a separate thread, unless the `bRequiresGameThreadExecution` boolean is true. This might happen in parallel with other processors, but the execution order often won't allow it.

However, a single processor can still take a lot of frame time, and it might benefit from dividing tasks between multiple threads.

The processors that currently take the most frame time, are the target acquisition, avoidance and navigation processors. We could multithread the avoidance processor by copying it and making our own, but since that would cost too much time, we will skip this one. The target acquisition processor can easily be multithreaded by just replacing the normal for loop with a `ParallelFor`, as seen below.

```
#ifdef ENABLE_MULTITHREADING
    ParallelFor(Context.GetNumEntities(), [&](int32 EntityIndex)
#else
    for (int32 EntityIndex = 0; EntityIndex <
        Context.GetNumEntities(); ++EntityIndex)
#endif // ENABLE_MULTITHREADING
```

Code snippet 7: Replacing the normal for loop with a `ParallelFor`

We can do the exact same in our navigation processor. Alternatively, we could use a normal for loop and replace FindPathSync with FindPathASync, but from a quick test, this seems to be slower than FindPathSync in a ParallelFor.

We also multithreaded the movement and attacking processors, which in hindsight, was not needed since they already barely take any frame time at all.

Also make sure to use a mutex (FCriticalSection in Unreal Engine) whenever you don't want two threads accessing the same section of your code at the same time. This can be seen in an example below.

```
//Check if dead
if (TargetEntityHealth.CurrentHealth <= 0)
{
    Mutex.Lock();
    Context.Defer().AddTag<FDyingTag>(TargetEntity);
    Mutex.Unlock();
}
```

Code snippet 8: Using a mutex to prevent simultaneous writing to the deferred command buffer

References used to implement multithreading: [22], [54]

2.5. SPATIAL PARTITIONING

There are many types of spatial partitioning structures, ideally, we would test out multiple and compare which give the best result. However, this would shift the focus of the study too much towards spatial partitioning, which is not the goal.

The avoidance processor that is included in the plugin, uses a hash grid for its spatial partitioning. We will use an octree, since there will likely be a high variance in density of the units. At the start of the battle, they will be evenly divided, but after they meet each other, potentially less than 10% of the space will be used. Therefore, we expect better results for our case if we use an octree, since it should be faster at querying data when there is a high variance in density.

We could have also used a quadtree (2D version of an octree), since our movement is limited to a 2D plane right now, but that might change in the future so we use an octree instead.

Luckily, Unreal Engine has some spatial partitioning structures built into the engine, like the hash grid that was used for avoidance. They also have multiple octree implementations, the one we will use is TOctree2 [55] (the original TOctree is deprecated [56]), since it is the most similar to a classic octree. There is also FSparseDynamicOctree3 [57], which is a grid of octrees, which could be interesting but we will not experiment with it during this case study.

To implement the octree, we replace the arrays in the target acquisition subsystem with an octree. The documentation of TOctree2 is very limited and it is somewhat difficult to figure out how to properly use it, so we will look at other places in the engine code where it is used, and base our implementation on that.

After defining the octree, we will create functions that add, remove, and update our octree elements.

Updating an element is done by removing and reinserting it in the octree. This will get done through a processor that updates all entities in the octree every frame. This can become an expensive operation to do for every entity, so it would be a good idea to use a tag to mark an entity to require an update, that way we avoid having to update the octree for every entity every frame.

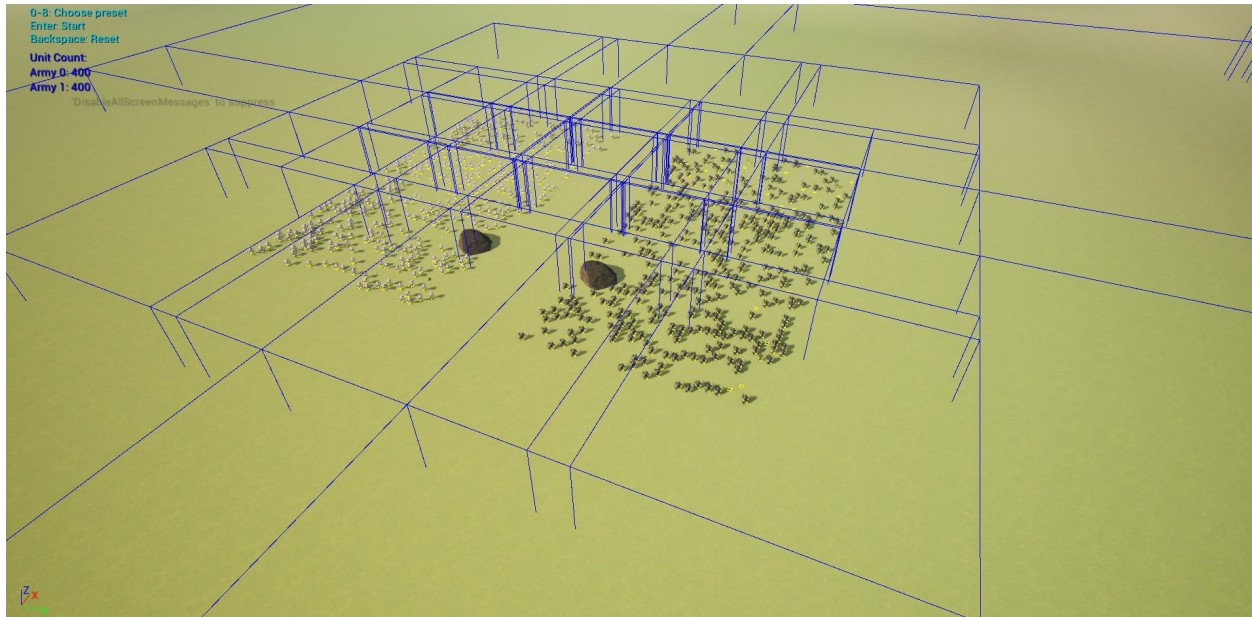


Figure 14: Visualization of the octree, the blue boxes represent the nodes of the tree

Next, for finding the closest target in the octree, we don't have access to all nodes directly, but we have some functions that will help us operate on elements in the nodes. First, we will use `FindNearbyElements`, to go over all elements in the same node as our entity. This will give us a close entity, but might be inaccurate, since there could be a neighboring node, that has an entity very close to the edge that is closer to us. So to get an accurate result, we will use `FindElementsWithBoundsTest` afterwards, with the radius of our bounds equal to the closest distance that we found so far. Our implementation is shown below.

```
for (int32 OctreeIndex{}; OctreeIndex < Octrees.Num(); ++OctreeIndex)
{
    //Skip those with the same army id
    if (OctreeIndex == ArmyIdList[EntityIndex].ArmyId)
    {
        continue;
    }

    //Check for elements in the same node
    Octrees[OctreeIndex].FindNearbyElements(TransformList[EntityIndex].GetTransform().GetLocation(), [&](const FUnitOctreeElement& OctreeElement)
    {
        const FMassEntityHandle& Handle = OctreeElement.EntityHandle;

        if (!EntityManager.IsEntityValid(Handle)) return;
    });
}
```

```

        auto HandleTransform = EntityManager.GetFragmentDataStruct(Handle,
        FTransformFragment::StaticStruct()).Get<FTransformFragment>();
        float DistanceSqr =
        FVector::DistSquared(TransformList[EntityIndex].GetTransform().GetLocation(), HandleTransform.GetTransform().GetLocation());
        if (DistanceSqr < ClosestDistanceSqr)
        {
            TargetAcquisitionList[EntityIndex].CurrentTarget = Handle;
            ClosestDistanceSqr = DistanceSqr;
        }
    });

    //Do bounds check to see if there's any elements in other nodes that are closer
    float ClosestDistance = FMath::Sqrt(ClosestDistanceSqr);
    FBoxCenterAndExtent Bounds{
    TransformList[EntityIndex].GetTransform().GetLocation(),
    FVector{ClosestDistance, ClosestDistance, ClosestDistance}
    };

    Octrees[OctreeIndex].FindElementsWithBoundsTest(Bounds, [&](const
    FUnitOctreeElement& OctreeElement)
    {
        const FMassEntityHandle& Handle = OctreeElement.EntityHandle;

        if (!EntityManager.IsEntityValid(Handle)) return;

        auto HandleTransform = EntityManager.GetFragmentDataStruct(Handle,
        FTransformFragment::StaticStruct()).Get<FTransformFragment>();
        float DistanceSqr =
        FVector::DistSquared(TransformList[EntityIndex].GetTransform().GetLocation(), HandleTransform.GetTransform().GetLocation());
        if (DistanceSqr < ClosestDistanceSqr)
        {
            TargetAcquisitionList[EntityIndex].CurrentTarget = Handle;
            ClosestDistanceSqr = DistanceSqr;
        }
    });
}

```

Code snippet 9: Finding the closest target with an octree

References used to implement spatial partitioning: [18], [19], [20], [21]

2.6. ANIMATION SHARING

After optimizing the game logic, we notice that our newest bottleneck is the animations. Skeletal mesh updates and animation blueprint ticks seem to take too much frame time.

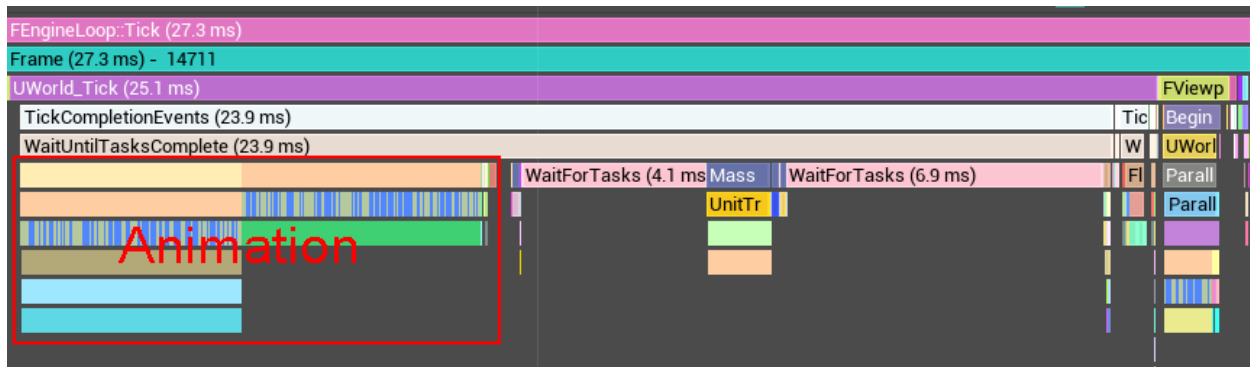


Figure 15: Screenshot from within Unreal Insights to show the amount of frame time the animations take

There are multiple ways to optimize animations, the most optimal solution would be to use vertex animations. This would take a long time to implement and might potentially be challenging to combine with the static mesh instance rendering from Mass that we are currently using.

Another option would be to use the animation budget allocator, however, this could cause noticeable animation lag with our large amount of units.

There is another option, the animation sharing manager. This seems to be an ideal solution, since we can keep using our skeletal meshes. And because all of our units share the same skeleton, we should be able to improve performance quite a lot by sharing animations.

To set up the animation sharing manager, we have to first create an instance of the manager in our gamemode.

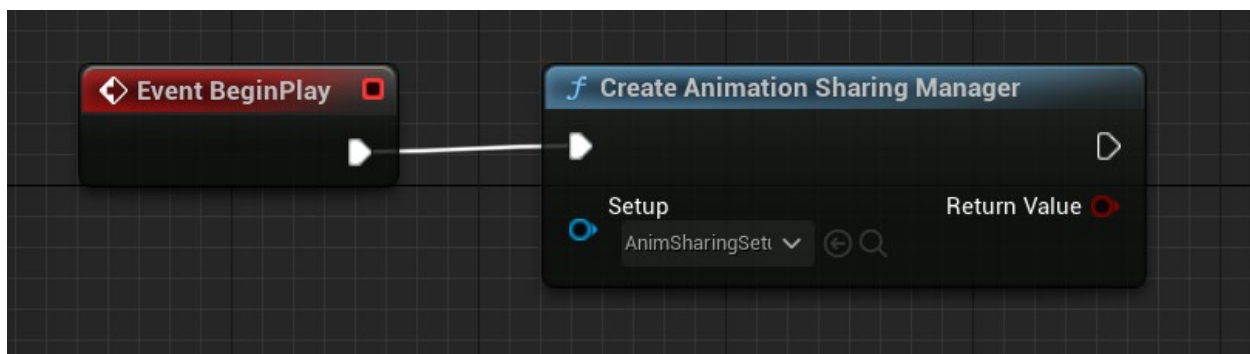


Figure 16: Creation of the animation sharing manager

Then, we have to define an AnimationSharingSetup asset. In here, we define which skeletons are used and which animations they have. We also have to enable blending and set an amount of different instances per animation, to try and make it less noticeable that animations are shared.

For our attack and death animations, we need to set them to be on demand, since we want an individual instance to play every time a unit attacks or dies.

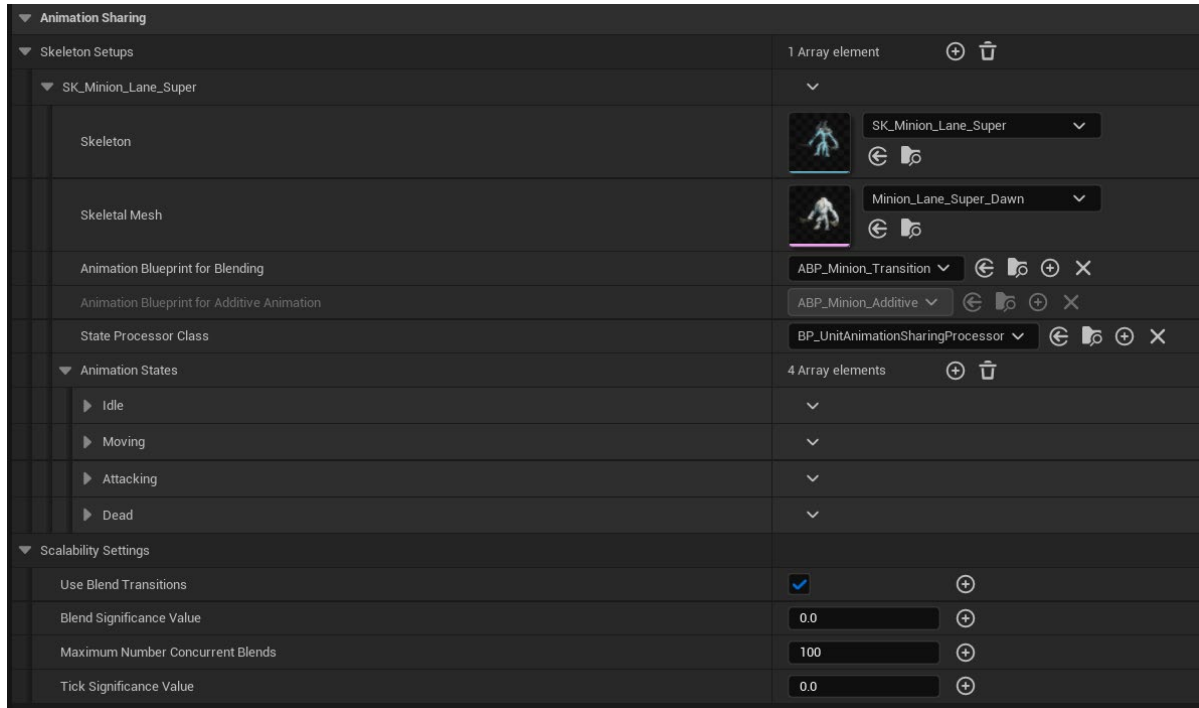


Figure 17: Configuration of the AnimationSharingSetup asset

Lastly, we have to register every actor with the animation sharing manager on BeginPlay, and unregister on EndPlay. Unregister is weirdly not available to blueprints, so we expose the function ourselves through C++;

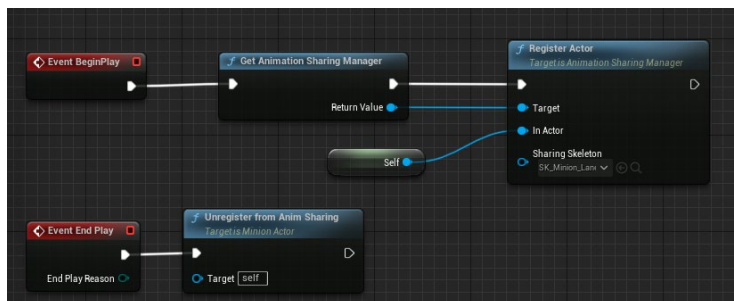


Figure 18: Registering and unregistering the actor with the animation sharing manager

Reference used to implement animation sharing: [27].

3. RESULTS

Results are only measured on one device, and each trace was only recorded once. This can lead to inaccurate results, but they will still present a general idea that can be insightful. We don't think more accurate measurements will heavily influence the conclusion.

You can find an excel sheet with all the recorded data here: <https://github.com/SimonSchaep/Research-UE5-Mass/blob/main/Results.xlsx>

Use the drop-down list in the top right of the master sheet to determine what amount of units you want to see data for.

You can then see the relevant frame times in the table.

The navigation and target acquisition costs in the simple battle simulator are difficult to distinguish, since they don't happen in order. We will present these frame costs combined under the target acquisition cost. So please be aware that the target acquisition cost also includes navigation in the case of the simple battle simulator.

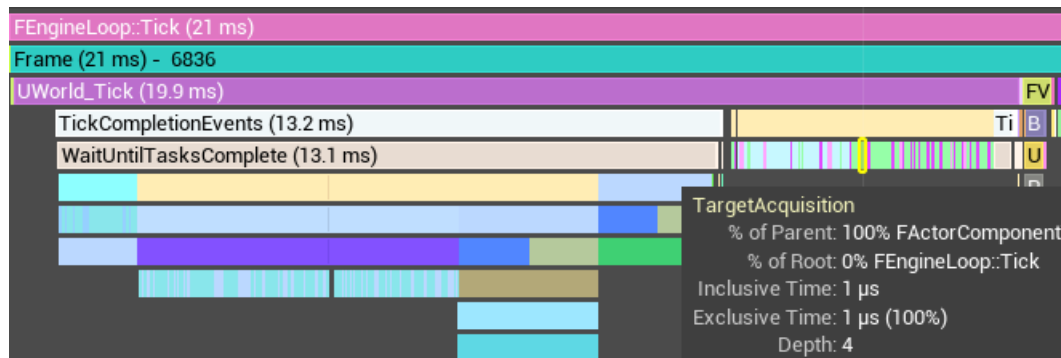


Figure 19: Screenshot from within Unreal Insights showing how component ticks are not grouped

For target acquisition with spatial partitioning, the time required to update the octree is also included in the target acquisition time.

References that were used for measuring: [23], [24], [46].

If you want to look at the traces recorded with Unreal Insights, that were used to extract this data, they are available here: https://drive.google.com/file/d/1dy27FoWe-HDYQR6C2MbmFpC7WUoz_FB9/view?usp=sharing

DISCUSSION

Let's first address some inconsistencies in the results when going to higher frame times. We took the frame snapshot after exactly 10 seconds of the trace start, but that means that differences in frame times can affect the amount of units that are still alive at 10 seconds, which means that frame time measurements could potentially be misleading.

We could solve that by taking a frame after only one second, but this might also make results inaccurate, since there will be different animations playing, density might be different for spatial partitioning, etc...

This also means that results with frame times above 100 will be much less useful.

We will be using charts to discuss the data, if you want to see the exact numbers we measured, please refer to the excel file with all the data here: <https://github.com/SimonSchaep/Research-UE5-Mass/blob/main/Results.xlsx>

1. FRAME TIME OVER UNIT COUNT

The game thread takes more frame time than the render thread and GPU in all cases, once we go past 800 vs 800 units. This means that our gameplay logic scales the worst over unit count, it is also our bottleneck and should be the focus of optimizations. We came to this conclusion by comparing Chart 1, Chart 3 and Chart 4.

On the chart below, we can see that the battle simulator with Mass unexpectedly performs worse than the simple battle simulator once we get to 6400 vs 6400 units.

We can also see that the addition of spatial partitioning seems to greatly improve the scaling of the frame time.

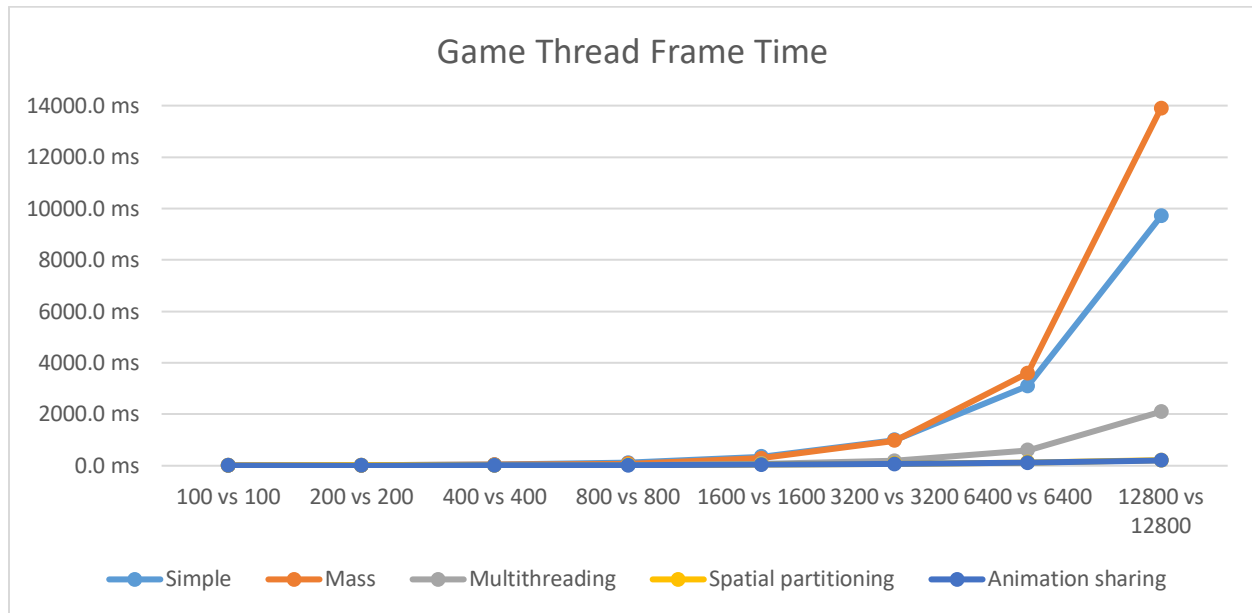


Chart 1: Game thread frame time over unit count

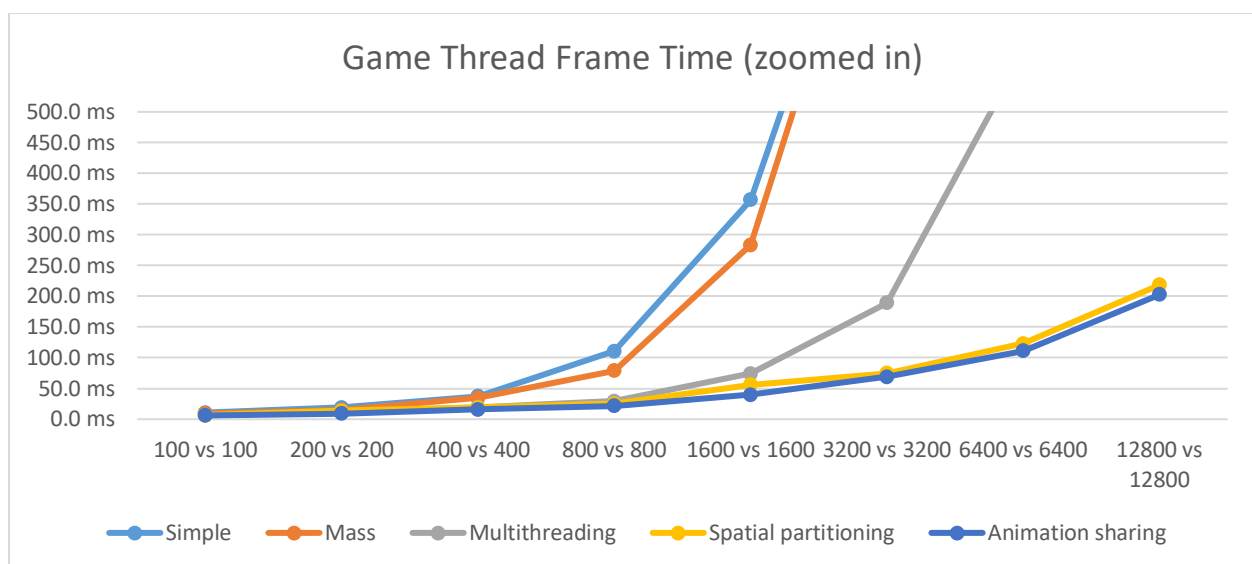


Chart 2: Game thread frame time over unit count (zoomed in)

In case of the render thread, it seems like Mass greatly reduces the frame time, as shown on the chart below. This is likely because of the instanced static meshes that are used for rendering at longer distances instead of actors. Something noticeable is that spatial partitioning seemingly improves rendering frame time. However, it is possible that because the frame time gets so high without spatial partitioning, there is a different amount of units alive at the time of the frame, which will of course influence rendering speed.

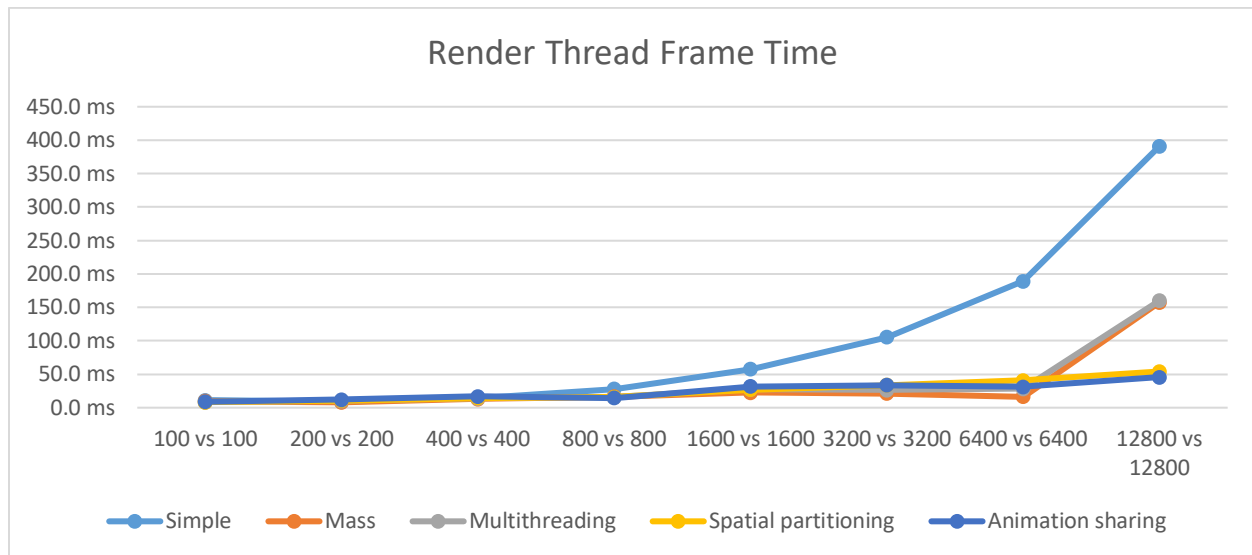


Chart 3: Render thread frame time over unit count

We can see on Chart 4 that the GPU frame time seems to vary a lot, seemingly randomly. The GPU time generally goes up when there are more units on screen. We can also see that Mass, or any other technique, doesn't have a clear obvious impact on GPU performance.

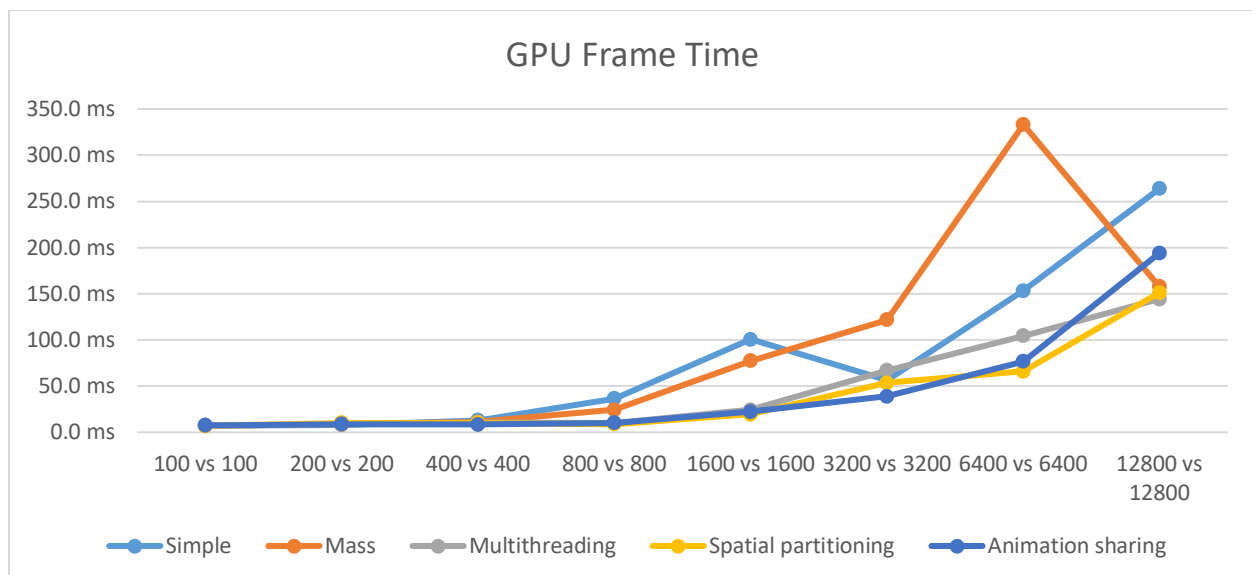


Chart 4: GPU frame time over unit count

For the simple battle simulator, we couldn't gather data for target acquisition and navigation separately, so both costs are included together under target acquisition. This makes it much more difficult to compare the simple battle simulator with the others.

We can see in Chart 5, that target acquisition seems to be scaling very badly, and is at least one of the reasons of why the game thread in general scales badly over unit count. The addition of multithreading seems to help a lot with the frame time, but doesn't improve the scaling. However, the addition of spatial partitioning seems to fully solve that issue.

It is noticeable that the Mass battle simulator has a worse frame time than the simple one, even with the simple one also including navigation. This could be caused by a mistake in our implementation or measurements, and would need to be further tested to find the reason.

In Chart 5, the spatial partitioning graph is behind the animation sharing one. For a zoomed in chart, look at Chart 6.

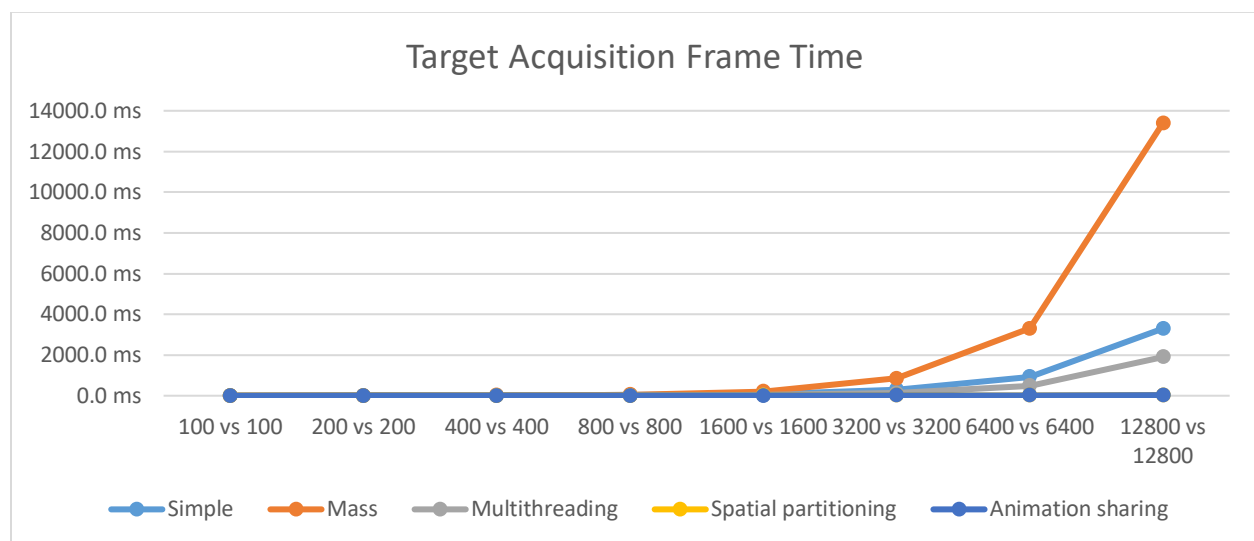


Chart 5: Target acquisition frame time over unit count

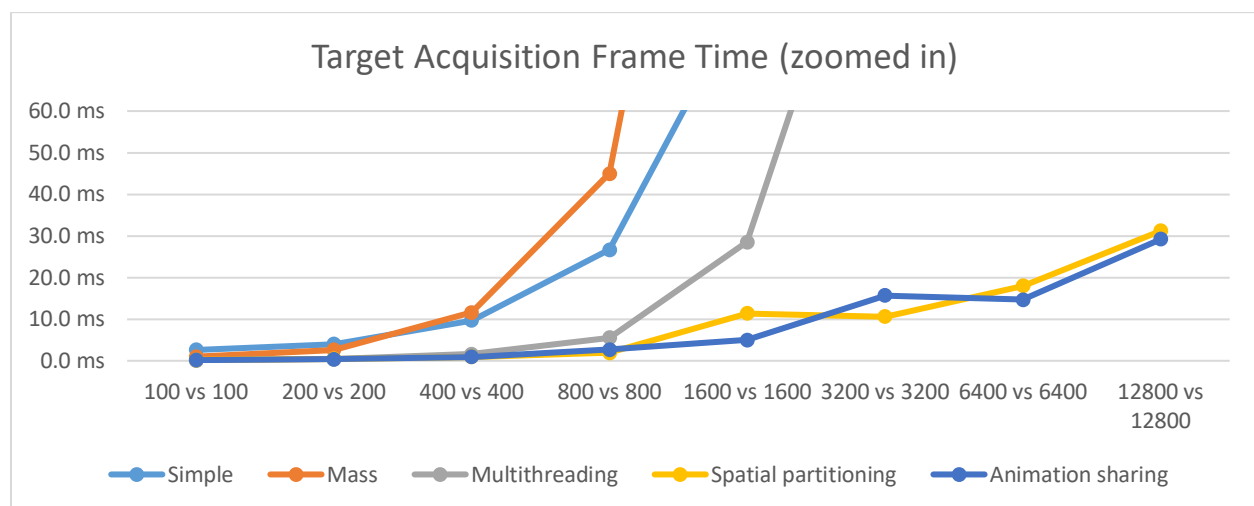


Chart 6: Target acquisition frame time over unit count (zoomed in).

The navigation frame time of the simple battle simulator was included in target acquisition time, since we couldn't distinguish those measurements. As a result it is excluded from the chart below.

On this chart, we can see that navigation seems to scale at roughly $O(n)$ time. Multithreading improved navigation frame time by quite a lot, but didn't affect the scaling.

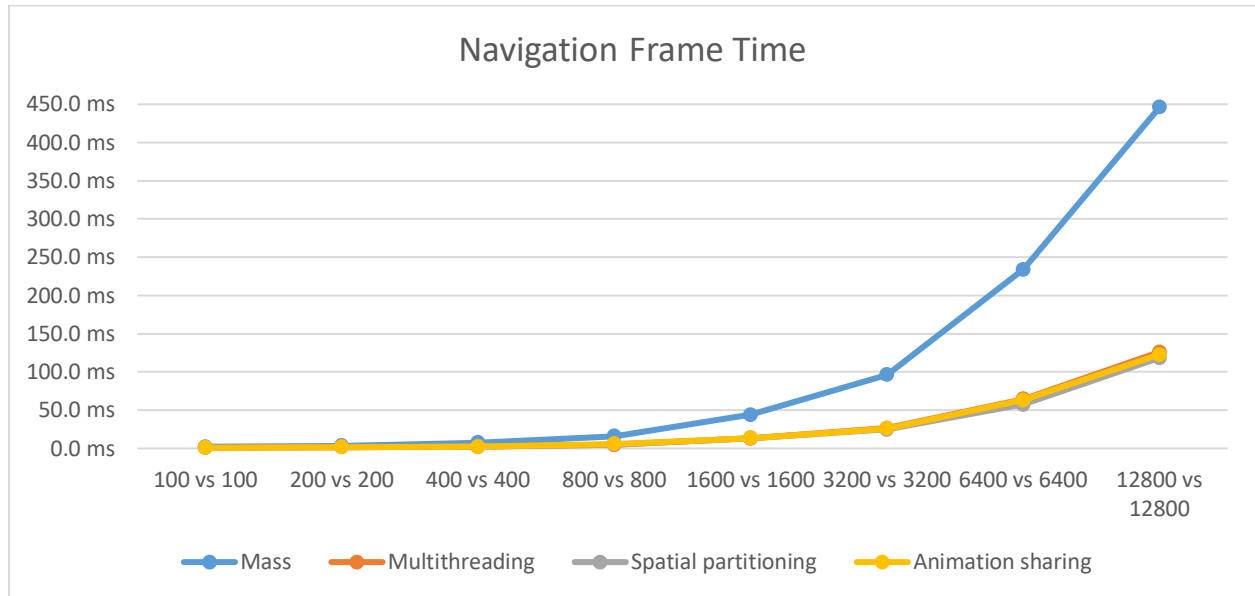


Chart 7: Navigation frame time over unit count

In Chart 8, we can see that the character movement component that was used in the simple battle simulator, seems to be very inefficient in general, even with only a few hundred units.

The movement in the Mass battle simulator is very fast, and the avoidance processor seems to be very optimized. It is not multithreaded however, so there is no difference caused by the addition of multithreading to the battle simulator.

Movement and avoidance take over 25 milliseconds in the final test, which means they might still need more optimizations if we want to run the battle simulator at 30 fps.

Multithreading could potentially be a sufficient solution.

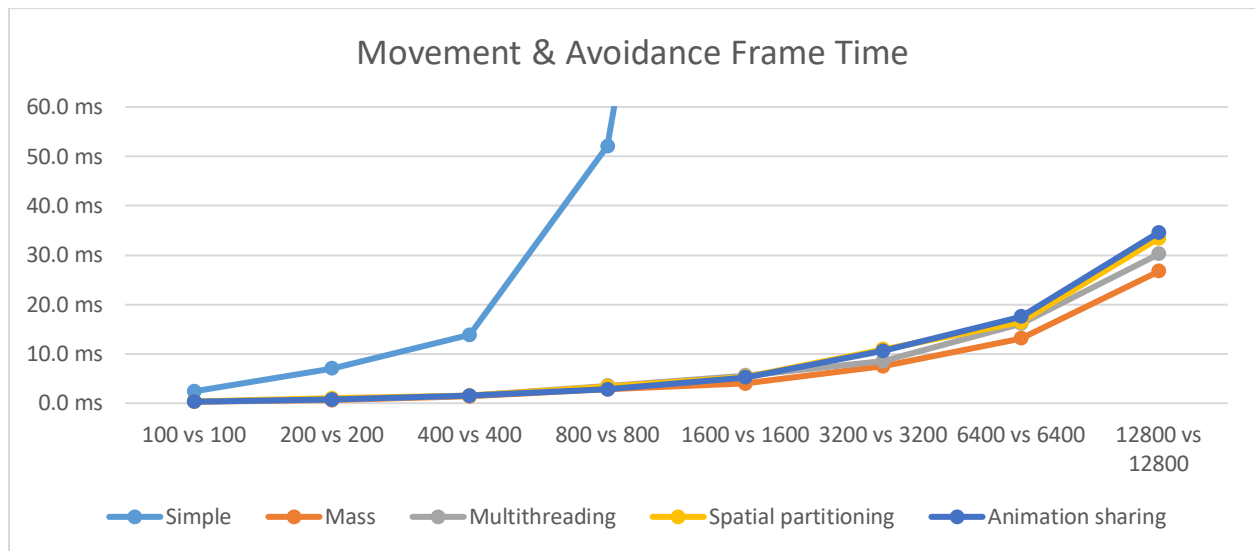


Chart 8: Movement and avoidance frame time over unit count

The time it takes to translate from Mass to the actors is noticeable enough to be worth analyzing it through Chart 9. In the first three tests, the frame time seems to steadily grow, after which it stabilizes. This is because at that point, most new units will be spawned outside of the range where actors get used for visualization.

The frame time in case of the animation sharing implementation deviates from the others. This might be because there is an extra step involved with the skeletal meshes copying data from the animation sharing manager. It could also be a measuring or implementation mistake.

Multithreading did not improve translation frame time, because we are not able to manipulate actor positions from a thread other than the game thread. This means we are not able to multithread this process at all.

Using vertex animations could potentially eliminate this entire cost, since we could use instanced static meshes to render everything in that case.

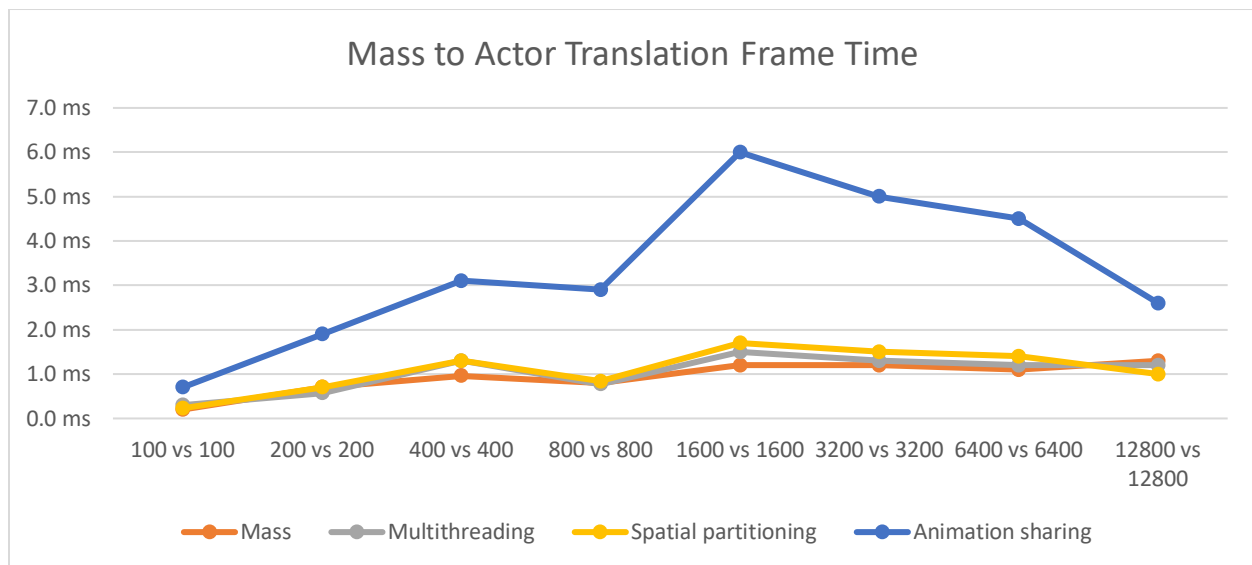


Chart 9: Mass to actor frame time over unit count

In Chart 10, we can see that animation frame times vary a lot, this might be because of the kind of animations that are playing at that specific frame, for example an attacking animations might be more costly than an idle one. Or perhaps it is more costly while the animations are blending between each other.

We can see that the simple battle simulator needs a lot more frame time for animations compared to the Mass implementations. This is because in Mass, the furthest away units are instanced static meshes and don't play any animations. This can sometimes be noticeable though, so the improved animation performance comes at a cost of quality in this case.

The animation sharing implementation greatly reduces animation frame time in all tests, which means that it works quite well.

Vertex animations might be a better solution than animation sharing, since it would delegate the animation work to the GPU and also solve the issue of far away units not having animations.

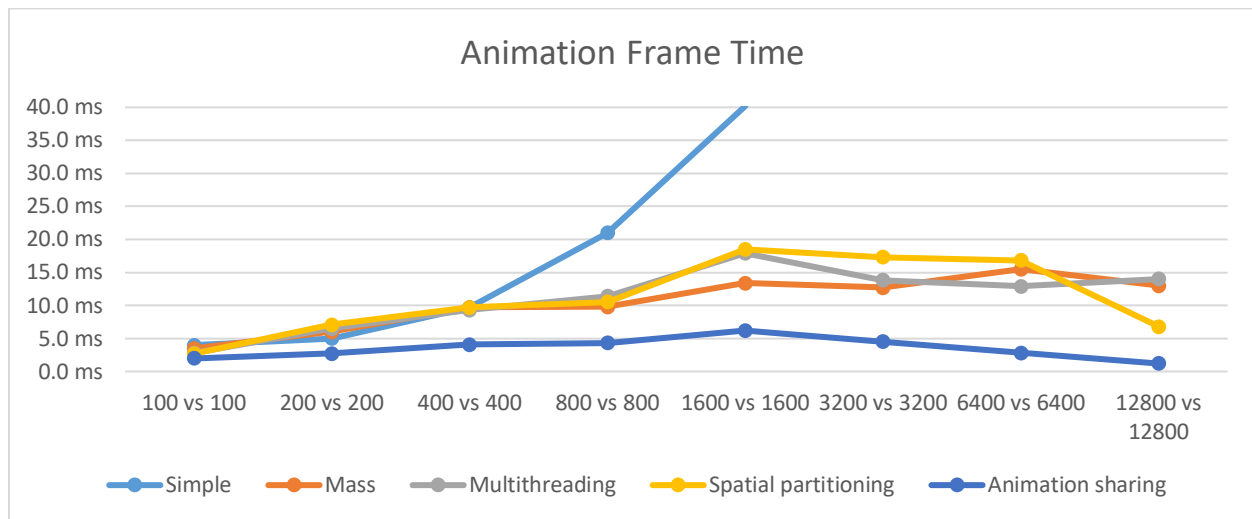


Chart 10: Animation frame time over unit count

2. GAME THREAD FRAME TIME COMPOSITION

Through the following charts, we will take a look at which elements take the most time in the game thread, since that is our biggest bottleneck. These measurements are from the 800 vs 800 tests, because that is about where the fps drops below 60 in all cases. You can see this

On Chart 11, we can see that the character movement component is clearly the biggest performance issue in the simple battle simulator. Creating a custom movement component and avoidance system might help with that. After the movement, target acquisition takes the most frame time.

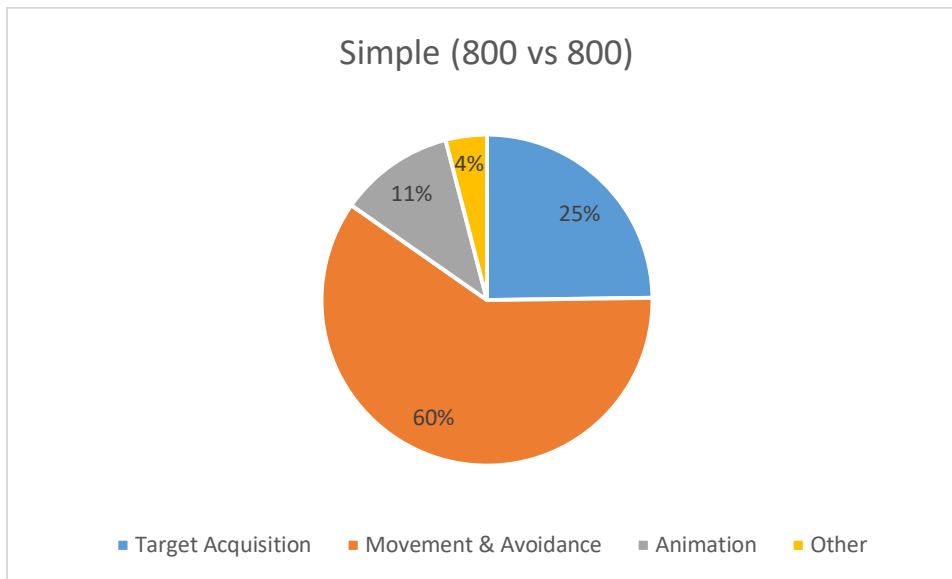


Chart 11: Game thread frame time distribution in the simple battle simulator with 800 vs 800 units

In the Mass battle simulator, target acquisition takes most of the frame time, which can be seen on Chart 12. It is also noticeable that the navigation takes a big chunk of frame time as well.

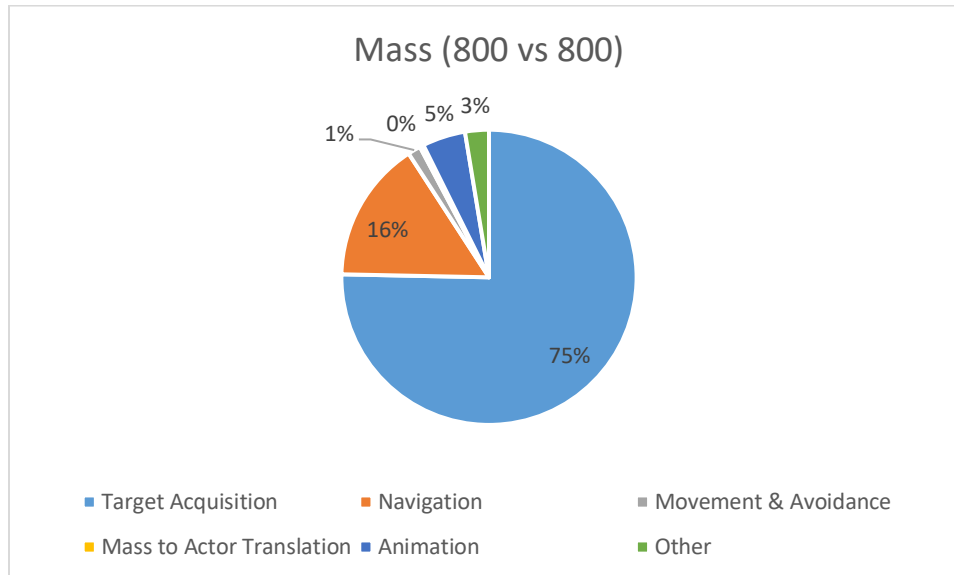


Chart 12: Game thread frame time distribution in the mass battle simulator with 800 vs 800 units

On the chart below, we can see that with the addition of multithreading, the distribution of frame time is much more even than before. However, target acquisition is still clearly the biggest chunk, while animation and navigation start to become very noticeable as well.

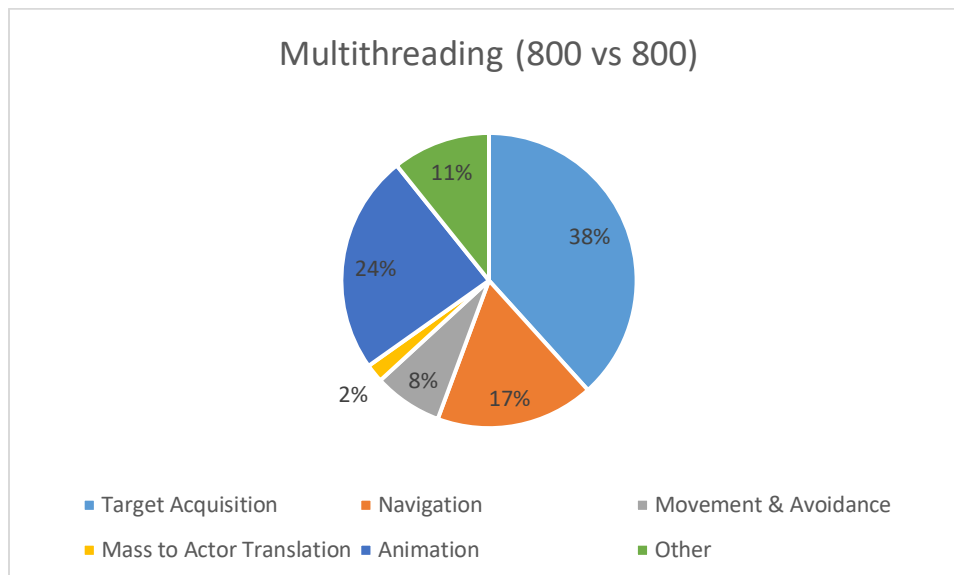


Chart 13: Game thread frame time distribution in the mass battle simulator with multithreading with 800 vs 800 units

Spatial partitioning resolves the scaling issue that target acquisition had. As a result, the biggest consumers of frame time are now animation and navigation. This can be seen in the chart below.

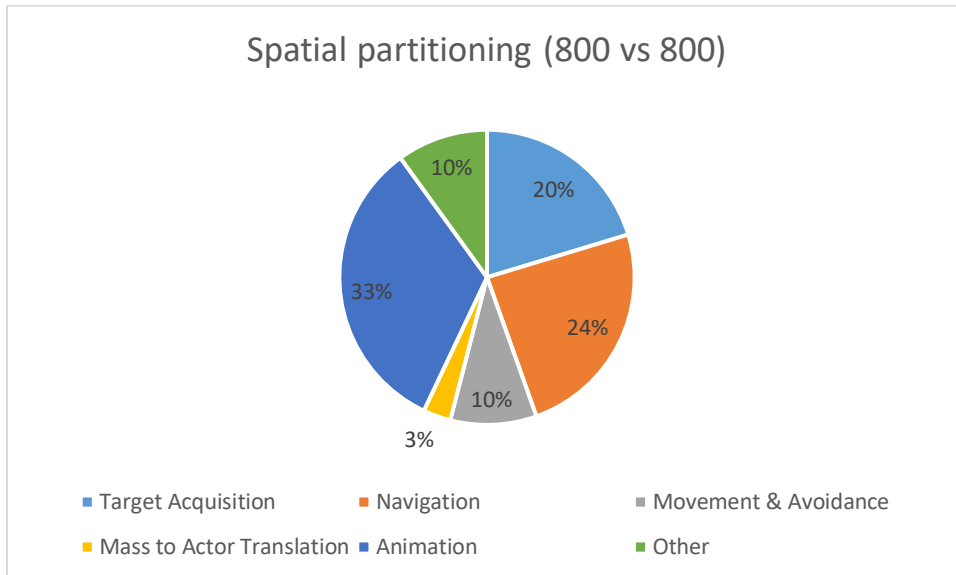


Chart 14: Game thread frame time distribution in the mass battle simulator with spatial partitioning with 800 vs 800 units

With our final addition of animation sharing, we drastically reduce the frame time that animation takes, as shown on the chart below. The newest game thread bottleneck has become the navigation.

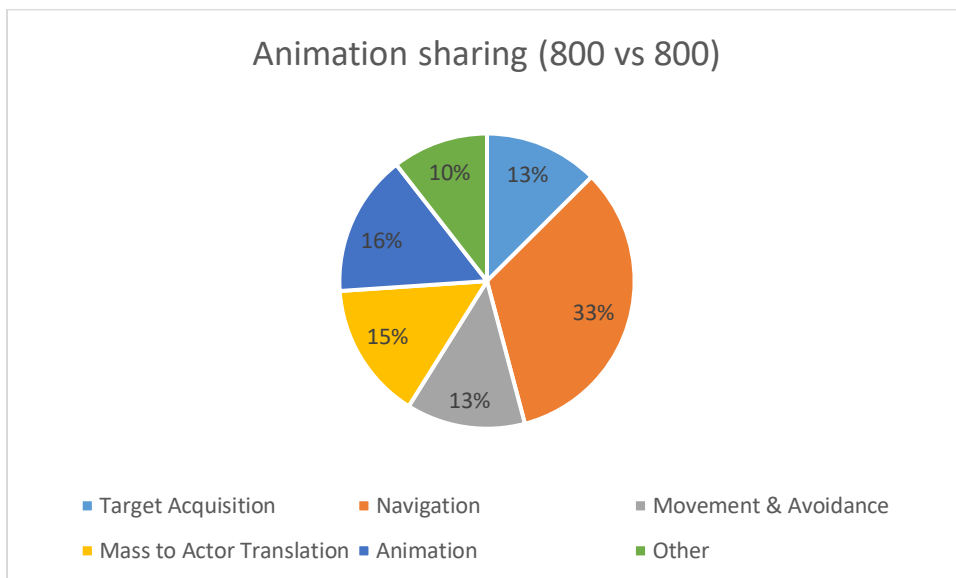


Chart 15: Game thread frame time distribution in the mass battle simulator with animation sharing with 800 vs 800 units

3. SUMMARY

With all these results we can now verify the validity of the hypotheses that were formed in the introduction.

The frame time on the game thread already reaches over 100 milliseconds in the test with 6400 vs 6400 units, with all our optimizations implemented. This means that we can conclude that with the optimization techniques we implemented, we cannot simulate 20 000 units at 30 fps.

We also don't reach this target on the render thread and GPU, so to reach 20 000 units at 30 fps, we would not only need to optimize the game thread, but also the render thread and GPU.

With our Mass implementation, game thread frame time was worse than in the simple battle simulator. However, all other aspect were improved.

Multithreading the target acquisition and navigation does improve overall performance when combined with Mass. However, it does not improve the scaling over unit count, which means it doesn't solve the fundamental issue with exponential algorithms like the target acquisition.

When using an octree spatial partitioning data structure for target acquisition, the scaling of the algorithm is no longer exponential. This in turn improves performance, especially with higher unit counts.

Animation sharing reduces the time needed to calculate animations by up to six times. While Mass also reduces animation frame time with higher unit counts.

CONCLUSION

While we didn't reach the target we expected when forming our hypotheses, we still gained some interesting insights from the results of our case study on how to optimize huge battles in a video game.

Firstly, we found out that the game thread is the primary bottleneck when simulating a large battle by comparing game thread, render thread and GPU frame times. However, when simulating over 6400 total units in the battle, the render thread and GPU will also need optimizations to reach a playable framerate.

Secondly, it is clear that spatial partitioning is very important to improve how well certain processes like target finding scale over unit count. As can be seen on Chart 5, this is especially noticeable when simulating more than 6400 total units.

Animation sharing also turned out to be a very effective optimization for animations. It generally improved animation frame time, making it a relatively small aspect of the game thread frame time, which you can see on Chart 15.

Using Mass, we were able to decrease frame time on almost all aspects of the simulation, except for target acquisition. We are not sure why target acquisition got worse with our implementation of Mass.

Mass also made it more convenient to implement other optimizations like multithreading.

We made an example of the usage of Mass available through this study, which other people can now use to learn how the system works and how to combine other optimization techniques with it.

If a game has different requirements than this case study, there is the possibility to do more specific optimizations. For example, if the battlefield does not have any obstacles, you could completely remove navigation. If it fits the battle, you could also group your units together in squads, which could also give a huge boost to game thread performance [58].

FUTURE WORK

There is a lot of potential future work that can be done on the basis of this study.

You could delve deeper into most aspects that were applied in the case study. For example, you could try out many different spatial partitioning structures and measure which ones have the biggest impact on performance in which situations. Or you could go deeper into Mass or other data oriented systems, to try and make the most optimal use of it.

This case study could also be expanded by continuing to optimize the battle simulator. There are two things that would probably have the biggest impact. One would be to optimize navigation, since that seems to be a big consumer of frame time. Another optimization would be to use vertex animations and only instanced static meshes to render the Mass entities. This could potentially even be combined with Niagara's rendering system to try and get the best possible rendering performance.

There could also be a study on how to add projectiles to the simulation. This is quite a big addition, since you might need hit detection, efficient spawning and cleaning up of entities, etc...

Another big topic that could be studied in a research project, is GPU programming. This is how the biggest battle simulator on the market, Ultimate Epic Battle Simulator, achieves millions of units in a single battle[4]. It is a very complex topic, but could definitely pay off.

CRITICAL REFLECTION

Overall, I am happy with what I accomplished during this research project. However, I did wish I had more time to fully optimize the battle simulator and reach my initial goal of 20 000 agents fighting each other.

I also would have liked to take more time to explore Mass further since I only scraped the surface, and there is lots more to learn about the system.

I underestimated how much work the CPU needs to do for animations and rendering as a whole. It would have helped to study some more literature about optimizing rendering.

I learned a lot about Mass and how a data oriented system can be implemented in a game engine. I would like to take the time to also look into Unity DOTS, and compare the two systems.

I also learned how to use the Unreal Engine source code to understand how their systems work, since that was my main source of information about Mass, because there are very few tutorials and documentation about more advanced Mass aspects.

LIST OF FIGURES

FIGURES

Figure 1: MassEntity archetype graph, it shows how archetypes are formed - MassEntity documentation (Epic Games)[9]	8
Figure 2: MassEntity processor graph, it shows how processors use EntityQueries to interact with archetypes - MassEntity documentation (Epic Games)[9].....	8
Figure 3: Screenshot of an army rendered with Niagara - Simulating Large Crowds In Niagara Unreal Engine (Epic Games) [25].....	11
Figure 4: Screenshot of a battle with 10 million units in Ultimate Epic Battle Simulator 2 - UEBS 2 The Making Of Pt1 (BrilliantGameStudios)[4]	13
Figure 5: Disabling component tick in the actor blueprint.....	18
Figure 6: Simple battle simulator animation state graph	19
Figure 7: Plugins required for using Mass in Unreal Engine 5	20
Figure 8: Example of a mass spawner configuration	21
Figure 9: Example of an EntityConfig data asset configuration.....	21
Figure 10: Visualization trait configuration.....	27
Figure 11: Draw call amount with old model Figure 12: Draw call amount with new model	28
Figure 13: LOD Settings after generating an LOD settings asset	29
Figure 14: Visualization of the octree, the blue boxes represent the nodes of the tree	31
Figure 15: Screenshot from within Unreal Insights to show the amount of frame time the animations take.....	33
Figure 16: Creation of the animation sharing manager	33
Figure 17: Configuration of the AnimationSharingSetup asset	34
Figure 18: Registering and unregistering the actor with the animation sharing manager	34
Figure 19: Screenshot from within Unreal Insights showing how component ticks are not grouped.....	35

CODE SNIPPETS

Code snippet 1: Acquiring the closest target	17
Code snippet 2: FArmyIdFragment definition. This is a simple example of how to define a fragment.....	22
Code snippet 3: TargetAcquisitionProcessor Execute functionality. A more complex example of a processor.....	23
Code snippet 4: Navigation mesh pathfinding in the navigation processor Execute function.....	24
Code snippet 5: Attack parameters shared fragment definition.....	25
Code snippet 6: DestroyEntitiesProcessor's three main functions	26
Code snippet 7: Replacing the normal for loop with a ParallelFor	29
Code snippet 8: Using a mutex to prevent simultaneous writing to the deferred command buffer	30

Code snippet 9: Finding the closest target with an octree	32
---	----

CHARTS

Chart 1: Game thread frame time over unit count.....	37
Chart 2: Game thread frame time over unit count (zoomed in)	37
Chart 3: Render thread frame time over unit count.....	38
Chart 4: GPU frame time over unit count.....	38
Chart 5: Target acquisition frame time over unit count	39
Chart 6: Target acquisition frame time over unit count (zoomed in)	39
Chart 7: Navigation frame time over unit count	40
Chart 8: Movement and avoidance frame time over unit count.....	41
Chart 9: Mass to actor frame time over unit count.....	42
Chart 10: Animation frame time over unit count.....	43
Chart 11: Game thread frame time distribution in the simple battle simulator with 800 vs 800 units.....	44
Chart 12: Game thread frame time distribution in the mass battle simulator with 800 vs 800 units	45
Chart 13: Game thread frame time distribution in the mass battle simulator with multithreading with 800 vs 800 units.....	45
Chart 14: Game thread frame time distribution in the mass battle simulator with spatial partitioning with 800 vs 800 units	46
Chart 15: Game thread frame time distribution in the mass battle simulator with animation sharing with 800 vs 800 units.....	46

REFERENCES

- [1] 'Artificial intelligence in video games', *Wikipedia*. Jan. 10, 2024. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Artificial_intelligence_in_video_games&oldid=1194670313
- [2] sophlogimo, 'What is the biggest number of soldiers you have seen in one TW battle?', r/totalwar. [Online]. Available: https://www.reddit.com/r/totalwar/comments/dgcrdc/what_is_the_biggest_number_of_soldiers_you_have/
- [3] 'Maximum number of troops on battlefield ? :: Total War: EMPIRE - Definitive Edition General Discussions'. [Online]. Available: <https://steamcommunity.com/app/10500/discussions/0/2765630416813779879/>
- [4] *10 MILLION Characters On Screen?! - Ultimate Epic Battle Simulator 2 The Making Of Pt1*, (Feb. 12, 2021). [Online Video]. Available: <https://www.youtube.com/watch?v=kpojDPlIjdQ>
- [5] 'CPU cache', *Wikipedia*. Dec. 29, 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=CPU_cache&oldid=1192515389
- [6] *CppCon 2014: Mike Acton 'Data-Oriented Design and C++'*, (Sep. 30, 2014). [Online Video]. Available: <https://www.youtube.com/watch?v=rX0ItVEVjHc>
- [7] *CppCon 2018: Stoyan Nikolov "OOP Is Dead, Long Live Data-oriented Design"*, (Oct. 26, 2018). [Online Video]. Available: <https://www.youtube.com/watch?v=vy8iQgmhbAU>
- [8] U. Technologies, 'DOTS - Unity's Data-Oriented Technology Stack'. [Online]. Available: <https://unity.com/dots>
- [9] 'MassEntity Overview'. [Online]. Available: <https://docs.unrealengine.com/5.2/en-US/overview-of-mass-entity-in-unreal-engine/>
- [10] 'Entity component system', *Wikipedia*. Dec. 20, 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Entity_component_system&oldid=1190943102
- [11] *The ECS Architecture - Performance in UE4*, (Feb. 06, 2021). [Online Video]. Available: <https://www.youtube.com/watch?v=kXd0VDZDSks>
- [12] 'Unreal Engine 5.0 Release Notes'. [Online]. Available: <https://docs.unrealengine.com/5.0/en-US/unreal-engine-5.0-release-notes/>
- [13] *Large Numbers of Entities with Mass in UE5 | Feature Highlight | State of Unreal 2022*, (Apr. 26, 2022). [Online Video]. Available: <https://www.youtube.com/watch?v=f9q8A-9DvPo>
- [14] *The Matrix Awakens: Generating a World | Tech Talk | State of Unreal 2022*, (Apr. 05, 2022). [Online Video]. Available: <https://www.youtube.com/watch?v=usJrcwN6T4I>
- [15] J. Keeling, 'Tutorial: Your First 60 Minutes with Mass', Epic Developer Community Forums. [Online]. Available: <https://forums.unrealengine.com/t/tutorial-your-first-60-minutes-with-mass/794314>
- [16] 'MassGameplay Overview'. [Online]. Available: <https://docs.unrealengine.com/5.2/en-US/overview-of-mass-gameplay-in-unreal-engine/>
- [17] 'Mass Avoidance Overview'. [Online]. Available: <https://docs.unrealengine.com/5.2/en-US/mass-avoidance-overview-in-unreal-engine/>
- [18] Symphonym, 'Choosing a spatial partitioning structure', r/gamedev. [Online]. Available: https://www.reddit.com/r/gamedev/comments/22lwg5/choosing_a_spatial_partitioning_structure/
- [19] M. Kelleghan, 'August 12 and 1997, 'Octree Partitioning Techniques'', Game Developer. [Online]. Available: <https://www.gamedeveloper.com/programming/octree-partitioning-techniques>
- [20] 'Space partitioning', *Wikipedia*. Oct. 16, 2022. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Space_partitioning&oldid=1116447126
- [21] Bob Nystrom, 'Spatial Partition · Optimization Patterns · Game Programming Patterns'. [Online]. Available: <https://gameprogrammingpatterns.com/spatial-partition.html>

- [22] 'UE5 Multithreading With FRunnable And Thread Workflow – Unreal C++ API'. [Online]. Available: <https://store.algosyntax.com/tutorials/unreal-engine/ue5-multithreading-with-frunnable-and-thread-workflow/>
- [23] [UE5] Understanding Render Thread and Animation Thread in Unreal Engine, (Mar. 06, 2023). [Online Video]. Available: <https://www.youtube.com/watch?v=RRwNIntV10I>
- [24] Maximizing Your Game's Performance in Unreal Engine | Unreal Fest 2022, (Nov. 03, 2022). [Online Video]. Available: <https://www.youtube.com/watch?v=Gulav71867E&list=PLGAgBU0oz5KZXHvaHAgas90IGBKJqsgzV>
- [25] Simulating Large Crowds In Niagara | Unreal Engine, (Jan. 15, 2021). [Online Video]. Available: <https://www.youtube.com/watch?v=CqXKSyAPWZY>
- [26] 'Animation Budget Allocator'. [Online]. Available: <https://docs.unrealengine.com/5.3/en-US/animation-budget-allocator-in-unreal-engine/>
- [27] 'Animation Sharing Plugin'. [Online]. Available: <https://docs.unrealengine.com/5.3/en-US/animation-sharing-plugin-in-unreal-engine/>
- [28] Stephen Phillips, 'Baking out vertex animation in editor with AnimToTexture | Community tutorial', Epic Developer Community. [Online]. Available: <https://dev.epicgames.com/community/learning/tutorials/daE9/unreal-engine-baking-out-vertex-animation-in-editor-with-animtotexture>
- [29] R. Banerjee, 'Answer to "What is the difference between OpenCL and OpenGL's compute shader?"', Stack Overflow. [Online]. Available: <https://stackoverflow.com/a/15868564>
- [30] C. Rau, 'Answer to "What is the difference between OpenCL and OpenGL's compute shader?"', Stack Overflow. [Online]. Available: <https://stackoverflow.com/a/15874988>
- [31] Maiss, 'What is the difference between OpenCL and OpenGL's compute shader?', Stack Overflow. [Online]. Available: <https://stackoverflow.com/q/15868498>
- [32] 'Compute Shader - OpenGL Wiki'. [Online]. Available: https://www.khronos.org/opengl/wiki/Compute_Shader
- [33] stevewhims, 'Compute Shader Overview - Win32 apps'. [Online]. Available: <https://learn.microsoft.com/en-us/windows/win32/direct3d11/direct3d-11-advanced-stages-compute-shader>
- [34] 'CUDA', Wikipedia. Jan. 05, 2024. [Online]. Available: <https://en.wikipedia.org/w/index.php?title=CUDA&oldid=1193733902>
- [35] 'CUDA'. [Online]. Available: <https://docs.unrealengine.com/5.1/en-US/API/Runtime/CUDA/>
- [36] Brad Nemire, 'CUDA Spotlight: GPU-Accelerated Agent-Based Simulation of Complex Systems', NVIDIA Technical Blog. [Online]. Available: <https://developer.nvidia.com/blog/cuda-spotlight-gpu-accelerated-agent-based-simulation-complex-systems/>
- [37] 'Fast Large-Scale Agent-based Simulations on NVIDIA GPUs with FLAME GPU', NVIDIA Technical Blog. [Online]. Available: <https://developer.nvidia.com/blog/fast-large-scale-agent-based-simulations-on-nvidia-gpus-with-flame-gpu/>
- [38] 'OpenCL', Wikipedia. Oct. 20, 2023. [Online]. Available: <https://en.wikipedia.org/w/index.php?title=OpenCL&oldid=1181029989>
- [39] 'OpenCL - The Open Standard for Parallel Programming of Heterogeneous Systems', The Khronos Group. [Online]. Available: <https://www.khronos.org/opencl/>
- [40] 'OpenCL Guide', Github. [Online]. Available: <https://github.com/KhronosGroup/OpenCL-Guide>
- [41] Jeremy Frank, 'Simple compute shader with CPU readback | Community tutorial', Epic Developer Community. [Online]. Available: <https://dev.epicgames.com/community/learning/tutorials/WkwJ/unreal-engine-simple-compute-shader-with-cpu-readback>

- [42] anonymous_user_526a5b67, 'How to use CUDA file in UE4?', Epic Developer Community Forums. [Online]. Available: <https://forums.unrealengine.com/t/how-to-use-cuda-file-in-ue4/90965>
- [43] SCIENT, '[C++][CUDA][UE4]CUDA 関数を Unreal Engine 4 で用いる - サイアメント技術メモ'. [Online]. Available: https://www.sciement.com/tech-blog/c/cuda_in_ue4/
- [44] anonymous_user_968689a2, 'OpenCL Plugin', Epic Developer Community Forums. [Online]. Available: <https://forums.unrealengine.com/t/opengl-plugin/18426>
- [45] 'Paragon (video game)', *Wikipedia*. Dec. 25, 2023. [Online]. Available: [https://en.wikipedia.org/w/index.php?title=Paragon_\(video_game\)&oldid=1191788101](https://en.wikipedia.org/w/index.php?title=Paragon_(video_game)&oldid=1191788101)
- [46] *New Features of Insights: Unreal Engine's Built In Profiling Tools | Inside Unreal*, (Mar. 30, 2023). [Online Video]. Available: https://www.youtube.com/watch?v=af_M38Z325I&list=PLGAgBU0oz5KZXHvaHAgas90IGBkJqsgzV
- [47] 'MassAI, Mass Crowd, State Tree with Custom Character | UE5 | Community tutorial', Epic Developer Community. [Online]. Available: <https://dev.epicgames.com/community/learning/tutorials/0be9/unreal-engine-massai-mass-crowd-state-tree-with-custom-character-ue5>
- [48] *UE5 MassAI Crowd in your project step by step guide Part 1*, (May 08, 2022). [Online Video]. Available: <https://www.youtube.com/watch?v=bVvYn7dEqMA>
- [49] *UE5 MassAI Crowd in your project step by step guide Part2 AnimationBlueprint*, (Jun. 29, 2022). [Online Video]. Available: <https://www.youtube.com/watch?v=4-qcSrAxGcC&list=PLKNsvWABWRceGFLyCWBz33f1aw4WjyYGs&index=3>
- [50] *UE5 Tutorial MassAI Crowd in any project in less then 15 mins.*, (Jul. 03, 2022). [Online Video]. Available: https://www.youtube.com/watch?v=2LvUB3_PAhI
- [51] *Unreal Engine 5 Mass Crowd Ai Tutorial*, (May 03, 2022). [Online Video]. Available: https://www.youtube.com/watch?v=w_6LGyACVz0
- [52] *Unreal Engine 5 MassEntity Plugin Demo*, (Jul. 11, 2022). [Online Video]. Available: <https://www.youtube.com/watch?v=0dwgnH3SoC0>
- [53] 'Unreal Engine 5.1 Release Notes'. [Online]. Available: <https://docs.unrealengine.com/5.1/en-US/unreal-engine-5.1-release-notes/>
- [54] 'Multithreading and Performance in Unreal', Epic Developer Community Forums. [Online]. Available: <https://forums.unrealengine.com/t/multithreading-and-performance-in-unreal/1216417>
- [55] 'TOctree2'. [Online]. Available: <https://docs.unrealengine.com/5.3/en-US/API/Runtime/Core/Math/TOctree2/>
- [56] 'TOctree_DEPRECATED'. [Online]. Available: https://docs.unrealengine.com/5.3/en-US/API/Runtime/Core/Math/TOctree_DEPRECATED/
- [57] 'FSparseDynamicOctree3'. [Online]. Available: <https://docs.unrealengine.com/5.1/en-US/API/Plugins/GeometricObjects/Spatial/FSparseDynamicOctree3/>
- [58] Phantomx1024, 'Optimizations For High Unit Count AI', *r/unrealengine*. [Online]. Available: https://www.reddit.com/r/unrealengine/comments/dmockq/optimizations_for_high_unit_count_ai/

ACKNOWLEDGEMENTS

I want to thank the lecturer of the Grad Work course, Alexander Deweppe, for giving insightful lectures on the topic of creating a research project.

I also want to thank my supervisor Alex Vanden Abeele, who organized several feedback sessions where we would discuss the state of the project. He also gave lots of useful feedback throughout the entire study.

I want to thank my coach Marijn Verspecht. For assisting with the general outline of the Grad Work course and giving feedback on my test presentation.

And I want to thank everyone at Howest and Digital Arts and Entertainment for giving me the opportunity to develop the skills that were needed to complete this project.

I also want to thank my family and friends for their support during this project.

APPENDICES

GitHub repository:

<https://github.com/SimonSchaep/Research-UE5-Mass>

Source code:

<https://github.com/SimonSchaep/Research-UE5-Mass/tree/main/UnrealProjects>

Excel file with results from the case study:

<https://github.com/SimonSchaep/Research-UE5-Mass/blob/main/Results.xlsx>

Trace files used to extract the data for the results:

https://drive.google.com/file/d/1dy27FoWe-HDYQR6C2MbmFpC7WUoz_FB9/view?usp=sharing