



## POLITECNICO DI BARI

DIPARTIMENTO DI INGEGNERIA ELETTRICA E DELL'INFORMAZIONE  
**Corso di Laurea in Ingegneria Informatica e dell'Automazione**

---

Tesi di Laurea in Ingegneria del Software

# Approccio Data-Oriented nei Motori di Gioco: Benchmarking e Sperimentazione con Unity DOTS per la Creazione di Scene ad Alta Intensità di Risorse

Relatrice  
**Prof.ssa Marina Mongiello**

Laureando  
**Domenico Rotolo**

---

Anno Accademico 2023 - 2024



# Abstract

Nelle moderne applicazioni grafiche ad alte prestazioni, come le simulazioni e i videogiochi, la frequenza di aggiornamento dello schermo e la gestione della latenza giocano un ruolo cruciale. Ogni frame deve essere generato e visualizzato in tempi estremamente ridotti, senza ritardi o interruzioni, per garantire un'esperienza fluida all'utente. In questo contesto, le scelte di paradigma e di architettura software possono fare una notevole differenza, influenzando direttamente la gestione della memoria e l'efficienza del codice.

Questa tesi si pone l'obiettivo di esplorare il paradigma di programmazione orientato ai dati (Data-Oriented Design, DOD) come alternativa significativa al tradizionale approccio orientato agli oggetti (Object-Oriented Programming, OOP). Viene condotta un'analisi delle caratteristiche fondamentali dei due paradigmi, sottolineando le problematiche legate ai costi performativi nell'OOP, specialmente in contesti dove l'accesso ottimizzato alla memoria è prioritario. Al contrario, il DOD si concentra sulla struttura e sull'accesso sequenziale dei dati, consentendo un uso più efficiente della cache e un accesso alla memoria più rapido e prevedibile.

Il cuore della trattazione è dedicato all'analisi di Unity DOTS (Data-Oriented Technology Stack), un insieme di librerie sviluppate da Unity Technologies con l'obiettivo di implementare il paradigma DOD all'interno del motore di gioco Unity. Unity DOTS include tre componenti principali: il C# Job System, il Burst Compiler e l'Entity Component System (ECS). Ogni componente è analizzato in dettaglio per illustrarne il contributo alla performance complessiva: il C# Job System consente la gestione e l'esecuzione parallela dei task, riducendo il carico computazionale complessivo. Il Burst Compiler, con il suo compilatore ad alte prestazioni, permette di ottimizzare le operazioni di basso livello per sfruttare al massimo l'hardware sottostante. Infine, l'ECS permette la scrittura di entità usando la filosofia del paradigma DOD.

Successivamente, la tesi descrive la progettazione e l'implementazione di un sistema applicativo basato su Unity DOTS, finalizzato a confrontare il paradigma OOP, implementato con classi MonoBehaviour, con quello DOD in termini di prestazioni. Il sistema applicativo è stato progettato per includere vari scenari di benchmarking, ognuno dei quali rappresenta una situazione tipica nei videogiochi e nelle simulazioni, come la gestione di numerosi oggetti in movimento o il calcolo

simultaneo di complessi stati di gioco e collisioni.

Infine, viene effettuata un'analisi dei risultati del benchmarking, i quali dimostrano come l'uso del DOD e di Unity DOTS possa portare a miglioramenti significativi in termini di performance.

## Struttura della Tesi

- **Confronto tra OOP e DOD:** descrizione dettagliata dei costi di frammentazione della memoria e delle problematiche di accesso ai dati nell'OOP e analisi del DOD e dei suoi vantaggi in termini di prestazioni.
- **Unity DOTS:** approfondimento tecnico sui componenti principali di DOTS (C# Job System, Burst Compiler, ECS), con dettagli su architettura, funzionalità, ottimizzazioni supportate e roadmap dello sviluppo.
- **Implementazione del sistema applicativo:** descrizione del sistema applicativo sviluppato per il confronto OOP-DOD, metodologia sperimentale e specifiche degli scenari di benchmark.
- **Risultati e analisi:** presentazione dei risultati ottenuti, analisi delle differenze prestazionali e delle implicazioni per lo sviluppo di software ad alte prestazioni.
- **Conclusioni:** sintesi dei risultati e considerazioni finali sull'utilizzo del DOD e di Unity DOTS nel miglioramento delle prestazioni, con proposte per future aree di ricerca e sperimentazione.

# Indice

<b>1 Confronto tra Programmazione Orientata agli Oggetti e Programmazione Orientata ai Dati</b>	1
1.1 Object-oriented programming . . . . .	1
1.1.1 Paradigmi dell'OOP . . . . .	1
1.2 Costi performativi dell'OOP . . . . .	2
1.3 Data-oriented design . . . . .	3
1.3.1 Composizione al posto dell'Ereditarietà . . . . .	3
1.3.2 Architettare i Dati prima di Architettare il Codice . . . . .	4
1.4 Ottimizzazione dell'Accesso in Memoria . . . . .	5
1.4.1 Prefetching . . . . .	5
1.4.2 False Sharing . . . . .	9
<b>2 Programmazione Orientata ai Dati Applicata: Unity DOTS</b>	13
2.1 Unity DOTS: definizioni e roadmap . . . . .	13
2.1.1 Unity DOTS (Data-Oriented Technology Stack) . . . . .	13
2.1.2 Storia e roadmap dello sviluppo di Unity DOTS . . . . .	14
2.2 Struttura di Unity DOTS . . . . .	14
2.2.1 C# Job System . . . . .	14
2.2.2 Gestione di tipi blittable: il pacchetto Collections . . . . .	20
2.2.3 Burst Compiler . . . . .	21
2.2.4 Entity Component System . . . . .	26
2.2.5 Conversione tra GameObjects ed Entità . . . . .	30
<b>3 Implementazione e Sperimentazione con Unity DOTS</b>	33
3.1 Informazioni Generali . . . . .	33
3.1.1 Scopo del Progetto . . . . .	33
3.1.2 Struttura del Progetto . . . . .	33
3.1.3 Modalità di Ottimizzazione . . . . .	34
3.1.4 Descrizione dell'UI . . . . .	34
3.1.5 Descrizione degli scenari . . . . .	37
3.2 Strumenti e sfide dello sviluppo . . . . .	40
3.2.1 Architettura MVC per UI . . . . .	40

3.2.2	Comunicazione tra UI in Monobehaviour ed Entity World . . . . .	40
3.2.3	Uso di Unity Profiler per evidenziare i colli di bottiglia . . . . .	43
3.2.4	Disabilitare il Frustum Culling . . . . .	44
<b>4</b>	<b>Creazione degli Scenari</b>	<b>47</b>
4.1	Creazione Scenario 1: Movimento di Entità in base alla Distanza dalla Videocamera . . . . .	49
4.1.1	Parallelizzazione con Job System . . . . .	49
4.2	Creazione Scenario 2: Calcolo Entità più Vicina a ogni altra Entità	51
4.2.1	Gestione collegamenti tra sfere . . . . .	51
4.2.2	Implementazione con Job System . . . . .	51
4.3	Creazione Scenario 3: Utilizzo di Entità con Corpo Rigido e Collisioni	54
4.3.1	Conversione GameObject con corpo rigido . . . . .	54
4.3.2	Parallelizzazione con Job System . . . . .	55
4.3.3	Creazione dell'oggetto Muro . . . . .	55
<b>5</b>	<b>Benchmarking e Analisi dei Risultati</b>	<b>57</b>
5.1	Implementazione del Benchmarking . . . . .	57
5.1.1	PerformanceProfiler API . . . . .	57
5.1.2	Riavvio automatizzato dello scenario . . . . .	58
5.2	Risultati del Benchmarking . . . . .	60
5.2.1	Scenario 1 . . . . .	60
5.2.2	Scenario 2 . . . . .	61
5.2.3	Scenario 3 . . . . .	63
<b>6</b>	<b>Conclusione e Sviluppi Futuri</b>	<b>65</b>
6.1	Sintesi dei Risultati Analizzati . . . . .	65
6.2	Criticità di Unity DOTS . . . . .	65
6.3	Ampliamento Ricerca e Sviluppi Futuri . . . . .	66
<b>Bibliografia</b>		<b>69</b>

# Capitolo 1

## Confronto tra Programmazione Orientata agli Oggetti e Programmazione Orientata ai Dati

### 1.1 Object-oriented programming

La programmazione orientata agli oggetti (OOP) è stata storicamente il paradigma di programmazione più influente. È ampiamente usata nell'istruzione e nell'industria, e quasi ogni Università insegna l'orientamento agli oggetti da qualche parte nel suo curriculum [1].

La strategia per risolvere i problemi con OOP è quella di dividerli in problemi più piccoli. Il paradigma può quindi essere suddiviso in tre sotto-paradigmi [2]:

#### 1.1.1 Paradigmi dell'OOP

##### Paradigma della struttura del programma

Le classi e gli oggetti rappresentano i blocchi fondamentali di un programma. Le classi definiscono il modello per gli oggetti, racchiudendo al loro interno i dati, rappresentati dagli attributi e i comportamenti, rappresentati dai metodi. L'ereditarietà, il polimorfismo e l'incapsulamento sono principi chiave in questo contesto.

##### Paradigma della struttura dello stato

Gli oggetti possiedono stati interni, rappresentati dai loro attributi, che possono cambiare quando l'oggetto interagisce con altri oggetti o processi. In questo ambito, l'incapsulamento è particolarmente importante, poiché consente agli oggetti di

nascondere i loro stati interni, rivelando solo ciò che è necessario tramite i metodi. L'idea di uno stato che può influenzare il comportamento di un oggetto è centrale in questo sotto-paradigma.

### **Paradigma della computazione**

Questo sotto-paradigma si focalizza sul passaggio di messaggi tra oggetti, un processo mediante il quale gli oggetti comunicano tra loro per eseguire compiti e calcoli. I metodi sono il mezzo attraverso cui avviene la computazione all'interno degli oggetti; l'interazione tra oggetti, realizzata attraverso le chiamate ai metodi, è ciò che guida l'esecuzione del programma. In questo contesto, il polimorfismo e il legame dinamico dei metodi sono fondamentali, poiché consentono a oggetti diversi di rispondere allo stesso messaggio in modi che siano contestualmente appropriati.

## **1.2 Costi performativi dell'OOP**

Tra i benefici teorici dell'OOP c'è prima di tutto la riconfigurabilità e il riutilizzo del codice: attraverso la gestione degli oggetti è facilitata la rimozione e la modifica delle funzionalità del programma e il riutilizzo degli oggetti stessi per consentire una maggiore coerenza e una riduzione degli errori. Inoltre, la programmazione orientata agli oggetti stimola l'architettura delle classi in modo da riflettere il mondo reale. Ciò permette al codice di essere più comprensibile e naturale da gestire. Un altro vantaggio è l'astrazione, che consente ai programmatore di affrontare i problemi a un livello più alto, senza doversi distrarre con i dettagli delle implementazioni a basso livello.

Detto ciò, la natura stessa dell'OOP presenta dei costi performativi, che, come vedremo nel capitolo successivo, sono particolarmente evidenti nello sviluppo di programmi che sono altamente reattivi, come i videogiochi. Innanzitutto, la preferenza per funzioni di piccole dimensioni e i numerosi livelli di astrazione complicano il flusso di esecuzione, rendendo difficile seguire e ottimizzare il percorso delle chiamate. Inoltre, poiché il codice che manipola direttamente un oggetto è parte integrante dell'oggetto stesso, esiste una tendenza naturale a trattare gli oggetti individualmente, piuttosto che processarli in blocchi più ampi, riducendo così le opportunità di ottimizzazione come, ad esempio, la parallelizzazione. Un altro impatto alle performance è dato dal fatto che la disposizione dei dati in memoria tende a essere frammentata: il codice OOP è spesso suddiviso in molti piccoli oggetti e questo porta i dati a essere sparsi in vari punti della memoria; per questo i pattern di allocazione nell'OOP tendono a essere poco efficienti e il codice spesso si affida a frequenti e piccole allocazioni di memoria e alla garbage collection (ciò sarà descritto nel dettaglio nel paragrafo 'Ottimizzazione dell'Accesso in Memoria') [3].

## 1.3 Data-oriented design

La programmazione orientata ai dati (DOD) è un paradigma moderno che si concentra su come i dati sono immagazzinati in memoria, piuttosto che sulle relazioni tra i dati stessi, per renderli così più facilmente trasformabili. L'obiettivo è quello di facilitare la programmazione di codice con una disposizione della memoria più efficiente, memorizzando i dati utilizzati frequentemente più vicini tra loro nella memoria. DOD, a differenza di OOP, cerca per questo motivo anche di separare i dati dalla logica [4].

### 1.3.1 Composizione al posto dell'Ereditarietà

DOD favorisce la composizione rispetto all'ereditarietà. Ciò è fondamentale, poiché la composizione rende più semplice separare i dati dalla logica.

**Ereditarietà** L'ereditarietà è un meccanismo in cui una classe (chiamata sottoclasse o classe figlia) eredita proprietà e comportamenti (metodi) da un'altra classe (chiamata superclasse o classe genitore). Ciò può portare a un codice strettamente collegato, dove le sottoclassi dipendono fortemente dalle superclassi, rendendo difficile apportare modifiche in futuro e complicando la struttura se la gerarchia diventa troppo complessa.

**Composizione** La composizione, invece, prevede la creazione di oggetti che sono composti da altri oggetti, piuttosto che dipendere dall'ereditarietà. Questo approccio permette maggiore flessibilità, perché gli oggetti possono essere combinati in modi diversi senza essere vincolati da una struttura di ereditarietà rigida.

Codice 1.1: Esempio Ereditarietà

---

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

class Shark : Fish
{
    // Shark eredita il metodo Swim dalla classe Fish
}
```

---

Codice 1.2: Esempio Composizione

---

```
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 Architettare i Dati prima di Architettare il Codice

La premessa centrale del Design Orientato ai Dati (DOD) è che i dati sono almeno importanti quanto il codice. Sia a livello macro che micro, i programmi riguardano essenzialmente la trasformazione e la produzione di dati. Il codice è visto come una catena di montaggio di dati: in un videogioco, ad esempio, il codice, a ogni tick, riceve l'input dell'utente e lo stato del gioco corrente e li trasforma in un nuovo stato per il rendering del frame successivo.

Questa definizione della programmazione può sembrare troppo semplificata e scontata, ma offre chiarezza sugli obiettivi del DOD. Avere un punto di partenza e di arrivo dei dati ben definito permette di costruire algoritmi che manipolano indipendentemente tra loro i vari stati di un dato e ciò consente al programmatore di ottimizzare più facilmente le singole trasformazioni e di individuare i colli di bottiglia. [3]

**Opportunità di ottimizzazione** Le seguenti opportunità di ottimizzazione, ad esempio, possono essere più facilmente scoperte con un modello a ‘catena di

montaggio’ (pipeline model):

- Necessità di trasformare dati grezzi in steps intermedi per offrire un’elaborazione più efficiente negli step successivi.
- Al contrario, necessità di ridurre gli steps di elaborazione di un dato.
- Gli stessi dati prodotti da più steps diversi possono essere salvati in un unico step precedente.
- Alcuni elementi che vengono processati uno per uno possono invece essere processati in massa per garantire un accesso alla memoria più efficiente.

L’accesso alla memoria è, per gli obiettivi di questo lavoro, uno dei vantaggi più influenti del DOD. Come anticipato in precedenza, quindi, lo analizziamo adesso più nel dettaglio.

## 1.4 Ottimizzazione dell’Accesso in Memoria

La lettura di un dato in memoria avviene attraverso la lettura della cache: quando la CPU esegue un’istruzione che necessita di leggere un indirizzo della memoria di sistema, l’hardware verifica se una copia dei dati a quell’indirizzo è presente nella prima cache. Se non lo è, l’hardware controlla la cache successiva, fino a leggere direttamente dalla memoria del sistema se il dato non è presente in cache (in questo caso si parla di ‘cache miss’). La maggior parte delle CPU moderne dispone di 3 livelli di cache, le cui dimensioni variano insieme al loro tempo di accesso.

### 1.4.1 Prefetching

Le prestazioni di un programma dipendono, quindi, anche dalla capacità di minimizzare le ‘cache misses’ per garantire un accesso più veloce ai dati richiesti per l’elaborazione. Consideriamo, ad esempio, i seguenti algoritmi:

Codice 1.3: Esempio Somma Matrici

---

```
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;
    }
}
```

---

Entrambe le funzioni restituiscono la somma di tutti gli elementi di una matrice: la prima itera sulle righe e la seconda itera sulle colonne. Utilizzando il seguente codice, possiamo confrontare i tempi di esecuzione dei due algoritmi per una matrice di dimensione qualsiasi:

Codice 1.4: Esempio Somma Matrici 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($"Righe: {size}, {stopwatch.ElapsedMilliseconds} ms");

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

---

Possiamo, quindi, visualizzare il tempo di esecuzione in ms al variare della dimensione della matrice nel seguente grafico:

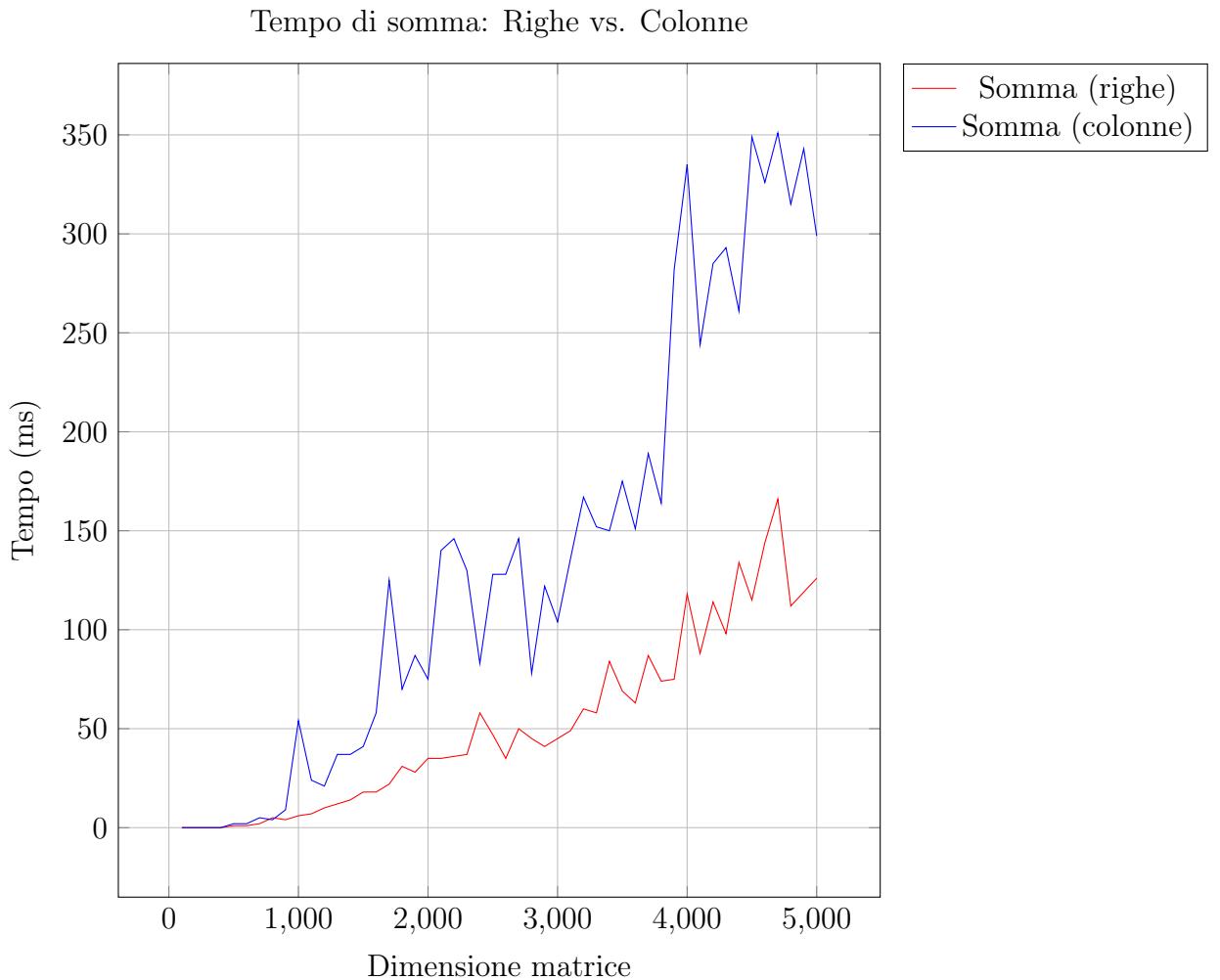


Figura 1.1: Tempo di esecuzione per la somma di matrici iterando sulle righe e iterando sulle colonne

La drastica differenza di prestazioni al variare del parametro ‘size’ dipende dalla composizione della cache; essa è, infatti, composta da ‘cache lines’ e, quando si legge un singolo byte di cache, in realtà si sta accedendo all’intera cache line. Dato che gli elementi di una matrice sono immagazzinati in memoria riga per riga, accedervi allo stesso modo sarà più prestante, poiché si accede sequenzialmente alla stessa linea di cache e quindi si minimizzano le ‘cache misses’.

L’impatto positivo sulle prestazioni nell’accedere a dati in memoria in maniera sequenziale è rafforzato dal ‘prefetching’: l’hardware, in maniera speculativa, mette in cache la cache line ‘ $n+1$ ’, se si sta facendo un ‘forward traversal’ della cache line ‘ $n$ '; allo stesso modo, mette in cache la cache line ‘ $n-1$ ’ in un ‘reverse traversal’ della cache line ‘ $n$ ’.

**Cache Coherency** La cache a livello hardware si assicura che i valori immagazzinati siano sempre coerenti con quelli in memoria, impostando a 'dirty' la corrispondente cache line e, quindi, invalidandola quando un dato è scritto in memoria.

### 1.4.2 False Sharing

Consideriamo i seguenti algoritmi, nei quali stiamo contando il numero di elementi pari di una matrice, suddividendo il lavoro in un numero 'threadCount' di threads:

Codice 1.5: Esempio 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 questo primo algoritmo, appena si riscontra un numero pari, viene incrementato un elemento di un vettore globale all'indice corrispondente all'indice del thread inizializzato (evidenziato in arancio). Dopo che tutti i threads hanno concluso l'esecuzione, si possono sommare i singoli elementi del vettore globale ‘counters’ per ottenere il risultato corretto.

Codice 1.6: Esempio Senza 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();
    }
}
```

---

Nel secondo algoritmo si eseguono le stesse operazioni, ma anziché incrementare il valore dell'elemento corrispondente al thread del vettore ad ogni nuovo numero

pari trovato, viene definita una variabile globale ‘localCounter’ (evidenziata in verde) e, alla fine dell’iterazione, il suo valore è assegnato all’elemento del vettore (evidenziato in arancio).

Questa modifica porta a un impatto considerevole sull’efficienza con l’aumentare dei threads:

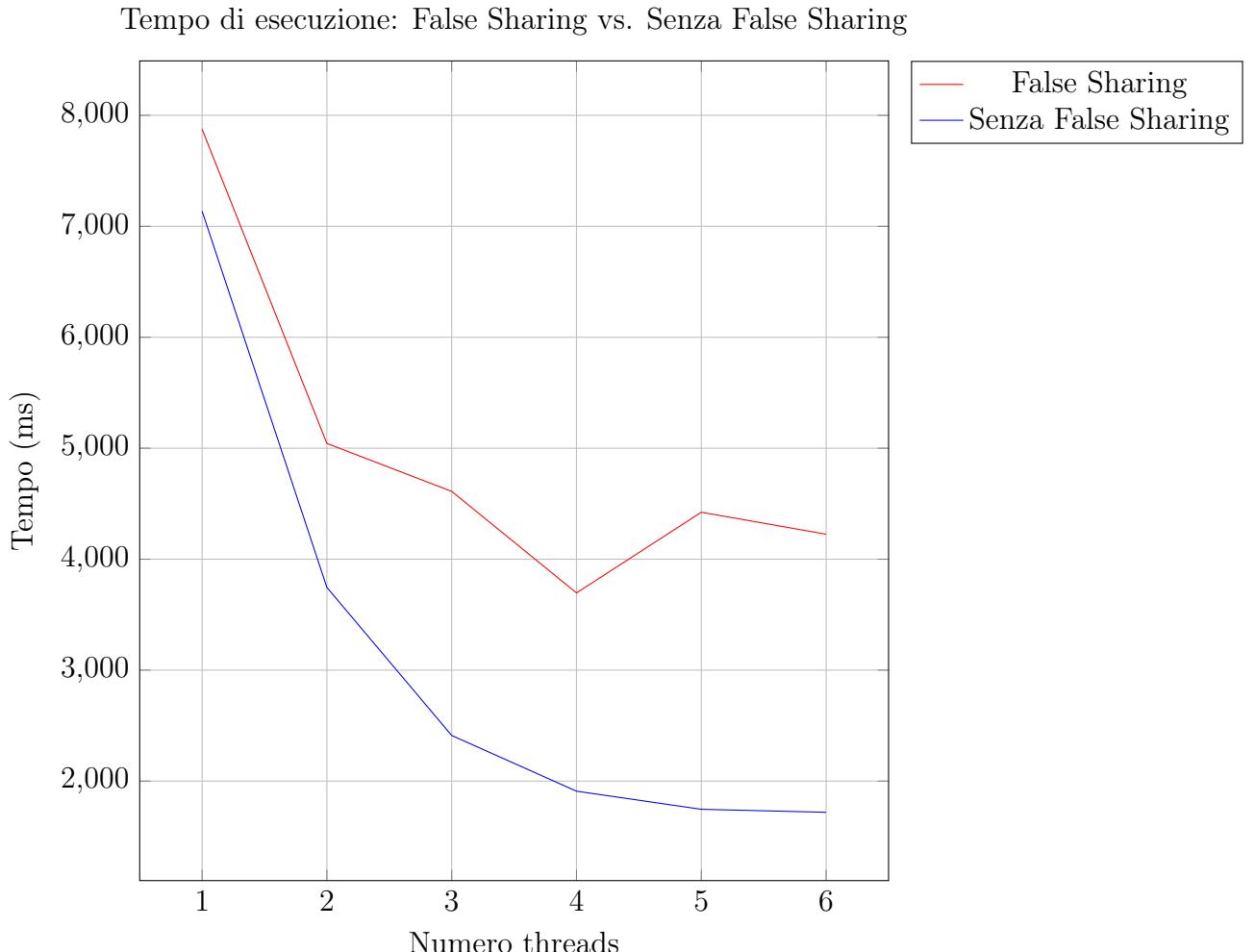


Figura 1.2: Tempo di esecuzione per trovare numeri pari in una matrice 20000x20000 con false sharing e senza false sharing

Ciò accade per il fenomeno del false sharing: nonostante i singoli threads non condividono le stesse risorse (ogni thread accede a un indice diverso del vettore ‘counters’), poichè il vettore è posizionato in memoria in maniera sequenziale, a ogni scrittura sul vettore, l’intera cache line (probabilmente contenente l’intero vettore) sarà invalidata. Per quanto concerne l’hardware, quindi, è come se i threads

accedono a una singola risorsa condivisa.

Nel secondo caso, invece, poiché i contatori locali sono definiti nei singoli threads, è improbabile che si trovino vicini in memoria; lo stesso fenomeno si ripropone invece quando viene assegnato il valore del contatore locale al vettore, ma ciò avviene solo una volta per thread, quindi, non crea overhead[5].

Il paradigma DOD sfrutta, quindi, il prefetching per migliorare le prestazioni di un programma e rende il codice più facilmente parallelizzabile con operazioni di massa su oggetti dello stesso tipo, cercando di minimizzare il false sharing. Nel prossimo capitolo si parlerà dell'implementazione di questi concetti nel motore di gioco ‘Unity’ con il suo Data-Oriented Technology Stack (DOTS).

# Capitolo 2

## Programmazione Orientata ai Dati Applicata: Unity DOTS

### 2.1 Unity DOTS: definizioni e roadmap

**Unity** Unity è un potente motore di sviluppo 3D in tempo reale, tradizionalmente basato su paradigmi orientati agli oggetti. È ampiamente utilizzato per lo sviluppo di videogiochi per la sua flessibilità e per il vasto set di strumenti, che consente agli sviluppatori di creare una varietà di applicazioni interattive. L'architettura tradizionale di Unity è in gran parte orientata agli oggetti; comprende una scena che contiene oggetti di gioco (GameObjects) in una gerarchia. Ogni GameObject è formato da diversi componenti che ne gestiscono diverse caratteristiche tra cui, ad esempio, la posizione, rotazione, dimensione (Transform); il corpo rigido (Rigidbody); la collisione con altri GameObject (Colliders); inoltre, è possibile definire componenti personalizzati che sono scripts MonoBehaviour per definire ulteriormente il comportamento dei GameObjects [6].

#### 2.1.1 Unity DOTS (Data-Oriented Technology Stack)

Unity DOTS rappresenta un cambiamento significativo rispetto a questo approccio orientato agli oggetti verso una metodologia più orientata ai dati. DOTS si basa sull'architettura Entity Component System (ECS), concentrando sulle prestazioni e sulla scalabilità attraverso i principi di progettazione orientata ai dati di cui si è discusso nel capitolo precedente. Utilizzando ECS, Unity DOTS enfatizza la composizione sull'ereditarietà, permettendo agli sviluppatori di definire solo i dati (Componenti) e i comportamenti (Sistemi) necessari per ciascuna Entità.

Oltre all'ECS, il pacchetto di librerie di Unity DOTS contiene:

- Il *C# Job System*, che permette di parallelizzare il lavoro su più entità in modo sicuro e veloce, usando il *Job System* interno di Unity scritto in C++.

- Il compilatore *Burst*, che traduce il bytecode IL/.NET in codice nativo altamente ottimizzato, utilizzando l'infrastruttura del compilatore LLVM [7].

### 2.1.2 Storia e roadmap dello sviluppo di Unity DOTS

Unity ha iniziato a sperimentare DOTS attorno al 2018 con la prima versione stabile dell'ECS pubblicata nel 2020-2021, accompagnata da diverse funzionalità per ottimizzare le prestazioni. Il 2023 ha visto ulteriori integrazioni come il miglioramento del supporto negli IDEs come Rider, che ha introdotto nuovi strumenti per facilitare lo sviluppo, consentendo la creazione di scripts e componenti in modo più semplice e veloce [8]. Tra i piani per gli sviluppi futuri c'è l'implementazione di un sistema di animazione e di controllo del personaggio che sfrutta ECS, una convergenza tra i GameObjects e le Entità e, anche se è al momento in una fase primordiale di sviluppo, la possibilità di navigare una scena basata su ECS nell'Editor [9].

## 2.2 Struttura di Unity DOTS

Libreria	Descrizione
<b>C# Job System</b>	Permette di creare jobs paralleli e multithreaded per migliorare le prestazioni del codice. Semplifica l'esecuzione asincrona delle operazioni.
<b>Burst Compiler</b>	Un compilatore che trasforma il codice C# in codice nativo ad alte prestazioni, ottimizzato per l'hardware specifico. Migliora significativamente la velocità di esecuzione.
<b>Collections</b>	Contiene strutture dati ottimizzate per l'uso in ambiente multithreaded (es. NativeArray, NativeList, ecc.). Sono fondamentali per lavorare con dati condivisi nei jobs.
<b>Entities (ECS)</b>	Sistema basato su entità e componenti (Entity Component System). Organizza i dati e la logica per ottimizzare le performance su larga scala, separando i dati dalla logica di elaborazione.

### 2.2.1 C# Job System

Per il design di Unity, gli aggiornamenti sui singoli componenti MonoBehaviour sono eseguiti solo sul main thread, quindi, tutta la logica del gioco viene eseguita solo su un core della CPU. Lo sviluppatore potrebbe creare manualmente nuovi threads, ma non c'è un modo sicuro e semplice per gestirli.

## Funzionamento del Job System

Il Job System gestisce un pool di threads di lavoro, uno per ogni core aggiuntivo della piattaforma di destinazione. Ad esempio, se Unity viene eseguito su un processore a otto core, viene creato un thread principale (main thread) e sette threads di lavoro (worker threads). Quando un worker thread è inattivo, viene estratto ed eseguito il prossimo job disponibile dalla coda. Ogni job, una volta avviato su un thread di lavoro, viene eseguito fino al termine senza interruzioni.

I Jobs possono essere pianificati (aggiunti alla coda) solo dal main thread. Inoltre, solo il main thread può chiamare `Complete()` su un Job per attendere che esso finisca (se non ha già terminato) e rilasciare così i dati usati, in modo che possano essere utilizzati in sicurezza sul main thread o passati a un altro Job [6].

## Verifiche di sicurezza e dipendenze tra i Jobs

Garantire la sicurezza e gestire le dipendenze tra i threads è fondamentale nella programmazione multithreaded per evitare condizioni di competizione, corruzione dei dati e altri problemi di concorrenza.

Ad esempio, se il Job System invia una *reference* dei dati dal codice del main thread a un Job, non si può verificare che il main thread stia leggendo i dati nello stesso momento in cui il Job li stia modificando. Questo scenario crea una *race condition*. Il Job System gestisce queste problematiche con l'isolamento dei dati nei singoli Jobs; ciò comporterà l'invio di una copia dei dati di input ai singoli Jobs piuttosto che un passaggio di dati per *reference* dal main thread. Il modo in cui il Job System copia i dati implica che un Job può accedere solo a tipi di dati *blittable* [10]; si parlerà delle conseguenze di ciò nel paragrafo successivo.

Se due Jobs, invece, hanno bisogno di utilizzare gli stessi dati, lo sviluppatore ha la possibilità di definire le dipendenze di un certo Job da altri Jobs quando esso viene aggiunto nella coda; si viene a creare, in questo modo, una catena di Jobs e un Job presente nella coda potrà partire solo dopo che tutte le sue dipendenze avranno terminato l'esecuzione.

Codice 2.1: Esempio Jobs Scheduling

---

```

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

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

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

        // Creazione dei jobs con dati condivisi
        var jobA = new IncrementJob { data = data };
        var jobB = new MultiplyJob { data = data };
        var jobC = new DecrementJob { data = data };

        // Quando un nuovo Job viene istanziato, il costruttore restituisce un
        // handle che permette di gestire il Job e usarlo come dipendenza per
        // altri Jobs

        // Programmazione dei Jobs con dipendenze
        JobHandle handleA = jobA.Schedule();
        JobHandle handleB = jobB.Schedule(handleA); // B dipende da A
        JobHandle handleC = jobC.Schedule(handleB); // C dipende da B

        // Quando il Job C sarà completato, anche tutti gli altri Jobs lo
        // saranno

        // Si attende il completamento di tutti i Jobs
        handleC.Complete();

        // Adesso si può usare il risultato
        Debug.Log("Valore finale: " + data[0]);

        data.Dispose();
    }
}

```

---

Poichè la struttura dati è *blittable*, la *garbage collection* non la gestirà, perciò bisogna rilasciarla manualmente con *data.Dispose()* (ciò sarà spiegato nel dettaglio nel paragrafo seguente).

Codice 2.2: Esempio Jobs Scheduling, definizione Jobs

---

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

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

    // La funzione 'Execute' sarà chiamata quando partirà l'esecuzione del Job
    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]--;
    }
}
```

---

## Tipologie di Jobs

Oltre alla semplice interfaccia *IJob*, che è stata mostrata nell'esempio precedente, il Job System presenta altre tipologie di Jobs utili per diversi scenari.

**IJobParallelFor** Esegue un task in parallelo, dove ogni thread ha un indice esclusivo per accedere ai dati condivisi in modo sicuro, utile per quando si possono compiere operazioni elemento a elemento in parallelo su un vettore di dati. La funzione

'Execute' avrà accesso al parametro *index*, che rappresenterà l'indice dell'elemento a cui si potrà accedere in modo sicuro.

Codice 2.3: Esempio 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 };

        // Il primo parametro è il numero di iterazioni totali
        // Il secondo parametro è la batch size (numero elementi elaborati 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;
        }
    }
}
```

---

Inoltre, Unity risolve internamente il problema del *false sharing* discusso nel capitolo precedente: con il vantaggio di accettare solo tipi di dato *unmanaged*, il Job System può gestire in modo ottimale la memoria, rendendo improbabile la presenza di elementi del vettore *data* sulla stessa *cache line*.

**IJobFor** Simile a *IJobParallelFor*, *IJobFor* permette di programmare il Job affinché non venga eseguito in parallelo, utile quando si ha bisogno di suddividere un problema vettoriale in diversi tasks, ma non si desidera che si eseguino in parallelo.

**IJobParallelForTransform** Esegue un task in parallelo, in cui ogni thread ha un *Transform* esclusivo dalla gerarchia dei trasformi, utile e ottimizzato per la necessità di modificare la posizione, la dimensione e/o la rotazione di diverse entità in una scena. Oltre all’indice del Transform, si avrà accesso anche all’oggetto *TransformAccess*, che permetterà di cambiare con semplicità i parametri citati [10].

Codice 2.4: Esempio 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	Descrizione
<b>IJob</b>	Esegue un singolo task su un thread dedicato al job.
<b>IJobParallelFor</b>	Esegue un task in parallelo. Ogni thread di lavoro che gira in parallelo ha un indice esclusivo per accedere ai dati condivisi in modo sicuro tra i threads.
<b>IJobParallelForTransform</b>	Esegue un task in parallelo. Ogni thread di lavoro che gira in parallelo ha un <i>Transform</i> esclusivo dalla gerarchia dei trasformi che su cui operare.
<b>IJobFor</b>	Simile a <i>IJobParallelFor</i> , ma consente di pianificare il job affinché non venga eseguito in parallelo.

### 2.2.2 Gestione di tipi blittable: il pacchetto Collections

**Blittable types** In C#, i dati possono essere gestiti (managed) o non gestiti (unmanaged). Il codice gestito è quello che viene eseguito all'interno del runtime di .NET, che gestisce memoria e garbage collection, mentre il codice non gestito è quello che accede direttamente alla memoria, spesso per interfacciarsi con librerie esterne scritte in linguaggi come C o C++.

La rappresentazione in memoria dei tipi blittable è la stessa sia nel codice managed che nel codice unmanaged; ciò significa che possono essere trasferiti direttamente tra il runtime gestito di .NET e il codice non gestito senza bisogno di conversioni o trasformazioni [11].

**Garbage Collection** La Garbage Collection è un meccanismo automatico utilizzato nel framework .NET (e in molti altri ambienti runtime) per gestire la memoria. Il suo scopo è recuperare la memoria allocata dagli oggetti che non sono più utilizzati, liberando così risorse e prevenendo memory leaks (perdita di memoria) [12].

I tipi unmanaged sono dati che non sono gestiti dal runtime di .NET e che, quindi, non sono soggetti alla garbage collection. Il pacchetto *Collections* fornisce strutture dati unmanaged (Native e Unsafe) che possono essere utilizzate nei jobs e nel codice compilato con Burst che si vedrà in seguito.

Per le collezioni *Native*, Unity effettua controlli di sicurezza automatici per assicurare la tempistica deallocazione, mentre per quelle *Unsafe* non ci sono controlli.

### Panoramica degli Allocators

Un allocatore gestisce la memoria unmanaged dalla quale è possibile effettuare allocazioni. Il pacchetto Collections include i seguenti allocators:

- **Allocator.Temp**: L'allocator più veloce, usato per allocazioni di breve durata, che vengono deallocate automaticamente alla fine di ogni frame o Job; non può essere passato ai Jobs.
- **Allocator.TempJob**: Allocazioni a breve termine, che possono essere passate ai Jobs; devono essere deallocate entro 4 frames dalla loro creazione.
- **Allocator.Persistent**: Allocazioni a durata indefinita, quindi, da gestire manualmente; non ci sono controlli di sicurezza per la durata dell'allocazione [13].

### 2.2.3 Burst Compiler

#### Compilazione tradizionale in Unity

Il codice C# in Unity viene compilato di default con Mono, un compilatore JIT (just-in-time) oppure, in alternativa, con IL2CPP, un compilatore AOT (ahead of time).

**JIT** Con la compilazione JIT, il codice C# scritto per Unity viene tradotto in tempo reale durante l'esecuzione del gioco o dell'applicazione; ciò significa che quando un'applicazione Unity viene eseguita su una macchina, Mono converte il bytecode del C# in codice macchina nativo, mentre l'applicazione è in esecuzione. Un vantaggio di questo approccio è la capacità del compilatore di ottimizzare il codice in base alle caratteristiche specifiche del dispositivo, come il tipo di processore, consentendo potenzialmente migliori prestazioni. Tuttavia, la compilazione JIT può comportare un ritardo iniziale, poiché la traduzione del codice avviene al momento dell'esecuzione, portando a un rallentamento dell'avvio.

**AOT** Con la compilazione AOT in Unity, il codice C# viene trasformato in codice nativo prima che l'applicazione venga distribuita o eseguita; ciò avviene attraverso IL2CPP, che converte il codice intermedio C# in codice C++ e poi in codice nativo, prima che l'applicazione venga eseguita. Tale modalità di compilazione è particolarmente utile quando si mira a piattaforme dove non è possibile o consigliabile utilizzare la compilazione JIT, come dispositivi mobili (Android e iOS) o console. Poiché il codice è già compilato in codice nativo, il tempo di avvio dell'applicazione risulta più veloce rispetto al JIT. Tuttavia, dato che la compilazione avviene in fase di build, non è possibile ottimizzare dinamicamente il codice in base al dispositivo specifico come accade con JIT [14].

## Compilazione con Burst

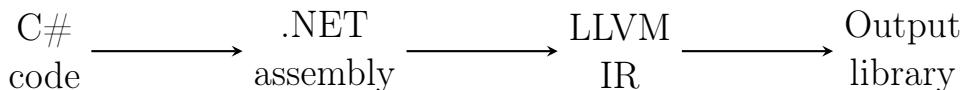
Il pacchetto Burst fornisce un terzo compilatore che esegue ottimizzazioni sostanziali, migliorando notevolmente le prestazioni e la scalabilità di problemi che richiedono pesanti calcoli.

Burst può compilare solo un sottoinsieme di C#, quindi, una buona parte del codice C# ‘tipico’ non può essere compilato con esso. La principale limitazione è che il codice compilato con Burst non può accedere agli oggetti managed, incluse tutte le istanze di classe; poiché ciò esclude la maggior parte del codice C# convenzionale, la compilazione con Burst viene applicata solo a parti selezionate del codice, come i Jobs [3].

**Funzionamento** Il codice di gioco C# viene compilato tramite l’assembly .NET, convertendo da IL (Linguaggio Intermedio) a LLVM IR (Rappresentazione Intermedia). Burst sfrutta le API di Unity per ottimizzare e compilare il codice, utilizzando una build personalizzata di LLVM.

LLVM (Low Level Virtual Machine) è un insieme di strumenti per lo sviluppo di compilatori, progettato per l’ottimizzazione del codice durante il tempo di compilazione e runtime;

il processo genera librerie native specifiche per la piattaforma tramite la compilazione AOT. Durante il runtime, le funzioni di Burst richiamano queste librerie native invece del codice .NET originale [15].



**Utilizzo** Basta annotare una classe o una funzione con l’attributo *[BurstCompile]* per compilarla con Burst, purchè il codice rispetti i vincoli mostrati in precedenza. Ci sono, inoltre, annotazioni facoltative che forniscono al compilatore informazioni aggiuntive sul codice, in modo da poterne ottimizzare ulteriormente la compilazione.

## Ottimizzazioni

**Loop Vectorization** Il compilatore Burst utilizza istruzioni SIMD (Single Instruction, Multiple Data), per eseguire, quando è possibile, più iterazioni di un loop contemporaneamente. Ciò è particolarmente utile nei processori moderni che supportano operazioni vettoriali. Ad esempio, questo è il risultato compilato con Burst del seguente codice C# che calcola la somma elemento per elemento tra due vettori:

Codice 2.5: Esempio Loop Vectorization C#

---

```

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

public class ParallelForTransformExample : MonoBehaviour
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);
}

```

---

Codice 2.6: Esempio Loop Vectorization Compilato con Burst

---

```

.LBB1_4:
    vmovdqu    ymm0, ymmword ptr [rdx + 4*rax]
    vmovdqu    ymm1, ymmword ptr [rdx + 4*rax + 32]
    vmovdqu    ymm2, ymmword ptr [rdx + 4*rax + 64]
    vmovdqu    ymm3, ymmword ptr [rdx + 4*rax + 96]
    vpadddd    ymm0, ymm0, ymmword ptr [rcx + 4*rax]
    vpadddd    ymm1, ymm1, ymmword ptr [rcx + 4*rax + 32]
    vpadddd    ymm2, ymm2, ymmword ptr [rcx + 4*rax + 64]
    vpadddd    ymm3, ymm3, ymmword ptr [rcx + 4*rax + 96]
    vmovdqu    ymmword ptr [rcx + 4*rax], ymm0
    vmovdqu    ymmword ptr [rcx + 4*rax + 32], ymm1
    vmovdqu    ymmword ptr [rcx + 4*rax + 64], ymm2
    vmovdqu    ymmword ptr [rcx + 4*rax + 96], ymm3
    add       rax, 32
    cmp       r8, rax
    jne       .LBB1_4

```

---

Come si può notare, Burst ha srotolato e vettorizzato il loop, trasformandolo in quattro istruzioni 'vpaddd', che eseguono otto addizioni intere ciascuna. In totale, ciò permette di effettuare 32 addizioni intere per ogni iterazione del loop.

Dal lato C#, lo sviluppatore può anche definire dei controlli per assicurarsi che il compilatore vettorizzi o meno qualche codice specifico:

Codice 2.7: Esempio Loop Vectorization intrinsics

---

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

        if (a[i] > b[i])
        {
            break;
        }

        a[i] += b[i];
    }
}
```

---

Il codice precedente non può essere ottimizzato con la vettorizzazione a causa dell'istruzione di branching; nonostante ciò, ci si aspetta che il codice sia vettorizzato dal compilatore (controllo evidenziato in arancio). Al momento della compilazione, si otterrà un errore [16].

**Memory Aliasing** L'aliasing in programmazione si verifica quando due o più riferimenti puntano alla stessa area di memoria; ciò può causare problemi durante le ottimizzazioni, poiché il compilatore non può determinare se il modificare una variabile influenzerebbe anche un'altra variabile *aliasata*. Di conseguenza, potrebbe essere necessario eseguire operazioni ridondanti per evitare effetti collaterali indesiderati:

Codice 2.8: Esempio Memory Aliasing C#

---

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

---

Codice 2.9: Esempio Memory Aliasing Compilato

---

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

---

Nel codice precedente, per evitare errori, un compilatore dovrebbe aspettarsi un possibile aliasing di 'b' o 'c' con 'a' e, quindi, aggiornare in maniera precauzionale il valore di 'a'. Burst, oltre ad analizzare automaticamente i casi dove non è possibile che ci sia aliasing, permette anche di annotare proprietà con *[NoAlias]* per consentirne l'ottimizzazione [17].

**Hint intrinsics** Lo sviluppatore, per ottimizzare ulteriormente il codice, può aggiungere delle *intrinsics* per far sì che il compilatore preferisca un certo ramo in un'istruzione di branch, se questo è più probabile degli altri:

Codice 2.10: Esempio Likely intrinsics

---

```
if (Unity.Burst.CompilerServices.Hint.Likely(b))
{
    // Il codice in questo ramo sarà ottimizzato da Burst con l'assunzione che
    // → sarà eseguito più probabilmente
}
else
{
```

---

Sono presenti, inoltre, annotazioni per comunicare al compilatore che una variabile è all'interno di un certo intervallo, che è costante o che non necessita di essere inizializzata.

Il programmatore può, quindi, dal codice C# di alto livello, controllare il risultato del codice compilato ottimizzato in base alle proprie aspettative sull'esecuzione [16].

## 2.2.4 Entity Component System

### Paradigma ECS

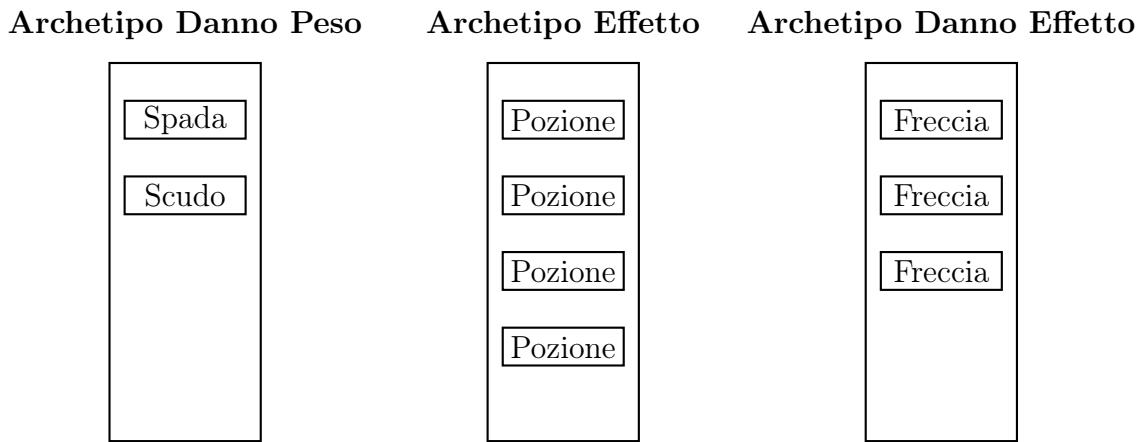
Il paradigma ECS è il cuore di Unity DOTS, poichè il codice strutturato con DOD consente e rende più efficaci le ottimizzazioni analizzate in precedenza.

Il pacchetto *Entities* fornisce un'implementazione dell'architettura ECS, composta da entità e componenti per i dati e da sistemi per il codice. Le entità sostituiscono i GameObjects della programmazione tradizionale; come i GameObjects, le entità sono composte da componenti, ma a differenza di questi, i componenti delle entità possono contenere solo dati, non hanno metodi propri e sono tipicamente degli struct (e non una classe di MonoBehaviour).

La logica di gioco, invece, è gestita dai sistemi, i quali hanno un metodo di aggiornamento invocato di solito una volta per frame, che legge e modifica i componenti delle entità.

### Archetypes

Tutte le entità che hanno lo stesso insieme di tipi di componenti sono raggruppate e immagazzinate nello stesso archetipo (archetype). Ad esempio:



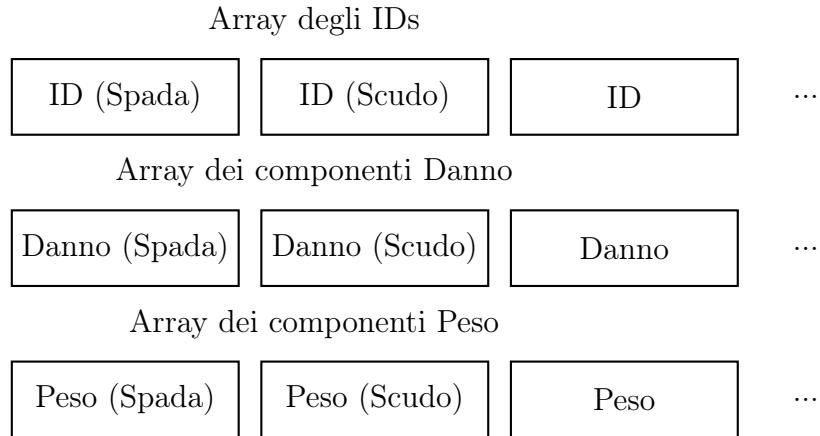
Le entità 'Spada' e 'Scudo' si trovano nello stesso archetipo, poichè entrambe hanno gli stessi tipi di componenti, *Danno* e *Peso*.

A runtime, un'entità potrebbe perdere o acquisire componenti (ad esempio, l'entità 'Freccia' potrebbe acquisire il componente 'Velocità'); in questo caso, l'entità si sposterà nell'archetipo corrispondente.

### Chunks

Dentro un archetipo, le entità sono immagazzinate in blocchi di memoria chiamati *chunks*, ognuno contenente fino a 128 entità. Gli IDs delle entità e i componenti di

ogni tipo nello stesso chunk sono immagazzinati sequenzialmente in arrays. Utilizzando l'esempio precedente, il chunk dell'archetipo 'Danno Peso' sarà strutturato in questo modo:



Per accedere a tutti i componenti e all'ID dell'entità *Spada*, per esempio, basterà accedere agli elementi di indice 0 di tutti gli arrays. Inoltre, se un'entità viene rimossa da un chunk, l'ultima entità del chunk verrà spostata all'indice rimasto vuoto per far sì che gli arrays rimangano compatti.

## Systems

Un vantaggio principale dell'organizzazione dei dati basata su archetipi e chunks è che permette queries e iterazioni efficienti sulle entità. Per iterare su tutte le entità, che hanno un certo insieme di componenti, una query cerca prima tutti gli archetipi che corrispondono ai criteri e poi itera sulle entità nei chunks degli archetipi. Poiché i componenti nei chunks risiedono in arrays compatti, vengono minimizzate le *cache misses*; inoltre, gli archetipi corrispondenti a una certa query possono essere memorizzate nella cache.

Prendiamo, ad esempio, il sistema seguente, che, ad ogni frame, muove tutte le entità *Enemy* verso destra:

Codice 2.11: Esempio System

---

```

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

public struct EnemyTag : IComponentData
{
    // Componente vuota, usata solo per 'marcare' le entità che si desidera
    // siano affette dalla query di EnemyMovementSystem
}

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

        // Usando le SystemAPI, si itera su tutte le entità che possiedono il
        // componente 'EnemyTag', oltre che un componente 'Translation' e
        // 'MoveSpeed'
        foreach (var (translation, speed) in SystemAPI.Query<RefRW<Translation>,
                RefRO<MoveSpeed>())
            .WithAll<EnemyTag>()
        {
            translation.ValueRW.Position += moveDirection * speed.ValueRO.Value
                * SystemAPI.Time.deltaTime;
        }
    }
}

```

---

Si può notare come, nel foreach, iteriamo sul componente *Translation* con modalità RW (Read-Write), poiché oltre a leggerne le proprietà, si sta modificando il valore (posizione). Per *MoveSpeed*, al contrario, si usa RO (Read-Only), poiché si ha necessità solo di leggere la proprietà (Velocità), ma non di modificarla. *SystemAPI.Time.deltaTime* rappresenta il tempo trascorso dal frame precedente; è utilizzato come fattore per assicurarsi che la posizione vari indipendentemente dal numero di frames al secondo.

### Integrazione con il Job System

Unity DOTS consente di integrare il Job System descritto in precedenza con il paradigma ECS, permettendo di eseguire lavori su specifiche queries e, quindi,

parallelizzando l'aggiornamento delle singole entità:

Codice 2.12: Esempio Job System

---

```
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); // Verso destra

        // EnemyMovementSystem crea un nuovo Job, derivato da 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]
        [ref] EnemyTag tag)
    {
        translation.Value += MoveDirection * speed.Value * DeltaTime;
    }
}
```

---

La query, precedentemente in *EnemyMovementSystem*, adesso è eseguita implicitamente in *MoveEnemiesJob*, basandosi sui parametri del metodo *Execute*. L'uso della keyword *ref* rappresenta modalità Read-Write, mentre *in* rappresenta modalità Read-Only [3].

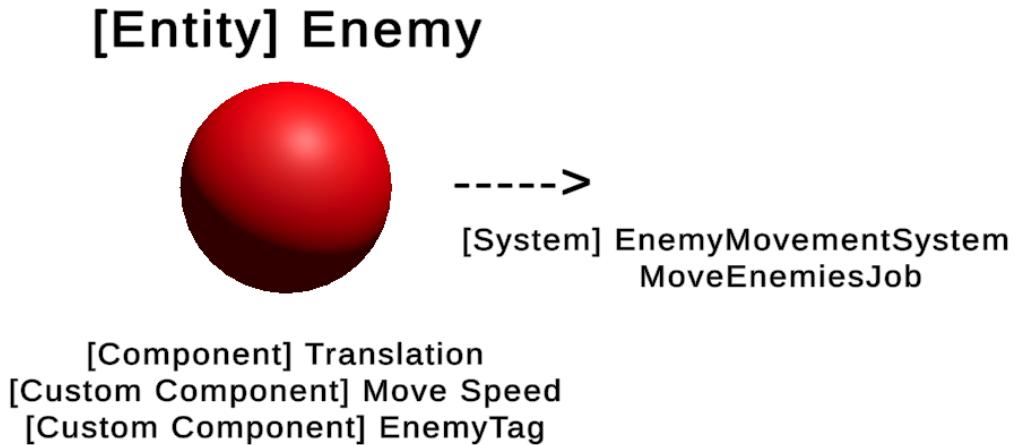


Figura 2.1: Esempio Enemy Movement

### 2.2.5 Conversione tra GameObjects ed Entità

Il sistema di scene di Unity è incompatibile con ECS; per questa ragione, per usufruire delle potenzialità del framework, come, ad esempio, quella di poter manipolare oggetti a livello grafico dall'editor di Unity, si fa uso delle *subscenes*.

Le subscenes sono annidate nella scena corrente e vengono processate tramite il *baking*, che viene eseguito ogni volta che si effettua una modifica alla subscena. Per ogni GameObject in una subscena, il processo di baking lo converte in un'entità, che viene poi caricata a runtime.

Per ogni componente del GameObject originale sono, quindi, necessari dei *bakers* che aggiungono il componente corrispettivo all'entità. Ad esempio, i bakers associati ai componenti grafici standard, come MeshRenderer, aggiungeranno componenti relativi alla grafica dell'entità.

**Bakers personalizzati** Per qualsiasi componente aggiuntivo non standard che lo sviluppatore intende creare, bisogna definire bakers per controllare quali componenti verranno aggiunti alle entità generate.

Ad esempio, si desidera creare il componente *Health*, che contiene un valore *float* che rappresenta i punti salute di un'entità:

Codice 2.13: Esempio Health Component

---

```
using Unity.Entities;

public struct Health : IComponentData
{
    public float Value; // Punti salute dell'entità
}
```

---

Oltre al componente *Health*, sarà necessario anche definire la classe MonoBehaviour *HealthAuthoring*, che potrà essere modificata a freddo nell'Unity Editor e il baker *Baker<HealthAuthoring> HealthBaker*, che convertirà il componente *HealthAuthoring* in *Health* a runtime:

Codice 2.14: Esempio Health Authoring e Baker

---

```
using Unity.Entities;
using UnityEngine;

public class HealthAuthoring : MonoBehaviour
{
    public float HealthValue; // Proprietà per impostare i punti salute iniziali
    → nell'inspector dell'editor di Unity
}

public class HealthBaker : Baker<HealthAuthoring>
{
    public override void Bake(HealthAuthoring authoring)
    {
        // GetEntity converte il GameObject in un'entità
        var entity = GetEntity(TransformUsageFlags.Dynamic);

        // Aggiunge il componente 'Health' e assegna come valore iniziale il
        → valore del componente 'HealthAuthoring' del GameObject
        AddComponent(entity, new Health { Value = authoring.HealthValue });
    }
}
```

---

Potrebbe risultare scomodo, in casi semplici, non poter aggiungere entità direttamente nelle scene; tuttavia, il processo di baking può essere utile in situazioni più

avanzate: i dati di authoring (i GameObjects che vengono modificati nell'Editor) sono separati efficacemente dai dati runtime (le entità generate dal baking), quindi, ciò che lo sviluppatore modifica a freddo e ciò che viene caricato a runtime non dovrà più necessariamente corrispondere [3].

Nel capitolo successivo si descriverà una sperimentazione e un benchmarking effettuati sulle componenti di Unity DOTS analizzate finora, applicate a scenari che richiedono un'alta intensità di risorse.

# Capitolo 3

## Implementazione e Sperimentazione con Unity DOTS

### 3.1 Informazioni Generali

#### 3.1.1 Scopo del Progetto

Il progetto ha come scopo una sperimentazione dettagliata e oggettiva delle capacità offerte dal sistema Unity DOTS attraverso un confronto con l'architettura tradizionale orientata agli oggetti, tipica dell'approccio basato su MonoBehaviour. L'applicazione, inoltre, fornisce all'utente la possibilità di effettuare un benchmarking degli scenari, consentendo di variare la complessità degli stessi mediante l'incremento del numero di entità presenti. Il punto di partenza per ciascun test è determinato da un numero di entità scelto dall'utente.

#### 3.1.2 Struttura del Progetto

Il progetto sviluppato in Unity è organizzato in un totale di 10 scene diverse. La prima scena è quella principale, dalla quale l'utente può selezionare uno dei tre scenari proposti oppure eseguire un benchmarking. Gli scenari disponibili sono suddivisi in tre scene per ciascuno, per sperimentare differenti livelli di prestazioni e ottimizzazione del codice. Ogni scenario rappresenta una composizione differente, che sarà analizzata in seguito, al fine di studiare e confrontare le performance in diverse situazioni.

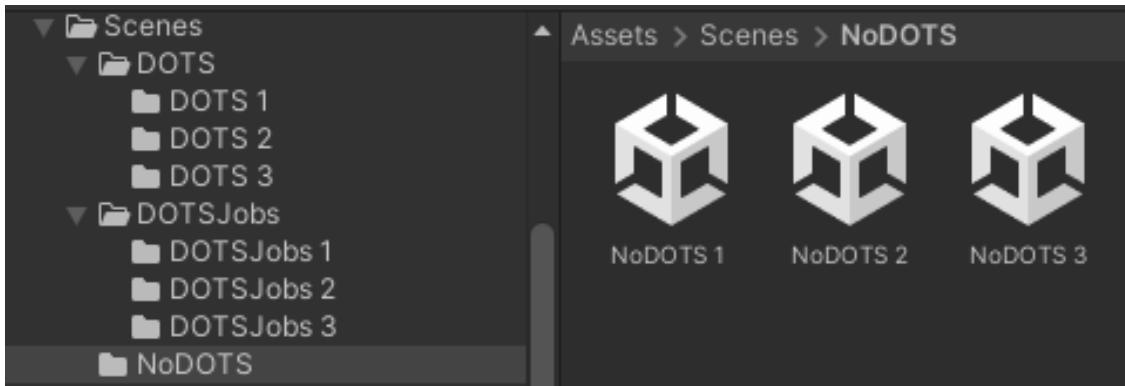


Figura 3.1: Struttura delle scene

### 3.1.3 Modalità di Ottimizzazione

Per ciascuno degli scenari proposti è possibile scegliere tre diverse modalità di ottimizzazione:

- **No Optimizations:** Questa modalità rappresenta lo scenario di base, sviluppato utilizzando l'architettura tradizionale orientata agli oggetti, con script implementati attraverso MonoBehaviour. Tale scenario funge da riferimento per il confronto delle performance con le altre modalità di ottimizzazione.
- **Burst:** In questa modalità, lo scenario viene riscritto utilizzando ECS e il codice viene compilato con Burst.
- **Burst + Jobs:** Questa modalità combina l'utilizzo di ECS e Burst, ma aggiunge anche il Job System. In tutti i casi in cui è necessario iterare su un numero di entità per modificarne le proprietà o effettuare calcoli, i Jobs vengono istanziati e gestiti in parallelo per migliorare ulteriormente le prestazioni complessive.

### 3.1.4 Descrizione dell'UI

**Hub** Nella scena 'Hub' è possibile selezionare uno scenario e il livello di ottimizzazione desiderato dal menù a tendina. Passando il mouse sul rispettivo pulsante, verrà mostrata a schermo una descrizione dell'ottimizzazione selezionata; inoltre, è possibile passare alla modalità 'benchmark' e modificarne le sue impostazioni: la durata delle snapshots, il numero di entità di base, l'incremento di entità per ogni snapshot e la durata di ogni snapshot. Verrà, inoltre, mostrata a schermo la durata complessiva del processo di benchmarking, in base alle impostazioni selezionate.

### 3.1 – Informazioni Generali

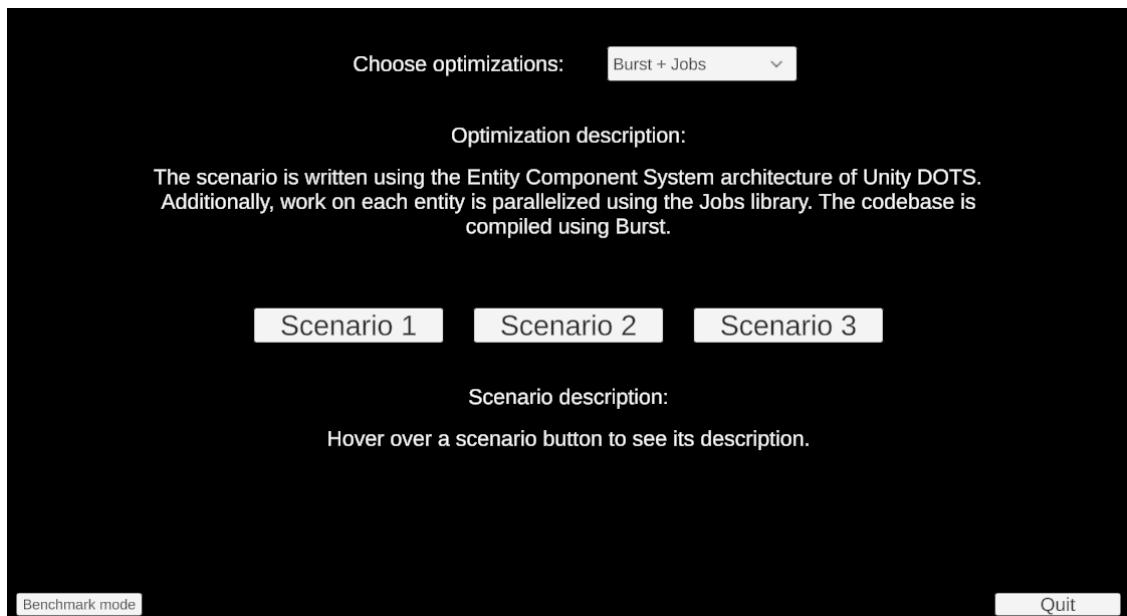


Figura 3.2: Hub UI

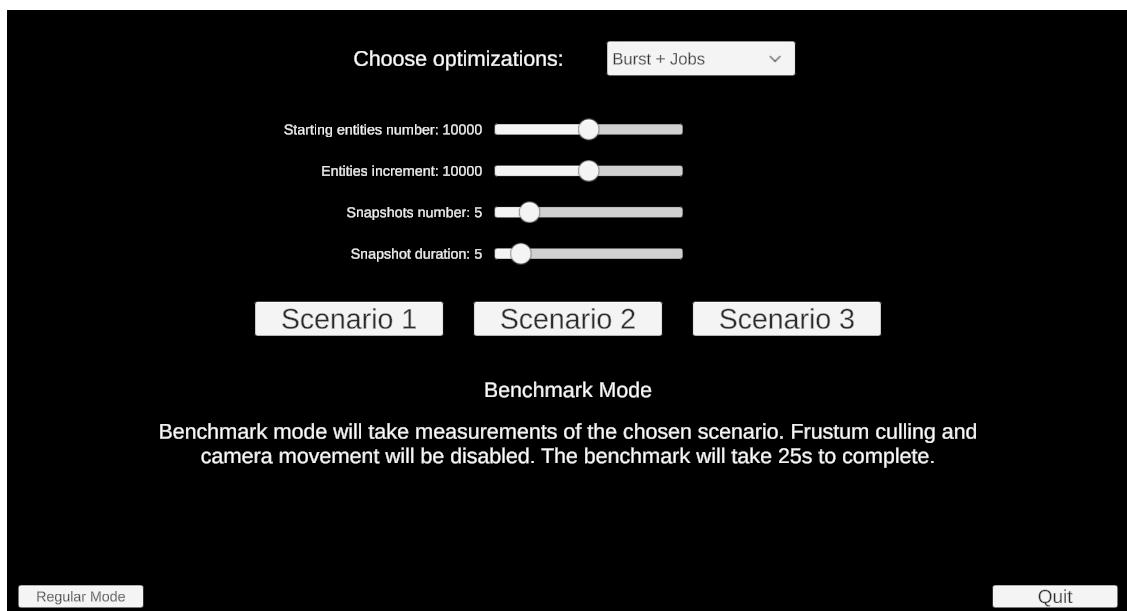


Figura 3.3: Hub Benchmark UI

**Scenario** Quando si seleziona uno scenario e il livello di ottimizzazione desiderato dalla scena 'Hub', si accederà alla scena specifica dello scenario. In questa scena ci sono due slider per modificare rispettivamente il parametro 'numEntities' e 'spawnRadius', un pulsante che riavvia la simulazione e uno che permette di tornare alla scena 'Hub'. Inoltre, premendo il tasto [ESC], è possibile nascondere l'UI per

guardarsi attorno nella scena e muoversi con i tasti 'WASD' o direzionali. Inoltre, è possibile variare l'altezza usando il tasto [SPAZIO] e [SHIFT]. Infine, è possibile muoversi più velocemente nello spazio con [LEFT CONTROL].

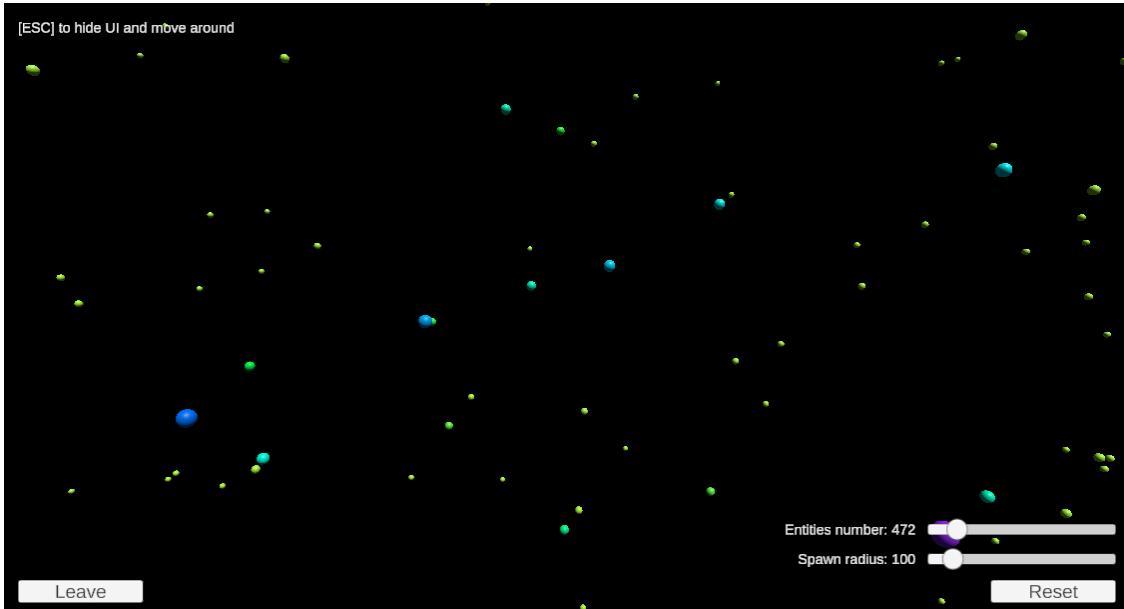


Figura 3.4: Scenario UI

In modalità benchmark, invece, non è possibile interagire in alcun modo con la simulazione (a parte uscire dalla stessa), fino alla fine del processo, in cui verranno mostrati a schermo i dati raccolti in formato JSON.

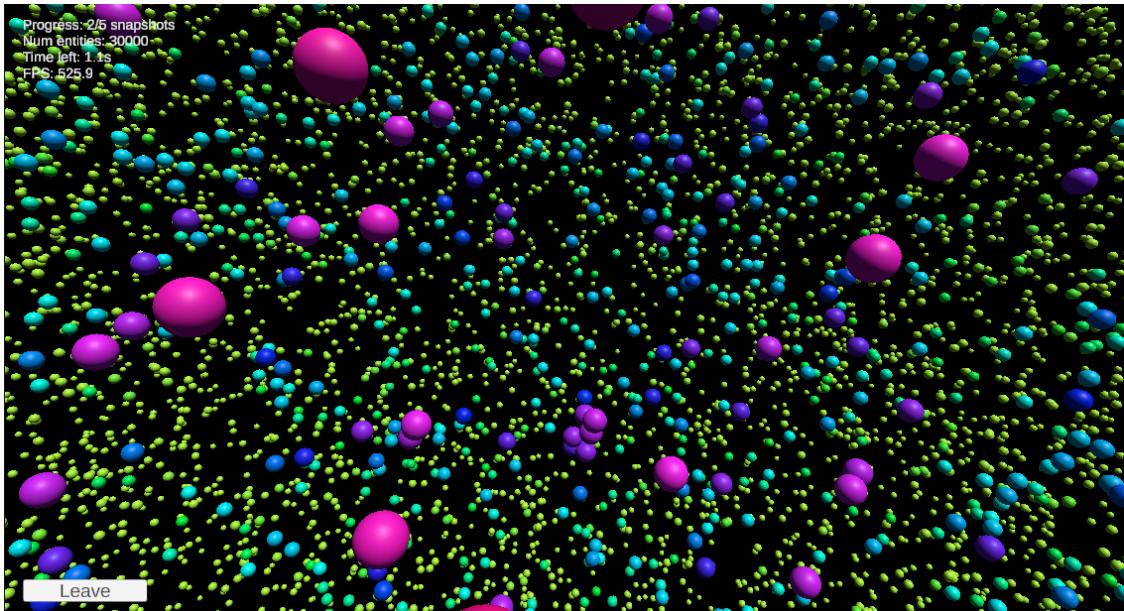


Figura 3.5: UI Benchmark in Corso



Figura 3.6: UI Benchmark Completo

### 3.1.5 Descrizione degli scenari

Tutti gli scenari presentano un 'numEntities' di entità 'sfere' che vengono istanziate in un raggio di 'spawnRadius'. La differenza tra gli scenari è il comportamento delle sfere.

**Scenario 1** Nel primo scenario le sfere si muovono in avanti e indietro lungo una direzione casuale assegnata con una velocità di base fissa. La velocità delle sfere, entro un certo limite, aumenta all'avvicinarsi dell'utente, insieme al colore. Il calcolo più computazionalmente complesso che ogni sfera deve compiere ogni frame è il calcolo della distanza tra la sfera e la posizione dell'utente (posizione dell'oggetto *MainCamera*). Si potrebbe, quindi, stimare che, per ogni frame, in questo scenario, vengono compiute  $O(\text{numEntities})$  operazioni computazionalmente rilevanti.

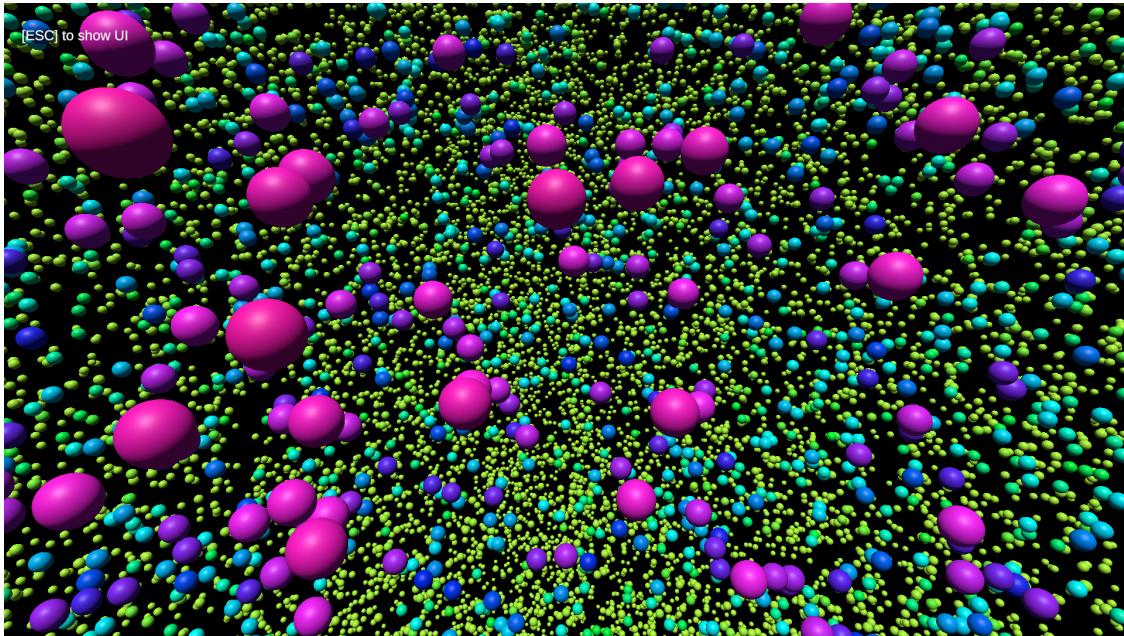


Figura 3.7: Scenario 1

**Scenario 2** Nel secondo scenario le sfere si muovono in avanti e indietro lungo una direzione casuale assegnata, ma con una velocità che stavolta rimane sempre fissa. Ogni sfera orienta un oggetto cilindrico chiamato 'Link' verso la sfera più vicina. Il calcolo più computazionalmente complesso che ogni sfera deve compiere per ogni frame è quello di calcolare la distanza da tutte le altre sfere e di trasformare l'entità 'Link' in modo che punti la sfera con la distanza più bassa. Si potrebbe, quindi, stimare che, per ogni frame, in questo scenario, vengono compiute  $O(\text{numEntities}^2)$  operazioni computazionalmente rilevanti.

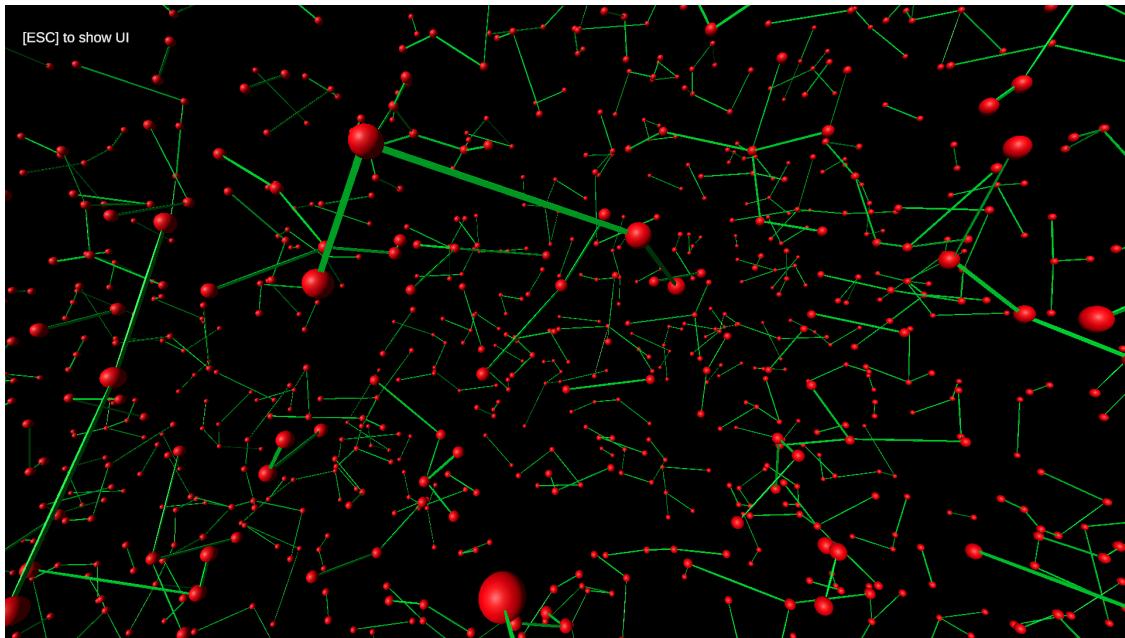


Figura 3.8: Scenario 2

**Scenario 3** Nel terzo e ultimo scenario, ogni sfera possiede un corpo rigido e dei bounds di collisioni ed è spinta mediante una forza iniziale verso una direzione casuale. Ogni sfera riconoscerà una collisione, cambiando il proprio colore in base al tipo di collisione (sfera-sfera o sfera-muro). I muri si allargheranno in base al parametro 'spawnRadius'.

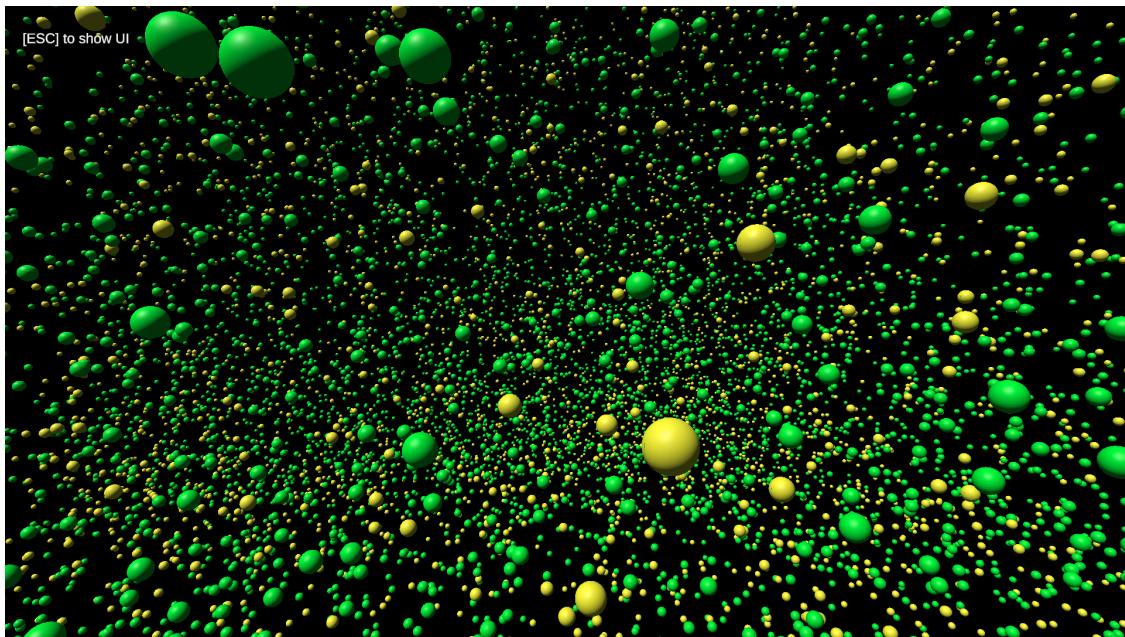


Figura 3.9: Scenario 3

## 3.2 Strumenti e sfide dello sviluppo

Prima di analizzare nel dettaglio il funzionamento degli scenari con i diversi livelli di ottimizzazione, risulta necessario riportare delle informazioni sullo sviluppo del progetto nel complesso.

### 3.2.1 Architettura MVC per UI

Nel progetto, gli scripts per il funzionamento di uno scenario non vengono riutilizzati per un altro scenario; nonostante ciò, sono necessari degli scripts comuni a tutti gli scenari che gestiscono l’interfaccia grafica e le impostazioni degli scenari e del benchmark. Questi scripts sono stati ideati con il paradigma MVC (Model View Controller). Ci sono, quindi, scripts di tre tipi:

- **Handlers:** Scripts Monobehaviour che rispondono direttamente all’input dell’utente (come il movimento della videocamera o il cambiamento delle impostazioni dello scenario); se necessario, invocano gli scripts di tipo APIs. Nel modello MVC gli Handlers corrispondono a ’View’.
- **APIs:** Classi statiche le cui funzioni sono invocate dagli Handlers (ad esempio, per salvare le impostazioni di benchmarking). Nel modello MVC le APIs corrispondono a ’Controller’.
- **Objects:** Classi serializzabili contenenti solo dati. Nel progetto sono usate per la rappresentazione delle possibili impostazioni di uno scenario o del benchmarking. Nel modello MVC gli Objects corrispondono a ’Model’.

### 3.2.2 Comunicazione tra UI in Monobehaviour ed Entity World

Una sfida dello sviluppo di scenari interamente composti da codice compilabile con Burst è quella di dover gestire il passaggio di informazioni, quali le impostazioni degli scenari stessi dal mondo dei GameObjects tradizionale a quello delle Entità; ciò è necessario, poiché gli oggetti grafici come gli sliders e, in generale, l’intero sistema di UI di Unity è ancora incompatibile con DOTS. Fortunatamente, è necessario caricare le impostazioni dello scenario (`numEntities` e `spawnRadius`) soltanto quando quest’ultimo viene resettato. La soluzione è quella di usare un system `SettingsLoaderSystem`, il quale, nel suo `OnUpdate`, recupera i parametri richiesti dalla classe `ScenarioSettingsAPIs` e li salva in un componente di tipo `SettingsLoader`. Questo sistema, dato che accede all’oggetto `ScenarioSettings` tramite lo script di tipo API, non può essere compilato con Burst, poiché Burst gestisce solo tipi di dati unmanaged, come spiegato in precedenza.

Codice 3.1: SettingsLoaderSystem.cs

---

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

    [BurstCompile]
    public void OnCreate(ref SystemState state)
    {
        // Necessario per poter reperire il componente SettingsLoader in
        // → OnUpdate
        state.RequireForUpdate<SettingsLoader>();
    }

    // Questa funzione non può essere compilata in Burst, perché gestisce tipi
    // → managed tramite ScenarioSettingsAPIs
    public void OnUpdate(ref SystemState state)
    {
        var config = SystemAPI.GetSingleton<SettingsLoader>();

        if (config.Initialized) return;

        // Questa flag è utilizzata per far sì che il codice in OnUpdate venga
        // → eseguito solo una volta per SettingsLoader (il quale viene resettato
        // → quando lo scenario si riavvia)
        config.Initialized = true;

        // Le impostazioni sono salvate in SettingsLoader, in modo che gli altri
        // → sistemi possono reperirle senza dover usufruire di tipi managed
        config.numSpheres = ScenarioSettingsAPIs.GetNumEntities();
        config.spawnRadius = ScenarioSettingsAPIs.GetSpawnRadius();
        config.benchmarkMode = ScenarioSettingsAPIs.IsBenchmarkMode();

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

---

Gli altri sistemi presenti nella codebase, che hanno bisogno di reperire le impostazioni dello scenario per funzionare, potranno essere compilati con Burst, perché reperiranno i dati direttamente dal componente SettingsLoader, assicurandosi che l'OnUpdate di SettingsLoaderSystem descritto in precedenza sia già avvenuto.

Codice 3.2: Snippet di SpheresSpawnerSystem.cs

---

```
// Annotazione necessaria per garantire che le impostazioni siano reperite al
// momento giusto dal componente SettingsLoader
[UpdateAfter(typeof(SettingsLoader.SettingsLoaderSystem))]
[BurstCompile]
public partial struct SpheresSpawnerSystem : ISystem
{
    // [...]

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

        // Codice che utilizza 'settings.numEntities' e 'settings.spawnRadius'
        // per inizializzare le sfere nella scena
        // [...]
    }
}
```

---

Infine, in ogni scena, in cui desideriamo caricare le impostazioni, possiamo aggiungere dall'Inspector in Unity, un GameObject con il componente MonoBehaviour SettingsLoaderAuthoring, il quale, con il processo di baking descritto nel capitolo precedente, istanzierà un'entità con il componente SettingsLoader all'avvio della scena. Il paradigma dello script di authoring è presente in tutti i componenti che si analizzeranno nella trattazione di questo progetto.

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 Uso di Unity Profiler per evidenziare i colli di bottiglia

Il Profiler di Unity raccoglie e visualizza dati relativi alle prestazioni dell'applicazione in diverse aree, come la CPU, la memoria, il rendering e l'audio. È uno strumento utile per identificare le aree in cui è possibile migliorare le prestazioni e ottimizzarle; consente di individuare con precisione aspetti come il codice, le risorse, le impostazioni delle scene, il rendering della camera e le impostazioni di build che influenzano le prestazioni. I risultati vengono mostrati tramite una serie di grafici, che permettono di visualizzare i picchi nelle prestazioni. Inoltre, è possibile visualizzare esattamente quali funzioni richiedono più tempo di esecuzione in un frame per evidenziare dove è utilizzata la maggior parte del tempo di elaborazione e, quindi, le operazioni che, se ottimizzate, potrebbero portare a un aumento notevole delle prestazioni.

Ad esempio, si può confrontare il profiler dell'esecuzione dello scenario 1 senza l'uso delle ottimizzazioni di Unity DOTS con quello che presenta Burst e la parallelizzazione in Jobs.

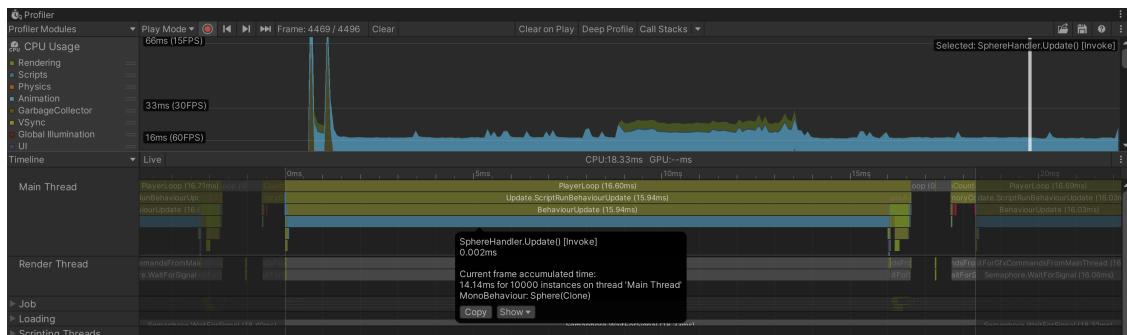


Figura 3.10: Profiler di Unity senza DOTS

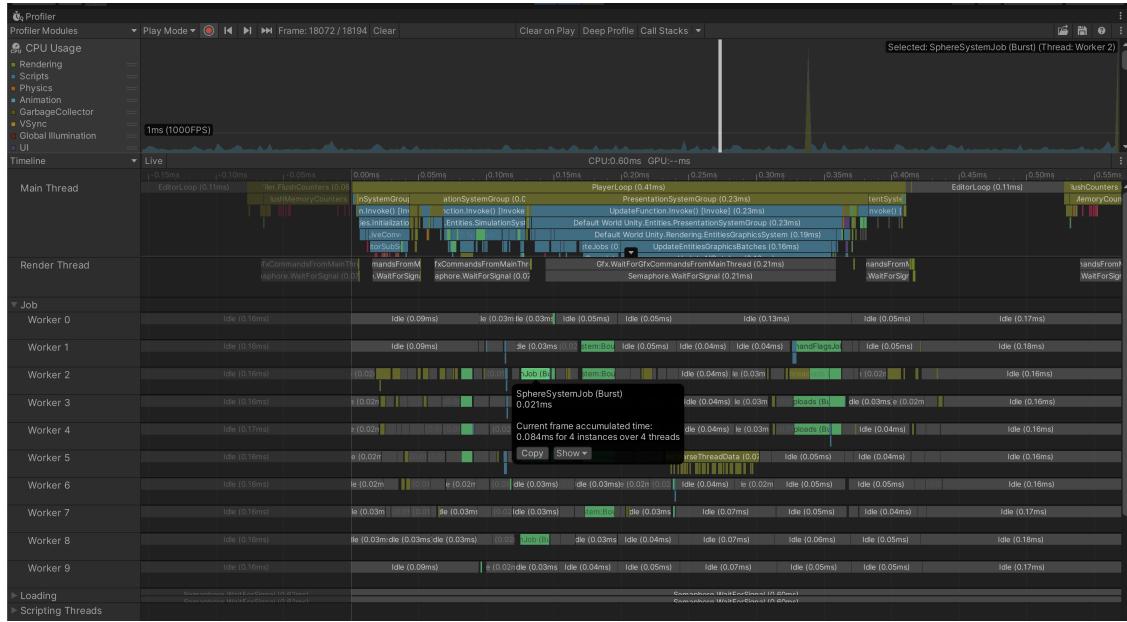


Figura 3.11: Profiler di Unity con DOTS

Come si può notare, nel primo caso, la funzione Update dell'oggetto Sphere prende una gran parte del tempo di elaborazione del frame; nel secondo caso, invece, il lavoro viene diviso in diversi worker threads.

### 3.2.4 Disabilitare il Frustum Culling

Il Frustum Culling è un'ottimizzazione a livello del game engine di Unity che permette di non renderizzare gli oggetti che il giocatore non può vedere, perché sono fuori dal campo d'azione della videocamera [18].

Anche se questa ottimizzazione aumenta di molto le prestazioni negli scenari che saranno analizzati, durante il processo di benchmarking è necessario che la simulazione, indipendentemente dalla posizione del giocatore, abbia la stessa intensità computazionale.

Per questa ragione, nel progetto sono presenti degli scripts, sia nella versione MonoBehaviour che in quella con ECS, che disabilitano il frustum culling durante il processo di benchmarking, aumentando le dimensioni della bounding box delle sfere, in modo che Unity le renderizzi sempre.

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),
                // imposta l'estensione dei bordi dell'entità a un numero
                ↵ arbitrariamente alto
                Extents = new Unity.Mathematics.float3(1000, 1000, 1000)
            };
        }
    }
}
```

---



# Capitolo 4

## Creazione degli Scenari

**SphereSpawnerSystem** Come accennato in precedenza, la classe SphereSpawnerSystem si occupa di istanziare le entità Sphere e inizializzare i loro parametri in base allo scenario. In questa fase non c'è una significativa differenza di implementazione dalla programmazione tradizionale in MonoBehaviour; per istanziare una nuovo entità verranno chiamati i metodi dell'EntityManager, anzichè quelli dei GameObject e in entrambi i casi saranno impostati i valori iniziali dei componenti dell'Entità/del GameObject. Il sistema con ECS, in più, dovrà aspettare che le impostazioni dello scenario siano state convertite dal SettingsLoaderSystem e, inoltre, poichè può eseguire codice esclusivamente nell'OnUpdate, dovrà disattivarsi dopo aver inizializzato le entità richieste; ciò è possibile usando la flag 'Initialized', inizialmente impostata a 'false', del componente SpheresSpawner.

Codice 4.1: OnUpdate SpheresSpawnerSystem.cs - Scenario 1

---

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

    // Le impostazioni dello scenario
    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.AddComponentData(entity, new LocalTransform
        {
            Position = position,
            Rotation = Quaternion.identity,
            Scale = 1f
        });

        // Nel primo scenario, ad esempio, le sfere hanno una velocità di base
        ↵ fissa ma una direzione casuale
        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 Creazione Scenario 1: Movimento di Entità in base alla Distanza dalla Videocamera

### Posizione della MainCamera

La sfida dell’implementazione in DOTS di questo scenario è quella della conversione della posizione della videocamera nel mondo delle entità. L’oggetto MainCamera, infatti, non è un’entità ma un GameObject tradizionale, perciò non può essere gestito in codice compilato con Burst. Per questa ragione, in entrambe le implementazioni Job e Jobless dello scenario 1, il sistema SphereSystem si procurerà la posizione corrente della videocamera in una funzione non compilata con Burst e, successivamente, invocherà un Job o una funzione compilata con Burst, usando la posizione convertita in tipo ‘float3’, compatibile con ECS.

Codice 4.2: SphereJoblessSystem.cs - Scenario 1

---

```
[BurstCompile]
public partial class SphereJoblessSystem : SystemBase
{
    // Questa funzione non è compilata con Burst, poichè gestisce il tipo
    // → CameraHandler.currentCameraTransform
    protected override void OnUpdate()
    {
        var deltaTime = SystemAPI.Time.deltaTime;

        // La posizione della MainCamera viene implicitamente convertita da
        // → Vector3 a float3
        InternalUpdate(deltaTime,
        // → CameraHandler.currentCameraTransform.position);
    }

    [BurstCompile]
    private void InternalUpdate(float deltaTime, float3 cameraPosition)
    {
        // Entities.WithNone<SphereJobTag>() [...]
        // Query e aggiornamento posizione, velocità e colore delle sfere
        // [...]
    }
}
```

---

### 4.1.1 Parallelizzazione con Job System

Se è selezionata l’ottimizzazione ‘Burst + Jobs’ per questo scenario, le sfere che sono istanziate hanno il componente ‘SphereJobTag’. In questo modo, lo script

che utilizza il Job System può operare, tramite query, solo su Entità in cui questa ottimizzazione è abilitata e lo script Jobless, invece, influenzera solo quelle che non possiedono tale tag (come mostrato nello snippet di codice precedente). Questo paradigma è uguale per tutti gli scenari.

Nello scenario 1, in particolare, si utilizza un Job di tipo IJobEntity, che itera in modo parallelo su tutte le entità corrispondenti a una certa query.

Codice 4.3: SphereSystemJob.cs - Scenario 1

---

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

    // Iterazione su tutte le entità che possiedono un componente
    // LocalTransform, Sphere, URPMaterialPropertyBaseColor, SphereMove e
    // SphereJobTag
    [BurstCompile]
    private void Execute(ref LocalTransform localTransform, ref Sphere sphere,
        ref URPMaterialPropertyBaseColor baseColor, in SphereMove sphereMove, in
        SphereJobTag sphereJobTag)
    {
        // Calcolo velocità in base alla distanza dalla MainCamera
        var speedMultiplier = CalculateSpeedMultiplier(localTransform.Position);

        // Calcolo posizione in base al tempo trascorso dal frame precedente e
        // la velocità
        var delta = sphere.Speed * speedMultiplier * DeltaTime *
            sphereMove.Direction;
        var newPosition = localTransform.Position + delta;

        // Aggiornamento posizione (e inversione di rotta in caso la sfera si
        // allontani troppo dalla sua posizione iniziale)
        if (math.distance(sphereMove.InitialPosition, newPosition) >
            sphere.Spread)
            sphere.Speed *= -1;
        else
            localTransform = localTransform.Translate(delta);

        // Cambio colore in base alla velocità
        var color = Color.HSVToRGB(speedMultiplier, 1, 1);
        baseColor.Value = new float4(color.r, color.g, color.b, 1);
    }
}
```

---

Lo SphereSystem, quindi, nella versione che utilizza il Job System, anzichè eseguire codice direttamente nel sistema, istanzierà un SphereSystemJob e lo programmerà (schedule) in parallelo:

Codice 4.4: SphereSystem.cs - Scenario 1

---

```
public partial class SphereSystem : SystemBase
{
    // Per l'uso di CameraHandler.currentCameraTransform, questa funzione non
    // può essere compilata con Burst, al contrario del Job istanziato
    protected override void OnUpdate()
    {
        // Tempo che intercorre tra frame, usato per calcolare la posizione
        // indipendentemente dal frame rate
        var deltaTime = SystemAPI.Time.deltaTime;

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

        job.ScheduleParallel();
    }
}
```

---

## 4.2 Creazione Scenario 2: Calcolo Entità più Vicina a ogni altra Entità

### 4.2.1 Gestione collegamenti tra sfere

La particolarità di questo scenario è che l’Entità Sphere adesso possiede anche un’entità figlia: ‘Link’. Sarà necessario trasformare questa entità in modo che punti alla sfera più vicina.

Dunque, è ottimale creare due sistemi separati, il primo che muove le singole Sphere in modo equivalente a quanto analizzato nello Scenario 1, ma con velocità fissa, e un altro che calcola le distanze e aggiorna l’entità Link.

### 4.2.2 Implementazione con Job System

La versione Jobless di questo scenario è implementabile, con qualche accorgimento in più, in modo simile a quanto analizzato nello Scenario 1; più interessante, invece,

è l'implementazione con il Job System. La soluzione adottata è stata quella di far partire in sequenza due lavori:

- **CalculateClosestSphereJob:** Per ogni Sphere viene calcolata la sfera più vicina, prendendo come input un array di posizioni delle sfere.
- **UpdateLinkTransformJob:** Per ogni Link viene aggiornato il suo Transform, prendendo come input la posizione della sfera più vicina, calcolata nel Job precedente.

Utilizzare due jobs diversi, in cui il secondo dipende dal primo, è ottimale, perchè si itera su Entità diverse.

Codice 4.5: LinkSphereSystem.cs - Scenario 2

---

```
[BurstCompile]
public void OnUpdate(ref SystemState state)
{
    // Query per ottenere tutte le sfere
    var parentQuery = SystemAPI.QueryBuilder().WithAll<LocalTransform,
        SphereJobTag>().Build();

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

    // Primo job per calcolare la distanza di tutte le sfere. Il risultato sarà
    // scritto in ClosestSphereIndices, che verrà usato come input nel Job
    // successivo
    var calculateJob = new CalculateClosestSphereJob
    {
        SpherePositions = spherePositions,
        NumSpheres = numSpheres,
        ClosestSphereIndices = closestSphereIndices
    };

    // Il lavoro è di tipo IJobParallelFor, in modo da poter eseguire più
    // operazioni su diversi elementi dell'array di input in bulk
    var calculateHandle = calculateJob.Schedule(numSpheres, 64);

    // LocalToWorldLookup permette di ottenere, in modo thread safe, l'entità
    // figlia 'Link', conoscendo l'entità 'Sphere'
    var localToWorldLookup =
        SystemAPI.GetComponentLookup<LocalToWorld>(isReadOnly: false);
    localToWorldLookup.Update(ref state);

    // Aggiornamento delle dipendenze dei lavori per far sì che
    // UpdateLinkTransformJob parta solo quando CalculateClosestSphereJob ha
    // terminato
    JobHandle combinedDependencies =
        JobHandle.CombineDependencies(calculateHandle, state.Dependency);

    // Secondo Job che aggiorna i transforms delle entità Link in base alla
    // sfera più vicina, calcolata in precedenza
    var updateJob = new UpdateLinkTransformJob
    {
        ClosestSphereIndices = closestSphereIndices,
        SpherePositions = spherePositions,
        LocalToWorldLookup = localToWorldLookup,
        LinkEntities = linkEntities
    };

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

    state.Dependency = updateHandle;      53

    // Discarding degli arrays nativi utilizzati al termine di entrambi i lavori
    // (in altre parole, quando updateJob ha terminato)
    sphereEntities.Dispose(state.Dependency);
    // [...]
}
```

---

## 4.3 Creazione Scenario 3: Utilizzo di Entità con Corpo Rigido e Collisioni

### 4.3.1 Conversione GameObject con corpo rigido

Fortunatamente, Unity DOTS presenta già componenti authoring che permettono di convertire i componenti GameObjects del corpo rigido (RigidBody) e delle collisioni (SphereCollision) nei corrispettivi componenti e sistemi nel mondo delle Entità. Per questa ragione, è necessario solo aggiungere al prefab Sphere questi componenti inclusi nel pacchetto *Unity.Physics*:

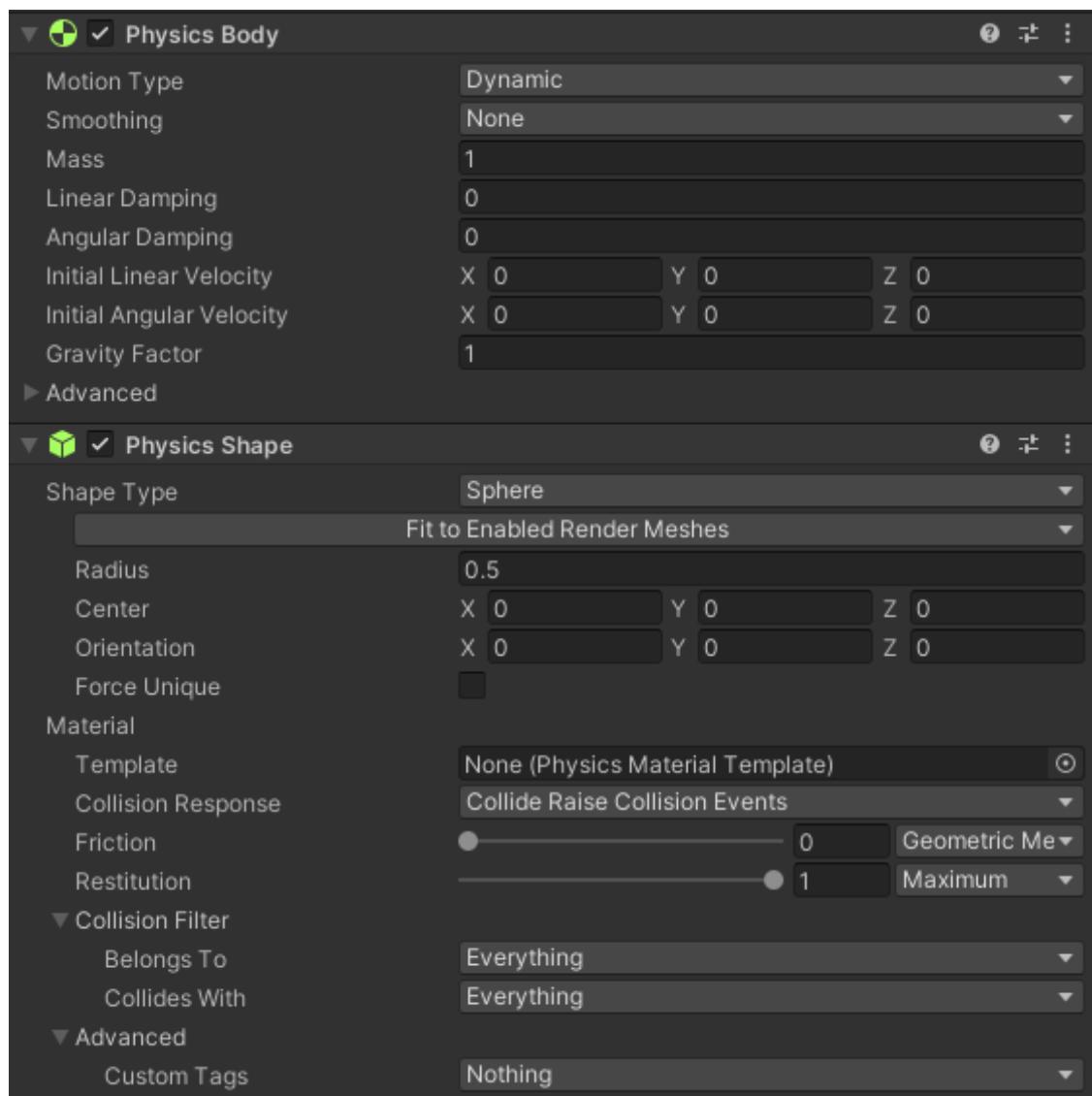


Figura 4.1: Inspector Sphere prefab con Rigidbody

### 4.3.2 Parallelizzazione con Job System

L'implementazione di questo scenario usando il Job System è interessante, perché utilizza un Job di tipo *ICollisionEventsJob*, il quale permette di iterare in parallelo su tutte le collisioni che avvengono nel mondo e sulle entità coinvolte. Lo script, quindi, dovrà controllare i componenti delle entità coinvolte per stabilire se la collisione è avvenuta sfera contro sfera o sfera contro muro e cambiare il colore delle sfere.

Codice 4.6: SphereCollisionJob.cs - Scenario 3

---

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

    [BurstCompile]
    // Invocato ogni volta che si verifica una collisione nel mondo delle entità
    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);

        // Aggiorna colori in base a entityAIsWall e entityBIsWall [...]
    }
}
```

---

### 4.3.3 Creazione dell'oggetto Muro

Il muro (wall) è una singola entità con un *Mesh collider* personalizzato a forma di cubo ma con le facce rivolte verso l'interno per contenere le sfere. Per ottenere questo effetto, il mesh è stato creato usando Blender, un software open-source per la creazione di contenuti 3D, utilizzato anche per la modellazione. Si è partiti da un semplice cubo e sono state invertite le normali delle sue facce. Successivamente, l'oggetto è stato esportato per poi essere importato in Unity come mesh personalizzato.

## Creazione degli Scenari

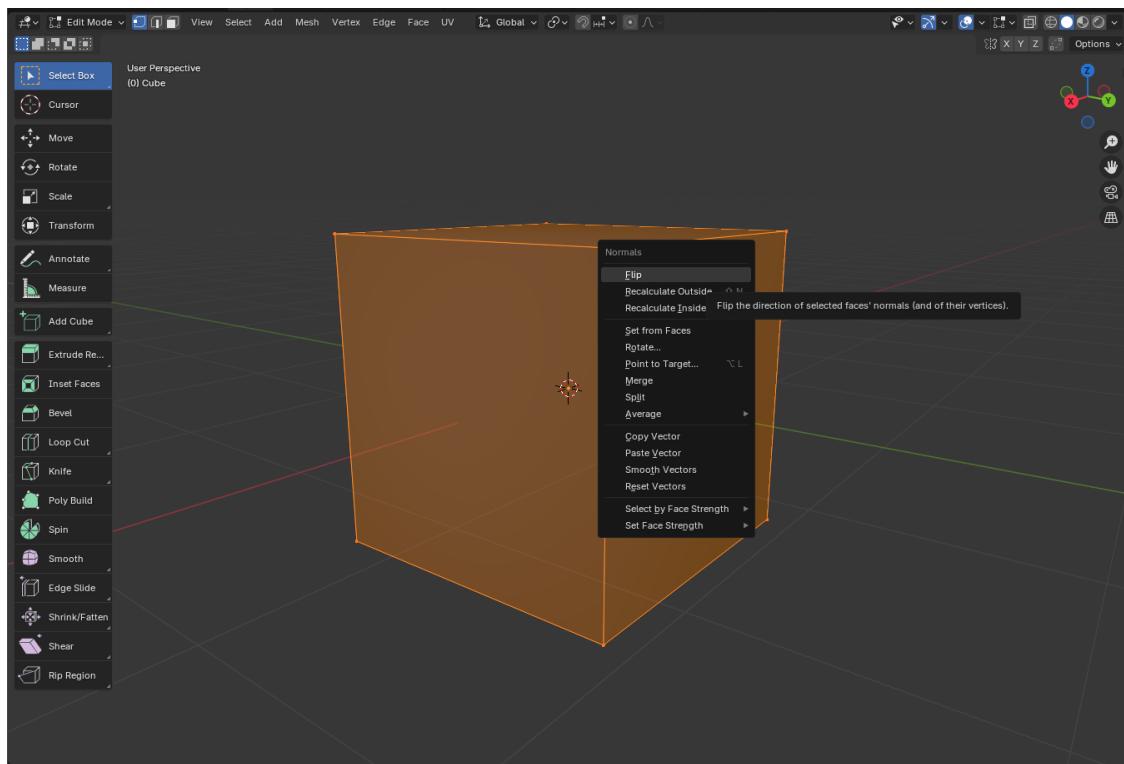


Figura 4.2: Inversione delle normali in Blender

Inoltre, sia il muro che le sfere possiedono un *Physic Material* personalizzato aggiunto al loro Mesh Collider, il quale annulla completamente la frizione e rende le collisioni completamente elastiche; in questo modo, anche se la simulazione rimane in esecuzione per un tempo prolungato, le sfere non smettono di rimbalzare.

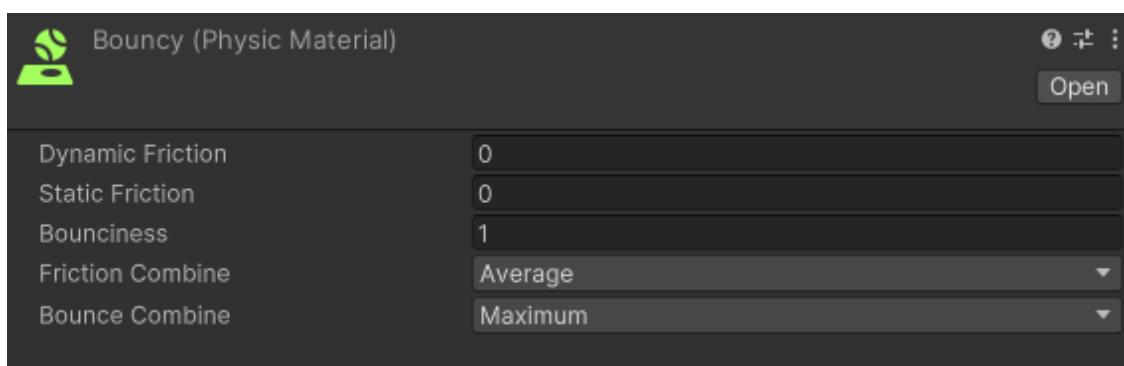


Figura 4.3: Physic Material Bouncy

Dopo la descrizione delle sfide e delle peculiarità dell’implementazione degli scenari, si può passare all’analisi del sistema di benchmarking e dei risultati da esso ottenuti.

# Capitolo 5

## Benchmarking e Analisi dei Risultati

### 5.1 Implementazione del Benchmarking

#### 5.1.1 PerformanceProfiler API

Per il benchmarking è stata creata la classe *PerformanceProfiler*, di tipo API personalizzata, che permette di calcolare la media di FPS lungo un intervallo di tempo; ciò avviene calcolando e immagazzinando la distanza tra frame in un certo intervallo e restituendo la media di queste misurazioni al termine del processo:

Codice 5.1: Snippet di 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 Riavvio automatizzato dello scenario

La classe MonoBehaviour *ScenarioBenchmarkHandler* di tipo Handler ha il compito di riavviare lo scenario quando si è raggiunta la durata dello snapshot di benchmarking, di modificare le impostazioni dello scenario per lo snapshot successivo e di mostrare a schermo i risultati quando il processo di benchmarking è terminato (ovvero si è raggiunto il numero di snapshots).

**Coroutines** L'Handler ha la necessità di attendere un numero fisso di secondi prima di fermare il PerformanceProfiler e riavviare lo scenario; ciò è facilmente realizzabile usando le *Coroutines*, un sistema nativo di Unity che usa internamente le C# Tasks e che permette facilmente di invocare del codice asincrono da scripts MonoBehaviour.

Una coroutine, inoltre, consente di distribuire i compiti su diversi frame, sospendendo l'esecuzione e restituendo il controllo al sistema per poi riprendere dal punto in cui si era interrotto nel frame successivo [19].

Codice 5.2: Snippet di ScenarioBenchmarkHandler.cs

---

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

        // Termina benchmarking se si è raggiunto il numero di snapshot
        if (AverageFPS.Count >= _settings.benchmarkNumSnapshots)
        {
            ShowResults();
            ResetProgress();
            return;
        }

        // Invoca la coroutine Benchmark
        StartCoroutine(Benchmark());
    }

    private IEnumerator Benchmark()
    {
        _isBenchmarking = true;

        // Inizia il profiling dall'API PerformanceProfiler
        PerformanceProfiler.Instance.StartProfiling();

        // Sospensione della coroutine per la durata del benchmark
        yield return new
        {
            WaitForSecondsRealtime(_settings.benchmarkSnapshotDuration);
        };

        // Termina il profiling e aggiunge il risultato alla lista AverageFPS
        AverageFPS.Add(PerformanceProfiler.Instance.StopProfiling());

        // Se non si è raggiunto il numero di snapshots, incrementa il numero di
        // entità
        if (AverageFPS.Count >= _settings.benchmarkNumSnapshots)
            ScenarioSettingsAPIs.SetNumEntities(0);
        else
            ScenarioSettingsAPIs.SetNumEntities(
                ScenarioSettingsAPIs.GetNumEntities() +
                _settings.benchmarkIncrement);

        // Ricarica la scena
        SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex);
    }
}
```

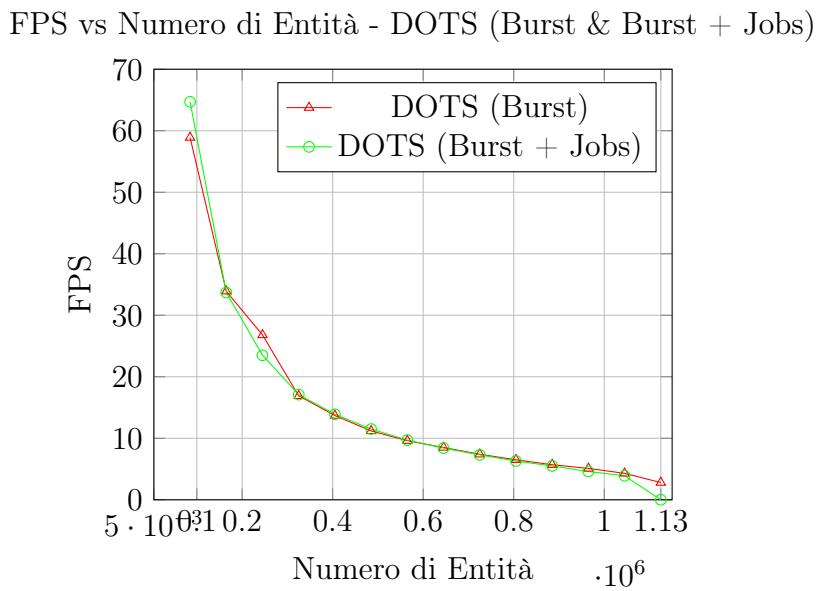
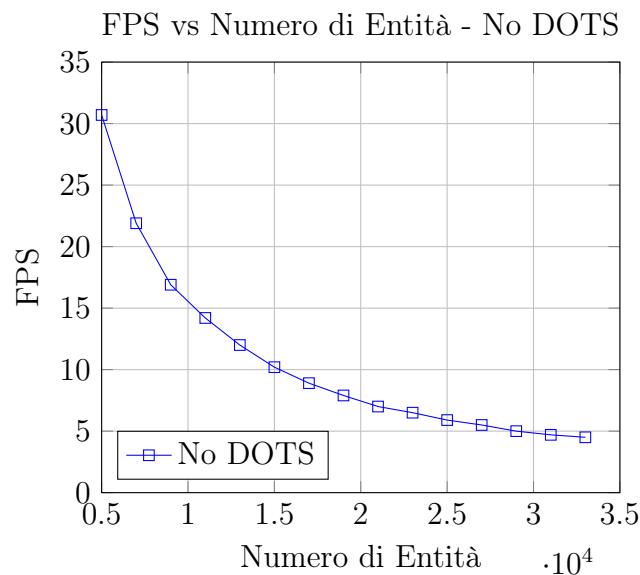
---

## 5.2 Risultati del Benchmarking

Il benchmarking è stato effettuato come Proof of Concept con un Macbook Pro chip M3 Pro, memoria 36GB, macOS Sonoma (14.6.1). In seguito sono riportati i risultati per i tre scenari esposti.

Sono presentati i grafici separati tra NoDOTS e DOTS e DOTS + Jobs per rendere il confronto più leggibile, dato che la configurazione NoDOTS gestisce un numero di gran lunga inferiore di GameObjects rispetto alle altre configurazioni.

### 5.2.1 Scenario 1



Dai dati raccolti, si osserva che, usando la programmazione tradizionale, si riesce a gestire un massimo di circa 33.000 GameObjects prima che il frame rate scenda sotto i 5 FPS, rendendo il progetto non più utilizzabile.

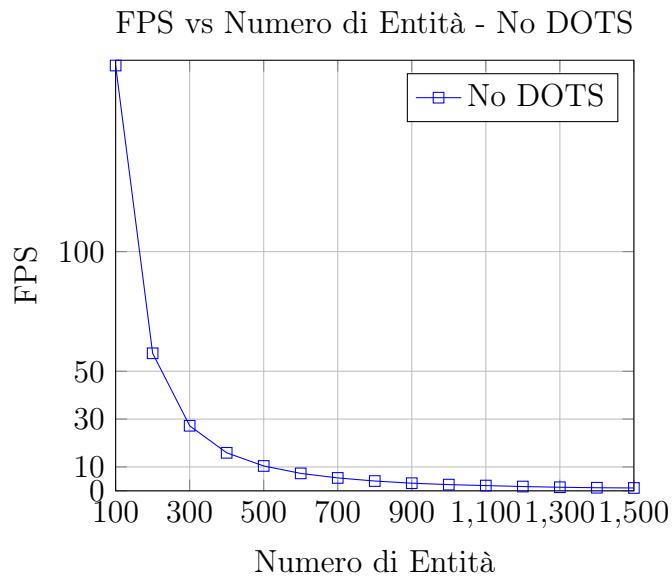
Nella ottimizzazione con Burst e quella con Burst + Jobs, il sistema è in grado di gestire un numero significativamente maggiore di entità, fino a 1.125.000. Il frame rate scende sotto i 5 FPS a partire da circa 885.000 entità.

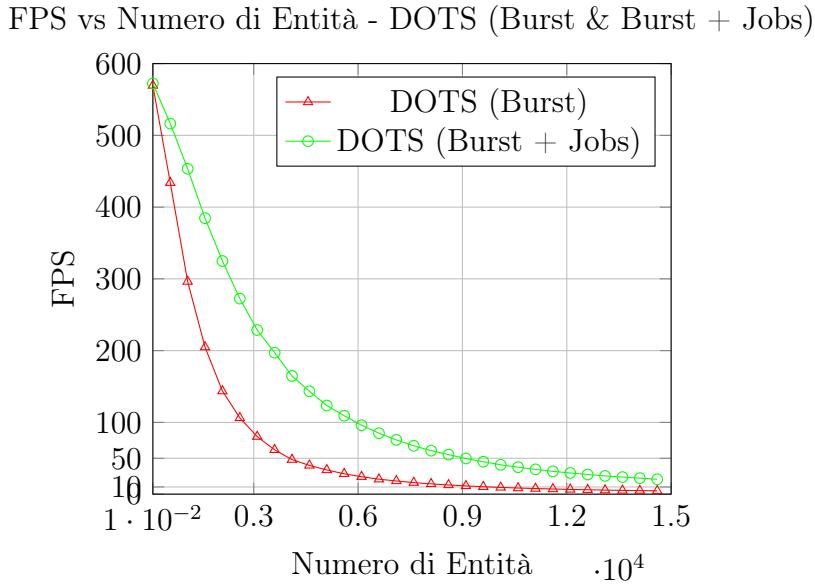
Considerando la soglia di fluidità pari a 25 FPS, si nota che senza DOTS questo limite viene raggiunto con un numero di entità pari a circa 6.500. In confronto, nelle altre ottimizzazioni, la fluidità si mantiene sopra i 25 FPS fino a circa 245.000 entità.

Infine, per questo particolare scenario, la parallelizzazione con i Jobs non apporta miglioramenti rilevanti rispetto al solo uso di DOTS; è probabile che ciò che causa latenza sia il rendering di un numero così massivo di entità, piuttosto che il calcolo della distanza che ogni entità effettua ad ogni frame; l'unica differenza tra le due ottimizzazioni è che il calcolo è effettuato in parallelo nell'ottimizzazione con il Job system, perciò non si osserva, per questo scenario particolare, il miglioramento atteso.

### 5.2.2 Scenario 2

Si nota che il processo di benchmarking riporta il numero di sfere presente nella scena; in questo particolare scenario ogni sfera ha un'ulteriore entità 'Link'. Il numero totale di entità da renderizzare è, quindi, il doppio di quello riportato.



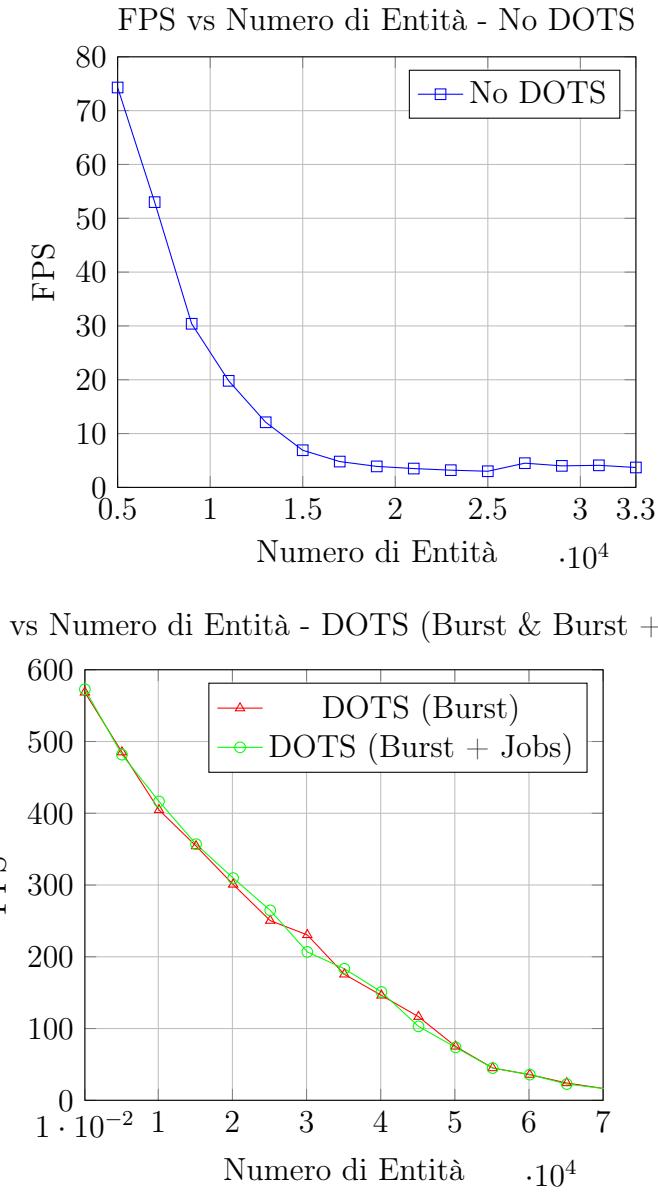


Dai dati raccolti, si osserva che senza DOTS, l'aumento esponenziale dei calcoli effettuati da ogni sfera causa un drastico abbassamento delle prestazioni man mano che il numero di entità cresce. Il sistema riesce a gestire solo fino a circa 200 sfere prima che il frame rate scenda sotto la soglia dei 25 FPS, rendendo il progetto non più fluido. Oltre le 500 sfere, il frame rate scende rapidamente sotto i 10 FPS, rendendo la simulazione inutilizzabile.

Con le ottimizzazioni DOTS, il sistema è in grado di gestire un numero significativamente maggiore di entità. Le prestazioni decrescono esponenzialmente con l'aumento delle entità, come previsto, poiché ogni GameObject esegue operazioni  $O(numEntities^2)$ . Il frame rate scende sotto i 25 FPS a partire da circa 6.600 entità nel caso del solo Burst Compiler e da circa 13.100 entità con l'uso combinato di Jobs e Burst Compiler. Il limite di 5 FPS viene raggiunto a circa 14.600 entità sia con il solo Burst Compiler sia con l'uso di Jobs

In questo scenario, l'ottimizzazione con Jobs risulta molto più efficace, poiché la complessità computazionale (dovuta ai calcoli di distanza) può essere suddivisa tra più threads, migliorando significativamente le prestazioni e consentendo di mantenere la fluidità fino a un numero di entità di gran lunga maggiore rispetto al solo uso di Burst.

### 5.2.3 Scenario 3



Dai dati raccolti, si osserva che nella configurazione senza DOTS, le prestazioni calano progressivamente man mano che aumenta il numero di sfere nella simulazione fisica. Il sistema riesce a gestire solo fino a circa 9.000 sfere prima che il frame rate scenda sotto la soglia dei 25 FPS. Oltre questo numero, il frame rate scende rapidamente, rendendo la simulazione inutilizzabile oltre i 15.000 GameObjects, dove il frame rate è inferiore a 10 FPS.

Nella ottimizzazione con DOTS in quella con DOTS + Jobs, si nota una gestione molto più efficiente degli oggetti fisici: la soglia dei 25 FPS viene raggiunta intorno alle 65.100 entità.

Il limite dei 5 FPS viene raggiunto a circa 70.100 entità sia nel caso del solo Burst Compiler che con l'aggiunta dei Jobs; ciò suggerisce che, in questo scenario, l'ottimizzazione con Jobs non porta a miglioramenti significativi rispetto al solo Burst Compiler, poiché il carico di lavoro fisico legato alla gestione delle collisioni e delle dinamiche fisiche è probabilmente limitato dalla capacità di calcolo seriale delle operazioni fisiche piuttosto che dalla loro distribuzione sui threads.

Tuttavia, le ottimizzazioni DOTS risultano comunque estremamente efficaci rispetto alla configurazione senza DOTS, consentendo di simulare un numero di gran lunga maggiore di entità fisiche senza compromettere la fluidità della simulazione.

# Capitolo 6

## Conclusione e Sviluppi Futuri

### 6.1 Sintesi dei Risultati Analizzati

La seguente tabella riassume i dati ricavati e analizzati nel capitolo precedente:

Scenario	Ottimizzazione	Entità a 25 FPS	Entità a 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%)

Tabella 6.1: Numero di entità gestite a 25 FPS e a 5 FPS per i tre scenari con le diverse ottimizzazioni con la percentuale di miglioramento rispetto a No DOTS accanto a ciascun valore.

Nel complesso, l'uso delle librerie DOTS risulta decisamente più efficiente quando è necessario costruire scene complesse con un gran numero di entità. Il Job System, in particolare, migliora notevolmente le prestazioni se gli scripts delle singole entità richiedono calcoli intensi e se questi sono facilmente parallelizzabili, cioè indipendenti fra loro.

### 6.2 Criticità di Unity DOTS

Nonostante l'evidente miglioramento delle prestazioni, dopo aver sviluppato il progetto di benchmarking, si possono evidenziare alcune problematiche riscontrate, che non si sarebbero verificate, se fosse stato utilizzato MonoBehaviour:

- **Incompatibilità con l'Editor di Unity** La base di tutte le criticità è che l'Unity Engine è stato scritto e ampliato con l'OOP, perciò, come evidenziato in precedenza, molte caratteristiche dell'editor sono incompatibili. L'unico modo, al momento, per attenuare queste incompatibilità, è quello di inserire nella scena un GameObject tradizionale, che segue la posizione della rispettiva entità del mondo ECS, ma può presentare caratteristiche MonoBehaviour (come il sistema di Animazione); ciò, negli scenari analizzati in precedenza, avrebbe limitato le prestazioni, poichè il principale collo di bottiglia sarebbe stato il numero di GameObjects presenti nella scena.
- **Codice Authoring boilerplate** Ogni componente di un'entità, che è piazzata nella scena dall'Unity Editor e non istanziata soltanto a runtime, necessita di un corrispettivo script authoring che, nella maggior parte dei casi, riporta le stesse proprietà del componente; ciò aumenta il tempo di scrittura del codice e il numero di classi. Questo problema può essere facilmente risolto attraverso features dell'IDE utilizzato, le quali possono già automaticamente generare scripts MonoBehaviour di authoring per il componente richiesto.
- **Sistema di collisioni incompleto** Nell'ultima versione di Unity DOTS, al momento della scrittura di questa trattazione, con il componente Physics Shape di Unity Physics non si ha la possibilità di impostare la modalità di controllo delle collisioni (Discrete o Continous). Il controllo discreto aggiorna le collisioni dell'oggetto per ogni frame fisico e potrebbe portare a risultati inaspettati, come oggetti con velocità elevata che passano attraverso muri sottili. Il controllo continous, invece, consuma più risorse, ma controlla tutti i punti tra la posizione dell'oggetto e la posizione nel frame fisico precedente per trovare una collisione. Si parlerà di come si potrebbe risolvere questo problema nella sezione successiva [20].

Nel complesso, le criticità di Unity DOTS sono dovute anche al suo recente rilascio, ma queste risultano accettabili anche in ambienti di sviluppo in produzione, se si usano queste librerie per ottimizzare parti specifiche e computazionalmente rilevanti del sistema applicativo, piuttosto che l'intero progetto.

### 6.3 Ampliamento Ricerca e Sviluppi Futuri

Per concludere, è possibile ampliare la trattazione e il progetto in futuro con lo studio di ulteriori librerie di Unity DOTS ed eventualmente con nuovi scenari di benchmarking:

- Si potrebbe analizzare **Entities Graphics** nel dettaglio, la libreria che permette il rendering ottimizzato di un grande numero di entità sullo schermo, e

personalizzare la pipeline di rendering per intervenire direttamente sul flusso dei dati verso la GPU, ottimizzando ulteriormente il rendering in base alle esigenze specifiche del progetto.

- La libreria **Netcode for Entities** permette di trasmettere e di sincronizzare entità nel mondo ECS, utile in giochi con molti giocatori o in simulazioni distribuite; il servizio multiplayer non è *peer to peer*, ma presenta un *authoritative server*. La libreria permette ai clients di eseguire una "predizione" dello stato futuro del gioco, riducendo la percezione della latenza della rete, mentre il server corregge eventuali discrepanze con lo stato reale nello sfondo [3].

Inoltre, lo scenario 3 utilizza la libreria di Unity Physics per il controllo delle collisioni, la quale, come riportato nel paragrafo precedente, presenta delle criticità. Si potrebbe riscrivere il codice ECS, utilizzando sistemi che eseguono *raycasting* dalle diverse entità per trovare le collisioni; questo metodo potrebbe essere più facilmente parallelizzabile e, quindi, oltre a rendere la simulazione fisica più accurata, potrebbe anche aumentare le prestazioni con l'utilizzo del Job System.

Infine, al momento della stesura di questa tesi, Unity 6 non è ancora stato rilasciato, ma la data di rilascio è stata annunciata nel *Unite 2024 Keynote* e sarà il 17 Ottobre 2024. Dato che questa versione porterà migliorie sostanziali delle librerie Unity DOTS (tra le tante, una maggiore compatibilità con l'Unity Editor e un generale miglioramento delle prestazioni), risulterebbe utile aggiornare il progetto all'ultima versione ed effettuare nuovamente il benchmarking degli scenari analizzati [21].



# Bibliografia

- [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*

## Bibliografia

---

- [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*