IT UNIVERSITY OF COPENHAGEN

MSc In Games

# High-Performance Game Programming (Autumn 2025)

Fabio Mangiameli (fabm@itu.dk)
Student number: 23242

Tom Pries (tpri@itu.dk)
Student number: 21791

19th of December 2025, Copenhagen, Denmark

# Introduction

For our simulation, we envisioned a space battle between an army of small agile spaceships, which we will refer to as *Fighters*, and a few tanky big spaceships, called *Star Destroyers*. We want the army of Fighters to behave like boid swarms, avoid the tanky Star Destroyers and asteroids, form groups, and attack the Star Destroyers based on parameters that are exposed to the user.
The simulation is endlessly running, with Fighters and Star Destroyers spawning whenever the count drops below the amount entered by the user. The same applies to asteroids, which float around in space, colliding with other asteroids and space ships. The exposed parameters allow the user to scale the simulation and play around with the crowd dynamics of our boid swarm implementation and how they do in the fight against the Star Destroyers. Our target was to create a performant simulation where 2000 Fighters are in a battle against at least 2 Star Destroyers. With this amount of Fighters we are able to see an interesting flocking simulation  and with two Star Destroyers, grouping can be observed in the Fighter army as they will split up and attack both Destroyers. We aimed to achieve a frame rate as high as possible for this scenario.

## Controls

| Input | Action |
|---|---|
| WASD | Move camera |
| Mouse Movement | Rotate camera |
| Left Mouse Button | Increase camera speed |
| F11 | Toggle settings |
| F12 | Toggle performance stats |

*Table 1: Simulation input options*

## Adjustable Settings

| Variable | Effect |
|---|---|
| **General Settings** | |
| Running Type | Switch between running on Main Thread, Scheduled and Scheduled Parallel |
| **Fighter Settings** | |
| Spawn Amount | Amount of Fighters spawned in the simulation |
| Min Speed | Minimum movement speed of a Fighter |
| Max Speed | Maximum movement speed of a Fighter |
| Min Rotation Speed | Minimum rotation speed of a Fighter |
| Max Rotation Speed | Maximum rotation speed of a Fighter |
| Detection Radius | Radius in which Fighters can detect other Fighters/Obstacles |
| Alignment Factor | Scaling for alignment direction (average forward direction of all fighters in swarm) |
| Crowding Factor | Scaling for crowding direction (direction to swarm center) |
| Neighbour Counter Force Factor | Scaling for neighbour direction (average directions from neighbour to Fighter) |
| Avoidance Factor | Scaling for avoidance direction (average directions from obstacles to Fighter) |
| Target Trend Factor | Scaling for target direction (direction from Fighter to target) |
| Target Min Distance | Minimum distance to target after which a Fighter will start retreating to find a new target |
| Fire Cooldown | Cooldown between firing shots |
| **Star Destroyer Settings** | |
| Spawn Amount | Amount of Star Destroyers spawned in the simulation |
| Health | Health points of a newly spawned Destroyer |
| Speed | Star Destroyer Movement Speed |
| Movement Radius | Radius in which the new movement target will be determined |
| Projectile Radius | Radius in which projectiles will do damage (No visual effect) |

*Table 2: Simulation settings editable in the UI*

# Implementation

## Fighter Implementation

Fighters are the little space crafts in the simulation and can be spawned by adjusting the spawn amount in the settings. We tested counts from 500 up to 10000. These small spacecraft are designed to seek out and attack the larger cruisers, attempting to destroy them through coordinated behavior. Their movement is governed by a flocking simulation, which produces smooth, collective motion and enhances both the realism and dynamism of their behavior. Figure 1 shows how the Fighters are flying in the simulation arranged as a swarm.

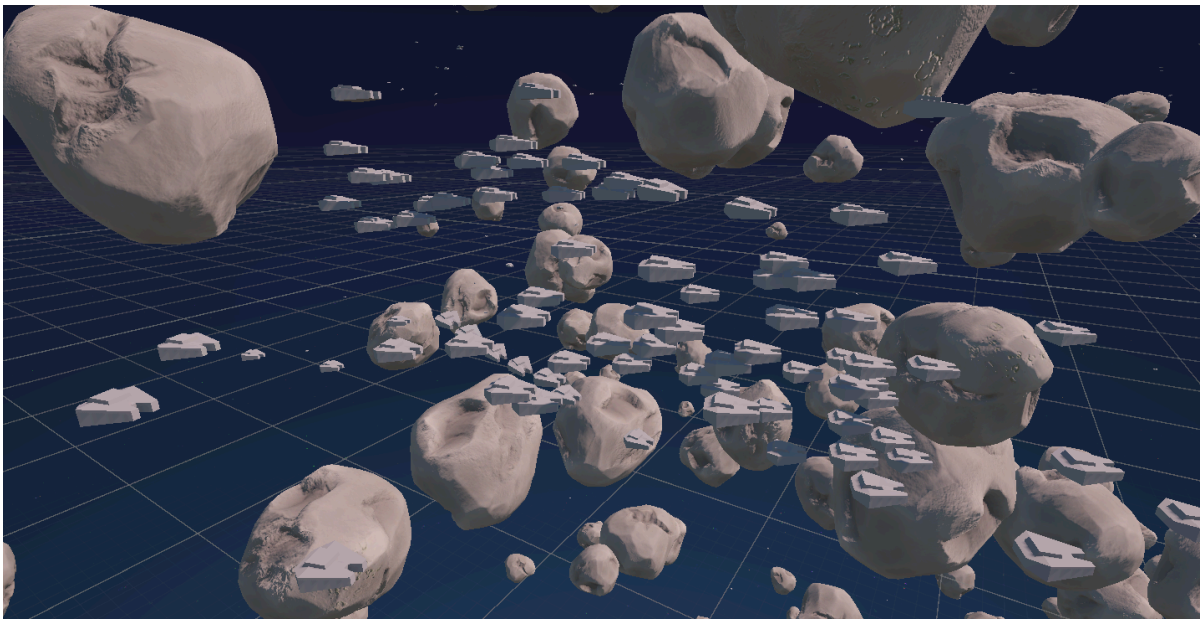


*Figure 1: Fighters flying in a swarm through the simulation*

### Fighter Entity Structure

The Fighter Entity is a collection of multiple Components and DynamicBuffers. The setup we construct in the *Baker* of the *FighterAuthoring* is as follows:

- *FighterComponent*
    - Stores data needed for fighter movement, shooting and its current state
- *PhysicsCustomTags*
    - Needed for physics queries in systems
- *HealthComponent*
    - Stores the amount of health left for a fighter.
- *DynamicBuffer<NearbyFighterElement>*
    - Stores all Fighters nearby
- *DynamicBuffer<AvoidingEntityBufferElement>*
    - Stores all obstacles which need to be avoided
- *DynamicBuffer<HitBufferElement>*
    - Stores the hits created through the shooting of the Fighter

Components added on the prefab are:

- *MeshRenderer*

- *MeshFilter*
- *SphereCollider*

## Fighter Systems

The logic of every Fighter is controlled through multiple systems, each covering a different part of the simulation. These systems handle movement, shooting, nearby object detection, and spawning.

**Spawning**
Fighters are spawned together with Star Destroyers in the ShipSpawnSystem. The system calculates a delta to determine whether Fighters are missing compared to the maximum allowed number. If Fighters are missing, the system spawns the required amount. During spawning, a random transform is generated and applied to each Fighter. The position is chosen within a bounding box, the dimensions of which can be configured through the settings.

**Movement**
The logic responsible for Fighter movement is processed by the following systems:

- *FighterAvoidanceSystem*
- *FighterFindTargetSystem*
- *FighterSwarmSystem*
- *FighterMovementSystem*

All of these systems contribute to determining the movement for the next frame. The first three systems calculate values that are stored on the *FighterComponent*. These values are then used by the *FighterMovementSystem* to calculate the final movement. Further in the report is a more detailed description of how movement is calculated.

**Shooting**
Fighters need to be able to shoot laser blasts at cruisers in order to destroy them. This logic is handled by the FighterShootingSystem.

This is a simple system that fires a raycast in the forward direction of the Fighter and checks if it collides with a Star Destroyer. After a successful hit, the Fighter will store a *HitBufferElement* and add it in the buffer. Later this will subtract one health point from the Star Destroyer.

**Nearby Object Detection**
Fighters need to detect nearby objects such as other Fighters or asteroids. This logic is handled by the *NearbySearchSystem*. For the flocking simulation to work, each Fighter stores its neighbors within a defined radius. This value is called *NeighbourDetectionRadius* and can be adjusted through the UI settings. To find nearby Fighters, the system calls the *CalculateDistance* function on the *CollisionWorld*, which is part of the *PhysicsWorld*. This function returns a *NativeList* of *DistanceHit* objects that can be iterated over. From each *DistanceHit*, the hit Entity can be retrieved. The system then checks whether the Entity has the *PhysicsCustomTags* component. If it does, the tag is evaluated. Fighters have their own tag called “Fighter”, which allows the system to identify them and store a reference in the *DynamicBuffer* of *NearbyFighterElement*. In the same loop, the system also checks for entities with the *Avoid* tag. These include asteroids and Star Destroyers. If one of these objects is detected, a reference is stored in the *DynamicBuffer* of *AvoidingEntityBufferElement*.

## Fighter Movement Logic

The fighters use a flocking simulation to stick together and avoid obstacles. Multiple systems are required to apply direction vectors to each ship in order to create a resulting velocity. To achieve this flocking behavior, several factors are taken into account to calculate the final velocity vector.

**Alignment Direction**

Each fighter calculates an average alignment direction based on the forward directions of nearby fighters. These neighbors are stored in the *NeighbourBuffer* referenced by the entity.

$$d_{align} = \sum_{n}(d_n)$$

Here $d_n$ is the forward direction of a neighbor.

**Avoidance Direction**

Obstacles are registered in a Dynamic Buffer on the entity. When an obstacle is detected, a counter force is calculated to create an avoidance direction. This direction is determined from the hit position in the query space of the sphere used to detect obstacles and the position of the obstacle, such as Star Destroyers or asteroids. For every obstacle $o$ in the entity's avoidance buffer, the calculation is:

$$d_{avoid} = \sum_{o} (d_o \times w_o)$$

Where $d_o$ is the normalized direction from the hit position to the fighter position, and $w_o$ is a factor based on the distance between the two.

**Crowding Direction**

To keep fighters together, the crowd center is calculated as the average position of all neighbors. The direction to this center is then calculated from the fighter's position.

$$d_{crowd} = \frac{\sum_{n}(p_n)}{N} - p$$

Where $p$ equal the fighter position, $p_n$ is the position of a neighbour and $N$ is the amount of neighbours.

**Neighbour Direction**

While fighters should stick together, they also need to avoid colliding with each other. A separate neighbor force direction is calculated for this purpose, distinct from the avoidance vector because it is weighted differently in the final movement calculation. This direction is the average vector pointing away from each neighbor:

$$d_{neighbour} = \sum_{n} (p - p_n)$$

Where $p$ equal the fighter position and $p_n$ is the position of a neighbour.

**Target Direction**

Fighters select a target based on their state, either retreating or attacking. In the attacking state, the fighter searches for the closest Star Destroyer and sets it as the target. While attacking, if the fighter reaches a minimum distance to the target, it calculates a retreat position at a specified distance in the

forward direction and sets this as the new target. The target direction is simply the vector from the fighter to the target position:

$$d_{target} = p_{target} - p$$

Where $p$ equal the fighter position and $p_{target}$ is the position of the target.

**Final Movement Direction**
Once all the individual directions are calculated, the final movement direction is computed in the FighterMovementSystem:

$$D = \sum_i \left( \frac{d_i}{|d_i|} \times w_i \right)$$

Here, $d_i$ represents one of the direction vectors such as $d_{avoid}$ defined above, and $w_i$ is the corresponding weight. The direction values are normalized before being multiplied with their assigned weight. These weights are adjustable through the UI to control the influence of each direction on the final movement vector.

From this point on the velocity and rotation speed can be calculated. Depending on the difference between the angle of the Fighter forward direction and the final movement direction, a linear speed and rotation speed is calculated. The target rotation is a quaternion calculated with *LookRotationSafe* and the movement direction and the up vector as an input. To gain the new rotation, a spherical interpolation between the current rotation and the target rotation is done based on the rotation speed variable. Once the rotation is applied, the velocity is calculated with the forward direction of the Fighter forward direction multiplied with the linear speed variable and the delta time.

# Star Destroyer Systems

## Star Destroyer Entity Structure

The Star Destroyer Entity is a collection of multiple Components and a DynamicBuffer. The setup we construct in the *Baker* of the *StarDestroyerAuthoring* is as follows:

- *StarDestroyerComponent*
    - Stores the properties required for moving the Star Destroyer
- *PhysicsCustomTags*
    - Needed for physics queries in systems
- *HealthComponent*
    - Stores the amount of health left for a Star Destroyer.
- *TargetEntityComponent*
    - Used to detect a Star Destroyer as a target for the Fighters

Components added to the prefab:

- *MeshRenderer*
- *MeshFilter*
- *CustomCollider*

## Movement System

Since it takes Fighters a long time to destroy a Star Destroyer, we had to come up with a solution that prevents Star Destroyers from leaving the battlefield, i.e., moving out of the predefined bounds. The straightforward solution would be to randomly pick a position within the bounding box, move the Star Destroyer towards that point, and select a new random point once it is reached. This, however, would result in very hard-coded, unnatural-looking movements.
Therefore, we opted to use a quadratic Bezier curve for the movement: We assign each Star Destroyer three random points upon spawning. These three points are used to calculate a Bezier curve. We interpolate this curve with a progress parameter that increases every update based on the speed of the Star Destroyer; the resulting position will be the Star Destroyer's new position. Once the end of the current Bezier curve is reached, we need three new points to calculate a new Bezier curve. Given that we do not want the Star Destroyer to jump between positions, the starting point of the Bezier curve will be the old endpoint. The second point of the new curve is calculated by subtracting the difference between the endpoint and the former second point from the endpoint, as seen in the Figure below.

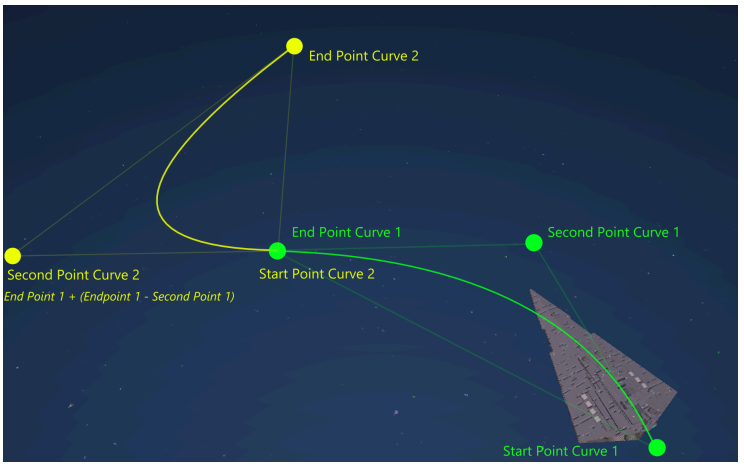



*Figure 2: Movement of Star Destroyers*

This way, the Star Destroyer's initial direction does not change when we swap curves. The new endpoint will be a random point. During this whole process, we keep the initial y-position of the Star Destroyer, such that it only moves in the xz-plane.

## Firing System

In order to make the Star Destroyers shoot at the fighters, we added Canon entities on the ship. These canons are either canons that are on top or below the ship, as can be seen in the figure below.

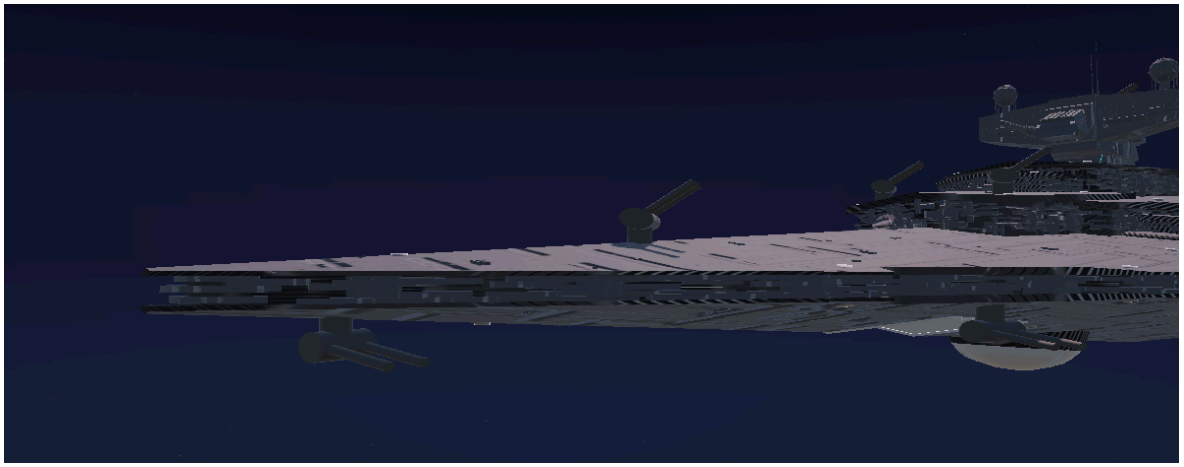


*Figure 3: Top and bottom Canon Entities on the Star Destroyer*

These canons operate by first selecting a target, then rotating towards that target, and finally shooting in the target's direction by spawning a laser entity. We iterated over how the canons should select a target. Initially, we collected the Local- and LocalToWorld transforms of each fighter in two native arrays, and passed them to the “OrientateCanonsJob” where we iterated over every transform and saved the one that is closest to the canon. There was an additional check that checked whether the y-position of the fighter was higher or lower than the canons. Based on whether the canon is a top or bottom canon, the target is valid or invalid. This is done to prevent a lower canon from shooting through the Star Destroyer because it is targeting a fighter that is located behind the Star Destroyer. This simple y-check is a simple and cheap solution, compared to testing if the Star Destroyer is in the way with a raycast. After the iteration over all fighters is complete, for the last part of the job, the canon is rotated in the direction of the target by the canon's rotation speed. If the canon’s forward direction is within an error margin of 10 degrees from the target direction, the “IsAimingAtTarget” flag is set.

Profiling our simulation showed us that this job took the longest out of all our jobs. We therefore eventually opted for a different approach. Instead of iterating over all fighters, we randomly look at a fixed number of fighters (10), and if one of them satisfies the y-check condition, it is the target of the canon. This reduces the complexity of the job from O(*N*) to O(*1*) and additionally introduces some randomness, which is a nice side effect as now different cannons aim at different fighters.
Then we spawn a laser entity and a VFX entity. We add a TimedDestructionComponent to both of them, such that they destroy themselves after a given time. For the laser entity, we additionally move it in its forward direction by adding the laser's forward transform scaled by its speed and delta time. The “LaserCollisionSystem” checks the hits the collider of the laser has with other entities. If a collision is registered with a Fighter, the Fighter entity is added to the hit buffer of the laser.

# Asteroids Implementation

The asteroids (figure 4) in the game are our rigid bodies in the simulation which was one of the hard requirements of the project. In the first iteration they were simple sphere colliders with a collision system which let them explode once they collided with something beside a Fighter. An advantage to this approach when using rigid bodies, is that we don't need to check for overlaps in the spawning of the asteroids, if two asteroids are spawned in the same location, the physics engine will simply resolve the collision and they will start to drift apart. Another advantage is that because this is a space simulation, we can disable gravity and friction and apply a velocity once to the rigid body to make them float around. Therefore, no extra *AsteroidMovementSystem* is needed.

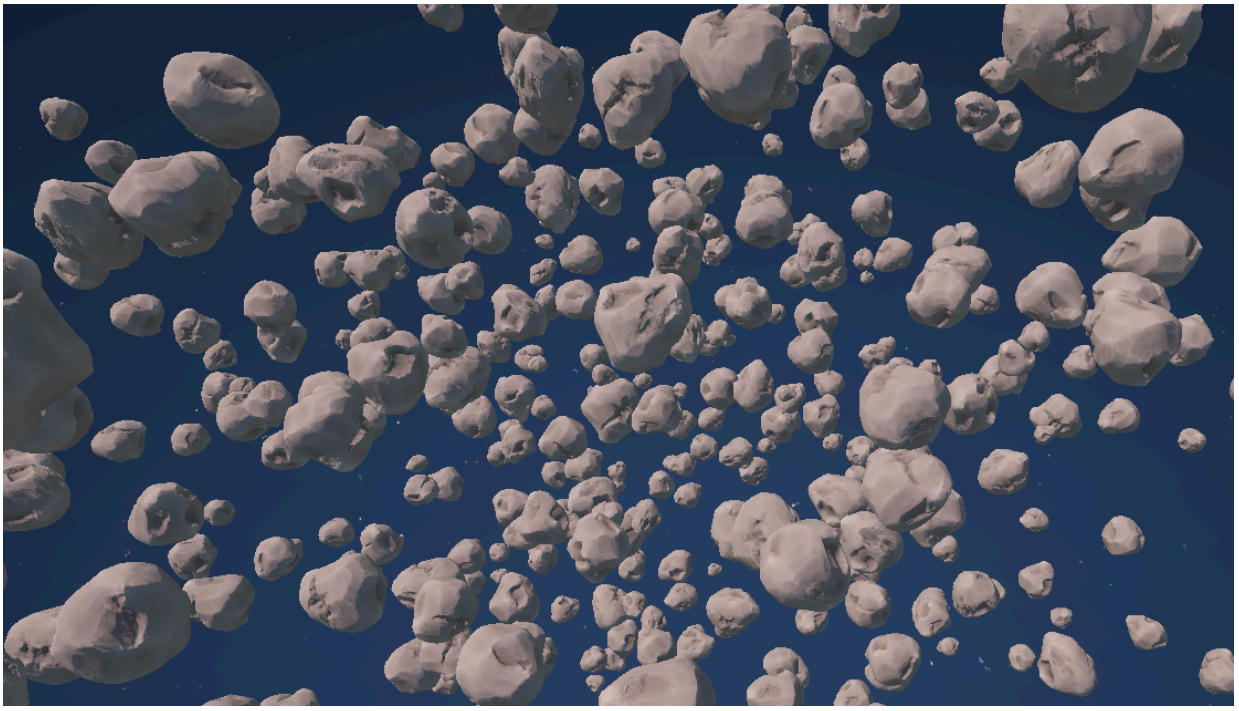


*Figure 4: Asteroids randomly arranged in the simulation*

## Asteroids Entity Structure

The Asteroid Entity is a collection of multiple Components and a DynamicBuffer. The setup we construct in the *Baker* of the *AsteroidAuthoring* is as follows:

- *AsteroidComponent*
    - Stores the sphere radius of an asteroid
- *PhysicsCustomTags*
    - Needed for physics queries in systems
- *HealthComponent*
    - Stores the amount of health left for an asteroid.
- *DynamicBuffer<HitBufferElement>*

    - Stores the hits created when an asteroid collides with a Fighter

Components added on the prefab:

- *MeshRenderer*
- *MeshFilter*
- *SphereCollider*
- *Rigidbody*

## Asteroids System

The behavior of asteroids is defined by two systems. The first one is the *AsteroidSpawnSystem* and the second one is the *AsteroidCollisionSystem*. As their names suggest, these systems handle the spawning of asteroids and the collision logic between asteroids and Fighters.

**Asteroid Spawning**

Asteroids only spawn at the beginning of the simulation and within the defined spawning bounds, which can be edited in the config. The config also defines the number of asteroids that will be spawned. Each asteroid receives a random position and rotation, as well as a random scale factor. This results in visually different asteroids within the simulation. Based on randomness, linear and angular velocities are also calculated and applied to the rigid body, causing the asteroids to drift through space. For the *AsteroidCollisionSystem*, the sphere radius of each asteroid is required. To allow easy access, the radius is stored on the *AsteroidComponent* of the entity. This avoids the need to use a rigid body lookup to retrieve the sphere radius during collision detection.

**Asteroid Collisions**

As mentioned earlier, not all entities in the simulation use rigid body components, but collisions with objects such as Fighters still need to trigger events. For this reason, the *AsteroidCollisionSystem* was implemented. The system works in a similar way to the *NearbySearchSystem*. It uses a sphere with the asteroid's collider radius to detect collisions within this area. Once a collision occurs, the system checks whether the Fighter tag is set on the detected entity. If this is the case, the entity is added to the *DynamicBuffer* of *HitBufferElement's*. This means that once a Fighter flies into an asteroid, it will be destroyed.

## Entity Destruction System

There are three systems involved in the destruction of entities. One of these systems is the *TimeDestructionSystem*, which starts a job that runs on all entities that have the *TimedDestructionComponent* and the *HealthComponent*, and it increases the stored lifetime of the component. Once this lifetime exceeds a previously set maximum lifetime, it sets the health of the *HealthComponent* to 0. Another system, which works independently of the *TimeDestructionSystem*, is the DamageSystem. It iterates over all entities that have a *HitBufferElement* and decreases the health of the target entity by the amount of damage defined by the damage-causing entity.

The final system that is part of the destruction of entities is the *DestructionSystem*. It is updated after both of the previously described systems. In this system, we iterate over all entities that have a health component. For each entity, we check if the health value is below or equal to zero. If it is, we get all linked entities of the entity with the health component by getting the LinkedEntityGroup buffer from that entity from the EntityManager. During profiling, we saw that converting this LinkedEntityGroup to a NativeArray took almost 90% of the entire update time for some frames. This was the case because we initially retrieved the linked entities for every entity, which resulted in .AsNativeArray is being called close to 4000 times per frame in extreme cases, this resulted in a duration of around 570ms for the DestructionSystem. Once we noticed the cost of this, we only retrieved the linked entities when the entity also has the StarDestroyerComponent, since they are the only entities we use that actually have linked entities.  Finally, we store the commands to destroy the entity and the linked entities, in the case of the Star Destroyer, in the Entity Command Buffer, which we playback after iterating over all entities with a health component.

# Performance Analysis

In the following, we are going to analyze the results of profiling our simulation. We are going to look at some examples that we were able to optimize by analyzing the simulation's performance in the profiler.

### CanonFireSystem

A good example of how we used the profiler to iteratively optimize our code is the CanonFireSystem. In the implementation part of the CanonFireSystem, we mentioned that we changed our initial approach of iterating over all Fighters to just a fixed number of them, to reduce the runtime of the system from O($N$) to O($1$). Profiling the simulation showed us, however, that the system was still among the slowest. One further improvement was to move the spawning of the lasers and the VFX entity to the job that also orients the canons. This further reduced the duration of the system. Nevertheless, we could still sometimes observe spikes. Looking at the hierarchy view of the CanonFireSystem showed us that the system was actually not slow but waiting on the NearbySearchSystem to finish, because it claimed to be writing to the Fighters transforms.
We were able to remove that dependency by passing the Fighter transforms as “in” and not “ref” into the NearbySearchSystem, as it actually does not write to them. With the CanonFireSystem not having to wait for another system, it is now almost unnoticeable in the profile analyzer.

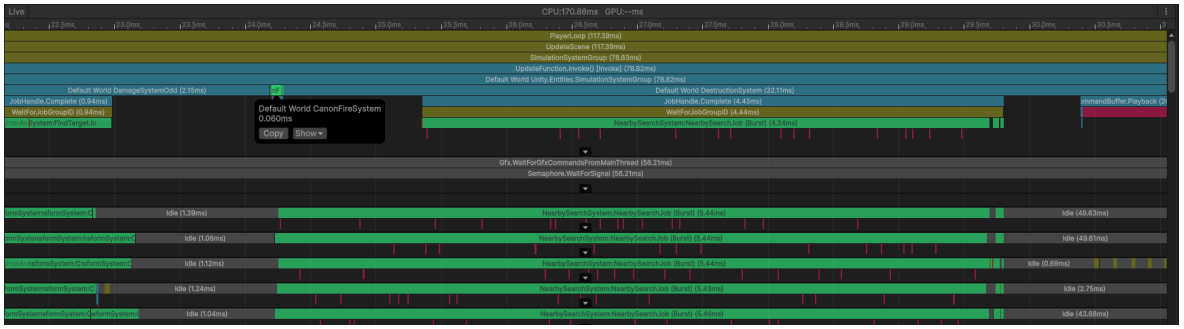



*Figure 5: The optimized CanonFireSystem in the profiler.*

## VFX Memory Leak

When we started to stress-test our simulation and let it run for an extended period of time, the simulation kept crashing due to a memory leak. We were looking into unmanaged code and manually allocated memory that we did not manage properly, but we could not find any. No matter the scale of the simulation, we would observe the untracked memory filling up over time. After carefully testing the simulation with different aspects disabled, we figured out that only entities that have a VFX graph component were causing the memory leak. This led us to finding a blog post about a bug with DX12, which causes untracked private memory to pile up and eventually crash Unity. We downgraded to DX11, and this fixed the issue.

# Benchmark

To obtain comparable benchmark results across the different scheduling schemes, namely scheduled parallel, scheduled, and main thread execution, we fixed the number of asteroids and Star Destroyers for every run and only increased the number of Fighters. This approach allowed us to isolate the performance impact caused by scaling the flocking behavior, combat logic, and nearby entity interactions. All benchmark runs were executed using a release build of the simulation to ensure realistic performance measurements.

Each test was conducted with two Star Destroyers and 500 asteroids, while the number of Fighters was increased incrementally to 500, 2.000, 5.000, and 10.000. All values were configured through the in-game UI, ensuring that the simulation settings remained consistent across all runs.

The benchmark was performed on a machine with the following specifications:

- AMD Ryzen 7 7840HS
- NVIDIA GeForce RTX 4070 Laptop GPU
- Samsung M425R2GA3PB0-CWMOD DDR5 32GB

| ***Scheduling / Number of Fighters*** | | **Average Frame Rate** | | **Average Frame Time** | |
|---|---|---|---|---|---|
| | | **min** | **max** | **min** | **max** |
| **500** | **Parallel** | 337 FPS | 354 FPS | 2.82 ms | 2.97 ms |
| | **Scheduled** | 313 FPS | 313 FPS | 3.07 ms | 3.19 ms |
| | **Main Thread** | 306 FPS | 324 FPS | 3.08 ms | 3.27 ms |
| **2000** | **Parallel** | 199 FPS | 213 FPS | 4.68 ms | 5.01 ms |
| | **Scheduled** | 121 FPS | 128 FPS | 7.80 ms | 8.23 ms |
| | **Main Thread** | 118 FPS | 132 FPS | 7.56 ms | 8.48 ms |
| **5000** | **Parallel** | 90 FPS | 93 FPS | 10.71 ms | 11.15 ms |
| | **Scheduled** | 43 FPS | 46 FPS | 21.99 ms | 23.29 ms |
| | **Main Thread** | 44 FPS | 46 FPS | 21.55 ms | 22.67 ms |
| **10000** | **Parallel** | 39 FPS | 46 FPS | 20.97 ms | 24.74 ms |
| | **Scheduled** | 14 FPS | 15 FPS | 68.30 ms | 72.11 ms |
| | **Main Thread** | 13 FPS | 14 FPS | 70.97 ms | 73.81 ms |

*Table 3: Data collected during the performance test*

Table 3 summarizes the collected performance data. For each configuration, we measured the average frame rate and average frame time over 300 frames of the simulation. In addition, the minimum and maximum values observed during this period were recorded. This measurement strategy captures both

typical performance and worst-case behavior, which is particularly relevant in a simulation where entity destruction and combat events can cause short but significant performance spikes.

As mentioned earlier, our primary target scenario was a simulation with 2.000 Fighters and 2 Star Destroyers. However, evaluating both lower and substantially higher entity counts provides a clearer understanding of how the simulation scales and where its practical limits lie. The results presented in Table 3 indicate that parallel scheduling consistently outperforms both scheduled and main thread execution, with the performance differences becoming more pronounced as the number of Fighters increases.

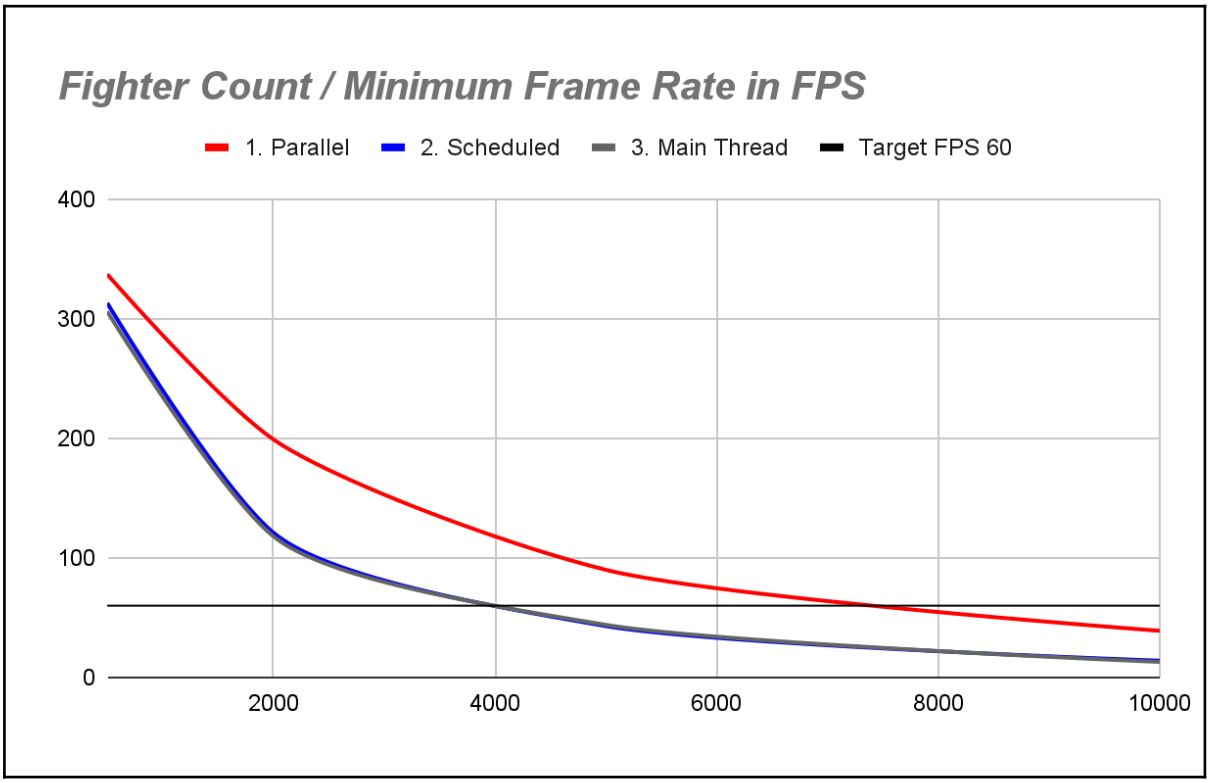



*Diagram 1: Change of the measured minimum frame rate when increasing amount of Fighters in the simulation*

To better illustrate these trends, Diagram 1 and Diagram 2 visualize the change in frame rate and frame time as the Fighter count increases. For clarity and comparability, we focus on the worst-case measurements by visualizing the minimum average frame rate and the maximum average frame time observed over the 300-frame sampling window.

Diagram 1 shows the change in the minimum measured frame rate as the number of Fighters increases. The results reveal a clear and accelerating decline in frame rate as the simulation scales. Running the simulation on the main thread and with scheduled jobs yields very similar performance across all tested configurations, suggesting that scheduling alone does not provide a significant benefit without parallel execution. In both cases, the target frame rate of 60 FPS is no longer maintained once the Fighter count exceeds approximately 4.000. In contrast, parallel scheduling allows the simulation to remain above the target frame rate up to roughly 7.000 Fighters, demonstrating the effectiveness of distributing the workload across multiple CPU cores.

Diagram 2 presents the corresponding increase in frame time as the Fighter count grows. As in the frame rate analysis, the maximum average frame times are used to reflect worst-case behavior. The results closely mirror those shown in Diagram 1. The target frame time of 16 ms is exceeded at similar Fighter counts for each scheduling method. Main thread and scheduled execution again exhibit nearly identical behavior, while parallel scheduling consistently results in lower frame times across all tested scales. This observation reinforces the conclusion that the applied optimizations for parallel execution have a meaningful impact on the overall performance and scalability of the simulation.

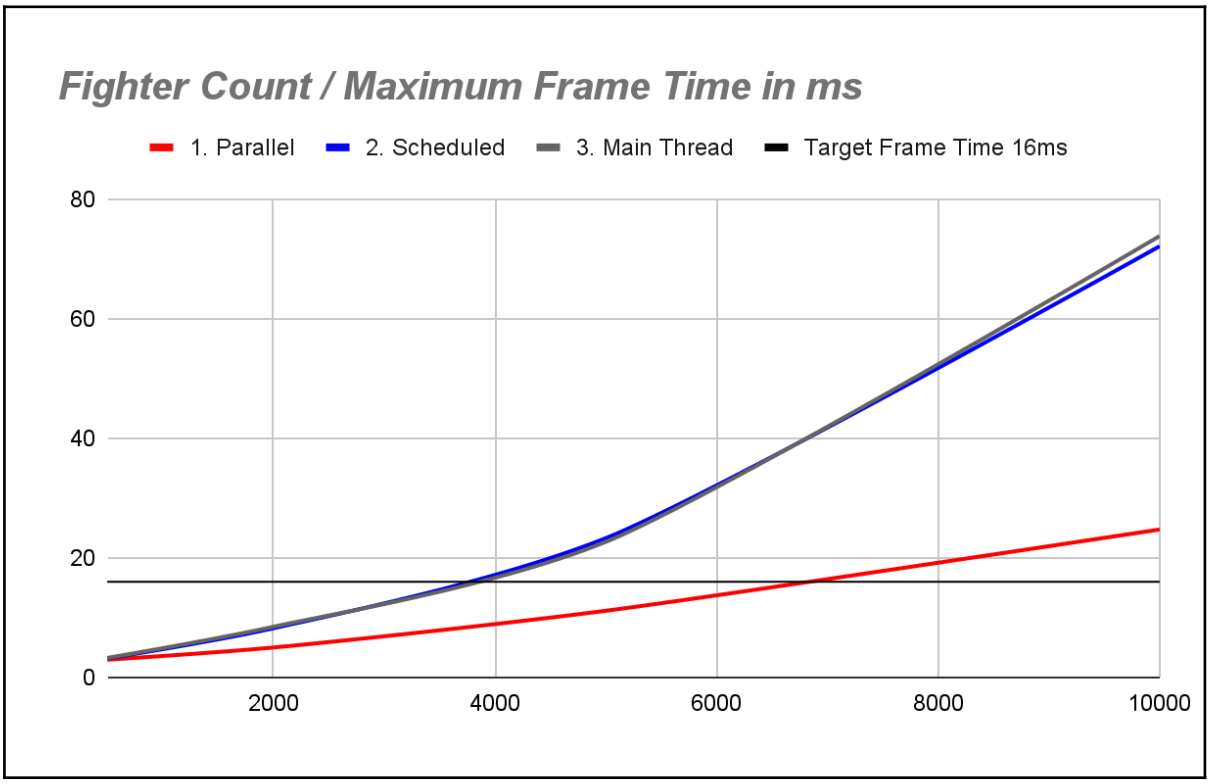
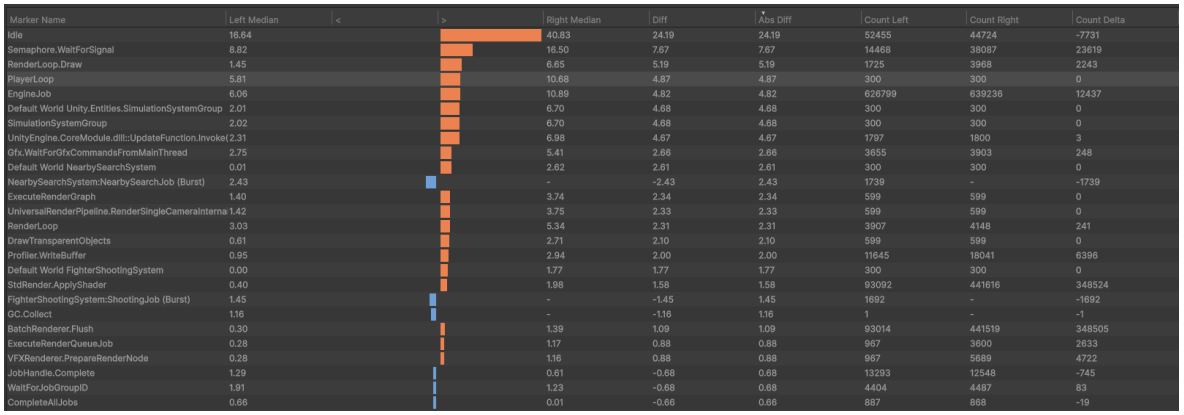


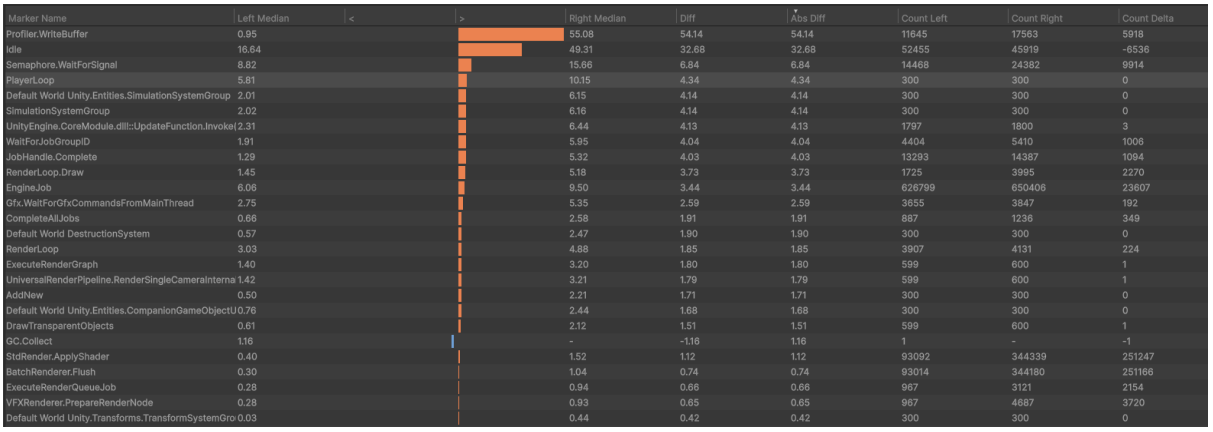

*Diagram 2: Change of the measured maximum frame time when increasing amount of Fighters in the simulation*

In Figure 6 and 7 a detailed comparison of the usage of the main thread and the worker and render thread between different runtimes can be seen. What is most evident is that the Idle marker name is more prominent in both the scheduled and also the main thread runtype compared to the running the simulation in parallel. This is the case because all worker threads are considered in the comparison, and when we run all our jobs on main or just schedule them on a single worker thread but not in parallel all the other worker threads are going to be idle. Profiler.WriteBuffer can be seen in the comparison between parallel scheduled and scheduled which should be disregarded, as it is overhead caused by the profiler. When looking at the comparison between the simulation running on the main thread vs parallel, we can see that the RenderLoop.Draw call is taking 5ms longer when the system is running on the main thread. There should be no correlation between how we run our system and the duration of the render loop, we therefore asume that the high visual variation based on what is happening in the simulation (e.g. a large number of Fighters being destroyed at once, spawning many explosions) caused this difference.



| Marker Name | Left Median | Right Median | Diff | Abs Diff | Count Left | Count Right | Count Delta |
|---|---|---|---|---|---|---|---|
| Idle | 16.64 | 40.83 | 24.19 | 24.19 | 52455 | 44724 | -7731 |
| Semaphore.WaitForSignal | 8.82 | 16.50 | 7.67 | 7.67 | 14468 | 38087 | 23619 |
| RenderLoop.Draw | 1.45 | 6.65 | 5.19 | 5.19 | 1725 | 3968 | 2243 |
| PlayerLoop | 5.81 | 10.68 | 4.87 | 4.87 | 300 | 300 | 0 |
| EngineJob | 6.06 | 10.89 | 4.82 | 4.82 | 626799 | 639236 | 12437 |
| Default World Unity.Entities.SimulationSystemGroup | 2.01 | 6.70 | 4.68 | 4.68 | 300 | 300 | 0 |
| SimulationSystemGroup | 2.02 | 6.70 | 4.68 | 4.68 | 300 | 300 | 0 |
| UnityEngine.CoreModule.dll::UpdateFunction.Invoke( | 2.31 | 6.98 | 4.67 | 4.67 | 1797 | 1800 | 3 |
| Gfx.WaitForGfxCommandsFromMainThread | 2.75 | 5.41 | 2.66 | 2.66 | 3655 | 3903 | 248 |
| Default World NearbySearchSystem | 0.01 | 2.62 | 2.61 | 2.61 | 300 | 300 | 0 |
| NearbySearchSystem:NearbySearchJob (Burst) | 2.43 | - | -2.43 | 2.43 | 1739 | - | -1739 |
| ExecuteRenderGraph | 1.40 | 3.74 | 2.34 | 2.34 | 599 | 599 | 0 |
| UniversalRenderPipeline.RenderSingleCameraInterna | 1.42 | 3.75 | 2.33 | 2.33 | 599 | 599 | 0 |
| RenderLoop | 3.03 | 5.34 | 2.31 | 2.31 | 3907 | 4148 | 241 |
| DrawTransparentObjects | 0.61 | 2.71 | 2.10 | 2.10 | 599 | 599 | 0 |
| Profiler.WriteBuffer | 0.95 | 2.94 | 2.00 | 2.00 | 11645 | 18041 | 6396 |
| Default World FighterShootingSystem | 0.00 | 1.77 | 1.77 | 1.77 | 300 | 300 | 0 |
| StdRender.ApplyShader | 0.40 | 1.98 | 1.58 | 1.58 | 93092 | 441616 | 348524 |
| FighterShootingSystem:ShootingJob (Burst) | 1.45 | - | -1.45 | 1.45 | 1692 | - | -1692 |
| GC.Collect | 1.16 | - | -1.16 | 1.16 | 1 | - | -1 |
| BatchRenderer.Flush | 0.30 | 1.39 | 1.09 | 1.09 | 93014 | 441519 | 348505 |
| ExecuteRenderQueueJob | 0.28 | 1.17 | 0.88 | 0.88 | 967 | 3600 | 2633 |
| VFXRenderer.PrepareRenderNode | 0.28 | 1.16 | 0.88 | 0.88 | 967 | 5689 | 4722 |
| JobHandle.Complete | 1.29 | 0.61 | -0.68 | 0.68 | 13293 | 12548 | -745 |
| WaitForJobGroupID | 1.91 | 1.23 | -0.68 | 0.68 | 4404 | 4487 | 83 |
| CompleteAllJobs | 0.66 | 0.01 | -0.66 | 0.66 | 887 | 868 | -19 |

*Figure 6: Scheduled parallel vs main*



| Marker Name | Left Median | Right Median | Diff | Abs Diff | Count Left | Count Right | Count Delta |
|---|---|---|---|---|---|---|---|
| Profiler.WriteBuffer | 0.95 | 55.08 | 54.14 | 54.14 | 11645 | 17563 | 5918 |
| Idle | 16.64 | 49.31 | 32.68 | 32.68 | 52455 | 45919 | -6536 |
| Semaphore.WaitForSignal | 8.82 | 15.66 | 6.84 | 6.84 | 14468 | 24382 | 9914 |
| PlayerLoop | 5.81 | 10.15 | 4.34 | 4.34 | 300 | 300 | 0 |
| Default World Unity.Entities.SimulationSystemGroup | 2.01 | 6.15 | 4.14 | 4.14 | 300 | 300 | 0 |
| SimulationSystemGroup | 2.02 | 6.16 | 4.14 | 4.14 | 300 | 300 | 0 |
| UnityEngine.CoreModule.dll::UpdateFunction.Invoke( | 2.31 | 6.44 | 4.13 | 4.13 | 1797 | 1800 | 3 |
| WaitForJobGroupID | 1.91 | 5.95 | 4.04 | 4.04 | 4404 | 5410 | 1006 |
| JobHandle.Complete | 1.29 | 5.32 | 4.03 | 4.03 | 13293 | 14387 | 1094 |
| RenderLoop.Draw | 1.45 | 5.18 | 3.73 | 3.73 | 1725 | 3995 | 2270 |
| EngineJob | 6.06 | 9.50 | 3.44 | 3.44 | 626799 | 650406 | 23607 |
| Gfx.WaitForGfxCommandsFromMainThread | 2.75 | 5.35 | 2.59 | 2.59 | 3655 | 3847 | 192 |
| CompleteAllJobs | 0.66 | 2.58 | 1.91 | 1.91 | 887 | 1236 | 349 |
| Default World DestructionSystem | 0.57 | 2.47 | 1.90 | 1.90 | 300 | 300 | 0 |
| RenderLoop | 3.03 | 4.88 | 1.85 | 1.85 | 3907 | 4131 | 224 |
| ExecuteRenderGraph | 1.40 | 3.20 | 1.80 | 1.80 | 599 | 600 | 1 |
| UniversalRenderPipeline.RenderSingleCameraInterna | 1.42 | 3.21 | 1.79 | 1.79 | 599 | 600 | 1 |
| AddNew | 0.50 | 2.21 | 1.71 | 1.71 | 300 | 300 | 0 |
| Default World Unity.Entities.CompanionGameObjectU | 0.76 | 2.44 | 1.68 | 1.68 | 300 | 300 | 0 |
| DrawTransparentObjects | 0.61 | 2.12 | 1.51 | 1.51 | 599 | 600 | 1 |
| GC.Collect | 1.16 | - | -1.16 | 1.16 | 1 | - | -1 |
| StdRender.ApplyShader | 0.40 | 1.52 | 1.12 | 1.12 | 93092 | 344339 | 251247 |
| BatchRenderer.Flush | 0.30 | 1.04 | 0.74 | 0.74 | 93014 | 344180 | 251166 |
| ExecuteRenderQueueJob | 0.28 | 0.94 | 0.66 | 0.66 | 967 | 3121 | 2154 |
| VFXRenderer.PrepareRenderNode | 0.28 | 0.93 | 0.65 | 0.65 | 967 | 4687 | 3720 |
| Default World Unity.Transforms.TransformSystemGro | 0.03 | 0.44 | 0.42 | 0.42 | 300 | 300 | 0 |

*Figure 7: Scheduled parallel vs scheduled*

# Discussion

By looking at the simulation in the profiler, we found several performance problems and improve them step by step. We, for example, optimized systems that were slower than expected, like the CanonFireSystem. We managed to find and fix a tricky memory issue related to visual effects that caused crashes over time. Our benchmarks also showed clear differences between the tested scheduling methods, with parallel execution performing best as the number of Fighters increased.

Overall, this confirmed that using multiple CPU cores is important for keeping the simulation running smoothly as it scales.

A last shortcoming of our simulation is how we spawn space ships. When a large amount has to be spawned at once, it cloggs up the main thread because we did not manage to efficiently parallelize it without causing other problems. We prototypes recycling the Fighters by just resetting their position, but it did not yield the expected performance improvement, but instead caused different issues. Nevertheless, we believe that object pooling is something that should be further investigated for our project as it promises a final improvement, avoiding structural changes that have to run on the main thread.

## Further Improvements

### Star Destroyers

The current implementation of the Star Destroyers has a lot of room for improvement, both in performance and game design. For once, currently, the Star Destroyers do not avoid each other and just pass through each other. Similarly, the lasers of the Star Destroyers do not collide with Star Destroyers and deal damage to them, but just pass through them. Additionally, the movement of the Star Destroyers depends on random positions. It can happen that the new target point of a Star Destroyer lies right behind it, which causes it to rotate at a very high speed. These are improvement points that we would have liked to implement if we had more time.
In terms of performance, the model of the Star Destroyer could be optimized. Currently, it is a model consisting of more than 1000 parts. These parts should be combined in one model, and the complexity of the mesh should be decreased.

### VFX

We observed that the rendering takes significantly longer with an increasing number of Fighters. This is due to the fact that more entities that have a VFX graph are spawned. Upon closer investigation, we could see that the DrawTransparentObjects call can take as much time as 30% of the total frame time, with most of the transparent objects being the explosion particles and the trails of the Fighters. In a future improvement, the usage of VFX graphs, especially the ones on the main entity (Fighters), should be heavily reduced.

# Conclusion

Our project simulates a large-scale space battle between thousands of Fighters fighting against larger Star Destroyers, avoiding physically simulated Asteroids. We implemented the projects using Unity DOTS and focused on high-performance despite heavy VFX and large numbers of entities. A main challenge and goal was to implement swarm behaviour among the thousands of Fighters, which we believe we achieved while maintaining performance. We also exposed parameters which affect the swarm behaviour and number of entities.

Finally, we evaluated how the simulation performs across different scheduling strategies. Our findings are that scheduling jobs in parallel pays off and drastically improves performance, which became especially evident as we scaled up the number of entities. It was an insightful learning experience to use the profiler to see performance issues and see how they diminish as we iteratively identified issues and fixed them.

The project provided valuable insight into designing and optimizing data-oriented simulations in Unity. While there is still room for improvement in both performance and system design, the final result meets the project goals and serves as a solid foundation for further exploration of large-scale, high-performance game simulations.