Paemeleire Twan

Game engine optimization techniques compared and paired, aimed at bullet-hell games

Supervisor: Vanden Abeele Alex

Coach: Verspecht Marijn

Graduation Work 2025-2026

Digital Arts and Entertainment

Howest.be

CONTENTS

ABSTRACT & KEY WORDS

1. ENGLISH

This paper goes over the implementation and combinations of different optimization techniques for a game that involves a lot of objects being active on the screen at once. This is done in a bullet-hell context, though it may very well be applied to other situations, yielding similar results.

Additionally, a bullet-hell simulation environment is created. We will use it to conduct two tests for every optimization configuration. The first is aimed at rapid spawning and de-spawning of bullets in a short amount of time. While the second is aimed at simulating a ton of bullets on the screen at once, that all need to be calculated upon. Performance metrics such as FPS and GPU/CPU time will be measured during these tests and laid out and compared in graphs.

The results of these tests tell us that the basic create and destroy method becomes very inconsistent and unreliable with lots of objects, due to lag spikes. On the other hand, object pooling helps a lot with the scenario presented in the first test. While both a GPU-compute shader and ECS (even while not used to its full potential due to time limitations) help most with the scenario presented in the second test.

We then also implemented a combination of a GPU-compute shader and pooling, which outperformed every other optimization technique studied in this paper by great lengths.

2. DUTCH

Dit artikel behandelt de implementatie en combinaties van verschillende optimalisatietechnieken voor een game waarin veel objecten tegelijkertijd actief zijn op het scherm. Dit gebeurt in een bullet-hell-context, maar kan ook heel goed worden toegepast op andere situaties, met vergelijkbare resultaten.

Daarnaast wordt een bullet-hell-simulatieomgeving gecreëerd, die we zullen gebruiken om twee tests uit te voeren voor elke optimalisatieconfiguratie. De eerste is gericht op het snel spawnen en de-spawnen van bullets in een korte tijd. De tweede is gericht op het simuleren van een groot aantal bullets tegelijk op het scherm, die allemaal moeten worden berekend. Prestatiestatistieken zoals FPS en GPU/CPU-tijd worden tijdens deze tests gemeten en in grafieken weergegeven en vergeleken.

De resultaten van deze tests laten zien dat de basismethode voor het creëren en vernietigen erg inconsistent en onbetrouwbaar wordt bij veel objecten, als gevolg van lag-pieken. Aan de andere kant helpt object pooling veel bij het scenario dat in de eerste test wordt gepresenteerd. Zowel een GPU-compute shader als ECS (ook al wordt het potentieel ervan vanwege tijdsbeperkingen niet volledig benut) helpen het meest bij het scenario dat in de tweede test wordt gepresenteerd.

Vervolgens hebben we ook een combinatie van een GPU-compute shader en pooling geïmplementeerd, die veel beter presteerde dan alle andere optimalisatietechnieken die in dit artikel zijn onderzocht.

KEY WORDS

Game optimization, object pooling, entity-component system, compute shader, Unity

PREFACE

Ever since I started playing games, I've always been a massive fan of games that provide a hefty challenge to the player. This led me to fall in love with the bullet-hell genre, as it often provides some difficult gameplay and is seen often combined with the roguelike genre, of which I'm also an avid enjoyer.

When I started my road of becoming a game developer, I joined multiple game jams. During one of these game jams, I was tasked with making a bullet-hell-like boss in 2D. It was here that I discovered my love for making bullet-hell focused games. This made me wonder how big titles that fall in this genre, like Returnal, keep their games running smoothly even though an incredible number of bullets are being generated and calculated upon at once.

While this game jam didn't give an opportunity for looking into a lot of optimizations aimed at bullet-hell games, it did very much spark my curiosity to do so. Ever since then, I've started working on my own bullet-hell roguelike as a side project. While I don't have tons of time to spare on it, I've always enjoyed working on it and it strengthened my love for the genre, as well as programming games in general.

As such, when being given the opportunity to choose a research topic, I quickly settled on looking into optimizations aimed at bullet-hell games. Because this is not only something I'm very interested in, but on top of that, the results could potentially contribute to my own personal project. Depending on what I come to conclude, my findings could help a lot in being able to allow my game to simulate way more bullets at once without dropping performance than before.

I'm hoping to learn and evolve as a programmer through this research, while also making some meaningful discoveries on already built optimization techniques by attempting to combine them.

LIST OF FIGURES

INTRODUCTION

Optimization has, and always will be vital in the gaming industry, due to games continuing to push both hardware and software limitations. Because of this, I thought it would be interesting and valuable to analyze and compare different optimization techniques. These will be tested both on their own and in pairs, to see what the influence of combining them may be on both implementation effort and performance changes.

The findings discussed in this paper are in no way exclusive to a bullet-hell setting, this was just the environment that I aimed the chosen optimization techniques at. The findings in this paper can be applied to any situation involving the simulation of lots of objects on screen, such as particle systems, crowd simulation, etc. Obviously, some changes will need to be made to adapt to the specific use case, but the main findings of this paper should still hold up.

The paper tries to answer the specific following question:

How do object pooling, GPU compute shaders, and an Entity-Component System (ECS) individually and pairwise affect runtime performance in a 2D bullet-hell simulation across varying projectile counts and behavior complexities?

Besides a specific research question, hypotheses are presented below. These will be concluded upon based on the findings listed further in the paper.

H0: Combining different techniques doesn't increase performance more compared to using them on their own individually, due to the overhead required to use them together in a correct manner.

H1: Combining techniques (pooling + ECS, pooling + GPU compute shader) increases runtime performance compared to single techniques when projectile counts are high, but may be neutral or slightly worse at low bullet counts due to overhead.

H2: Implementing combinations, especially pooling + ECS, increases development effort compared to single techniques. Pooling + ECS might not even be possible due to the vastly different nature and structure of ECS projects.

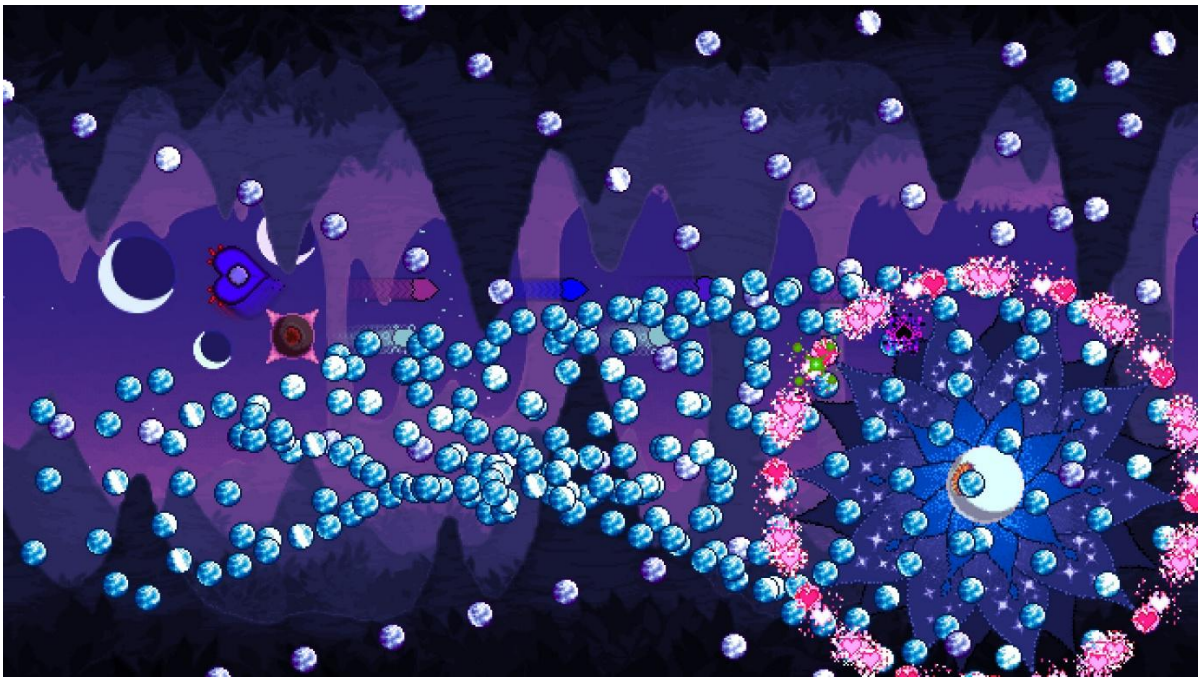
H3: The benefit of GPU compute shaders increases with per-bullet computational complexity (for example: homing logic, collision checks) while pooling gives constant benefits regardless of per-bullet compute. ECS gives largest benefits when many components are present or when cache locality matters.

LITERATURE STUDY / THEORETICAL FRAMEWORK

Before going over technical topics, we start with a brief introduction to bullet-hell games, so that the use-cases for this research may be better understood.

Bullet-hell is a specific subgenre of shoot 'em up games that can be primarily identified by the large number of projectiles a player is tasked with avoiding. Some well-known modern bullet-hell games include Enter the Gungeon, Returnal and Vampire Survivors. Because of how precise the gameplay is when avoiding a bullet sometimes, it becomes very noticeable when any frame-hiccups or other performance issues happen. Combining this with how punishing bullet-hells often are, it's incredibly important for bullet-hell games to run well.

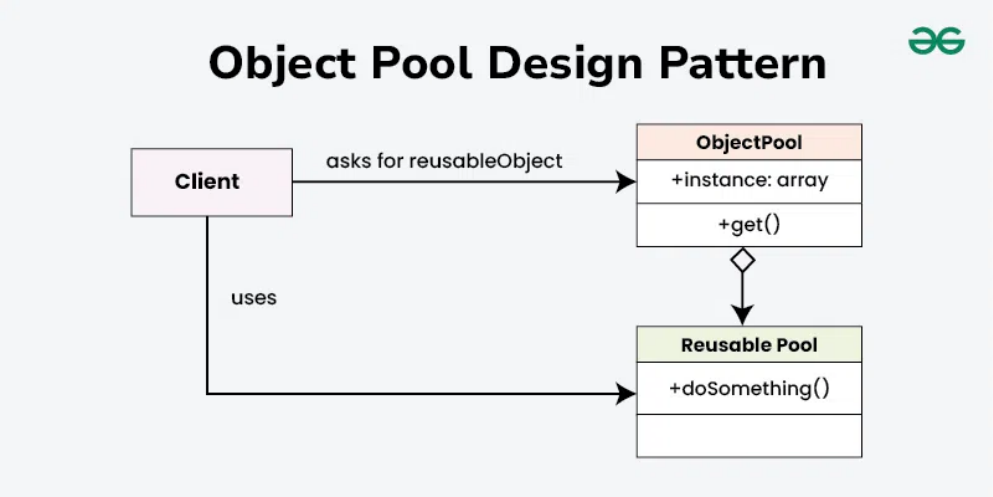
PICTURE 1 (Returnal by Housemarque, 2021)



PICTURE 2 (The Void Rains Upon Her Heart by Veyeral Games, 2018)

The images above display 2 of the most common situations in bullet-hell games. PICTURE 1 shows an example of situations where big bursts of bullets get spawned at once/with very little time between one another. PICTURE 2 shows an example of situations where many bullets are active at once on the screen. Both scenarios happen continuously in this genre of games. As such, it is extremely important for them to be optimized well, so that no lag occurs during these moments. There are a few possible optimization techniques for these situations, which we explore in the next chapter.

Object Pooling is a software design pattern, in which a "pool" of initialized objects are kept ready to be used by the program, rather than having to be created and destroyed on demand constantly, it can drastically lower the burden placed on the CPU when having to rapidly create and destroy objects (Unity Technologies, 2025). In our case, this would be a pool of bullets, a client of the pool can then request a bullet object and perform operations on it. When the client is done with said bullet, it will return it to the pool to be reused, rather than destroy it.



PICTURE 3 (Object Pool Design Pattern by GeeksforGeeks, 2024)

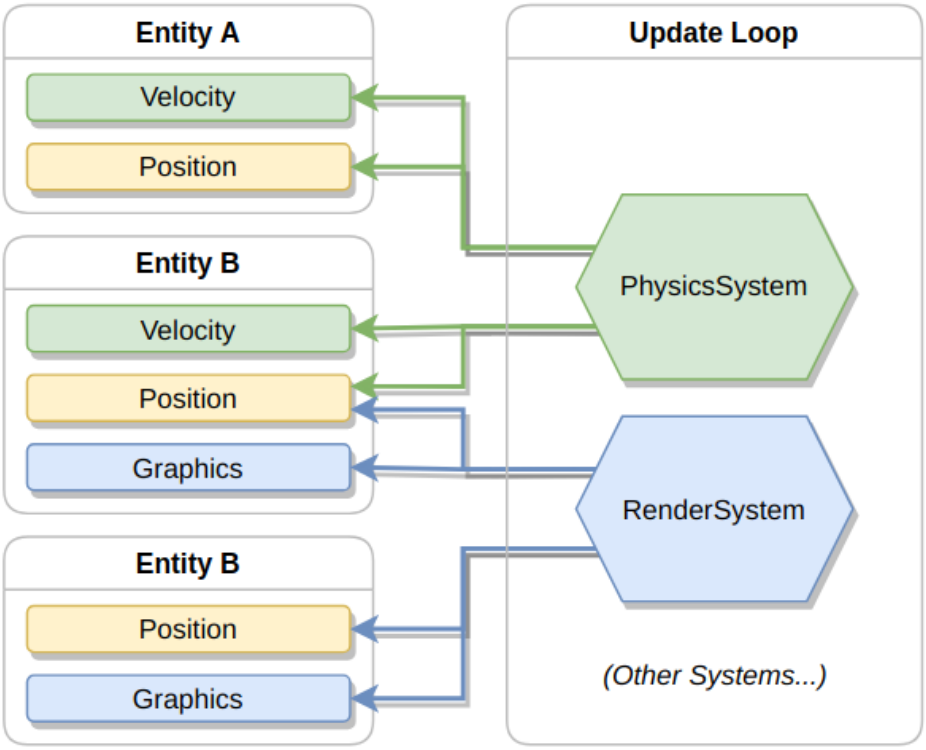
While object pooling can help improve performance a lot, it's not useful or even an advantage in all situations. Object pooling should be used in situations where, for example, object creation and/or destruction is expensive or there is limited resource availability. However, object pooling should be used mindfully as it does have some drawbacks, such as the added maintenance overhead. Additionally, it also adds object complexity, because objects need to be reset when they are returned to the pool, which can cause some issues that would not be present without the use of this pattern. Another potential downside of using an object pool is that it may waste memory on unneeded objects if the usage pattern of the objects is so irregular that they are barely reused or if it is impossible to estimate the required pool size (Robert Nystrom, 2021). To sum up object pooling, the most important thing is to use it in the appropriate situations, if used and implemented well, it can be a massive improvement to game performance stability. It can help drastically reduce the number of negative spikes in performance during moments when a lot of objects need to be spawned.

The next optimization technique that will be investigated is using a **GPU-compute shader**. This entails the use of a graphics processing unit, which typically handles computation only for computer graphics, to perform computations in applications traditionally handled by the central processing unit (Wikipedia, 2025). The main problem of simulating a bullet-hell (or other things like particle systems) on the CPU is the performance cost that comes with computing many objects. Transferring all calculations (such as movement, collision, out-of-bounds checking) from the CPU to the GPU can severely increase performance (Onufriienko D. M., 2024).

The reason why moving these calculations to the GPU speeds them up is because the GPU can split the large number of calculations into smaller groups and then run all calculations in a parallel manner. While GPUs operate at lower frequencies, they typically have many more processing units than a CPU does. Which makes it so that GPUs can process far more data per second than a traditional CPU. This larger amount of processing units gets put to its advantage by using an execution model called SIMT (Single instruction, multiple threads), SIMT is a model used in parallel computing where a single central control unit sends an instruction to multiple processing units for them to then all run simultaneous, synchronous and fully independent parallel execution of that one instruction. (Wikipedia, 2025)

Using a GPU-compute shader helps most with the scenario shown in PICTURE 2. It can help massively with keeping smooth performance when an extraordinarily large number of bullets/objects are present on screen at once and each of them needs individual calculations executed on them.

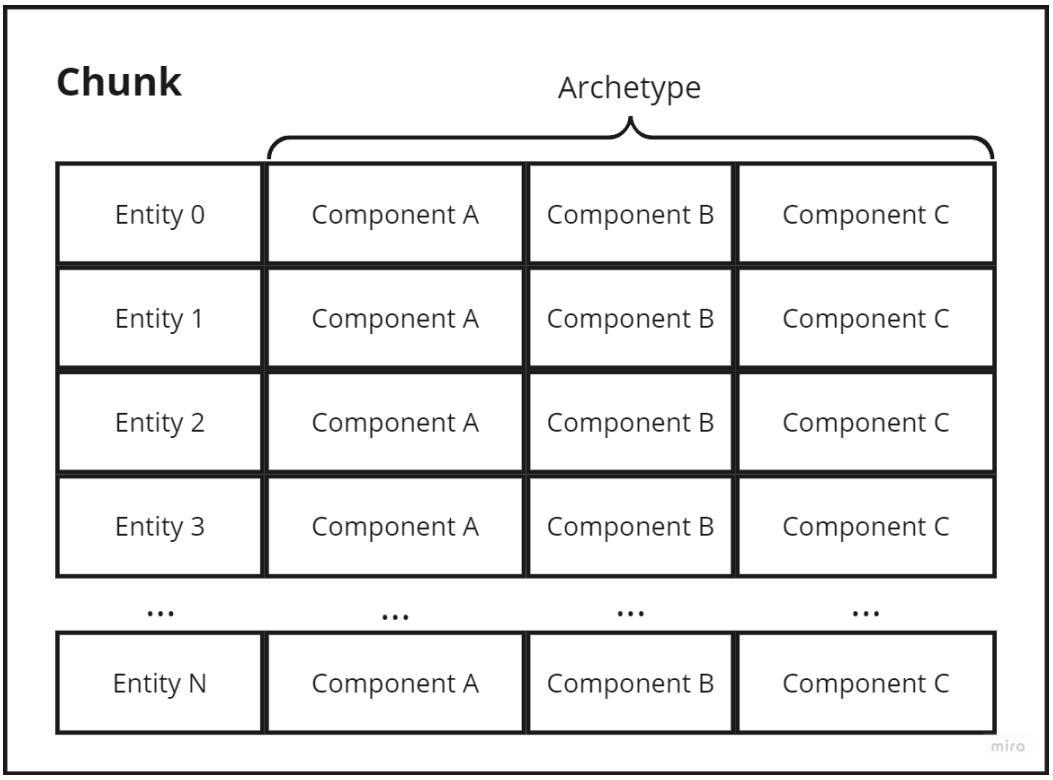
The final optimization technique I will be exploring is using an **ECS (Entity component system)** pattern. An entity is what represents a general-purpose object, for example in a game engine every game object is represented as an entity (Wikipedia, 2025). A component is something that encapsulates both a specific piece of behavior/functionality and data. This allows them to interact through defined interfaces without the need to expose the inner workings of each individual component. (GeeksforGeeks, 2025) Game engines like for example Unity use these components in a CBA (Component based architecture), this means that each entity will exist out of multiple components, which define their behavior. This means that to update all the entities, an engine using a CBA will sequentially loop over every game object (entity) and all its components. A downside of this approach is that all these components and game objects are in no way contiguous in memory, which doesn't make for good cache usage, causing performance to take a hit because of all the cache misses that happen.



PICTURE 4 (React, 2025)

An ECS architecture moves logic away from components, it prioritizes composition over inheritance. Now, every entity is not defined by a type-hierarchy, but rather by the components associated with it. The big difference with a CBA is that these components no longer hold any behavior/logic, rather they hold exclusively data. This data is then acted on by systems. Systems act globally over all entities that have the required components (Wikipedia, 2025). This can be seen in PICTURE 4. The physics system acts only on entities that have both a velocity and position component, while the render system acts only on entities having both a graphics and position component. When an entity doesn't have all required components for a system, that entity will be ignored by the system.

Circling back to the high amount of cache-misses issue that comes with a CBA, an ECS architecture actually solves this problem. The ECS architecture allows for laying the memory out in a different, way more efficient method. I'll be going over how Unity does it specifically, because my experiment will be done in this engine, though ECS implementations outside of this engine will use a similar way of going about memory layout.



PICTURE 5 (Yeray Tarifa Mateo, 2024)

The Unity DOTS (Data-Oriented Technology Stack, which includes the ECS functionality) uses an archetype approach for organizing the memory layout of both entities and components. This means that all objects that adhere to an archetype get grouped together. This is done by storing every archetype in a different chunk (a 16KB block of memory in this case) that contains several entities and components that correspond to said archetype. This chunk is separated into a variable number of arrays, that are all laid out contiguously in memory. The number of arrays depends on the number of different components the archetype has. In this case there would be 4 different arrays, 1 for the entity ID's and the three others are for the three different component types. These components all need to be in different arrays, because the data they hold (and thus their size in memory) will be different. If the chunk reaches its size limit, a new chunk for the same archetype will be created. (Yeray Tarifa Mateo, 2024)

The central takeaway from the previous paragraph is that these components are now laid out contiguously in memory. This means the systems that operate on them can very easily and quickly loop over them all, with near-perfect cache usage. While using an ECS architecture does require you to think about structuring your game and code in a completely different way, it comes with the advantage of some considerable performance improvements, due to using a data-based architecture. Using a data-oriented approach is especially useful in games where lots of entities need to be simulated at a time, as such, this is perfect for a bullet-hell where every bit of optimization counts.

In the next section, I'll be going over how each of these three optimization techniques compare in different situations. I will also be trying out combinations of them, to see if this improves performance even further, or causes bottlenecks in the collaboration of two of the techniques.

RESEARCH

1. EXPERIMENT STRUCTURE

1.1. BULLET-HELL TESTING ENVIRONMENT

For the first part of my experiment, I made a bullet-hell environment that will be used to run tests in. This environment is basically a simulation of a bullet-hell scenario. As shown in PICTURE 6, its main purpose is to say when a bullet should be spawned and in what direction it should go. However, the system responsible for spawning and controlling the bullet itself depends entirely on what technique we are using. Each of these techniques has its own logic, from now on I will refer to these different scripts for each technique as "handlers".

```
for (int shooterIdx = 0; shooterIdx < _currentSimulationData.NumberOfShooters; shooterIdx++)
{
    // Calculate direction
    float angleInRadians = _shooterAngles[shooterIdx] * Mathf.Deg2Rad;
    Vector2 shootDirection = new Vector2(Mathf.Cos(angleInRadians), Mathf.Sin(angleInRadians)).normalized;

    // Request bullet spawn (handled externally)
    BulletData bulletData = new BulletData
    {
        SpawnPosition = transform.position,
        SpawnDirection = shootDirection,
        MovementSpeed = _currentSimulationData.BulletMoveSpeed
    };
    _currentBulletHandler.OnBulletSpawnRequested(bulletData, _bulletParentTransform);

    // Wait between shooters
    if (_currentSimulationData.DelayBetweenShooters > 0)
    {
        if (_currentSimulationData.ContinueRotationDuringShooterDelay) StopCoroutine(_shooterRotationCoroutine);
        yield return new WaitForSecondsRealtime(_currentSimulationData.DelayBetweenShooters);
        if (_currentSimulationData.ContinueRotationDuringShooterDelay) _shooterRotationCoroutine = StartCoroutine(ShooterRotation());
    }
}
```

PICTURE 6 (Bullet-hell simulation)

1.2. HANDLERS

A handler is the term that will be used to refer to a script that handles the creation, destruction and updating of bullets in this simulation. This entails that every optimization method (and combinations of them) each have their own respective handler. This is done so that logic is completely separated as can also be tested upon as such.

1.2.1. CREATE & DESTROY HANDLER

This handler is the very basic implementation of spawning and destroying objects. This means that whenever a bullet is created, new memory is allocated, and whenever one is destroyed, this memory gets cleared. An implementation like this is seen as the completely unoptimized version. However, behind the scenes, Unity's garbage collector (GC) does help a bit with reducing object destruction related performance spikes. The GC will scan all managed objects to determine which are still in use, and which it may clean up. By default, this process happens by dividing the work over multiple frames, which can significantly reduce the amount of GC spikes, making object destruction (especially in a case like a bullet-hell) much more stable. (Unity Technologies, 2025)

```
2 references | 0 changes | 0 authors, 0 changes
public override void OnBulletSpawnRequested(BulletData bulletData, Transform bulletParent)
{
    SimpleBullet bullet = Instantiate(_bulletPrefab).GetComponent<SimpleBullet>();
    bullet.OnDestroyRequested.AddListener(OnBulletDestroyRequested);
    bullet.InitializeBullet(bulletData);
    bullet.transform.SetParent(bulletParent, true);
}

2 references | 0 changes | 0 authors, 0 changes
public override void OnBulletDestroyRequested(SimpleBullet bullet)
{
    Destroy(bullet.gameObject);
}
```

PICTURE 7 (Create & Destroy handler)

As seen in PICTURE 7, the create & destroy handler is very simple. Whenever a bullet spawn is requested, a new object is created from a prefab. We then hook into this bullet's OnBulletDestroyRequested event, which is invoked when going out of bounds, by a script on the bullet prefab. Finally, when the event triggers, we just destroy the game object.

1.2.2. OBJECT POOLING HANDLER

The second handler that I implemented is the object pooling handler. While Unity already has its own object pooling implementation, I decided to make one myself to ensure full transparency, and so that I may be able to prewarm the pool.

```
1 reference | 0 changes | 0 authors, 0 changes
private void CreateInstance()
{
    var projectileObject = Instantiate(_prefab);
    projectileObject.SetActive(false);
    projectileObject.transform.SetParent(_bulletParent);
    SimpleBullet bullet = projectileObject.GetComponent<SimpleBullet>();
    _allBullets.Add(bullet);
    _inactiveBullets.Push(bullet);
}

2 references | 0 changes | 0 authors, 0 changes
public SimpleBullet Get(BulletData data)
{
    if (_inactiveBullets.Count == 0) CreateInstance();
    var projectile = _inactiveBullets.Pop();
    projectile.InitializeBullet(data);
    return projectile;
}

2 references | 0 changes | 0 authors, 0 changes
public void Release(SimpleBullet bullet)
{
    bullet.gameObject.SetActive(false);
    _inactiveBullets.Push(bullet);
}
```

PICTURE 8 (Custom object pool)

This object pool implementation is nothing out of the ordinary, it has a stack of inactive bullets, that we just pop from when a bullet is requested. When there are no bullets left on this stack, we create a new instance and add it.

```
2 references | 0 changes | 0 authors, 0 changes
public override void OnBulletSpawnRequested(BulletData bulletData, Transform bulletParent)
{
    SimpleBullet bullet = _objectPool.Get(bulletData);
    bullet.OnDestroyRequested.AddListener(OnBulletDestroyRequested);
    bullet.gameObject.SetActive(true);
}
3 references | 0 changes | 0 authors, 0 changes
public override void OnBulletDestroyRequested(SimpleBullet bullet)
{
    bullet.OnDestroyRequested.RemoveListener(OnBulletDestroyRequested);
    _objectPool.Release(bullet);
}
```

PICTURE 9 (Object pooling handler)

The handling of spawning and destroying is still relatively easy, the only thing that changed compared to the create & destroy handler is that we now get an instance from the pool and then also release it back, instead of creating and destroying, which avoids having to instantiate and clean up memory all the time. Additionally, we also need to ensure that we un-hook from the event when destruction is requested, otherwise we would progressively add more listeners to the event, which would break both logic, and over time, performance.

1.2.3. GPU COMPUTE SHADER HANDLERS

The third and fourth handlers that I made were done using a GPU compute shader. Originally, I was planning on only having one handler that uses a GPU compute shader. However, while working on writing this handler, I realized that I could go about using the compute shader in two different ways. One would do its pooling on the CPU side, and the other handler would then fully manage all its pooling on the GPU.

1.2.3.1. CPU SIDE POOLING HANDLER

As seen in PICTURE 10, when managing the pooling on the CPU side, it requires us to fetch all data from the GPU, loop over every object, check which are active, and then update the pool according to said data.

```
Unity Message | 0 references | 0 changes | 0 authors, 0 changes
private void Update()
{
    if (!_isRunning) return;

    // Dispatch compute shader
    _computeShader.SetFloat("deltaTime", Time.deltaTime);
    int groups = Mathf.CeilToInt((float)_maxBullets / _numberOfThreads);
    _computeShader.Dispatch(_updateKernel, groups, 1, 1);
    ReclaimDeadBullets();
    DrawBullets();
}
1 reference | 0 changes | 0 authors, 0 changes
private void ReclaimDeadBullets()
{
    GPUBullet[] bullets = new GPUBullet[_maxBullets];
    _bulletBuffer.GetData(bullets);

    _freeListCount = 0;

    for (int i = 0; i < _maxBullets; i++)
    {
        if (bullets[i].active == 0)
        {
            _freeList[_freeListCount++] = i;
        }
    }
}
```

PICTURE 10 (GPU compute shader with CPU pooling handler)

The way that we do our pooling is by keeping a free list, which holds indices of the bullets on the GPU that are currently free to be used to spawn a bullet.

```
2 references | 0 changes | 0 authors, 0 changes
public override void OnBulletSpawnRequested(BulletData bulletData, Transform bulletParent)
{
    if( _freeListCount <= 0)
    {
        Debug.LogWarning("No free bullets available in pool");
        return;
    }

    // "Pop" from free list
    int index = _freeList[--_freeListCount];
    // Spawn bullet
    float zPlane = 0f;
    GPUBullet bullet = new GPUBullet
    {
        position = new Vector3(bulletData.SpawnPosition.x, bulletData.SpawnPosition.y, zPlane),
        velocity = bulletData.SpawnDirection * bulletData.MovementSpeed,
        active = 1
    };
    _bulletBuffer.SetData(new GPUBullet[] { bullet }, 0, index, 1);
}
```

PICTURE 11 (CPU compute shader with CPU pooling spawn handling)

So then when a bullet spawn is requested, we “pop” an index from the free list, create a new bullet object and then write it to the specified free index in the bullet buffer on the GPU.

I later realized that a more efficient way to do this would be to keep track of a buffer on the GPU that holds the indices of the bullets that went out of bounds that frame. The CPU would then read and clear this buffer every frame and update its free list accordingly. It doesn't remove the constant fetching data from the GPU, but it would remove the need to loop over all bullets every frame. Unfortunately, due to time limitations, this additional optimization was not implemented nor tested.

PICTURE 12 showcases how the GPU compute shader itself functions when it gets dispatched as seen in PICTURE 10. The GPU will loop over all bullets in parallel, it will then continue to update all active bullets, and check whether some might be out of bounds. If a bullet goes out of bounds, its active flag gets set to false once the GPU is done. The CPU will then read this data back and see that this bullet is inactive, adding its index to the free list as shown in PICTURE 10.

```
struct Bullet
{
    float3 position;
    float3 velocity;
    uint active;
};
RWStructuredBuffer<Bullet> bullets;
uint bulletCount;
float deltaTime;
float3 boundsMin;
float3 boundsMax;
[numthreads(256, 1, 1)]
void BulletUpdateComputeShader(uint id : SV_DispatchThreadID)
{
    if (id >= bulletCount) return;
    Bullet b = bullets[id];
    if (b.active == 0) return;

    // Move bullet
    b.position += b.velocity * deltaTime;
    // Kill if outside of bounds
    if (b.position.x < boundsMin.x || b.position.x > boundsMax.x ||
        b.position.y < boundsMin.y || b.position.y > boundsMax.y)
    {
        b.active = 0;
    }
    bullets[id] = b;
}
```

PICTURE 12 (GPU compute shader with CPU pooling bullet update compute shader)

1.2.3.2. GPU SIDE POOLING HANDLER

The major difference with doing pooling on the GPU is that there's no longer any need to "sync" up data between the CPU and GPU. All the CPU does is request a bullet spawn, and then the GPU fully handles all the rest.

```
Unity Message | 0 references | 0 changes | 0 authors, 0 changes
private void Update()
{
    if (!_isRunning) return;
    // Dispatch compute shader
    _computeShader.SetFloat("deltaTime", Time.deltaTime);
    int groups = Mathf.CeilToInt((float)_maxBullets / _numberOfThreads);
    _computeShader.Dispatch(_updateKernel, groups, 1, 1);
    DrawBullets();
}
2 references | 0 changes | 0 authors, 0 changes
public override void OnBulletSpawnRequested(BulletData bulletData, Transform bulletParent)
{
    _computeShader.SetVector("spawnPosition", bulletData.SpawnPosition);
    _computeShader.SetVector("spawnVelocity", bulletData.SpawnDirection * bulletData.MovementSpeed);
    _computeShader.Dispatch(_spawnKernel, 1, 1, 1);
}
```

PICTURE 13 (GPU compute shader with GPU side pooling handler)

However, when doing the pooling on the GPU, it does require us to use 2 additional kernels as seen in PICTURE 14. The initialize kernel is used once at startup to populate the pool. The spawn kernel is used much more often (whenever a bullet spawn is requested), it takes an index from the inactive indices buffer, then sets its position and other data. Finally, it writes this data back into the bullet buffer.

```
#pragma kernel BulletSpawnComputeShader
float3 spawnPosition;
float3 spawnVelocity;
[numthreads(1, 1, 1)]
void BulletSpawnComputeShader(uint id : SV_DispatchThreadID)
{
    uint bulletIndex = inactiveBulletIndicesConsume.Consume();
    Bullet b;
    b.position = spawnPosition;
    b.velocity = spawnVelocity;
    b.active = 1;
    bullets[bulletIndex] = b;
}

#pragma kernel BulletPoolInitComputeShader
[numthreads(256, 1, 1)]
void BulletPoolInitComputeShader(uint id : SV_DispatchThreadID)
{
    if (id >= bulletCount) return;
    inactiveBulletIndices.Append(id);
}
```

PICTURE 14 (GPU compute shader with GPU side pooling spawn and initialize kernels)

The actual bullet update logic is once again quite simple, the GPU just loops over all bullets in parallel and updates the active ones. Finally, it checks whether the bullet is now out of bounds, if it is, we set the active flag of the bullet to false. To ensure our pooling works correctly we then append it to the buffer of inactive bullet indices, which makes the object pooling fully GPU based, which leaves the CPU to only have to set spawn data and then request a bullet spawn as shown in PICTURE 13.

```
[numthreads(256, 1, 1)]
void BulletUpdateWithPoolingComputeShader(uint id : SV_DispatchThreadID)
{
    if (id >= bulletCount) return;
    Bullet b = bullets[id];
    if (b.active == 0) return;

    // Move bullet
    b.position += b.velocity * deltaTime;
    // Kill if outside of bounds
    if (b.position.x < boundsMin.x || b.position.x > boundsMax.x ||
        b.position.y < boundsMin.y || b.position.y > boundsMax.y)
    {
        b.active = 0;
        inactiveBulletIndices.Append(id);
    }
    bullets[id] = b;
}
```

PICTURE 15 (GPU compute shader with GPU side pooling update compute shader)

1.2.3.3. SIMILARITIES

As seen in PICTURE 12 and PICTURE 15, both pooling on the CPU and GPU do have their main GPU compute shader logic in common. They each will go over all the bullets in parallel, every frame. They will calculate which are out of bounds and then handle the deactivation each in their respective way. Additionally, the drawing of objects is also

done in an identical way. Each has a position and then DrawMeshInstancedIndirect is used along with a shader placed on a material to correctly draw all bullets using GPU instancing.

```
1 reference | 0 changes | 0 authors, 0 changes
private void DrawBullets()
{
    _bulletMaterial.SetBuffer("_Bullets", _bulletBuffer);
    Bounds bounds = new Bounds(Vector3.zero, Vector3.one * 10000f);
    Graphics.DrawMeshInstancedIndirect(QuadMesh.Get(), 0, _bulletMaterial, bounds, _argsBuffer);
}
```

PICTURE 16 (GPU compute shader bullet instance drawing)

One major downside of doing the drawing and computing like this, is that we are no longer dealing with game objects, this makes debugging very tricky. But the biggest drawback of not dealing with game objects in this case is the fact that we lose all built in functionality from Unity components like for example the animator component. I thought of 2 ways to work around this, the first way would be to reimplement functionality like the animator on the GPU as well. This could be done by keeping track of frames and making a custom simple "flipbook" animation system. The other way we could achieve combining game objects with a GPU compute shader would be to find a way to efficiently sync data between CPU and GPU, we could then do the calculations on the GPU, fetch them on the CPU, and then apply these changed positions and other data to existing game objects. While neither of these two were looked into/implemented, it could be worth looking into to see if there's any value to an approach like this, or if it adds more complexity without worthy performance gains.

1.2.4. ECS HANDLERS

The final handler that was implemented was the ECS handler. As described in the theoretical framework, ECS works completely differently than any of the other optimization techniques, due to it requiring a vastly different programming mindset.

Because of my bullet-hell environment being made using regular game objects, I had to work around some things, usually the bullet-hell simulation would be a system. However, due to the implementation I did, I had to do some additional communication between regular game objects and ECS systems.

```
2 references | 0 changes | 0 authors, 0 changes
public override void OnBulletSpawnRequested(BulletData bulletData, Transform bulletParent)
{
    // Spawn bullet via ECS
    Entity request = _entityManager.CreateEntity(typeof(BulletSpawnRequest));

    _entityManager.SetComponentData(request, new BulletSpawnRequest
    {
        Position = bulletData.SpawnPosition,
        Direction = bulletData.SpawnDirection * bulletData.MovementSpeed,
        Scale = _bulletScale
    });
}
```

PICTURE 17 (ECS handler)

When a bullet spawn is requested, the ECS handler will create an entity via the entity manager and add a BulletSpawnRequest component to it, which holds data about how the bullet should spawn and behave.

```
15 references | 0 changes | 0 authors, 0 changes
public struct BulletSpawnRequest : IComponentData
{
    public Vector3 Position;
    public Vector2 Direction;
    public float Scale;
}
1 reference | 0 changes | 0 authors, 0 changes
partial struct BulletSpawnSystem : ISystem
{
    [BurstCompile]
    0 references | 0 changes | 0 authors, 0 changes
    public void OnUpdate(ref SystemState state)
    {
        Entity bulletEntity = SystemAPI.GetSingleton<BulletPrefabReference>().BulletEntity;
        var entityCommandBuffer = SystemAPI.GetSingleton<EndSimulationEntityCommandBufferSystem.Singleton>().CreateCommandBuffer(state.WorldUnmanaged);
        foreach (var (request, entity) in SystemAPI.Query<RefRO<BulletSpawnRequest>>().WithEntityAccess())
        {
            Entity bullet = entityCommandBuffer.Instantiate(bulletEntity);
            entityCommandBuffer.SetComponent(bullet,
                LocalTransform.FromPositionRotationScale(request.ValueRO.Position, Unity.Mathematics.quaternion.identity, request.ValueRO.Scale));
            entityCommandBuffer.SetComponent(bullet, new MoveDirection
            {
                MoveDirectionValue = request.ValueRO.Direction
            });
            entityCommandBuffer.DestroyEntity(entity);
        }
    }
}
```

PICTURE 18 (ECS bullet spawn system)

The bullet spawn system is responsible for detecting and consuming spawn requests. The system will query for all entities that have a bullet spawn request component. It will then continue to loop over them all, creating a new entity for each. This entity it created based on a sort of “prefab” entity, after which we set its transform and move direction components, to ensure that the bullet behaves as requested. The final step in the spawning progress is to then “consume” the request and delete the entity corresponding to it.

```
1 reference | 0 changes | 0 authors, 0 changes
partial struct BulletMoverSystem : ISystem
{
    [BurstCompile]
    0 references | 0 changes | 0 authors, 0 changes
    public void OnUpdate(ref SystemState state)
    {
        var bounds = SystemAPI.GetSingleton<CameraBounds>();
        var entityCommandBuffer = SystemAPI.GetSingleton<EndSimulationEntityCommandBufferSystem.Singleton>()
                                           .CreateCommandBuffer(state.WorldUnmanaged);
        foreach ((RefRW<LocalTransform> localTransform, RefRO<MoveDirection> moveDirection, Entity entity)
            in SystemAPI.Query<RefRW<LocalTransform>, RefRO<MoveDirection>>().WithEntityAccess())
        {
            Vector2 direction = moveDirection.ValueRO.MoveDirectionValue;
            localTransform.ValueRW.Position += new Unity.Mathematics.float3(direction, 0)  * SystemAPI.Time.DeltaTime;

            if (localTransform.ValueRO.Position.x < bounds.Min.x || localTransform.ValueRO.Position.x > bounds.Max.x ||
                localTransform.ValueRO.Position.y < bounds.Min.y || localTransform.ValueRO.Position.y > bounds.Max.y) // Bullet is out of bounds
            {
                entityCommandBuffer.DestroyEntity(entity);
            }
        }
    }
}
```

PICTURE 19 (ECS bullet move system)

PICTURE 19 shows the system responsible for the main logic of moving all bullets and checking when they go out of bounds. Every frame, it will query for entities that have both a LocalTransform and MoveDirection component. It will then loop over all found entities, update their position and check whether they have left the specified bounds. When leaving the bounds, the entity will be destroyed.

Additionally, this code can be heavily sped up using the Unity Job System, unfortunately due to time constraints this was not implemented nor tested. A final note on PICTURE 19 is the use of RefRW (Read-Write) and RefRO

(Read-only), we should always use RefRO where possible. RefRO is used in PICTURE 19 on the MoveDirection component, because we never change any of its data, only access it. On the other hand, we are forced to use RefRW for the LocalTransform component, because we want to make changes to the entity's position data, so we need write access.

2. EXPERIMENT

2.1 SETUP AND MEASUREMENT

While running the bullet-hell simulation, a script is running in the background which polls GPU, CPU and FPS every frame. From this data we then later get both average and median values according to their respective calculations. These average and median values will be the values displayed in the graphs below.

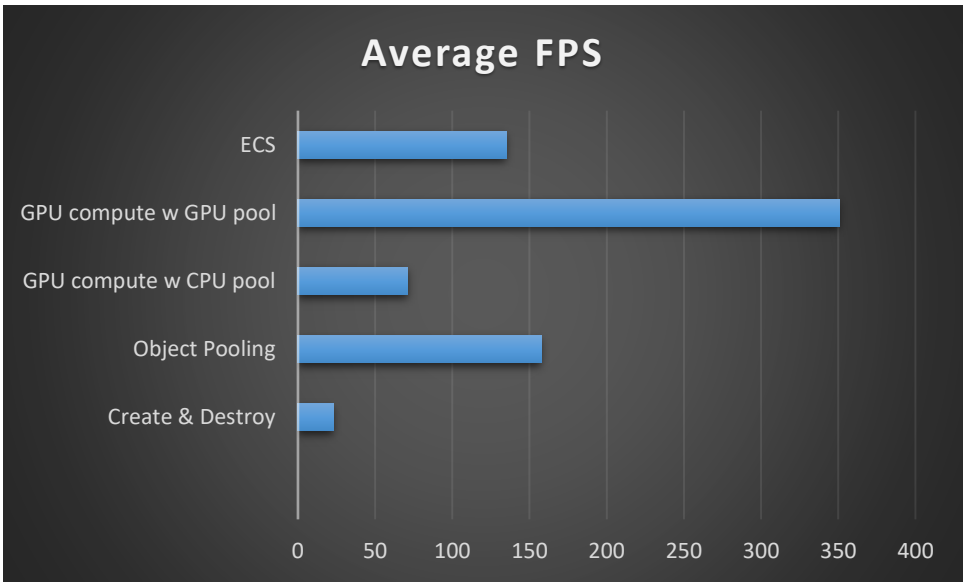
For the test itself, we will be running two simulations, each of these focuses on one of two important bullet-hell situations, as mentioned in the theoretical framework. The first one constantly spawns a lot of bullets (6400/second) that move very quickly, causing a constant stream of a lot of spawning and de-spawning. The second one spawns less bullets (200/second) that move very slowly, which makes it so a ton of bullets will be on screen and in need of being simulated at the same time.

I will only be showing the charts of the average FPS, because almost all of them match up with the charts for the median FPS. However, if a case occurs in which this is not true, I will be showing both and commenting on why they're different.

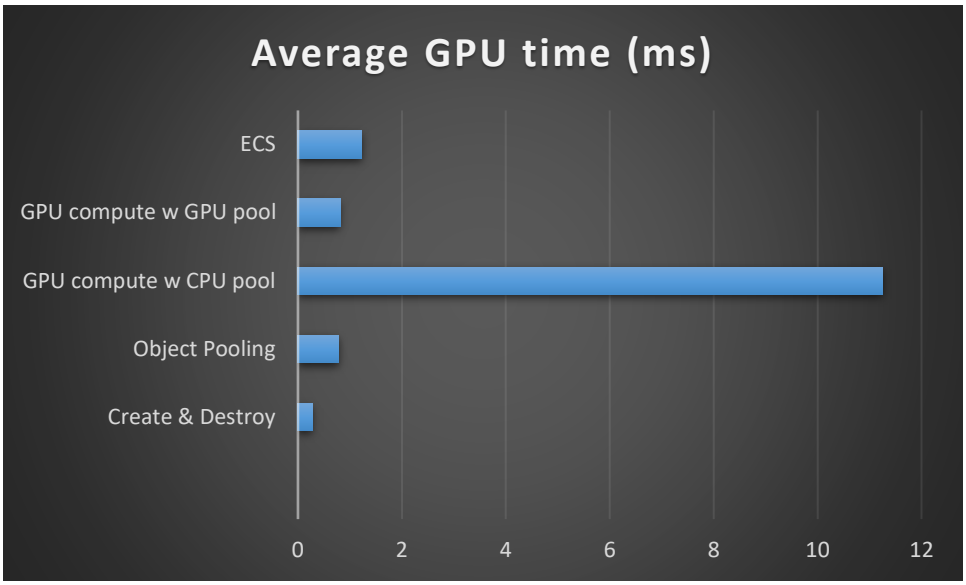
The tests are run on a laptop with the following hardware:

- GPU: NVIDIA GeForce RTX 4060 Laptop GPU
- RAM: 16GB DDR4
- CPU: 12th Gen Intel(R) Core(TM) i7-12700H

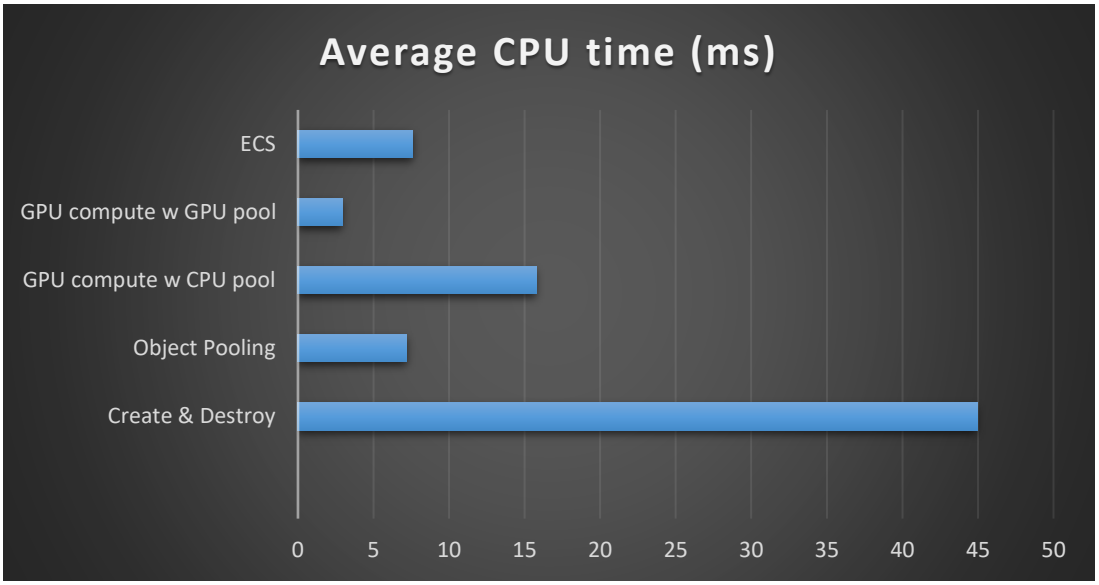
2.2 HIGH NUMBER OF FAST BULLETS TEST



PICTURE 20 (High number of fast bullets average FPS)

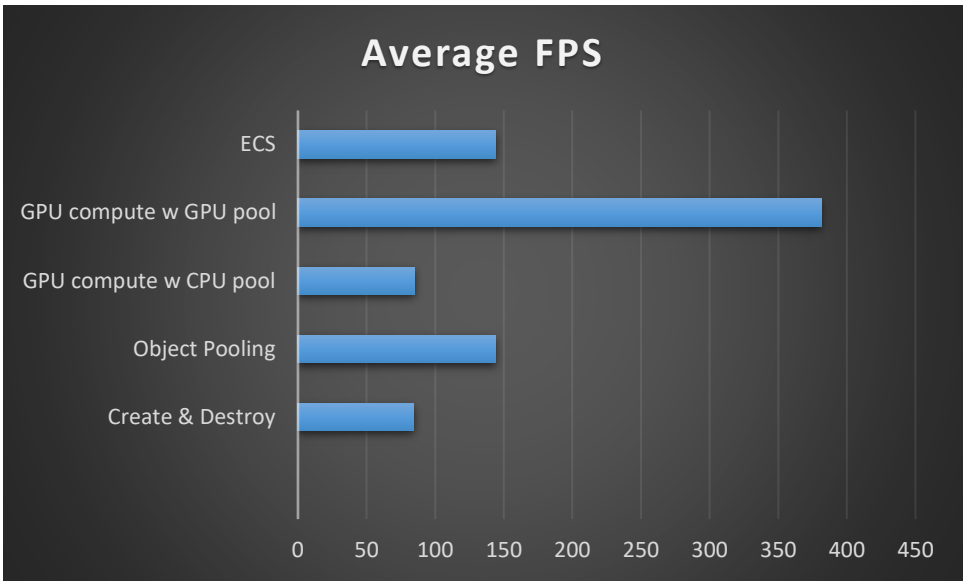


PICTURE 21 (High number of fast bullets average GPU time)

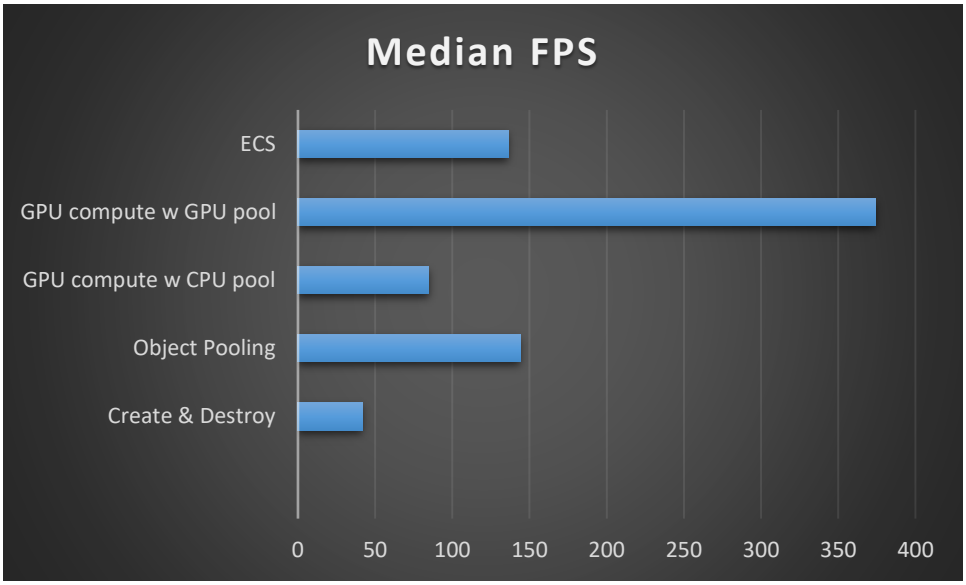


PICTURE 22 (High number of fast bullets average CPU time)

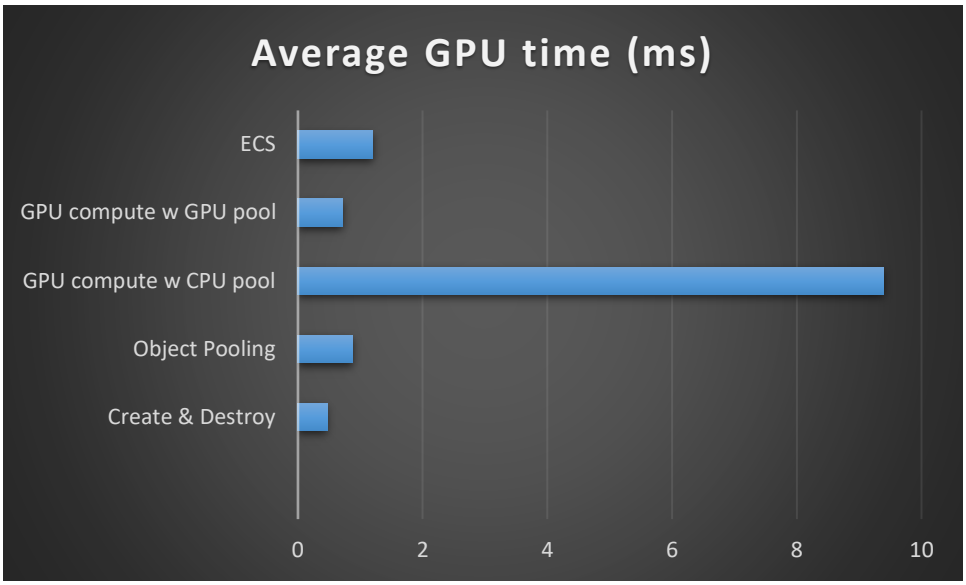
2.3 MEDIUM NUMBER OF SLOW BULLETS TEST



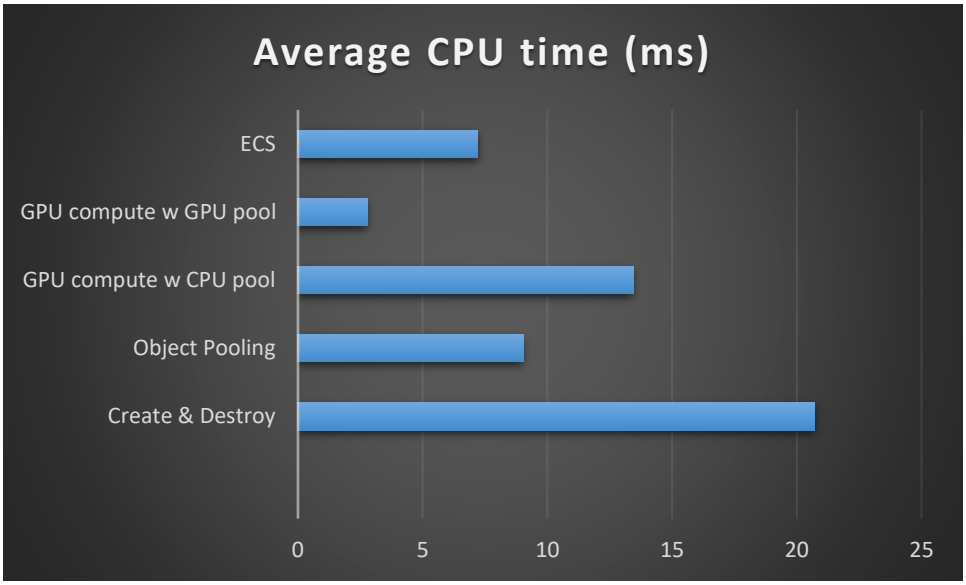
PICTURE 23 (Medium number of slow bullets average FPS)



PICTURE 24 (Medium number of slow bullets median FPS)



PICTURE 25 (Medium number of slow bullets average GPU time)



PICTURE 26 (Medium number of slow bullets average CPU time)

DISCUSSION

The results shown in PICTURE 20-26 give us very similar results for both tests, as such, I will be talking about them combined, except for cases where any noticeable difference occurred.

When looking at PICTURE 20 & 23, at first sight we can immediately notice that the GPU shader with pooling on the GPU is the most performant by a long shot. ECS and object pooling are very similar performance-wise. However, the extent to which the advantages of ECS were used is quite minimal. Namely, ECS did not use the Unity job system, nor did it use pooling. If either of these two were implemented, its performance would most likely be largely better than object pooling with regular game objects.

Another interesting observation that can be made from PICTURE 21 & 25 is the fact that the GPU compute shader with CPU side pooling is the handler that takes up the most GPU time by a long shot. This is because of its constant need to synchronize data between CPU and GPU, which takes a lot of time.

We can however notice one difference between a graph showing average and a graph showing median values. Namely PICTURE 23 & 24 that show FPS values for the second test, we can see that the average is about 40 FPS more than the median. This shows that the second test runs in a very unstable manner, there were a couple of very fast, light frames that increased the average by a lot. However, in general, the player feels the median FPS, which is much lower, due to most of the frames being way slower than the average might make out to be.

An important advantage that all the optimization techniques have, is that alongside FPS, they also help a lot with increasing the stability at which your game runs, increasing the player experience. This can be confirmed by the fact that their average and median don't differ by a large amount, meaning that the average FPS shown in the graphs is an accurate representation of how the game would run.

CONCLUSION

1. SUMMARY OF RESEARCH

The main goal of this research was to implement multiple optimization techniques, and see if some of them may be combined with one another in some way. Next, we ran two test simulations with each of the optimization techniques from which we then got runtime performance data such as FPS and CPU/GPU time.

We then combined this data into graphs which we used to analyze and compare the different techniques. These analyzations lead to the conclusion that we will be going over below.

2. ANSWER TO RESEARCH QUESTION

Each of the optimization techniques improved performance to different extents. In a lot of cases, using just an object pool will probably be enough if the behavior per object is as simple as it is in the case of this research. However, when having more complexity, you will want to use a GPU-compute shader or ECS to ensure performance stays up to expectations. We also saw that combining optimization techniques such as the GPU-compute shader and object pooling yielded impressive performance improvements compared to the other techniques.

3. HYPOTHESIS 0

Combining different techniques doesn't increase performance more compared to using them on their own individually, due to the overhead required to use them together in a correct manner.

This hypothesis is wrong, the handler that was a combination of both a GPU-compute shader and object pooling ended up actually being the best-performing handler by a long shot in both test scenarios. Additionally, using this handler was also very easy, and as such, this hypothesis is very much disproved.

4. HYPOTHESIS 1

Combining techniques (pooling + ECS, pooling + GPU compute shader) increases runtime performance compared to single techniques when projectile counts are high, but may be neutral or slightly worse at low bullet counts due to overhead.

While I did not manage to combine ECS and pooling in time, I will evaluate this hypothesis for the object pooling and GPU-compute shader combination. In this case the hypothesis would be wrong, the performance of this combination handler is superior both in cases with a lot of bullets and ones with less bullets. This was proved with the two different experiments, where the bullet counts heavily changed. Despite it heavily changing, the handler's performance remains vastly better than any single technique by itself.

5. HYPOTHESIS 2

Implementing combinations, especially pooling + ECS, increases development effort compared to single techniques. Pooling + ECS might not even be possible due to the vastly different nature and structure of ECS projects.

Implementing combinations does certainly increase development effort, especially in the case of combining the GPU-compute shader and the pooling. Though the increased development is most definitely worth the performance gains that it provides along with combining these techniques in this case. While I did not manage to implement a combination of ECS and object pooling, from the knowledge and insights into ECS that I gained from working on this paper, I believe combining them is definitely possible.

6. HYPOTHESIS 3

The benefit of GPU compute shaders increases with per-bullet computational complexity (for example: homing logic, collision checks) while pooling gives constant benefits regardless of per-bullet compute. ECS gives largest benefits when many components are present or when cache locality matters.

There will be no concrete answer to the first part of the hypothesis because my experiment did not end up including bullets with more complex behavior, though I do speculate it to be true. The other two parts of the hypothesis are both correct. Object pooling gives a benefit to spawning and de-spawning objects, but does not bring any changes when it comes to how the bullets are updated. On the other hand, ECS provides an improvement to how bullets are updated, and has excellent cache-usage. However, combining ECS with object pooling might be able to combine both advantages.

FUTURE WORK

The field of optimization has always been a big topic in game development, and as such, there are lots of things that may still be explored that are related to my research.

1. OPTIMIZATION FOR ECS

While ECS was implemented for this research, it was in no way used to its full potential, a good starting ground for further research would be to implement the job system and pooling. This could then also be followed by seeing if there might be a way to combine the advantages of ECS with the advantages of using a GPU-compute shader.

2. FINDING WAYS TO WORK AROUND GPU COMPUTE SHADER LIMITS

In the research section, some severe limitations were displayed that come along with using a GPU-compute shader. Mainly the facts that we lose access to using premade components. It would be very useful to look into if there is any way to work around this that doesn't fully negate the impressive performance gains that come with using a GPU-compute shader.

3. LOOK FURTHER INTO MEMORY LAYOUT OPTIMIZATIONS

ECS already has a very particular way of laying out its memory so that the cache gets used in a very efficient manner. Through research into memory layout and seeing if there might be an even more effective way to go about it may yield some interesting results.

CRITICAL REFLECTION

I started on this research with the goal of learning more about optimization techniques for when many objects need to be simulated and seeing how these techniques may be combined. I feel as if I have certainly reached this goal, the research and experimenting that were involved in the writing of this paper allowed me to familiarize myself with GPU compute shaders and ECS.

I was already familiar with the workings of an object pool and some very basics of a GPU compute shader, but the process of writing this paper improved my knowledge in both significantly because of conducting tests and analyzing their results.

Throughout the entire process, I have also come to the conclusion that research is not a personal passion of mine and that I would much rather create/make things myself. Despite this, I do find that it was a very valuable experience due to all the knowledge and skills I gained from it.

REFERENCES

GeeksforGeeks. (2024). Object Pool Design Pattern.
https://www.geeksforgeeks.org/java/object-pool-design-pattern

GeeksforGeeks. (2025). *Component-Based Architecture - System Design.*
https://www.geeksforgeeks.org/system-design/component-based-architecture-system-design

Housemarque. (2021). Returnal [Video game]. Sony Interactive Entertainment.
https://housemarque.com/games/returnal

Onufriienko D. M. (2024). Using a compute shader for an adaptive particle system. Mathematical Modeling and Computing. Vol. 11
https://science.lpnu.ua/mmc/all-volumes-and-issues/volume-11-number-1-2024/using-compute-shader-adaptive-particle-system

React. (2025). *React ECS Documentation.*
https://react-ecs.ldlework.com/docs

Robert Nystrom. (2021). *Game Programming Patterns.*
https://gameprogrammingpatterns.com/object-pool.html

Saari, M. (2025). Optimization for bullet hell games.
https://www.theseus.fi/bitstream/handle/10024/894844/Saari_Mikko.pdf

Unity Technologies. (2025*). Introduction to Object Pooling.*
https://learn.unity.com/tutorial/introduction-to-object-pooling

Unity Technologies. (2025). *Garbage collection modes.*
https://docs.unity3d.com/6000.3/Documentation/Manual/performance-incremental-garbage-collection.html

Veyeral Games (2018). *The Void Rains Upon Her Heart (Early Access) [Video game].* The Hidden Levels. https://store.steampowered.com/app/790060/The_Void_Rains_Upon_Her_Heart

Wikipedia. (2025). *Object pool pattern.* https://en.wikipedia.org/w/index.php?title=Object_pool_pattern

Wikipedia. (2025). *General-purpose computing on graphics processing units.* https://en.wikipedia.org/wiki/General-purpose_computing_on_graphics_processing_units

Wikipedia. (2025). *Single instruction, multiple threads.* https://en.wikipedia.org/wiki/Single_instruction,_multiple_threads

Wikipedia. (2025). *Entity component system.* https://en.wikipedia.org/wiki/Entity_component_system

Yeray Tarifa Mateo. (2024). *Entity Component Systems And Data-Oriented Design.* https://upcommons.upc.edu/server/api/core/bitstreams/5687c958-e301-4f69-a96b-f56f250074ac/content

ACKNOWLEDGEMENTS

Firstly, I would like to thank my supervisor, Alex Vanden Abeele, for his valuable feedback and guidance along the entire way of this paper. I also want to express my gratitude to my coach, Marijn Verspecht, for her tips & tricks surrounding time management and for keeping up with the progress.

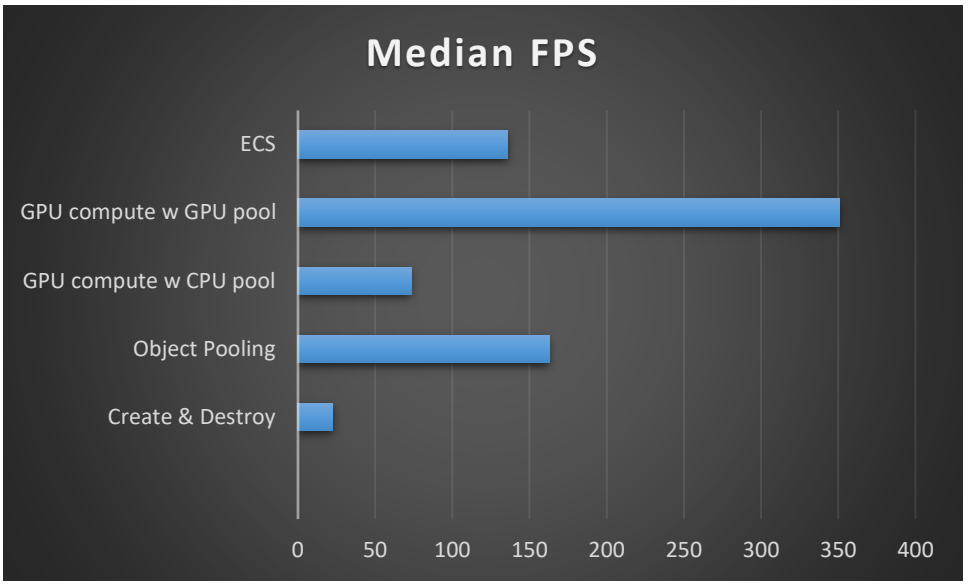
Besides that, I thank my parents, partner, and friends, for all their motivation, and endless support along the way.

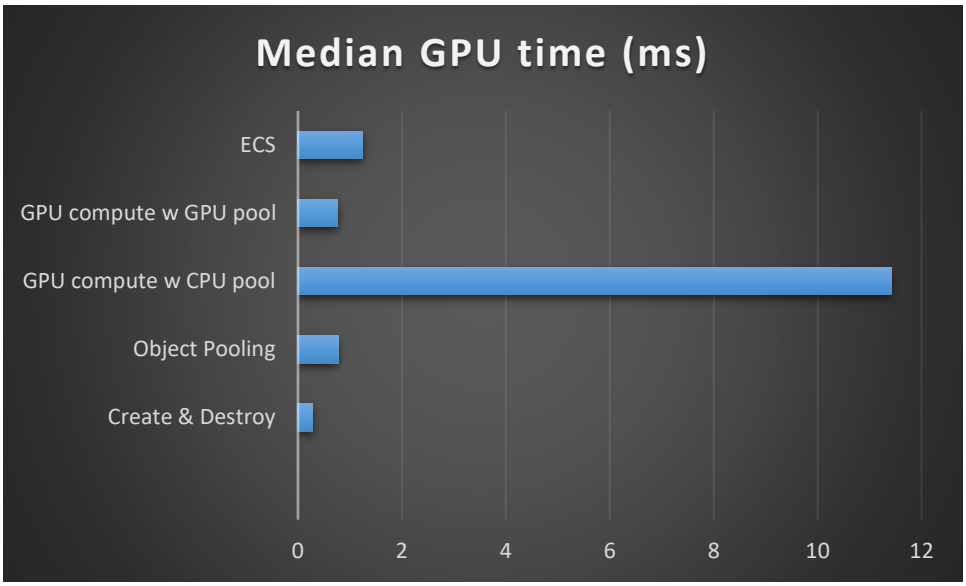
APPENDICES

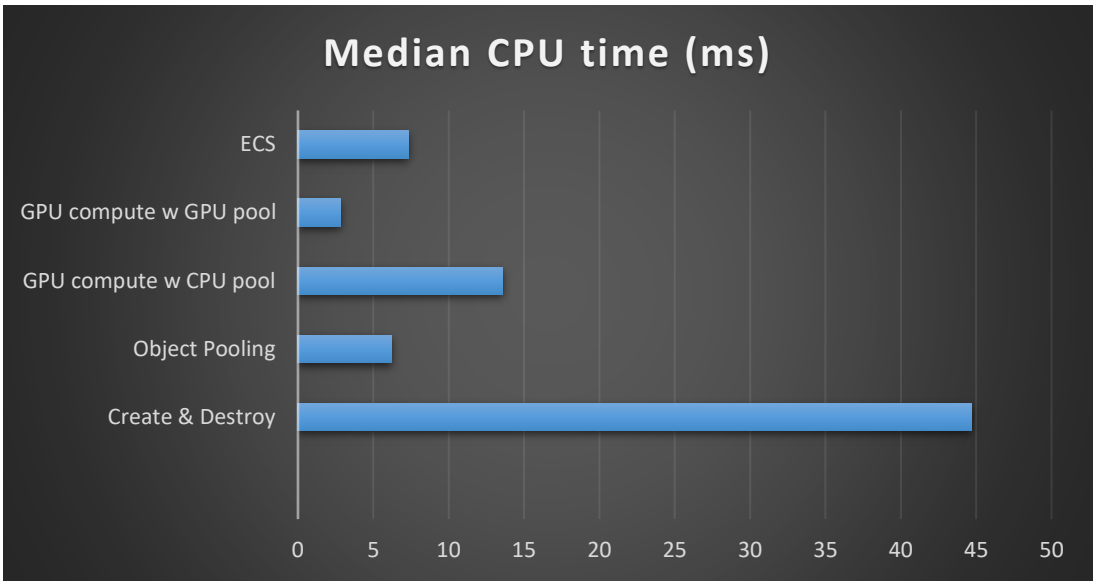
1. EXPERIMENT RESULT METRICS

1.1 HIGH NUMBER OF FAST BULLETS TEST RESULTS

Simulation Type	Average FPS	Median FPS	Average CPU time (ms)	Median CPU time (ms)	Average GPU time (ms)	Median GPU time (ms)
Create & Destroy	22.7356548	22.351862	44.94814682	44.70019913	0.274992585	0.281599998
Object Pooling	157.604416	162.645294	7.175396919	6.230099678	0.783941448	0.774143994
GPU compute w CPU pool	71.2555084	73.338501	15.76782703	13.59860039	11.25101757	11.41350365
GPU compute w GPU pool	350.795471	350.803925	2.936949968	2.846400023	0.809267998	0.766975999
ECS	134.864273	136.010101	7.606441498	7.342400074	1.229050636	1.24723196

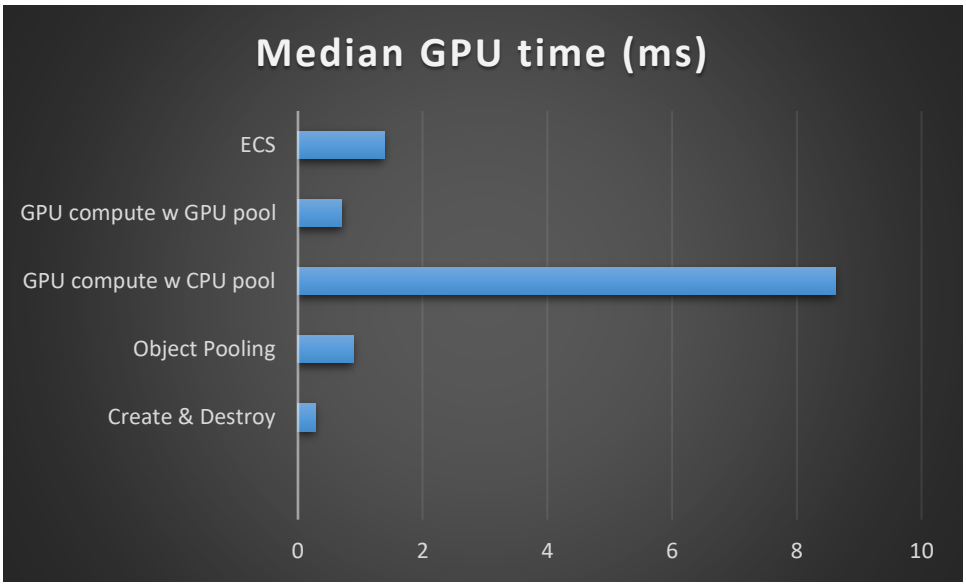

Median FPS
ECS
GPU compute w GPU pool
GPU compute w CPU pool
Object Pooling
Create & Destroy
0
50
100
150
200
250
300
350
400

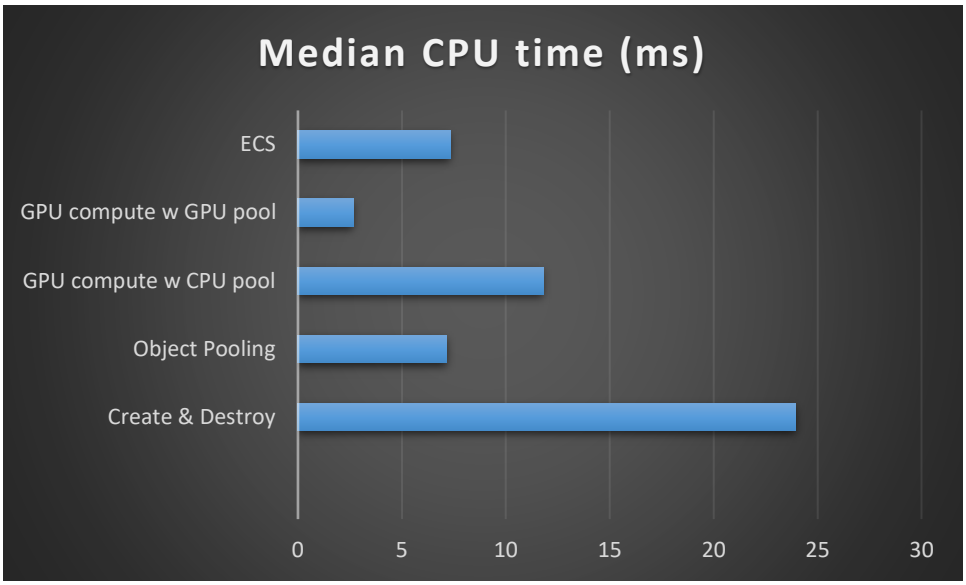

Median GPU time (ms)
ECS
GPU compute w GPU pool
GPU compute w CPU pool
Object Pooling
Create & Destroy
0
2
4
6
8
10
12



1.2 MEDIUM NUMBER OF SLOWER BULLETS TEST RESULTS

Simulation Type	Average FPS	Median FPS	Average CPU time (ms)	Median CPU time (ms)	Average GPU time (ms)	Median GPU time (ms)
Create & Destroy	84.26164246	41.73779678	20.71590614	23.9538002	0.471382558	0.280575991
Object Pooling	144.0656433	144.1992188	9.033903122	7.15899992	0.878290415	0.888831973
GPU compute w CPU pool	85.06076813	84.87093353	13.44734573	11.79100037	9.396018982	8.617983818
GPU compute w GPU pool	381.4266968	374.4338379	2.781110525	2.660000086	0.71850431	0.705536008
ECS	144.276886	136.3391418	7.205388546	7.334249973	1.199340343	1.381376028





2. BULLET-HELL SIMULATION ENVIRONMENT

```
[System.Serializable]
11 references | 0 changes | 0 authors, 0 changes
public class BulletData
{
    public Vector3 SpawnPosition;
    public Vector2 SpawnDirection;
    public float MovementSpeed;
}

Unity Script (1 asset reference) | 3 references | 0 changes | 0 authors, 0 changes
public class BulletHellSimulation : MonoBehaviour
{
    [SerializeField] private List<SimulationData> _simulationDataList;
    [SerializeField] private List<BaseHandler> _handlerList;
    [SerializeField] private PerformanceLogger _performanceLogger;
    [SerializeField] private Transform _bulletParentTransform;
    [SerializeField] private TextMeshProUGUI _progressText;

    private SimulationData _currentSimulationData;
    public BaseHandler _currentBulletHandler;
    private int _currentSimulationDataIndex = 0;
    private int _currentHandlerIndex = -1;

    private List<float> _shooterAngles = new List<float>();
    private Coroutine _shooterRotationCoroutine = null;
    private bool _hasDoneWarmup = false;

    private bool _isWaitingForHandlerAction = false;
    10 references | 0 changes | 0 authors, 0 changes
    public bool IsWaitingForHandlerAction
    {
        get { return _isWaitingForHandlerAction; }
        set { _isWaitingForHandlerAction = value; }
    }

    1 reference | 0 changes | 0 authors, 0 changes
    public BaseHandler CurrentBulletHandler
    {
        get { return _currentBulletHandler; }
    }
```

```
2 references | 0 changes | 0 authors, 0 changes
public SimulationData SimulationData
{
    get { return _currentSimulationData; }
}

public UnityEvent OnWarmupSequenceStart = new UnityEvent();
public UnityEvent OnWarmupSequenceEnd = new UnityEvent();

public UnityEvent OnStartInitializingHandler = new UnityEvent();
public UnityEvent OnFinishInitializingHandler = new UnityEvent();

public UnityEvent OnStartResettingHandler = new UnityEvent();
public UnityEvent OnFinishResettingHandler = new UnityEvent();

public UnityEvent OnStabilizationStarted = new UnityEvent();
public UnityEvent OnStabilizationFinished = new UnityEvent();

public UnityEvent OnAllSimulationsFinished = new UnityEvent();

Unity Message | 0 references | 0 changes | 0 authors, 0 changes
private void Start()
{
    _performanceLogger.OnFinishedWritingData.AddListener(OnLogWritingFinished);
}

0 references | 0 changes | 0 authors, 0 changes
public void StartSimulation()
{
    OnLogWritingFinished();
}
```

```
1 reference | 0 changes | 0 authors, 0 changes
private IEnumerator SimulateBulletHell()
{
    // Initialize all shooter angles
    _shooterAngles.Clear();
    float angleStep = 360f / _currentSimulationData.NumberOfShooters;
    for (int shooterIdx = 0; shooterIdx < _currentSimulationData.NumberOfShooters; shooterIdx++)
    {
        _shooterAngles.Add(angleStep * shooterIdx);
    }

    float totalTimeInSimulation = 0.0f;
    bool isTimeBasedSimulation = _currentSimulationData.AmountOfSequencesBeforeEnd <= 0;
    int totalSequencesCompleted = -1; // Starting at -1 to account for warmup sequence
    int roundsCompletedInCurrentSequence = 0;
    float timeInCurrentSequence = 0.0f;
    bool sequenceIsTimeBased = _currentSimulationData.AmountOfRoundsBeforeSequenceTrigger <= 0;

    _shooterRotationCoroutine = StartCoroutine(ShooterRotation());

    // Initialize sequence
    _isWaitingForHandlerAction = true;
    OnStartInitializingHandler.Invoke();
    if(_hasDoneWarmup) _performanceLogger.OnStartLoading();
    _currentBulletHandler.OnSequenceInitializeRequested(_bulletParentTransform);
    yield return new WaitUntil(() => !_isWaitingForHandlerAction);
    if (_hasDoneWarmup) _performanceLogger.OnLoadingFinished();
    OnFinishInitializingHandler.Invoke();

    OnStabilizationStarted.Invoke();
    yield return new WaitForSecondsRealtime(_currentSimulationData.StabilizationWaitTime);
    OnStabilizationFinished.Invoke();

    while (true)
```

```
while (true)
{
    _currentBulletHandler.OnSequenceStarting();
    if (_hasDoneWarmup) _performanceLogger.OnSequenceStart();
    if (!_hasDoneWarmup) OnWarmupSequenceStart.Invoke();
    for (int shooterIdx = 0; shooterIdx < _currentSimulationData.NumberOfShooters; shooterIdx++)
    {
        // Calculate direction
        float angleInRadians = _shooterAngles[shooterIdx] * Mathf.Deg2Rad;
        Vector2 shootDirection = new Vector2(Mathf.Cos(angleInRadians), Mathf.Sin(angleInRadians)).normalized;

        // Request bullet spawn (handled externally)
        BulletData bulletData = new BulletData
        {
            SpawnPosition = transform.position,
            SpawnDirection = shootDirection,
            MovementSpeed = _currentSimulationData.BulletMoveSpeed
        };
        _currentBulletHandler.OnBulletSpawnRequested(bulletData, _bulletParentTransform);

        // Wait between shooters
        if (_currentSimulationData.DelayBetweenShooters > 0)
        {
            if (_currentSimulationData.ContinueRotationDuringShooterDelay) StopCoroutine(_shooterRotationCoroutine);
            yield return new WaitForSecondsRealtime(_currentSimulationData.DelayBetweenShooters);
            if (_currentSimulationData.ContinueRotationDuringShooterDelay) _shooterRotationCoroutine = StartCoroutine(ShooterRotation());
        }
    }
    ++roundsCompletedInCurrentSequence;
    timeInCurrentSequence += _currentSimulationData.DelayBetweenShooters * _currentSimulationData.NumberOfShooters;
    timeInCurrentSequence += _currentSimulationData.DelayAfterFullShooterRound;

    // Check if sequence is completed based on what type it is
    bool sequenceTimeCompleted = sequenceIsTimeBased && (timeInCurrentSequence >= _currentSimulationData.TimeBeforeSequenceTrigger);
    bool sequenceRoundCompleted = !sequenceIsTimeBased && (roundsCompletedInCurrentSequence >= _currentSimulationData.AmountOfRoundsBeforeSequenceTrigger);
    bool sequenceJustCompleted = sequenceTimeCompleted || sequenceRoundCompleted;
    if (sequenceJustCompleted)
```

```
if (sequenceJustCompleted)
{
    if (_hasDoneWarmup) _performanceLogger.OnSequenceFinished();
    _currentBulletHandler.OnSequenceEnding();

    roundsCompletedInCurrentSequence = 0;
    timeInCurrentSequence = 0.0f;
    ++totalSequencesCompleted;

    // Reset and initialize sequence
    _isWaitingForHandlerAction = true;
    OnStartResettingHandler.Invoke();
    _currentBulletHandler.OnSequenceResetRequested(_bulletParentTransform.gameObject);
    yield return new WaitUntil(() => !_isWaitingForHandlerAction);
    OnFinishResettingHandler.Invoke();
}

// Check if simulation should end
if (isTimeBasedSimulation)
{
    totalTimeInSimulation += _currentSimulationData.DelayBetweenShooters * _currentSimulationData.NumberOfShooters;
    totalTimeInSimulation += _currentSimulationData.DelayAfterFullShooterRound;
    if (totalTimeInSimulation >= _currentSimulationData.TimeBeforeEnd) break;
}
else if(!isTimeBasedSimulation)
{
    if (totalSequencesCompleted >= _currentSimulationData.AmountOfSequencesBeforeEnd) break;
}

// Check if sequence is complete
if (sequenceJustCompleted)
```

```
        // Check if sequence is complete
        if (sequenceJustCompleted)
        {
            if (!_hasDoneWarmup) OnWarmupSequenceEnd.Invoke();

            _isWaitingForHandlerAction = true;
            OnStartInitializingHandler.Invoke();
            if (_hasDoneWarmup) _performanceLogger.OnStartLoading();
            _currentBulletHandler.OnSequenceInitializeRequested(_bulletParentTransform);
            yield return new WaitUntil(() => !_isWaitingForHandlerAction);
            if (_hasDoneWarmup) _performanceLogger.OnLoadingFinished();
            OnFinishInitializingHandler.Invoke();

            if (!_hasDoneWarmup) _hasDoneWarmup = true;

            OnStabilizationStarted.Invoke();
            yield return new WaitForSecondsRealtime(_currentSimulationData.StabilizationWaitTime);
            OnStabilizationFinished.Invoke();
        }
        else
        {
            // Wait between full shooter rounds
            if (_currentSimulationData.DelayAfterFullShooterRound > 0)
            {
                if (_currentSimulationData.ContinueRotationDuringFullShooterRoundDelay) StopCoroutine(_shooterRotationCoroutine);
                yield return new WaitForSecondsRealtime(_currentSimulationData.DelayAfterFullShooterRound);
                if (_currentSimulationData.ContinueRotationDuringFullShooterRoundDelay) _shooterRotationCoroutine = StartCoroutine(ShooterRotation());
            }
        }
    }
    if(_shooterRotationCoroutine != null)
    {
        StopCoroutine(_shooterRotationCoroutine);
        _shooterRotationCoroutine = null;
    }
    _performanceLogger.OnSimulationFinished();
}
```

```
2 references | 0 changes | 0 authors, 0 changes
private void OnLogWritingFinished()
{
    // Loop over every handler for each simulation data
    ++_currentHandlerIndex;
    _hasDoneWarmup = false;
    if (_currentHandlerIndex >= _handlerList.Count)
    {
        _currentHandlerIndex = 0;
        ++_currentSimulationDataIndex;
    }
    if (_currentSimulationDataIndex < _simulationDataList.Count)
    {
        _currentSimulationData = _simulationDataList[_currentSimulationDataIndex];
        _currentBulletHandler = _handlerList[_currentHandlerIndex];
        StartCoroutine(SimulateBulletHell());
        // Check progress
        _progressText.text = $"{_currentSimulationDataIndex * _handlerList.Count + _currentHandlerIndex + 1} / " + $"{_simulationDataList.Count * _handlerList.Count}";
    }
    else
    {
        _progressText.text = "-/-";
        _currentSimulationDataIndex = 0;
        _currentHandlerIndex = -1;
        OnAllSimulationsFinished.Invoke();
    }
}

3 references | 0 changes | 0 authors, 0 changes
private IEnumerator ShooterRotation()
{
    while(true)
    {
        for (int shooterIdx = 0; shooterIdx < _currentSimulationData.NumberOfShooters; shooterIdx++)
        {
            // Rotate shooters
            _shooterAngles[shooterIdx] += _currentSimulationData.ShooterRotationSpeed * Time.deltaTime;
            if (_shooterAngles[shooterIdx] >= 360f) _shooterAngles[shooterIdx] -= 360f;
        }
        yield return null;
    }
}
```