# UNIVERSITÀ DEGLI STUDI DI MILANO

FACOLTÀ DI SCIENZE E TECNOLOGIE




CORSO DI LAUREA MAGISTRALE IN INFORMATICA


# SCALABILITÀ DI A* ALL'INTERNO DI UN GAME ENGINE


Relatore: Prof. Dario Maggiorini
Correlatore: Prof. Giuliano Grossi

Tesi di Laurea di:
Manuel Salvadori
Matr. Nr. 886725



anno accademico 2017-2018

# Indice

# Elenco delle figure

* per gentile concessione di Amit Patel. `http://theory.stanford.edu/~amitp/`

# Elenco dei sorgenti

# Capitolo 1

# Introduzione

## 1.1 Obiettivi

Col passare del tempo i videogame sono diventati simulazioni sempre più complesse. Una conseguenza di ciò è che il carico computazionale aumenta proporzionalmente alla complessità della simulazione. Allo stesso tempo gli utenti si aspettano un realismo e una fluidità sempre maggiori. Lo standard attuale prevede un framerate nell'ordine dei 60 fps, che ammette quindi un tetto massimo di soli 16 ms per il calcolo di ogni frame (Durante la trattazione per *framerate* si intenderà sempre la frequenza di aggiornamento dell'applicazione, e non della frequenza di aggiornamento dello schermo che sarà invece indicata col termine *refreshrate*). Questo limite è il frutto di un compromesso tra carico computazionale richiesto e fluidità della simulazione percepita dall'occhio umano. L'evoluzione tecnologica dell'hardware da sola non basta a sopperire al continuo aumento del carico computazionale. Diventa quindi imperativo saper sfruttare al meglio le risorse computazionali a disposizione utilizzando tecniche adeguate e ottimizzando il più possibile.

In questa tesi ci si focalizzerà su un aspetto in particolare, in ambito intelligenza artificiale, che da sempre richiede uno sforzo computazionale elevato, ovvero quello del pathfinding, nel contesto di un game engine. Il pathfinding risolve il problema del trovare un cammino minimo percorribile tra due posizioni in un ambiente popolato da ostacoli. Formalmente questo si traduce nella ricerca del cammino minimo tra due nodi di un grafo, dove il grafo è la rappresentazione formale dell'ambiente considerato. Esistono numerosi algoritmi che risolvono

il problema dei cammini minimi; quello più utilizzato, per vari motivi che verranno discussi nel Capitolo 3, nell'industria del videogame è A* (pronunciato *A star*), una variante euristica del conosciutissimo algoritmo di Dijkstra [1].

Il pathfinding diventa particolarmente oneroso se è necessario calcolare un percorso per molti agenti contemporaneamente, all'aumentare dei quali il costo potrebbe diventare proibitivo. Questa tesi si prefigge l'obiettivo di aumentare la scalabilità del pathfinding sfruttando le potenzialità offerte da una nuovissima tecnologia, messa a disposizione dal popolare game engine Unity [2]. Questa tecnologia, dal nome Unity ECS [3], consente di sfruttare al meglio tutta la potenza offerta dalle moderne CPU, suddividendo il carico di lavoro attraverso tutti i core disponibili. Si analizzeranno infine le prestazioni ottenute, comparandole con le prestazioni di una versione sequenziale (single threaded) analoga.

La tesi è organizzata come segue: nel Capitolo 2 sarà esaminato lo stato dell'arte. Nel Capitolo 3 verrà descritto formalmente il problema del pathfindig, illustrandone i principali algoritmi risolutivi. Nel Capitolo 4 si introdurranno il data-oriented design, il pattern entity-component-system e la tecnologia Unity ECS, descrivendone nel dettaglio i principi architetturali ed il funzionamento. Nel Capitolo 5 verrà descritto nel dettaglio il progetto sviluppato. Nel Capitolo 6 verranno discussi i risultati ottenuti e infine nel Capitolo 7 verranno tratte le conclusioni e discussi i possibili sviluppi futuri.

# Capitolo 2

# Stato dell'arte

Al meglio delle nostre ricerche, non sono noti lavori a livello accademico riguardanti l'applicazione del pattern Entity-Component-System al problema del pathfinding. Tuttavia la letteratura è piena di lavori riguardanti la possibilità di sfruttare la programmazione parallela GPGPU, su framework quali CUDA[4] sviluppato da NVIDIA [5], per ottenere miglioramenti di prestazione significativi sia utilizzando A* che utilizzando delle sue varianti.

In particolare possiamo ricordare Avi Bleiweiss che nel 2008 [6] ha proposto un approccio al pathfinding multi agente altamente parallelizzato su GPU usando CUDA. In questo lavoro viene considerato un grafo sparso rappresentato tramite liste di adiacenza, mentre l'euristica utilizzata all'interno di A* è la distanza euclidea. Le richieste di ogni agente sono prese in carico da singoli thread che computano il cammino minimo in modalità SIMD. Gli svantaggi di un implementazione GPGPU sono i trasferimenti tra memorie host e device, necessari in quanto la GPU dispone di una sua memoria (device) separata dalla memoria principale usata dalla CPU (host), oltre che eventuali divergenze all'interno dei warp (all'interno del framework CUDA, un *warp* è un gruppo di 32 thread che eseguono la stessa istruzione). Nonostante tutto, in questo lavoro si ottiene un guadagno di 24 volte rispetto alla versione sequenziale.

Nel 2013 Caggianese e Erra [7] propongono un approccio logicamente differente ma tecnologicamente simile. Il framework utilizzato è sempre CUDA, quindi il parallelismo è sempre su GPU. L'algoritmo proposto prende il nome di PHA* (Parallel Hierarchical A*) e lavora solo su grafi a griglia (grid-based graph o tile-based graph). La mappa viene prima divisa in blocchi quadrati di egual dimensione. Ogni blocco è a sua volta suddiviso in regioni di celle contigue.

Da questa astrazione di alto livello è ricavato un nuovo grafo. Il pathfinding si divide poi in due passaggi: nel primo viene utilizzato Dijkstra per trovare un percorso sul grafo di alto livello; ogni thread si occupa di tutti gli agenti accomunati da nodi di partenza e arrivo uguali. Nel secondo passo, per ogni agente viene lanciato un thread che si occupa di computarne il percorso minimo, attraverso A* (utilizzando come euristica la distanza Manhattan), nel grafo di basso livello. I percorsi trovati non sono ottimali, ma la loro lunghezza è maggiore del 10/27% a seconda della dimensioni di blocco. Sempre in relazione alle dimensioni di blocco è il guadagno in termini di tempo.

Tutt'altro approccio, invece, è quello formulato da Brand e Bidarra [8] nel 2012. Nel loro lavoro viene proposto un algoritmo, PRS (Parallel Ripple Search), che consente la parallelizzazione del calcolo di A* di un singolo percorso, invece che della parallelizzazione del calcolo di più percorsi come nei casi precedenti. La parallelizzazione viene poi portata in essere su più core di CPU dal classico multi-threading (in questo caso è stato utilizzato, per motivi di performance, il linguaggio C++). Il funzionamento di PRS consiste nel trovare un percorso tra i nodi start e goal in grafo di più alto livello. Successivamente, ad ogni core viene assegnato il compito di calcolare, tramite A*, i percorsi tra i vari waypoint trovati precedentemente, nel grafo di basso livello. In questo modo si può ottenere un guadagno di 2,5-10 volte per percorsi lunghi almeno 200 nodi; al di sotto di tale soglia l'overhead introdotto fa perdere tutto il guadagno in termini di prestazioni. Inoltre i path prodotti non sono ottimali ma sono in media più lunghi del 4%.

Nel 2015 Zhou e Zeng [9] elaborano un altro approccio di parallelizzazione dei singoli cammini, ma a differenza di Brand e Bidarra lo fanno su GPU. In particolare, la strategia adottata è quella di parallelizzare la computazione delle euristiche in primo luogo e in secondo luogo di parallelizzare l'espansione di A*. Nel dettaglio, per ogni nodo, A* valuta i costi $g$ e $f$ di tutti i nodi appartenenti al vicinato. La valutazione dei costi dei vicini è indipendente, quindi potenzialmente parallelizzabile. In questo caso, il miglioramento delle prestazioni rispetto alla versione sequenzale è nell'ordine di 30 volte, mantenendo l'ottimalità dei percorsi trovati.

Sono state esplorate anche soluzioni ibride, come quella proposta nel 2011 da Demeulemeester, Hollemeersch, Mees, Pieters, Lambert e Van de Walle [10]. L'idea è quella di fondere il classico pathfinding con le tecniche di *flow field* per permettere il pathfinding di migliaia di agenti simultaneamente. Di nuovo viene considerata una suddivisione della mappa di gioco in tile da cui è poi ricavato il grafo usato per il pathfinding. In un primo step vengono selezionati

i nodi start e goal che sono in prossimità degli agenti. A* è poi utilizzato per calcolare i percorsi necessari. Successivamente i percorsi vengono espansi in una regione di interesse che copre tutte le tile percorse; solo in questa regione viene poi generato il campo di flusso utilizzato in seguito dalle unità per il movimento. Tecnologicamente, la prima parte di pathfinding classico è parallelizzata su GPU così come la seconda, che è gestita come image processing, ovvero la regione trovata è trattata come un immagine su cui vengono eseguite delle operazioni di filtraggio, e quindi altamente parallelizzabile. Il framework utilizzato, ancora una volta è quello di NVIDIA CUDA.

Tutti gli approcci al pathfinding visti finora danno grandi aumenti delle performance e quindi della scalabilità. Tuttavia, ad eccezione del Parallel Ripple Search, gli approcci basati su GPU hanno tutti lo svantaggio di utilizzare preziose risorse che sarebbe preferibile siano dedicate al render grafico. Inoltre, da un punto di vista pratico, i programmatori devono adottare delle tecniche per applicare tali approcci di individuazione dei percorsi senza gravi interruzioni nella pipeline di rendering. Questo fatto rende poco pratico utilizzare tali soluzioni soprattutto nel contesto dei game engine, in parte anche per la limitazione del framework CUDA di funzionare solo ed esclusivamente su schede grafiche NVIDIA.

# Capitolo 3

# Pathfinding

Il pathfinding è un elemento chiave dell'intelligenza artificiale (AI) nei videogame. Grazie ad esso vediamo le unità muoversi raggiungendo la loro destinazione quando si gioca a Starcraft[11]. Permette ai nemici in Destiny[12] di correre attorno alle barricate per arrivare al giocatore. Il pathfinding consente ai personaggi controllati dal computer di muoversi verso destinazione efficientemente evitando ostacoli e pericoli. Nonostante un significativo progresso visto in questi anni, il problema del pathfinding continua ad attrarre le attenzioni dei ricercatori. Questo fatto si spiega in parte nelle richieste di performance che le implementazioni del pathfinding devono soddisfare in un videogame, dove le risorse sono limitate. Inoltre, il problema non riguarda solo il campo dell'intelligenza artificiale in ambito videogame, ma si applica a numerosi altri campi quali la robotica, il controllo aereo, le applicazioni di navigazione stradale e altro ancora. Il contesto su cui ci si focalizzerà in questa tesi è comunque quello dei videogame. In questo caso, generalmente, il problema del pathfinding può essere definito in uno spazio bidimensionale. Anche per videogame 3D il problema del pathfinding può essere ridotto a un problema bidimensionale, in quanto usualmente il movimento è consentito solo sul terreno. In altri casi, come nelle simulazioni di volo, il pathfinding è invece definito in uno spazio tridimensionale. Le tecniche di base e gli algoritmi utilizzati sono comunque gli stessi. In questo capitolo verrà esaminato formalmente questo problema, descrivendone i principali algoritmi risolutivi, le rappresentazioni della mappa tipiche e quali sono gli approcci più comunemente utilizzati nell'industria del videogame. Sarà inoltre preso in considerazione il problema della scalabilità.

## 3.1 Definizione

In generale per pathfinding si intende il ritrovamento di un percorso ottimale tra due punti di un ambiente. Esso è strettamente correlato con il problema dei cammini minimi nella teoria dei grafi, che si può descrivere come la ricerca di un cammino tra due nodi di un grafo pesato tale che la somma dei pesi degli archi che lo costituiscono sia minima. Formalmente: sia dato un grafo $G$ con $V$ e $E$ gli insiemi dei suoi vertici e archi; sia $e_{i,j} \in E$ l'arco incidente ai vertici $v_i$ e $v_j \in V$ e sia $f : E \to \mathbb{R}$ la funzione che associa ad ogni arco il suo peso. Il cammino minimo tra $v$ e $v'$ è il cammino $P = (v_1, v_2, v_3, \cdots, v_n)$ dove $v_1 = v$ e $v_n = v'$ e dove $v_i$ è adiacente a $v_{i+1}$ per $1 \leq i < n$, tale che è minima la somma $\sum_{i=1}^{n-1} f(e_{i,i+1})$.

Si può infatti modellare l'ambiente considerato in un grafo, i cui nodi rappresentano delle posizioni chiave e i cui archi abbiano come peso il costo in termini di distanza percorsa, di sicurezza o di qualsiasi altro parametro il design di gioco richieda. I percorsi trovati dal pathfinding saranno ottimali rispetto al parametro scelto.

## 3.2 Algoritmi

L'idea di base che accomuna tutti gli algoritmi che risolvono il problema del pathfinding è quella che, partendo dal nodo iniziale, vengono valutati i nodi nel suo vicinato (per vicinato si intende l'insieme di tutti i nodi adiacente al nodo dato). Iterativamente poi, per ogni nodo vicino vengono valutati i relativi vicinati. Questo processo viene chiamato *espansione* della frontiera, dove per *frontiera* si intende l'insieme dei nodi in fase di valutazione. Inizialmente la frontiera è costituita dal solo nodo di partenza; successivamente si iterano i seguenti passaggi, fintanto che la frontiera contiene dei nodi:

1. seleziono e rimuovo un nodo dalla frontiera;

2. espando la frontiera considerando il vicinato del nodo corrente. Qualsiasi nodo che non sia stato ancora visitato viene aggiunto alla frontiera, mentre il nodo corrente è aggiunto all'insieme dei nodi visitati.

Questo processo ci indica solamente come viene esplorato tutto il grafo. Tuttavia, tenendo traccia dei costi dei archi attraversati è possibile ricostruire il percorso dal nodo iniziale al nodo finale. Inoltre è possibile specificare un condizione di uscita anticipata: invece che continuare ad iterare finché la frontiera non è completamente vuota, si itera solo fino al ritrovamento del

nodo goal. Quello che differenzia i vari algoritmi, che verranno descritti a breve, è come viene calcolato il costo e come viene selezionato il prossimo nodo; in altre parole in che modo la frontiera si espande. Negli esempi in pseudocodice che seguiranno, ma più in generale nella letteratura, la frontiera e l'insieme dei nodi già valutati prendono il nome di *open set* e *closed set*. Vengono ora esposti tre principali algoritmi: l'algoritmo di Dijkstra, l'algoritmo gBFS (greedy best-first search) e infine l'algoritmo A*.

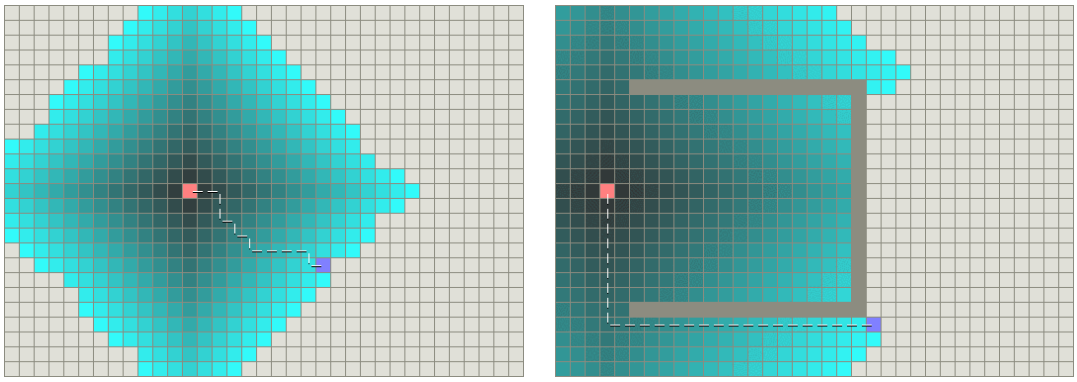
### 3.2.1 Dijkstra

Questo algoritmo è stato descritto per la prima volta da Edsger W. Dijkstra nel 1956 ma fu pubblicato solo tre anni più tardi nel 1959 [1]. L'algoritmo di Dijkstra è il più importante algoritmo di pathfinding ottimale (cioè garantisce di trovare il percorso più breve), in quanto formula la strategia base che tutti gli altri algoritmi di ricerca dei cammini minimi usano.

```
OpenSet = {}
G_costs = {}
cameFrom = {}

for each vertex v in Graph:
    G_costs[v] = INFINITY
    cameFrom[v] = UNDEFINED
    add v to OpenSet

G_costs[start] = 0

while OpenSet is not empty:
    u = vertex in OpenSet with min G_costs[u]

    remove u from OpenSet

    for each neighbor v of u:
        alt = G_costs[u] + cost(u, v)

        if alt < G_costs[v]:
            G_costs[v] = alt
            cameFrom[v] = u

return G_costs[], cameFrom[]
```

Sorgente 1: Algoritmo di Dijkstra

L'algoritmo originale, in realtà, consente di trovare i cammini minimi da un nodo iniziale verso tutti gli altri nodi del grafo; per limitare la ricerca al solo cammino verso un nodo target è necessario inserire un ulteriore condizione di uscita: se il nodo corrente è il nodo finale



(a) Espansione uniforme della frontiera (b) I cammini trovati sono ottimali

Figura 1: Algoritmo di Dijkstra su griglia. L'intensità del colore azzurro rappresenta il costo G.

allora la ricerca ha termine. Una condizione per il funzionamento dell'algoritmo richiede che i pesi degli archi del grafo siano non negativi. La complessità temporale nel caso peggiore dell'algoritmo originale è $\mathcal{O}(N^2)$ dove $N$ è il numero totale dei nodi del grafo. Un'implementazione più rapida è stata scoperta più tardi nel 1987 da Fredman e Tarjan [13]; questa nuova versione dell'algoritmo utilizza una coda di priorità (Fibonacci heap) per memorizzare l'open set. In questo caso si raggiunge una complessità nel caso peggiore di $\mathcal{O}(E + N \log N)$ con $E$ numero degli archi. L'algoritmo di Dijkstra prende in input un grafo, un nodo start e un nodo target. L'espansione della frontiera si propaga uniformemente in tutte le direzioni partendo dal nodo start, visitando i nodi che gli sono più vicini. Non appena viene raggiunto il nodo target è possibile tracciare il cammino minimo dal nodo target a ritroso fino al nodo iniziale. Un importante concetto dell'algoritmo di Dijkstra è il calcolo del valore dei costi $G$. Ad ogni nodo del grafo viene associato un costo $G$. Il valore del costo $G$ del nodo iniziale è zero, per ovvie ragioni. Il costo $G$ dei suoi vicini è calcolato come somma del costo $G$ del nodo corrente (nel caso della prima iterazione coincide col nodo iniziale) con il peso dell'arco che connette i due nodi:

$$G(n_i) = G(n_j) + f(e_{i,j})$$

dove i nodi $n_i$ e $n_j$ sono connessi dall'arco $e_{i,j}$. Il processo è ripetuto iterativamente. La frontiera si espande poi valutando il nodo nell'open set che ha il costo $G$ minore: qui si capisce l'importanza di una struttura dati come la coda di priorità per migliorare la complessità temporale complessiva dell'algoritmo.

Nelle comuni presentazioni dell'algoritmo di Dijkstra, inizialmente tutti i nodi sono inseriti all'interno della cosa di priorità (vedi Sorgente 1). Tuttavia, questo non è strettamente necessario: l'algoritmo può cominciare con la coda di priorità contenente solo un elemento (il nodo iniziale) e inserire i nuovi elementi solo al momento della prima visita. Questa variante ha lo stesso limite nel caso peggiore della variante originale, ma mantiene una coda di priorità più piccola in pratica, velocizzando le operazioni interne alla coda. Inoltre, non inserire tutti i nodi del grafo rende possibile estendere l'algoritmo al ritrovamento dei cammini minimi da un singolo nodo al più vicino di un insieme di nodi target in un grafo infinito o in un grafo che sia troppo grande per essere rappresentato in memoria. Nella letteratura in intelligenza artificiale questo algoritmo prende il nome di Uniform Cost Search [14][15] e può essere espresso in pseudocodice come:

```
openSet = PriorityQueue()
openSet.put(start, 0)
closedSet = {}
cameFrom = {}
G_cost = {}
cameFrom[start] = None
G_cost[start] = 0

while not openSet.empty():
   current = openSet.get()
   closedSet.put(current)

   if current == goal:
      break

   for next in neighbors(current):
      new_cost = G_cost[current] + cost(current, next)

      if next not in closedSet or new_cost < G_cost[next]:
         G_cost[next] = new_cost
         priority = new_cost
         openSet.put(next, priority)
         cameFrom[next] = current
```

Sorgente 2: Algoritmo Uniform Cost Search

L'algoritmo di Dijkstra ci garantisce il ritrovamento di un cammino ottimo. Tuttavia la porzione di grafo esplorata nel raggiungimento dell'obiettivo è molto grande e, soprattutto nel contesto dei videogame, non è abbastanza ottimizzata in termini di tempo. Il prossimo algoritmo tenta infatti di risovere questo problema.

### 3.2.2 Greedy Best-First Search

L'espansione della frontiera nell'algoritmo di Dijkstra, come detto in precedenza, è uniforme in tutte le direzioni. Un approccio migliore sarebbe quello di espandere la frontiera privilegiando la direzione verso il nodo target. Questo può essere fatto sostituendo, nel calcolo del costo di ogni nodo, la distanza percorsa con la distanza rimanente al nodo goal. Questo approccio è chiamato *greedy best-first search* [16] o *pure heuristic search* [17]. La distanza rimanente all'obiettivo non è nota, si deve quindi ottenere una stima di essa. In altre parole deve essere definita una funzione euristica che restituisca questa stima.

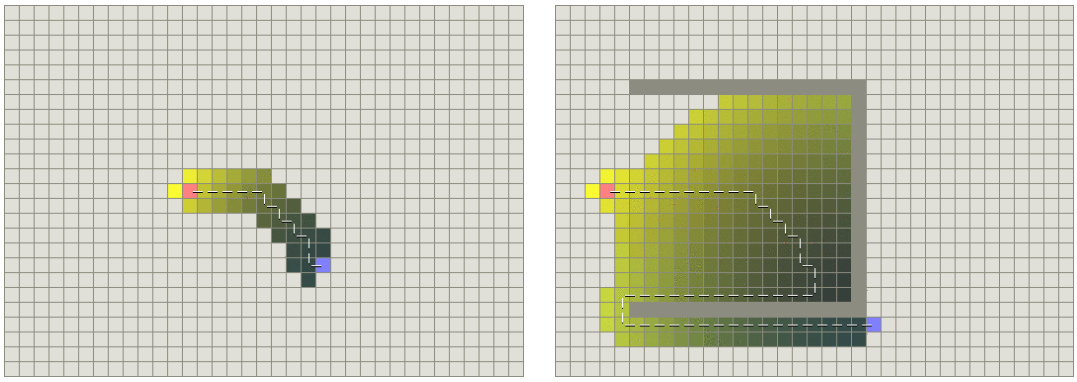
```
openSet = PriorityQueue()
openSet.put(start, 0)
closedSet = {}
cameFrom = {}
cameFrom[start] = None

while not openSet.empty():
   current = openSet.get()
   closedSet.put(current)

   if current == goal:
      break

   for next in graph.neighbors(current):
      if next not in closedSet:
         priority = heuristic(goal, next)
         openSet.put(next, priority)
         cameFrom[next] = current
```

Sorgente 3: Algoritmo Greedy Best-First Search

L'algoritmo Greedy Best-First-Search funziona esattamente come lo Uniform Cost Search, ad eccezione del fatto che si ha a disposizione una stima euristica di quanto ogni nodo sia lontano dal nodo target. Invece che selezionare il nodo col costo $G$ minore, ovvero il nodo più vicino al punto di partenza, l'algoritmo seleziona il nodo più vicino all'obiettivo. Il costo associato ad ogni nodo $n$ in questo caso viene chiamato costo $H$:

$$H(n) = euristic(n, n_{goal})$$

Il concetto di stima euristica verrà discusso in maniera più approfondita nella prossima sezione. In generale, Greedy Best-First-Search trova la soluzione più velocemente rispetto all'algoritmo di Dijkstra perchè usa appunto la funzione euristica che guida il processo di ricerca verso



(a) Espansione mirata della frontiera (b) I cammini trovati non sono ottimali

Figura 2: Algoritmo Best-First Search su griglia. L'intensità del colore giallo rappresenta il costo H.

l'obiettivo in modo mirato. Ad esempio, se il nodo goal si trova a sud rispetto al nodo iniziale, Greedy Best-First-Search tenderà a esplorare tutti quei cammini che portano verso sud. In figura 2, il colore giallo rappresenta i nodi con un alto costo $H$ (maggior distanza al goal) mentre il nero rappresenta i nodi con un basso costo $H$ (maggior vicinanza al goal). Si può vedere che l'algoritmo trova il cammino molto più rapidamente ed esplorando una porzione di grafo minore in comparazione con l'algoritmo di Dijkstra (vedi figura 1). La porzione di grafo esplorata dipendo molto anche dalla funzione euristica adottata. La pecca più grande tuttavia, è quella che l'algoritmo non garantisce in alcun modo l'ottimalità dei cammini trovati. L'aggressività con cui Best-First Search si porta verso il nodo goal infatti, insieme alla mancata considerazione dello stato precedente (costo $G$) fanno in modo che in molte situazioni il cammino trovato sia più lungo rispetto a quello ottimo. Inoltre questo algoritmo è incompleto, non garantisce cioè il ritrovamento del percorso in quanto c'è il rischio di intraprendere un percorso che non conduce all'obiettivo. La complessità nel tempo dell'algoritmo Best-First Search è $\mathcal{O}(b^m)$ dove $b$ è il massimo fattore di diramazione e $m$ è la massima profondità dell'albero di ricerca.

Entrambi gli algoritmi visti finora hanno degli svantaggi che li portano ad essere poco desiderabili in un contesto applicativo come quello dei videogame. Da una parte l'algoritmo di Dijkstra garantisce il ritrovamento di cammini ottimali ma a un costo temporale troppo alto, mentre dall'altra Best-First Search risolve il problema della complessità temporale ma fallisce nel trovare percorsi ottimi. Vedremo ora un terzo algoritmo che tenta, riuscendoci, di unire i punti forza dei due precedenti senza averne i difetti: l'algoritmo A*.

### 3.2.3 A*

A* è stato sviluppato nel 1968 da Hart, Nilsson e Raphael come parte del progetto Shakey, il quale aveva come obiettivo la realizzazione di un robot mobile che potesse pianificare ogni sua azione autonomamente [18].

```
openSet = PriorityQueue()
openSet.put(start, 0)
closedSet = {}
cameFrom = {}
cameFrom[start] = None
Gcosts = {}
Gcosts[start] = 0

while not openSet.empty():
   current = openSet.get()
   closedSet.put(current)

   if current == goal:
      break

   for next in graph.neighbors(current):
      new_cost = Gcosts[current] + cost(current, next)
      if next not in closedSet or new_cost < Gcosts[next]:
         Gcosts[next] = new_cost
         Fcost = new_cost + heuristic(goal, next)
         openSet.put(next, Fcost)
         cameFrom[next] = current
```

Sorgente 4: Algoritmo A*

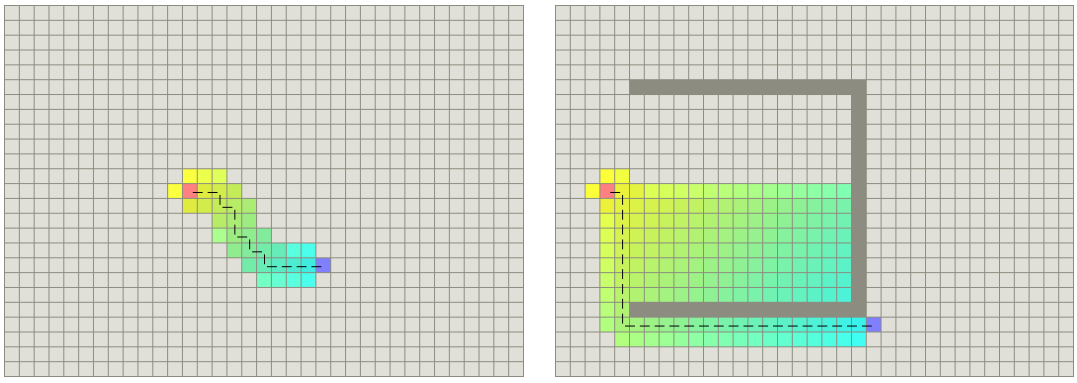
Come Best-First Search, A* è un algoritmo di ricerca informato, ovvero la selezione del nodo successivo avviene in base a una funzione di valutazione, che ne stima il costo dell'arrivo a destinazione. Come Dijkstra invece, A* tiene traccia dei costi dal nodo origine. Chiamiamo $F(n)$ la funzione di valutazione sul nodo $n$. Per gli algoritmi visti in precedenza abbiamo che:

$$Dijkstra : F(n) = G(n) \qquad GBFS : F(n) = H(n)$$

ricordando che $G(n)$ indica il costo esatto del percorso dall'origine al nodo $n$ e che $H(n)$ indica invece la stima euristica del costo per arrivare dal nodo $n$ a destinazione. La felice intuizione di Raphael è stata quella di combinare i costi $G(n)$ e $H(n)$:

$$F(n) = G(n) + H(n)$$

La funzione di valutazione in A* è quindi la somma dei costi $G(n)$ e $H(n)$. Il comportamento di A* dipende largamente dalla funzione euristica utilizzata. In particolare, per far si che A* garantisca il ritrovamento di cammini minimi, la valutazione euristica non deve mai essere una sovrastima. Un funzione euristica che non è mai sovrastima è detta euristica *ammissibile*. Può essere verificato che A* non considera più nodi di qualunque altro algoritmo di ricerca ammissibile, a meno che l'algoritmo alternativo non abbia una stima euristica più accurata. In questo senso A* è l'algoritmo computazionalmente più efficiente che garantisce la ricerca del percorso più breve. Nei casi limite, l'algoritmo degenera in quello di Dijkstra se la stima $H(n)$ è sempre zero, mentre se la funzione euristica è una sovrastima ed è molto grande rispetto al termine $G(n)$, ovvero $H(n) = \mathcal{O}(G(n))$, allora A* degenera nell'algoritmo Best-First Search. L'algoritmo, ad eccezione del calcolo della funzione di valutazione, ricalca quello dello Uniform Cost Search: ad ogni iterazione, il nodo con il costo $F(n)$ minore viene prelevato dall'open set. I costi $F(n)$ e $G(n)$ dei nodi appartenenti al vicinato sono aggiornati di conseguenza e in seguito gli stessi nodi vengono inseriti nell'open set. L'algoritmo continua ad iterare fin quando il nodo goal non viene raggiunto (o finché l'open set contiene elementi). Il costo $F(n)$ del nodo target è quindi il costo del percorso ottimo, dato che la stima $H(n)$ del nodo goal vale zero in un euristica ammissibile.



(a) Espansione mirata della frontiera (b) I cammini trovati non sono ottimali

Figura 3: Algoritmo A* su griglia. L'intensità del colore giallo rappresenta il costo H, quello azzurro il costo G.

Come si può vedere in Figura 3, A* mantiene l'espansione mirata della frontiera verso il nodo target, senza rinunciare all'ottimalità dei percorsi trovati. Lo "star", spesso denotato da un asterisco, *, si riferisce al fatto che A* usa una funzione euristica ammissibile. A* è inoltre

completo (a meno che ci siano infiniti nodi da esplorare), ovvero garantisce di trovare sempre un percorso tra i nodi start e goal se questo esiste.

Per quanto riguarda la complessità temporale deve essere fatta un'osservazione. Tipicamente, nell'informatica teorica, si misura il tempo di esecuzione in relazione al numero di archi e nodi esaminati. Nel caso peggiore c'è bisogno di esaminare l'intero grafo, quindi la complessità sarà $\mathcal{O}(|V| + |E|)$. Nella comunità AI tuttavia, si tende a misurare il tempo di esecuzione diversamente. Si considera spesso uno specifico tipo di grafo: un albero con un fattore di diramazione $b$. Inoltre, in molte situazioni, il grafo considerato è spesso infinito o molto grande. Tipicamente ci si sforza di evitare di esaminare tutto il grafo (è proprio questo uno dei punti di forza dell'algoritmo). Quindi, esprimere la complessità in termini di $|V|$ e $|E|$ ha poco senso: $|V|$ potrebbe essere infinito, e in ogni caso, non si ha in programma di esaminare tutto il grafo; conseguentemente tutto quello che conta è l'esatto numero di nodi esaminati. È perciò spesso più significativo considerare il tempo di esecuzione in termini del fattore di diramazione dell'albero di ricerca ($b$) e di profondità del nodo goal al suo interno ($d$). Tipicamente, una volta trovato il nodo target, l'algoritmo si ferma. In questo albero, se esaminiamo ogni nodo con profondità $\leq d$ prima di trovare il nodo goal, finiremo per esaminare $\mathcal{O}(b^d)$ nodi prima di fermarci. Quindi si può pensare di visitare un sottoinsieme del grafo con $|V| = \mathcal{O}(b^d)$ (dove ora $V$ include solo i nodi effettivamente visitati) e $|E| = \mathcal{O}(b^d)$ ($E$ include solo gli archi interessati), in altre parole : $\mathcal{O}(|V| + |E|) = \mathcal{O}(b^d + b^d) = \mathcal{O}(2b^d) = \mathcal{O}(b^d)$. In questo contesto $\mathcal{O}(b^d)$ è perciò più informativo rispetto $\mathcal{O}(|V| + |E|)$, seppur nessuna delle due notazioni sia errata.

Per via delle sue caratteristiche A* è la scelta preferita nelle applicazioni pratiche che abbiano il requisito di trovare un cammino in tempi brevi. Come già detto, queste caratteristiche dipendono molto dalle proprietà della funzione euristica scelta. Vediamo ora nel dettaglio queste proprietà ed alcuni esempi di funzioni euristiche comunemente adottate.

## 3.3 Stima euristica

### 3.3.1 Ammissibilità

Nell'ambito degli algoritmi di ricerca, in particolare negli algoritmi relativi al problema del pathfinding, una funzione euristica è detta *ammissibile* se essa non sovrastima mai il costo del

raggiungimento dell'obiettivo [16]. In altre parole, il costo stimato per raggiungere il goal non è più alto del più basso costo possibile dalla posizione corrente al goal:

$$H(n) \leq H^*(n) \quad \forall n$$

dove $H^*(n)$ indica il costo esatto del raggiungimento del goal a partire da $n$. Intuitivamente, una funzione ammissibile è "ottimistica", in quanto sottostima sempre il costo effettivo. Un algoritmo di ricerca che utilizzi un'euristica ammissibile è sempre ottimale, ovvero i cammini trovati sono cammini minimi.

### 3.3.2 Consistenza

Una funzione euristica è detta *consistente*, o *monotona*, se la sua stima è sempre minore o uguale alla distanza stimata da un qualunque nodo nel vicinato al nodo goal, sommato al costo del raggiungimento di tale vicino dal nodo corrente [16], ovvero se è verificata la disuguaglianza triangolare. Formalmente, per ogni nodo $n$ e ogni suo successore $n'$, il costo stimato del raggiungimento del nodo target da $n$ non è più grande del costo del passo da $n$ a $n'$ più il costo stimato del raggiungimento del nodo target da $n'$:

$$H(n) \leq C(n, n') + H(n') \quad \forall n$$

dove $C(n, n')$ indica il costo esatto del raggiungimento del nodo $n'$ partendo da $n$. Può essere dimostrato per induzione, che una euristica consistente è anche ammissibile, ovvero non sovrastima mai il costo del raggiungimento del goal (il viceversa invece non è sempre vero). Conseguentemente, un algoritmo di ricerca che utilizzi un'euristica consistente è sempre un algoritmo ottimale. Le funzioni euristiche consistenti sono chiamate monotone poiché la stima del costo finale di una soluzione parziale è monotonicamente non decrescente:

$$F(n_j) \leq F(n_{j+1}) \quad \forall j \mid 1 \leq j \leq d$$

dove $d$ è lunghezza del cammino ottimo e con:

$$F(n_j) = G(n_j) + H(n_j)$$

dove $G(n_j)$ è il costo del miglior cammino dal nodo iniziale al nodo $n_j$.

### 3.3.3 Ruolo dell'euristica in A*

Uno degli aspetti che rende versatile A* è la possibilità di controllare il comportamento dell'algoritmo in base non solo al tipo di funzione euristica utilizzata, ma anche in base al suo peso all'interno della funzione di valutazione. A un estremo, se $H(n)$ vale sempre zero, solo $G(n)$ incide nella valutazione del nodo e A* si riduce all'algoritmo di Dijkstra, garantendo il ritrovamento del cammino più corto. Se $H(n)$ è sempre minore o uguale al costo del percorso dal nodo $n$ al goal, allora A* garantisce l'ottimalità dei percorsi trovati. Tanto più $H(n)$ è piccolo, tanti più nodi vengono valutati rendendo l'algoritmo più lento. Se $H(n)$ vale esattamente quanto il costo del cammino da $n$ all'arrivo, allora A* valuterà solamente i nodi che compongono il cammino ottimo. In genere non è possibile avere un euristica esatta, tranne che in alcuni casi speciali. Nel caso in cui $H(n)$ sia una sovrastima, l'algoritmo sarà più veloce ma i cammini trovati perdono l'ottimalità. All'altro estremo, se $H(n) = \mathcal{O}(G(n))$ allora $G(n)$ diviene irrilevante e A* converge nell'algoritmo Greedy Best-First Search. È quindi possibile tarare il comportamento di A* in base alle esigenze bilanciando il rapporto tra $G(n)$ e $H(n)$: infatti in alcuni casi può essere desiderabile avere una velocità di esecuzione maggiore ottenendo dei percorsi che sono subottimali, mentre in altri è imperativo ottenere il cammino che sia il più corto possibile a scapito del tempo impiegato per calcolarlo. Una considerazione va fatta riguardo alla somma tra $G(n)$ e $H(n)$: per avere risultati coerenti, è necessario che i valori di tra $G(n)$ e $H(n)$ abbiano la stessa scala. Se ad esempio $G(n)$ è misurata in metri mentre $H(n)$ è misurata come numero di step rimanenti, il risultato non sarà quello atteso, ma l'algoritmo potrebbe trovare percorsi errati o impiegare più iterazioni del dovuto. Di seguito vengono descritte alcune delle più comuni funzioni euristiche utilizzate.

### 3.3.4 Distanza euclidea

La distanza euclidea, chiamata anche distanza $L_2$ o distanza pitagorica, consiste semplicemente nella misura della lunghezza del percorso in linea retta tra due nodi. Formalmente coincide con il teorema di Pitagora in due dimensioni:

$$H(n) = \sqrt{(n_x - g_x)^2 + (n_y - g_y)^2}$$

dove $g$ rappresenta il nodo target. Questa euristica è indicata se le unità possono muoversi in qualsiasi direzione. Va fatta attenzione al problema della scala: in questo caso se per $G(n)$ è

utilizzata un altro tipo di distanza, i percorsi trovati saranno comunque ottimi ma sarà esplorata una porzione maggiore di grafo. La distanza euclidea è ammissibile.

### 3.3.5 Distanza Manhattan

Questa distanza, chiamata anche distanza $L_1$ o geometria del taxi, è stata introdotta da Hermann Minkowski secondo il quale la distanza tra due punti è la somma del valore assoluto delle differenze delle loro coordinate. Tra due nodi questo si traduce in:

$$H(n) = C(|n_x - g_x| + |n_y - g_y|)$$

Dove $C$ indica il costo del movimento tra due nodi adiacenti. La distanza $L_1$ è indicata per grafi a griglia in cui il movimento possibile per le unità è limitato agli assi Nord-Sud e Ovest-Est. La distanza Manhattan, a differenza di quella euclidea non è ammissibile.

### 3.3.6 Distanza diagonale

La distanza diagonale, indicata anche come distanza $L_\infty$ o distanza Čebyšëv, come la distanza Manhattan è un euristica utilizzata nei grafi a griglia. Essa consente il movimento anche lungo le diagonali, a volte chiamato movimento a otto direzioni:

$$H(n) = C_1(|n_x - g_x| + |n_y - g_y|) + (C_2 - 2C_1)\min(|n_x - g_x|, |n_y - g_y|)$$

Generalmente, su griglie uniformi, vale: $C_2 = \sqrt{2}C_1$. Anche la distanza diagonale è un euristica ammissibile.

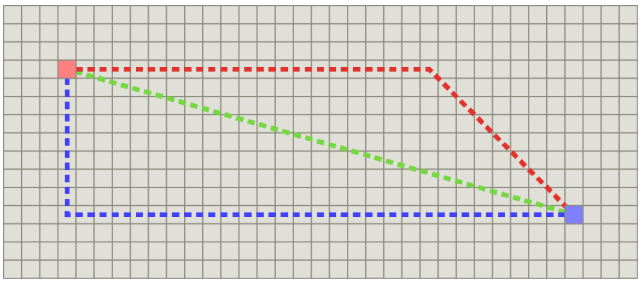
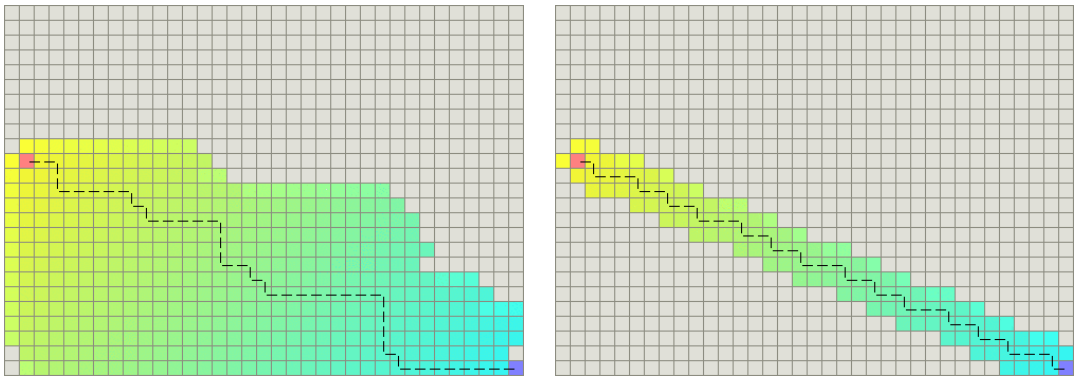


Figura 4: Euristiche comuni. In verde la distanza $L_2$, in blu la distanza $L_1$ mentre in rosso la distanza $L_\infty$.

### 3.3.7 Equalità dei percorsi

Nelle mappe a griglia, succede spesso che esistano molti percorsi aventi la stessa lunghezza. A* in questo caso è costretto a esplorare tutti i percorsi che hanno lo stesso costo $F(n)$, quando potrebbe limitarsi a uno solo di questi. Uno dei possibili modi per rompere l'equalità dei percorsi è quello di modificare leggermente il modo in cui l'open set ordina i nodi. Normalmente, nel caso in cui un nodo presenta lo stesso valore di $F(n)$ di uno già presente, la priorità viene data al nodo scoperto per primo. Se invece, in caso di parità del costo $F(n)$, l'ordinamento avviene in base all'euristica $H(n)$, solo un percorso viene privilegiato, risolvendo appunto il problema dell'equalità.



(a) Ordinamento solo in base a $F(n)$ (b) Ordinamento in base a $F(n)$ e a $H(n)$

Figura 5: Equalità dei percorsi.

## 3.4 Rappresentazione delle mappe

La cosa più importante da realizzare, riguardo le rappresentazioni usate nel pathfinding, è quella che in tutti i casi si tratta semplicemente di un grafo. Il modo più semplice di pensare in questi termini, nel contesto del pathfinding, è che ogni nodo del grafo rappresenta una possibile posizione che un agente potrebbe assumere, mentre gli archi indicano che è possibile per un agente muoversi tra i nodi che connettono. Ogni arco possiede un costo associato che rappresenta lo sforzo richiesto per un agente nel muoversi tra un nodo e il successivo. L'essenza del pathfinding è quindi quella del trovare un cammino tra nodi in un grafo, idealmente minimizzandone il costo. La rappresentazione della mappa può fare una grande differenza in termini di performance e di qualità del percorso trovato. Gli algoritmi utilizzati tendono ad

essere esponenziali: se si raddoppia la distanza tra i nodi start e goal, ci vorrà più del doppio del tempo per trovare il cammino minimo. In generale, minore è il numero di nodi nel grafo rappresentante la mappa, più veloce e performante A* sarà. Inoltre, più i nodi sono vicini alle posizioni che gli agenti devono raggiungere, migliore sarà la qualità dei percorsi trovati. Questo generalmente significa aumentare la densità dei nodi e quindi il loro numero complessivo. Occorre quindi trovare un buon compromesso in base alle proprie esigenze. Una proprietà che deve essere sempre presente, è quella che il grafo deve essere un grafo connesso, altrimenti A* potrebbe non terminare entrando in un ciclo infinito. Una parte fondamentale nel pathfinding nei videogame è quindi quella di saper tradurre nel modo migliore la mappa di gioco, astraendola in un grafo con le caratteristiche migliori per facilitare la ricerca dei cammini agli algoritmi incaricati. Di seguito verranno esposti le più comuni rappresentazioni di mappa utilizzate.

### 3.4.1 Griglia

Una mappa a griglia usa una suddivisione uniforme del mondo in regioni di forma regolare. Griglie di uso comune comprendono tile quadrate, triangolari o esagonali. Le griglie sono facili da costruire e da comprendere e sono molto utilizzate in tanti videogame come rappresentazione del mondo di gioco. Una cella può essere percorribile oppure contenere un ostacolo; il movimento è consentito solo sulle celle percorribili. Ogni cella viene tradotta in un nodo, avente un arco che lo connette con i nodi rappresentanti tutte le celle percorribili adiacenti. Un tipo particolare di griglia prende il nome di griglia *lattice*. Questa è una griglia quadrata uniforme i cui costi di movimento valgono $1$ per movimenti orizzontali e verticali mentre $\sqrt{2}$ per movimenti in diagonale. Questo tipo di griglia è quello utilizzato nel progetto sviluppato. Uno svantaggio delle griglie è quello che è difficile rappresentare in modo efficiente ambienti che presentano strettoie di dimensione variabile. Il problema si pone nella scelta della risoluzione della griglia. Per rappresentare nel dettaglio un ambiente che presenta del passaggi stretti, la dimensione della cella deve essere piccola; ma questo implica una maggior densità di celle. Incrementare il numero di celle significa avere molti più percorsi potenziali che dovranno essere presi in considerazione da A* rendendolo meno performante. Quindi, nell'ottica del far girare l'algoritmo di pathfinding il più velocemente possibile, occorre aumentare la dimensione delle celle il più possibile senza perdere importanti dettagli.

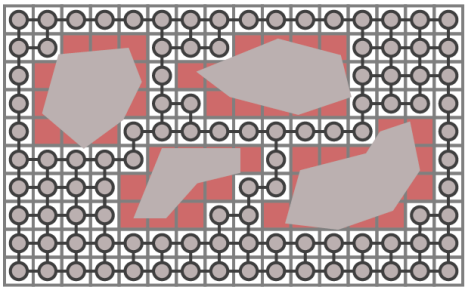


Figura 6: Rappresentazione di una mappa a griglia

### 3.4.2 Waypoint

L'idea di fondo di una sistema a waypoint è quella di piazzare manualmente un certo numero di nodi in posizioni chiave all'interno del mondo di gioco. In questo modo gli agenti possono trovare la propria via attorno ad ostacoli statici. Sfortunatamente, se il movimento degli agenti rimane sempre lungo i percorsi tra i waypoint, il loro comportamento risulterà limitato e quindi irrealistico. Occorre quindi piazzare i vari waypoint nel modo migliore possibile per rendere meno evidente i difetti di questo tipo di rappresentazione.

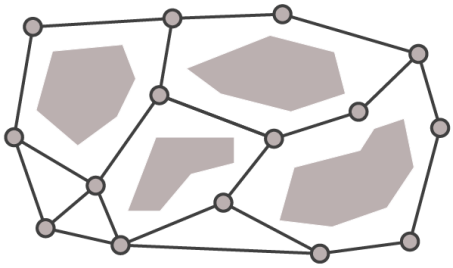


Figura 7: Rappresentazione di una mappa a Waypoint

### 3.4.3 Navigation mesh

Le navigation mesh sono un modo efficiente di rappresentare le regioni percorribili all'interno del mondo di gioco. Una comune rappresentazione vede suddividere la mappa in una serie di poligoni convessi connessi; nella fattispecie si usano triangoli. Proprio come nelle rappresentazioni a griglia, ogni poligono rappresenta un nodo mentre ogni lato condiviso tra due poligoni rappresenta un arco tra i due relativi nodi. Un'altra possibilità è quella di associare ai nodi i vertici dei poligoni, mentre agli archi vengono associati gli spigoli tra due vertici. Una

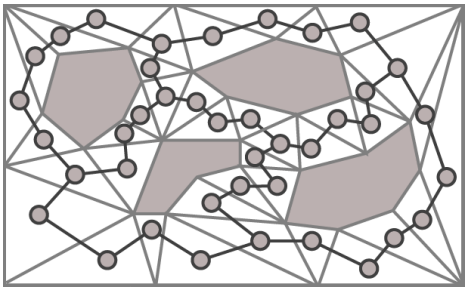


Figura 8: Rappresentazione della mappa come navigation mesh

rappresentazione come navigation mesh funziona bene con un livello variabile di dettaglio. Al contrario delle griglie, parte dell'ambiente poligonale può essere rappresentato dettagliatamente, senza forzare altre zone ad essere ugualmente dettagliate. Inoltre è possibile sfruttare il modello 3D della mappa per la generazione della navmesh.

## 3.5 Pathfinding nel game development

Un algoritmo come A* è molto efficiente nel trovare percorsi ottimali. Tuttavia nel mondo del game development, spesso si ha a che fare con mappe di grandi dimensioni e si ha a disposizione solo una frazione del tempo di CPU. Le prestazioni di A* in molti casi non sono sufficienti. Le soluzioni adottate nell'industria sono principalmente due: in primo luogo uno sforzo è compiuto per limitare il numero di nodi il più possibile; in secondo luogo sono adottati approcci gerarchici. Sono popolari anche rappresentazioni di mappa ibride, come ad esempio una navmesh per una rappresentazione globale ad alto livello e una griglia per rappresentazioni locali dove c'è necessità di maggior dettaglio. Gli approcci gerarchici riducono il problema in multipli livelli di ricerca. Ad esempio, un cavaliere deve raggiungere una locazione in una città lontana. Il primo passo è quello di raggiungere l'uscita dell'abitazione. Una volta fuori, in sella al proprio cavallo, il cavaliere dovrà uscire dalla città, prendere una strada principale, arrivare alla città di destinazione, trovare il palazzo cercato e una volta dentro trovare un percorso per arrivare alla stanza dove deve compiere la sua missione. Ci sono vari livelli di ricerca in questo esempio:

- a livello di edifici ci si preoccupa di trovare un percorso tra le stanze;
- a livello di città si passa da una strada all'altra fino a trovare l'uscita senza curarci delle stanze dei palazzi che si incontrano;

- a livello di mondo si considerano le strade principali senza considerare le strade all'interno delle città di passaggio.

Dividere il problema in livelli permette di ignorare molti percorsi che non sono rilevanti. Il problema si snellisce, diventa maggiormente gestibile e risolverlo diventa più veloce ed efficiente. Una mappa gerarchica ha molti livelli nella sua rappresentazione. Una gerarchia eterogenea tipicamente ha un numero fissato di livelli, ognuno con caratteristiche diverse. I livelli possono essere di tipi differenti (griglie, navigation mesh, waypoint). Una gerarchia omogenea ha un numero arbitrario di livelli tutti con le stesse caratteristiche. Quadtree e octree possono essere considerati gerarchie omogenee.

## 3.6 Scalabilità del pathfinding

Le tecniche viste sinora risolvono in parte il problema del pathfinding nel contesto dei videogame. Rimane tuttavia il problema della scalabilità. Gli approcci gerarchici sono generalmente efficienti ma il problema rimane comunque esponenziale e, all'aumentare delle richieste contemporanee di pathfinding, il sistema potrebbe non riuscire a gestirle senza gravi rallentamenti. Se il numero di agenti sale vertiginosamente, non è più possibile risolvere i percorsi in tempo reale. Nel capitolo 2 sono state esposte numerose tecniche che cercano di risolvere il problema, utilizzando spesso approcci basati sul calcolo parallelo su GPU, con vantaggi e svantaggi soprattutto se si pensa a una possibile implementazione in un game engine. Nel progetto sviluppato in questa tesi si tenta un'altra strada, ovvero quella della programmazione data-oriented multi-threading. A questo scopo sarà utilizzato un framework sviluppato da Unity Technologies [3], per il proprio game engine Unity, che implementa il design pattern architetturale Entity-Component-System.

# Capitolo 4

# Entity Component System

Entity-component-system (ECS) è un design pattern che fa parte della categoria dei pattern architetturali. Invece di usare il comune Object-Oriented Design, ECS sfrutta un altro paradigma chiamato Data-Oriented Design. Nel Data-Oriented Design c'è una separazione netta tra dati e logica, in questo modo è possibile applicare un set di istruzioni a un grande numero di oggetti in parallelo. ECS, inoltre, fa largo uso del principio di "composizione al posto dell'ereditarietà", ovvero il comportamento polimorfico e il riutilizzo del codice deve essere ottenuto mediante la composizione (contenere altre classi che implementano la funzionalità desiderata), invece dell'ereditarietà (essere una sottoclasse). Il pattern ECS permette di attenersi agevolmente al principio di singola responsabilità, il quale afferma che ogni elemento di un programma (classe, metodo, variabile) deve avere una sola responsabilità, e che tale responsabilità debba essere interamente incapsulata dall'elemento stesso.

## 4.1 Data-Oriented Design

In informatica, il data-oriented design (da non confondere con la programmazione data-driven) è una tecnica di programmazione usata prevalentemente nel video game development. Il concetto del data-oriented design è quello che il codice viene costruito intorno alle strutture di dati, descrivendo ciò che si vuole ottenere in termini di manipolazioni di queste strutture. Questo approccio è diventato popolare specialmente durante la settima generazioni di console (Playstation 3 e Xbox 360), quando il pericolo di cache miss era particolarmente pronunciato.

Nei moderni sistemi, la memoria principale ha una frequenza di clock molto più bassa rispetto a quella della CPU; di conseguenza il problema della località dei dati domina le performance, richiedendo accorgimenti negli accessi di memoria per limitare la perdita di cicli di clock. Le console generalmente hanno delle CPU relativamente poco potenti in favore delle unità grafiche (GPU). È quindi imperativo che il codice sia efficiente per evitare colli di bottiglia di Von Neumann. Nei tradizionali principi di design della programmazione object-oriented (OOP), è presente una bassa località dei dati, in particolare se è usato il polimorfismo a runtime. Anche se superficialmente la programmazione orientata agli oggetti sembra organizzare il codice attorno ai dati, in pratica non è così. La OOP si preoccupa di organizzare il codice *sorgente* attorno ai *tipi* di dati, invece che raggruppare fisicamente i campi e gli array in un formato efficiente per l'accesso. I processori sono progettati per lavorare bene in situazioni in cui ci sia una buona località dei dati con un conseguente buon utilizzo della cache. Quindi, nel data-oriented design, si tende, ogni volta che sia possibile, a organizzare tutto in grandi array omogenei. Inoltre, è preferibile affidarsi ad algoritmi brute force che siano però cache-friendly, invece che utilizzare algoritmi potenzialmente più efficienti che hanno un miglior costo $\mathcal{O}$ grande, ma che non si adattano alle limitazioni dell'architettura su cui girano. Questi accorgimenti, in un contesto real-time, ovvero eseguito ad ogni frame, (o più volte per frame), danno degli enormi guadagni in termini di performance. Il focus sui dati può inoltre portare a scrivere facilmente codice isolato, auto contenuto e intercambiabile; in altre parole il data-oriented design porta a una spiccata modularità del codice. Più nel dettaglio i principali vantaggi del data-oriented design sono:

## Parallelizzazione

Nell'ultimo decennio, i processori hanno avuto un evoluzione che ha portato ad avere multipli core di elaborazione. Chiunque abbia provato a scrivere del codice OOP e a parallelizzarlo col multi-threading può attestare quanto sia difficile, quanto sia prono agli errori e potenzialmente poco efficiente. Spesso si finisce con l'aggiungere molte primitive di sincronizzazione per prevenire l'accesso concorrente ai dati dei vari thread e solitamente molti di essi rimangono in attesa che altri thread abbiano completato il loro lavoro. Come risultato, il miglioramento delle performance può essere deludente. Nel data-oriented design, la parallelizzazione diventa molto più semplice: tipicamente si hanno dei dati in input, piccole funzioni che li processano e dei dati in output. Si può agilmente spezzare il lavoro di tali funzioni attraverso thread multipli con una sincronizzazione minima tra di essi.

### Utilizzo della cache

Oltre all'utilizzo di più core, uno dei concetti chiave del raggiungimento di ottime performance nei moderni hardware (che presentano accessi in memoria lenti accompagnati da multipli livelli di cache), è avere degli accessi di memoria cache-friendly. Il data-oriented design risulta essere molto efficiente nell'uso della instruction cache, poiché lo stesso codice è eseguito ripetutamente più volte. Inoltre, se organizziamo i dati in grandi blocchi contigui, essi si possono processare sequenzialmente, ottenendo un uso di cache pressoché perfetto e di nuovo le performance migliorano.

### Ottimizzazione

Quando si pensa ad oggetti e funzioni, le possibili ottimizzazioni vengono usualmente fatte a livello di metodi o a livello di algoritmi. Questo tipo di ottimizzazione è certamente vantaggiosa, ma pensando in termini di dati si può andare oltre. La logica di un videogame sta tutta nel trasformare un set di dati (asset, input, stato) in un altro set di dati (comandi grafici, nuovo stato). Tenendo presente il flusso dei dati, si possono prendere migliori decisioni basate su come i dati sono trasformati e come gli stessi sono utilizzati. Questo tipo di ottimizzazione può essere molto difficile da implementare con i più tradizionali metodi object-oriented.

### Modularità

I vantaggi del data-oriented design visti sinora sono tutti a favore delle performance. È indubbio che nei videogame le performance sono estremamente importanti. C'è spesso però un conflitto tra tecniche che aumentano le performance e tecniche che invece aumentano la semplicità di sviluppo e la leggibilità del codice. Fortunatamente, il data-oriented design beneficia sia in termini di performance che di facilità di sviluppo. Quando si scrive codice specifico nella trasformazione di dati, si ottengono delle piccole funzioni con poche dipendenze. Questo livello di modularità insieme alla mancanza di dipendenze rende più facile capire, rimpiazzare o aggiornare il codice.

Il data-oriented design non è tuttavia un rimedio assoluto a tutti i problemi del game development. Aiuta molto a scrivere codice con buone prestazioni, rende i programmi leggibili e facili da mantenere, ma ha anche degli svantaggi. Il più evidente è quello che l'approccio differisce dal quello che la maggior parte dei programmatori è abituata ad usare. Richiede infatti un

mental-shift che consenta di vedere i problemi secondo un'altra prospettiva (un po' come avviene quando si passa per la prima volta alla programmazione funzionale). Ci vuole del tempo e pratica per prendere confidenza con il nuovo modo di ragionare. Inoltre, proprio a causa del differente approccio, può essere difficile integrare o interfacciarsi a codice già scritto secondo le tecniche object-oriented.

Esempi di utilizzo del data-oriented design sono quello di DICE [19] all'interno del loro game engine Frostbite [20], in particolare nei sistemi di culling e nei sistemi di intelligenza artificiale [21].

## 4.2 Pattern Entity-Component-System

La prima apparizione del pattern ECS nell'industria è stata nel 2007 nel videogame "Operation Flashpoint: Dragon Rising" sviluppato da Codemaster [22] per Playstation 3 e Xbox 360. All'interno di Entity-Component-System (ECS) gli oggetti simulati sono rappresentati da *entità* le quali rappresentano un raggruppamento di *componenti*. Queste componenti descrivono le differenti proprietà dell'entità a cui fanno riferimento; ad esempio, la posizione o il renderer. Le componenti quindi costituiscono dati puri, senza alcuna logica. I sistemi sono invece dei moduli che sono responsabili di un certo aspetto della simulazione, ad esempio il movimento o il rendering. Se un entità contiene un insieme di componenti che è richiesta da un certo sistema, essa è integrata nel game loop di quel dato sistema. Ad esempio, un'entità avente la componente `unitàInMovimento`, la componente `posizione` e la componente `posizioneObiettivo`, sarà inclusa nel sistema `SistemaMuoviUnità`. Tale sistema sarà responsabile del movimento di tutte le unità che debbano muoversi. Ricapitolando, il pattern ECS si compone di:

- **Entity**: è un identificatore univoco di un oggetto di gioco. Spesso è rappresentato semplicemente da un intero;
- **Component**: è insieme di dati che riguardano uno specifico aspetto di un oggetto di gioco;
- **System**: rappresentano la logica di gioco. Ogni sistema si occupa di solo un aspetto di tale logica, agendo sulle componenti delle entità interessate.

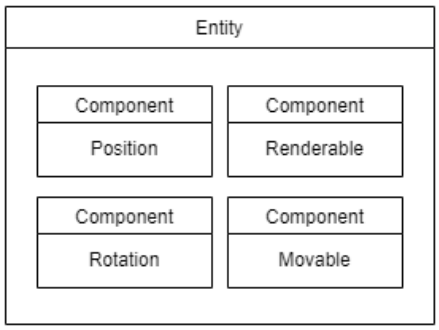



Figura 9: Entità in ECS

Il tradizionale sistema di implementazione degli oggetti di gioco fa uso dell'object-oriented design. Ogni oggetto è rappresentato da un istanza di una classe, che abilita l'estensione ad altri tipi di oggetti attraverso ereditarietà e polimorfismo. Questo approccio conduce rapidamente a un'intricata e rigida gerarchia di classi. All'aumentare degli oggetti di gioco, diventa sempre più difficile trovare un posto nella gerarchia, specialmente se tale oggetto necessita di molte nuove funzionalità. Il problema è risolto elegantemente dalla rappresentazione degli oggetti di gioco in entità, dato che un entità è semplicemente una composizione di componenti. È quindi semplice creare nuove entità, è sufficiente raggruppare le componenti che servono o definire nuove componenti all'occorrenza.

In un implementazione naïve di ECS, ogni sistema itera attraverso la lista completa di tutte le entità e ne seleziona solo quelle di cui ha bisogno. Il costo totale dell'iterazione per tutti i sistemi diventa troppo alto, specialmente quando il numero di sistemi e di entità cresce. Per ovviare al problema, ogni tipo di componente è memorizzato in un array separato, in modo che tutti i sistemi che operano su tali componenti iterino solo sugli oggetti interessati. Nelle comuni architetture ECS, quello che era uno svantaggio diviene un punto di forza in termini di performance, in quanto è possibile impiegare efficientemente le cache istruzioni e cache dati delle CPU. I benefici dei framework che adottino il pattern ECS includono quindi una buona separazione delle competenze (*separation of concerns*) e la possibilità di aggiungere o rimuovere componenti a runtime. Questo è facilitato dal fatto che le entità sono composte da componenti invece di ereditare le loro proprietà come nell'object-oriented design. Le architetture ECS sono quindi estremamente flessibili ed estendibili.

## 4.3 Unity ECS

Nel 2018 Unity Technologies [2], responsabile della creazione del game engine Unity, annuncia che è in corso di sviluppo una serie di cambiamenti nel framework del proprio engine. Viene infatti dichiarata la volontà di incrementare le performance di Unity, da sempre il punto debole del game engine, utilizzando un approccio data-oriented, ovvero quello di Entity-Component-System. Non solo, ad esso sarà affiancato un sistema multi-threading per il calcolo parallelo in C# dal nome *Unity Jobs* e un nuovo metodo di compilazione chiamato *Burst Compile*. Le novità saranno implementate durante il corso di vita delle release 2018.x e 2019.x dell'engine e sono basate, come il vecchio sistema, sul framework Mono/.NET. Attualmente le nuove tecnologie sono disponibili in una versione che è ancora fase di sviluppo e quindi non definitiva. È già possibile però sostituire il nuovo approccio a quello classico per un gran numero di situazioni, mentre per altre, come ad esempio il motore fisico, bisognerà attendere future versioni.

### 4.3.1 Sistema classico: Monobehaviour

Per capire davvero i vantaggi del nuovo approccio data-oriented e come questo impatti sulle performance, occorre prima vedere come funziona il sistema classico di Unity. In questo approccio, gli oggetti di gioco vengono rappresentati dai *gameObject* che sono istanze della classe `GameObject`. Ogni gameObject contiene i riferimenti dei componenti che ne aggiungono funzionalità (ogni componente è un istanza di una classe che contiene i relativi dati e logica), oltre che i riferimenti di uno o più script che ne alterano lo stato a runtime. Ogni script deriva dalla classe `Monobehaviour`, la quale contiene il metodo `Update()` che rappresenta il game loop. Gli oggetti e i componenti, che sono riferiti in questi script, sono memorizzati in modo sparso sullo heap; i dati quindi non sono in una forma adatta alle operazioni SIMD. Inoltre, dati e logica sono strettamente accoppiati. Questo significa che il riuso del codice è limitato, poiché il processo di elaborazione è legato a uno specifico set di dati. Un sistema di questo tipo è fortemente dipendente dai tipi riferimento. Inoltre, una situazione comune nei videogame è quella di applicare la stessa logica a un gran numero di oggetti. Si pensi ad esempio al movimento di un esercito. Ogni unità contiene il riferimento di un istanza dello script di movimento; la memoria sarà quindi occupata da un numero di istanze pari al numero di unità, benché ognuno di essi svolga esattamente lo stesso compito. Basterebbe in questo caso avere un unico sistema di movimento che iteri sulle unità modificandone la posizione; questo è proprio l'approccio di

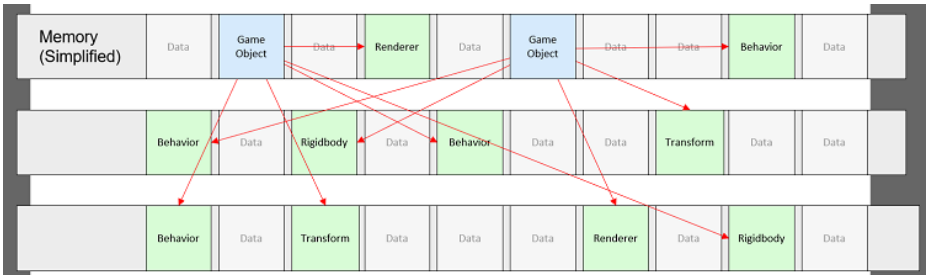



Figura 10: La memoria è frammentata tra oggetti, riferimenti e componenti.

ECS. Non solo, ma in questo caso viene naturale pensare a una vettorizzazione SIMD.

### 4.3.2 Funzionamento generale di Unity ECS

All'interno del pattern ECS abbiamo le entità, che rappresentano gli oggetti di gioco, le quali raggruppano un certo numero di componenti che sono poi manipolate dai sistemi. Il componente software che si occupa della creazione delle entità e della gestione dei sistemi si chiama `EntityManager`. Tramite esso è possibile creare, distruggere entità o aggiungere e rimuovere componenti a runtime, oltre che attivare e disattivare sistemi. All'avvio dell'applicazione, Unity si occupa del bootstrap, attivando in automatico tutti i sistemi presenti nella directory di lavoro, oltre che i sistemi integrati nell'engine quale ad esempio il sistema di rendering.

### 4.3.3 Entità e Archetipi

Si è già detto che in ECS gli oggetti di gioco sono rappresentati dalle entità. In Unity un'entità è identificata dalla struct `Entity`:

Come si può vedere un entità è identificata solo da una coppia univoca di interi. `Version` è

```
public struct Entity
{
        public int Index;
        public int Version;
}
```

Sorgente 5: Struct `Entity`

utilizzata in casi speciali in cui si voglia avere più versioni della stessa entità. Per creare un entità basta utilizzare il metodo `CreateEntity()` messo a disposizione dall'`EntityManager`, la cui istanza è richiamabile dall'oggetto `World`, che rappresenta appunto il mondo di gioco. Alla

nuova entità è poi possibile associare le componenti. Alcune componenti sono associate in

```
EntityManager em;
em = World.Active.GetOrCreateExistingManager<EntityManager>();
Entity e = em.CreateEntity();
em.AddComponent(e, ComponentType1);
em.AddComponent(e, ComponentType2);
```

Sorgente 6: Creazione di un entità.

automatico da Unity, ad esempio nel caso di entità che abbiano un componente `Mesh`, Unity associa il componente `TransformMatrix` utilizzato nel render. Per facilitare la creazione di entità è poi possibile definire un archetipo, che non è altro che un blueprint che elenca le componenti che una certa entità deve avere; in questo modo viene facilitata la creazione di più entità simili.

```
EntityManager em;
em = World.Active.GetOrCreateExistingManager<EntityManager>();

EntityArchetype ea = em.CreateArchetype(
        typeof(Component1),
        typeof(Component2)
);
Entity e = em.CreateEntity(ea);
```

Sorgente 7: Archetipo di entità.

### 4.3.4 Componenti

I componenti rappresentano le proprietà che una data entità deve avere. Unity mette a disposizione alcune componenti di uso comune, come il componente `Position` o il componente `InstanceMeshRenderer`. Tuttavia è possibile definire componenti custom (sarebbe oltremodo limitante altrimenti), con una struct che implementa l'interfaccia `IComponentData`. All'interno di un componente sono consentiti solo tipi blittable [23] (tipi di dato che hanno la stessa identica rappresentazione in memoria sia utilizzando il garbage collector che usando memoria nativa), e non sono consentiti tipi riferimento; questo per motivi di performance e layout della memoria come già detto in precedenza. Un tipo particolare di componente è lo `ISharedComponentData`: esso è utilizzabile ogni qualvolta un dato è condiviso tra molte entità. Un esempio è quello delle mesh; ogni entità che condivide lo stesso modello poligonale può utilizzare la stessa componente condivisa. In questo modo il dato viene memorizzato una volta

```
public struct Component : IComponentData
{
    public float Value;
}
```

Sorgente 8: Esempio di componente.

sola. Un problema sorge se si vuole memorizzare un array di dati: essendo un array un tipo riferimento non è possibile utilizzarlo in un componente. La gestione della memoria ordinando i dati in maniera cache-friendly risulterebbe difficoltosa, dato che non si conosce il numero di elementi in ogni array. La soluzione è un'altro tipo di componente: il `DynamicBuffer`. In questa componente viene dichiarato solo il tipo dell'array (es. `int`) e un numero di massima degli elementi contenuti. A runtime l'engine fara il possibile poi per organizzare i dati nella maniera migliore possibile.

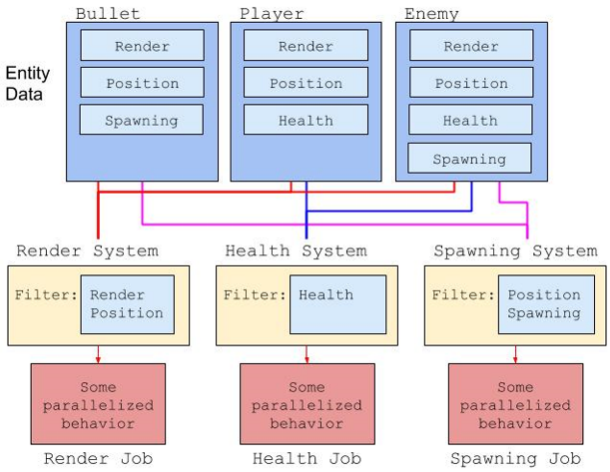



Figura 11: ECS in Unity

### 4.3.5 Sistemi e filtraggio delle entità

Nell'architettura Entity-Component-System ogni sistema si occupa di uno specifico compito. Un sistema, in Unity, è una classe che eredita dalla classe `ComponentSystem`. La logica viene implementata nel metodo `OnUpdate()` che è l'equivalente del metodo `Update()` nei

`MonoBehaviour`. L'istanza di ogni sistema è automaticamente costruita da Unity all'avvio dell'applicazione. Altri metodi utili sono `OnCreateManager()` e `OnDestroyManager()` che vengono invocati all'atto della creazione e distruzione del sistema. Ogni sistema opera solo sulle entità interessate; questa discriminazione avviene a livello di componenti mediante un filtraggio delle entità. Ad esempio, un sistema di movimento, che abbia come filtro le componenti `Position` e `CanMove`, agirà su tutte le entità che hanno tali componenti, indipendentemente dalle altre componenti che tali entità abbiano associate. Ci sono vari modi per filtrare e iterare sulle entità; una possibilità è l'utilizzo dell'injection. L'injection permette ai sistemi di dichiarare le proprie dipendenze; tali dipendenze vengono automaticamente iniettate nelle variabili contrassegnate dall'attributo `[Inject]`. Il filtraggio in questo caso avviene in due modi: il primo è attraverso un component group. Esso è una struct che descrive i componenti che un entità deve o non deve avere per essere filtrata. I singoli componenti sono memorizzati in `ComponentDataArray<>` insieme a un `EntityArray` che contiene le entità filtrate. Tutti gli array sono indicizzati alla stessa maniera (es. all'indice `1` di un `ComponentDataArray<>` è contenuta la componente che fa parte dell'entità memorizzata nell'`EntityArray` in posizione `1`). Il secondo tipo di iniezione è quello dei `ComponentDataArrayFromEntity<>`, in questo caso l'array conterrà tutte le componenti del tipo specificato indicizzato per entità.

```
public struct Group
{
        // ComponentDataArray permettono l'accesso ai IComponentData
        public ComponentDataArray<Position> Position;

        public ComponentArray<Health> Healths;

        // Accesso alle Entity ID.
        public EntityArray Entities;

        // Esclusione delle componenti MeshFilter dal gruppo
        public SubtractiveComponent<MeshFilter> MeshFilters;

        // la Length può essere iniettata per comodità
        public int Length;
}
//L'attributo inject inietta il component group nella variabile
[Inject] private Group m_Group;

//È possibile iniettare direttamente un component data array indicizzato
↪  per entità
[Inject] ComponentDataArrayFromEntity<Position> Positions;
```

Sorgente 9: Component group injection.

Un sistema come descritto qui sopra, può processare le entità filtrate sequenzialmente. Per abilitare il parallelismo occorre introdurre i Jobs System. Essi permettono di scrivere codice multithreaded che interagisce con il resto di Unity (in passato notoriamente single threaded) e rende facile scriverlo in modo corretto. Un sistema deve fornire ai job le loro dipendenze; a questo scopo esiste la classe `JobComponentSystem`. Gestire le dipendenze infatti è difficile, e il vantaggio di `JobComponentSystem` è che lo fa automaticamente. Job in diversi sistemi possono leggere da un componente in parallelo. Ma se un job invece scrive nel componente, allora dovra essere schedulato con una dipendeza verso gli altri job. Un sistema che derivi da `JobComponentSystem` ottiene il metodo `OnUpdate(JobHandle jh)`, in questo modo Unity riesce a gestire i vari Job in maniera efficiente e sicura.

### 4.3.6 C# Job System

Un aspetto essenziale del C# Job System è la sua integrazione con il sistema che Unity usa internamente (Unity native job system). Il codice scritto e Unity condivido i worker threads. Questa cooperazione evita la creazione di un numero di thread maggiore del numero di core CPU disponibili, la quale causerebbe contese sull'utilizzo delle risorse CPU. Un job system gestisce il codice multithreaded creando job invece che thread. Un job è una piccola unità di lavoro che esegue uno specifico compito. Esso riceve parametri e opera su dati, similarmente al funzionamento di una chiamata di metodo. I job possono essere auto contenuti oppure possono dipendere da altri job per poter eseguire. Internamente i job gestiscono un gruppo di worker thread tra tutti i core, usualmente un worker thread per core CPU, per evitare il context switching. Un job system crea e mette i job in una coda d'esecuzione; i worker threads ne prendono uno dalla coda e lo eseguono. La gestione delle dipendenze assicura che i job vengano eseguiti nell'ordine appropriato. Nella pratica, un job è una struct che implementa l'interfaccia `IJob` o un suo derivato; essa fornisce il metodo `Execute()` che deve contenere il codice che il job dovrà eseguire. All'interno di un job system deve essere quindi presente una struct `IJob`, e nell'`OnUpdate(JobHandle jh)` il job può essere schedulato. È possibile schedulare assieme differenti job oppure più job dello stesso tipo. Il metodo `Schedule()` di ogni job restituisce le dipendenze per il job successivo, in questo modo Unity può sapere quali job possono essere eseguiti parallelamente e quali no. Contestualmente allo scheduling dei job è possibile inserire delle primitive di sincronizzazione, nel caso sia necessario, ad esempio aspettando che un job finisca di eseguire prima di schedularne un altro.

```
public class FooSystem : JobComponentSystem
{
    public struct FirstJob : IJob
    {
        public void Execute()
        {
            //codice da eseguire
        }
    }

    public struct SecondJob : IJob
    {
        public void Execute()
        {
            //codice da eseguire
        }
    }

    protected override JobHandle OnUpdate(JobHandle inputDeps)
    {
        //al primo job sono passate le dipendenze dal parametro
        var firstJob = new FirstJob().Schedule(inputDeps);

        //è possibile inserire una primitiva di sincronizzazione
        firstJob.Complete();

        //al secondo job sono passate le dipendenze dal primo
        var secondJob = new SecondJob().Schedule(firstJob);

        return secondJob;
    }
}
```

Sorgente 10: Tipico job system.

Nello scheduling dei job, ci può essere solo un job che esegue uno specifico task. In un videogame è comune invece eseguire la stessa operazione su un grande numero di oggetti. La gestione di queste situazioni è presa in carico da un tipo di job chiamato `IJobParallelFor`. Un job `ParallelFor` viene eseguito su più core, venendo diviso in più job, ognuno dei quali si occupa della gestione di una parte del workload. `IJobParallelFor` si comporta quindi come un `IJob`, ma invece che avere un singolo metodo `Execute()`, invoca un `Execute()` per ogni elemento nel workload di input. Tale metodo ha come parametro un intero che rappresenta l'indice per l'accesso ai singoli elementi della sorgente di dati all'interno dell'implementazione del job. Quando viene schedulato un `IJobParallelFor`, occorre specificare manualmente quanti `Execute()` devono essere eseguiti. Più nel dettaglio, C# Job System divide il lavoro in batch da distribuire ai vari core. Ogni batch contiene un sottoinsieme degli `Execute()`. Vengono poi schedulati fino a un job, nello Unity native job system, per core CPU a cui vengono passati i batch da processare.

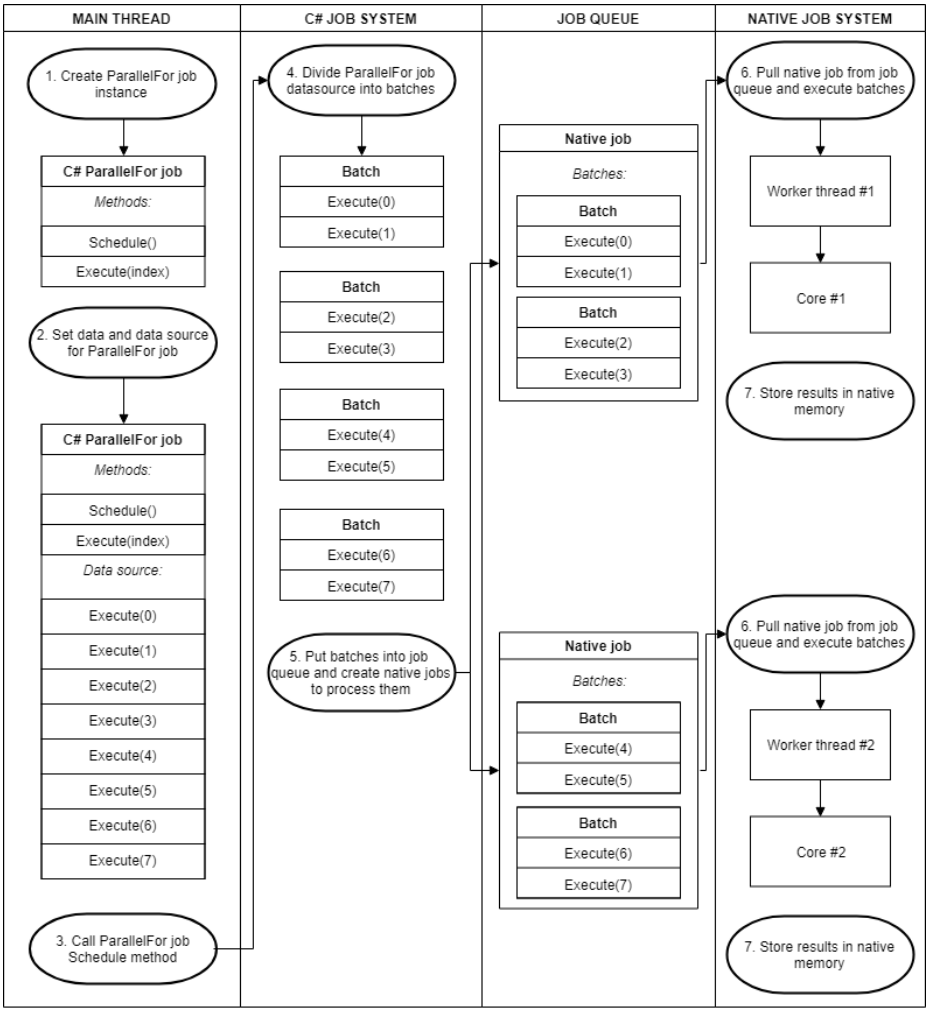



Figura 12: Scheduling interno di un IJobParallelFor

Quando un native job completa i suoi batch prima degli altri, "ruba" i batch rimanenti agli altri job. Ne ruba solo metà alla volta, questo per garantire la località dei dati in cache. Per ottimizzare il processo, va specificato manualmente il numero dei batch. Tale valore, che deve essere compreso tra $1$ e $64$, controlla quanti native job vengono generati, e quanto finemente è fatta la ridistribuzione del lavoro tra i thread. Avere un valore basso, consente di avere una distribuzione equa del workload tra i thread. Tuttavia introduce un po' di overhead, quindi a volte è meglio avere un valore più alto. Partire da $1$ e incrementare gradualmente il numero dei batch fino ad un degrado delle performance è una valida strategia per trovare un valore adatto.

### 4.3.7 Barriera e Command Buffer

Attraverso i job system è possibile eseguire codice multithreaded. Tuttavia, per motivi di sicurezza e di concorrenza, non è possibile creare o distruggere entità né aggiungere o rimuovere componenti ad esse se non nel main thread. A questo scopo Unity mette a disposizione le barriere o `BarrierSystem`. Una barriera va dichiarata all'interno del job system, come classe vuota che eredita da `BarrierSystem`. Non c'è codice nella classe poiché serve solo come punto di sincronizzazione. La barriera va poi iniettata con l'attributo `[Inject]` nel sistema il quale potrà accedere al `CommandBuffer` tramite il metodo `CreateCommandBuffer()` per poi passarlo al job a cui serve. All'interno del job il `CommandBuffer` si comporta come un `EntityManager`, registrando le azioni da compiere. Quando il job ha concluso e il `BarrierSystem` viene aggiornato, automaticamente sul main thread vengono eseguiti i comandi nell'ordine in cui sono stati creati. Questo passaggio extra è necessario così che la gestione della memoria rimanga centralizzata e il determinismo delle entità e dei componenti generati sia garantito.

### 4.3.8 Native memory e native container

Per garantire una gestione della memoria ottimale con un conseguente aumento delle performance, all'interno dei job viene utilizzata solo la memoria nativa. La native memory è la memoria fornita e gestita direttamente dal sistema operativo e non dal garbage collector di C# (esattamente come avviene in linguaggi come C++). I dati in memoria nativa sono organizzati nei native container. Un native container fornisce un wrapper C# relativamente sicuro a un area di memoria nativa. Al suo interno contiene un puntatore a un allocazione manuale di memoria. Usato insieme allo Unity C# Job System, un native container consente a un job di accedere a dati condivisi col main thread, invece che lavorare su una copia. È possibile creare native container custom, con il vantaggio di non utilizzare nessuna memoria gestita dal garbage collector e di averne il totale controllo sulle allocazioni. I dati contenuti devono essere tuttavia di tipo blittable. L'uso dei native container nei job può essere controllato dal job debugger che può assicurare che tutti gli accesi al container siano sicuri e che sia lanciata un eccezione in caso di race condition o di comportamenti non deterministici. Essendo un native container gestito completamente dal programmatore, è imperativo liberare la memoria alla fine del suo utilizzo per evitare pericolosi memory leak.

I native container predefiniti in Unity sono:

- NativeArray: una versione nativa di un array;
- NativeList: un NativeArray ridimensionabile;
- NativeHashMap: un insieme di chiavi-valore;
- NativeMultiHashMap: multipli valori per chiave;
- NativeQueue: una coda FIFO.

In fase di allocazione, è necessario definire la durata di vita del container tramite un parametro di tipo `Allocator`:

- Temp: ha la allocazione più veloce possibile. La durata di vita è inferiore alla durata di un frame;
- TempJob: ha un allocazione più lenta di un Temp ma più rapida di un Persistent. Ha una durata di vita fino a quattro frame ed è thread-safe;
- Persistent: ha l'allocazione più lenta ma ha una lunga durata, se necessario tutta la durata di vita dell'applicazione. È un wrapper a una chiamata diretta a una malloc.

Se al termine di vita del container esso non è stato deallocato, il sistema di sicurezza si occupa di lanciare un eccezione.

### 4.3.9 Burst Compiler

L'ultimo tassello del novità introdotte in Unity è quello del Burst Compiler. Burst è un compilatore che traduce il bytecode IL/.NET generato da C# in codice nativo altamente ottimizzato usando LLVM. Il codice è ottimizzato sfruttando le capacità della piattaforma di destinazione. Burst può essere utilizzato per incrementare le performance dei job scritti nello C# Job System, in effetti è stato pensato proprio per questo motivo. Il codice che si vuole ottimizzare va taggato con l'attributo `[BurstCompile]`. Al momento tuttavia, questa possibilità è ristretta a un numero limitato di costrutti.

# Capitolo 5

# Progetto

## 5.1 Obiettivi

Lo scopo del progetto di tesi è quello della verifica sperimentale dell'efficacia di un approccio data-oriented multithreaded al problema della scalabilità del pathfinding nel contesto di un game engine. Con questo obiettivo in mente, è stato realizzato un prototipo basato sulla architettura Entity-Component-System proposta dal game engine Unity. Tale prototipo consiste in una simulazione real time in cui vengono generati un certo numero variabile di agenti in una mappa popolata da ostacoli. Gli agenti, grazie al pathfinding di A* trovano e percorrono un cammino verso la propria posizione obiettivo. Per incrementare il realismo della simulazione, si è deciso di implementare una collision avoidance reciproca degli agenti; questo è stato fatto utilizzando una libreria di terze parti chiamata RVO2 [24] che assolve appunto a questo compito.

## 5.2 Mappa

La mappa della simulazione è costituita da un area quadrata popolata da palazzi disposti omogeneamente. Essa è rappresentata da un grafo lattice le cui dimensioni, in termini di nodi, sono impostabili dal programmatore nella classe `Settings`. Questo tipo di rappresentazione è stata scelta perché è una di quelle che maggiormente risente del problema della scalabilità, a causa dell'alta densità di nodi presenti. Tuttavia, l'approccio seguito nella realizzazione del sistema

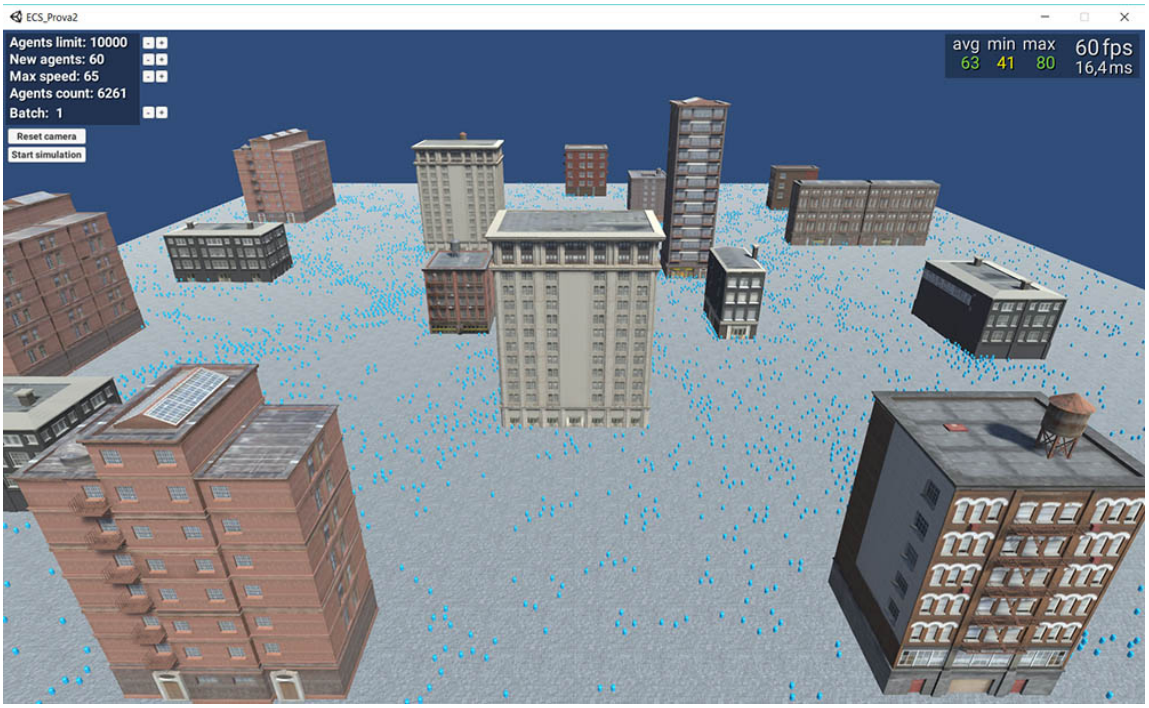



Figura 13: Mappa della simulazione

di pathfinding è indipendente dalla topologia del grafo e può essere applicato a un qualsiasi grafo pesato connesso.

## 5.3 Entità e Componenti

La decomposizione data-oriented del problema vede principalmente due attori in gioco: i nodi del grafo e gli agenti. Essi sono quindi modellati dalle entità archetipo `nodeArchetype` e `agentArchetype`. Un nodo ha bisogno solo di due informazioni per essere definito, ovvero la propria posizione e il fatto di poter essere attraversato o meno. L'archetipo di un entità nodo è quindi costituito dalla componente `Position`, che memorizza al suo interno un `float3` (un vettore di tre float), dalla componente `Walkable`, memorizzante un valore booleano, e dalla componente `Node`. Quest'ultima è una componente tag, senza dati, che è necessaria per facilitare le operazioni di filtraggio. Un agente invece, oltre alla propria posizione, deve memorizzare la propria posizione obiettivo, deve memorizzare il fatto, per tale agente, di essere già stato processato dal sistema di pathfinding e infine una lista di waypoint che costituisce il percorso trovato dopo il processamento. L'archetipo di un entità agente è quindi costituito

```
agentArchetype = entityManager.CreateArchetype(
            typeof(Position),
            typeof(MeshInstanceRenderer),
            typeof(Agent),
            typeof(Target),
            typeof(Waypoints)
        );

nodeArchetype = entityManager.CreateArchetype(
            typeof(Position),
            typeof(Node)
        );
```

Sorgente 11: Archetipi utilizzati.

dalla componente tag `Agent`, da un componente `Position`, da un componente `Target` e da una componente `Waypoints`. Ad essi si aggiunge anche il componente `MeshInstanceRenderer`, responsabile dell'immagazzinamento della mesh e del materiale necessari al render grafico. Mesh e materiale sono gli stessi per ogni agente; per questo motivo `MeshInstanceRenderer` è una componente condivisa, quindi di tipo `ISharedComponentData`. Tutte le altre componenti sono invece delle comuni `IComponentData` ad eccezione della componente `Waypoints` che è costituita da un `DynamicBuffer`, dato che non è nota a priori la quantità di waypoint per ogni percorso trovato. Per quanto riguarda la componente `Walkable`, non è possibile utilizzare un tipo `bool`, poiché esso non è di tipo blittable. Per aggirare il problema si è dovuto implementare una struct `TBool` che converte il valore booleano in un valore intero, `1` o `0`, memorizzato in una variabile di tipo `byte`.

## 5.4 Sistemi

Dopo aver descritto l'organizzazione dei dati, si illustra ora come è gestita la logica della simulazione. Come recita il principio di singola responsabilità, ogni sistema si deve occupare solo di uno specifico task. In questa simulazione i compiti, auto contenuti, che devo essere eseguiti sono:

- la generazione del grafo;
- la generazione degli agenti;
- il processo di ritrovamento dei percorsi (pathfinding);
- il movimento degli agenti

```
//componenti entità agente
public struct Agent : IComponentData {}

public struct Target : IComponentData
{
    public float3 Value;
}

[InternalBufferCapacity(25)]
public struct Waypoints : IBufferElementData
{
    public int2 Value;
}

//componenti entità nodo
public struct Node : IComponentData {}

public struct Walkable : IComponentData
{
    public TBool Value;
}

public struct TBool
{
    private readonly byte _value;
    public TBool(bool value) { _value = (byte)(value ? 1 : 0); }
    public static implicit operator TBool(bool value) { return new
    ↪  TBool(value); }
    public static implicit operator bool(TBool value) { return value._value
    ↪  != 0; }
}
```

Sorgente 12: Componenti sviluppati.

È bene notare che ognuno di questi compiti, ad eccezione del primo che viene svolto una sola volta ad inizio applicazione, è intrinsecamente indipendente dagli altri e possono quindi essere eseguiti in ordine arbitrario o perfino in parallelo all'interno del game loop. Per ognuno di essi è stato quindi implementato uno specifico sistema che che realizza la logica.

## 5.5 Generazione del grafo

Come già detto in precedenza, la mappa è modellata in un grafo lattice. Ad occuparsi delle generazione del grafo e quindi delle relative entità nodo è il sistema `GridGeneratorSystem`. Esso, per ovvi motivi, viene eseguito solamente a inizio applicazione. Nella scena sono presenti numerosi palazzi che rappresentano gli ostacoli che gli agenti devono evitare. Il sistema deve quindi, oltre ad occuparsi della creazione delle entità, rilevare quali nodi sono ostruiti e agire di conseguenza. All'interno di un doppio ciclo for che itera sulle coordinate `x` e `y` dei

nodi, viene infatti creata l'entità secondo l'archetipo `nodeArchetype`, a cui viene assegnata la propria posizione nella componente `Position`. Viene poi fatto il rilevamento della potenziale ostruzione del nodo tramite il metodo `Physics.CheckBox()`. Tale metodo, che prende come parametri una posizione e il semilato di un cubo, controlla se ci sono intersezioni. Per il rilevamento delle intersezioni, occorre che gli oggetti che si vuole rilevare siano dotati di un collider. Per questo motivo i palazzi sono rappresentati da gameObject invece che da entità, dato che, come già detto in precedenza, il motore fisico (e quindi i collider) non è ancora disponibile nel formato ECS. Il valore negato del risultato di tale controllo, viene assegnato alla componente `Walkable`. Il sistema `GridGeneratorSystem` contiene inoltre alcuni metodi helper statici, per operazioni quali la conversione da coordinate nodi a coordinate mondo oppure il ritrovamento del nodo più vicino a na data posizione.

```
for (int x = 0; x < s_gridSize.x; x++)
{
        float xPos = GridToWorldPosX(x);

        for (int y = 0; y < s_gridSize.y; y++)
        {
                float yPos = GridToWorldPosY(y);

                node = entityManager.CreateEntity(nodeArchetype);
                entityManager.SetComponentData(node, new Position { Value =
                ↪  new float3(xPos, 0, yPos) });

                entityManager.AddComponentData(node, new Walkable
                {
                        Value = !Physics.CheckBox(new Vector3(xPos, 0,
                        ↪  yPos), Vector3.one * (nodeSize / 2f))
                });
                grid[x, y] = node;
        }
}
```

Sorgente 13: Generazione del grafo.

## 5.6 Generazione degli agenti

Il compito di generare gli agenti è preso in carico dal sistema `SpawnAgentSystem`. Esso agisce sulla base di tre parametri modificabili a runtime: il numero massimo di agenti consentito nella simulazione, il numero di agenti da generare ad ogni passaggio, e la frequenza di tali passaggi. I parametri di default sono 5000 agenti massimi, con una generazione 100 agenti alla volta ogni 20 frame. All'interno del proprio game loop, il sistema quindi crea le entità agente secondo l'archetipo `agentArchetype`. Per ognuna di esse viene generata un posizione iniziale

e una posizione obiettivo, memorizzate nei componenti `Position` e `Target`. Alle entità viene anche assegnato il proprio aspetto tramite il componente condiviso `MeshInstanceRenderer`. È possibile definire in editor un blueprint di tale componente, contenente una mesh e una materiale. Da codice, il componente condiviso viene poi caricato e reso assegnabile alle entità. Entrambi i sistemi visti sinora non compiono operazioni di filtraggio; il loro scopo infatti è quello della creazione di entità e non di manipolazione delle stesse. Operazioni di filtraggio in questo caso non sono quindi necessarie.

## 5.7 Pathfinding

Il pathfinding viene gestito interamente dal sistema `AStarSystem`. Esso filtra le entità agenti che sono in attesa di essere processate tramite il component group `AgentGroup`. Sono richieste

```
private struct AgentGroup
{
        [ReadOnly] public ComponentDataArray<Position> Position;
        [ReadOnly] public ComponentDataArray<Agent> Agent;
        [ReadOnly] public ComponentDataArray<Target> Target;
        [ReadOnly] public EntityArray AgentEntity;
        public readonly int Length;
}
```

Sorgente 14: AgentGroup

anche le componenti `Waypoints`, necessarie per immagazzinare il percorso trovato, che vengono invece iniettate in un `ComponentDataFromEntity`. Il processamento delle entità all'interno del sistema è ad opera del `IJobParallelFor` nominato `AStarJob`. Essendo un parallelFor job, esso processa tutte le entità in parallelo sui vari core CPU. La prima cosa che viene svolta è la rimozione del componente `Target` dall'entità corrente tramite il `CommandBuffer` della barriera iniettata `AStarBarrier`; questo per segnalare il fatto che tale entità è già stata processata e quindi non verrà più selezionata dal sistema l'iterazione succesiva. Il job successivamente, per ogni entità, lancia il metodo `AStarSolver` che, presi in ingresso le coordinate dei due nodi più vicini alle posizioni iniziale e obiettivo, insieme all'entità corrente, si occupa di ritrovare il percorso minimo tramite A*. L'euristica utilizzata è la distanza diagonale, il cui metodo, per motivi di performance, è reso `inline`. Le strutture dati utilizzate per rappresentare l'open set, il closed set e i costi $G(n)$ sono dei native container. In particolare, il closed set è memorizzato in un `NativeArray<MinHeapNode>`, i costi $G(n)$ in un `NativeArray<int>` mentre per

```
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static int OctileDistance(int2 start, int2 end)
{
        int dx = math.abs(start.x - end.x);
        int dy = math.abs(start.y - end.y);
        return 10 * (dx + dy) + (14 - 2 * 10) * math.min(dx, dy);
}
```

Sorgente 15: Inlining della funzione euristica

l'open set, per garantirne la massima efficienza, si è implementata una versione nativa di un min heap, ovvero un `NativeMinHeap`. Esso minimizza il costo delle operazioni di inserimento e sopratutto di `pop()` dell'elemento col costo $F(n)$ minore. Dettagli sulla sua implementazione verranno forniti nel prossimo paragrafo. Per quanto riguarda la computazione di A*, si è scelto di utilizzare una rappresentazione ausiliaria dei nodi processati. Tale struttura dati, sottoforma di struct, memorizza i costi $F(n)$ e $H(n)$ del nodo rappresentato, per massimizzare l'efficienza del `NativeMinHeap`. La struct, che prende il nome di `MinHeapNode`, contiene anche le informazioni sul nodo predecessore che saranno utilizzate per ricostruire il percorso minimo a fine algoritmo. Un'altra ottimizzazione è quella di utilizzare un `NativeList` per la memorizzazione temporanea dei nodi del vicinato. Essa è allocata prima del ciclo principale di A* ed è riutilizzata ad ogni iterazione, risparmiando cicli di CPU per l'allocazione locale. Una volta arrivati al nodo target, il metodo `CreatePath()` si occupa di tracciare il percorso a ritroso fino al nodo iniziale, seguendo la catena di parentela tra i nodi memorizzati nel closed set. La serie di nodi trovati è poi scremata nei soli nodi che presentano un cambio di direzione.

```
if (currentNode.Position.x == goal.x && currentNode.Position.y == goal.y)
{
        var path = new NativeList<int2>(Allocator.TempJob);
        var current = currentNode;

        while(current.ParentPosition.x != -1)
        {
                path.Add(current.Position);
                current = closedSet[GetIndex(current.ParentPosition)];
        }
        path.Add(current.Position);

        CreatePath(agent, ref path);

        path.Dispose();
        break;
}
```

Sorgente 16: Ricostruzione del percorso.

Essi rappresentano i waypoint del cammino trovato e sono quindi assegnati alla componente `Waypoints` di ogni entità. Lo scheduling del job `AStarJob` è effettuata sulla base del numero di entità presenti nell'`AgentGroup` mentre il numero di batch è impostabile a runtime (di default è `1`).

### 5.7.1 NativeMinHeap

La gestione dell'open set è un punto critico di ogni algoritmo di pathfinding. In questo senso si è optato per una sua rappresentazione con un min heap in memoria nativa, seguendo le linee guida impartite da Unity [25]. In particolare, i dati sono organizzati secondo un min heap unario, ovvero ogni elemento ha un unico figlio ad eccezione della foglia che ne ha zero. La radice è costituita dall'elemento che ha costo minore. Il `NativeMeanHeap` è costruito attorno al tipo di dato che deve contenere, cioè il `MinHeapNode`. In memora nativa, i dati sono

```
public struct MinHeapNode
{
    public Entity NodeEntity { get; }
    public int2 ParentPosition { get; set; }
    public int2 Position { get; }
    public int F_Cost { get; }
    public int H_Cost { get; }
    public int Next { get; set; }
}
```

Sorgente 17: Struct MinHeapNode.

memorizzati in un buffer identificato da un puntatore di tipo `(void *)`, allocato tramite una chiamata `UnsafeUtility.Malloc()` che prende come parametro, oltre alla capacità del buffer stesso, un `Allocator`. I `MinHeapNode` vengono memorizzati nel buffer come se fosse un array, dove ognuno degli elementi punta all'indice del suo successore, in tale array, tramite il campo

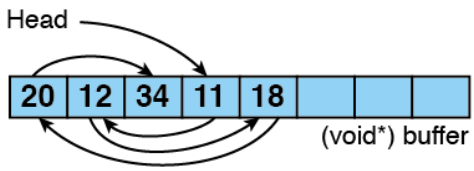



Figura 14: Struttura del min heap.

`int Next`. Una campo della struct `NativeMinHeap` di tipo `int` chiamato `head` punta all'indice

dell'elemento dal costo minimo. Tramite il metodo `Push()` è possibile inserire un `MinHeapNode` all'interno del min heap; esso verrà memorizzato nel buffer e, in base ai propri costi $F(n)$ e $H(n)$, i puntatori agli indici vengono aggiornati per mantenere l'ordinamento crescente dello heap. In caso di equità dei costi $F(n)$, l'ordinamento va in base al costo $H(n)$. Similarmente, è possibile recuperare il nodo dal costo minore tramite il metodo `Pop()`, che restituisce l'elemento puntato da `Head` e lo rimuove dal `NativeMinHeap`. La complessità nel caso peggiore delle due operazioni è costante, $\mathcal{O}(1)$, per l'operazione di `Pop()` mentre è lineare, $\mathcal{O}(n)$, per l'operazione di `Push()`.

```
public void Push(MinHeapNode node)
{
        if (head < 0)
        {
                head = length;
        }
        else if (node.F_Cost < Get(head).F_Cost || (node.F_Cost ==
        ↪  Get(head).F_Cost && node.H_Cost < Get(head).H_Cost))
        {
                node.Next = head;
                head = length;
        }
        else
        {
                var currentPtr = head;
                var current = Get(head);

                while (current.Next >= 0 && Get(current.Next).F_Cost <=
                ↪  node.F_Cost)
                {
                        if (node.F_Cost == Get(current.Next).F_Cost &&
                        ↪  node.H_Cost < Get(current.Next).H_Cost)
                                break;

                        currentPtr = current.Next;
                        current = Get(current.Next);
                }
                node.Next = current.Next;
                current.Next = length;
                UnsafeUtility.WriteArrayElement(buffer, currentPtr,
                ↪  current);
        }
        UnsafeUtility.WriteArrayElement(buffer, length, node);
        this.length += 1;
}
```

Sorgente 18: Metodo Push().

### 5.7.2 Strutture dati condivise

Un altra ottimizzazione implementata è stata quella dell'utilizzo di strutture dati condivise tra tutti gli `Execute()` del `parallelFor` job `AStarJob`. In un primo momento, tali strutture dati, ovvero l'open set, il closed set e i costi $G(n)$, venivano allocate e deallocate per ogni agente processato. Tuttavia, con l'aumentare del numero di agenti da processare, i tempi di allocazione di tali strutture diventa significativo. La soluzione è stata quella di allocare una sola volta i container preposti, con una capacità tale da poter immagazzinare i dati di tutte le entità agenti filtrate dal sistema di pathfinding. Successivamente, all'interno del job `AStarJob`, ogni `Execute()` si prende una parte dei container condivisi tramite il metodo `Slice()`. Tra un iterazione del game loop e l'altra, è necessario quindi solo resettare la porzione di memoria occupata dallo slice dei container tramite un operazione di `MemClear` che è molto più conveniente rispetto alla allocazione e deallocazione della memoria ogni volta. Lo svantaggio è quello che la memoria occupata sarà maggiore, ma i benefici sul tempo impiegato superano di gran lunga tale svantaggio. I container sono quindi dichiarati come campi della classe `AStarSystem`; la allocazione e deallocazione avviene nei metodi `OnStartManager()` e `OnDestroyManager()`. Lo slice dei container in `AStarJob` è guidato dall'indice di cui è dotato il metodo `Execute` in un `IJobParallelFor`: la porzione di memoria restituita sarà quindi compresa tra `(indice * lunghezza)` e `(indice * lunghezza + lunghezza)`.

```
// data containers sliced from shared data structures
var openSet    = OpenSet.Slice(index * _maxLength, _maxLength);
var closedSet  = ClosedSet.Slice(index * _maxLength, _maxLength);
var G_Costs    = Gcosts.Slice(index * _maxLength, _maxLength);

// reset of data containers
openSet.Clear();

void* buffer = closedSet.GetUnsafePtr();
UnsafeUtility.MemClear(buffer, (long)closedSet.Length *
↪  UnsafeUtility.SizeOf<MinHeapNode>());

buffer = G_Costs.GetUnsafePtr();
UnsafeUtility.MemClear(buffer, (long)G_Costs.Length *
↪  UnsafeUtility.SizeOf<int>());
```

Sorgente 19: Slice dei container condivisi e loro reset.

## 5.8 Movimento e collision avoidance

L'ultimo sistema, `RVO2System`, si occupa di gestire il movimento delle unità. Gli agenti devono muoversi lungo il proprio percorso, calcolato da A*, e al contempo devono evitare di collidere fra di loro. La collision avoidance è risolta dalla libreria RVO2, che utilizza un algoritmo chiamato Optimal Relative Collision Avoidance. L'algoritmo e la libreria sono stati sviluppati da Van Den Berg, Guy, Lin, e Manocha presso la North Carolina University nel 2011 [24]. La libreria originale è stata scritta in C++, ma successivamente è stata tradotta anche in C#, ed è disponibile con licenza Apache 2.0. La libreria gestisce il movimento degli agenti calcolando la velocity ottimale che ogni unità dovrebbe avere, al fine di non collidere, basandosi sulla loro posizione attuale. Le posizioni degli agenti sono immagazzinate in una struttura di accelerazione, nello specifico un kdTree. Per ogni agente vengono poi selezionate tutte quelle che si trovano entro un certa soglia, impostabile dal programmatore, sulla quale viene poi calcolata la nuova velocity. Una volta calcolata la velocity per tutti gli agenti, essi vengono aggiornati con i nuovi valori. Riassumendo:

- ogni unità viene mossa secondo la propria velocity;
- viene aggiornato il kdTree con le nuove posizioni;
- vengono calcolati il vicinato e le nuove velocity;
- le unità vengono aggiornate con i nuovi valori.

Recuperare e assegnare la nuova posizione agli agenti è compito del programmatore, mentre invece gli ultimi tre passaggi sono gestiti da RVO2, internamente al metodo pubblico `DoStep()`. Quello che si è fatto è stato spezzare questi passaggi e creare per ognuno un job, abilitando quindi il parallelismo dove possibile. Un altra modifica si è resa necessaria: la libreria non consente di aggiungere o rimuovere agenti alla simulazione dinamicamente. Per implementare efficientemente queste funzionalità si è sostituita la lista, che immagazzina le informazioni sulle unità, con un dizionario, le cui chiavi costituiscono l'identificatore di ogni agente. In quest'ottica sono stati quindi sviluppati quattro job, tutti di tipo `parallelFor`, ad eccezione del job incaricato di aggiornare il kdTree, in quanto le operazioni che effettuano sono eseguite su ogni agente e sono indipendenti fra loro. Nello specifico, `agentsJob` si occupa di muovere le unità, di settarne la posizione obiettivo (che coincide con il waypoint corrente) e di rimuovere gli agenti dalla simulazione una volta arrivati a destinazione. Il job `treeJob`

```
protected override JobHandle OnUpdate(JobHandle inputDeps)
{
    var agentsJob = new RVOJob
    {
        Positions = GetComponentDataFromEntity<Position>(),
        Indexes = new NativeArray<int>(Simulator.Instance.getAgentsKeysArray(), Allocator.TempJob),
        waypoints = GetBufferFromEntity<Waypoints>(),
        commands = _rvoBarrier.CreateCommandBuffer().ToConcurrent()

    }.Schedule(Simulator.Instance.getNumAgents(), 64, inputDeps);
    agentsJob.Complete();

    var treeJob = new BuildKdTreeJob().Schedule(agentsJob);
    treeJob.Complete();

    var numAgents = Simulator.Instance.getNumAgents();
    var indexes = new NativeArray<int>(Simulator.Instance.getAgentsKeysArray(), Allocator.TempJob);

    var stepJob = new RVOStep{Indexes = indexes}.Schedule(numAgents, 64, treeJob);
    stepJob.Complete();

    var updateJob = new RVOUpdate{Indexes = indexes, deltaTime = Time.deltaTime *
    ↪  maxSpeed}.Schedule(numAgents, 64, stepJob);
    updateJob.Complete();

    indexes.Dispose();

    return updateJob;
}
```

Sorgente 20: Scheduling dei job nel sistema `RVOSystem`.

si occupa di ricostruire il kdTree utilizzato per il calcolo efficiente del vicinato di ogni unità. Questa operazione è svolta una sola volta ad ogni iterazione e, internamente, è computata ricorsivamente. La parallelizzazione di funzioni ricorsive è difficile (se non impossibile), poco pratica e spesso inefficiente in termini di risorse. Il job quindi è un semplice `IJob`. Abbiamo poi `stepJob` che si occupa di calcolare, in sequenza, il vicinato di ogni agente sfruttando il kdTree, e sulla base di questo la nuova velocity. Il job `updateJob` infine gestisce l'aggiornamento dei valori delle velocity di ogni unità. Lo scheduling dei quattro job prevede una catena di dipendenze nell'esatto ordine in cui sono stati esposti. È necessario inoltre attendere che ogni job abbia terminato il proprio compito prima di eseguire il job successivo; sono quindi state inserite delle barriere di sincronizzazione mediante il metodo `Complete()`. Il primo job inoltre è l'unico che abbia necessità di un filtraggio: ha infatti bisogno delle componenti `Position` e `Waypoints`, fornite tramite injection. Sono esclude invece le entità che possiedano la componente `Target`, in quante esse non sono ancora state processate dal sistema di pathfinding. I rimanenti job invece hanno bisogno di accedere solo alla simulazione RVO2, che è rappresentata da un singleton, istanza statica della classe `RVO2.Simulation` e quindi accedibile anche senza injection.

### 5.8.1 Optimal Relative Collision Avoidance

Viene ora illustrato, in questa parentesi teorica, il funzionamento di massima dell'Optimal Relative Collision Avoidance. Dati due agenti $A$ e $B$, viene definito $VO^{\tau}_{A|B}$ l'insieme di tutti i vettori velocità relativi di $A$ rispetto a $B$ che inducono a una collisione entro la finestra temporale $\tau$. Geometricamente, $VO^{\tau}_{A|B}$, è rappresentato da un cono troncato il cui vertice coincide con l'origine dello spazio velocità, i cui lati siano tangenti alla circonferenza avente come raggio $r_A + r_B$ centrata in $p_B - p_A$, troncato dalla circonferenza $(r_A + r_B)/\tau$ in posizione $(p_B - p_A)/\tau$. Siano $v_A$ e $v_B$ le velocità dei due agenti: la definizione ci indica che se $v_A - v_B \in VO^{\tau}_{A|B}$ allora avverrà una collisione. Viceversa se $v_A - v_B \notin VO^{\tau}_{A|B}$ allora abbiamo la certezza che non ci sarà collisione almeno fino al tempo $\tau$. In generale, per ogni insieme di velocità $V_B$, se $v_B \in V_B$ e $v_A - v_B \notin VO^{\tau}_{A|B} \oplus V_B$ allora $A$ e $B$ non collideranno nella finestra temporale $\tau$ ($\oplus$ indica la somma di Minkowski). Questo porta alla definizione dell'in-

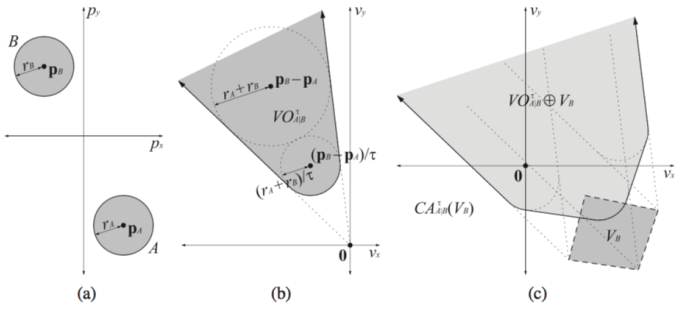
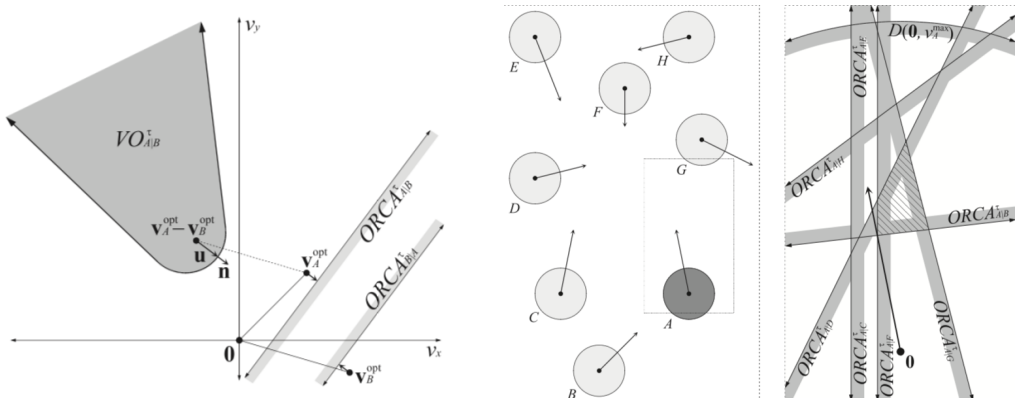



Figura 15: Velocity obstacle. $(a)$ agenti A e B. $(b)$ geometria di $VO^{\tau}_{A|B}$. $(c)$ somma di Minkowski [24]

sieme delle velocità *collision-avoiding* $CA^{\tau}_{A|B}(V_B)$ per $A$ posto che $B$ selezioni la sua velocity da $V_B$:

$$CA^{\tau}_{A|B}(V_B) = \{v \mid v \notin VO^{\tau}_{A|B} \oplus V_B\}$$

Gli insiemi $V_A$ e $V_B$ sono reciprocamente collision-avoiding se vale che $V_A \subseteq CA^{\tau}_{A|B}(V_B)$ e $V_B \subseteq CA^{\tau}_{B|A}(V_A)$. Se gli insiemi coincidono allora sono detti reciprocamente *massimali.* Date queste definizioni, quello che si ricerca sono gli insieme delle velocità $V_A$ e $V_B$ che le rispettino, ovvero che garantiscano la non collisione degli agenti $A$ e $B$ fino al tempo $\tau$. Esistono infinite coppie di insiemi che soddisfano tali requisiti; tra di loro ne esiste un paio che massimizza la quantità di velocità che si avvicina a quella ottimale. Questi insiemi prendono il nome di $ORCA^{\tau}_{A|B}$ e $ORCA^{\tau}_{B|A}$. Siano $v_A^{opt}$ e $v_B^{opt}$ le velocità assunte dalle unità, sia $u$ il vettore che va da $(v_A^{opt} - v_B^{opt})$ al punto più vicino sul confine del velocity obstacle (il cono troncato) e sia $n$ la




(a) Semipiani definiti dall'ORCA (b) ORCA con $n$ agenti.

Figura 16: Rappresentazione geometrica dell'ORCA. [24]

normale al punto toccato da $u$. Allora $u$ rappresenta il più piccolo cambiamento della velocità relativa di $A$ rispetto a $B$ per evitare la collisione. Ogni agente condivide la responsabilità della collision avoidance, quindi $A$ adatta la sua velocità di un fattore $\frac{1}{2}u$ confidando che l'altra metà sia assunta da $B$. Geometricamente, l'insieme $ORCA^{\tau}_{A|B}$ per $A$ è rappresentato dal semipiano nella direzione di $n$ definito dal punto $v^{opt}_A + \frac{1}{2}u$:

$$ORCA^{\tau}_{A|B} = \{v \mid (v - (v^{opt}_A + \frac{1}{2}u))\, n \geq 0\}$$

L'insieme $ORCA^{\tau}_{B|A}$ per $B$ è definito simmetricamente. Nel caso siano presenti *n* agenti, verrà trovato un semipiano $ORCA$ per ognuno di loro. L'intersezione delle regioni rappresenta l'insieme delle velocità ottimali che permette all'agente considerato di evitare tutti gli altri. La velocità selezionata sarà quella, in tale insieme, che più si avvicina alla velocità ideale dell'agente, ovvero quella che è in direzione della posizione obiettivo.

## 5.9 Considerazioni finali

La realizzazione di questo progetto è stata un sfida per vari motivi, in primis l'acquisizione dell'approccio data-oriented nella risoluzione del problema. Inoltre la mancanza di una documentazione completa, chiara ed esaustiva, oltre al grado di immaturità delle API, è sicuramente stato un ostacolo durante la fase di implementazione, superabile solo con un processo di trial and error e di esamina dei progetti esempio ufficiali. Ricordiamo infatti che l'intero framework

Unity ECS e gli Unity Jobs sono stati rilasciati al pubblico in modalità preview e che le API non sono in alcun modo definitive, ma al contrario sono ancora in fase di sviluppo. Durante il corso dello sviluppo di questo progetto infatti, più volte le API sono state modificate. Ad esempio il sistema di injection delle componenti nei sistemi è stato deprecato poco dopo il termine dello sviluppo del progetto, in favore di un nuovo sistema di iterazione chiamato *chunk iteration*. Questo è segno che, da parte degli sviluppatori, si stanno ancora esplorando varie possibilità per la crescita di questo framework. Nonostante tutto, la scelta di utilizzare questo framework si è rivelata vincente. La modularità del codice scritto e la semplicità con cui è stato e può essere esteso infatti sono dei punti di forza rispetto a un approccio classico object-oriented. Le performance inoltre, grazie anche al parallelismo reso possibile dai job, si sono rivelate essere convincenti. Il guadagno prestazionale verrà quantificato nel dettaglio nel prossimo capitolo.

# Capitolo 6

# Risultati sperimentali

## 6.1 Raccolta dei dati

In questo capitolo vengono esaminate le performance del prototipo sviluppato, comparandole con quelle di una versione analoga, implementata col sistema classico di Unity object-oriented; questa versione è totalmente single threaded. I dati sono stati raccolti su una macchina Alienware 15R4 con processore Intel Core i7-8750H 3.9 GHz hexa core, 8 GB di memoria RAM e scheda grafica Nvidia GeForce GTX 1070. Per valutare gli aspetti essenziali dei due prototipi, quali il framerate complessivo e il tempo di esecuzione dei sistemi, si sono considerati due scenario diversi. In un caso si prende in esame la sostenibilità dell'esecuzione nel tempo, mentre nell'altro avviene uno stress test, separatamente per pathfinding e collision avoidance, per capire quali sono il limiti delle due architetture software. A questo scopo si sono svolte diverse prove con un numero crescente di unità per scoprire quanto bene scalano i due diversi prototipi.

## 6.2 Scenario uno

Il primo scenario vede una generazione continua di agenti. Essi vengono generati in una posizione casuale lungo il lato sinistro e inferiore della mappa. La posizione obiettivo che dovranno raggiungere è una posizione casuale lungo il lato opposto della mappa. Lo scenario è

caratterizzato da un numero massimo di agenti presenti contemporaneamente e dalla quantità di agenti generati in parallelo, nello specifico ogni venti frame. A questo ritmo, quindi, le unità generate vengono processate tutte assieme nello stesso frame dal sistema di pathfinding e successivamente, mediante collision avoidance percorrono il cammino trovato. Giunte alla posizione obiettivo le unità vengono rimosse dalla simulazione. La generazione si ferma nel caso sia stato raggiunto il limite massimo di agenti, per riprendere solo dopo che un numero sufficienti di unità sia stata eliminata, abbassando così il numero totale di agenti presenti.

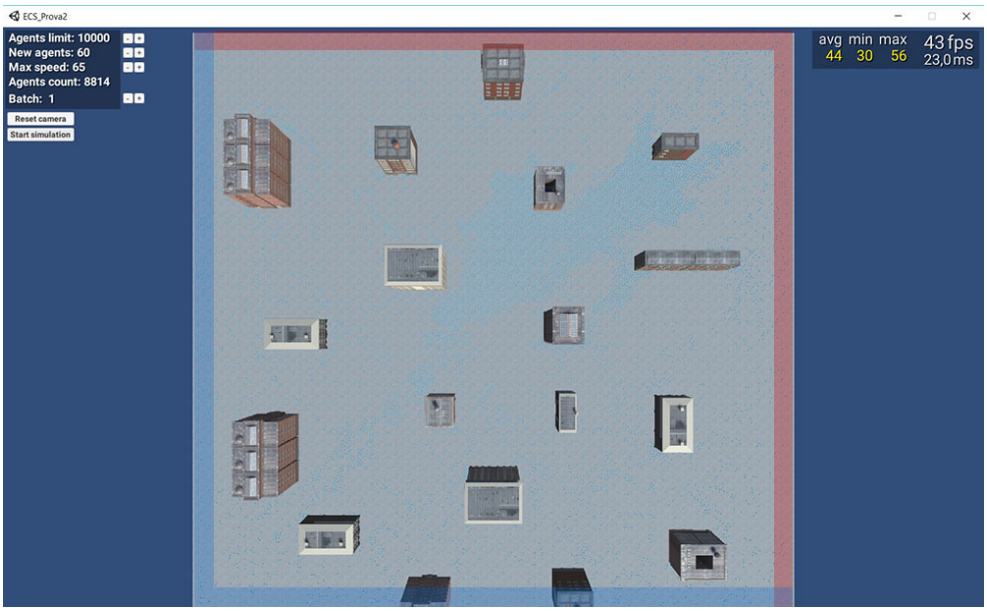



Figura 17: Scenario uno. In blu è evidenziata l'area di partenza mentre in rosso quella di arrivo.

## 6.3 Scenario due

Il secondo scenario prevede due fasi distinte. Nella prima fase vengono generate un certo numero di entità disposte in formazione in un lato della mappa. Esse vengono generate e processate dal sistema di pathfinding ad ogni frame fino al raggiungimento del numero massimo prefissato di unità. Nella seconda fase gli agenti si fanno strada verso il lato opposto della mappa, rimettendosi in formazione una volta arrivate a destinazione. Nello specifico la posizione target è simmetrica, rispetto al centro della mappa, alla posizione iniziale. La scelta di avere posizioni fissate e quindi non casuali come nello scenario uno, è stata presa per avere dei risultati coerenti ad ogni esecuzione delle due diverse versioni e poter quindi fare delle comparazioni

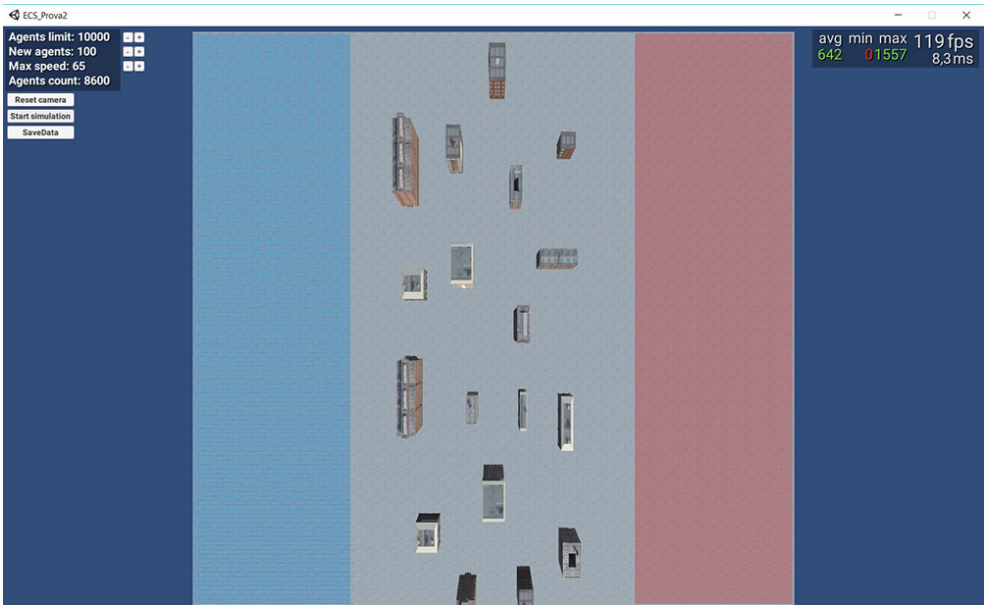


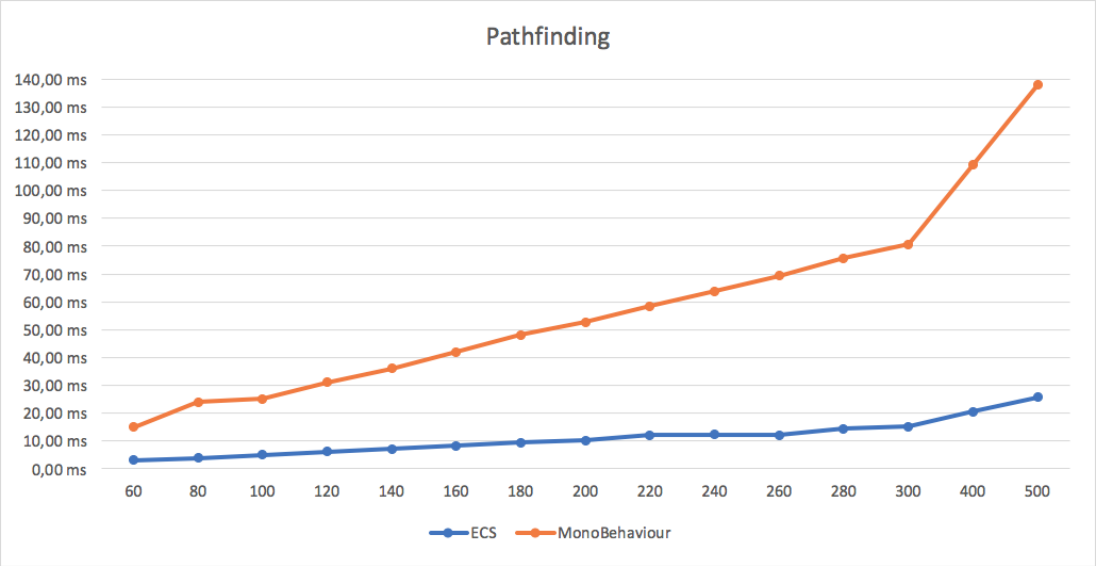

Figura 18: Scenario due. In blu è evidenziata l'area di partenza mentre in rosso quella di arrivo.

più accurate. I parametri quindi, in questo scenario, sono il numero totale di unità e il numero di unità processate ad ogni frame.

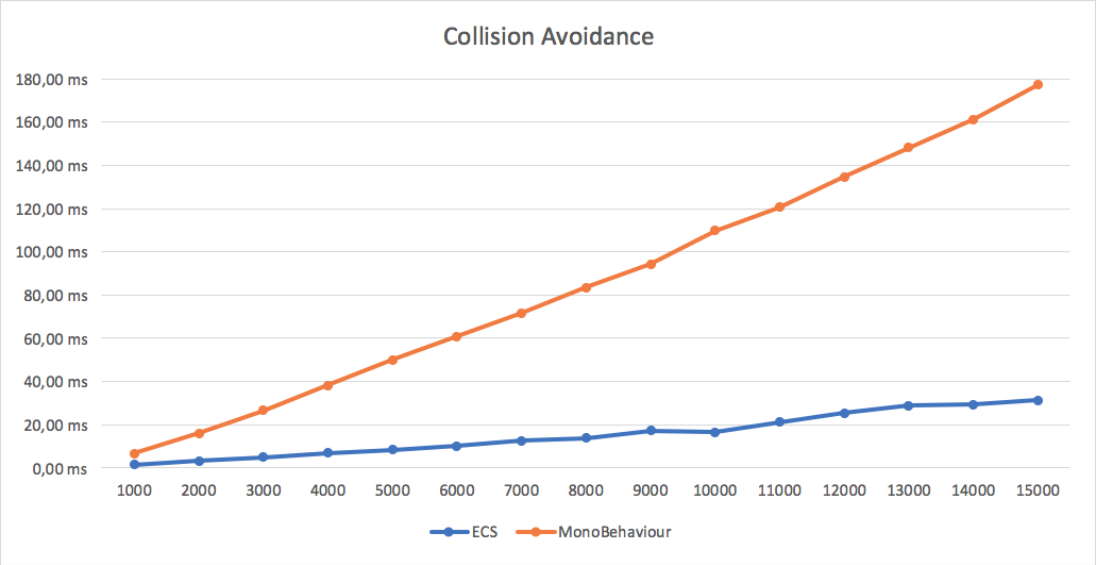
## 6.4 Indice prestazionale

Vengono ora esaminati i tempi di esecuzione e lo speed-up degli stessi ottenuto. Lo scenario di riferimento è lo scenario uno. I dati sono stati raccolti misurando il tempo effettivo di esecuzione, ad ogni frame, dei sistemi `AStarSystem` e `RVOSystem`, nella implementazione ECS, e degli script `MonoBehavoiur` analoghi nella versione object-oriented single threaded. Nel caso del pathfinding, sono state effettuate più misurazioni con un numero crescente di agenti processati per frame, da 60 a 500, mentre nel caso della collision avoidance ad incrementare è il numero totale di agenti presenti nella simulazione, da 1000 a 15.000. I tempi mostrati in tabella sono il risultato di una media dei valori misurati su un campione di mille frame.

Come si può vedere nei grafici in Figura 19, il tempo di esecuzione, in entrambi i casi, incrementa linearmente al numero di agenti coinvolti. Per entrambi pathfinding e collision avoidance, lo speed-up della versione data-oriented è, a meno di fluttuazioni, costante nella misura di circa 5x. Questo significa che già con un numero limitato di agenti il guadagno è notevole, tuttavia su grandi numeri i millisecondi risparmiati diventano davvero significativi.

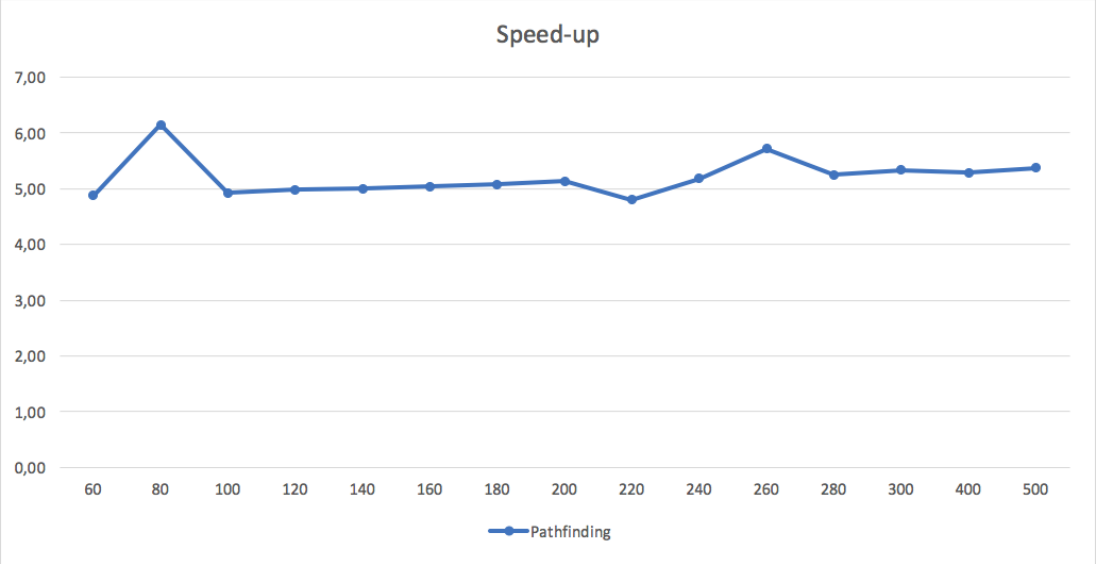



(a) Pathfinding.

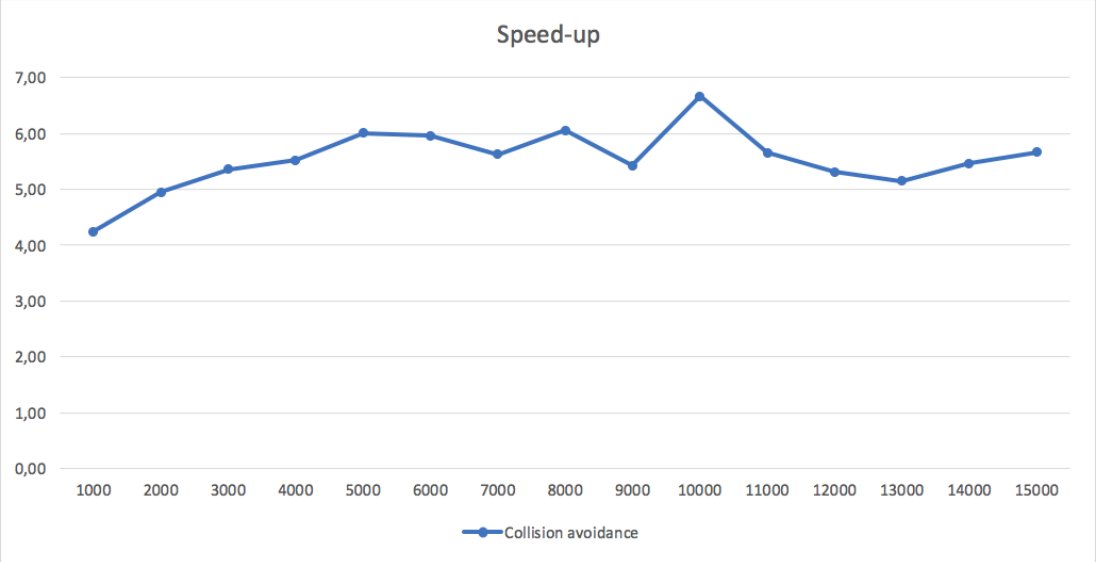



(b) Colllision avoidance.

Figura 19: Grafici dei tempi di esecuzione.




(a)




(b)

Figura 20: Grafici degli speed-up di (a) pathfinding e (b) collision avoidance in relazione al numero di agenti coinvolti.

Si può notare infatti come, nella versione sequenziale, i tempi di esecuzioni siano proibitivi in un contesto real time già a partire da 100 unità processate per frame nel caso del pathfinding, e a partire da 4.000 unità presenti contemporaneamente nel caso della collision avoidance. La versione data-oriented multithreading invece restituisce performance real time in tutte le prove per quanto riguarda il pathfinding, mentre nel caso della collision avoidance incomincia ad essere problematica la presenza di almeno 13.000 unità. Si è inoltre testato, per quanto riguarda il sistema di pathfinding, il peso del parametro *numero di batch* dello scheduling del parallel job `AStarJob`. Si è trovato che il valore ideale di 1 è anche, in questo caso, quello più performante, mentre aumentando il numero di batch i tempi di esecuzione si allungano fino ad addirittura quadruplicare per un numero di 64 batch. I test sono stati effettuati sul processamento di 100 unità per frame. I risultati, esposti nella tabella qui sotto, sono tempi di esecuzione medi calcolati su di un campione di circa sessanta frame.

| **Batch count** | 1 | 2 | 4 | 8 | 16 | 32 | 64 |
|---|---|---|---|---|---|---|---|
| **Tempo** | 5,70 ms | 6,60 ms | 6,98 ms | 6,81 ms | 9,31 ms | 12,63 ms | 20,26 ms |

## 6.5 Prestazioni generali

In questa sezione vengono esaminate le performance generali delle due versioni del prototipo. A questo scopo lo scenario utilizzato è lo scenario due. Le prestazioni sono misurate secondo il framerate dell'applicazione, quindi in frame al secondo (fps). Sono stati scelti per i parametri *agenti processati per frame* e *agenti totali presenti* dei valori di riferimento, ovvero 200, 300, 400 e 500 agenti processati per frame per il pathfinding e 10.000, 15.000 e 20.000 unità totali per la collision avoidance. Sono state quindi eseguite tante prove quante le combinazioni fra questi valori, registrandone il framerate per la durate di dodici secondi. Le prove sono state effettuate nuovamente poi, spegnendo il render delle unità, per valutare quanto esso incida sul framerate complessivo. Ci sono infatti situazioni in cui non è necessario renderizzare le unità, ad esempio in applicazioni server che si occupano di computare solo la logica di gioco, delegando il rendering ai client. Anche in questo caso, lo speed-up della versione entity-component-system si attesta intorno alle cinque volte, confermando il risultato precedente. Inoltre, questa versione è riuscita a garantire un framerate fluido, oltre i 60 fps per molte delle prove effettuate, e comunque sempre sopra ai 30 fps (ad eccezione della collision avoidance per 20.000 unità, con un valore di 23 fps). La versione object-oriented single threaded invece non arriva

mai nemmeno ai 30 fps, con un valore massimo, per il pathfinding di 200 unità a frame di soli 14 fps e con valori minimi nell'ordine dei 4 fps. Le cose migliorano solo leggermente senza il rendering delle unità. In sostanza, la versione sequenziale non è sufficientemente prestante per avere delle performance real time, mentre la versione ECS supera agilmente questa prova. Nelle tabelle seguenti sono mostrati i valori degli speed-up ottenuti e dei framerate medi delle versioni ECS e MonoBehaviour.

Speed-up pathfinding

| **Unità** | **200** | **300** | **400** | **500** | **media** |
|---|---|---|---|---|---|
| **Render** | 5,10 | 5,28 | 5,14 | 5,06 | **5,15** |
| **No render** | 4,88 | 4,75 | 5,24 | 5,37 | **5,06** |

Speed-up collision avoidance

| **Unità** | **10.000** | **15.000** | **20.000** | **media** |
|---|---|---|---|---|
| **Render** | 6,02 | 5,70 | 5,61 | **5,78** |
| **No render** | 5,71 | 5,51 | 5,17 | **5,46** |

FPS medi pathfinding

| **Unità** | **200** | **300** | **400** | **500** |
|---|---|---|---|---|
| **ECS** | 84,08 | 63,56 | 49,18 | 39,89 |
| **ECS no render** | 110,18 | 75,27 | 60,59 | 48,69 |
| **MB** | 14,72 | 12,04 | 9,55 | 7,88 |
| **MB no render** | 22,56 | 15,84 | 11,56 | 9,06 |

FPS medi collision avoidance

| **Unità** | **10.000** | **15.000** | **20.000** |
|---|---|---|---|
| **ECS** | 50,25 | 31,05 | 22,62 |
| **ECS no render** | 56,47 | 35,49 | 24,82 |
| **MB** | 8,34 | 5,44 | 4,03 |
| **MB no render** | 9,89 | 6,44 | 4,80 |

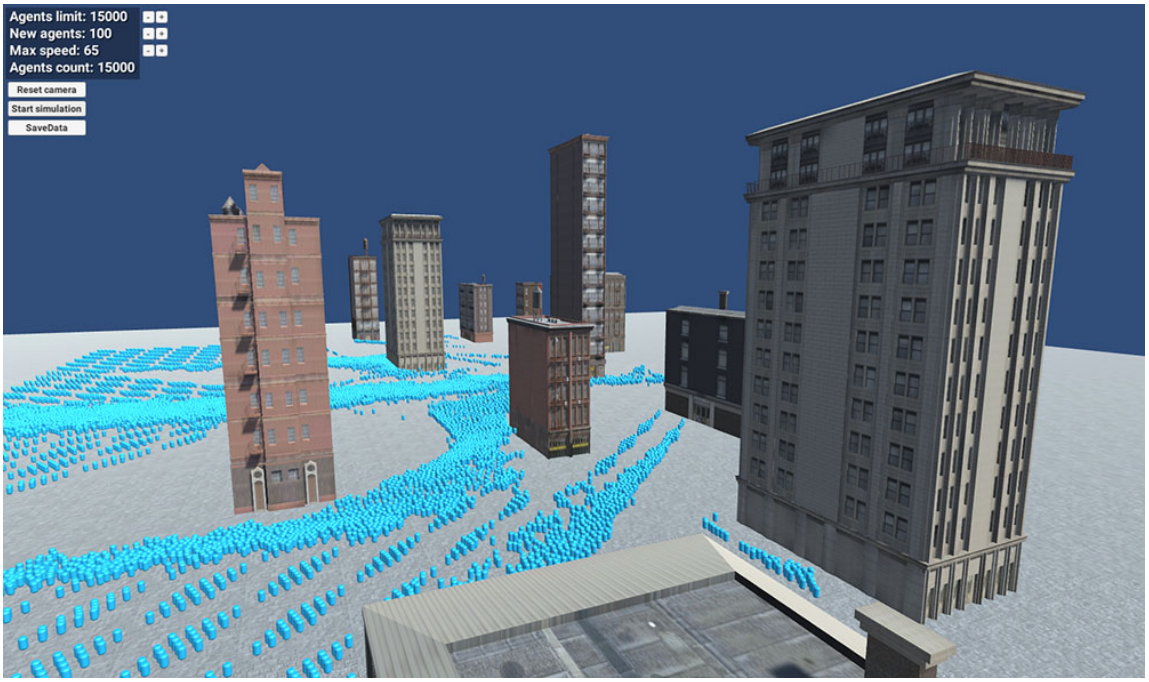
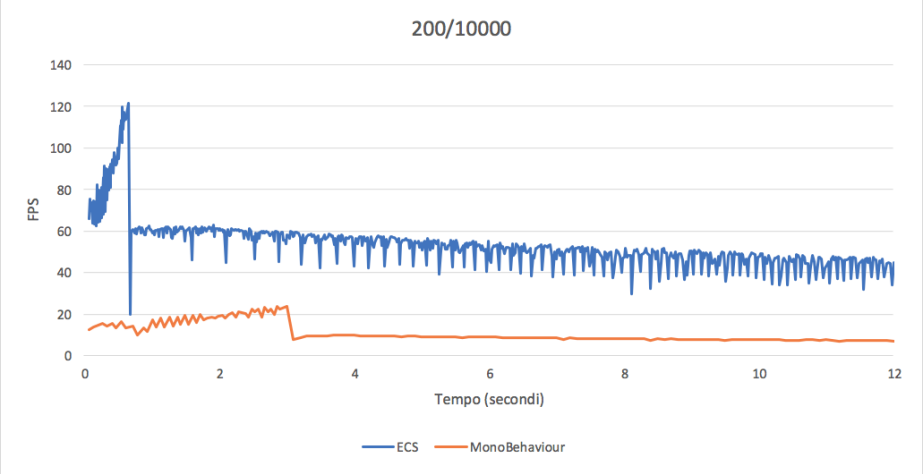
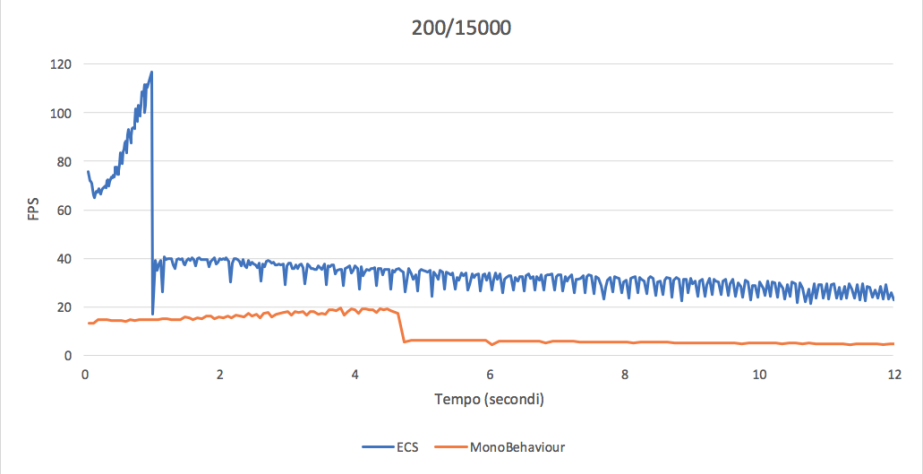



Figura 21: Scenario due, dettaglio.

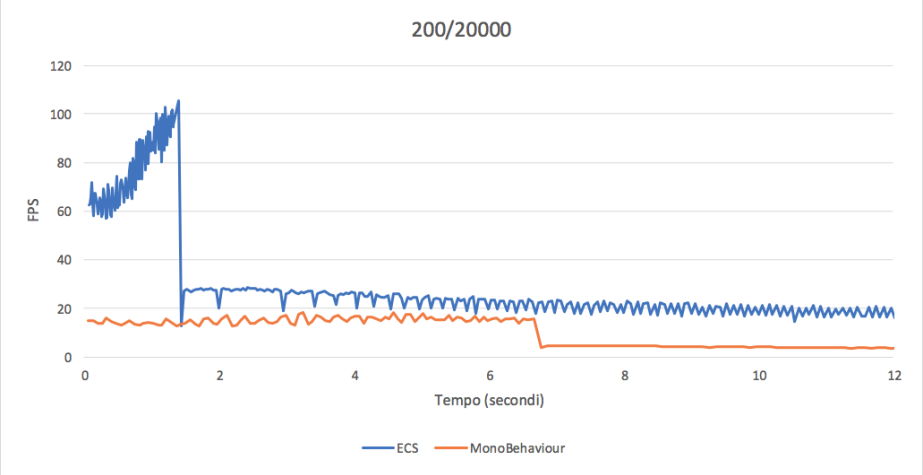
Di seguito vengono mostrati i grafici dell'andamento del framerate nel tempo per tutte le prove effettuate nello scenario due. Nei grafici si vede chiaramente la separazione tra le due fasi. Nella prima fase, dove avviene il processamento delle unità da parte del pathfinding, l'andamento del framerate è crescente. Questo perché, durante la generazione delle unità, ad esse è assegnata una posizione progressiva che va da l'estremo della mappa verso l'interno. Poiché la posizione target è speculare rispetto al centro, le unità generate dopo avranno dei percorsi più corti che quindi verranno processati tendenzialmente più un fretta. Un'altra cosa da notare è che tra la prima e la seconda fase, nella versione ECS, c'è un frame con valore fps molto più basso rispetto ai successivi. Questo è spiegato dal fatto che i sistemi sono automaticamente disattivati qualora non abbiano alcuna entità da processare. Il sistema `RVOSystem` durante la prima fase non ha nulla da processare e quindi è disattivato. Al primo frame della seconda fase c'è quindi un overhead dovuto alla sua riattivazione. Un'altra cosa che si può notare è che il framerate nella seconda fase, nella versione data-oriented, è meno regolare rispetto alla versione single threaded, che invece è molto stabile. Questo fatto è probabilmente dovuto all'overhead introdotto dalla gestione dei job e dei relativi thread.




(a) 10.000 unità totali.

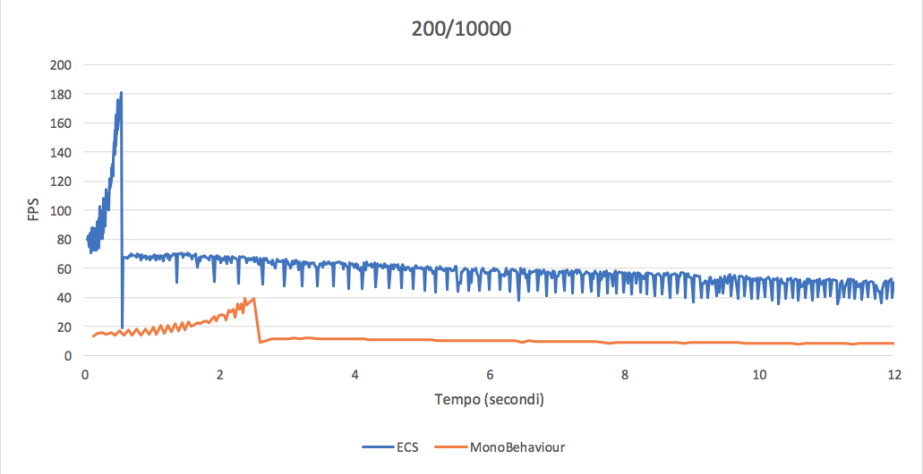



(b) 15.000 unità totali.

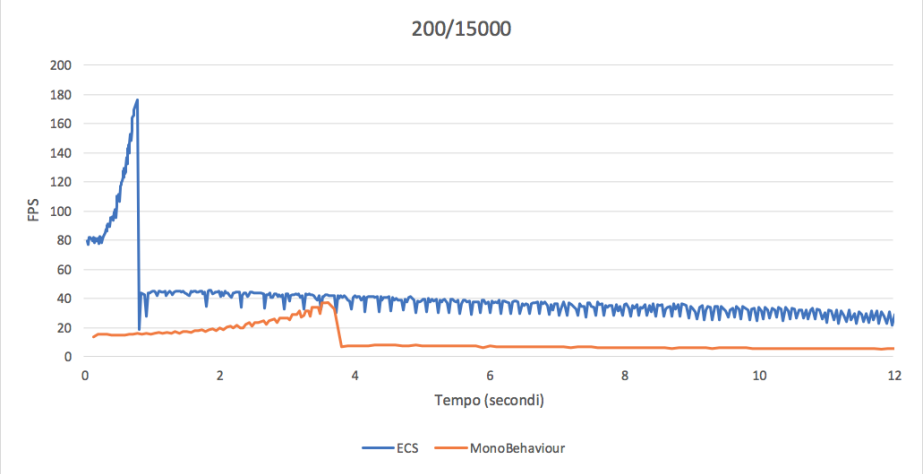



(c) 20.000 unità totali.

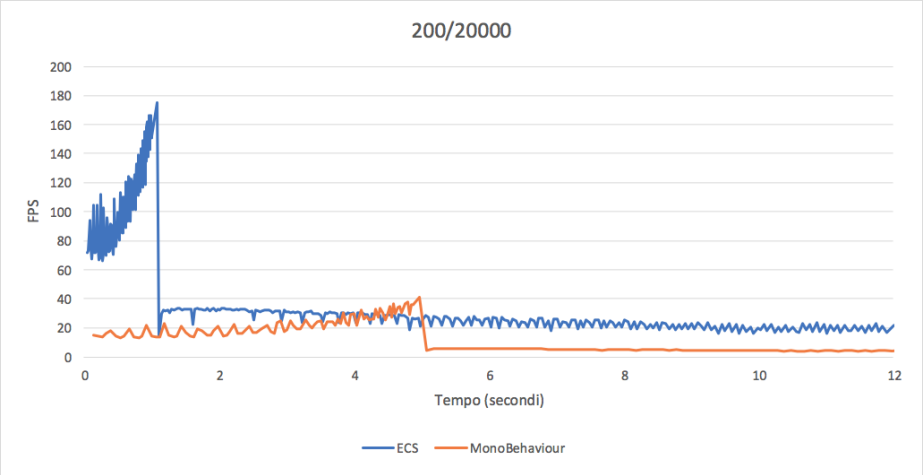
Figura 22: Andamento del framerate per 200 unità processate ogni frame.




(a) 10.000 unità totali.

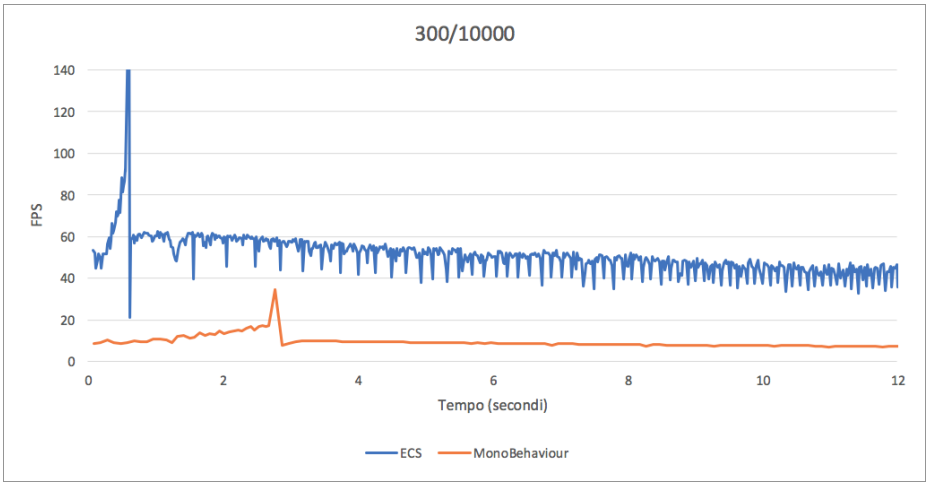



(b) 15.000 unità totali.

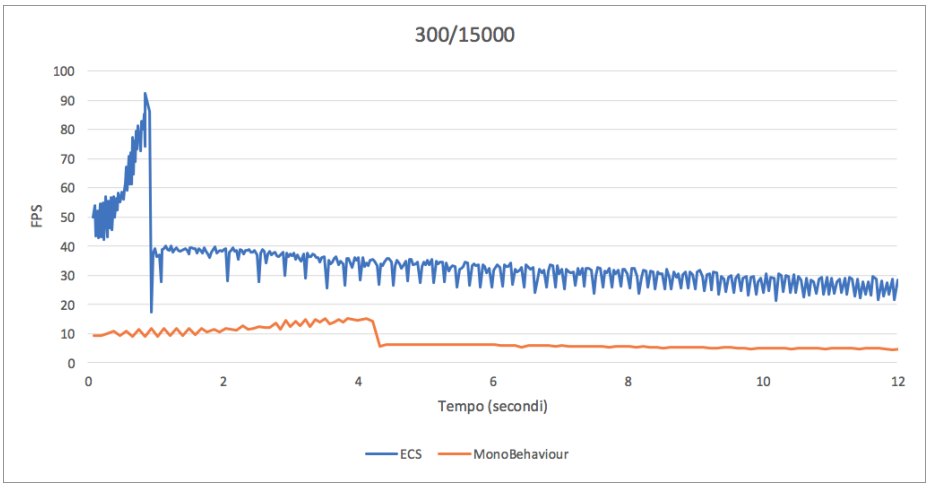



(c) 20.000 unità totali.

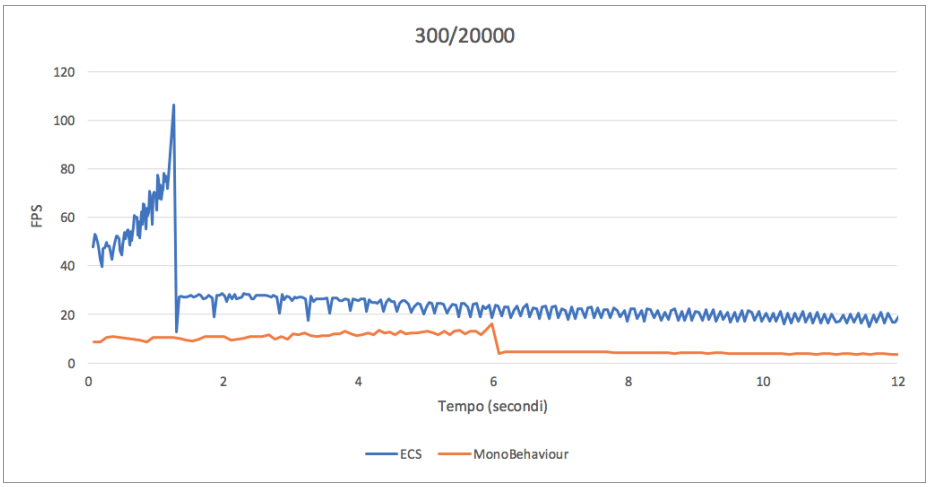
Figura 23: Andamento del framerate per 200 unità processate ogni frame (render unità disabilitato).




(a) 10.000 unità totali.

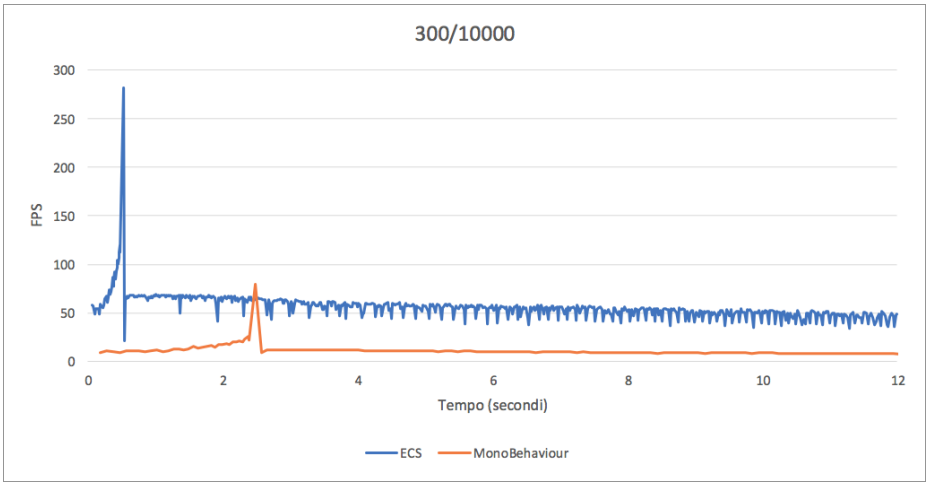



(b) 15.000 unità totali.

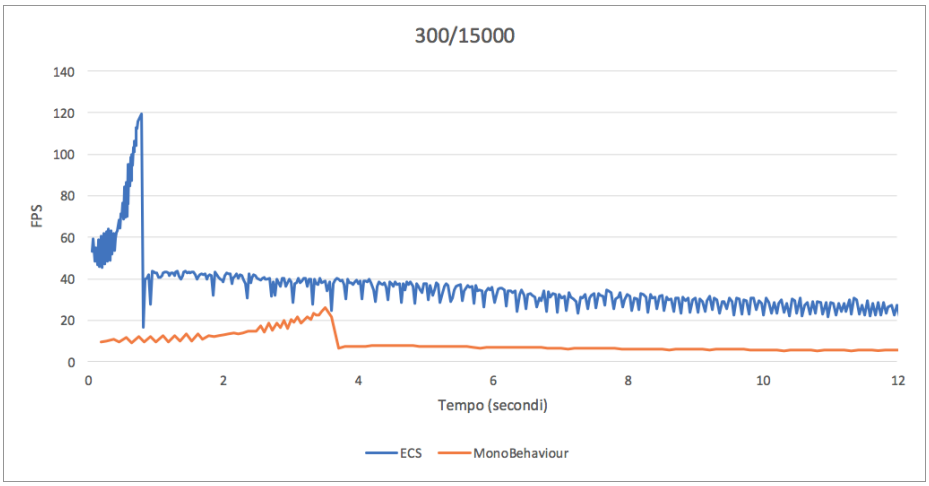



(c) 20.000 unità totali.

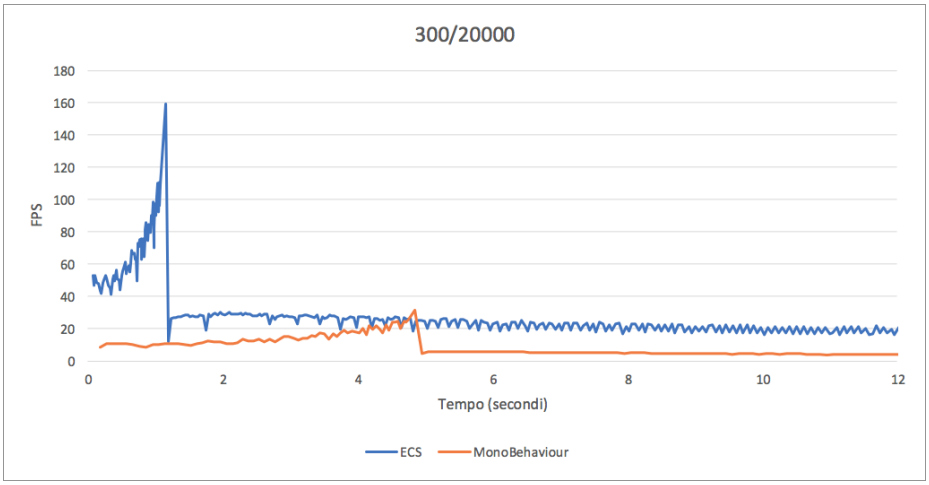
Figura 24: Andamento del framerate per 300 unità processate ogni frame.




(a) 10.000 unità totali.

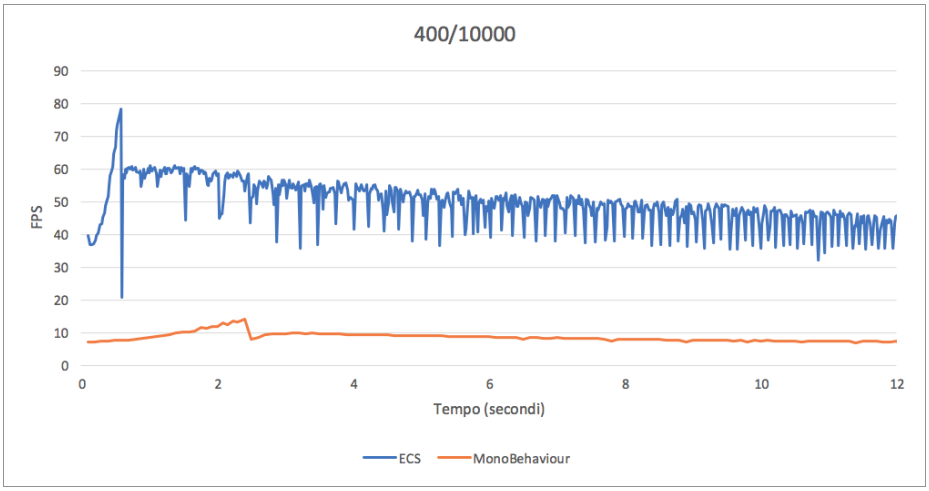



(b) 15.000 unità totali.

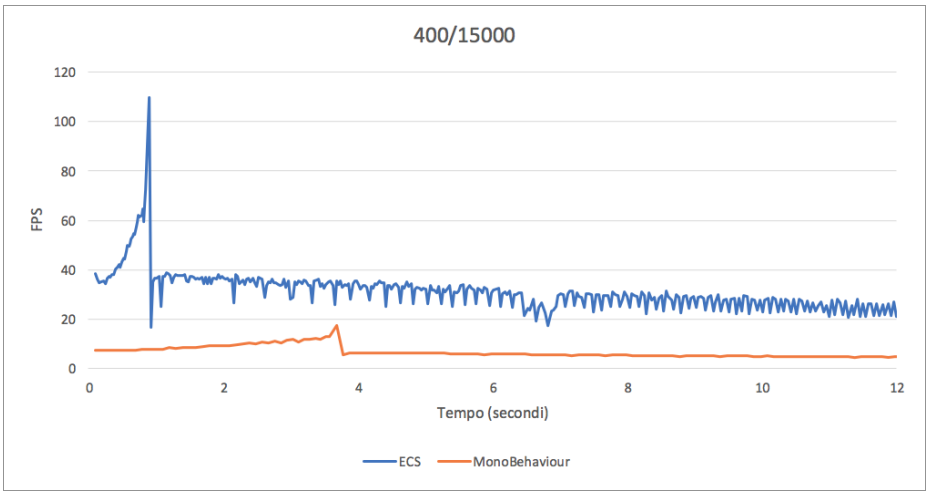



(c) 20.000 unità totali.

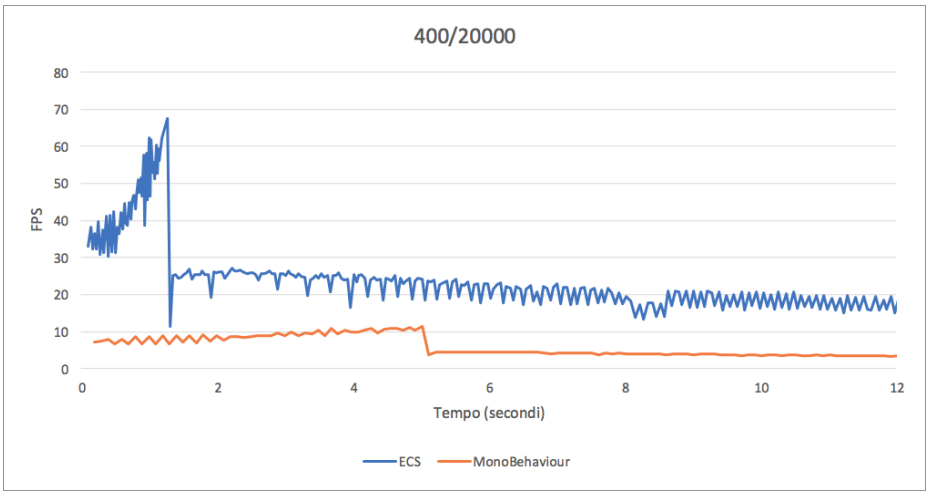
Figura 25: Andamento del framerate per 300 unità processate ogni frame (render unità disabilitato).




(a) 10.000 unità totali.

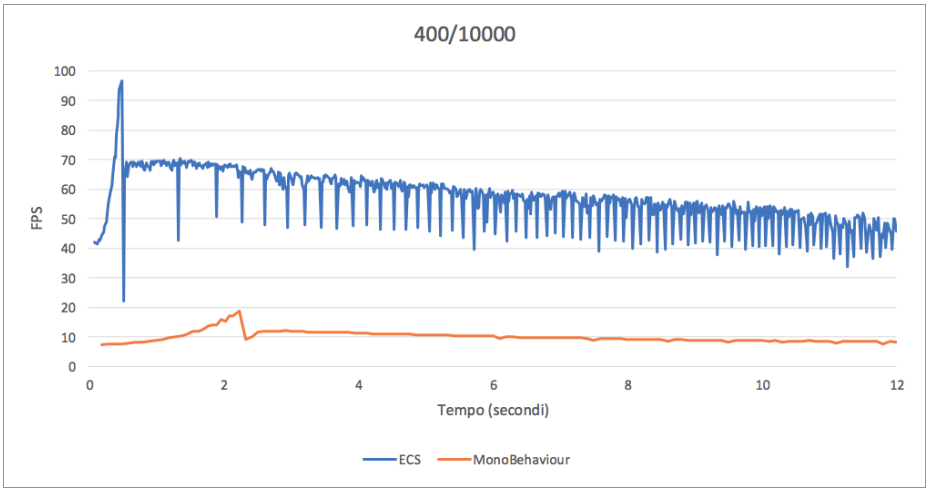



(b) 15.000 unità totali.

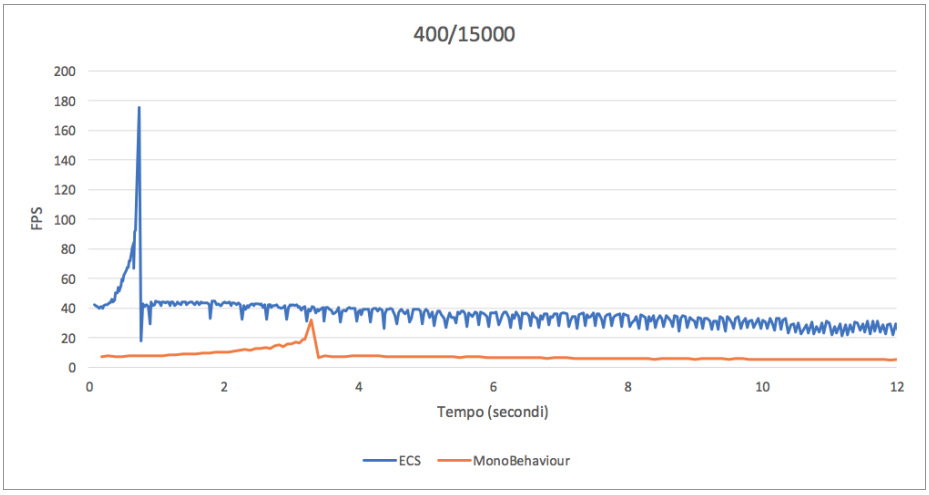



(c) 20.000 unità totali.

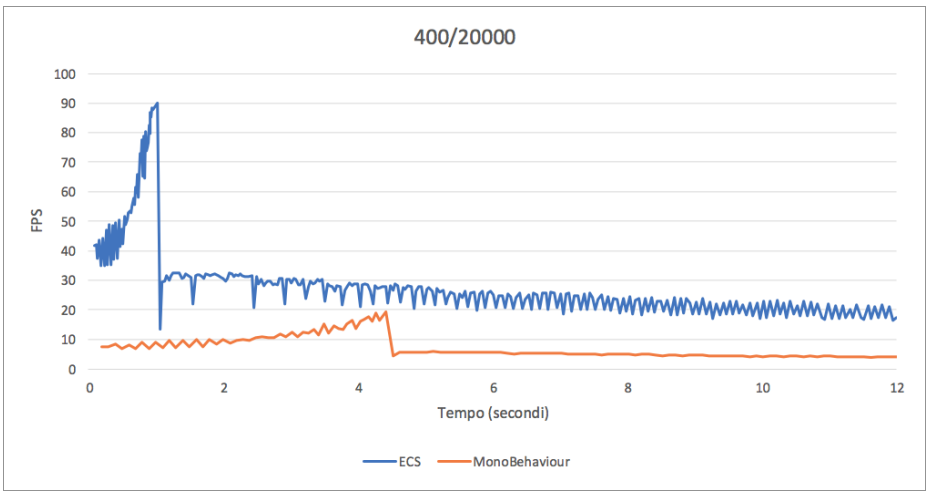
Figura 26: Andamento del framerate per 400 unità processate ogni frame.




(a) 10.000 unità totali.

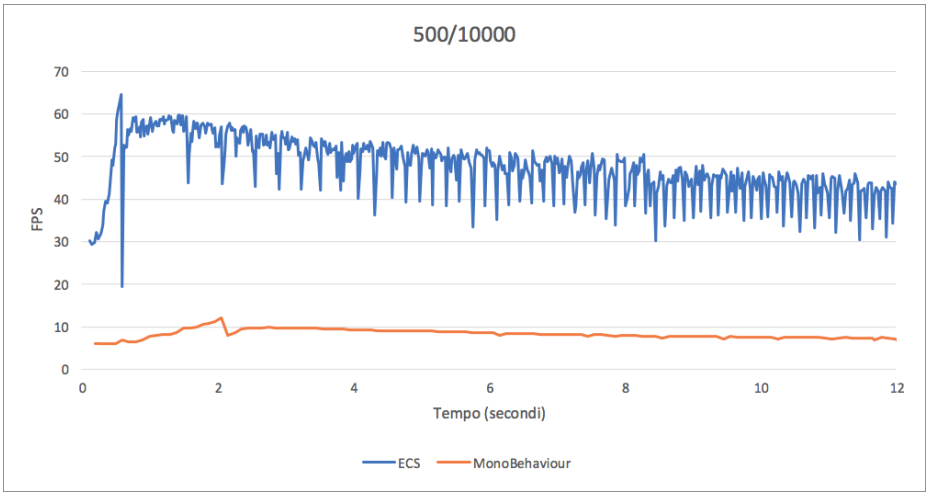



(b) 15.000 unità totali.

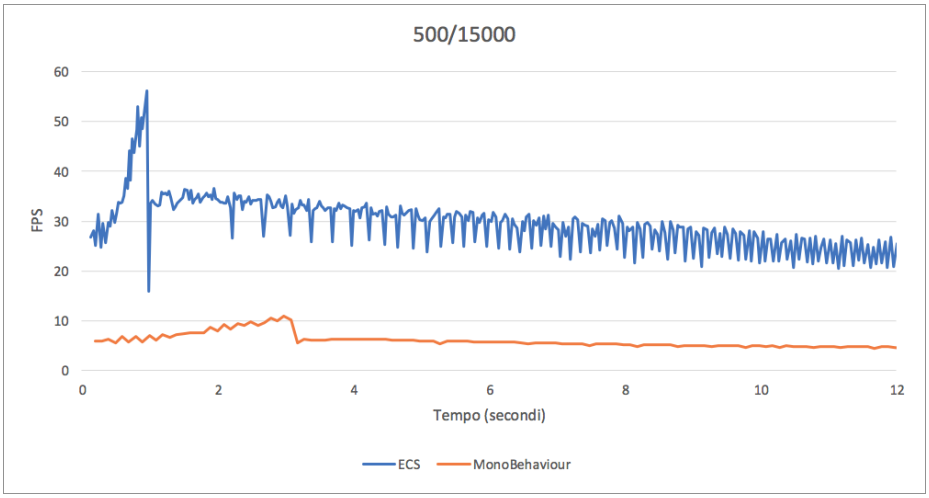



(c) 20.000 unità totali.

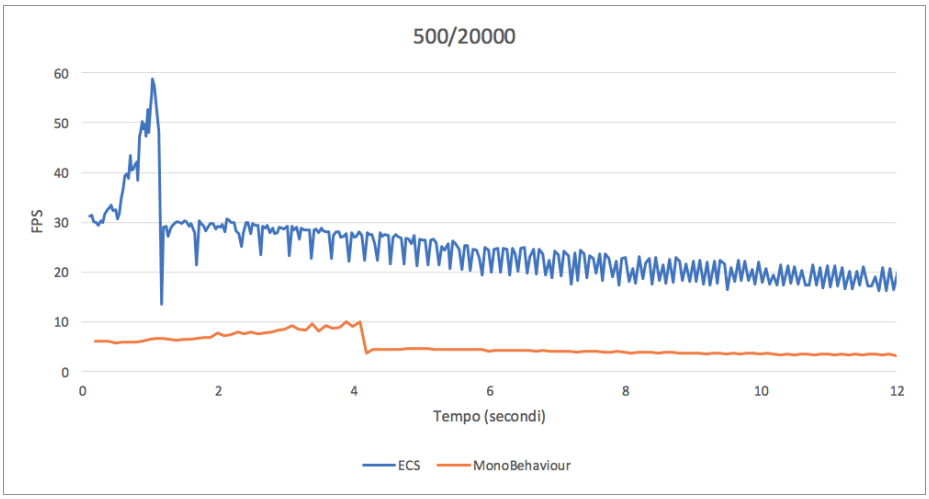
Figura 27: Andamento del framerate per 400 unità processate ogni frame (render unità disabilitato).




(a) 10.000 unità totali.

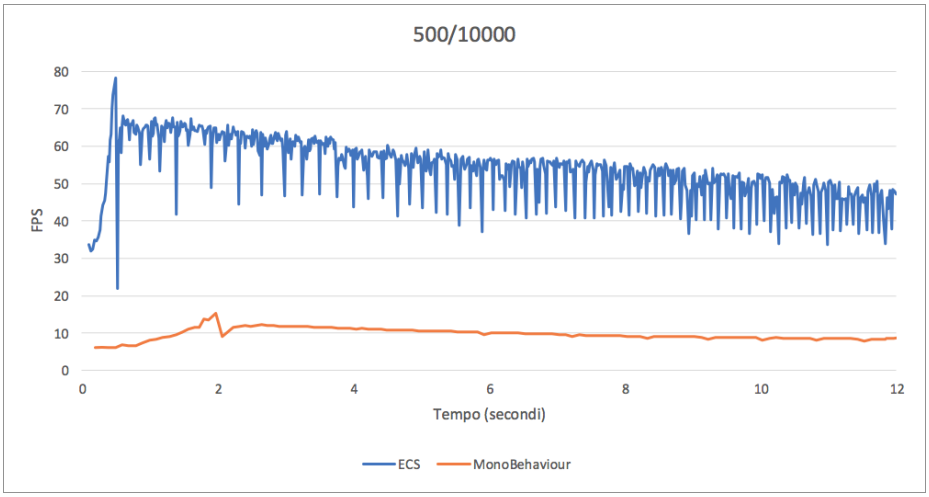



(b) 15.000 unità totali.

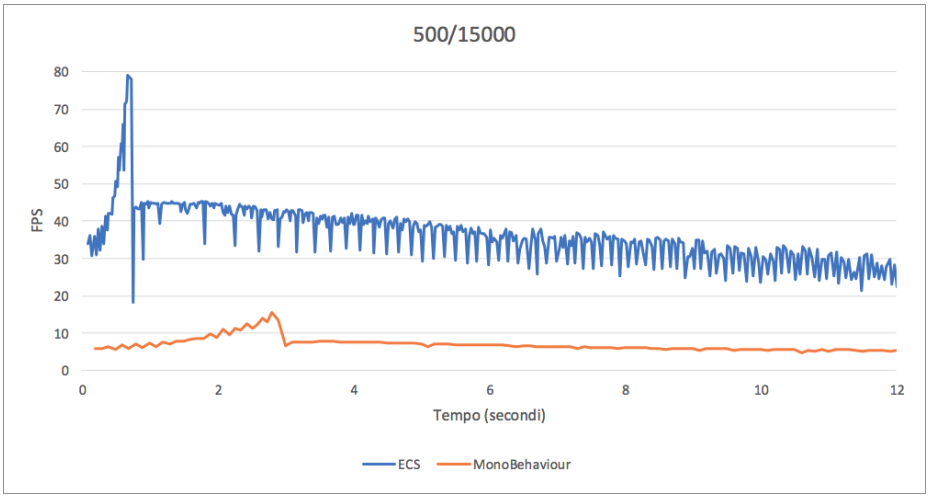



(c) 20.000 unità totali.

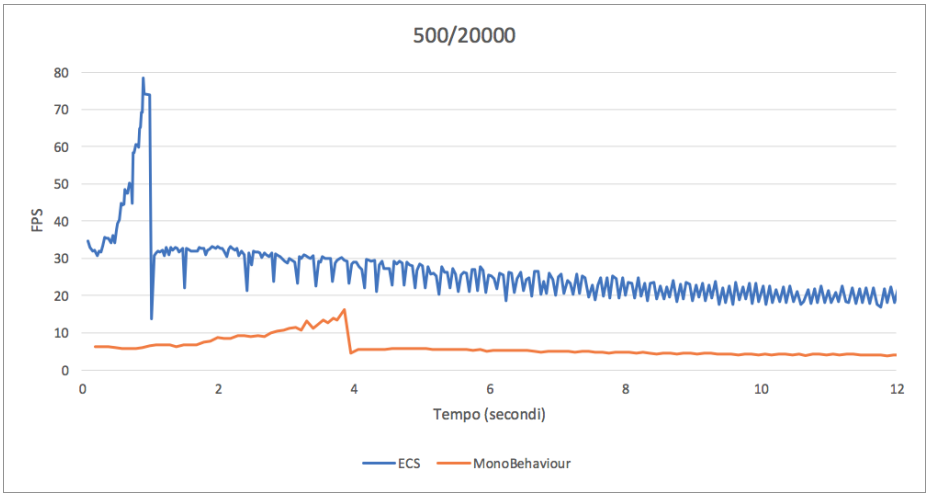
Figura 28: Andamento del framerate per 500 unità processate ogni frame.




(a) 10.000 unità totali.




(b) 15.000 unità totali.




(c) 20.000 unità totali.

Figura 29: Andamento del framerate per 500 unità processate ogni frame (render unità disabilitato).

# Capitolo 7

# Conclusioni

In questo lavoro di tesi si è voluto indagare sui potenziali benefici di una tecnica data-oriented multi threading nella risoluzione del problema della scalabilità del pathfinding all'interno di un game engine. Le performance del prototipo, sviluppato secondo l'approccio data-oriented di entity-component-system, sono risultate essere soddisfacenti. I dati raccolti dimostrano l'efficacia di questa tecnica in comparazione al classico approccio object-oriented sequenziale. Lo speed-up di 5x, seppur minore rispetto a tecniche GPGPU, è comunque significativo e in molti casi può fare la differenza tra performance real time e performance non real time. Ha inoltre il vantaggio di lasciare libera la GPU di occuparsi solo del render grafico e non richiede dispendiosi trasferimenti di memoria. Lo speed-up ottenuto è dipendente dal numero di core della macchina su cui si è raccolto i dati, in questo caso sei. Naturalmente, con un numero maggiore di core si beneficerà di uno speed-up ancora maggiore, aumentando quindi la scalabilità del pathfinding. Anche dalla prospettiva del software design i vantaggi sono evidenti. L'approccio data-oriented ha consentito lo sviluppo di un prototipo estremamente modulare, facilmente estendibile e con una buona leggibilità. Queste caratteristiche facilitano lo sviluppo del software da parte degli sviluppatori, potenzialmente riducendone i tempi necessari, al netto di un periodo di adattamento al nuovo modo di programmare e di approcciare il problema.

## 7.1 Sviluppi futuri

In questa tesi ci si è focalizzati sul problema della scalabilità del pathfinding utilizzando l'algoritmo A*. Lavori ulteriori sono necessari per capire quanto il numero di core CPU a disposizione infici sullo speed-up ottenibile; sarebbe utile scoprire se l'aumento delle prestazioni è lineare al numero di core e se c'è un limite a tale numero, oltre il quale le performance non aumentano più oppure l'overhead introdotto dalla gestione dei thread annulli i vantaggi ottenibili. Possibili sviluppi futuri potrebbero essere quelli della verifica dell'efficacia di questa tecnica data-oriented multi threading a metodologie di pathfinding più complesse, quali gli approcci gerarchici, molto utilizzati nel game development, o a varianti specializzate di A*. Un'altro approccio al pathfinding, da sviluppare secondo il pattern entity-component-system, potrebbe essere quello della precomputazione dei cammini minimi e delle tecniche di flow field. Un'altra possibilità è quella di considerare mappe dinamiche in qui gli ostacoli cambino e di conseguenza anche il grafo su cui viene computato A*. In futuro sono molte le possibilità che il data-oriented design può offrire nel contesto dei game engine; sarebbe infatti interessante verificare i benefici di queste tecniche anche in altre aree del game development oltre al pathfinding e all'intelligenza artificiale.

# Bibliografia


[1] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[2] Unity Technologies. Unity Website. `https://unity3d.com`, 2018 (visitato il 19 febbraio 2019).

[3] Unity Technologies. Unity ECS and Job System. `https://unity.com/unity/features/job-system-ECS`, 2018 (visitato il 19 febbraio 2019).

[4] NVIDIA Corporation. CUDA: Compute unified device architecture programming guide. `https://developer.nvidia.com/cuda-zone`, 2018 (visitato il 19 febbraio 2019).

[5] NVIDIA Corporation. Nvidia website. `https://www.nvidia.com/`, 2018 (visitato il 19 marzo 2019).

[6] Avi Bleiweiss. GPU accelerated pathfinding. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 65–74. Eurographics Association, 2008.

[7] Giuseppe Caggianese and Ugo Erra. Parallel hierarchical A* for multi agent-based simulation on the GPU. In *European Conference on Parallel Processing*, pages 513–522. Springer, 2013.

[8] Sandy Brand and Rafael Bidarra. Multi-core scalable and efficient pathfinding with Parallel Ripple Search. *Computer Animation and Virtual Worlds*, 23(2):73–85, 2012.

[9] Yichao Zhou and Jianyang Zeng. Massively Parallel A* Search on a GPU. In *AAAI*, pages 1248–1255, 2015.

[10] Aljosha Demeulemeester, Charles-Frederik Hollemeersch, Pieter Mees, Bart Pieters, Peter Lambert, and Rik Van de Walle. Hybrid path planning for massive crowd simulation on the gpu. In *International Conference on Motion in Games*, pages 304–315. Springer, 2011.

[11] Blizzard. Starcraft website. `https://starcraft.com/`, 2019 (visitato il 19 marzo 2019).

[12] Bungie. Destiny website. `https://www.destinythegame.com/`, 2019 (visitato il 19 marzo 2019).

[13] Michael L Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, 1987.

[14] Ariel Felner. Position paper: Dijkstra's algorithm versus uniform cost search or a case against dijkstra's algorithm. In *Fourth annual symposium on combinatorial search*, 2011.

[15] Dana S Nau. Expert computer systems. *Computer*, 16(2):63–85, 1983.

[16] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,, 2003.

[17] Richard E Korf. *Artificial intelligence search algorithms*. Computer Science Department, University of California, 1996.

[18] P. E. Hart, N. J. Nilsson, and B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, July 1968.

[19] DICE. DICE website. `http://www.dice.se/`, 2019 (visitato il 19 marzo 2019).

[20] DICE. Frostbite website. `https://www.ea.com/frostbite/`, 2019 (visitato il 19 marzo 2019).

[21] Electronic Arts DICE Follow. A step towards data orientation. `https://www.slideshare.net/DICEStudio/a-step-towards-data-orientation`, 2010 (visitato il 27 febbraio 2019).

[22] Codemasters. Codemasters website. `http://www.codemasters.com/`, 2019 (visitato il 19 marzo 2019).

[23] Rpetrusha. Microsoft docs. Blittable and non-blittable types. `https://docs.microsoft.com/en-us/dotnet/framework/interop/blittable-and-non-blittable-types`, (visitato il 27 febbraio 2019).

[24] Jur Van Den Berg, Stephen J Guy, Ming Lin, and Dinesh Manocha. Reciprocal n-body collision avoidance. In *Robotics research*, pages 3–19. Springer, 2011.

[25] Unity Technologies. Native Container Attribute. `https://docs.unity3d.com/ScriptReference/Unity.Collections.LowLevel.Unsafe.NativeContainerAttribute.html`, 2018 (visitato il 2 marzo 2019).