POLITECNICO DI BARI

DEPARTMENT OF ELECTRICAL AND INFORMATION ENGINEERING
Bachelor's Degree in Computer Engineering and Automation

Dissertation in Software Engineering

Data-Oriented Approach in Game Engines: Benchmarking and Experimentation with Unity DOTS for Creating Resource-Intensive Scenes

Supervisor
Prof.ssa Marina Mongiello

Candidate
Domenico Rotolo

Academic Year 2023 - 2024

Abstract

In modern high-performance graphic applications, such as simulations and video games, screen refresh rates and latency management play a crucial role. Each frame must be generated and displayed within extremely short timeframes, without delays or interruptions, to ensure a smooth user experience. In this context, software paradigm and architecture choices can make a significant difference, directly influencing memory management and code efficiency.

This thesis aims to explore the data-oriented design (DOD) programming paradigm as a significant alternative to the traditional object-oriented programming (OOP) approach. An analysis of the fundamental characteristics of the two paradigms is conducted, highlighting the performance costs associated with OOP, especially in contexts where optimized memory access is a priority. Conversely, DOD focuses on the structure and sequential access of data, enabling more efficient cache usage and faster, more predictable memory access.

The core of this thesis is dedicated to analyzing Unity DOTS (Data-Oriented Technology Stack), a suite of libraries developed by Unity Technologies to implement the DOD paradigm within the Unity game engine. Unity DOTS includes three main components: the C# Job System, the Burst Compiler, and the Entity Component System (ECS). Each component is analyzed in detail to illustrate its contribution to overall performance: the C# Job System facilitates the management and parallel execution of tasks, reducing the overall computational load. The Burst Compiler, with its high-performance compiler, optimizes low-level operations to fully leverage the underlying hardware. Lastly, ECS enables the writing of entities using the philosophy of the DOD paradigm.

Subsequently, the thesis describes the design and implementation of an application system based on Unity DOTS, aimed at comparing the OOP paradigm, implemented with MonoBehaviour classes, with DOD in terms of performance. The application system was designed to include various benchmarking scenarios, each representing a typical situation in video games and simulations, such as managing numerous moving objects or simultaneously calculating complex game states and collisions.

Finally, an analysis of the benchmarking results is conducted, demonstrating

how the use of DOD and Unity DOTS can lead to significant performance improvements.

Thesis Structure

- **Comparison between OOP and DOD**: detailed description of memory fragmentation costs and data access issues in OOP, along with an analysis of DOD and its performance advantages.
- **Unity DOTS**: technical deep dive into the main components of DOTS (C# Job System, Burst Compiler, ECS), detailing architecture, functionality, supported optimizations, and development roadmap.
- **Implementation of the application system**: description of the developed application system for OOP-DOD comparison, experimental methodology, and specifications of benchmark scenarios.
- **Results and analysis**: presentation of the obtained results, analysis of performance differences, and implications for high-performance software development.
- **Conclusions**: summary of the results and final considerations on the use of DOD and Unity DOTS for performance improvement, with proposals for future research and experimentation areas.

Contents

Chapter 1

Comparison Between Object-Oriented Programming and Data-Oriented Programming

1.1 Object-Oriented Programming

Object-oriented programming (OOP) has historically been the most influential programming paradigm. It is widely used in both education and industry, and almost every university includes object-oriented programming in its curriculum [1].

The strategy to solve problems with OOP is to divide them into smaller problems. The paradigm can therefore be subdivided into three sub-paradigms [2]:

1.1.1 OOP Sub-Paradigms

Program Structure Paradigm

Classes and objects represent the fundamental building blocks of a program. Classes define the blueprint for objects, encapsulating data represented by attributes and behaviors represented by methods. Inheritance, polymorphism, and encapsulation are key principles in this context.

State Structure Paradigm

Objects have internal states, represented by their attributes, which can change when the object interacts with other objects or processes. In this context, encapsulation is particularly important as it allows objects to hide their internal states, exposing only what is necessary through methods. The idea that the state can influence an object's behavior is central to this sub-paradigm.

Computation Paradigm

This sub-paradigm focuses on message passing between objects, a process by which objects communicate with each other to perform tasks and computations. Methods are the means by which computation occurs within objects; interaction between objects, achieved through method calls, drives program execution. In this context, polymorphism and dynamic method binding are essential as they allow different objects to respond to the same message in contextually appropriate ways.

1.2 Performance Costs of OOP

Among the theoretical benefits of OOP are primarily reconfigurability and code reuse: managing objects facilitates the removal and modification of program functionalities and the reuse of objects to allow for greater consistency and error reduction. Additionally, object-oriented programming encourages class architecture that reflects the real world, making the code more comprehensible and natural to manage. Another advantage is abstraction, which allows programmers to address problems at a higher level without being distracted by low-level implementation details.

That said, the very nature of OOP presents performance costs that, as we will see in the next chapter, are particularly evident in the development of highly reactive programs such as video games. Firstly, the preference for small functions and numerous levels of abstraction complicates the execution flow, making it difficult to follow and optimize the call paths. Furthermore, because the code manipulating an object is inherently part of the object itself, there is a natural tendency to treat objects individually rather than processing them in larger blocks, thus reducing optimization opportunities such as parallelization. Another impact on performance stems from the fragmented memory layout: OOP code is often divided into many small objects, which causes data to be scattered across various memory locations; thus, allocation patterns in OOP tend to be inefficient, and the code often relies on frequent small memory allocations and garbage collection (this will be described in detail in the section 'Memory Access Optimization') [3].

1.3 Data-Oriented Design

Data-oriented design (DOD) is a modern paradigm that focuses on how data is stored in memory rather than on the relationships between the data itself, making it more easily transformable. The goal is to facilitate coding with a more efficient memory layout by storing frequently used data closer together in memory. Unlike OOP, DOD also seeks to separate data from logic [4].

1.3.1 Composition Over Inheritance

DOD favors composition over inheritance. This is fundamental as composition makes it easier to separate data from logic.

Inheritance Inheritance is a mechanism where a class (called a subclass or child class) inherits properties and behaviors (methods) from another class (called a superclass or parent class). This can lead to tightly coupled code, where subclasses are heavily dependent on superclasses, making future modifications difficult and complicating the structure if the hierarchy becomes too complex.

Composition Composition, on the other hand, involves creating objects that are composed of other objects rather than relying on inheritance. This approach allows greater flexibility as objects can be combined in different ways without being constrained by a rigid inheritance structure.

Codice 1.1: Example of Inheritance

```
class Fish
{
    public void Swim()
    {
        Console.WriteLine("Swimming");
    }
}

class Shark : Fish
{
    // Shark inherits the Swim method from the Fish class
}
```

Codice 1.2: Example of Composition

```
class SwimBehaviour
{
    public void Swim()
    {
        Console.WriteLine("Swimming");
    }
}

class Shark
{
    private SwimBehaviour swimBehaviour;

    public Shark()
    {
        swimBehaviour = new SwimBehaviour();
    }

    public void DoSwim()
    {
        swimBehaviour.Swim();
    }
}
```

1.3.2 Designing Data Before Designing Code

The central premise of Data-Oriented Design (DOD) is that data is at least as important as code. Both at the macro and micro levels, programs are essentially about transforming and producing data. Code is seen as an assembly line for data: in a video game, for instance, the code, at each tick, receives user input and the current game state, transforming them into a new state for rendering the next frame.

This definition of programming may seem overly simplified or obvious, but it provides clarity regarding DOD objectives. Having well-defined starting and ending points for data allows building algorithms that independently manipulate various data states, making it easier for programmers to optimize individual transformations and identify bottlenecks. [3]

Optimization Opportunities The following optimization opportunities, for example, can be more easily discovered with an assembly-line model (pipeline model):

- The need to transform raw data into intermediate steps to enable more efficient processing in subsequent steps.
- Conversely, the need to reduce the number of data processing steps.
- Data produced by multiple different steps can be saved in a single preceding step.
- Elements processed one at a time can instead be processed in bulk to ensure more efficient memory access.

For the purposes of this work, memory access is one of the most influential advantages of DOD. As previously anticipated, we now analyze it in more detail.

1.4 Memory Access Optimization

Reading data from memory is performed through cache access: when the CPU executes an instruction that requires reading a system memory address, the hardware checks if a copy of the data at that address is present in the first-level cache. If not, the hardware checks the next-level cache until it directly reads from system memory if the data is not present in any cache (this is known as a 'cache miss'). Most modern CPUs have three levels of cache, with sizes and access times varying across levels.

1.4.1 Prefetching

The performance of a program depends on its ability to minimize 'cache misses' to ensure faster access to the data required for processing. Consider the following algorithms:

Codice 1.3: Matrix Summation Example

```
public class MatrixSum
{
    public static int SumMatrixRows(int[,] matrix)
    {
        int rows = matrix.GetLength(0);
        int cols = matrix.GetLength(1);
        int sum = 0;

        for (int i = 0; i < rows; i++)
        {
            for (int j = 0; j < cols; j++)
            {
                sum += matrix[i, j];
            }
        }

        return sum;
    }

    public static int SumMatrixColumns(int[,] matrix)
    {
        int rows = matrix.GetLength(0);
        int cols = matrix.GetLength(1);
        int sum = 0;

        for (int j = 0; j < cols; j++)
        {
            for (int i = 0; i < rows; i++)
            {
                sum += matrix[i, j];
            }
        }

        return sum;
    }
}
```

Both functions return the sum of all elements in a matrix: the first iterates over the rows, while the second iterates over the columns. Using the following code, we can compare the execution times of the two algorithms for a matrix of any size:

Codice 1.4: Matrix Summation Main

```
class Program
{
    static void Main()
    {
        for (int i = 1; i <= 50; i++)
        {
            Test(i * 100);
        }
    }

    static void Test(int size)
    {
        int[,] matrix = new int[size, size];

        for (int i = 0; i < size; i++)
        {
            for (int j = 0; j < size; j++)
            {
                matrix[i, j] = 1;
            }
        }

        Stopwatch stopwatch = new Stopwatch();

        stopwatch.Start();
        int sumRows = MatrixSum.SumMatrixRows(matrix);
        stopwatch.Stop();
        Console.WriteLine($"Rows: {size}, {stopwatch.ElapsedMilliseconds} ms");

        stopwatch.Restart();
        int sumColumns = MatrixSum.SumMatrixColumns(matrix);
        stopwatch.Stop();
        Console.WriteLine($"Columns: {size}, {stopwatch.ElapsedMilliseconds}
        ↪  ms");
    }
}
```

We can visualize the execution time in milliseconds as the matrix size varies in the following chart:

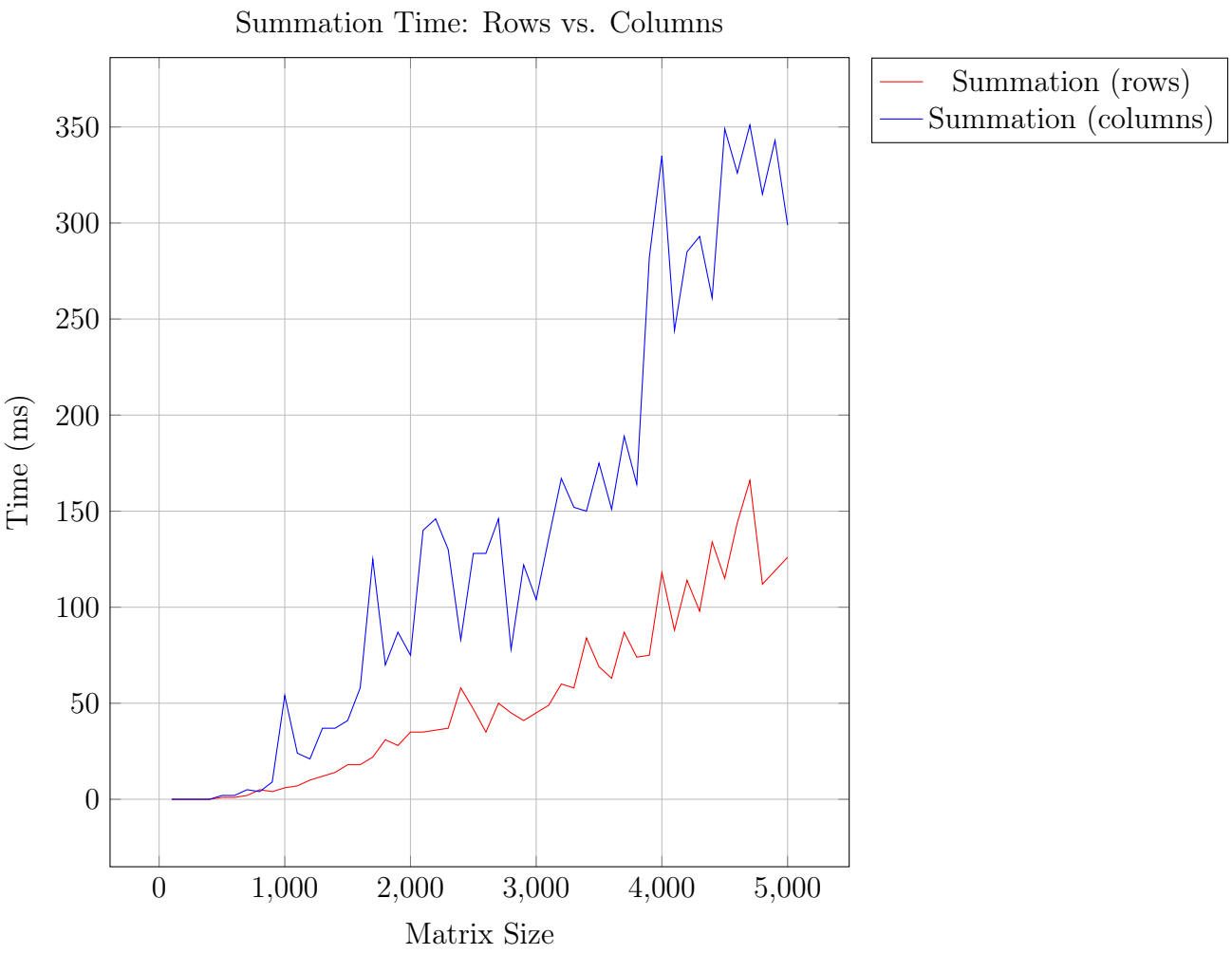


Figure 1.1: Execution Time for Summing Matrix by Rows vs. Columns

The drastic performance difference as the 'size' parameter varies depends on the cache structure; it is, in fact, composed of 'cache lines,' and when a single byte of cache is read, the entire cache line is actually accessed. Since matrix elements are stored in memory row by row, accessing them in the same manner will be faster as it sequentially accesses the same cache line, minimizing 'cache misses.'

Sequential memory access benefits performance due to prefetching: the hardware speculatively caches the 'n+1' cache line during a forward traversal of the 'n' cache line; similarly, it caches the 'n-1' cache line during a reverse traversal.

Cache Coherency The cache at the hardware level ensures that stored values remain consistent with those in memory by marking the corresponding cache line as 'dirty' and invalidating it when a data write occurs in memory.

1.4.2 False Sharing

Consider the following algorithms, where we are counting the number of even elements in a matrix by splitting the workload across a given number of threads (‘threadCount‘):

Codice 1.5: Example with False Sharing

```
static int[] countersWithFalseSharing;

static void RunWithFalseSharing(int threadCount)
{
    Thread[] threads = new Thread[threadCount];
    for (int i = 0; i < threadCount; i++)
    {
        int threadIndex = i;
        threads[i] = new Thread(() =>
        {
            int rowsPerThread = matrixSize / threadCount;
            int startRow = threadIndex * rowsPerThread;
            int endRow = startRow + rowsPerThread;

            for (int row = startRow; row < endRow; row++)
            {
                for (int col = 0; col < matrixSize; col++)
                {
                    if (matrix[row, col] % 2 == 0)
                    {
                        countersWithFalseSharing[threadIndex]++;
                    }
                }
            }
        });
        threads[i].Start();
    }

    foreach (var thread in threads)
    {
        thread.Join();
    }
}
```

In this first algorithm, whenever an even number is found, an element of a global vector is incremented at the index corresponding to the initialized thread’s index (highlighted in orange). After all threads have completed execution, the individual elements of the global vector ‘counters‘ can be summed to obtain the correct result.

Codice 1.6: Example Without False Sharing

```
static int[] countersWithoutFalseSharing;

static void RunWithoutFalseSharing(int threadCount)
{
    Thread[] threads = new Thread[threadCount];
    for (int i = 0; i < threadCount; i++)
    {
        int threadIndex = i;
        threads[i] = new Thread(() =>
        {
            int localCounter = 0;
            int rowsPerThread = matrixSize / threadCount;
            int startRow = threadIndex * rowsPerThread;
            int endRow = startRow + rowsPerThread;

            for (int row = startRow; row < endRow; row++)
            {
                for (int col = 0; col < matrixSize; col++)
                {
                    if (matrix[row, col] % 2 == 0)
                    {
                        localCounter++;
                    }
                }
            }

            countersWithoutFalseSharing[threadIndex] = localCounter;
        });
        threads[i].Start();
    }

    foreach (var thread in threads)
    {
        thread.Join();
    }
}
```

In the second algorithm, the same operations are performed, but instead of incrementing the corresponding thread's vector element for every new even number found, a local counter variable 'localCounter' (highlighted in green) is defined within each thread. At the end of the iteration, its value is assigned to the corresponding vector element (highlighted in orange).

This modification results in a significant performance improvement as the number of threads increases:

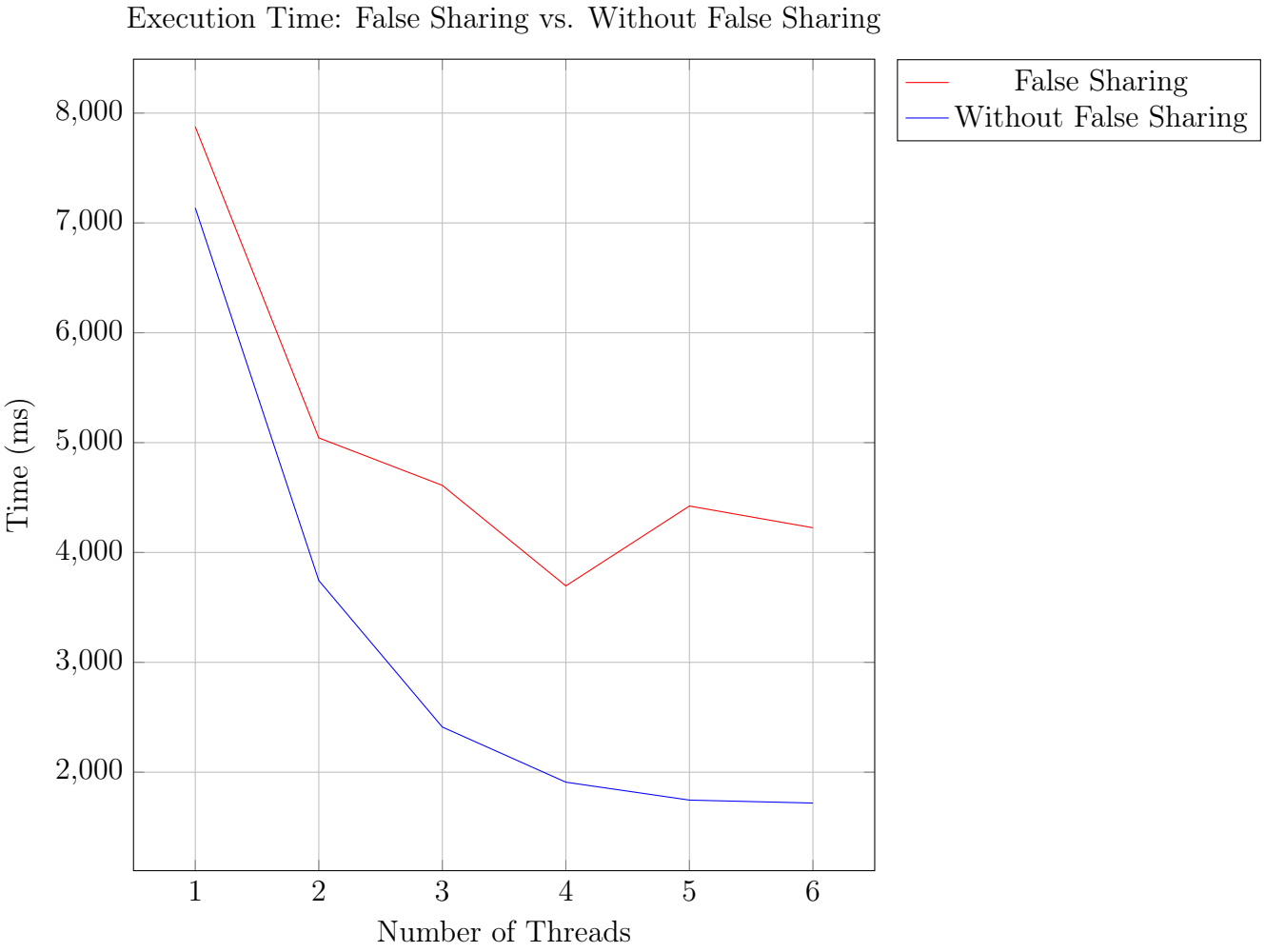


Figure 1.2: Execution Time to Find Even Numbers in a 20000x20000 Matrix with and without False Sharing

This behavior is due to the phenomenon of false sharing: although the individual threads do not share the same resources (each thread accesses a different index of the ‘counters‘ vector), because the vector is sequentially stored in memory, every write to the vector invalidates the entire cache line (which likely contains the whole vector). For the hardware, it is as though the threads are accessing a single shared resource.

In the second case, however, since the local counters are defined within individual threads, they are unlikely to be stored close to each other in memory. The same phenomenon reoccurs when the local counter's value is assigned to the vector, but

this happens only once per thread, thus not creating overhead [5].

The DOD paradigm, therefore, leverages prefetching to improve program performance and makes code more easily parallelizable by enabling bulk operations on objects of the same type, while minimizing false sharing. In the next chapter, we will discuss the implementation of these concepts in the Unity game engine using its Data-Oriented Technology Stack (DOTS).

Chapter 2

Data-Oriented Programming Applied: Unity DOTS

2.1 Unity DOTS: Definitions and Roadmap

Unity Unity is a powerful real-time 3D development engine traditionally based on object-oriented paradigms. It is widely used for game development due to its flexibility and extensive set of tools, enabling developers to create a variety of interactive applications. Unity's traditional architecture is largely object-oriented, featuring a scene containing GameObjects arranged in a hierarchy. Each GameObject is composed of various components managing different characteristics such as position, rotation, scale (`Transform`); rigid body dynamics (`RigidBody`); collision detection (`Colliders`); and custom components defined as `MonoBehaviour` scripts to further define GameObject behavior [6].

2.1.1 Unity DOTS (Data-Oriented Technology Stack)

Unity DOTS represents a significant shift from this object-oriented approach toward a more data-oriented methodology. DOTS is based on the Entity Component System (ECS) architecture, focusing on performance and scalability through the principles of data-oriented design discussed in the previous chapter. Using ECS, Unity DOTS emphasizes composition over inheritance, allowing developers to define only the data (Components) and behaviors (Systems) required for each Entity.

In addition to ECS, Unity DOTS includes:

- The *C# Job System*, which enables safe and fast parallelization of tasks across multiple entities using Unity's internal *Job System* written in C++.
- The *Burst Compiler*, which translates IL/.NET bytecode into highly optimized native code using the LLVM compiler infrastructure [7].

2.1.2 History and Roadmap of Unity DOTS Development

Unity began experimenting with DOTS around 2018, with the first stable version of ECS released in 2020-2021, accompanied by various features to optimize performance. In 2023, further integrations were introduced, such as improved support in IDEs like Rider, which added new tools to simplify development, enabling quicker creation of scripts and components [8].

Future plans include implementing an animation and character control system utilizing ECS, integrating GameObjects with Entities, and developing the ability to navigate ECS-based scenes within the Unity Editor, though this feature is currently in its early stages [9].

2.2 Structure of Unity DOTS

Library	Description
C# Job System	Enables the creation of parallel and multithreaded jobs to improve code performance. Simplifies asynchronous operation execution.
Burst Compiler	A compiler that transforms C# code into high-performance native code optimized for specific hardware, significantly enhancing execution speed.
Collections	Contains data structures optimized for multithreaded environments (e.g., `NativeArray`, `NativeList`, etc.), essential for working with shared data in jobs.
Entities (ECS)	Entity Component System (ECS) framework for organizing data and logic to optimize large-scale performance by separating data from processing logic.

2.2.1 C# Job System

In Unity's traditional design, updates to individual `MonoBehaviour` components are executed on the main thread, meaning the entire game logic runs on a single CPU core. Developers could manually create new threads, but there was no safe and straightforward way to manage them.

How the Job System Works

The Job System manages a pool of worker threads, one for each additional core on the target platform. For instance, if Unity runs on an eight-core processor, a main thread is created along with seven worker threads. When a worker thread is idle, it executes the next available job from the queue. Each job, once started on a worker

thread, runs to completion without interruption.

Jobs can only be scheduled (added to the queue) from the main thread. Additionally, only the main thread can call `Complete()` on a Job to wait for it to finish (if it hasn't already) and release its data for safe use on the main thread or to pass it to another Job [6].

Safety Checks and Job Dependencies

Ensuring safety and managing dependencies between threads is crucial in multithreaded programming to avoid race conditions, data corruption, and other concurrency issues.
For example, if the Job System sends a reference to data from the main thread code to a Job, there's no guarantee that the main thread isn't reading the data simultaneously as the Job modifies it. This scenario creates a *race condition*.

The Job System mitigates these issues by isolating data within individual Jobs, sending a copy of input data rather than passing it by reference from the main thread. As a result, only blittable data types can be passed to Jobs [10]; the implications of this are discussed in the following section.
If two Jobs need to use the same data, developers can define dependencies between Jobs when they are scheduled, creating a chain of Jobs where a Job can only start after all its dependencies have finished executing.

Codice 2.1: Example of Job Scheduling

```
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using UnityEngine;

public class JobDependenciesExample : MonoBehaviour
{
    private NativeArray<int> data;

    void Start()
    {
        // Use a 'blittable' data type
        data = new NativeArray<int>(1, Allocator.TempJob);

        // Create jobs with shared data
        var jobA = new IncrementJob { data = data };
        var jobB = new MultiplyJob { data = data };
        var jobC = new DecrementJob { data = data };

        // Schedule Jobs with dependencies
        JobHandle handleA = jobA.Schedule();
        JobHandle handleB = jobB.Schedule(handleA);  // B depends on A
        JobHandle handleC = jobC.Schedule(handleB);  // C depends on B

        // Wait for all Jobs to complete
        handleC.Complete();

        // Use the result
        Debug.Log("Final Value: " + data[0]);

        data.Dispose();
    }
}
```

The data structure is blittable, so the `garbage collection` does not manage it. Thus, it must be manually released using `data.Dispose()` (explained further in the next section).

Codice 2.2: Example Job Scheduling, Job Definitions

```
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using UnityEngine;

struct IncrementJob : IJob
{
    public NativeArray<int> data;

    public void Execute()
    {
        data[0]++;
    }
}

struct MultiplyJob : IJob
{
    public NativeArray<int> data;
    public void Execute()
    {
        data[0] *= 2;
    }
}

struct DecrementJob : IJob
{
    public NativeArray<int> data;
    public void Execute()
    {
        data[0]--;
    }
}
```

Job Types

In addition to the simple `IJob` interface demonstrated earlier, the Job System provides other job types useful for various scenarios.

IJobParallelFor Executes a task in parallel, where each thread has an exclusive index to access shared data safely. This is useful for performing element-wise operations on a dataset in parallel. The `Execute` function will have access to the `index` parameter, representing the index of the data element being processed.

Codice 2.3: Example of IJobParallelFor

```
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using UnityEngine;

public class ParallelForExample : MonoBehaviour
{
    private NativeArray<int> data;

    void Start()
    {
        data = new NativeArray<int>(10, Allocator.TempJob);
        var job = new ParallelForJob { data = data };

        // The first parameter is the total number of iterations
        // The second parameter is the batch size (number of elements processed
        ↪  per thread)
        JobHandle handle = job.Schedule(data.Length, 1);
        handle.Complete();

        data.Dispose();
    }

    struct ParallelForJob : IJobParallelFor
    {
        public NativeArray<int> data;

        public void Execute(int index)
        {
            data[index] = index * 2;
        }
    }
}
```

Unity internally resolves the *false sharing* issue discussed in the previous chapter: by requiring only `unmanaged` data types, the Job System can optimally manage memory, minimizing the likelihood of data elements being stored on the same cache line.

IJobFor Similar to `IJobParallelFor`, but allows scheduling the job for non-parallel execution. This is useful when a vectorized problem needs to be broken down into separate tasks without requiring parallel execution.

IJobParallelForTransform Executes a task in parallel, where each thread has an exclusive `Transform` from the hierarchy. This is optimized for modifying the position, scale, or rotation of multiple entities in a scene. In addition to the `Transform` index, it provides access to a `TransformAccess` object, simplifying these modifications [10].

Codice 2.4: Example of IJobParallelForTransform

```
using Unity.Burst;
using Unity.Jobs;
using UnityEngine;
using UnityEngine.Jobs;

public class ParallelForTransformExample : MonoBehaviour
{
    public Transform[] transforms;

    void Start()
    {
        TransformAccessArray transformAccessArray = new
        ↪  TransformAccessArray(transforms);
        var job = new ParallelForTransformJob();
        JobHandle handle = job.Schedule(transformAccessArray);
        handle.Complete();

        transformAccessArray.Dispose();
    }

    struct ParallelForTransformJob : IJobParallelForTransform
    {
        public void Execute(int index, TransformAccess transform)
        {
            transform.position += Vector3.up;
        }
    }
}
```

Job Type	Description
IJob	Executes a single task on a job-dedicated thread.
IJobParallelFor	Executes a task in parallel, with each parallel thread having an exclusive index to access shared data safely.
IJobParallelForTransform	Executes a task in parallel, with each thread exclusively accessing a `Transform` from the transform hierarchy.
IJobFor	Similar to `IJobParallelFor`, but schedules the job for sequential execution rather than parallel execution.

2.2.2 Managing Blittable Types: The Collections Package

Blittable Types In C#, data can be managed or unmanaged. Managed code runs within the .NET runtime, which handles memory management and garbage collection, while unmanaged code accesses memory directly, often interfacing with external libraries written in languages like C or C++.

Blittable types have identical memory representation in both managed and unmanaged code, meaning they can be directly transferred between the .NET runtime and unmanaged code without requiring conversions or transformations [11].

Garbage Collection Garbage collection is an automatic mechanism used in the .NET framework (and many other runtime environments) to manage memory. Its purpose is to reclaim memory allocated to objects no longer in use, freeing resources and preventing memory leaks [12].

Unmanaged types are not handled by the .NET runtime and thus are not subject to garbage collection. The `Collections` package provides unmanaged data structures (Native and Unsafe) that can be used in jobs and Burst-compiled code.

For `Native` collections, Unity performs automatic safety checks to ensure proper deallocation timing, while `Unsafe` collections have no such safeguards.

Overview of Allocators

An allocator manages unmanaged memory from which allocations can be made. The `Collections` package includes the following allocators:

- **Allocator.Temp**: The fastest allocator, used for short-term allocations that are automatically deallocated at the end of each frame or Job; it cannot be passed to Jobs.
- **Allocator.TempJob**: Used for short-term allocations that can be passed to Jobs; must be deallocated within 4 frames of creation.

- **Allocator.Persistent**: Used for allocations with indefinite duration, requiring manual deallocation; no safety checks are performed for allocation duration [13].

2.2.3 Burst Compiler

Traditional Compilation in Unity

C# code in Unity is compiled by default using Mono, a JIT (just-in-time) compiler, or alternatively with IL2CPP, an AOT (ahead-of-time) compiler.

JIT Compilation With JIT compilation, the C# code written for Unity is translated in real time during game or application execution. This means that when a Unity application runs on a machine, Mono converts the C# bytecode into native machine code during runtime. The advantage of this approach is the compiler's ability to optimize the code based on the specific device's characteristics, such as the processor type, potentially improving performance. However, JIT compilation can introduce an initial delay because the code translation occurs at runtime, slowing down application startup.

AOT Compilation With AOT compilation in Unity, C# code is transformed into native code before the application is deployed or executed. This process uses IL2CPP, which converts C# intermediate language code into C++ and then into native code. AOT compilation is particularly beneficial for platforms where JIT is not feasible or advisable, such as mobile devices (Android and iOS) or consoles. Because the code is already compiled into native code, application startup is faster compared to JIT. However, since the compilation occurs at build time, dynamic optimizations based on the specific device are not possible, as in JIT [14].

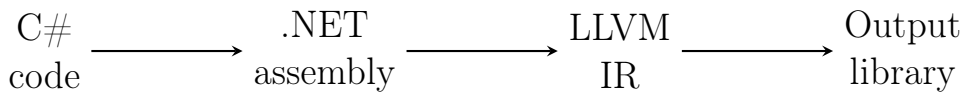
Compilation with Burst

The Burst package introduces a third compiler that performs substantial optimizations, significantly improving performance and scalability for computation-heavy problems.

Burst can only compile a subset of C#, so much of typical C# code cannot be compiled with it. The main limitation is that Burst-compiled code cannot access managed objects, including all class instances. Since this excludes most conventional C# code, Burst compilation is applied only to selected parts of the code, such as Jobs [3].

How It Works Game C# code is compiled into .NET assemblies, converting from IL (Intermediate Language) to LLVM IR (Intermediate Representation). Burst leverages Unity APIs to optimize and compile the code using a customized LLVM build.

LLVM (Low Level Virtual Machine) is a toolchain for compiler development designed to optimize code at compile time and runtime. The process generates platform-specific native libraries via AOT compilation. During runtime, Burst functions call these native libraries instead of the original .NET code [15].



Usage To compile a class or function with Burst, annotate it with the `[BurstCompile]` attribute, provided the code adheres to the previously mentioned constraints. Optional annotations are also available to give the compiler additional information about the code, further optimizing the compilation process.

Optimizations

Loop Vectorization The Burst compiler utilizes SIMD (Single Instruction, Multiple Data) instructions to execute multiple iterations of a loop simultaneously when possible. This is especially beneficial on modern processors that support vectorized operations.

For example, consider the following Burst-compiled C# code snippet, which computes the element-wise sum of two arrays:

Codice 2.5: Example of Loop Vectorization in C#

```
using Unity.Burst;
using Unity.Jobs;

public class VectorizationExample
{
    private static unsafe void Bar([NoAlias] int* a, [NoAlias] int* b, int
    ↪  count)
    {
        for (var i = 0; i < count; i++)
        {
            a[i] += b[i];
        }
    }

    public static unsafe void Foo(int count)
    {
        var a = stackalloc int[count];
        var b = stackalloc int[count];

        Bar(a, b, count);
    }
}
```

When compiled with Burst, the loop is unrolled and vectorized, resulting in the following assembly code:

Codice 2.6: Vectorized Loop Compiled with Burst

```
.LBB1_4:
    vmovdqu     ymm0, ymmword ptr [rdx + 4*rax]
    vmovdqu     ymm1, ymmword ptr [rdx + 4*rax + 32]
    vpaddd      ymm0, ymm0, ymmword ptr [rcx + 4*rax]
    vpaddd      ymm1, ymm1, ymmword ptr [rcx + 4*rax + 32]
    vmovdqu     ymmword ptr [rdx + 4*rax], ymm0
    vmovdqu     ymmword ptr [rdx + 4*rax + 32], ymm1
    add         rax, 16
    cmp         r8, rax
    jne         .LBB1_4
```

In this example, Burst unrolled the loop and transformed it into SIMD operations, processing multiple iterations in a single instruction. This transformation significantly improves execution speed.

Additionally, developers can use intrinsic hints to ensure specific loops are vectorized or to enforce certain optimization behaviors:

Codice 2.7: Example of Vectorization Intrinsic Hint

```
private static unsafe void Bar([NoAlias] int* a, [NoAlias] int* b, int count)
{
    for (var i = 0; i < count; i++)
    {
        Unity.Burst.CompilerServices.Hint.ExpectVectorized(); // Ensure
        ↪ vectorization
        a[i] += b[i];
    }
}
```

If the compiler cannot vectorize the code due to unsupported patterns (e.g., branching), an error will be generated, allowing developers to adjust their implementation accordingly [16].

Memory Aliasing Memory aliasing occurs when multiple references point to the same memory location, making it difficult for the compiler to determine whether modifying one variable affects another. This can prevent certain optimizations, as redundant operations may be necessary to ensure correctness.

Consider the following example:

Codice 2.8: Example of Memory Aliasing in C#

```
public static void MayAlias(ref int a, ref int b, ref int c)
{
    b = a;
    c = a;
}
```

When compiled, the following assembly code is generated:

Codice 2.9: Memory Aliasing Assembly Code

```
mov eax, dword ptr [rcx]   ; eax = a
mov dword ptr [rdx], eax   ; b = a
mov eax, dword ptr [rcx]   ; eax = a (redundant load)
mov dword ptr [r8], eax    ; c = a
ret
```

In this example, the redundant load occurs because the compiler assumes that `b` or `c` might alias with `a`. Burst provides the `[NoAlias]` attribute to inform the compiler that memory aliasing is not possible, enabling better optimization.

Hint Intrinsics Developers can use intrinsics to provide additional hints to the compiler, such as indicating that a specific branch is more likely to be executed:

Codice 2.10: Example of Likely Intrinsics

```
if (Unity.Burst.CompilerServices.Hint.Likely(condition))
{
    // Optimized assuming this branch is more probable
}
else
{
    // Alternate path
}
```

Other intrinsics allow developers to define value ranges, constants, or variables that do not require initialization, enabling further optimizations [16].

2.2.4 Entity Component System (ECS)

The ECS Paradigm

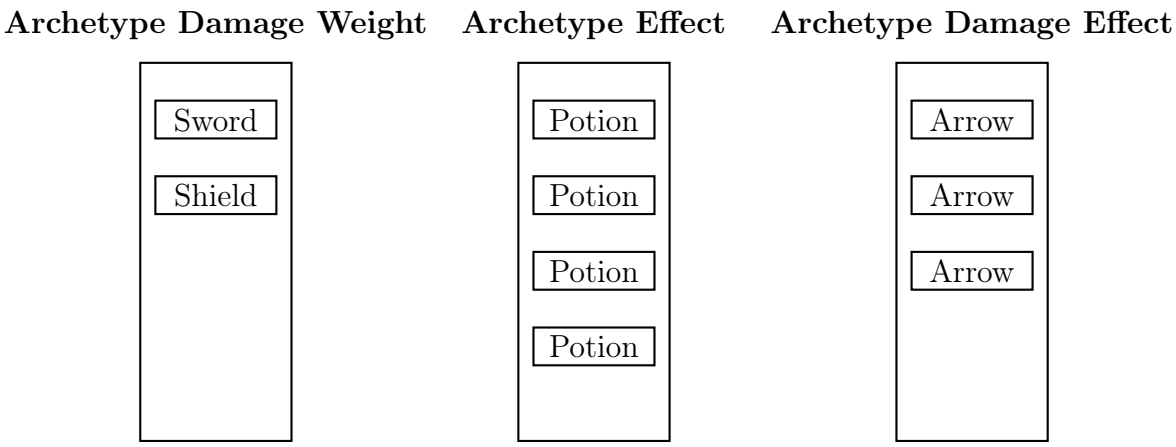
The ECS paradigm is at the heart of Unity DOTS, as it facilitates and enhances the effectiveness of the optimizations discussed earlier. The `Entities` package provides an implementation of ECS, composed of entities and components for data and systems for logic. Entities replace the GameObjects of traditional programming; like GameObjects, entities are composed of components, but unlike GameObjects, entity components contain only data, have no methods, and are typically structs

rather than MonoBehaviour classes.

Game logic is handled by systems, which have an update method usually invoked once per frame, reading and modifying entity components.

Archetypes

All entities sharing the same set of component types are grouped and stored within the same archetype. For example:

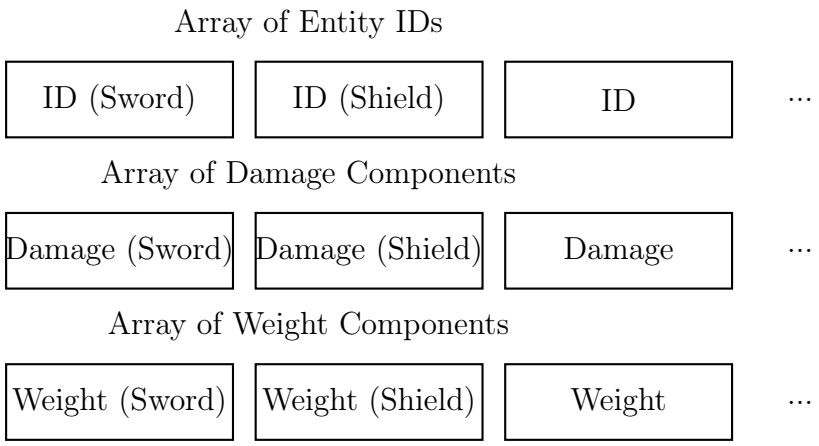


The entities "Sword" and "Shield" belong to the same archetype because they share the same component types, `Damage` and `Weight`.

At runtime, an entity may gain or lose components (e.g., the "Arrow" entity might acquire a "Speed" component). When this occurs, the entity moves to the corresponding archetype.

Chunks

Within an archetype, entities are stored in memory blocks called *chunks*, each capable of holding up to 128 entities. The IDs of entities and the components of each type within the same chunk are stored sequentially in arrays. Using the previous example, the chunk for the "Damage and Weight" archetype would be structured as follows:



To access all the components and the ID of the "Sword" entity, for instance, you would access the elements at index 0 in each array. Furthermore, if an entity is removed from a chunk, the last entity in the chunk is moved to the vacant index to keep the arrays compact.

Systems

One major advantage of the archetype and chunk-based data organization is that it allows efficient queries and iterations over entities. To iterate over all entities with a specific set of components, a query first identifies all archetypes matching the criteria and then iterates through the entities in the chunks of those archetypes. Because components in chunks reside in compact arrays, cache misses are minimized. Additionally, archetypes matching a query can be cached for reuse.

Consider the following system, which moves all `Enemy` entities to the right each frame:

Codice 2.11: Example System

```
using Unity.Burst;
using Unity.Entities;
using Unity.Transforms;
using Unity.Mathematics;

public struct EnemyTag : IComponentData
{
    // Empty component used to "tag" entities for the EnemyMovementSystem query
}

public partial struct EnemyMovementSystem : ISystem
{
    public void OnUpdate(ref SystemState state)
    {
        float3 moveDirection = new float3(1, 0, 0); // Move right

        // Using SystemAPI, iterate over all entities with the 'EnemyTag',
        ↪  'Translation', and 'MoveSpeed' components
        foreach (var (translation, speed) in SystemAPI.Query<RefRW<Translation>,
        ↪  RefRO<MoveSpeed>>()
                 .WithAll<EnemyTag>())
        {
            translation.ValueRW.Position += moveDirection * speed.ValueRO.Value
            ↪  * SystemAPI.Time.DeltaTime;
        }
    }
}
```

In the `foreach` loop, we iterate over the `Translation` component in read-write mode (`RefRW`) because its value (position) is modified. For `MoveSpeed`, we use read-only mode (`RefRO`) because we only need to read its value (speed) without modifying it.
`SystemAPI.Time.DeltaTime` represents the time elapsed since the last frame and is used as a factor to ensure the position changes consistently, regardless of the frame rate.

Integration with the Job System

Unity DOTS allows integration of the Job System with the ECS paradigm, enabling tasks to execute over specific queries, thereby parallelizing the updates of individual entities:

Codice 2.12: Example Job System with ECS

```
using Unity.Burst;
using Unity.Entities;
using Unity.Jobs;
using Unity.Transforms;
using Unity.Mathematics;

public struct EnemyTag : IComponentData {}

public partial struct EnemyMovementSystem : ISystem
{
    public void OnUpdate(ref SystemState state)
    {
        float deltaTime = SystemAPI.Time.DeltaTime;
        float3 moveDirection = new float3(1, 0, 0); // Move right

        // Create a new Job for EnemyMovementSystem derived from IJobEntity
        var job = new MoveEnemiesJob
        {
            DeltaTime = deltaTime,
            MoveDirection = moveDirection
        };

        job.ScheduleParallel();
    }
}

public partial struct MoveEnemiesJob : IJobEntity
{
    public float3 MoveDirection;
    public float DeltaTime;

    public void Execute(ref Translation translation, in MoveSpeed speed, in
    ↪  EnemyTag tag)
    {
        translation.Value += MoveDirection * speed.Value * DeltaTime;
    }
}
```

The query previously in `EnemyMovementSystem` is now implicitly executed in `MoveEnemiesJob` based on the parameters of the `Execute` method. The `ref` keyword indicates read-write mode, while `in` indicates read-only mode [3].

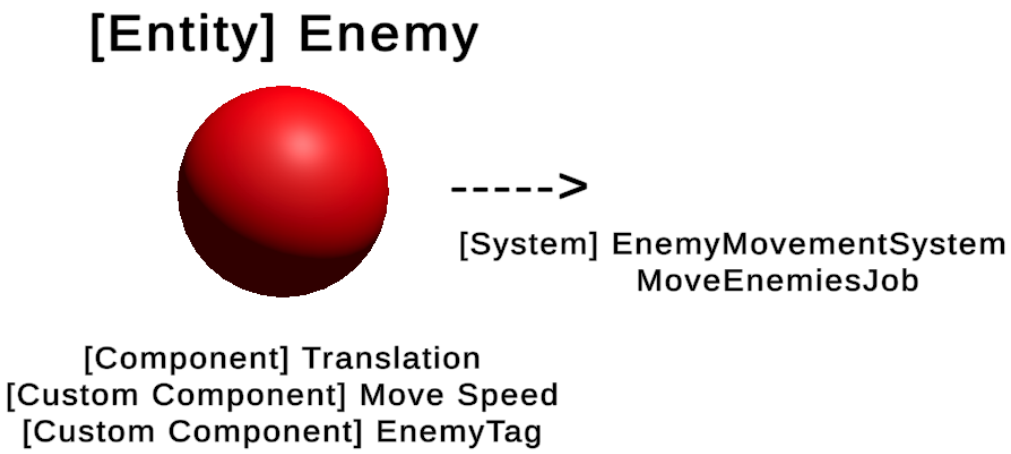


Figure 2.1: Enemy Movement Example

2.2.5 Conversion Between GameObjects and Entities

Unity's traditional scene system is incompatible with ECS. To take advantage of ECS features while still utilizing Unity's graphical editor for scene manipulation, *subscenes* are used.
Subscenes are nested within the current scene and are processed via *baking*, which occurs whenever modifications are made to the subscene. During the baking process, each GameObject in the subscene is converted into an entity, which is then loaded at runtime.
For each component of the original GameObject, a corresponding *baker* is required to add the appropriate component to the resulting entity. For example, standard graphical components such as `MeshRenderer` are converted into graphical components for the entity.

Custom Bakers For any additional non-standard components created by the developer, custom bakers must be defined to control which components are added to the generated entities.

For example, suppose we want to create a `Health` component representing an entity's health points as a float:

Codice 2.13: Example Health Component

```
using Unity.Entities;

public struct Health : IComponentData
{
    public float Value; // Health points of the entity
}
```

In addition to the `Health` component, you must define a `MonoBehaviour` class, `HealthAuthoring`, which can be edited in Unity's Inspector, and a `Baker<HealthAuthoring>` class, `HealthBaker`, which converts the `HealthAuthoring` component into a `Health` component at runtime:

Codice 2.14: Example Health Authoring and Baker

```
using Unity.Entities;
using UnityEngine;

public class HealthAuthoring : MonoBehaviour
{
    public float HealthValue; // Initial health points set in the Unity Editor
    ↪ Inspector
}

public class HealthBaker : Baker<HealthAuthoring>
{
    public override void Bake(HealthAuthoring authoring)
    {
        // Convert GameObject into an entity
        var entity = GetEntity(TransformUsageFlags.Dynamic);

        // Add the 'Health' component with the initial value from the
        ↪ 'HealthAuthoring' component
        AddComponent(entity, new Health { Value = authoring.HealthValue });
    }
}
```

Although directly adding entities to scenes might seem inconvenient for simple cases, the baking process proves advantageous in more complex scenarios: authoring data (GameObjects edited in the Unity Editor) is effectively separated from runtime

data (entities generated by baking). This separation ensures flexibility, as the data developers modify during design time does not have to directly correspond to runtime entities [3].

In the next chapter, we will describe a set of experiments and benchmarks conducted on the Unity DOTS components analyzed so far, applied to resource-intensive scenarios.

Chapter 3

Implementation and Experimentation with Unity DOTS

3.1 General Information

3.1.1 Project Goal

The goal of this project is to conduct a detailed and objective experimentation of the capabilities offered by Unity DOTS, comparing it with the traditional object-oriented architecture typical of MonoBehaviour-based approaches. Additionally, the application provides users with the ability to benchmark scenarios by varying their complexity through the number of entities. The starting point for each test is determined by a user-specified number of entities.

3.1.2 Project Structure

The Unity project is organized into a total of 10 different scenes. The first scene serves as the main hub, from which the user can select one of the three proposed scenarios or perform benchmarking. Each scenario is divided into three separate scenes to experiment with different levels of performance and code optimization. Each scenario represents a distinct composition, which will be analyzed later to study and compare performance under various conditions.

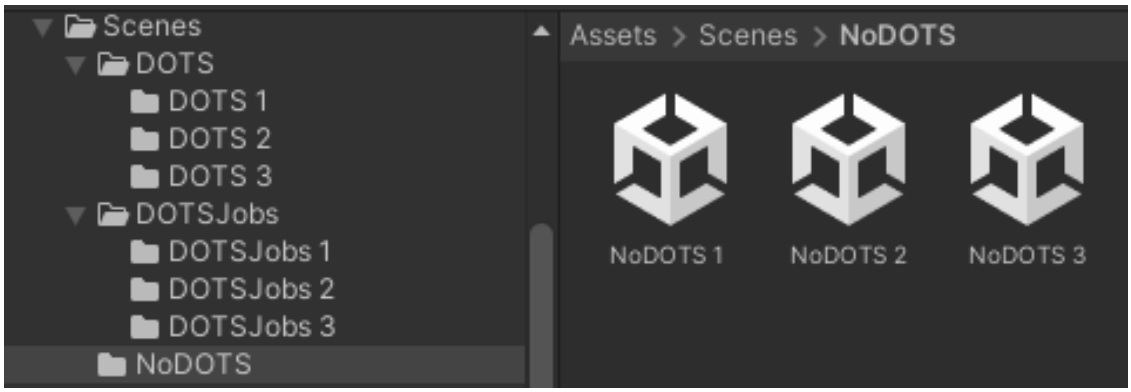


Figure 3.1: Scene Structure

3.1.3 Optimization Modes

For each of the proposed scenarios, three different optimization modes are available:

- **No Optimizations:** This mode represents the baseline scenario, developed using the traditional object-oriented architecture with scripts implemented via MonoBehaviour. It serves as a reference for comparing performance with other optimization modes.
- **Burst:** In this mode, the scenario is rewritten using ECS, and the code is compiled with Burst.
- **Burst + Jobs:** This mode combines ECS and Burst while adding the Job System. For any case requiring iteration over multiple entities to modify their properties or perform calculations, jobs are instantiated and managed in parallel to further enhance overall performance.

3.1.4 UI Description

Hub In the 'Hub' scene, users can select a scenario and the desired optimization level from a dropdown menu. Hovering over the respective button displays a description of the selected optimization on the screen. Additionally, users can switch to benchmark mode and modify its settings, including snapshot duration, the base number of entities, entity increment per snapshot, and the duration of each snapshot. The total benchmarking duration based on these settings is also displayed.

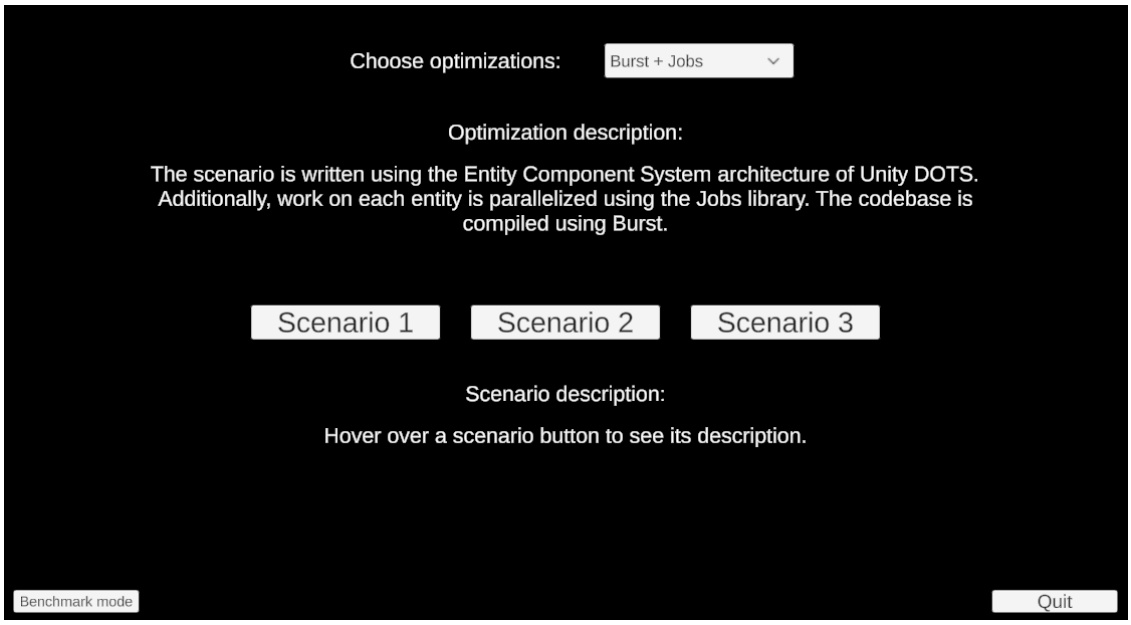


Figure 3.2: Hub UI

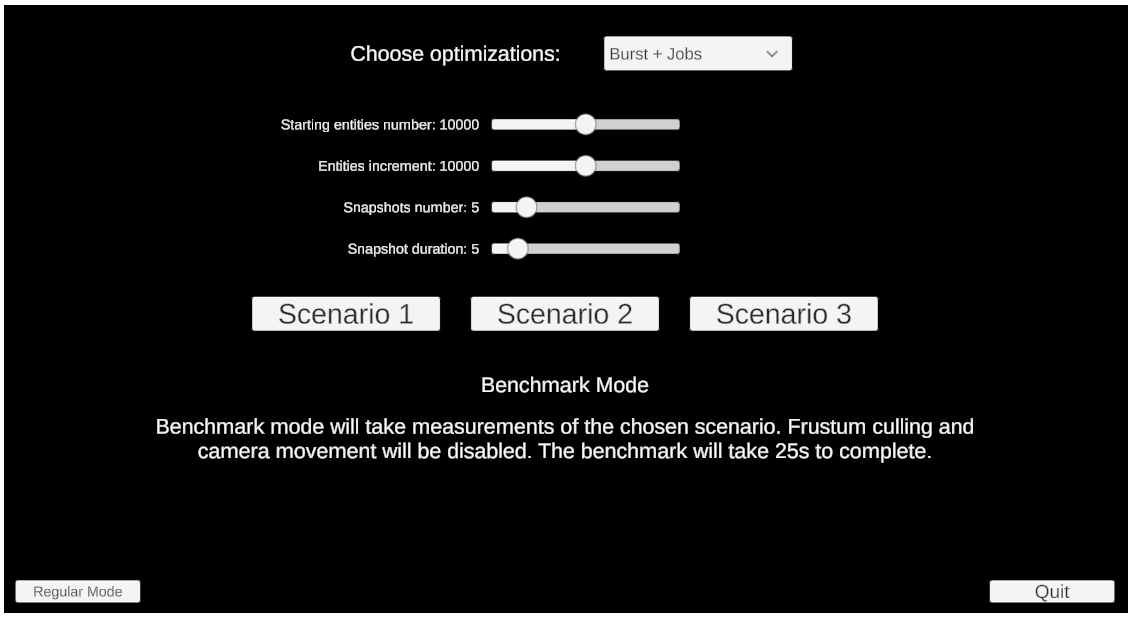


Figure 3.3: Hub Benchmark UI

Scenario When a scenario and desired optimization level are selected from the 'Hub' scene, the user enters the specific scenario scene. This scene features two sliders to modify the parameters `numEntities` and `spawnRadius`, a button to restart the simulation, and a button to return to the 'Hub' scene. Additionally, pressing

the [ESC] key hides the UI, allowing users to explore the scene and navigate using the 'WASD' or arrow keys. The height can be adjusted using [SPACE] and [SHIFT], while faster movement is enabled by holding [LEFT CONTROL].

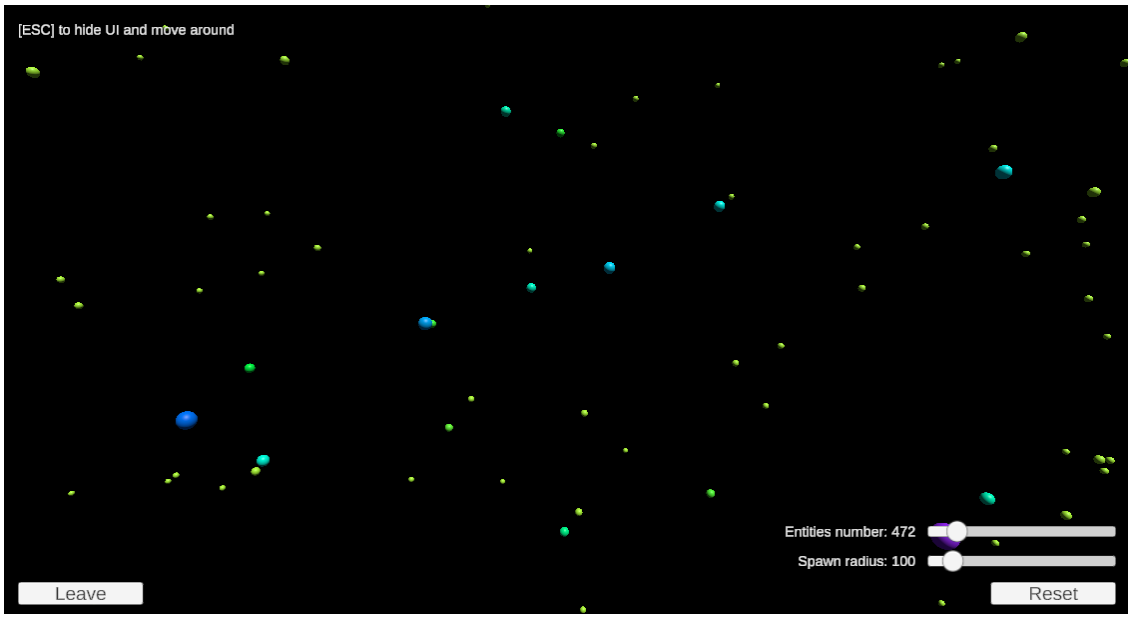


Figure 3.4: Scenario UI

In benchmark mode, interaction with the simulation is restricted (except for exiting), and at the end of the process, the collected data is displayed on-screen in JSON format.

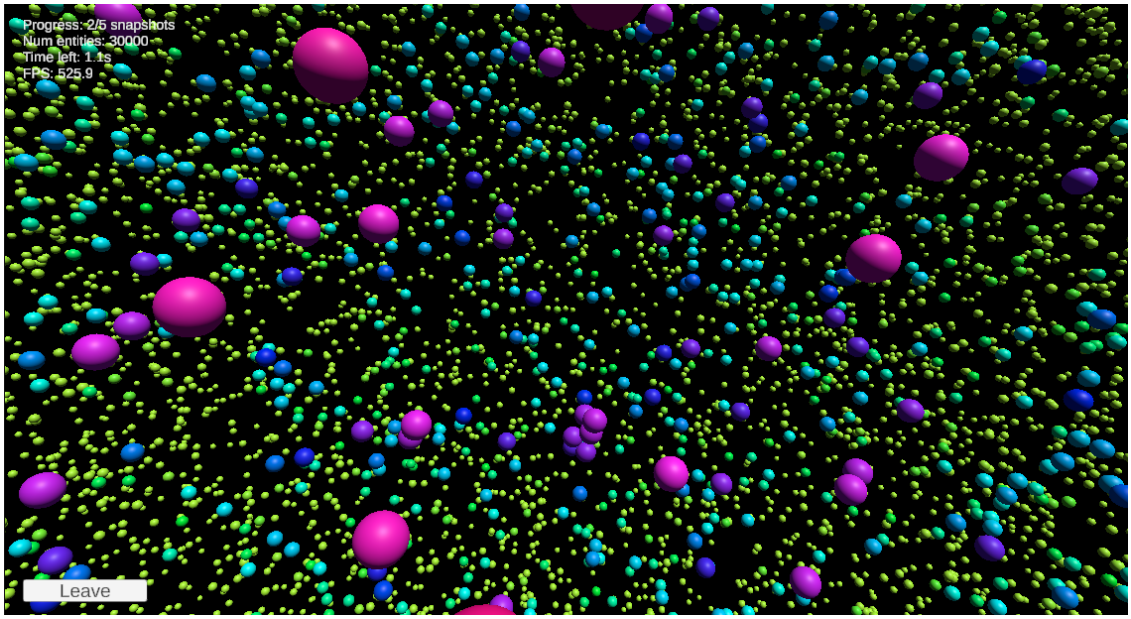


Figure 3.5: Benchmark in Progress UI

Figure 3.6: Benchmark Complete UI

3.1.5 Scenario Descriptions

All scenarios involve a `numEntities` number of 'sphere' entities instantiated within a `spawnRadius`. The difference between the scenarios lies in the behavior of the spheres.

Scenario 1 In the first scenario, spheres move back and forth along a random direction with a fixed base speed. The spheres' speed increases within a certain limit as they approach the user, along with their color intensity. The most computationally complex operation performed by each sphere per frame is calculating the distance between the sphere and the user's position (the `MainCamera` object position). For this scenario, approximately `O(numEntities)` computationally significant operations are performed per frame.

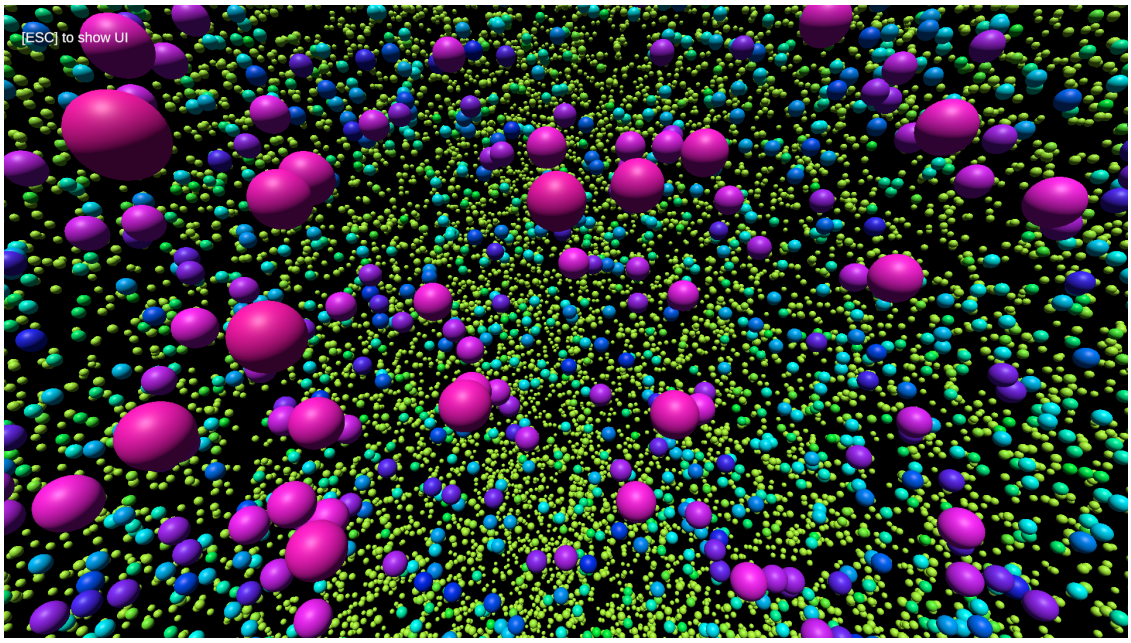


Figure 3.7: Scenario 1

Scenario 2 In the second scenario, spheres move back and forth along a random direction but with a speed that remains constant. Each sphere orients a cylindrical object called 'Link' toward the nearest sphere. The most computationally complex operation performed by each sphere per frame is calculating the distance to all other spheres and transforming the 'Link' entity to point to the nearest sphere. In this scenario, approximately $\texttt{O(numEntities}^2\texttt{)}$ computationally significant operations are performed per frame.

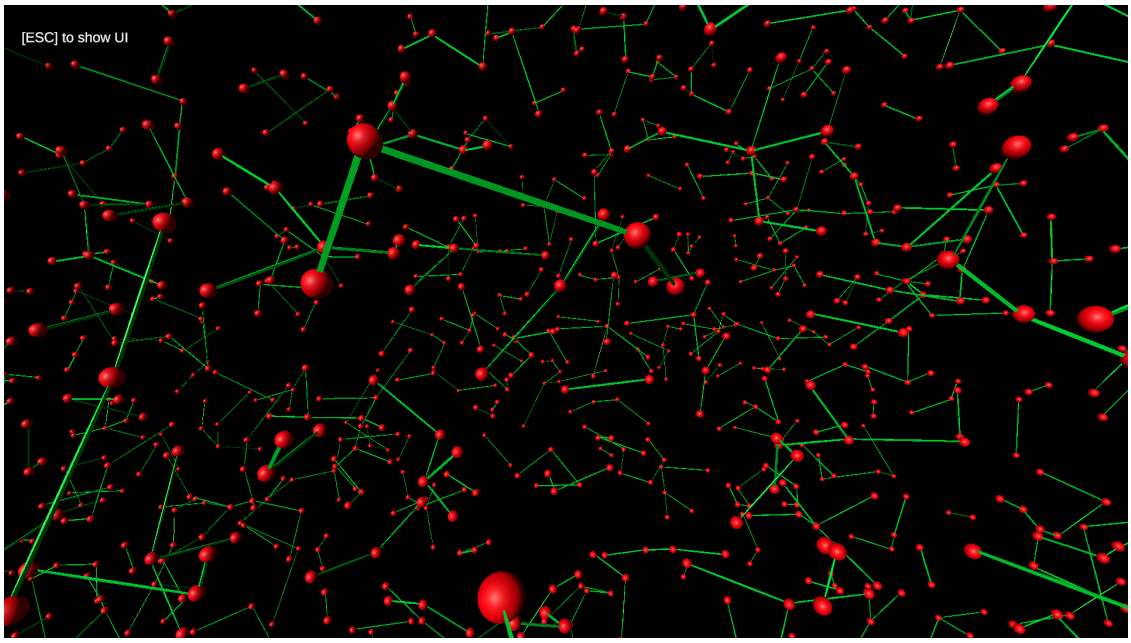


Figure 3.8: Scenario 2

Scenario 3 In the third and final scenario, each sphere has a rigid body and collision bounds and is propelled in a random direction by an initial force. Each sphere detects collisions, changing its color based on the type of collision (sphere-sphere or sphere-wall). The walls expand based on the `spawnRadius` parameter.

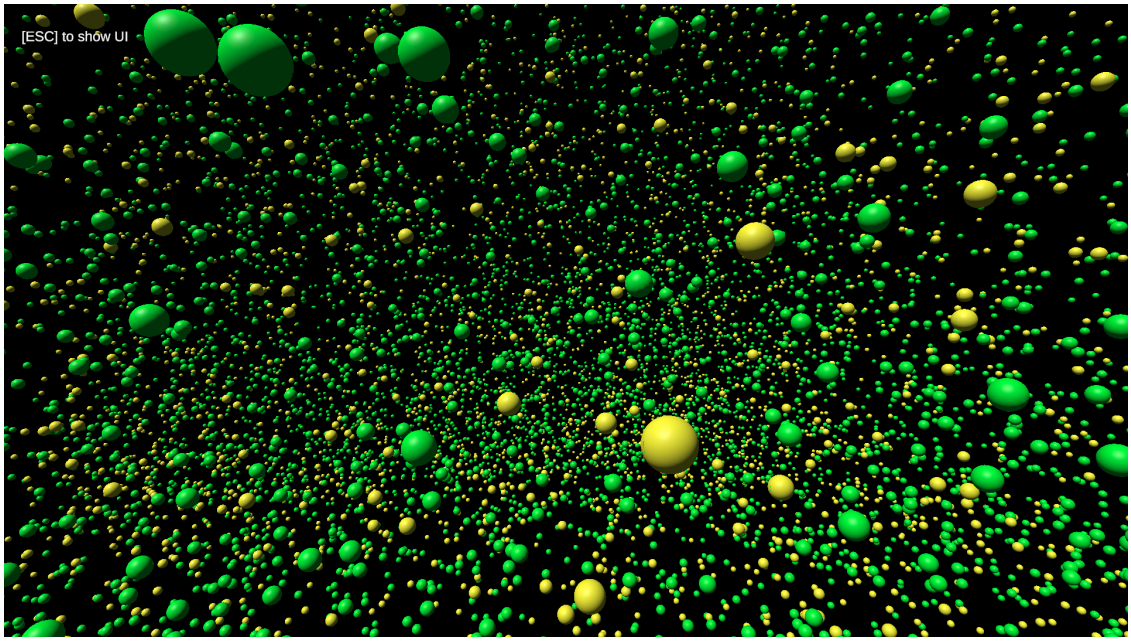


Figure 3.9: Scenario 3

3.2 Development Tools and Challenges

Before analyzing the functioning of the scenarios under different optimization levels, it is essential to report information about the overall project development.

3.2.1 MVC Architecture for UI

Scripts for each scenario are not reused across other scenarios; however, common scripts are required to manage the graphical interface and settings for the scenarios and benchmarking. These scripts are designed using the MVC (Model View Controller) paradigm. Thus, there are three types of scripts:

- **Handlers:** MonoBehaviour scripts that directly respond to user input (e.g., camera movement or changing scenario settings). If necessary, they invoke API scripts. In the MVC model, Handlers correspond to the 'View'.
- **APIs:** Static classes whose functions are invoked by Handlers (e.g., saving benchmarking settings). In the MVC model, APIs correspond to the 'Controller'.
- **Objects:** Serializable classes containing only data. In the project, they represent possible settings for a scenario or benchmarking. In the MVC model, Objects correspond to the 'Model'.

3.2.2 Communication Between MonoBehaviour UI and Entity World

A key challenge in developing scenarios entirely composed of Burst-compilable code is managing the transfer of information, such as scenario settings, from the traditional GameObject world to the Entity world. This is necessary because graphical objects like sliders and the overall Unity UI system are still incompatible with DOTS. Fortunately, scenario settings (`numEntities` and `spawnRadius`) only need to be loaded when the scenario is reset.

The solution involves a `SettingsLoaderSystem`, which retrieves the required parameters from the `ScenarioSettingsAPIs` class during its `OnUpdate` method and stores them in a `SettingsLoader` component. Since this system accesses managed objects via `ScenarioSettingsAPIs`, it cannot be compiled with Burst, as Burst only handles unmanaged data types, as previously discussed.

Codice 3.1: SettingsLoaderSystem.cs

```
[BurstCompile]
public partial struct SettingsLoaderSystem : ISystem
{
    private int _numSpheres;
    private float _spawnRadius;

    [BurstCompile]
    public void OnCreate(ref SystemState state)
    {
        // Required to access the SettingsLoader component in OnUpdate
        state.RequireForUpdate<SettingsLoader>();
    }

    // This function cannot be compiled with Burst due to interaction with
    ↪  managed types via ScenarioSettingsAPIs
    public void OnUpdate(ref SystemState state)
    {
        var config = SystemAPI.GetSingleton<SettingsLoader>();

        if (config.Initialized) return;

        // This flag ensures that the code in OnUpdate is executed only once per
        ↪  SettingsLoader (reset when the scenario restarts)
        config.Initialized = true;

        // Scenario settings are saved in SettingsLoader so other systems can
        ↪  access them without managed types
        config.numSpheres = ScenarioSettingsAPIs.GetNumEntities();
        config.spawnRadius = ScenarioSettingsAPIs.GetSpawnRadius();
        config.benchmarkMode = ScenarioSettingsAPIs.IsBenchmarkMode();

        SystemAPI.SetSingleton(config);
    }
}
```

Other systems in the codebase that rely on scenario settings can be compiled with Burst, as they retrieve data directly from the `SettingsLoader` component, ensuring that `SettingsLoaderSystem`'s `OnUpdate` has already been executed.

Codice 3.2: Snippet of SpheresSpawnerSystem.cs

```
// Annotation ensures settings are retrieved at the correct time from the
↪  SettingsLoader component
[UpdateAfter(typeof(SettingsLoader.SettingsLoaderSystem))]
[BurstCompile]
public partial struct SpheresSpawnerSystem : ISystem
{
    // [...]

    [BurstCompile]
    public void OnUpdate(ref SystemState state)
    {
        // [...]
        var settings = SystemAPI.GetSingleton<SettingsLoader.SettingsLoader>();

        // Code using 'settings.numEntities' and 'settings.spawnRadius' to
        ↪  initialize spheres in the scene
        // [...]
    }
}
```

In each scene where settings need to be loaded, a GameObject with the MonoBehaviour component `SettingsLoaderAuthoring` can be added via the Unity Inspector. This process, as described in the previous chapter, bakes the GameObject into an entity with the `SettingsLoader` component upon scene startup. The authoring script paradigm is present for all components analyzed in this project.

Codice 3.3: SettingsLoaderAuthoring.cs

```
public class SettingsLoaderAuthoring : MonoBehaviour
{
    public class Baker : Baker<SettingsLoaderAuthoring>
    {
        public override void Bake(SettingsLoaderAuthoring spawnerAuthoring)
        {
            var entity = GetEntity(TransformUsageFlags.None);
            AddComponent(entity, new SettingsLoader());
        }
    }
}
```

3.2.3 Using Unity Profiler to Identify Bottlenecks

The Unity Profiler collects and visualizes performance-related data across various areas such as CPU, memory, rendering, and audio. It is a useful tool for identifying areas where performance can be improved. The Profiler allows pinpointing aspects like code, resources, scene settings, camera rendering, and build configurations that influence performance. Results are displayed through graphs, highlighting performance peaks. Additionally, it reveals which functions consume the most execution time per frame, enabling optimization focus on critical operations.

For instance, we can compare the Profiler output for Scenario 1 without Unity DOTS optimizations and with Burst and Job parallelization.

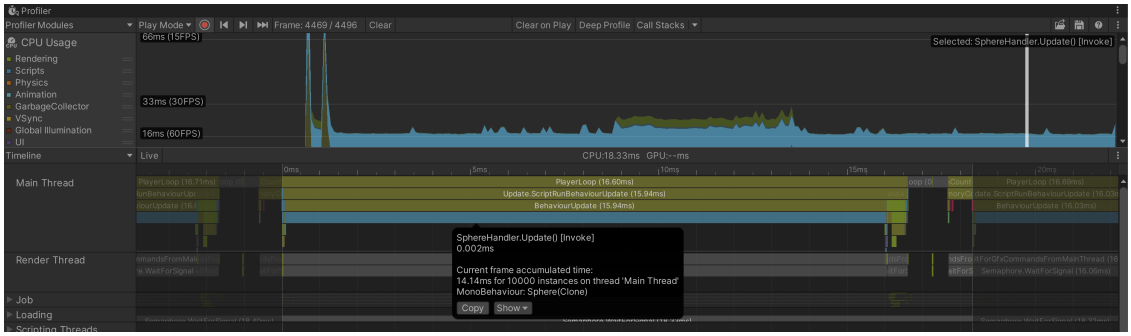


Figure 3.10: Unity Profiler Without DOTS

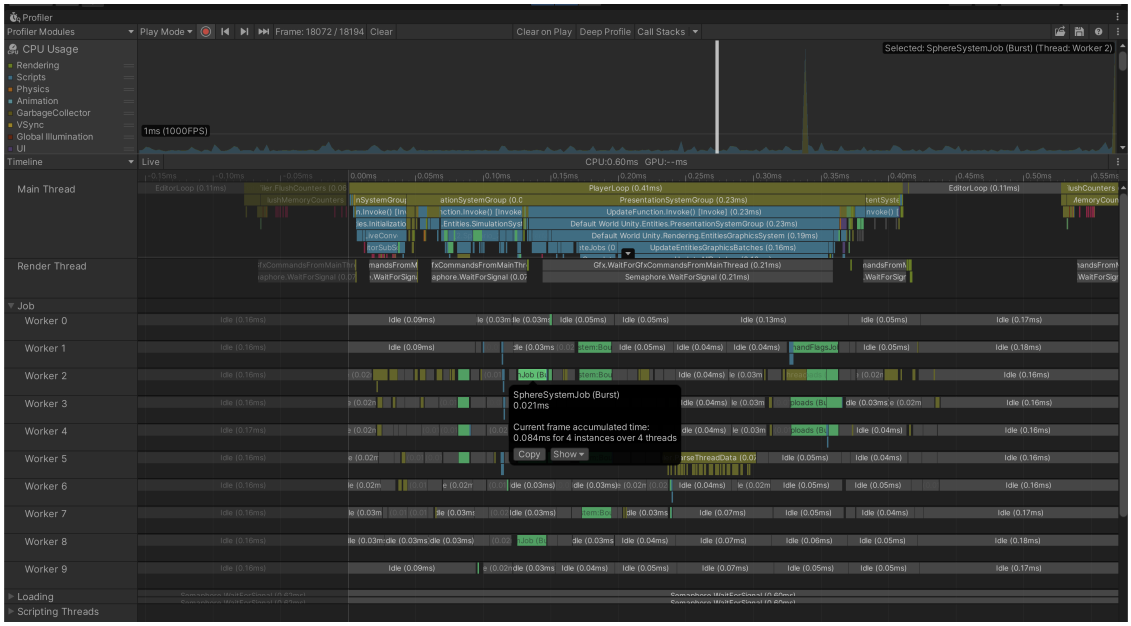


Figure 3.11: Unity Profiler With DOTS

As shown, in the first case, the `Update` function of the `Sphere` object occupies a

significant portion of the frame's execution time. In the second case, the workload is distributed across multiple worker threads.

3.2.4 Disabling Frustum Culling

Frustum culling is an optimization at Unity's game engine level that prevents rendering objects outside the camera's view frustum [18].
Although this optimization significantly enhances performance, during benchmarking, it is necessary to ensure that the simulation has consistent computational intensity regardless of the player's position.
Thus, the project includes scripts, both in MonoBehaviour and ECS versions, to disable frustum culling during benchmarking by enlarging the spheres' bounding boxes to ensure Unity always renders them.

Codice 3.4: FrustumCullingDisablerSystem.cs

```
[UpdateAfter(typeof(SettingsLoader.SettingsLoaderSystem))]
[BurstCompile]
public partial struct FrustumCullingDisablerSystem : ISystem
{
    [BurstCompile]
    public void OnCreate(ref SystemState state) =>
    ↪  state.RequireForUpdate<SettingsLoader.SettingsLoader>();

    [BurstCompile]
    public void OnUpdate(ref SystemState state)
    {
        var settings = SystemAPI.GetSingleton<SettingsLoader.SettingsLoader>();

        if (!settings.benchmarkMode) return;

        foreach (var (bounds, disabled) in
                 SystemAPI.Query<RefRW<RenderBounds>,
                 ↪  RefRW<FrustumCullingDisablerTag>>())
        {
            if (disabled.ValueRO.Initialized) continue;
            disabled.ValueRW.Initialized = true;

            bounds.ValueRW.Value = new Unity.Mathematics.AABB
            {
                Center = new Unity.Mathematics.float3(0, 0, 0),
                // Set the entity's bounding box extents to an arbitrarily large
                ↪  value
                Extents = new Unity.Mathematics.float3(1000, 1000, 1000)
            };
        }
    }
}
```

Chapter 4

Scenario Creation

SphereSpawnerSystem As mentioned earlier, the SphereSpawnerSystem class is responsible for instantiating the Sphere entities and initializing their parameters based on the scenario. At this stage, there is no significant difference in implementation compared to traditional MonoBehaviour programming; to instantiate a new entity, the EntityManager methods are called instead of GameObject methods, and in both cases, the initial values of the Entity/GameObject components are set.

The ECS-based system, in addition, must wait for the scenario settings to be converted by the SettingsLoaderSystem and, furthermore, since it can execute code exclusively in OnUpdate, it must deactivate itself after initializing the required entities; this is possible by using the 'Initialized' flag, initially set to 'false', of the SpheresSpawner component.

Codice 4.1: OnUpdate SpheresSpawnerSystem.cs - Scenario 1

```
[BurstCompile]
public void OnUpdate(ref SystemState state)
{
    // Used for the 'Initialized' flag
    var config = SystemAPI.GetSingleton<SpheresSpawner>();

    // Scenario settings
    var settings = SystemAPI.GetSingleton<SettingsLoader.SettingsLoader>();

    if (config.Initialized) return;
    config.Initialized = true;
    SystemAPI.SetSingleton(config);

    for (var i = 0; i < settings.numSpheres; i++)
    {
        var position = new float3(Random.Range(-settings.spawnRadius,
        ↪  settings.spawnRadius),
            Random.Range(-settings.spawnRadius, settings.spawnRadius),
            Random.Range(-settings.spawnRadius, settings.spawnRadius));

        var entity = state.EntityManager.Instantiate(config.SpherePrefab);

        state.EntityManager.SetComponentData(entity, new LocalTransform
        {
            Position = position,
            Rotation = Quaternion.identity,
            Scale = 1f
        });

        // In the first scenario, for example, spheres have a fixed base speed
        ↪  but a random direction
        state.EntityManager.AddComponentData(entity, new SphereMove
        {
            InitialPosition = position,
            Direction = new float3(Random.Range(-1f, 1f), Random.Range(-1f, 1f),
            ↪  Random.Range(-1f, 1f))
        });
    }
}
```

4.1 Scenario 1 Creation: Entity Movement Based on Distance from the Camera

MainCamera Position

The challenge of implementing this scenario in DOTS lies in converting the camera's position into the entity world. The MainCamera object is not an entity but a traditional GameObject, and therefore cannot be handled in Burst-compiled code.

For this reason, in both the Job and Jobless implementations of Scenario 1, the SphereSystem will retrieve the current camera position in a function not compiled with Burst and subsequently invoke a Job or Burst-compiled function using the position converted into 'float3', compatible with ECS.

Codice 4.2: SphereJoblessSystem.cs - Scenario 1

```
[BurstCompile]
public partial class SphereJoblessSystem : SystemBase
{
    // This function is not compiled with Burst because it manages the
    ↪  CameraHandler.currentCameraTransform type
    protected override void OnUpdate()
    {
        var deltaTime = SystemAPI.Time.DeltaTime;

        // The MainCamera position is implicitly converted from Vector3 to
        ↪  float3
        InternalUpdate(deltaTime,
        ↪  CameraHandler.currentCameraTransform.position);
    }

    [BurstCompile]
    private void InternalUpdate(float deltaTime, float3 cameraPosition)
    {
        // Entities.WithNone<SphereJobTag>() [...]
        // Query and update of sphere positions, speeds, and colors
        // [...]
    }
}
```

4.1.1 Parallelization with Job System

If the 'Burst + Jobs' optimization is selected for this scenario, the instantiated spheres will have the 'SphereJobTag' component. This way, the script using the Job System can operate, through queries, only on entities where this optimization

is enabled, while the Jobless script will affect only those without the tag (as shown in the previous code snippet). This paradigm is the same for all scenarios.

In Scenario 1, specifically, an IJobEntity-type Job is used, which iterates in parallel over all entities corresponding to a particular query.

Codice 4.3: SphereSystemJob.cs - Scenario 1

```
[BurstCompile]
public partial struct SphereSystemJob : IJobEntity
{
    public float DeltaTime;
    public float3 CameraPosition;

    // Iteration over all entities possessing LocalTransform, Sphere,
    ↪ URPMaterialPropertyBaseColor, SphereMove, and SphereJobTag components
    [BurstCompile]
    private void Execute(ref LocalTransform localTransform, ref Sphere sphere,
        ref URPMaterialPropertyBaseColor baseColor, in SphereMove sphereMove, in
        ↪ SphereJobTag sphereJobTag)
    {
        // Calculate speed based on distance from the MainCamera
        var speedMultiplier = CalculateSpeedMultiplier(localTransform.Position);

        // Calculate position based on elapsed time since the previous frame and
        ↪ speed
        var delta = sphere.Speed * speedMultiplier * DeltaTime *
        ↪ sphereMove.Direction;
        var newPosition = localTransform.Position + delta;

        // Update position (and reverse course if the sphere strays too far from
        ↪ its initial position)
        if (math.distance(sphereMove.InitialPosition, newPosition) >
        ↪ sphere.Spread)
            sphere.Speed *= -1;
        else
            localTransform = localTransform.Translate(delta);

        // Change color based on speed
        var color = Color.HSVToRGB(speedMultiplier, 1, 1);
        baseColor.Value = new float4(color.r, color.g, color.b, 1);
    }
}
```

Therefore, in the version using the Job System, the SphereSystem will instantiate a SphereSystemJob instead of executing code directly in the system and schedule

it to run in parallel:

Codice 4.4: SphereSystem.cs - Scenario 1

```
public partial class SphereSystem : SystemBase
{
    // For using CameraHandler.currentCameraTransform, this function cannot be
    ↪  compiled with Burst, unlike the instantiated Job
    protected override void OnUpdate()
    {
        // Time elapsed between frames, used to calculate position independently
        ↪  of frame rate
        var deltaTime = SystemAPI.Time.DeltaTime;

        var job = new SphereSystemJob
        {
            DeltaTime = deltaTime,
            CameraPosition = CameraHandler.currentCameraTransform.position
        };

        job.ScheduleParallel();
    }
}
```

4.2 Scenario 2 Creation: Calculating Closest Entity to Each Other Entity

4.2.1 Handling Sphere Links

The unique feature of this scenario is that the Sphere entity now also has a child entity: 'Link.' It will be necessary to transform this entity so that it points to the nearest sphere.

Thus, it is optimal to create two separate systems, the first moving individual Spheres equivalently to what was analyzed in Scenario 1, but with a fixed speed, and another that calculates distances and updates the Link entity.

4.2.2 Implementation with Job System

The Jobless version of this scenario can be implemented similarly to Scenario 1 with some additional precautions; more interesting, however, is its implementation using the Job System. The adopted solution is to sequentially execute two jobs:

- **CalculateClosestSphereJob:** For each Sphere, the nearest sphere is calculated, taking an array of sphere positions as input.
- **UpdateLinkTransformJob:** For each Link, its Transform is updated, taking the position of the nearest sphere, calculated in the previous Job, as input.

Using two separate jobs where the second depends on the first is optimal because they iterate over different entities.

Codice 4.5: LinkSphereSystem.cs - Scenario 2

```
[BurstCompile]
public void OnUpdate(ref SystemState state)
{
    // Query to get all spheres
    var parentQuery = SystemAPI.QueryBuilder().WithAll<LocalTransform,
    ↪  SphereJobTag>().Build();

    var sphereEntities = parentQuery.ToEntityArray(Allocator.TempJob);
    // Populate arrays [...]

    // First job to calculate distances between all spheres. The result will be
    ↪  written to ClosestSphereIndices, which will be used as input for the
    ↪  next Job
    var calculateJob = new CalculateClosestSphereJob
    {
        SpherePositions = spherePositions,
        NumSpheres = numSpheres,
        ClosestSphereIndices = closestSphereIndices
    };

    // The job is of type IJobParallelFor, allowing bulk operations on multiple
    ↪  elements of the input array
    var calculateHandle = calculateJob.Schedule(numSpheres, 64);

    // LocalToWorldLookup allows thread-safe access to the 'Link' child entity
    ↪  given the 'Sphere' entity
    var localToWorldLookup =
    ↪  SystemAPI.GetComponentLookup<LocalToWorld>(isReadOnly: false);
    localToWorldLookup.Update(ref state);

    // Update job dependencies so that UpdateLinkTransformJob starts only after
    ↪  CalculateClosestSphereJob has completed
    JobHandle combinedDependencies =
    ↪  JobHandle.CombineDependencies(calculateHandle, state.Dependency);

    // Second Job that updates the transforms of Link entities based on the
    ↪  nearest sphere calculated earlier
    var updateJob = new UpdateLinkTransformJob
    {
        ClosestSphereIndices = closestSphereIndices,
        SpherePositions = spherePositions,
        LocalToWorldLookup = localToWorldLookup,
        LinkEntities = linkEntities
    };

    JobHandle updateHandle = updateJob.Schedule(numSpheres, 64,
    ↪  combinedDependencies);

    state.Dependency = updateHandle;

    // Discarding native arrays used at the end of both jobs (i.e., when
    ↪  updateJob has completed)
    sphereEntities.Dispose(state.Dependency);
    // [...]
}
```

4.3 Scenario 3 Creation: Using Entities with Rigid Bodies and Collisions

4.3.1 Converting GameObject with Rigid Body

Fortunately, Unity DOTS already includes authoring components that allow conversion of GameObject components for rigid bodies (RigidBody) and collisions (SphereCollision) into corresponding components and systems in the Entity world. For this reason, it is only necessary to add these components, included in the *Unity.Physics* package, to the Sphere prefab:

Figure 4.1: Inspector Sphere prefab with Rigidbody

4.3.2 Parallelization with Job System

The implementation of this scenario using the Job System is interesting because it uses an *ICollisionEventsJob*-type Job, which allows parallel iteration over all collisions occurring in the world and the involved entities. The script must then check the components of the involved entities to determine whether the collision occurred between two spheres or between a sphere and a wall, and then change the color of the spheres accordingly.

Codice 4.6: SphereCollisionJob.cs - Scenario 3

```
[BurstCompile]
public struct SphereCollisionJob : ICollisionEventsJob
{
    [ReadOnly] public ComponentLookup<SphereJobTag> SphereTagLookup;
    [ReadOnly] public ComponentLookup<WallTag> WallTagLookup;

    [BurstCompile]
    // Invoked whenever a collision occurs in the Entity world
    public void Execute(CollisionEvent collisionEvent)
    {
        var entityA = collisionEvent.EntityA;
        var entityB = collisionEvent.EntityB;

        var entityAIsSphere = SphereTagLookup.HasComponent(entityA);
        var entityBIsSphere = SphereTagLookup.HasComponent(entityB);

        bool entityAIsWall = WallTagLookup.HasComponent(entityA);
        bool entityBIsWall = WallTagLookup.HasComponent(entityB);

        // Update colors based on entityAIsWall and entityBIsWall [...]
    }
}
```

4.3.3 Creating the Wall Object

The wall is a single entity with a custom cube-shaped *Mesh collider* but with inward-facing faces to contain the spheres. To achieve this effect, the mesh was created using Blender, an open-source software for creating 3D content, also used for modeling. A simple cube was used as the starting point, and its face normals were inverted. The object was then exported and imported into Unity as a custom mesh.

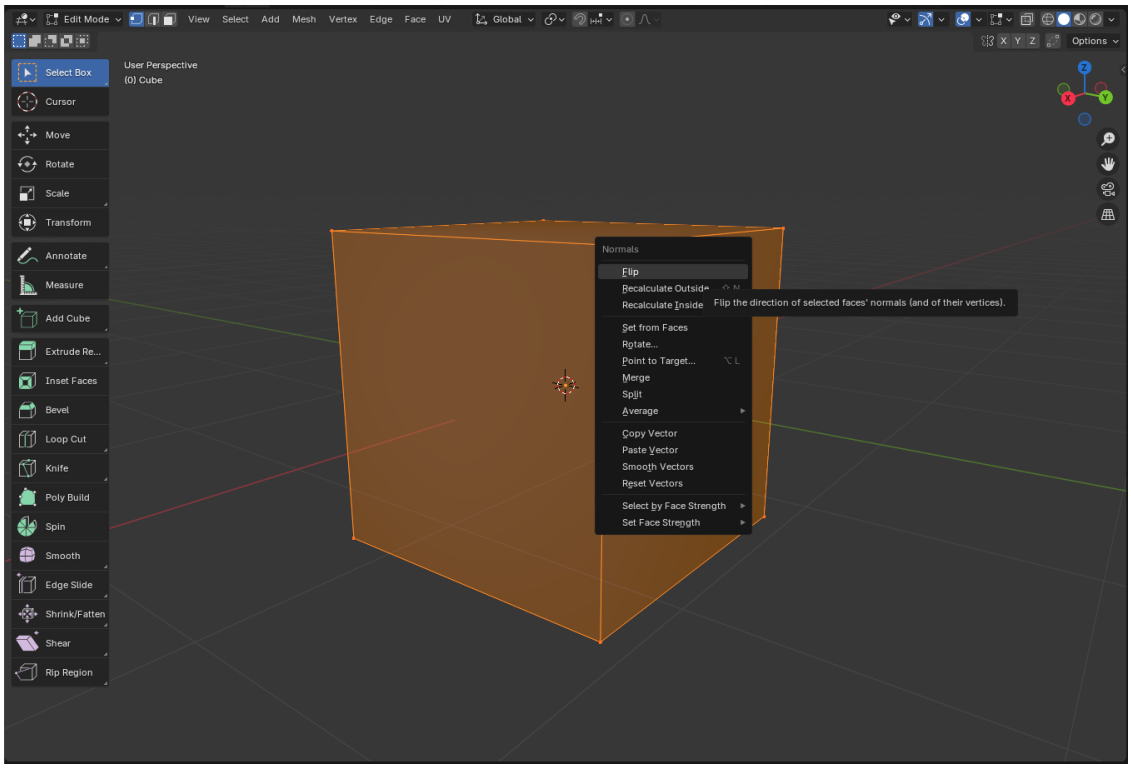


Figure 4.2: Inverting normals in Blender

Additionally, both the wall and the spheres have a custom *Physic Material* added to their Mesh Collider, which completely cancels friction and makes collisions entirely elastic. This ensures that even if the simulation runs for a prolonged period, the spheres do not stop bouncing.

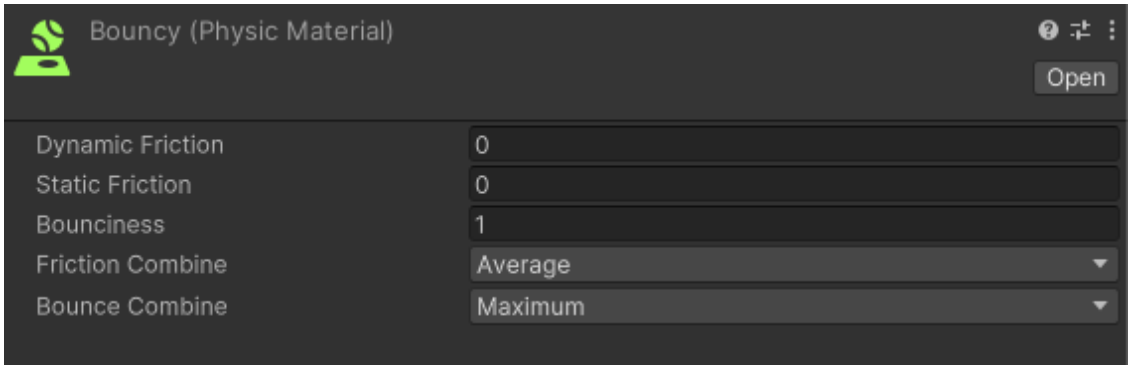


Figure 4.3: Bouncy Physic Material

After describing the challenges and peculiarities of scenario implementations, the analysis of the benchmarking system and its obtained results can follow.

Chapter 5

Benchmarking and Analysis of Results

5.1 Benchmarking Implementation

5.1.1 PerformanceProfiler API

For benchmarking, a custom API class *PerformanceProfiler* was created to calculate the average FPS over a time interval. This is achieved by calculating and storing the time difference between frames within a given interval and returning the average of these measurements at the end of the process:

Codice 5.1: Snippet of PerformanceProfiler.cs

```
public class PerformanceProfiler : MonoBehaviour
{
    // [...]

    private void CollectData()
    {
        var fps = 1.0f / Time.unscaledDeltaTime;
        fpsList.Add(fps);
    }

    public float StopProfiling()
    {
        profilingActive = false;
        return GenerateAggregatedData();
    }

    private float GenerateAggregatedData()
    {
        var fpsSum = fpsList.Sum();
        var averageFps = fpsSum / fpsList.Count;
        return averageFps;
    }
}
```

5.1.2 Automated Scenario Restart

The *ScenarioBenchmarkHandler* MonoBehaviour class, of type Handler, is responsible for restarting the scenario when the benchmarking snapshot duration is reached, updating the scenario settings for the next snapshot, and displaying the results once the benchmarking process is complete (i.e., the number of snapshots has been reached).

Coroutines The handler needs to wait for a fixed number of seconds before stopping the PerformanceProfiler and restarting the scenario. This can be easily achieved using *Coroutines*, a native Unity system that internally uses C# Tasks, allowing asynchronous code to be invoked from MonoBehaviour scripts.

A coroutine also allows tasks to be distributed across multiple frames by suspending execution and returning control to the system, resuming from the point where it left off in the next frame [19].

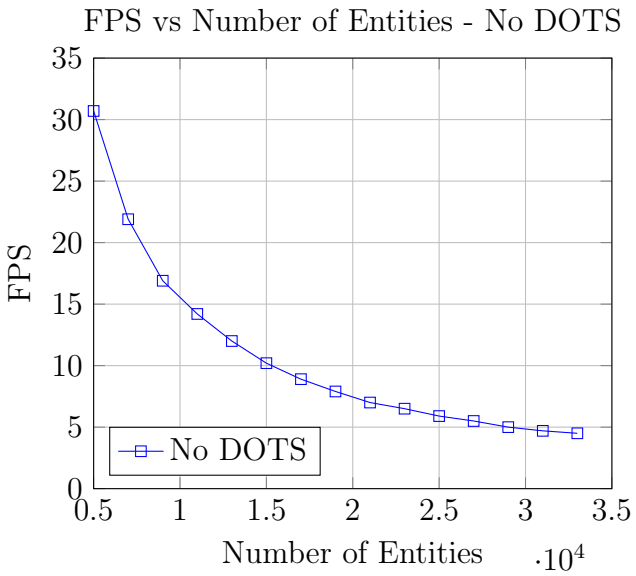
Codice 5.2: Snippet of ScenarioBenchmarkHandler.cs

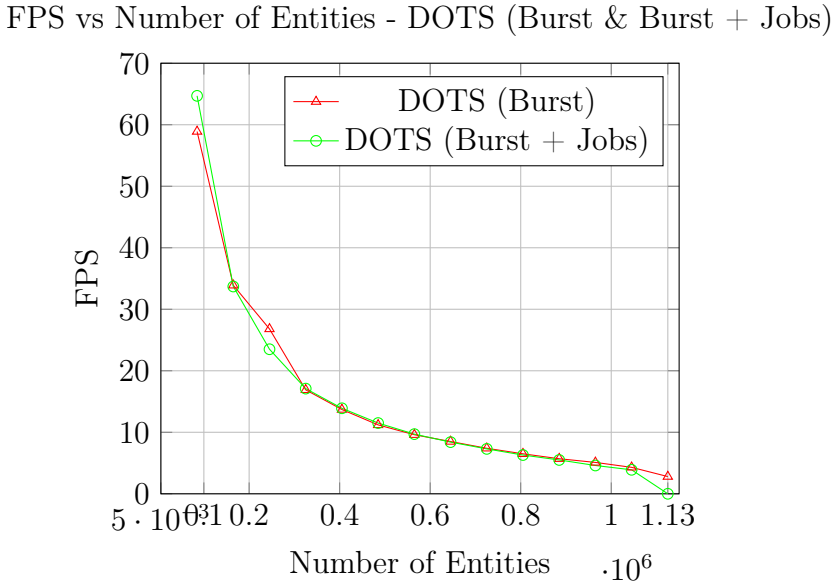
```
public class ScenarioBenchmarkHandler : MonoBehaviour
{
    private void Start()
    {
        // Init [...]

        // Ends benchmarking if the number of snapshots is reached
        if (AverageFPS.Count >= _settings.benchmarkNumSnapshots)
        {
            ShowResults();
            ResetProgress();
            return;
        }

        // Invokes the Benchmark coroutine
        StartCoroutine(Benchmark());
    }

    private IEnumerator Benchmark()
    {
        _isBenchmarking = true;

        // Starts profiling using the PerformanceProfiler API
        PerformanceProfiler.Instance.StartProfiling();

        // Suspends the coroutine for the duration of the benchmark
        yield return new
        ↪  WaitForSecondsRealtime(_settings.benchmarkSnapshotDuration);

        // Ends profiling and adds the result to the AverageFPS list
        AverageFPS.Add(PerformanceProfiler.Instance.StopProfiling());

        // If the number of snapshots has not been reached, increases the number
        ↪  of entities
        if (AverageFPS.Count >= _settings.benchmarkNumSnapshots)
            ScenarioSettingsAPIs.SetNumEntities(0);
        else
            ScenarioSettingsAPIs.SetNumEntities(
                ScenarioSettingsAPIs.GetNumEntities() +
                ↪  _settings.benchmarkIncrement);

        // Reloads the scene
        SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex);
    }
}
```

5.2 Benchmarking Results

The benchmarking was conducted as a proof of concept on a MacBook Pro with an M3 Pro chip, 36GB memory, and macOS Sonoma (14.6.1). The results for the three scenarios are shown below.
The graphs are separated into NoDOTS, DOTS, and DOTS + Jobs configurations to improve readability, as the NoDOTS configuration handles far fewer GameObjects compared to the other configurations.

5.2.1 Scenario 1

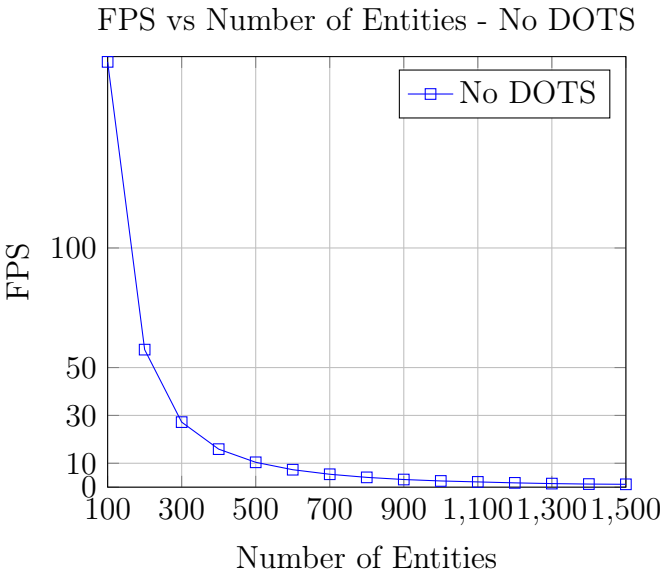


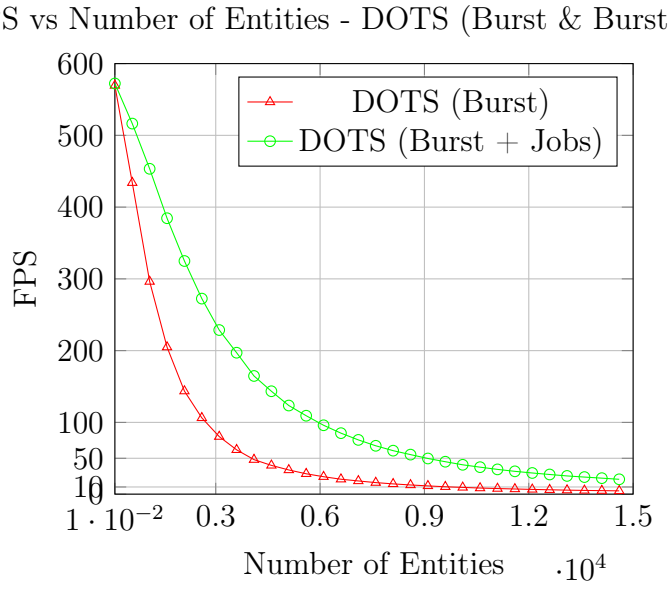


From the collected data, it can be observed that using traditional programming, the system can handle a maximum of about 33,000 GameObjects before the frame rate drops below 5 FPS, making the project unusable.
With Burst optimization and Burst + Jobs, the system is capable of handling a significantly larger number of entities, up to 1,125,000. The frame rate drops below 5 FPS at around 885,000 entities.
Considering the smoothness threshold of 25 FPS, it is noted that without DOTS, this limit is reached with approximately 6,500 entities. In contrast, with other optimizations, smoothness is maintained above 25 FPS up to approximately 245,000 entities.
Finally, for this particular scenario, parallelization with the Job System does not yield significant improvements compared to using DOTS alone. It is likely that what causes latency is the rendering of such a massive number of entities, rather than the distance calculation performed by each entity per frame. The only difference between the two optimizations is that the calculation is performed in parallel in the Job System optimization, which is why the expected improvement is not observed for this particular scenario.

5.2.2 Scenario 2

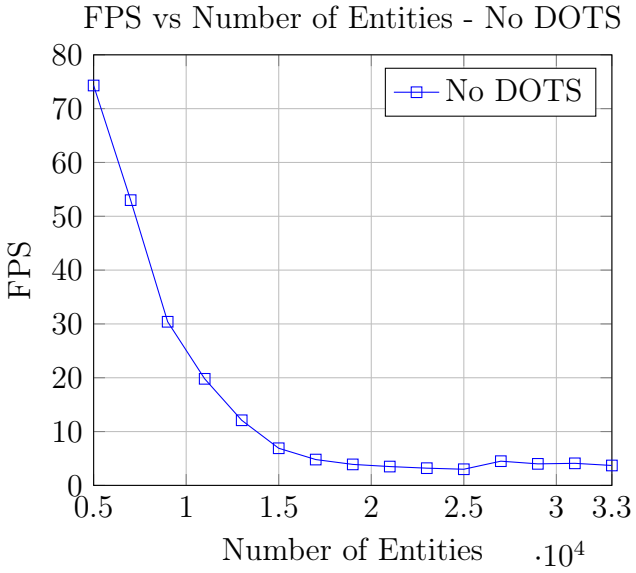
It should be noted that the benchmarking process reports the number of spheres present in the scene. In this particular scenario, each sphere also has an additional 'Link' entity. Therefore, the total number of entities to render is double the reported number.

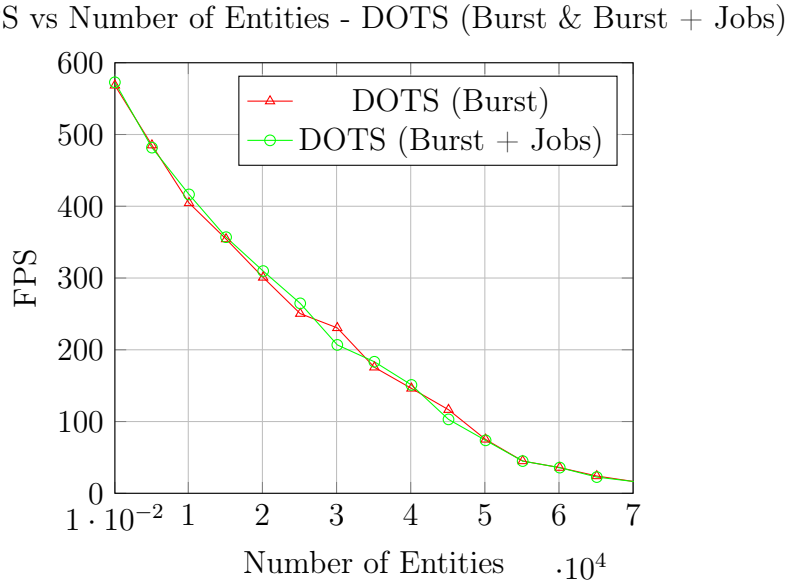




From the collected data, it can be observed that without DOTS, the exponential increase in calculations performed by each sphere causes a drastic drop in performance as the number of entities grows. The system can handle only up to about 200 spheres before the frame rate drops below the 25 FPS threshold, making the project no longer smooth. Beyond 500 spheres, the frame rate drops rapidly below 10 FPS, rendering the simulation unusable.
With DOTS optimizations, the system can handle a significantly larger number of entities. Performance decreases exponentially as entities increase, as expected, since each GameObject performs O($numEntities^2$) operations. The frame rate drops below 25 FPS at around 6,600 entities in the case of the Burst Compiler alone and at around 13,100 entities with the combined use of Jobs and the Burst Compiler. The 5 FPS limit is reached at about 14,600 entities for both optimizations.
In this scenario, optimization with Jobs is much more effective, as the computational complexity (due to distance calculations) can be divided across multiple threads, significantly improving performance and allowing smoothness to be maintained with a much larger number of entities compared to the Burst-only configuration.

5.2.3 Scenario 3





From the collected data, it can be observed that in the configuration without DOTS, performance progressively decreases as the number of spheres in the physical simulation increases. The system can handle only up to about 9,000 spheres before the frame rate drops below the 25 FPS threshold. Beyond this number, the frame rate rapidly decreases, rendering the simulation unusable with more than 15,000 GameObjects, where the frame rate is below 10 FPS.
In the optimization with DOTS and the optimization with DOTS + Jobs, much more efficient management of physical objects is observed: the 25 FPS threshold is reached at approximately 65,100 entities.

The 5 FPS limit is reached at around 70,100 entities in both the Burst Compiler-only case and with the addition of Jobs. This suggests that, in this scenario, optimization with Jobs does not bring significant improvements compared to the Burst Compiler alone. This is likely because the physical workload related to managing collisions and physical dynamics is limited by the serial computational capacity of physical operations rather than their distribution across threads.
However, DOTS optimizations remain extremely effective compared to the non-DOTS configuration, allowing a much larger number of physical entities to be simulated without compromising the simulation's smoothness.

Chapter 6

Conclusion and Future Developments

6.1 Summary of Analyzed Results

The following table summarizes the data obtained and analyzed in the previous chapter:

Scenario	Optimization	Entities at 25 FPS	Entities at 5 FPS
Scenario 1	No DOTS	6,500	33,000
	Burst	245,000 (+3,669%)	885,000 (+2,582%)
	Burst + Jobs	245,000 (+3,669%)	885,000 (+2,582%)
Scenario 2	No DOTS	200	500
	Burst	6,600 (+3,200%)	14,600 (+2,820%)
	Burst + Jobs	13,100 (+6,450%)	14,600 (+2,820%)
Scenario 3	No DOTS	11,000	17,000
	Burst	65,100 (+491%)	70,100 (+312%)
	Burst + Jobs	65,100 (+491%)	70,100 (+312%)

Table 6.1: Number of entities managed at 25 FPS and at 5 FPS for the three scenarios with different optimizations, along with the percentage improvement over No DOTS next to each value.

Overall, the use of DOTS libraries proves significantly more efficient when creating complex scenes with a large number of entities. The Job System, in particular, greatly enhances performance when the individual entity scripts require intensive computations and these are easily parallelizable, i.e., independent of each other.

6.2 Challenges of Unity DOTS

Despite the evident performance improvements, after developing the benchmarking project, some issues were identified that would not have occurred if MonoBehaviour had been used:

- **Incompatibility with the Unity Editor:** The root of all challenges lies in the fact that the Unity Engine was developed and extended using OOP. As noted earlier, many editor features are incompatible with DOTS. Currently, the only way to mitigate these incompatibilities is by adding traditional GameObjects to the scene that follow the positions of their respective ECS entities but can exhibit MonoBehaviour features (like the Animation system). In the analyzed scenarios, this would have limited performance because the primary bottleneck would have been the number of GameObjects in the scene.
- **Boilerplate Authoring Code:** Each entity component placed in the scene using the Unity Editor (not instantiated only at runtime) requires a corresponding authoring script. In most cases, these scripts mirror the properties of the component, increasing coding time and the number of classes. This issue can be easily addressed using IDE features that can auto-generate MonoBehaviour authoring scripts for the required component.
- **Incomplete Collision System:** As of the latest Unity DOTS version at the time of this writing, the Physics Shape component in Unity Physics does not allow setting the collision detection mode (Discrete or Continuous). Discrete detection updates object collisions per physics frame and might lead to unexpected results, such as fast-moving objects passing through thin walls. Continuous detection, while more resource-intensive, checks all points between the object's position and its position in the previous physics frame to find a collision. Possible solutions to this issue will be discussed in the next section [20].

Overall, Unity DOTS's challenges are also due to its recent release. These challenges are acceptable even in production development environments if these libraries are used to optimize specific computationally demanding parts of the application system rather than the entire project.

6.3 Expanding Research and Future Developments

In conclusion, this study and project could be expanded in the future with the analysis of additional Unity DOTS libraries and potentially new benchmarking scenarios:

- A detailed analysis of **Entities Graphics**, the library that enables optimized rendering of a large number of entities on-screen, could be conducted. Customizing the rendering pipeline to directly intervene in the data flow to the GPU could further optimize rendering based on the specific project needs.
- The **Netcode for Entities** library allows the transmission and synchronization of entities within the ECS world. This is useful for multiplayer games or distributed simulations. The multiplayer service is not peer-to-peer but features an authoritative server. The library enables clients to "predict" the future state of the game, reducing perceived network latency, while the server corrects any discrepancies with the actual state in the background [3].

Additionally, Scenario 3 uses the Unity Physics library for collision handling, which, as noted earlier, has its challenges. ECS code could be rewritten to use systems that perform *raycasting* from various entities to detect collisions. This method could be more easily parallelizable, potentially improving physical simulation accuracy and performance when leveraging the Job System.

Finally, at the time of this thesis, Unity 6 has not yet been released. However, its release date was announced during the *Unite 2024 Keynote*, scheduled for October 17, 2024. Since this version will bring substantial improvements to Unity DOTS libraries (including better compatibility with the Unity Editor and general performance enhancements), updating the project to the latest version and re-benchmarking the analyzed scenarios would be worthwhile [21].

Bibliography

[1] Kölling, M. (1999). *The problem of teaching object-oriented programming, Part I.* Journal of Object-Oriented Programming, 11(8), 8–15.

[2] White, G., & Sivitanides, M. (2005). *Cognitive differences between procedural programming and object-oriented programming.* Information Technology, Learning, and Performance Journal, 23(1), 25–33.

[3] Unity Technologies. (n.d.). *Unity's Data-Oriented Technology Stack (DOTS) for advanced developers*: https://unity.com/dots

[4] Fedoseev, K., Askarbekuly, N., Uzbekova, E., & Mazzara, M. (2020). *A case study on object-oriented and data-oriented design paradigms in game development.* In *Proceedings of the 2020 International Conference on Games and Learning Alliance (GALA)*, 35-44. Springer, Cham. DOI:10.1007/978-3-030-62364-3_4

[5] code::dive conference 2014 - Scott Meyers *Cpu Caches and Why You Care.*

[6] https://docs.unity3d.com/Manual/GameObjects.html *Unity GameObjects Manual.*

[7] https://unity.com/dots *Unity DOTS.*

[8] https://blog.jetbrains.com/dotnet/2023/03/16/unity-dots-support-in-rider-2023-1/ *Unity DOTS support in Rider 2023.*

[9] https://unity.com/roadmap/unity-platform/dots *Unity DOTS roadmap.*

[10] https://docs.unity3d.com/Manual/JobSystemOverview.html *Unity Job System manual.*

[11] https://learn.microsoft.com/en-us/dotnet/framework/interop/blittable-and-non-blittable-types *Dotnet Blittable Types reference.*

[12] https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection/ *Dotnet Garbage Collection reference.*

[13] https://docs.unity3d.com/Packages/com.unity.collections@2.4/manual/allocator-overview.html *Unity Collections Allocators manual.*

[14] https://docs.unity3d.com/Manual/IL2CPP.html *Unity IL2CPP manual.*

[15] https://www.youtube.com/watch?v=WnJV6J-taIM *Using Burst Compiler to optimize for Android | Unite Now 2020*

[16] https://docs.unity3d.com/Packages/com.unity.burst@1.8/manual/optimization-overview.html *Unity Burst Optimization manual*

[17] Unite LA *ECS Track: Deep Dive into the Burst Compiler*
[18] https://learnopengl.com/Guest-Articles/2021/Scene/Frustum-Culling *earnOpenGL Frustum Culling*
[19] https://docs.unity3d.com/Manual/Coroutines.html *Unity Coroutines Manual*
[20] https://docs.unity3d.com/Manual/ContinuousCollisionDetection.html *Unity Continuous collision detection Manual*
[21] https://unity.com/blog/unite-2024-keynote-wrap-up *Unite 2024 Keynote*