SAVONIA
University of Applied Sciences

THESIS – BACHELOR'S DEGREE PROGRAMME
TECHNOLOGY, COMMUNICATION AND TRANSPORT

OPTIMIZATION FOR BULLET HELL GAMES

AUTHOR Mikko Saari

SAVONIA UNIVERSITY OF APPLIED SCIENCES

THESIS
Abstract

Field of Study Technology, Communication and Transport	
Degree Programme Degree Programme in Information Technology	
Author(s) Mikko Saari	
Title of Thesis OPTIMIZATION FOR BULLET HELL GAMES	
Date 09.06.2025	Pages/Number of appendices 21 pages
Client Organisation /Partners Savonia University of Applied Sciences	

Abstract

The purpose of this thesis was to investigate and test different solutions for improving the performance of games involving the creation and management of large number of game objects. While the thesis focuses on the genre of bullet hell games and the use of Unity game engine as its frame of reference, some of its contents can be applied into other game genres and game engines as well.

The topic was approached through study of learning materials and documentation of three different optimization design patterns: object pooling, multithreading with Unity's Job system and data-oriented Entity Component System. The findings from the theoretical part were then used for the development of test scene using the design patterns to further deepen the understanding. The limitations and challenges as well as their solutions uncovered while developing the test scenario were documented. The effectiveness of the design patterns in comparison to unoptimized version was also estimated by gathering performance data of using test runs that were comparable with each other.

Based on the results of the practical testing it was decided that combination of Entity Component System with unoptimized game objects would both solve the performance issues while being the easiest to develop. Object pooling did not show any gains in performance during testing and added complexity to objects lifecycle made debugging problems challenging. Job system was skipped over during the development of the test scene due to findings done during theoretical research.

Use of Job system in combination with Entity Component System stays as an open possibility in future if further optimization is necessary. Furthermore, as Unity is continuing the development of Entity Component System, its limitations and implementation are open for major changes within future versions of Unity game engine.

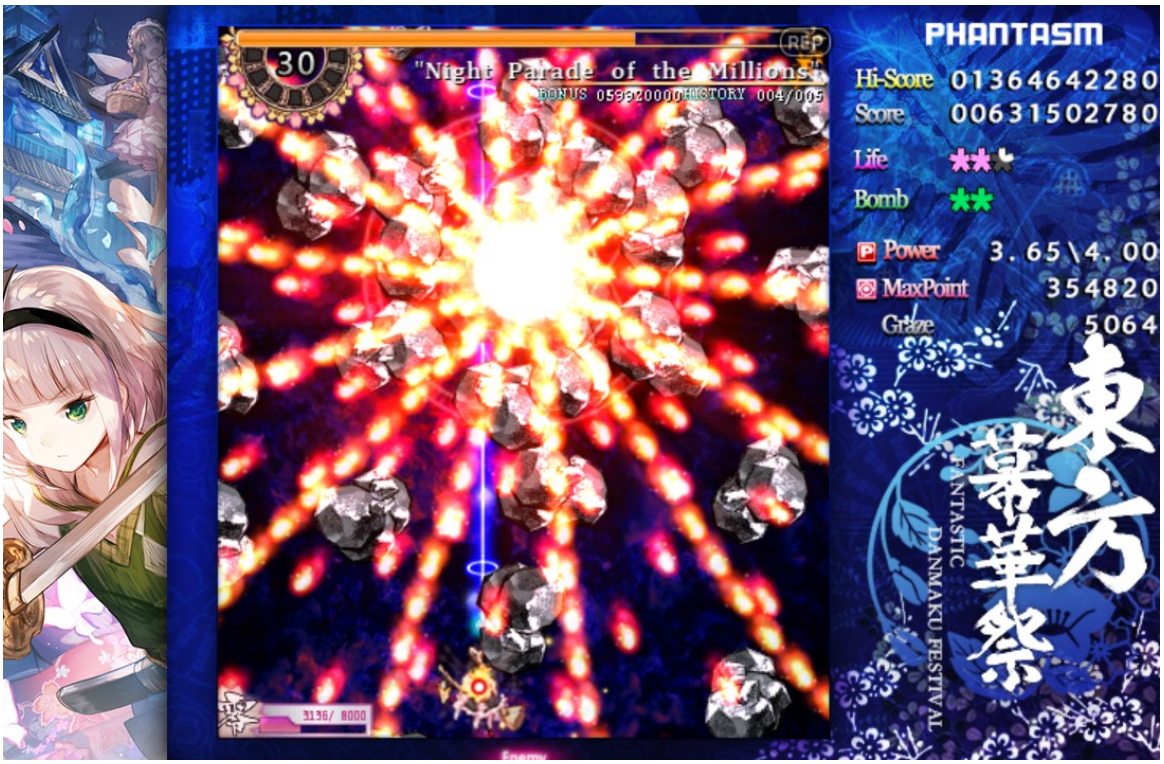
Keywords
Game Development, Unity, Architecture, Optimization, Desing Pattern, Bullet Hell, DOTS, ECS, Job System, Multithreading, Object Pooling

CONTENTS

1 INTRODUCTION

Bullet hell is a subgenre of shoot 'em up games that is defined by the large amounts of enemy bullets for the player to dodge, as seen for example in picture 1. The genre originates from arcades and this history can still be seen in many aspects of the design of the games as well as the terminology related to the genre. Some notable games of the genre are Ikaruga, DoDonPachi and most of the Touhou series (GiantBomb, n.d.).

Some bullet hell influences can also be seen outside of arcade style shoot'em ups, for example "Enter the Gungeon" is tagged as bullet hell roguelike on its steam page (2016) and it differs greatly from the typical arcade style bullet hells.



PICTURE 1. Touhou Fantastic Danmaku Festifal 2, Touhou inspired fangame that acts as a good example of modern arcade style bullet hell game.

This thesis aims to find out what design patterns and architecture decisions can be used for the genre using performance optimization as a point of view. The method used was to investigate different solutions through available documentation, learning materials and other online sources. A prototype was also developed using different approaches to test and compare them in practice and to get a practical viewpoint into possible challenges related to the implementation of the different solutions.

2 ARCHITECTURE MODELS FOR OPTIMIZATION

2.1 Object pools

Object pooling is a design pattern in which used gameobjects are pre-instantiated and recycled to eliminate the performance heavy instantiate and destroy calls while the game is running. The pattern is used mainly for performance and is well suited for situations where there is need to rapidly create and destroy instances of the same prefab objects. Bullets on the top-down shooter game are clear example of this kind of need, making object pooling a good candidate when looking for optimization. (Unity Technologies, n.d.b).

While object pooling does not eliminate the need to create the initial objects into the pool it can be done when it does not have meaningful impact on gameplay, for example start of the level or transition between level and boss fight when there are no enemies or bullets on the screen. (Wilmer 2022, pages 46-52.)

One consideration when using object pooling is that it does keep the deactivated objects in memory. While this can help alleviate stutters with garbage collection it is also good to remember that storing large amounts of objects in pool does reserve resources, even when the objects are not in their active state. Recycling also does complicate the lifecycle of the pooled object, increasing the risks of bugs. Depending on the implementation object pooling might also not be thread-safe, meaning it can introduce race conditions in multithreaded context. (Unity Technologies, n.d.e).

Object pooling is a well-established pattern and while it is possible to create custom solution based on it, Unity also offers build-in API for it starting from versions 2021 (Unity Technologies, n.d.e). The pattern is utilized with, for example, Unity's particle systems. All in all, the pattern is used often and considered among best practices for its target use cases. (Wilmer 2022, pages 46-52.).

2.2 Unity Job system

Under normal circumstances Unity runs the developed application on a single main thread. This means that normally only a single CPU core is utilized. The job system is Unity's solution for writing multithreaded code with automatic management of the worker threads and task allocation. Performance gains are achieved by allocating tasks declared as jobs to be run on the worker threads parallel to each other and the main thread, better utilizing CPUs with multiple cores. (Unity Technologies, n.d.d).

Job system can be used both with and without utilization of Entity Component System. (Unity Technologies, n.d.d). However there are performance overhead costs tied to management of the jobs. The exact impact of the overhead costs depends on multiple factors like size and the number of jobs, number of CPU cores and Unity version that is being used. The impact of the overhead was more meaningful with smaller the job combined with high thread counts and while the newer versions are still affected to some extent, the problem was amplified with the older versions of Unity. Either way, the advantages of the job system are diminished when the individual tasks are small, for example updating the location of single gameobject. (Vacheresse, 2023). For this reason, in the context of use in the bullet hell game it might be necessary to use manager script that updates the

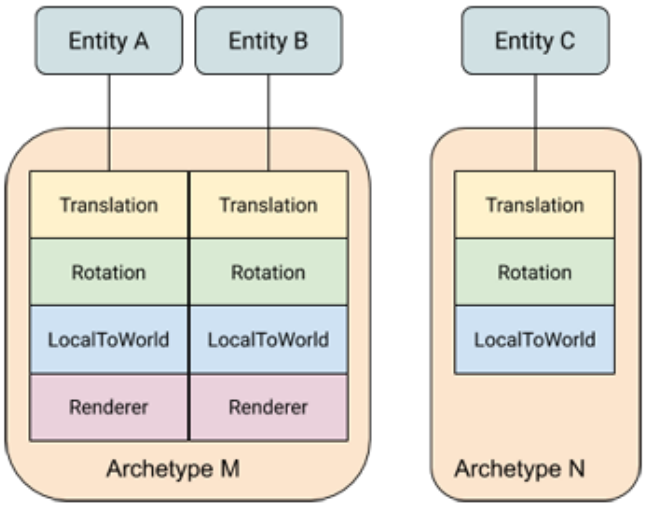
location large batches of the bullets in single class instead of each bullet having script instance and job for updating only their own location. This change in structure would be like that of the role of Systems in ECS (Entity Component System) architecture, even if working with gameobjects instead of entities.

2.3 Entity Component System

Entity Component System (ECS) is a data-oriented architecture that separates data from behavior logic and it is integral part of Unity's Data-Oriented Technology Stack (DOTS.) As the name suggests, ECS consists of three main parts: Entities, Components and Systems.

Entities are representations of things in the game, and they only handle the identity of the object without holding any data or logic themselves. Instead, all the data is stored in components which belong to the entities. The behavior logic, on the other hand, is handled by the systems. Unlike object-oriented architecture used with gameobjects, systems are not tied to the entities they affect. Instead, single instance of each system queries the affected entities by their components and update all affected entities in single instance of system utilizing for example foreach loop. (Unity Technologies, n.d.g).

While ECS is made to work well with the job system and the two are often paired together ECS also provides performance gains independent of the job system. Memory management is one factor that boosts performance. In short, the entities are grouped together into archetypes, based on the components the entity holds. In other words, each archetype consists of all the entities with unique combinations of components as demonstrated by picture 2.



PICTURE 2. Example of entity archetypes. Entities A and B have the same components as each other so they share entity archetype. (Unity Technologies n.d.c).

With ECS all the entities that share entity archetype are saved together in memory chunks. This allows more efficient memory management as all the entities and by extension components are stored together. Similarly, when querying for entities affected by the system instead of needing to go

through all the entities in the game, it is enough to go over all the entity archetypes, which are much smaller in number. (Unity Technologies, n.d.c).

The drawbacks of using ECS are that ECS currently lacks some of the components available when working with normal gameobjects. One major example of this is animations as animator components used with gameobjects cannot be used when working with entities and there is no official replacement for ECS workflows released yet. (Unity Forum, n.d.).

Other limitations of ECS included that since entities required the use of subscenes they would not mix easily with normal gameobjects. (Unity Technologies, n.d.g). On the other hand, while using gameobject representation in addition to entity could solve some of the ECS limitations, it can also lead to sacrificing some of the performance gained using ECS as well.

ECS is still relatively new and under active development. While this means that new features are likely to be released in the future and might fix some of the limitations found at the time of writing this thesis, ECS being under active development also means that old learning materials on the topic were and will be often outdated. (Unity, 2024).

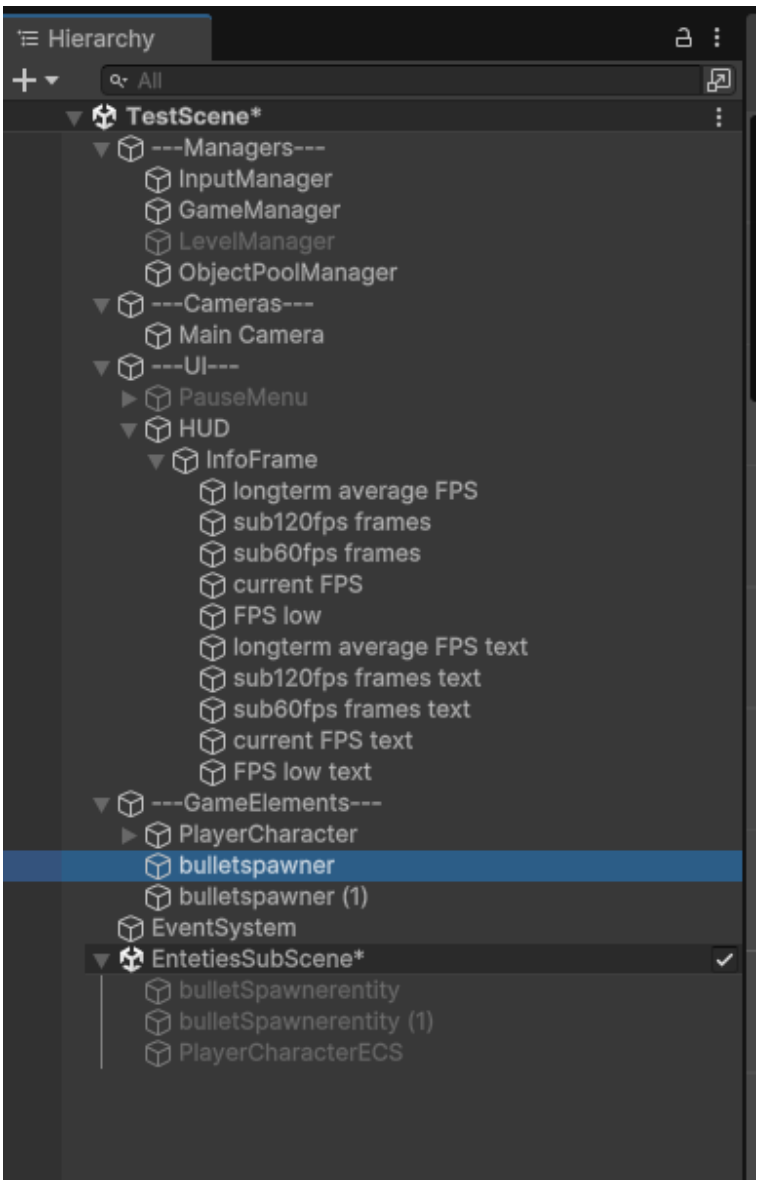
3 BUILDING THE PROTOTYPE

The prototype scene was developed to investigate and test different design patterns and other architectural decisions before starting the development of the content for the actual game project that acts as the context for this thesis. Unity was chosen as the game engine because it is well suited to the development of 2D games and has more extensive learning materials available compared to Godot. Earlier personal experience with the engine as well as Unity's established position in the game development scene also played a big role in decision-making.

The test scene was kept simple, and many of the elements that would be present in the complete game were not utilized, even if some of them were present in the scene. Also, some elements visible in Picture 2 were specific to later versions of the test scene and were not present in the naïve approach. These elements include ObjectPoolManager object used for the object pooling solution and the entities subscene with its contents that were used for the ECS solution.

The main elements of the test scene were the two bullet spawners used for creating the test pattern, bullet prefab that were initialized by the bulletspawners and the player character that was able to collide with the bullets spawned. Out of these bullet prefabs were not included in the hierarchy view outside of runtime as they were initialized from the prefab template while the game was running.

Other gameobjects shown in picture 3 that were utilized include InputManager for reading player inputs, the camera and different text fields showing performance metrics were used to estimate the effectiveness of the optimization solutions. These metrics included an average framerate over the test duration, the count of frames that had the framerate under 120 and 60 fps, the current framerate and finally, the minimum framerate that was reset on 5 second intervals, that was mainly used for detecting when the stutters happened so they could be analyzed using profiler tools. Empty gameobjects were also used for organization purposes in a way that one would use folders as other gameobjects were parented under them as child objects, allowing clear grouping and minimizing content in the hierarchy view.



PICTURE 3. Unitys hierarchy window displays all content of the test scene while the game is not running. The picture is taken after the addition of ECS solution with the ECS elements disabled. Elements not necessary for the test scene are also disabled in the screen capture. The bullet pattern used for testing the performance is generated by the two bulletspawner gameobjects under GameElements section. Script for calculating and updating fps data is a component of InfoFrame panel gameobject that also acts as a parent for the text fields showing said information.

Due to the test scene lacking features such as character animations, moving backgrounds and particle effects it is important to remember that the performance of the actual game will be worse than the performance of the test scene. The machine used for development and testing was also more powerful than target system requirements for the game project. On the other hand, there is also difference in performance between running the game in editor and the complete build of the game, with the build being more performant when compared to running on editor. Because of this the results of the tests might not reflect the complete game and can only be used to compare different solutions with each other and the results that can be considered acceptable in the test scene might not be good enough in the final product. It is important to consider that there would be some room

for the impact of adding elements of the game that were not present in the test scene when evaluating the results.

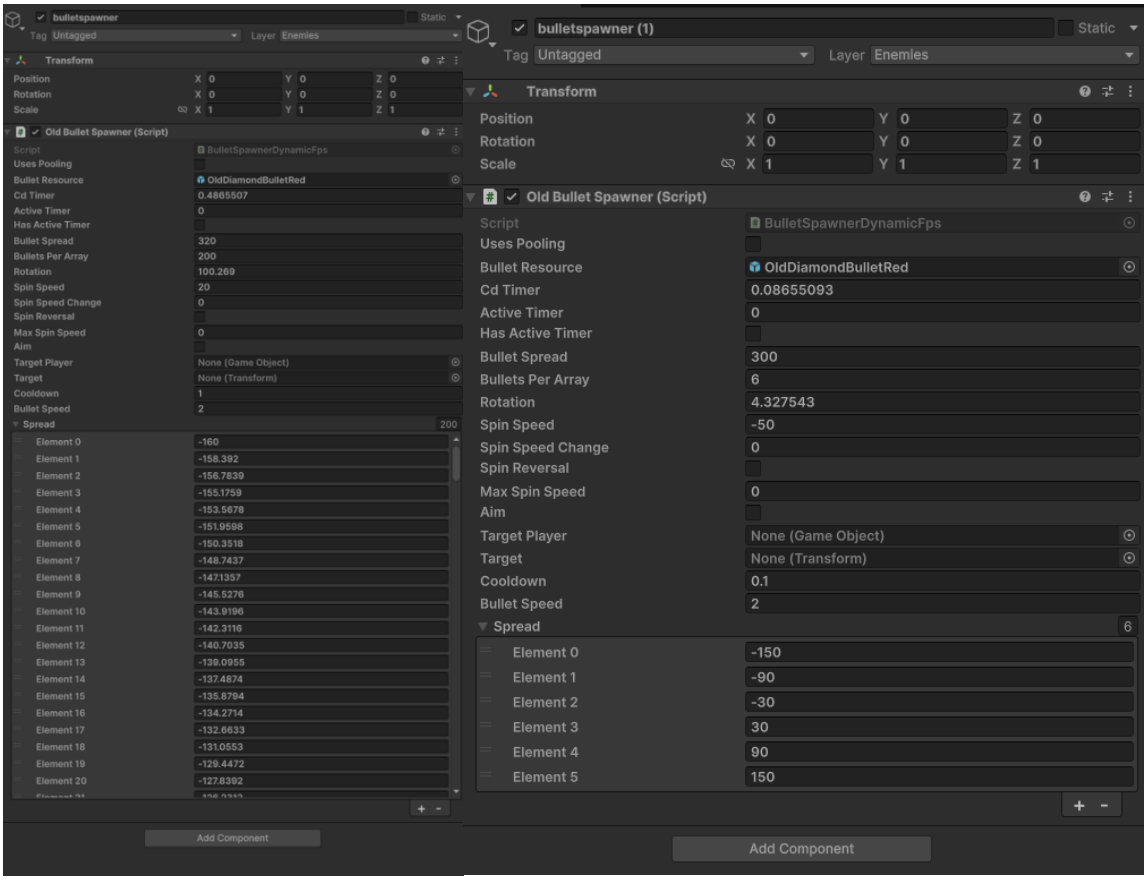
3.1 Designing the test pattern

The bullet pattern used for testing was built using two empty gameobjects with BulletSpawner monobehaviour script in the middle of the game area for spawning the bullets in plausible pattern, suited to be included into for example boss battle. The bulletspawner script exposed multiple fields to the editor to control the created bullet pattern (Picture 4.) The same pattern was used with the same bullet amounts for all the different approaches to be able to compare the results between the different design approaches.

Bullet Resource field was used to save reference to bullet prefab of which the script instantiates copies of. Cooldown timer (Cd Timer field in the picture 4) holds the time until the next instantiation of the bullets. The value was decreased every frame by the time between the frames and when the value drops below 0 it got set to equal the cooldown value, controlling the time between the bullets. Setting up the initial value for cooldown timer field in turn would set delay before first bullets get created. On the other hand, Active Timer and its corresponding Boolean field are used to determine when the script would get disabled. For the test pattern as we wanted the tests to run long times to even out the measurements the Boolean was set to false to not disable the test pattern.

Bullets per array and bullet spread values were used to set both the size of the arch for bullets and how many bullets the arch consisted of. Based on these two variables, angles for individual bullets were calculated in the spread list.

Last of the values used for the test pattern were the current rotation which got updated each frame based on the rotation speed declared in spin speed field. For this pattern statistic rotation speed was used but it would also be possible to set acceleration for it as well as the max speed after which the acceleration is reversed making the rotation speed slow down and reverse until hitting the max speed again.



PICTURE 4. The two bulletspawner gameobjects during runtime showing the values used for creating the test pattern. The one on the left instantiated 200 bullets on 320-degree arc with 1 second interval while rotating counterclockwise at the speed of 20. The one on the right side instantiated 6 bullets at once with 0.1 second intervals between them while rotating clockwise. Using the 300-degree arc with 6 bullets gives even spacing for a whole circle, if 360 degrees were used two bullets at the end of the arc would overlap each other. The outcome of these variables can be seen in picture 6.

The rest of the values for bullet spawner script would only be used if the pattern was directed based on the player's location. This behavior was not used this time as declared by aim Boolean being false.

All in all, the values provided by the script allowed for a decent amount of flexibility in creating different patterns and clearly exceeds the needs of this test scene. On the other hand, it does not cover every scenario or need of the final game. As such new versions to be used in addition to the current bullet spawner script must be created during the development of the game with specific refinements based on the patterns that are hard to create utilizing the current script.

As this version of the bulletspawner script was able to only spawn a consistent number of bullets with consistent interwall between them, more complex patterns, including the one used for the test scene, needed to be constructed by using multiple objects with bulletspawner script together. This way, even thou this version bulletspawner script had its limitations it could be used for building multiple different complex patterns. This flexibility is further increased due to the possibility to tie the

script into any gameobject instead of it having to be directly on the enemy character as well as possibility of spawning any prefab, not only simple enemy bullets. For example, the prefab instantiated with bulletsplawner script does not need to be the actual enemy projectile itself, instead it could include more complex structure utilizing child gameobjects where the child objects would be the actual projectiles. An example of pattern where this approach should be used can be seen in Picture 5 where an empty parent gameobject in the middle would be initialized by the bulletspawner and would then be used to control the child objects, the actual bullets in ring formation around the parent, in unison.

PICTURE 5. Bullet pattern where the blue rings could be replicated by spawning empty game object in the middle and having the actual projectiles as the child objects on the edge of the ring. This allows easy rotation and movement of the rings using the parent object.

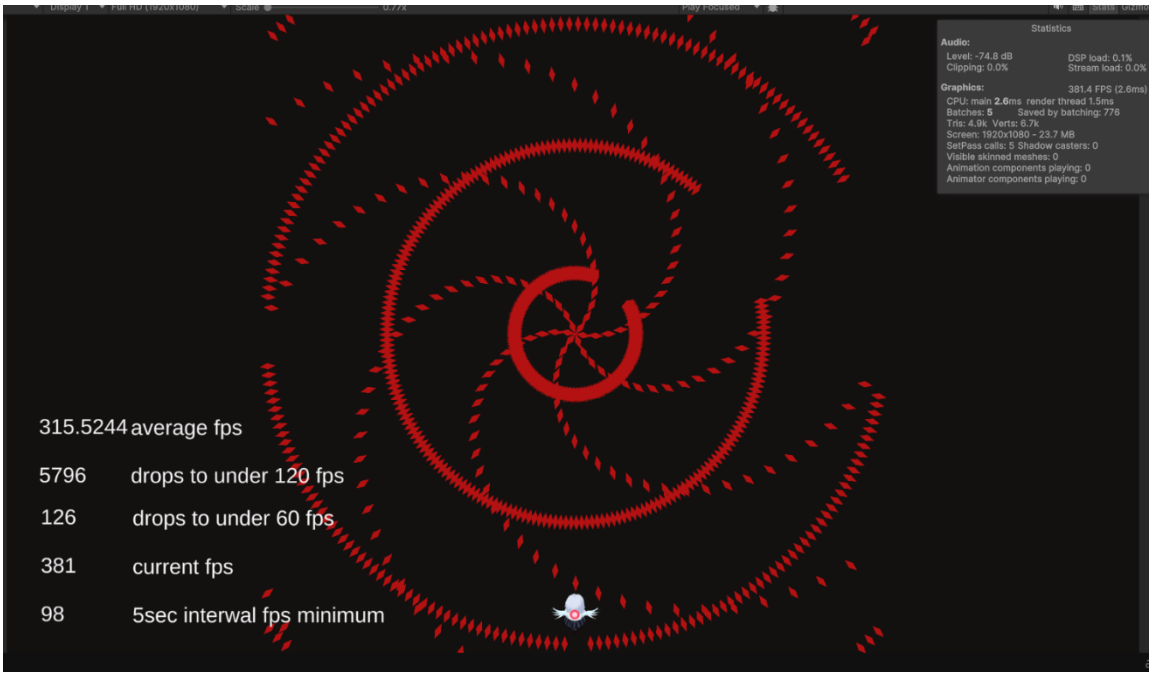
3.2 Naïve approach

The first version of the test scene was built without the use of any of the optimization architectures to test if there was need to optimize the performance and to get the baseline for estimating the performance of the prototype scene and the effectiveness of different optimization architectures. The only optimization considerations made were to separate the colliders into different layers so that the bullets would not detect collisions with each other and to move the bullets using rigidbody component instead of transform to avoid recreating the colliders for bullets each frame. The bullets were also deleted from the scene after they went outside the game area so that moving them would no longer take unnecessary resources.

The number of bullets was determined based on both what would be reasonable for a test scene and what the scene would still be able to handle. The results showed that while the current fps was still acceptable, there were periodic drops below the targets that would have a negative impact on the experience with the game. There was also noticeable stuttering each time a new ring of bullets,

consisting of 200 gameobjects, was created. This was also noticeable in the performance profiler that showed big spikes in even intervals, corresponding to the spawning of the rings.

Based on the profiler Physics2D.FindNewContacts calls were a big cause of the performance problems. The cause behind the issue was bullets checking contacts with each other, even if the bullet objects collider was set to capture contacts only with object from the player layer. However, to avoid using resources spent calculating the unnecessary collisions the change had to be made into the layer collision matrix under project settings as well. Once implemented, the performance improved greatly. That said, even after the improvement there were some drops under 60 fps caused by garbage collector as well as more common drops under 120fps meaning it was still worthwhile to investigate if these drops could be mitigated by taking advantage of different optimization architectures. The results of the test run along with the pattern used to test the performance can be seen in the picture 6.



PICTURE 6. Test scene after the collision matrix was set to only consider collisions with player character. Results for the naïve approach test run can be seen on the bottom left. The setup for the pattern can be seen in Picture 4.

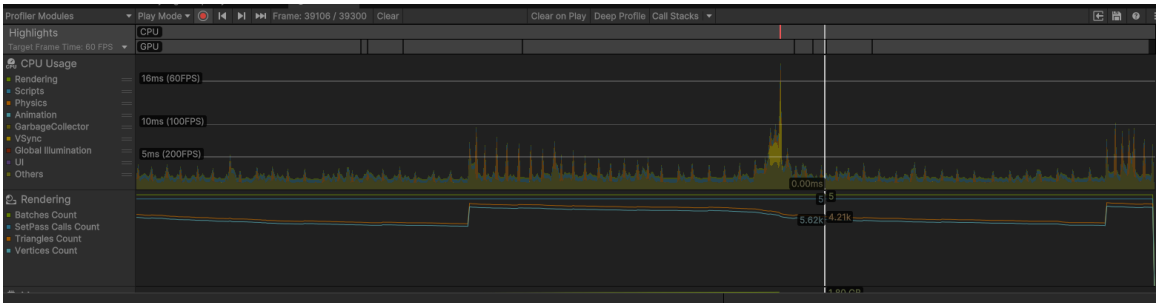
To test the relationship between the creation of new bullets and the total amount of active bullets in the scene, the game area limits were removed which increased the time each bullet was active before being removed and thus increased the total amount of bullets in the scene without changing the rate of bullet spawning. This caused the current fps to drop linearly in correspondence to the total amount of active bullets. There was also a smaller impact on drops under target framerates.

While in the test scene dynamic frame rate was used in the final game it will be switched to use locked framerate, usually 60 fps, with possibility to add multiples of base framerate as an option to settings. Because of these factors, finding ways to optimize the drops under the target framerate was considered more important than the average fps readings and so the total amount of active objects, while playing the role in performance, can be considered to play lesser role in comparison to the strain of instantiating large batches of objects at once. Thus, later will be chosen as the main

area to focus on optimization efforts, at least at the beginning, using drops under 120 fps and drops under 60 fps as primary metrics. To be able to compare the count of frames under target framerate, it was decided that a fixed benchmark time of 10 minutes would be used.

3.3 Implementing object pooling

When investigating the naïve approach, it can be seen from the profiler that the framerate is mostly stable and in reasonable readings with periodic spikes as seen in picture 7 that cause the fps to drop below 60. These drops appear to be caused by Unity's garbage collector that examines all objects in the managed memory to remove unreferenced objects and free up memory. (https://docs.unity3d.com/Manual/performance-garbage-collector.html)



PICTURE 7. Profiler during framerate drop caused by garbage collection. Garbage collection is highlighted in the picture.

To address this implementation of object pooling for bullets was tested as it would recycle bullet objects instead of instantiating new ones and destroying the bullet objects after they have been used, adding things for garbage collection to clean.

3.3.1 Technical object pool solution

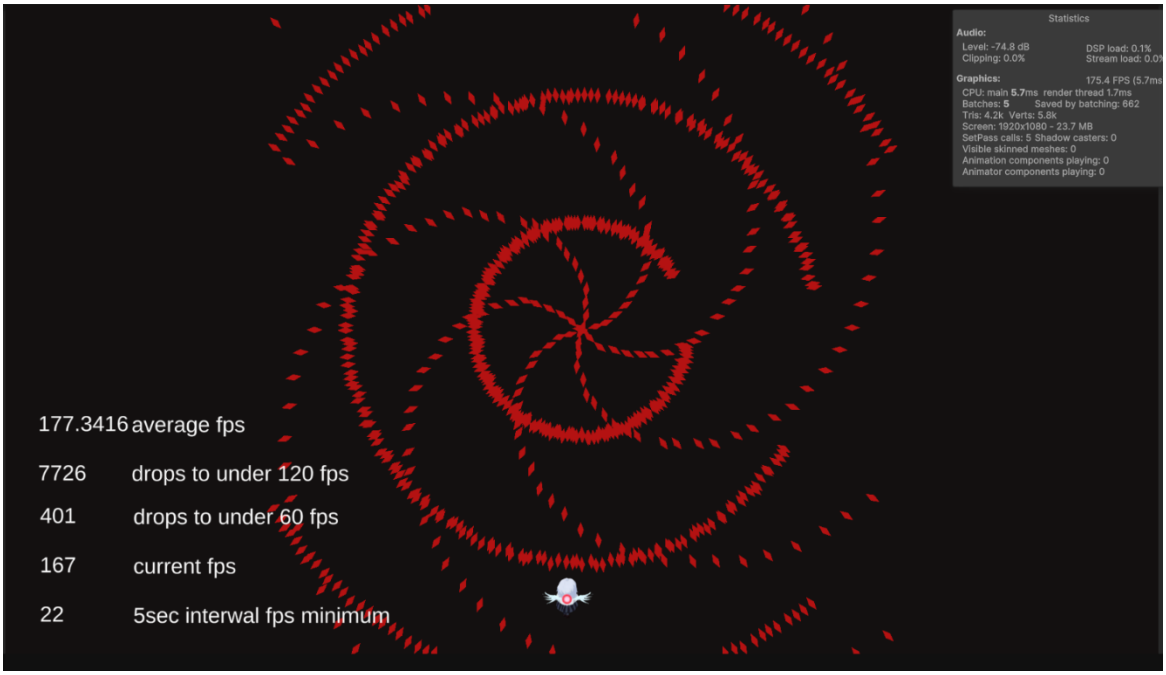
While Unity does have premade libraries for object pooling, custom solution was developed both for learning purposes and to customize the solution to the needs of the game project. The solution consisted of new object pool manager and poolable object scripts as well as minor alterations to existing bullet and bullet spawner scripts.

Object pool manager script was made responsible for managing the existing pools with functions for pulling objects from the pool as well as pushing them into the pool after the object was disabled. Since every bullet prefab would require its own pool, it was decided that the pools would be saved in dictionary. This allowed all object pools to be managed by a single instance of object pool manager which enabled the use of singleton pattern for easier referencing. The individual pools could be prefilled with bullet assets when initialized, however as trying to pull object from an empty pool would cause problems the pull function would also initialize new bullets if the pool was empty.

Alterations to the previously existing bullet and bullet spawned scripts were also necessary. These changes consisted of replacing the destruction of bullet game object with disabling it instead as well as calling pull function of the object pool manager instead of instantiating new object in the bullet spawner script. The Boolean flag was created to choose whether to utilize object pooling so that the previous, naive approach could be preserved in functional, mostly unchanged form.

3.3.2 Object pooling results and encountered problems

Multiple problems were encountered while developing object pooling, most of which remain without a good explanation or solution. It is, however, reasonable to assume that the problems are caused by either the more complex lifecycle of the objects and/or race conditions. For example, if bullets use rigidbody component, they will not reset the position when being pulled from pool and the recycling would break. Current assumption for the reason is that the bullet detects the edge of screen causing the bullet to be disabled again and returned to pool before the position has been reset. For some reason this bug doesn't happen if the bullet does not have rigidbody component.



PICTURE 8. Test run results for object pool implementation. Bug on the object pooling has made the pattern nondeterministic which can be seen on the bullet circles.

However, even in the best-case scenario object pool still caused unwanted effects on the shapes and behavior of the bullet patterns, seen in picture 8 in comparison to the desired pattern of picture 6. Not all the problems could be explained with previous uses of the object either as prefilling the pool behaved differently when compared to initializing new objects from the pull function. In fact, the pattern created corresponded with the naïve approach only in the case of new bullet being created due to the pool being empty.

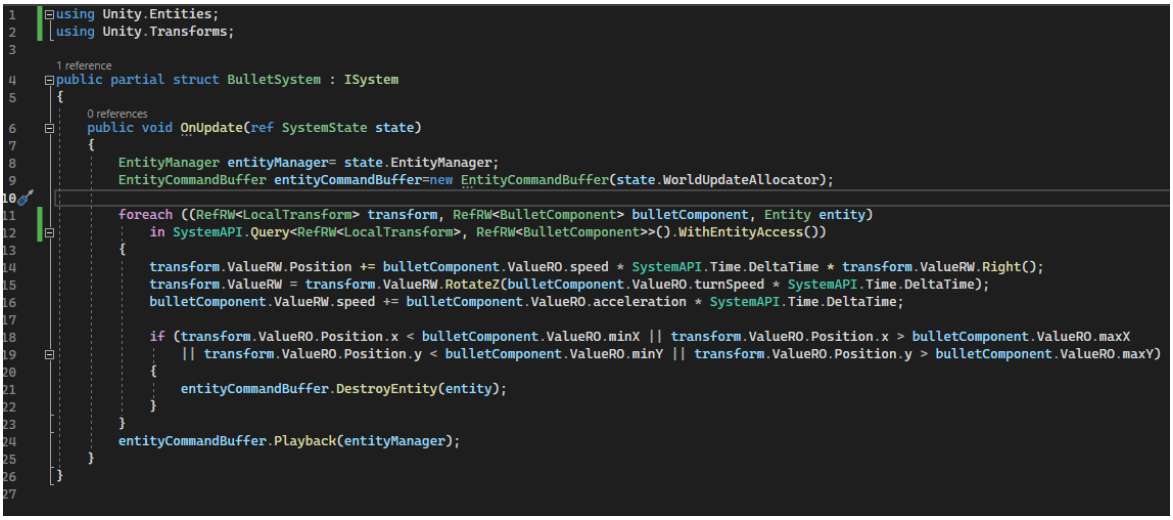
No improvement in performance could be seen and the hit caused by garbage collection appeared unchanged. In fact, test results indicate worse performance when object pool is in use. Part of the difference can be explained by editor requiring more time, around 3.2ms per frame when using object pooling compared to 1.8ms per frame of the naïve approach. While this should not affect the performance of complete build it is not large enough to explain the difference on average fps readings. Performance on complete build was not tested.

It is likely that the problems encountered were caused by the problems in the custom implementation that was developed as the results differed from expected. This means it is likely possible to fix some if not all of the bugs encountered, as well as investigate what is causing garbage collection by

which some of the expected gains might be achieved. However, due to the difficulty of debugging object pools due to more complex object lifecycles, it was decided that the best course of action was to investigate other options available before returning to object pool pattern, if needed.

3.4 Implementing ECS

In ECS architecture the data that was tied to each object and the logic for changing said data were separated into component and system scripts. Entities were introduced as a replacement for gameobjects usually used within Unity and acted as container for different components tied to the object. Contrary to monobehaviour scripts used with gameobjects, systems holding the logic affecting entities were not directly tied to any entity or object themselves. Instead, each frame they queried the entities they affected based on the components of said entity and updated all returned entities utilizing foreach loop. An example of this can be seen in Picture 9 which depicts the system used for updating the position of all the entities with BulletComponent component.



```
using Unity.Entities;
using Unity.Transforms;

1 reference
public partial struct BulletSystem : ISystem
{
    0 references
    public void OnUpdate(ref SystemState state)
    {
        EntityManager entityManager= state.EntityManager;
        EntityCommandBuffer entityCommandBuffer=new EntityCommandBuffer(state.WorldUpdateAllocator);

        foreach ((RefRW<LocalTransform> transform, RefRW<BulletComponent> bulletComponent, Entity entity)
            in SystemAPI.Query<RefRW<LocalTransform>, RefRW<BulletComponent>>().WithEntityAccess())
        {
            transform.ValueRW.Position += bulletComponent.ValueRO.speed * SystemAPI.Time.DeltaTime * transform.ValueRW.Right();
            transform.ValueRW = transform.ValueRW.RotateZ(bulletComponent.ValueRO.turnSpeed * SystemAPI.Time.DeltaTime);
            bulletComponent.ValueRW.speed += bulletComponent.ValueRO.acceleration * SystemAPI.Time.DeltaTime;

            if (transform.ValueRO.Position.x < bulletComponent.ValueRO.minX || transform.ValueRO.Position.x > bulletComponent.ValueRO.maxX
                || transform.ValueRO.Position.y < bulletComponent.ValueRO.minY || transform.ValueRO.Position.y > bulletComponent.ValueRO.maxY)
            {
                entityCommandBuffer.DestroyEntity(entity);
            }
        }
        entityCommandBuffer.Playback(entityManager);
    }
}
```

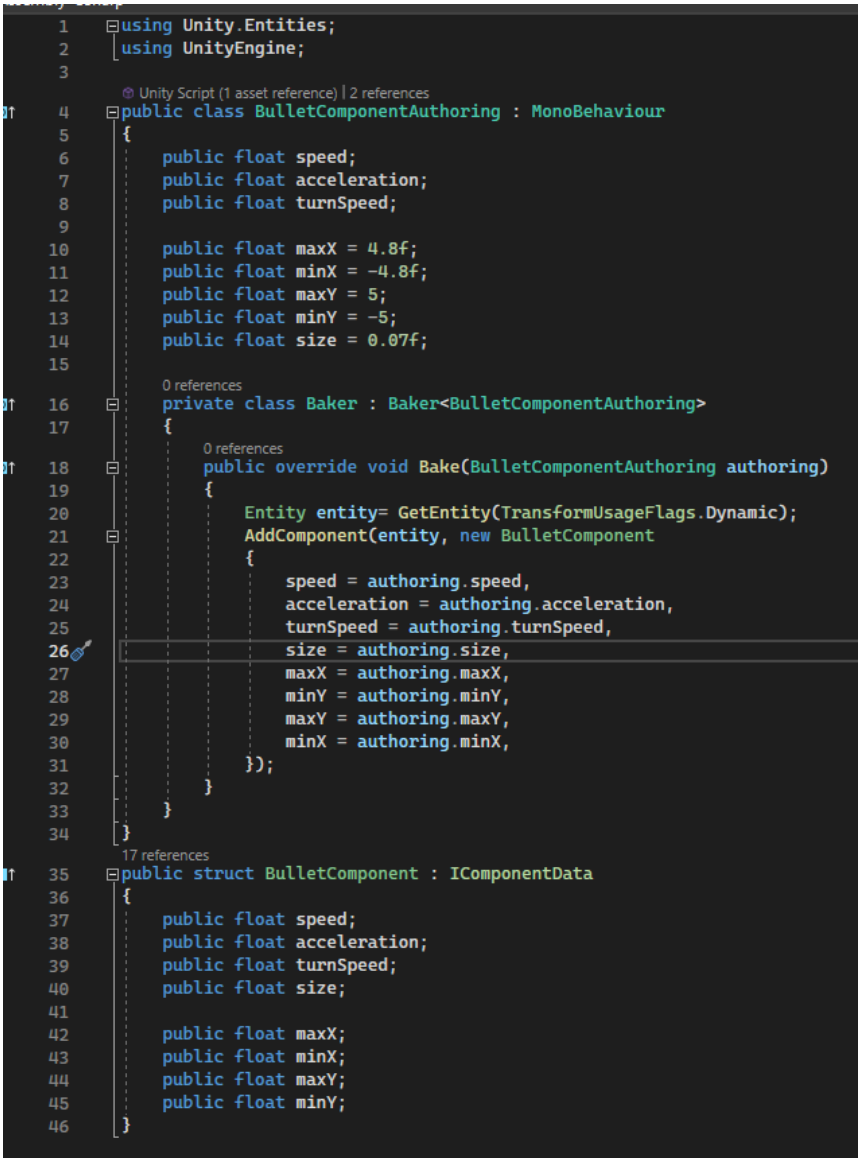
PICTURE 9. System for updating the position of all the bullets in ECS approach. The bullets that went over the game area were destroyed.

3.4.1 Creation of entities

There were two possible approaches for creating entities. One was to create the entity and its components through code in runtime while the other one was to convert gameobject into entity through baking (Unity Technologies, n.d.a). The later approach was chosen for this project for its similarities with gameobject workflows and the possibility to use prefab assets.

For converting gameobjects into entities authoring scripts were made for both bullet spawner and bullet prefabs. The authoring script inherited the monobehaviour class like other scripts tied to gameobjects and exposes the same variables as the corresponding component to allow editing the values of said variables from editor. The same script also has implementation of abstract baker class that converts the gameobject into entity and maps the variables from authoring class into corresponding ECS component that got tied to the created entity. Example of authoring and baker classes used for converting bullets can be seen in picture 10.

In cases where there was gameobject reference to prefab, for example with bullet spawner having reference to bullet prefab, that prefab can also be baked by calling bakers getEntity method with reference to the gameobject that needed converting. This was the easiest if not only way to get reference to the entity version of prefab making it possible to effectively instantiate said prefab as an entity.



```
using Unity.Entities;
using UnityEngine;

Unity Script (1 asset reference) | 2 references
public class BulletComponentAuthoring : MonoBehaviour
{
    public float speed;
    public float acceleration;
    public float turnSpeed;

    public float maxX = 4.8f;
    public float minX = -4.8f;
    public float maxY = 5;
    public float minY = -5;
    public float size = 0.07f;

    0 references
    private class Baker : Baker<BulletComponentAuthoring>
    {
        0 references
        public override void Bake(BulletComponentAuthoring authoring)
        {
            Entity entity= GetEntity(TransformUsageFlags.Dynamic);
            AddComponent(entity, new BulletComponent
            {
                speed = authoring.speed,
                acceleration = authoring.acceleration,
                turnSpeed = authoring.turnSpeed,
                size = authoring.size,
                maxX = authoring.maxX,
                minY = authoring.minY,
                maxY = authoring.maxY,
                minX = authoring.minX,
            });
        }
    }
}
17 references
public struct BulletComponent : IComponentData
{
    public float speed;
    public float acceleration;
    public float turnSpeed;
    public float size;

    public float maxX;
    public float minX;
    public float maxY;
    public float minY;
}
```

PICTURE 10. Authoring and component scripts for the bullets. The BulletComponentAuthoring was set to be the only script component of the bullet prefab used with the ECS approach.

3.4.2 Collision detection with mixed entities and gameobjects

The use of entities has its limitations as well, for example systems like animator being inaccessible. For this reason, it was decided to limit the use of ECS only to situations where it is necessary, keeping most of the elements as gameobjects instead. Only objects that will be made as entities are bullets, bullet spawners and different pickups or score items that can be present in large quantities.

The problem with this approach was created due to entities requiring them to be placed under separate entities subscene instead of the normal scene used by gameobjects. While this happens automatically if entities subscene is available, it means that direct interaction between gameobjects and

entities is not possible. For example, collisions between the two cannot be detected. As a workaround for this entity representation of the player and other gameobjects that need to interact with entities was created. Since there is only one player the player controller script tied to the player gameobject was made into singleton. This allowed querying it and through it the transform component of its gameobject each frame from the entity side, making the entity match the location of the player gameobject. This entity representation was then used to catch the interactions with entity bullets. For now, when the collision was detected the colliding bullet was destroyed but in the future information of the collision can be sent through events to the gameobject side.

Even with the entity representation of player character created catching the collisions was more complicated in the ECS side when compared to gameobject interactions. Even if the game itself is two-dimensional ECS only supports 3d colliders. Also, ECS seems to lack onTriggerEnter events used in gameobject side so to detect collisions SphereCastAll method from physics library was used. The method worked in similar manner to raycasting except it created collider to detect hits instead of simple ray. In the case of SphereCastAll method the collider created was a sphere and all colliders hit were returned. (Unity Technologies, n.d.f).

It is to be noted that calling SphereCast or similar functionality each frame requires more resources than updating the location of normal collider would. This caused a significant performance drop when the querying for collisions was done from the bullets due to their large numbers. However, once the logic for querying for collisions was moved to singular player the difference in performance when compared to version without collision detection could not be observed.

3.4.3 ECS results

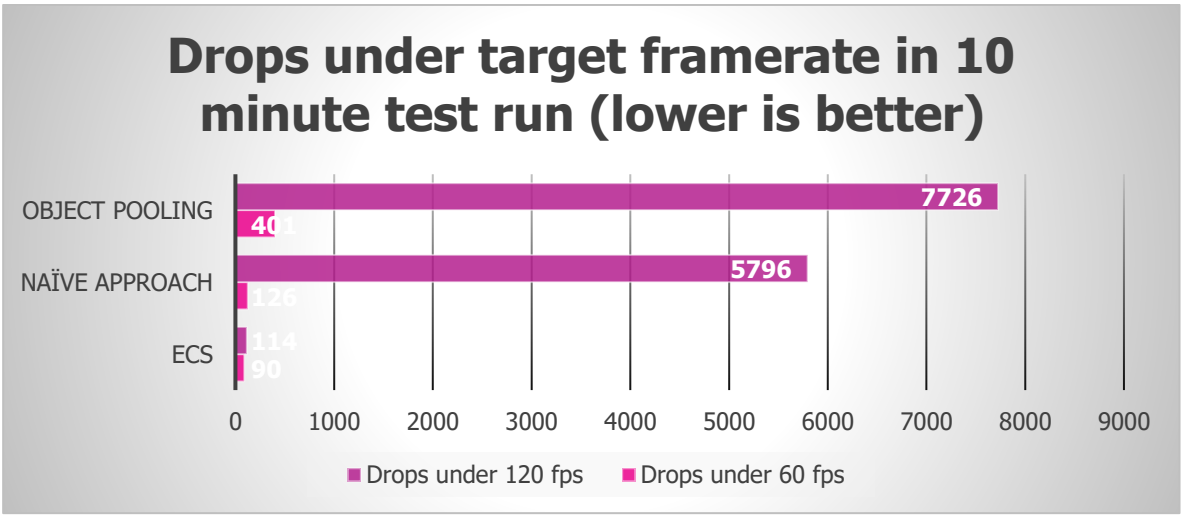
After solving the problems mentioned the results gained from ECS were impressive. Average fps over 10-minute test run was around 540. Furthermore, only 114 frames dropped to under 120 fps, most of which did so due to editor related reasons. Even when the total amount of bullets was increased to over five times the original test pattern ECS version gave acceptable results, were the game to be locked to 60fps. For comparison the increased number of bullets made the naïve approach to drop into an average sub 30 fps.

The effect of multithreading the ECS systems with the job system was left unexplored due to the original research targets being reached without it. However, the ECS architecture should be better suited to multithreading than the original monobehavior code, in the case the need for further optimization would arise in the future.

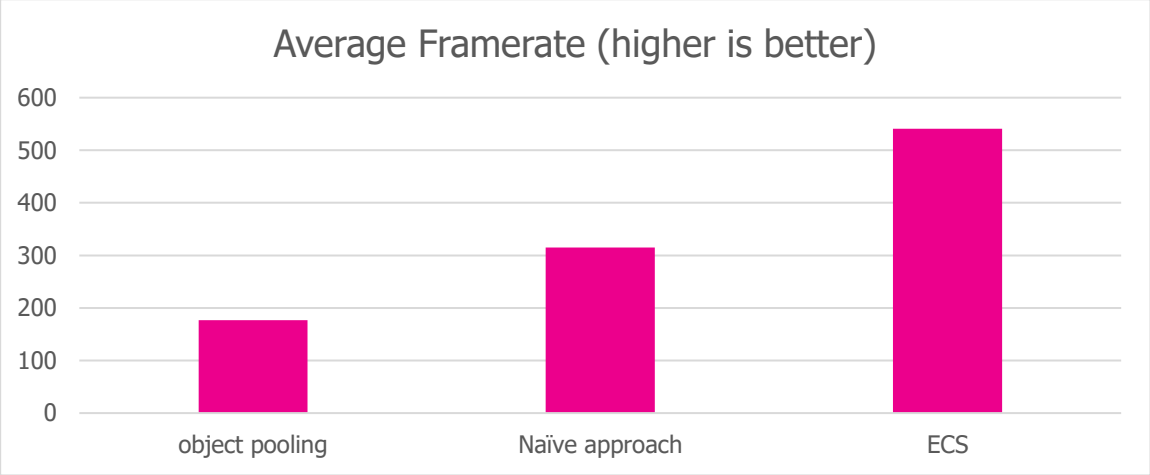
4 RESULTS, CONCLUSIONS AND DISCUSSION

When I started to work with this thesis, I had hypothesis that while Entity Component System and Data Oriented Technology Stack would offer the highest performance gains, they would ultimately not be needed. Clearest area needing optimization with the naïve approach was related to instantiating the bullets and stuttering caused by garbage collection, both issues that based on theory I assumed would have been solved by object pooling. Furthermore, I assumed that object pooling would be easy to work with while ECS and Unity's Data-Oriented Technology Stack (DOTS) would be harder to use. These assumptions appeared to be wrong during the development, however, as the object pooling didn't show any performance improvements, as seen in Pictures 11 and 12, and had plenty of still unsolved problems with breaking the desired bullet pattern. Add to that the problems that would arise when needing to pool multiple different bullet prefabs instead of just one used in the test scene and the option of investigating alternatives started to seem appealing compared to trying to keep on debugging, knowing that the end result would be less performant compared to ECS and DOTS, even if the pooling would have worked perfectly.

ECS on the other hand ended up being vastly easier to implement than originally expected. Plenty of problems and limitations were also encountered, for example the inability to directly interact with gameobjects and lack of animator component. However, these limitations were either solvable or acceptable considering the target application. Creating Entity representation of the player character and synchronizing its position with gameobject version using events made detecting collisions possible. As for the animation, for 2d project animations sprite sheet animations would have been used either way, switching sprite of the object each frame which is doable from the code even without using animator component. All in all, ECS proved to be usable when limited to the areas where it will be necessary. It will not be used for every aspect of the game but for the areas where it will be needed, it solves the problems related to performance, at least as they are caused by the number of bullets.



PICTURE 11. Number of drops under framerate thresholds during 10 minute test runs, comparing performance of object pooling and Entity Component System to the baseline of Naïve approach.



PICTURE 12. Average framerate during during 10 minute test runs, comparing performance of object pooling and Entity Component System to the baseline of Naïve approach.

While the original plan was to test the job system before ECS I ended up deciding against that for two reasons. First, the primary performance issue at the time was stutters caused by garbage collection, not affected by multithreading. The second reason was that as mentioned in the theory section, due to overhead with the small jobs updating the location of the bullets would have likely needed to be done in single instance of manager script, like it was handled in ECS architectures Systems. On the other hand, after ECS results, it was no longer needed to add the job system anymore as the performance goals were already met, even if the two should work together well. This leaves the job system as future potential if the need arises to improve the project further in the future.

Another area of further research arises from the Unity's continuous development of the ECS and DOTS. According to the development roadmap the current issues relating to components one would use with gameobjects missing from ECS side as well as the problems of using gameobjects and entities side by side are being worked on for future releases of the game engine (Unity, 2024).

The project fulfilled all the goals set for it. It acted as a good learning experience, offering my first concrete experience working with ECS and deepening my knowledge of other architectural design patterns. The project also gave clarity for the solutions that will be used in the actual development of the bullet hell game project in the future around performance concerns raised by large amounts of objects. I believe it was good to test the different solutions now using prototype project for testing instead of doing the same investigation of different solutions when encountering problems middle of development of the actual game project.

REFERENCES

GiantBomb. n.d. Bullet Hell. https://www.giantbomb.com/bullet-hell/3015-321/. Referenced 12.4.2025.

Steam. 2016. Enter the Gungeon. https://store.steampowered.com/app/311690/Enter_the_Gungeon/. Referenced 15.4.2025.

Unity. 2024. The Unity Engine Roadmap. Video YouTube-videoservice, published 20.9.2024. https://www.youtube.com/watch?v=pq3QokizOTQ. Referenced 20.5.2025.

Unity Technologies. n.d.a. Baking. https://docs.unity3d.com/Packages/com.unity.entities@1.2/manual/baking.html. Referenced 20.5.2025.

Unity Technologies. n.d.b. Introduction to Object Pooling. https://learn.unity.com/tutorial/introduction-to-object-pooling#. Referenced 2.3.2025.

Unity Technologies. n.d.c. ECS concepts https://docs.unity3d.com/Packages/com.unity.entities@0.17/manual/ecs_core.html Referenced 20.5.2025.

Unity Technologies. n.d.d. Job system overview. https://docs.unity3d.com/Manual/job-system-overview.html. Referenced 10.5.2025.

Unity Technologies. n.d.e. ObjectPool. https://docs.unity3d.com/6000.0/Documentation/ScriptReference/Pool.ObjectPool_1.html. Referenced 3.3.2025.

Unity Technologies. n.d.f. Physics.SphereCastAll. https://docs.unity3d.com/6000.0/Documentation/ScriptReference/Physics.SphereCastAll.html. Referenced 20.5.2025.

Unity Technologies. n.d.g. Understand the ECS workflow. https://docs.unity3d.com/Packages/com.unity.entities@1.4/manual/ecs-workflow-intro.html. Referenced 20.5.2025.

Vacheresse, K. 2023. Improving job system performance scaling in 2022.2 – part 2: Overhead. https://unity.com/blog/engine-platform/improving-job-system-performance-2022-2-part-2. Unity Blog. Referenced 10.5.2025.

Wilmer, L. 2022. Level up your code with game programming patterns. E-book. https://unity.com/resources/level-up-your-code-with-game-programming-patterns/. Referenced 2.3.2024.

Unity Forum. n.d. How to make 2D animation on Unity ECS? https://discussions.unity.com/t/how-to-make-2d-animation-on-unity-ecs/1518532. Referenced 20.5.2025.