ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA
Corso di Laurea in Ingegneria e Scienze Informatiche

IMPLEMENTAZIONE E ANALISI DEL MODELLO FLOCKING CON FLAME GPU

Elaborato in
HIGH PERFORMANCE COMPUTING E SISTEMI COMPLESSI

Relatore
Prof. MORENO MARZOLLA

Presentata da
LUCA DELUIGI

Anno Accademico 2018 – 2019

"Raise your quality standards as high as you can live with, avoid wasting your time on routine problems, and always try to work as closely as possible at the boundary of your abilities. Do this, because it is the only way of discovering how that boundary should be moved forward." — Edsger W. Dijkstra

Indice

Introduzione

In questa tesi verranno studiati i modelli basati su agenti e i software al supporto delle simulazioni ad agenti. In particolare sarà approfondito il modello Flocking, che riguarda la formazione di stormi di uccelli in volo. Questo modello è parte della libreria di un software per simulazioni ad agenti chiamato NetLogo, ma verrà implementato con un nuovo ambiente di sviluppo ad alte prestazioni che fa uso della GPU, chiamato FLAME GPU. Questa nuova implementazione permette al modello di essere simulato con una popolazione molto più vasta. È interessante studiare come si comporta il modello secondo alcuni parametri e come cambiano le prestazioni dalla versione in NetLogo. Saranno analizzati e confrontati NetLogo e FLAME GPU, i due principali supporti per realizzare simulazioni ad agenti. In particolare sarà approfondito FLAME GPU, usato nell'implementazione di Flocking. Inoltre sarà discussa anche l'implementazione di un altro modello sulla formazione di stormi a "V" chiamato Flocking Vee Formations.

L'importanza dei modelli ad agenti sta nell'ampio spettro delle possibili applicazioni, con lo scopo di studiare, sperimentare e comprendere fenomeni reali in un ambiente virtuale piuttosto che "in vitro". Le simulazioni permettono di osservare come si evolve un modello tramite un computer. Non tutte le simulazioni richiedono la stessa quantità di risorse: a seconda dell'algoritmo implementato è possibile che una simulazione sia troppo complessa per essere eseguita in tempi utili. Per questo motivo si cerca di sfruttare al massimo le potenzialità dell'hardware disponibile utilizzando algoritmi ad alte prestazioni. Nel caso di FLAME GPU viene applicata un'estrema parallelizzazione delle istruzioni eseguite grazie all'uso delle GPU come hardware per il calcolo.

Le GPU, *Graphics Processing Units*, sono dispositivi originariamente creati e ottimizzati per le operazioni grafiche nei computer. Da diversi anni ormai sono in grado di eseguire compiti *General Purpose*, ossia simili a quelli eseguiti dalla CPU. Il vantaggio delle GPU è quello di un'architettura che comprende molte ALU, *Arithmetic and Logic Unit*, ossia unità di calcolo indipendenti che lavorano in parallelo per risolvere un compito più velocemente di quanto sia in grado di fare la CPU, che generalmente possiede meno ALU ma lavora ad una frequenza di clock maggiore ed una latenza più bassa.

Capitolo 1

I Modelli basati su agenti

In questo capitolo saranno illustrati i concetti chiave degli ABM (Agent Based Models) e le loro più importanti caratteristiche e applicazioni, con particolare attenzione al modello "Flocking" e alla sua variante "Flocking Vee Formations".

1.1 Definizione

I *Modelli basati su agenti* sono una particolare classe di modelli computazionali che fa uso del concetto di agente. Un modello *computazionale* è una descrizione matematica e formale di un processo, implementabile in un computer, che permette di riprodurre un fenomeno reale in un ambiente virtuale. Tra i possibili casi di studio in cui è interessante la costruzione di un modello del genere vi sono i *sistemi complessi*, ossia sistemi composti di più parti che interagiscono tra loro, in cui la visione globale mostra caratteristiche interessanti, difficilmente ricavabili o deducibili osservandone le parti in modo individuale. Grazie all'informatica molti sistemi complessi sono stati tradotti in codice ed eseguiti su delle macchine per poter essere studiati.

Tra i più importanti elementi che fanno parte di un sistema complesso vi sono gli *agenti*. Un **agente** è un'unità autonoma in grado di interagire con le altre unità e con l'ambiente, seguendo uno specifico insieme di regole. Un **modello basato su agenti** è pertanto un insieme di definizioni di tali agenti, dell'ambiente in cui sono immersi e del insieme di regole che governa ogni elemento che fa parte del sistema e ogni interazione possibile. Infine, la **simulazione di un modello** è la messa in pratica delle definizioni e delle regole allo scopo di osservare quelle caratteristiche che il modello è stato costruito per descrivere.

1.2 L'approccio bottom-up

I modelli basati su agenti si occupano di definire i singoli attori di un sistema complesso, ossia gli agenti, l'ambiente in cui essi vengono immersi e un insieme di regole che governano le interazioni tra tutti i componenti del sistema. Questo tipo di approccio è riconducibile all'approccio *bottom-up* della progettazione software, in quanto si occupa primariamente di definire i concetti di basso livello (i singoli agenti, *bottom*) e solo alla fine osserva i risultati ad alto livello (i comportamenti del gruppo e i dati analitici, *up*). L'idea dietro ai modelli basati su agenti è quella di descrivere in modo semplificato i singoli attori del sistema, in modo da considerare solo i dati, i comportamenti e le interazioni strettamente necessarie all'osservazione dei fenomeni, ottimizzando così il consumo di risorse richieste dalla simulazione, in quanto vengono eliminate tutte quelle caratteristiche presenti nella realtà che non contribuiscono in modo percepibile o interessante allo studio del fenomeno in oggetto.

Un esempio di modello, opposto all'approccio bottom-up, è quello basato sulle equazioni, che adotta il metodo *top-down*. Se i modelli ad agenti definiscono i comportamenti a basso livello e osservano quelli ad alto livello, i modelli basati sulle equazioni osservano i comportamenti ad alto livello e li modellano direttamente, senza considerare ciò che avviene alle singole istanze ma considerando il sistema solamente nel suo insieme. Tali modelli, detti EBM (*Equation Based Model*), consistono in un insieme di equazioni che descrivono direttamente i comportamenti e i fenomeni ad alto livello facendo assunzioni approssimate e statistiche su quelli di basso livello [1, pp. 335, 336]. Ad esempio, per descrivere l'andamento demografico di una nazione, non è possibile simulare ogni singolo essere umano vivente come avviene nel film di fantascienza "The Matrix"[2], in quanto non solo non è permesso dalle attuali tecnologie ma sarebbe un grande spreco di risorse. Invece, coi modelli basati sulle equazioni, si fanno delle assunzioni semplificate sul soggetto dello studio e si formula matematicamente un modello che descrive direttamente, in questo caso, l'andamento della popolazione nazionale dei prossimi anni rispetto ai dati raccolti sui precedenti.

La scelta di un approccio piuttosto che un altro solitamente dipende dal tipo di sistema che si vuole modellare e dalle caratteristiche che si vuole osservare. Un modello matematico ad equazioni potrebbe essere troppo complesso per essere realizzato, ma è possibile che un modello ad agenti corrispondente sia troppo complesso per essere simulato e osservato. In questi casi è necessario scendere a compromessi. In altri casi un approccio può essere migliore dell'altro. In particolare, i vantaggi più importanti dell'approccio ad agenti sono [3, pp.12-14]:

- La facilità nella costruzione

- Facilità nella distinzione tra spazio fisico e spazio delle interazioni, che non necessariamente dipende da quello fisico
- L'ulteriore livello di validazione, ossia la possibilità di validare il modello sia osservandolo nell'insieme che in ogni parte singolarmente
- Maggior supporto alle sperimentazioni col modello
- La facilità di traduzione da agenti a entità reali laddove le sperimentazioni portassero risultati utili nella pratica

Un approfondimento di tali vantaggi è disponibile nell'articolo *"Agent-based modeling vs. equation-based modeling: A case study and users' guide"* [3, pp.11-13].

1.3 Principi dei modelli basati su agenti

Due processi fondamentali che la modellazione basata su agenti ha in comune con la programmazione ad oggetti (Object Oriented Programming) sono *astrazione* e *incapsulamento*. L'astrazione è il mezzo principale con il quale si modellano gli agenti: si parte dalle entità reali che fanno parte del sistema e si stabiliscono proprietà, stati e comportamenti. Solitamente però le entità reali non vengono modellate in modo verosimile, ma vengono modellate in funzione del fenomeno che si cerca di studiare. Ad esempio, per spiegare gli urti tra particelle, è possibile prendere il biliardo come modello. In questo caso per la modellazione è interessante il moto e le proprietà fisiche di ogni palla, mentre è superflua l'informazione del numero che la palla ha impresso sulla superficie. Astrarre e selezionare sono due passaggi fondamentali per rendere i modelli ad agenti più semplici da studiare, e per quanto riguarda l'informatica, da implementare. L'incapsulamento è un altro importante principio proprio sia dell'OOP che degli agenti: quando un agente o un oggetto interagisce con un altro, non espone il comportamento e l'intero stato o insieme di proprietà, ma interagisce solamente tramite un sottoinsieme selezionato di informazioni che nascondono dati ritenuti superflui o necessariamente nascosti per come è fatto il dominio reale. Ad esempio se si volesse modellare una partita di scacchi, un giocatore, che sarà un agente, non deve poter sapere cosa pensa il suo avversario in un dato momento. Queste caratteristiche permettono di modellare sistemi complessi i cui agenti si comportano autonomamente, seppur seguendo regole prestabilite, anche senza un'entità superiore che controlla l'intero sistema dall'alto.

1.4 Applicazioni dei modelli ad agenti

Le applicazioni dei modelli basati su agenti intersecano numerose materie. In matematica si studiano gli automi cellulari, in chimica e fisica le simulazioni di interazioni tra particelle, nelle scienze naturali gli effetti dei terremoti, e si potrebbero fare innumerevoli altri esempi. In particolare in questa tesi verrà preso in considerazione un modello usato in Zoologia, riguardante lo studio degli stormi di uccelli.

1.4.1 Caso di studio: il volo degli uccelli

L'argomento di questa tesi è un modello basato su agenti che simula il volo degli uccelli e la formazioni di stormi, noto come "Flocking". Questo modello in realtà mostra più in generale come dei semplici comportamenti individuali possano risultare in comportamenti globali, a livello di sistema, non affatto banali. Lo stesso principio vale per molti animali che si spostano in gruppo, quali ad esempio i pesci.

1.4.2 Il modello "Flocking"

Il modello Flocking è un esempio di modello basato su agenti che descrive il fenomeno del volo degli uccelli, in particolare quello della formazione di stormi di volatili che volano in gruppo, talvolta senza la presenza di un leader o coordinatore che governi gli spostamenti degli altri uccelli. Quando si parla di modello di Flocking però non si fa riferimento ad un univoco e ben definito insieme di entità e di regole, ma ci si riferisce ad un insieme di modelli anche molto diversi fra loro che hanno in comune solamente l'oggetto di studio, ossia la formazione di stormi durante il volo di un gruppo di volatili. Il modello studiato in questa tesi è quello proposto dalla libreria di modelli di NetLogo, vedi sezione 2.1 e [4]. Questo modello è basato su un altro modello di flocking creato nel 1986 da Craig Reynolds [5], che definisce delle regole base sulle quali è possibile costruire diversi modelli che mostrano ad alto livello comportamenti simili di formazione di stormi. Il software creato da Reynolds diede il nome al modello che implementa, detto "Boids".

1.4.3 Boids

Boids è un software scritto in Symbolics Common Lisp da Craig Reynolds [5]. Il modello implementato consiste in un insieme di agenti immersi in uno spazio bidimensionale toroidale, dotati di posizione e direzione di movimento. Ognuno di questi agenti rappresenta un volatile ed è dotato di un limitato

campo visivo, sia in angolo che in lontananza. Ogni individuo viene solamente influenzato dagli altri agenti che è in grado di vedere. Ogni agente segue tre regole fondamentali che accomunano Boids a tutti i suoi derivati, in quanto sono sufficienti per presentare le formazioni di stormi:

- **Separazione** - Ogni agente, se troppo vicino ad un altro, cercherà di allontanarsi dalla "folla" nel modo più veloce possibile
- **Allineamento** - Ogni agente cercherà di adattare la propria velocità (direzione, verso e modulo) a quella degli altri componenti dello stormo
- **Coesione** - Infine, ogni agente cercherà di mantenere lo stormo unito avvicinandosi lentamente al centro dello stormo, calcolato come media delle posizioni dei compagni visibili

Tali regole definiscono un semplice comportamento individuale, considerando i soli tre stati, ma a livello globale queste semplici regole sono sufficienti a simulare il comportamento di stormi di uccelli. Per un corretto funzionamento del modello tuttavia è necessario che la capacità di cambiamento direzionale sia limitata, in modo da evitare che la convergenza del moto da solitario a raggruppato in stormi di un agente risulti istantanea e quindi inverosimile. Inoltre è importante che vi sia un numero sufficiente di agenti rispetto alle dimensioni dello spazio disponibile per far si che avvengano delle interazioni. Un'altra nota interessante sta nel fatto che le implementazioni base del modello e dei suoi derivati più semplici non includono nessun elemento casuale nel comportamento; nonostante ciò è ugualmente possibile osservare comportamenti apparentemente caotici ad alto livello a partire da uno stato iniziale di posizionamento (e direzione) casuale. Infatti, come notato da Reynolds nel suo articolo, gli stormi non si comportano linearmente, ossia le direzioni di due stormi che si intersecano non si sommano ma avvengono dei comportamenti apparentemente caotici quando ciò avviene. Questa e molte altre interazioni tra stormi sono osservabili solo grazie all'esecuzione di simulazioni, di cui si parlerà nel capitolo 2.

1.5 NetLogo Flocking

NetLogo è un ambiente di sviluppo per la programmazione di modelli multi-agente, ossia una classe di modelli che include i modelli basati su agenti, in cui non è necessario che gli individui siano autonomi [6]. Il software include, oltre all'ambiente di sviluppo, una ricca libreria di modelli che è possibile esplorare e studiare sia per capire come funziona la programmazione di un modello sia per

analizzare i risultati delle simulazioni. I modelli della libreria NetLogo sono suddivisi in discipline; Flocking fa parte della sezione Biologia [Figura 1.1].

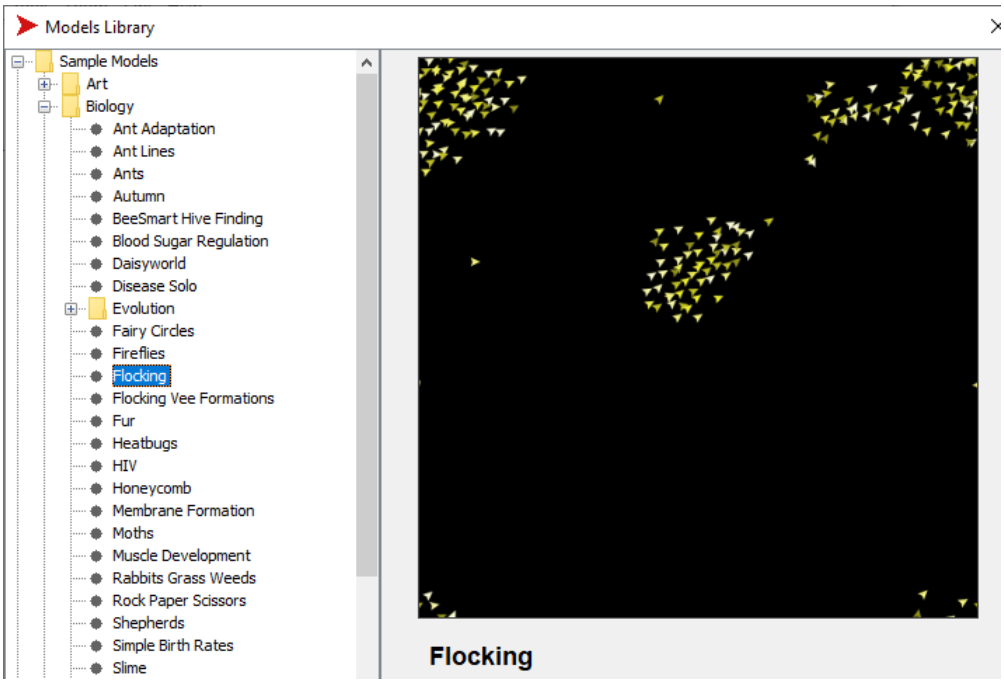

Figura 1.1: Porzione della finestra di selezione dei modelli dalla libreria.

Il modello implementato in NetLogo include una breve documentazione che ne spiega le principali caratteristiche, tra cui il fatto che nonostante l'algoritmo differisca da quello originale di Boids, è sufficiente rispettare i principi delle tre regole base per osservare gli stormi. Il codice è scritto in una variante del Logo, un linguaggio di programmazione creato per scopi educativi [7]. Osservando il codice, è possibile dedurre facilmente un diagramma a stati finiti che mostra il funzionamento di una "tartaruga", ossia di un agente NetLogo [Figura 1.2].

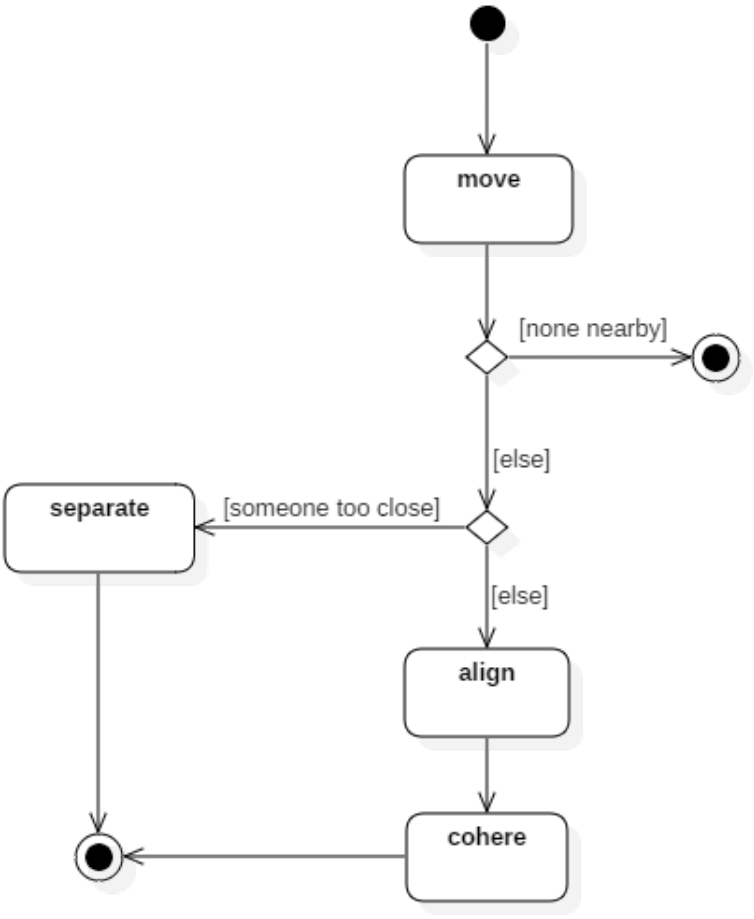

Figura 1.2: Diagramma delle attività di un agente in UML 2.0.

Come è possibile notare dalla presenza di un punto di inizio e uno di uscita, che corrispondono alla chiamata e alla terminazione della funzione "go" del codice, il tempo in NetLogo è discretizzato in una sequenza di iterazioni. Ad ogni iterazione, l'evento *go* viene lanciato, l'agente esegue le operazioni opportune in base alla configurazione dei suoi vicini e ritorna allo stato iniziale, in attesa del prossimo *go*. Rispetto al modello Boids originale, sono state apportate modifiche ai dettagli implementativi dell'algoritmo, come illustrato nelle sezioni successive.

1.5.1 Separazione

Durante la separazione da un altro volatile troppo vicino viene stabilita la nuova direzione del soggetto tramite una rotazione in senso opposto al movi-

mento del compagno. L'angolo della rotazione viene calcolato come differenza tra le direzioni del soggetto e del vicino [Figura 1.3]. Questa implementazione funziona in molti casi ma in alcuni non risulta efficace. In particolare, se la direzione dei due agenti è quasi la stessa, la rotazione del soggetto che tenta di separarsi diventa estremamente lenta o quasi nulla.

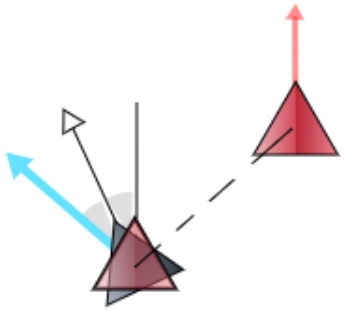

Figura 1.3: L'agente di colore scuro cerca di allontanarsi ruotando simmetricamente la propria direzione rispetto a quella del compagno più vicino, il triangolo rosso. La risultante dipende dal massimo angolo di rotazione ed è la direzione ciano.

Un'altra possibile implementazione della separazione potrebbe essere quella di considerare la direzione che va dalla posizione del soggetto alla posizione del vicino, linea tratteggiata in figura 1.3. Stabilita questa direzione, si cerca di raggiungere la direzione opposta, o meglio, il verso opposto, in modo da allontanarsi il più possibile e più in fretta possibile dal vicino.

1.5.2 Allineamento

L'allineamento consiste nel ruotare verso la risultante del calcolo della media delle direzioni dei compagni visibili [Figura 1.4]. Se agli agenti non fosse imposto un limite di capacità di rotazione, ogni agente che incontra un altro agente in solitudine imposterebbe la propria direzione esattamente come quella del vicino, in un istante e contemporaneamente, continuando a scambiarsi le direzioni, senza mai allinearsi effettivamente.

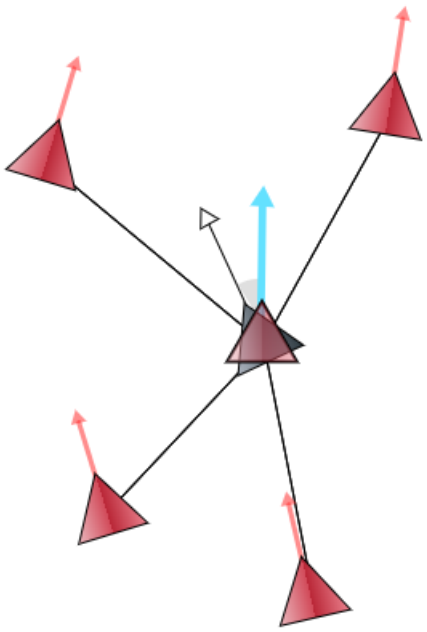

Figura 1.4: L'agente di colore scuro cerca di allineare la propria direzione a quella media degli agenti che vede intorno a se, rappresentata in celeste.

Dopo numerose iterazioni, a seconda dei parametri della simulazione come l'angolo massimo di rotazione per ogni iterazione, è possibile osservare come grazie all'allineamento la maggior parte degli agenti segua la stessa direzione. L'implementazione dell'allineamento in NetLogo corrisponde a quella originale del software Boids.

1.5.3 Coesione

La coesione in generale indica la tendenza di un agente ad avvicinarsi al centro dello stormo. Nell'implementazione di Flocking in NetLogo questo viene calcolato come media delle direzioni dal soggetto verso ogni vicino visibile, mentre in Boids è la direzione verso la media delle posizioni. Le due direzioni non sono equivalenti [Figura 1.5].

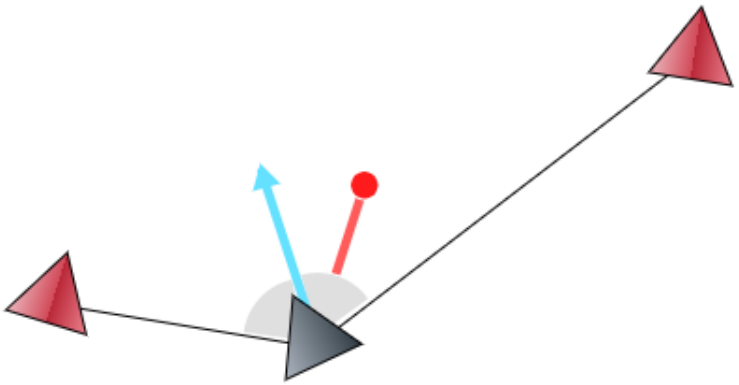

Figura 1.5: La direzione media verso i due vicini, in celeste, è diversa dalla direzione verso la posizione media, in rosso.

1.5.4 Osservazioni

I limiti imposti alla massima rotazione per ognuna delle tre azioni controllano il comportamento dell'intero sistema. Azzerando uno dei tre limiti in questione è come se si rimuovesse la regola corrispondente. La separazione costituisce un'eccezione, in quanto, se se ne azzerasse la rotazione, gli agenti che si trovano troppo vicini continuerebbero a muoversi in linea retta (vedi figura 1.2). Rimuovere alcune regole, o aggiungerne di nuove, e più in generale sperimentare con il modello, è proprio uno dei più importanti vantaggi degli ABM. Ad esempio, sarebbe interessante osservare come cambia il sistema se venisse aggiunto un limite all'ampiezza della vista frontale di ogni uccello in volo. Questa ed altre modifiche a Flocking sono state raccolte in un altro modello della libreria NetLogo, "Flocking Vee Formations".

1.6 NetLogo Flocking Vee Formations

Vee Formations è un adattamento del modello Flocking per produrre stormi a forma di "V". Le regole che gli agenti di questo modello seguono sono diverse da quelle di Flocking, come è possibile osservare nello schema ricavato dall'implementazione in codice NetLogo [Figura 1.6].

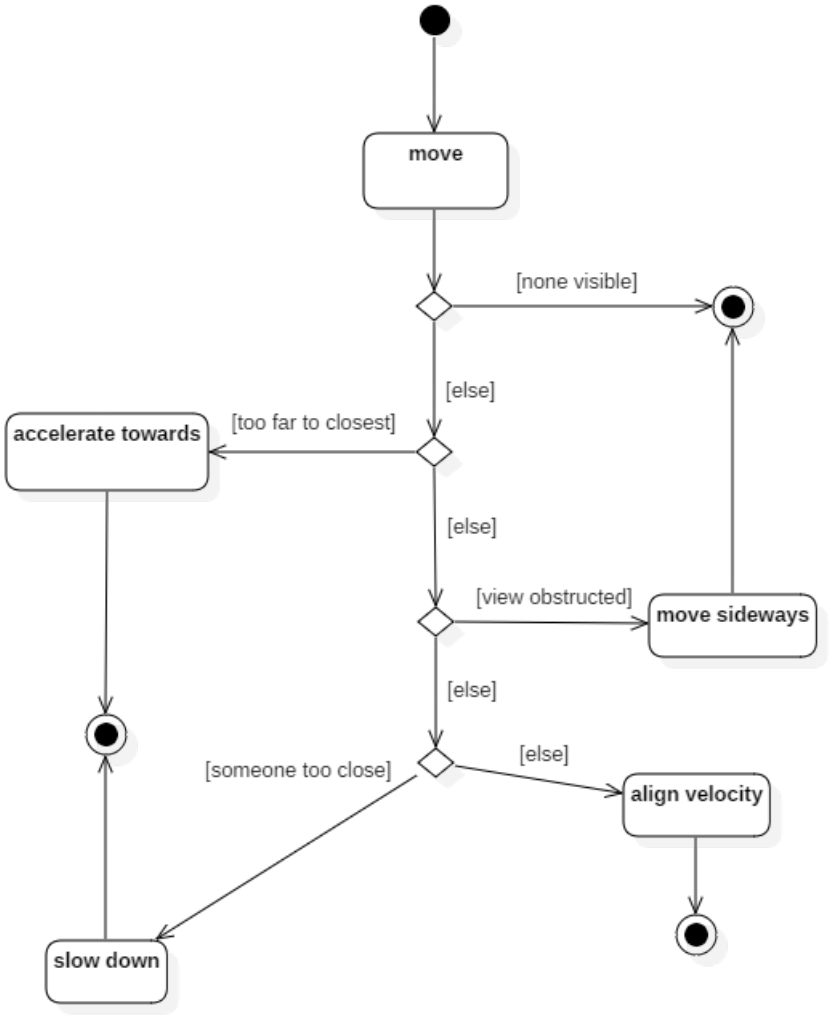

Figura 1.6: Diagramma delle attività di un agente in Flocking Vee Formations, in UML 2.0.

In questo modello gli agenti sono in grado di variare la propria velocità di movimento. In particolare, accelerano se si trovano troppo lontani dal compagno più vicino e decelerano se si trovano troppo vicini. Se la loro vista

è ostruita, secondo un certo parametro che definisce entro quale angolo deve essere presente un compagno per essere considerata tale, l'agente si sposta leggermente di lato per liberare il campo visivo. Quando la vista è libera l'agente adatta la propria velocità (direzione e modulo) a quella del compagno più prossimo. Questo comportamento permette agli uccelli di creare stormi e di modificarne la forma in modo da tendere ad una formazione a V. Non essendo presenti meccanismi per gestire il bilanciamento dei "bracci" delle V, quelle che si osservano sono V solitamente sbilanciate, o addirittura delle linee oblique, ossia delle V a cui manca una metà. In compenso in natura si osservano più spesso formazioni sbilanciate piuttosto che V perfette.

Sperimentando con i parametri della simulazione è possibile osservare come, al crescere della popolazione rispetto alle dimensioni dello spazio percorribile, le formazioni a V si formino più difficilmente. Questo è dovuto alle perturbazioni nel movimento di uno stormo che avvengono quando questo interseca un altro stormo o un singolo uccello che viaggia in direzione trasversale, cosa che avviene più frequentemente quando diminuiscono gli spazi vuoti.

Capitolo 2

Simulazione di un ABM

Osservare come è fatto un modello basato su agenti è sicuramente un passaggio fondamentale per la sua comprensione, ma ancora più importante è osservare i comportamenti ad alto livello che emergono quando il modello ad agenti entra in attività. Lo scopo di una simulazione è quello di mettere in moto un modello di qualsiasi tipo in un ambiente virtuale, risparmiando le costose operazioni necessarie a farlo in vitro o comunque in un ambiente fisico e reale.

Simulare un modello ad agenti non è un compito banale: le principali difficoltà sono le limitate capacità computazionali delle macchine a disposizione e la complessità della definizione del modello. Esistono numerose soluzioni software e hardware specifiche per molte classi di problemi di modellazione e simulazione ad agenti. Nello specifico, verranno analizzate le caratteristiche, i vantaggi e gli svantaggi di due software di modellazione: **NetLogo** e **FLAME**.

2.1 NetLogo

NetLogo [6] è un ambiente di sviluppo per modelli multi-agente, scritto in Java da Uri Wilensky nel 1999, manutenuto dal *Center for Connected Learning and Computer-Based Modeling* da allora. NetLogo è un esempio di software libero, il che ha permesso a molti studenti e ricercatori di farne uso e di espanderne le funzionalità. La versione base di NetLogo viene eseguita sulla Java Virtual Machine, ma è in fase di sviluppo una versione completamente *Web Based*, scritta in Javascript per i browser. Sia la versione desktop che quella web sono pertanto multipiattaforma, in quanto possono essere eseguite su tutti i dispositivi che supportano i moderni browser o l'ambiente di esecuzione Java.

Ogni modello di NetLogo è costituito da tre elementi fondamentali:

- **Interfaccia**, ossia la parte di visualizzazione;

- **Informazioni**, che comprendono la documentazione sintetica del modello e qualche altra informazione utile;

- **Codice**, che definisce i comportamenti e le interazioni delle parti del modello, e parte della visualizzazione.

2.1.1 L'interfaccia

L'interfaccia di un modello è ciò che si vede durante una simulazione. Questo include una visuale dell'ambiente in cui sono immersi i componenti del modello multi-agente, dei pulsanti per controllare la simulazione e qualsiasi altro indicatore o strumento di interazione che il creatore del modello decida di inserire. La creazione di un'interfaccia è semplificata da NetLogo e si riduce ad un *drag and drop* di elementi di GUI (Graphical User Interface) e ad una fase di aggiustamento dei parametri per ogni elemento piazzato. Alcuni controlli, come la velocità della simulazione, sono parte di NetLogo stesso, e sono meno personalizzabili degli altri. Nella parte bassa dello schermo, infine, è presente un interprete di comandi che permette di lanciare comandi di debug o di controllo sulla simulazione, sempre in linguaggio NetLogo [Figura 2.1].

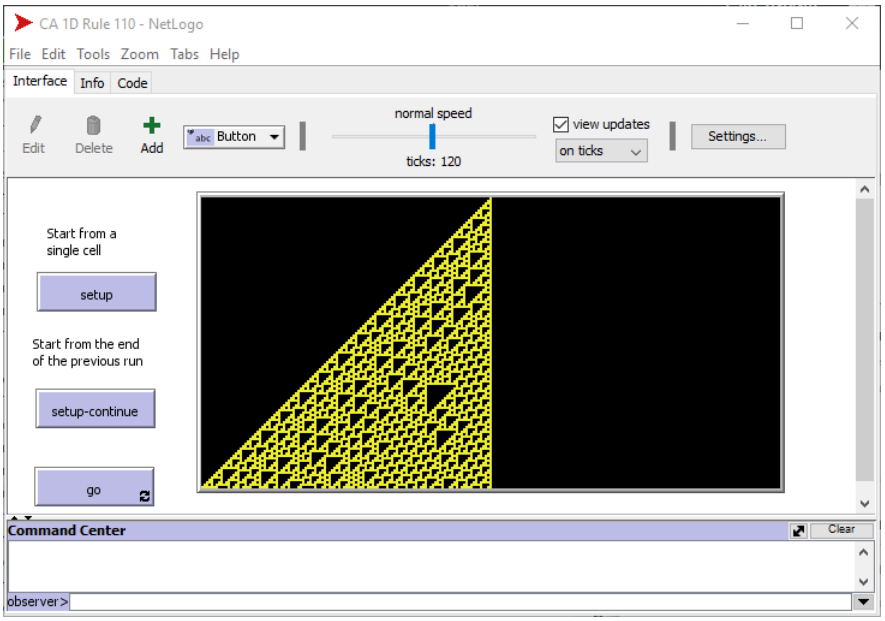

Figura 2.1: Interfaccia di NetLogo. La regione centrale, a sfondo bianco, è l'interfaccia del modello. Al di sotto si trova la shell dei comandi, al di sopra i controlli per modificare l'interfaccia e la barra degli strumenti di NetLogo.

2.1.2 Le informazioni

Ogni modello NetLogo ha la possibilità di includere la propria documentazione sintetizzata nella sezione "Info" accessibile tramite l'ancora visibile in figura 2.1. All'interno della scheda Informazioni si trovano dei paragrafi con rispettive intestazioni contenenti il titolo, solitamente costituito da una breve domanda a cui il paragrafo risponde, e del testo [Figura 2.2]. In alto è sempre

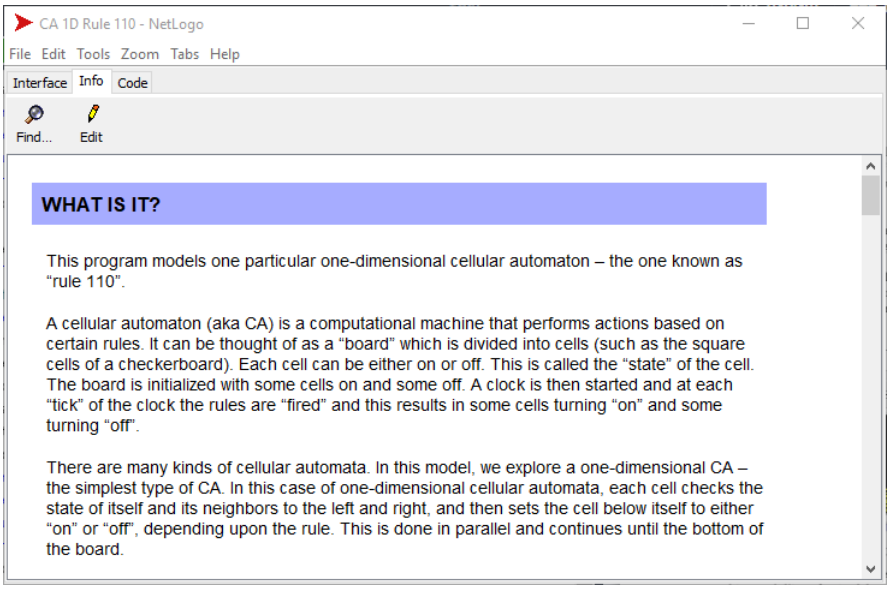

Figura 2.2: Interfaccia di NetLogo. Nella regione centrale è possibile leggere le informazioni sul modello, mentre tramite i pulsanti sotto la barra degli strumenti è possibile modificare il testo o ricercare parole chiave.

presente un insieme di controlli per l'editing testuale o la ricerca di informazioni. Se si clicca sul pulsante per modificare la documentazione del modello, è possibile osservare che questa viene scritta in Markdown e compilata.

2.1.3 Il codice

Il codice di un modello è scritto in un dialetto del linguaggio Logo, un linguaggio creato a scopi didattici, privo di uno standard vero e proprio. Questo linguaggio viene solitamente arricchito di primitive e funzionalità specifiche per la piattaforma in cui viene utilizzato, per poi essere compilato o interpretato. Il codice ha lo scopo di definire proprietà, regole e altri aspetti del modello, come alcuni dettagli di visualizzazione. Molte primitive definite da NetLogo possono essere eseguite solo in determinati contesti, ad esempio solo da un tipo di agente. In NetLogo gli agenti possono essere di quattro tipologie:

- *Turtles*, ossia tartarughe, che sono in grado di muoversi per il *mondo*. Il mondo è bidimensionale e suddiviso in una griglia di *patches*.
- *Patches*, ossia sezioni quadrate di mondo sulle quali le tartarughe sono in grado di muoversi;
- *Links*, ossia i collegamenti, che sono agenti che connettono due tartarughe;
- *Observer*, ossia l'osservatore, privo di una posizione, in grado di interagire con ogni altro agente dandovi istruzioni.

Ogni agente possiede delle proprietà già definite, ma è possibile aggiungerne di nuove con una parola chiave (`turtles-own`, `patches-own` e `links-own`). Inoltre è possibile dichiarare variabili globali con la keyword `global`. La tipizzazione è debole e sono supportati nativamente numerosi tipi di dati, incluse le collezioni e gli agenti stessi. Tra le variabili possedute nativamente dalle tartarughe vi sono le coordinate della posizione, `x` e `y`, la direzione, o `heading`, e il colore, `color`. Le tartarughe possono far parte di una razza (`breed`), il che permette di differenziarle anche nel comportamento. Ad esempio è possibile creare delle tartarughe "lupo" che tentano di divorare tartarughe "pecora". Dal codice è possibile controllare quasi ogni aspetto della visualizzazione del modello, a partire dalle immagini usate per rappresentare ogni razza di tartaruga, fino al controllo sul movimento della telecamera e del frame rate. In NetLogo le procedure possono essere dei *commands*, ossia dei comandi, o dei *reporters*, ossia delle istruzioni che permettono di calcolare un valore. Ad esempio, una funzione che sposta tutte le tartarughe è un comando e viene dichiarata con la parola chiave `to`, mentre una funzione che calcola la direzione media di un insieme di tartarughe è un reporter, e viene dichiarata con `to-report`. Alcuni reporters sono resi disponibili da NetLogo, come ad esempio `turtles`, che restituisce l'insieme di tutte le tartarughe. Molte informazioni aggiuntive su NetLogo e sul linguaggio sono reperibili nel manuale online [8].

L'interfaccia di programmazione è estremamente semplificata. Comprende la possibilità di compilare il codice, cercare del testo, gestire l'indentazione e la lista di tutte le procedure definite [Figura 2.3].

La primitiva *ask* In NetLogo è possibile lanciare comandi a *turtles*, *patches* o *links* tramite l'istruzione "ask". Tutto il codice che deve essere eseguito da un agente diverso dall'osservatore va inserito nel contesto corrispondente. Ad esempio, per far eseguire un'istruzione a tutte le tartarughe, è possibile usare il comando `ask turtles [commands]`. Una caratteristica fondamentale della primitiva ask è quella di eseguire i comandi specificati sull'insieme di

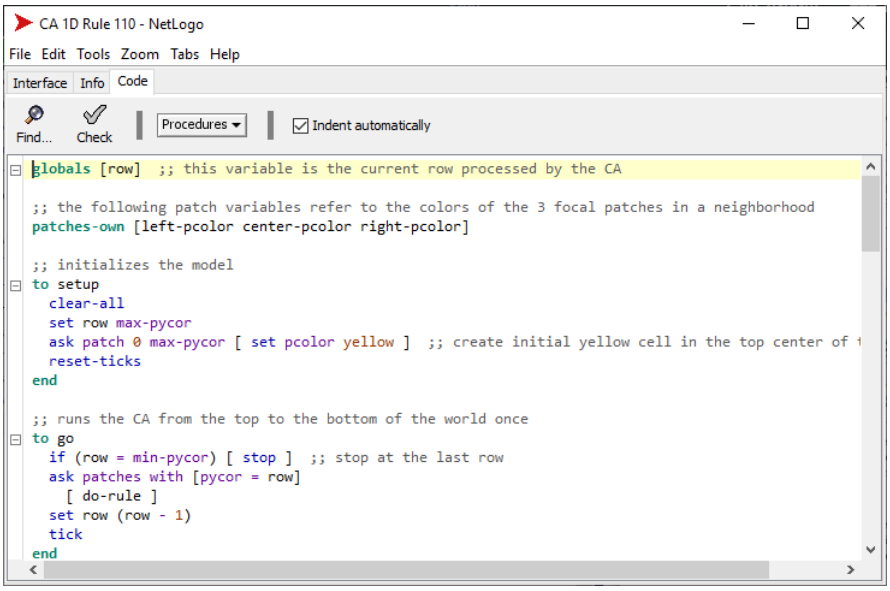

Figura 2.3: Interfaccia di programmazione in NetLogo.

agenti in modo sequenziale. Questo significa che finché un agente non ha completato l'esecuzione dei propri comandi, il successivo rimane in attesa del proprio turno. Inoltre, essendo il comando ask lanciato su un insieme e non su una lista, l'ordine di esecuzione dei comandi rispetto agli agenti non è costante. Secondo la documentazione infatti tale ordine è generato casualmente, per evitare che l'utente sfrutti un possibile ordine implicito o non documentato per far funzionare il programma correttamente. In un modello basato su agenti un'iterazione sequenziale su ogni agente potrebbe essere un'operazione troppo costosa, in termini di tempo. Una delle classiche soluzioni che si adotta in questi casi è quella di sfruttare il parallelismo, ossia la possibilità di eseguire più istruzioni contemporaneamente. Sebbene esistano delle primitive come `ask-concurrent` che permettono di simulare la concorrenza, che è ben lontana a livello pratico dal parallelismo, NetLogo non supporta la parallelizzazione di istruzioni [9]. Per modelli in cui la parallelizzazione di istruzioni è possibile, auspicabile o necessaria, scegliere NetLogo come ambiente di sviluppo potrebbe essere una cattiva idea, in quanto non verrebbe sfruttata questa utile qualità che non tutti i modelli possono vantare.

2.2 I modelli parallelizzabili

Non tutti i modelli computazionali necessitano di ottimizzazioni, e non tutti i modelli che necessitano ottimizzazioni hanno la possibilità di essere

realmente ottimizzati tramite tecniche di programmazione parallela, o più in generale di *High Performance Computing*. Non esiste un criterio univoco che stabilisce se un modello computazionale può essere implementato in modo da sfruttare un'ipotetica architettura parallela. Ogni modello può possedere determinate caratteristiche che ne impediscono la parallelizzazione, ma non è possibile sapere a priori quali siano e se è possibile trasformare il modello in uno equivalente che non possiede queste caratteristiche problematiche. D'altra parte però esistono modelli che si prestano meglio di altri ad essere ottimizzati tramite il parallelismo, e molti di questi modelli sono basati su agenti. Infatti, l'indipendenza delle parti di un modello è uno dei requisiti per poter strutturare un sistema che lavora in parallelo, ossia che esegue più istruzioni contemporaneamente senza creare errori o comportamenti imprevedibili. Nei sistemi ad agenti solitamente le unità lavorano in modo indipendente, talvolta interagendo tra loro o con l'ambiente.

2.2.1 Vantaggi di una versione parallela

La parallelizzazione offre vantaggi puramente pratici: permette alle simulazioni di eseguire in un tempo ragionevole istanze di simulazioni molto più complesse. Quanto più un modello o un'istanza sono complessi tanto più tempo ci vorrà per eseguirne una simulazione. Se si riuscisse invece ad ottimizzare il modello in modo da eseguire più calcoli contemporaneamente questo tempo potrebbe ridursi, seppur con dei limiti. Tra questi limiti è importante citare la **legge di Amdahl**.

$$S(p) = \frac{1}{\alpha + \frac{1-\alpha}{p}} \tag{2.1}$$

La legge di Amdahl impone un limite allo *speedup* $S(p)$, ossia il rapporto tra il tempo di esecuzione della versione seriale e quello della versione parallela con p unità di lavoro, $T_{seriale}/T_{parallelo}(p)$. Lo speedup è un'indice di miglioramento delle performance della versione parallela di un programma, con p unità indipendenti che lavorano, rispetto allo stesso programma in versione seriale oppure in versione parallela con $p = 1$. Nella formula α rappresenta la porzione di tempo impiegata ad eseguire codice non parallelizzato. In base alla formula, lo speedup massimo teorico dipende da α ed è $\frac{1}{\alpha}$.

Un altro limite importante da considerare sono le operazioni di I/O, in particolare quelle di output, molto importanti per una simulazione in quanto devono comunicare all'osservatore cosa sta succedendo ma che potrebbero rallentare l'esecuzione se non opportunamente gestite.

2.2.2 Parallelizzazione di un modello NetLogo

In NetLogo molti modelli non necessitano ottimizzazioni. Ad esempio se un modello mostra il funzionamento di un pistone, riuscire a mostrare migliaia di pistoni al lavoro piuttosto che uno singolo non ha un'utilità pratica o didattica. Solitamente i modelli più interessanti su larghe scale riguardano i sistemi complessi, presenti in diverse discipline all'interno della libreria NetLogo. Non tutti i modelli sono in grado di essere parallelizzati in modo consistente da poter offrire dei vantaggi reali. Ad esempio, se un modello ha numerose dipendenze tra i flussi di istruzioni che riguardano agenti diversi, potrebbe necessitare di un elevato numero di punti di sincronizzazione, che peggiorano le prestazioni della versione parallela. Se un modello fa uso di variabili globali a cui ogni agente accede in scrittura a turno, potrebbe essere difficile migliorare le prestazioni tramite la computazione parallela, in quanto esisterebbe comunque un punto di rallentamento, detto *bottleneck*, dovuto all'attesa del proprio turno per quella variabile. A volte però alcuni di questi casi sono convertibili in codice parallelizzabile: ad esempio, se le variabili globali vengono usate solamente in lettura, esse non rappresentano alcun ostacolo per un'eventuale versione parallela del codice. Oppure, se le variabili globali sono usate in un certo modo, ad esempio come dei contatori, è comunque possibile sfruttare alcuni pattern di programmazione per un'ottimizzazione in parallelo.

In generale, il processo di trasformazione del codice segue strade diverse per ogni modello e architettura parallela di riferimento. Nonostante ciò è comunque possibile fare delle considerazioni su alcune linee guida che è possibile seguire per quanto riguarda i modelli ad agenti. In particolare il tipo di parallelizzazione che solitamente si adotta in questi casi prevede l'esecuzione di più istruzioni uguali su porzioni di dati diversi. Questo tipo di suddivisione del lavoro in parallelo viene detto **data parallelism**, e si presta bene al caso degli agenti che per costruzione, di solito, adottano comportamenti simili tra loro ma che danno risultati diversi in base allo stato di partenza di ognuno. Le classi di computer che meglio si prestano ad un simile tipo di parallelizzazione sono **SIMD**, *Single Instruction Multiple Data*, che è in grado di eseguire contemporaneamente la stessa istruzione su dati diversi, e **MIMD**, che esegue istruzioni diverse su dati diversi.

Data l'impossibilità di utilizzare NetLogo o una sua estensione per eseguire istruzioni in parallelo sugli agenti di un modello è necessario cambiare ambiente di sviluppo.

2.3 FLAME

FLAME, che sta per *Flexible Large-scale Agent Modelling Environment*, è un ambiente di modellazione ad agenti in grado di generare un'applicazione eseguibile sulla maggior parte dei computer personali ma anche sui supercomputer dei centri di calcolo [10]. Le architetture di rifermento sono a memoria condivisa e distribuita, ossia quelle supportate da MPI, *Message Passing Interface*.

FLAME offre la possibilità di definire formalmente dei modelli ad agenti con determinate caratteristiche per poter generare automaticamente il programma di simulazione da eseguire sul proprio computer o in un centro di calcolo distribuito. Restringere la classe di modelli che FLAME supporta ha permesso la creazione di un generatore di codice per simulazioni ancora più ottimizzato. In particolare, sono state imposte restrizioni sugli agenti e sulle interazioni. Ad esempio, la comunicazione tra agenti avviene solamente tramite scambio di messaggi, e gli agenti sono immersi in uno spazio tridimensionale o bidimensionale a coordinate reali. In un modello di FLAME se un agente fosse in grado di comunicare solo entro un determinato raggio, l'ambiente in cui è immerso verrebbe partizionato in sezioni quadrate o cubiche all'interno delle quali ci sarebbero scambi di messaggi "riservati" evitando inutili comunicazioni con agenti troppo distanti. Questo ed altri esempi di ottimizzazioni sono stati inclusi in FLAME per poter essere usati in modo trasparente dall'utente che non dovrà così studiare come funziona la programmazione parallela. Un'altra importante considerazione da fare su FLAME riguarda la scelta di generare codice che adotta il paradigma di programmazione parallela a memoria distribuita tramite il protocollo MPI [11, pp. 5-7]. Nel caso di un centro di calcolo questo tipo di approccio risulta essere quello preferito, data la disponibilità di processori e memorie indipendenti che possono essere impiegati dal software simulativo. Per le altre macchine invece, che solitamente fanno uso di un singolo processore fisico con più core, è possibile che si crei un *overhead*, ossia un eccesso di dati o calcoli, dovuto alla simulazione di un ambiente a memoria distribuita tra processi che in realtà si trovano sulla stessa macchina. Nonostante FLAME riesca a creare una versione parallela della simulazione che sfrutta tutti i core di una CPU è comunque possibile e sensato sfruttare un'altra tecnologia per ottenere risultati decisamente migliori rimanendo nell'ambito di una computazione su singola macchina, la GPGPU (*General Purpose computing on Graphic Processing Units*). Ciò è possibile grazie ad un'estensione di FLAME chiamata FLAME GPU, di cui si parlerà nella sezione 3.

2.3.1 Definire un modello in FLAME

La definizione di un modello in FLAME è completamente diversa da quella in NetLogo. FLAME offre delle specifiche di formattazione per un file di testo in XML (*Extensible Markup Language*) tramite un file "schema". Alcuni editor di testo avanzati permettono la validazione in tempo reale del file XML rispetto allo schema. Esistono tag XML per definire un modello tramite più file, per definire costanti globali che rappresentano i parametri di "default" della simulazione, per stabilire quali sono i file da cui reperire i corpi delle funzioni e molto altro. Un approfondimento riguardante il file XML di definizione è disponibile nel manuale online [12].

Gli agenti Gli agenti in FLAME sono costituiti da un nome, una descrizione, una memoria, un insieme di funzioni e uno di stati. Questo tipo di agente viene detto *X-agent*, ed è basato sul modello computazionale delle *X-machine*, un'estensione alla definizione classica di macchina a stati finiti [13]. La memoria di un agente è un insieme di variabili con nome, tipo di dato e descrizione che ogni agente mantiene al suo interno. I tipi di dato possono essere quelli del C (numeri interi o a virgola mobile, vettori, strutture) o possono essere tipi di dato definiti nell'XML stesso. Le funzioni di un agente possiedono:

- Un nome
- Una descrizione
- Lo stato corrente assunto dall'agente prima della chiamata
- Lo stato in cui si troverà l'agente dopo la chiamata
- La condizione di transizione della funzione
- L'insieme degli input della funzione
- L'insieme degli output

I dati in input e output di una funzione sono costituiti esclusivamente da messaggi. Tutti i possibili messaggi che fanno parte del modello sono definiti in una lista all'interno del file XML. Ogni messaggio ha un nome, una descrizione e un insieme di variabili. Le funzioni hanno dei tag aggiuntivi che specificano come ordinare i messaggi in ingresso. L'insieme degli stati è definito attraverso quello delle funzioni, che devono creare un grafo delle transizioni sensato, altrimenti si possono verificare degli errori.

Le funzioni Per ogni agente che viene definito devono essere implementate, in un corrispondente file, tutte le funzioni dichiarate per quell'agente. Il collegamento tra una definizione di agente e un file è dato da un apposito tag nell'XML. Il file deve avere estensione ".c" ed essere scritto in linguaggio C. Ognuno di questi file deve implementare tutte le funzioni definite nell'XML e deve includere i file header di FLAME e dell'agente che implementa. Questi file header saranno generati automaticamente in una fase di compilazione, pertanto un normale ambiente di sviluppo per C potrebbe generare errori da ignorare, in quanto, dopo un'operazione di pulizia dei file temporanei, questi header non esisterebbero più. Il framework FLAME mette a disposizione delle API, *Application Programming Interface*, per l'accesso alla memoria dell'agente, alle costanti di simulazione, al controllo sull'allocamento o deallocamento di memoria per gli agenti e per i tipi di dato definiti nell'XML, nonché per l'invio e la ricezione dei messaggi.

Compilazione di un modello La compilazione del modello avviene tramite un programma chiamato *xparser*. Il programma prende in input, a riga di comando, il percorso del file XML di definizione del modello, e alcuni flag:

- `-s` abilita la modalità seriale, che non fa uso di MPI.
- `-p` abilita la generazione di codice parallelizzato con MPI. Questo flag non è compatibile con il precedente.
- `-f` abilita la modalità produzione, generando un eseguibile privo di funzionalità di debug.

I file generati da xparser sono divisi in gruppi:

- File di documentazione
  - Grafici sulle macchine a stati finiti, i cambi di stato, le funzioni e i punti di sincronizzazione
  - File di documentazione in LaTeX
- Sorgenti per la simulazione
  - Un Makefile per compilare i sorgenti con `make`
  - I sorgenti .c per la simulazione
  - I file header per la simulazione
  - Un Doxyfile per generare documentazione con `doxygen` [14]
- Sorgenti specifici per la simulazione del modello compilato

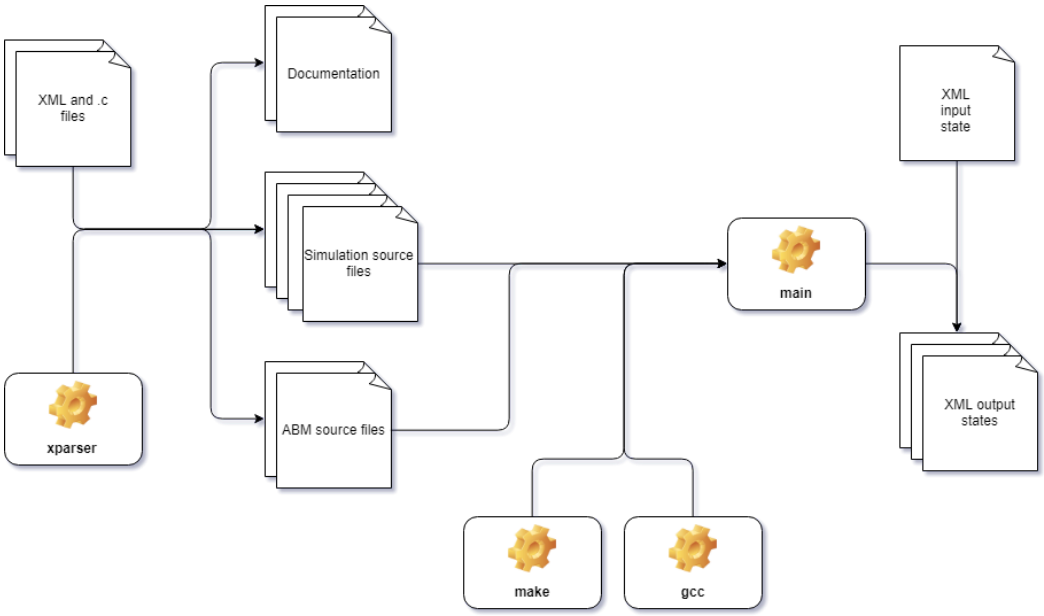

Figura 2.4: Schema della compilazione ed esecuzione di un progetto in FLAME e per ottenere gli XML degli stati in output.

– Un header file per ogni tipo di agente

Alla fine del processo di generazione, bisogna compilare i sorgenti usando make e un compilatore C come ad esempio GCC [15]. L'output del processo è un eseguibile chiamato main, che prende in input il numero di iterazioni, il file contenente lo stato iniziale in XML, il numero dei nodi MPI (approfondito nella sezione 3.7) e alcuni flag per controllare la frequenza di output della simulazione. Questo output avviene tramite XML compatibili con il formato dello stato iniziale, che a sua volta ha delle specifiche molto simili a quello di definizione del modello (figura 2.4).

2.3.2 Conclusioni su FLAME

Senza entrare in ulteriori dettagli in merito ai formalismi e alle tecniche pratiche di sviluppo con FLAME è già possibile notare le importanti differenze che lo separano da NetLogo. Di sicuro per un utente non informatico imparare il linguaggio C, le specifiche XML e la compilazione è davvero un impegno pesante se pensiamo che lo scopo di tutto ciò è semplicemente quello di simulare un modello ad agenti. L'unico vero vantaggio di FLAME sono le prestazioni della versione parallela con MPI rispetto a quella seriale di NetLogo, oltre al piccolo miglioramento dovuto all'uso del C, che compila in codice

macchina, piuttosto che Java che compila in un linguaggio intermedio interpretato a tempo di esecuzione. Questo vantaggio è però indebolito dal fatto che normalmente uno sviluppatore non ha le risorse di un centro di calcolo distribuito, pertanto FLAME viene messo in secondo piano dalla sua estensione FLAME GPU, che lavora su GPU Nvidia [16], possedute ormai da numerosi dispositivi. Le prestazioni delle simulazioni di FLAME su un singolo computer non rappresentano quindi un miglioramento sostanziale che lo renda una valida alternativa a NetLogo.

Nel prossimo capitolo sarà discussa approfonditamente l'estensione di FLAME per la GPU e sarà fatto un confronto tra i vantaggi dell'implementazione con il protocollo MPI o con il linguaggio CUDA per GPU Nvidia (sezione 3.7).

Capitolo 3

FLAME GPU

FLAME GPU è un'estensione del framework FLAME che fa uso della GPU [17]. In questo capitolo saranno descritti, in particolare, la definizione e il processo di compilazione di un modello.

3.1 Panoramica

I dispositivi compatibili con FLAME GPU sono GPU Nvidia che supportano CUDA [18]. Per poter lavorare con FLAME GPU è necessario un ambiente di sviluppo opportuno. Se si lavora da Windows è possibile usare Visual Studio per scrivere il codice e gli XML, ma per la compilazione è necessaria qualche operazione di configurazione. La versione di CUDA predefinita è modificabile tramite uno script presente nella cartella *tools* del repository ufficiale [19]. Il repository ufficiale di FLAME GPU è un ottimo punto di partenza per creare nuovi progetti grazie a degli strumenti, scritti in Python, che generano automaticamente file sorgenti e di Visual Studio per un nuovo progetto. Per eseguire questi utili script è necessario avere installato Python [20]. Dopo aver creato i file di un nuovo progetto è possibile aprire la soluzione con estensione `.sln` di Visual Studio e compilare il codice. In alternativa è possibile compilare anche tramite il Makefile generato insieme agli altri sorgenti sempre dallo script Python. Altre informazioni utili sono reperibili nella sezione "4. FLAME GPU Simulation and Visualisation" della documentazione di FLAME GPU [21]. Tutti i progetti creati dal repository di FLAME GPU si trovano nella cartella *examples*.

Come specificato nella documentazione, FLAME GPU non è un simulatore ma è un ambiente di sviluppo basato sui *template* che mappa una definizione di modello in codice per la simulazione. I template sono dei file che vengono processati e trasformati in sorgenti sulla base delle specifiche di modello scritte all'interno del file di definizione. Il file di definizione è in formato *XMML*,

che sta per X-Machine Markup Language, ossia l'XML unito allo schema per definire le X-Machine. Tramite le X-Machine si possono definire degli agenti che comunicano tra loro, hanno una memoria, degli stati e delle transizioni di stato. Questi elementi permettono di creare dei modelli che dopo un certo numero di iterazioni mostrano un comportamento globale interessante.

3.2 Definizione del modello

In FLAME GPU i modelli vengono definiti tramite dei file XML, come vale per FLAME. Lo schema di FLAME GPU è un'estensione dello schema di FLAME, ossia mantiene una retrocompatibilità con l'originale ma aggiunge varie informazioni utili alla generazione di codice per GPU. Lo scheletro principale del file XML è il seguente:

```
<gpu:xmodel
    xmlns:gpu="http://www.dcs.shef.ac.uk/~paul/XMMLGPU"
    xmlns="http://www.dcs.shef.ac.uk/~paul/XMML">
    <name>Model Name</name>    <!-- optional -->
    <gpu:environment>...</gpu:environment>
    <xagents>...</xagents>
    <messages>...</messages>
    <layers>...</layers>
</gpu:xmodel>
```

I file a cui gli URL fanno riferimento sono gli schemi di validazione dell'XML. L'*environment* è un oggetto XML:

```
<gpu:environment>
    <gpu:constants>...</gpu:constants>        <!-- optional -->
    <gpu:functionFiles>...</gpu:functionFiles><!-- required -->
    <gpu:initFunctions>...</gpu:initFunctions><!-- optional -->
    <gpu:stepFunctions>...</gpu:stepFunctions><!-- optional -->
    <gpu:exitFunctions>...</gpu:exitFunctions><!-- optional -->
    <gpu:graphs>...</gpu:graphs>              <!-- optional -->
</gpu:environment>
```

All'interno delle costanti, il cui tag è `<gpu:constants>`, è possibile definire una o più variabili che rappresenteranno i parametri della simulazione e i loro valori predefiniti. Questi valori potranno essere sovrascritti dal file XML contentente lo stato iniziale o dal codice della simulazione.

```
<gpu:variable>
```

```
    <type>int</type>
    <name>const_variable</name>
    <description>none</description>
    <defaultValue>1</defaultValue>
</gpu:variable>
```

I file contenenti le funzioni sono sorgenti con estensione `.c` contenenti le implementazioni dei metodi di ogni agente.

```
<gpu:functionFiles>
    <file>functions.c</file>
</gpu:functionFiles>
```

Le funzioni di *init*, *step* ed *exit* sono detti *simulation hooks*, ossia funzioni chiamate in determinate occasioni da FLAME GPU che permettono di eseguire del codice sulla CPU. In particolare la funzione init viene chiamata durante l'inizializzazione, la funzione step alla fine di ogni iterazione della simulazione, la funzione exit all'uscita del programma. Queste funzioni permettono di personalizzare il comportamento del programma a proprio piacimento e saranno implementate in uno dei sorgenti specificati nel tag `<gpu:functionFiles>`.

```
<gpu:initFunctions> <!-- same for step, or exit -->
    <gpu:initFunction>
        <gpu:name>initConstants</gpu:name>
    </gpu:initFunction>
</gpu:initFunctions>
```

Il tag `<gpu:graphs>` si usa per definire un grafo che stabilisce dei canali di comunicazione tra agenti attraverso i quali vengono trasmessi i messaggi. Può essere utile definirlo per alcuni modelli, ma non è una funzionalità che sarà approfondita in questa tesi.

3.3 Definizione degli agenti

Gli agenti si definiscono tramite il tag `<xagents>`, che contiene una lista di `<gpu:xagent>`:

```
<xagents>
    <gpu:xagent>
    <name>AgentName</name>
        <description>optional description</description>
        <memory>...</memory>
        <functions>...</functions>
```

```
        <states>...</states>
        <gpu:type>continuous</gpu:type>
        <gpu:bufferSize>1024</gpu:bufferSize>
    </gpu:xagent>
    <gpu:xagent>
        <!-- ... -->
    </gpu:xagent>
</xagents>
```

La memoria di un agente è una lista di variabili che ogni agente mantiene memorizzate privatamente. La sintassi è la stessa delle costanti, e il valore di default, se non specificato, è 0. I tipi disponibili dipendono dalla *CUDA capability*, ossia dalle specifiche che la GPU supporta. Ad esempio, per poter usare il `double` come tipo di dato è necessaria una compute capability di 1.3 o superiore. Sono disponibili anche i tipi di dato vettoriali.

Gli stati di un agente sono una lista di oggetti XML:

```
<states>
    <gpu:state>
        <name>state1</name>
    </gpu:state>
    <gpu:state>
        <name>state2</name>
    </gpu:state>
    <initialState>state1</initialState>
</states>
```

Questa specifica non esiste in FLAME, mentre è presente in FLAME GPU. Definire uno stato iniziale è obbligatorio, mentre non è possibile definirne uno di uscita. I nomi degli stati devono essere univoci.

Le funzioni degli agenti si definiscono secondo il seguente schema:

```
<functions>
    <gpu:function>
        <name>func_name</name>
        <description>function description</description>
        <currentState>state1</currentState>
        <nextState>state2</nextState>
        <inputs>...</inputs>            <!-- optional -->
        <outputs>...</outputs>          <!-- optional -->
        <xagentOutputs></xagentOutputs> <!-- optional -->
        <gpu:globalCondition>
            ...
```

```
        </gpu:globalCondition>          <!-- optional -->
        <condition>...</condition>      <!-- optional -->
        <gpu:reallocate>
           true
        </gpu:reallocate>               <!-- optional -->
        <gpu:RNG>true</gpu:RNG>         <!-- optional -->
    </gpu:function>
</functions>
```

Per ogni funzione è necessario definire un nome univoco, una transizione tra stati, che possono essere il medesimo, e una descrizione. Tramite una sintassi simile a quella usata da FLAME è possibile definire delle condizioni di ingresso che impediscono l'esecuzione della funzione se non risultano vere. Gli input e gli output di una funzione sono dei messaggi, argomento della sezione 3.4.

```
<inputs>
    <gpu:input>
        <messageName>message_name</messageName>
    </gpu:input>
</inputs>
<outputs>
    <gpu:output>
        <messageName>message_name</messageName>
        <gpu:type>single_message</gpu:type>
    </gpu:output>
</outputs>
```

Esiste la possibilità che una funzione produca nuovi agenti tramite il tag `<xagentOutputs>`, che non sarà approfondito. Il tag `<gpu:reallocate>` abilita la possibilità che un agente muoia al termine della funzione, mentre il tag `<gpu:RNG>` predispone la funzione per generare numeri pseudocasuali diversi per ogni agente. Il tag `<gpu:type>` definisce il tipo di agente, che può essere:

- `discrete`, ossia discreto, come un automa cellulare, che impone all'agente di muoversi in uno spazio discreto bidimensionale;
- `continuous`, ossia continuo, che permette all'agente di muoversi in uno spazio a tre dimensioni reali, indicato anche per agenti astratti che non hanno una posizione nello spazio.

`<gpu:bufferSize>` definisce invece la dimensione massima della popolazione di agenti di questo tipo. Solitamente viene arrotondata alla potenza di 2 più vicina e va impostata in base alla capacità di memoria della GPU. Non sono forniti molti dettagli su quale sia un buon metodo per trovare il valore giusto, ma è sempre possibile andare per tentativi.

3.4 Definizione dei messaggi e dei *layers*

Tramite il tag `<messages>` è possibile definire tutti i tipi di messaggi che gli agenti potranno scambiarsi:

```
<messages>
    <gpu:message>
        <name>message_name</name>
        <description>optional message description</description>
        <variables>...</variables>

        <!-- replace with a partitioning type -->
        ...<partitioningType/>...

        <gpu:bufferSize>1024</gpu:bufferSize>
    </gpu:message>
    <gpu:message>...</gpu:message>
</messages>
```

Oltre a nome, descrizione e variabili contenuti nel messaggio, sono presenti due importanti attributi: il partizionamento del messaggio e le dimensioni del buffer. Il partizionamento dei messaggi definisce come FLAME GPU deve ottimizzare lo scambio di messaggi tra agenti per massimizzare le prestazioni. Ad esempio, il partizionamento `<gpu:partitioningNone/>` consiste in una comunicazione da tutti gli agenti a tutti gli agenti, con un costo complessivo di $O(n^2)$, con n uguale al numero di agenti comunicanti. Un partizionamento `<gpu:partitioningDiscrete>` permette ad agenti che fanno parte di un automa cellulare di comunicare con altri agenti vicini entro un determinato raggio discreto. Al contrario, il partizionamento `<gpu:partitioningSpatial>` suddivide lo spazio a tre dimensioni reali in sezioni, e garantisce la ricezione di messaggi provenienti da agenti che si trovano entro un determinato raggio, secondo le coordinate x, y e z di ogni agente.

```
<gpu:partitioningSpatial>
    <gpu:radius>1</gpu:radius>
    <gpu:xmin>0</gpu:xmin>
    <gpu:xmax>10</gpu:xmax>
    <gpu:ymin>0</gpu:ymin>
    <gpu:ymax>10</gpu:ymax>
    <gpu:zmin>0</gpu:zmin>
    <gpu:zmax>10</gpu:zmax>
</gpu:partitioningSpatial>
```

Questo tipo di partizionamento sarà usato nei modelli di Flocking per migliorare le prestazioni, impostando il raggio di comunicazione pari a quello visivo di ogni volatile. I valori di minimo e massimo per ogni asse definiscono i limiti all'interno dei quali possono esistere dei messaggi. Messaggi nati al di fuori dei limiti possono generare comportamenti non definiti. Lungo le coordinate x e y, ma non lungo z, lo spazio dei messaggi viene considerato toroidale. Questa caratteristica è fondamentale per permettere l'implementazione di un modello NetLogo in FLAME GPU, poiché in NetLogo lo spazio bidimensionale è toroidale. Esiste anche il partizionamento basato sul grafo di comunicazione, ma come già detto non sarà trattato in questa tesi.

I *layers* rappresentano il controllo del flusso di esecuzione delle funzioni degli agenti. Ogni layer può contenere una o più funzioni da eseguire. L'ordine di esecuzione delle funzioni di uno stesso layer riguarda un singolo agente. Infatti la sincronizzazione tra agenti diversi avviene solo in corrispondenza della fine di un layer e dell'inizio del sucessivo.

```
<layers>
    <layer>
        <gpu:layerFunction>
            <name>function1</name>
        </gpu:layerFunction>
        <gpu:layerFunction>
            <name>function2</name>
        </gpu:layerFunction>
    </layer>
    <layer>
        <gpu:layerFunction>
            <name>function3</name>
        </gpu:layerFunction>
    </layer>
</layers>
```

I punti di sincronizzazione sono fondamentali per garantire la presenza di messaggi in input a funzioni che li necessitano.

3.5 Implementazione delle funzioni

Le funzioni di ogni agente vanno implementate nel corrispondente file `.c` in C, C++ o CUDA C. Il tipo di ritorno di ogni funzione da implementare è `int`, ossia un numero che viene valutato per stabilire se l'agente è morto. Un valore pari a 0 significa vivo, ogni altro significa morto. Il nome della funzione è lo

stesso dichiarato nell'XML, mentre gli argomenti in ingresso variano in base alle caratteristiche della funzione e al nome dell'agente. Uno di questi argomenti è un puntatore alla struttura che contiene le variabili di un agente per poterle modificare. Davanti alla funzione va messo un prefisso che corrisponde ad una macro definita nel file header di FLAME GPU, `__FLAME_GPU_FUNC__`. Questa macro permette l'esecuzione della funzione sul *device* CUDA, ossia sulla GPU. Il file header, come per FLAME, viene generato dinamicamente durante la compilazione.

Tramite delle apposite API è possibile inviare messaggi in output e iterare tra i messaggi in input di una funzione.

```
/* Esempio di invio: */
add_location_message(location_messages, id, x, y, z);

/* Esempio di iterazione: */
message = get_first_location_message(location_messages);
 while(message)
 {
     /* Qui si usa il contenuto del messaggio */
     message =
        get_next_location_message(message, location_messages);
 }
```

Per i messaggi con partizionamento continuo è necessario specificare la posizione dell'agente come parametro per la funzione che inizializza l'iterazione, nelle coordinate reali x, y e z; mentre per il partizionamento discreto vanno specificate le coordinate intere x e y. Esistono delle semplici API per generare nuovi agenti e per ottenere numeri casuali diversi per ogni agente; per altre informazioni in merito è possibile consultare il manuale [21].

Le funzioni di init, step ed exit non vanno dichiarate con la macro delle funzioni per la GPU in quanto vengono eseguite esclusivamente sulla CPU. Al loro interno è possibile caricare i valori delle costanti di simulazione, riscriverne il valore, creare nuovi agenti, effettuare conteggi o calcoli di minimo, massimo e media sulle variabili degli agenti e interrompere la simulazione.

3.6 Compilazione di un modello su FLAME GPU

La compilazione di un modello comincia con l'esecuzione del programma `XSLTProcessor.exe` situato nella cartella *tools* del repository ufficiale di FLAMEGPU [19]. In alternativa è possibile usare un altro programma compatibile

con XSLT, che sta per *eXtensible Stylesheet Language Transformations*, un linguaggio funzionale basato su XML adatto alla traduzione di file XML in altri linguaggi. XSLTProcessor va eseguito passando in input, a riga di comando, il percorso per il file di definizione del modello in XML, per il file *functions.xslt* e per i file con le implementazioni delle funzioni degli agenti. In seguito i file con estensione `.cu` andranno compilati come sorgenti CUDA C da `nvcc`. Successivamente saranno compilati tutti i sorgenti C e C++ tramite il compilatore MSVC (*Microsoft Visual C++*) e infine tutti i file oggetto saranno linkati insieme per produrre l'eseguibile finale (figura 3.1). L'intero processo di build può essere automatizzato grazie a Visual Studio o ad un Makefile.

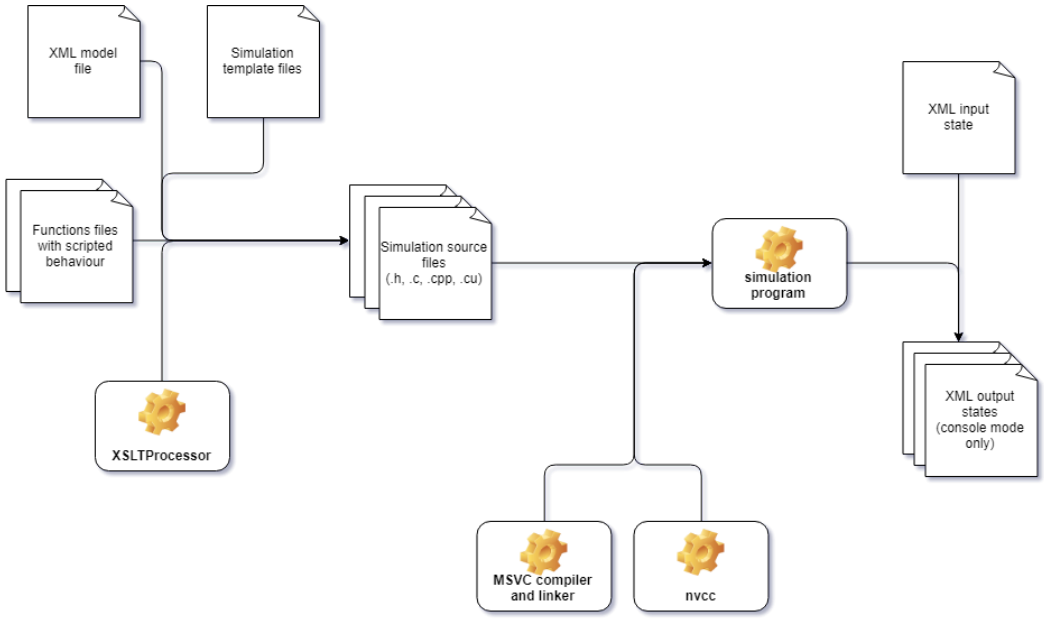

Figura 3.1: Schema della compilazione di un progetto con FLAME GPU, su Windows.

Esistono due modalità di compilazione. La modalità *Console* crea un eseguibile che prende in input lo stato iniziale e il numero di iterazioni e stampa, in forma di file XML, le iterazioni della simulazione. Dopo aver raggiunto il limite di iterazioni specificato il programma termina. La modalità *Visualizzazione* invece crea un eseguibile che prende in input lo stato iniziale ed esegue la simulazione fino alla sua interruzione, mostrando in output a schermo una visuale tridimensionale dello spazio e degli agenti in tempo reale. Per ogni modalità esistono due tipi di compilazione: *Debug* e *Release*. In modalità di debug viene aggiunto del codice di controllo per la parte in CUDA e per il debug della parte seriale, inoltre vengono disabilitate le ottimizzazioni da parte dei compilatori. In modalità di rilascio vengono riabilitate tutte le ottimizzazioni.

3.7 MPI e CUDA a confronto

Le tecnologie per la parallelizzazione usate da FLAME e FLAME GPU sono, rispettivamente, il protocollo MPI e l'architettura CUDA. MPI è un protocollo standard di comunicazione a scambio di messaggi tra processi, contrapposto al paradigma della parallelizzazione a memoria condivisa [22]. Tramite MPI è possibile effettuare tre gruppi di operazioni:

- **Comunicazione**, sia da punto a punto sia collettiva;
- **Sincronizzazione**, senza necessità di mutua esclusione sulle variabili poichè la memoria non è condivisa;
- **Interrogazioni**, che permettono di ottenere dati aggregati come ad esempio il numero di processi oppure l'ID del processo chiamante.

L'architettura logica adottata dal protocollo di MPI è quella di una rete di unità di lavoro che comunicano tramite scambio di messaggi. Ognuna di queste unità è detta un **nodo** MPI. Ogni nodo è solitamente un processo allocato ed eseguito su una macchina dedicata, o più in generale su un proprio hardware, per poter sfruttare la massima capacità computazionale disponibile. Su singola macchina è comunque possibile simulare una rete di processori indipendenti sfruttando, se disponibili, più core di un singolo processore. Un programma MPI dovrebbe cercare di sfruttare al massimo delle sue capacità tutto l'hardware disponibile: nel caso di un centro di calcolo dotato di più processori indipendenti dovrebbe fare uso di tutti i core disponibili per ognuno ed eventualmente di altri dispositivi hardware presenti nelle macchine. MPI e CUDA infatti possono essere integrati per distribuire un processo a più unità di calcolo che possiedono GPU Nvidia, se il programma ne dovesse trarre vantaggio.

CUDA è una piattaforma di calcolo parallelo distribuita insieme ad un modello di programmazione per GPU Nvidia [18]. CUDA permette di utilizzare le GPU per eseguire calcoli *general purpose*, sfruttando le numerose ALU di questi dispositivi ottimizzati per le operazioni grafiche. Sebbene non tutti i programmi traggano gli stessi vantaggi da una forte parallelizzazione come quella possibile sulle GPU, per molti algoritmi parallelizzabili potrebbe essere un vantaggio farne uso. Sicuramente per chi possiede una GPU di questo tipo e un singolo processore l'opzione del calcolo parallelizzato con CUDA sarà più efficiente.

Capitolo 4

Implementazione del modello Flocking con FLAME GPU

Il primo modello implementato con FLAME GPU è Flocking della libreria NetLogo. Il progetto è tracciato tramite Git e reso disponibile su GitHub [23] come fork del repository ufficiale di FLAME GPU, su un nuovo branch chiamato *flocking*. Il progetto è stato creato a partire dallo script in Python che permette di clonare uno degli esempi di FLAME GPU presenti nella crtella *examples*. In particolare è stato clonato *EmptyExample*, un esempio di modello vuoto ma compilabile.

4.1 Definizione del modello

4.1.1 Le costanti di simulazione

I parametri della simulazione sono stati definiti sulla base dei controlli disponibili nell'interfaccia del modello Flocking su NetLogo. In aggiunta a questi è stato implementato un campo visivo limitato anche in angolo, che è diventato un nuovo parametro della simulazione, `FOV`, ossia *Field of View* (Tabella 4.1).

4.1.2 Definizione degli agenti

Nel modello Flocking è presente un solo tipo di agente, dotato di una posizione nello spazio e una direzione. Lo spazio è bidimensionale come quello di NetLogo, pertanto la direzione può essere espressa semplicemente con un numero reale che indica l'angolo di rotazione in radianti rispetto ad una direzione fissa, in questo caso pari a $(1, 0)$. Gli assi lungo i quali gli agenti si spostano sono x ed y, in modo da sfruttare il partizionamento toroidale dei messaggi

nome variabile	**tipo**	**info**
population	int	Dimensione della popolazione
vision	float	Distanza massima di vista per un agente
speed	float	Distanza percorsa ad ogni iterazione
minimum_separation	float	Distanza oltre la quale due agenti sono "separati"
max_align_turn	float	Massima rotazione per l'allineamento, in gradi
max_cohere_turn	float	Massima rotazione per la coesione, in gradi
max_separate_turn	float	Massima rotazione per la separazione, in gradi
bounds	float	Dimensione del lato del quadrato che costituisce il mondo. Dovrebbe corrispondere a quello specificato nei limiti del partizionamento dei messaggi.
FOV	float	Angolo, in gradi, corrispondente all'ampiezza della vista frontale di un agente.

Tabella 4.1: I parametri della simulazione, il loro tipo e significato.

poiché in NetLogo lo spazio è toroidale. In aggiunta a ciò ogni agente è stato dotato di una variabile che ne rappresenta il colore, ai fini della visualizzazione (Listato 4.1).

```
<memory>
  <gpu:variable>
    <type>int</type>
    <name>colour</name>
  </gpu:variable>
  <gpu:variable>
    <type>float</type>
    <name>x</name>
  </gpu:variable>
  <gpu:variable>
    <type>float</type>
    <name>y</name>
  </gpu:variable>
  <gpu:variable>
    <type>float</type>
    <name>heading</name>
  </gpu:variable>
</memory>
```

Listato 4.1: Sezione del file di definizione del modello riguardante la memoria degli agenti.

Le funzioni di un agente sono state ridotte a due: `move` e `flock`. La funzione move controlla la velocità, uno dei parametri della simulazione, la direzione e la posizione corrente, poi calcola la nuova posizione dell'agente, sovrascriven-

dola. Successivamente invia un messaggio contenente i dati sulla direzione e sulla posizione dell'agente in modo da comunicarla ai vicini. La funzione flock implementa il comportamento degli agenti, prendendo in ingresso i messaggi relativi allo stato degli agenti nelle vicinanze (Listato 4.2).

```
<functions>
  <gpu:function>
    <name>move</name>
    <currentState>default</currentState>
    <nextState>default</nextState>
    <outputs>
      <gpu:output>
        <messageName>position</messageName>
        <gpu:type>single_message</gpu:type>
      </gpu:output>
    </outputs>
    <gpu:reallocate>false</gpu:reallocate>
    <gpu:RNG>false</gpu:RNG>
  </gpu:function>
  <gpu:function>
    <name>flock</name>
    <currentState>default</currentState>
    <nextState>default</nextState>
    <inputs>
      <gpu:input>
        <messageName>position</messageName>
      </gpu:input>
    </inputs>
    <gpu:reallocate>false</gpu:reallocate>
    <gpu:RNG>false</gpu:RNG>
  </gpu:function>
</functions>
```

Listato 4.2: Sezione del file di definizione del modello riguardante le funzioni degli agenti.

Le funzioni non prevedono comportamenti casuali e non prevedono la morte degli agenti.

Ogni agente possiede un solo possibile stato, chiamato *default* per semplicità, ed è di tipo *continuous*, in quanto si muove nello spazio a coordinate reali. L'uso del tipo `float` permette al modello di essere compatibile anche con le schede grafiche che non supportano il `double`.

I messaggi scambiati sono la posizione e la direzione (Listato 4.3).

```
<messages>
  <gpu:message>
    <name>position</name>
    <variables>
      <gpu:variable>
        <type>float</type>
        <name>x</name>
      </gpu:variable>
      <gpu:variable>
        <type>float</type>
        <name>y</name>
      </gpu:variable>
      <gpu:variable>
        <type>float</type>
        <name>z</name>
      </gpu:variable>
      <gpu:variable>
        <type>float</type>
        <name>dx</name>
      </gpu:variable>
      <gpu:variable>
        <type>float</type>
        <name>dy</name>
      </gpu:variable>
      <gpu:variable>
        <type>float</type>
        <name>heading</name>
      </gpu:variable>
    </variables>
    <gpu:partitioningSpatial>
      <gpu:radius>5</gpu:radius>
      <gpu:xmin>0</gpu:xmin>
      <gpu:xmax>70</gpu:xmax>
      <gpu:ymin>0</gpu:ymin>
      <gpu:ymax>70</gpu:ymax>
      <gpu:zmin>0</gpu:zmin>
      <gpu:zmax>5</gpu:zmax>
    </gpu:partitioningSpatial>
    <gpu:bufferSize>32768</gpu:bufferSize>
  </gpu:message>
</messages>
```

Listato 4.3: Sezione del file di definizione del modello riguardante i messaggi.

Il partizionamento continuo basato sulla distanza di vista è definito tramite il tag del raggio e i tag dei limiti dello spazio del messaggio. In questo caso, il numero delle istanze di un dato messaggio che possono essere presenti contemporaneamente in un dato momento è sempre uguale alla popolazione di agenti, per cui anche i buffer dei messaggi dovrebbero esserlo.

I layers definiti sono due, in quanto è necessario un singolo punto di sincronizzazione dopo la funzione che invia la posizione e prima della funzione che legge le posizioni dei vicini.

4.2 Implementazione delle funzioni

4.2.1 Le primitive di NetLogo

NetLogo, come linguaggio, offre alcune primitive che semplificano il codice e risparmiando all'utente l'implementazione di funzioni usate in modo ricorrente tra modelli differenti. Ad esempio, esistono primitive per "muovere in avanti" le tartarughe, oppure una primitiva che stabilisce se una tartaruga si trova nel cono visivo di un'altra. In FLAME GPU, invece, non esiste nessuna primitiva del genere. Per questo motivo è stato necessario implementare tutte quelle di cui Flocking faceva uso. Esempi di funzioni implementate:

- `subtract-headings`, che calcola la differenza tra due angoli nell'intervallo [-180, 180];
- il reporter `distance` che calcola la distanza tra due agenti;
- le primitive per il movimento, tra cui `fd`, che fa avanzare l'agente frontalmente di una certa distanza.

Il vantaggio di queste primitive in NetLogo è che rendono completamente trasparente all'utente il fatto che l'ambiente in cui gli agenti sono immersi sia toroidale. In FLAME GPU invece non esiste questa caratteristica, ad eccezione dello spazio di partizione dei messaggi.

4.2.2 La funzione *setup*

La funzione setup viene eseguita dalla CPU prima dell'inizio della simulazione, in quanto è stata registrata come funzione di init nel file XML del modello. La prima operazione che avviene è un controllo sull'attuale popolazione di agenti e sul valore della costante `population`. Se l'attuale popolazione fosse inferiore verrebbero generati abbastanza agenti in modo casuale fino a

raggiungere la quota stabilita. Come per ogni altro parametro di simulazione, anche questo può essere sovrascritto da un valore specificato nel file XML che contiene lo stato iniziale della simulazione. Ogni agente viene generato con una posizione e una direzione casuale, proprio come avviene in NetLogo. Un'altra operazione che avviene nel setup è la conversione, con sovrascrittura, di tutte le costanti che rappresentano degli angoli, da gradi a radianti. Questa conversione permette di usare, nel resto del codice, le costanti come se fossero degli angoli in radianti. L'utente può così specificarli in gradi, che non richiedono molti decimali per rappresentare gli angoli più noti. Il vantaggio dell'uso dei radianti nella simulazione è dovuto al fatto che le funzioni seno e coseno in C sono state implementate usando i radianti, quindi è possibile passare le costanti come argomenti senza spendere tempo in conversioni. In NetLogo, al contrario, ogni funzione trigonometrica è implementata usando i gradi al posto dei radianti.

4.2.3 Le funzioni degli agenti

La funzione `move` calcola i valori `dx` e `dy`, che sono rispettivamente lo spostamento dell'agente lungo l'asse x e lungo l'asse y. Usando una funzione che gestisce lo spazio come un toroide, viene sovrascritta la posizione corrente e comunicata la posizione dell'agente tramite un messaggio un output.

La funzione `flock` è suddivisa in più fasi. Nella prima parte vengono iterati i messaggi in ingresso, scartando quelli provenienti da agenti troppo distanti. FLAME GPU garantisce che entro il raggio specificato si ricevano tutti i messaggi, ma è possibile che si ricevano messaggi provenienti da agenti leggermente più distanti. Viene anche verificato che l'agente nelle vicinanze si trovi all'interno del cono visivo. In NetLogo questo controllo è implementato da una primitiva chiamata `in-cone`. Gli agenti che superano il filtro sono considerati "visibili" e contribuiscono al conteggio di alcuni valori che saranno usati nei passaggi successivi. La parte successiva della funzione implementa l'algoritmo del Flocking già visto in figura 1.2 e descritto nella sezione 1.5. In base all'azione intrapresa dall'agente viene assegnato un colore secondo lo schema della tabella 4.2. Ogni colore è definito da una costante numerica dichiarata come macro da FLAME GPU stesso. In modalità visualizzazione i colori saranno usati per i modelli 3D degli agenti.

4.3 La visualizzazione

In modalità visualizzazione ogni modello viene mostrato a video. FLAME GPU offre una sua visualizzazione di default per tutti i modelli. Questa visualizzazione mostra gli agenti nello spazio come delle sfere che si spostano e

assumono il colore stabilito dalla variabile `colour`, se presente. Tramite delle macro definite nell'header `visualization.h` è possibile controllare l'illuminazione, le dimensioni e il livello di dettaglio delle sfere (Figura 4.1). L'intera visualizzazione è implementata con OpenGL [24] ed è ottimizzata per collaborare con la parte di simulazione. Infatti non avviene nessuna copia dei dati del modello, quando viene mostrato a schermo, perché tutti i dati necessari si trovano già nella memoria della scheda grafica. Ogni agente esprime l'azione intrapresa nell'iterazione precedente tramite il colore assunto, secondo la legenda della tabella 4.2.

Colore	**Azione**
ciano	separazione
giallo	allineamento e coesione
magenta	nessun altro agente visibile

Tabella 4.2: Tabella del significato dei colori assunti dagli agenti nella visualizzazione.

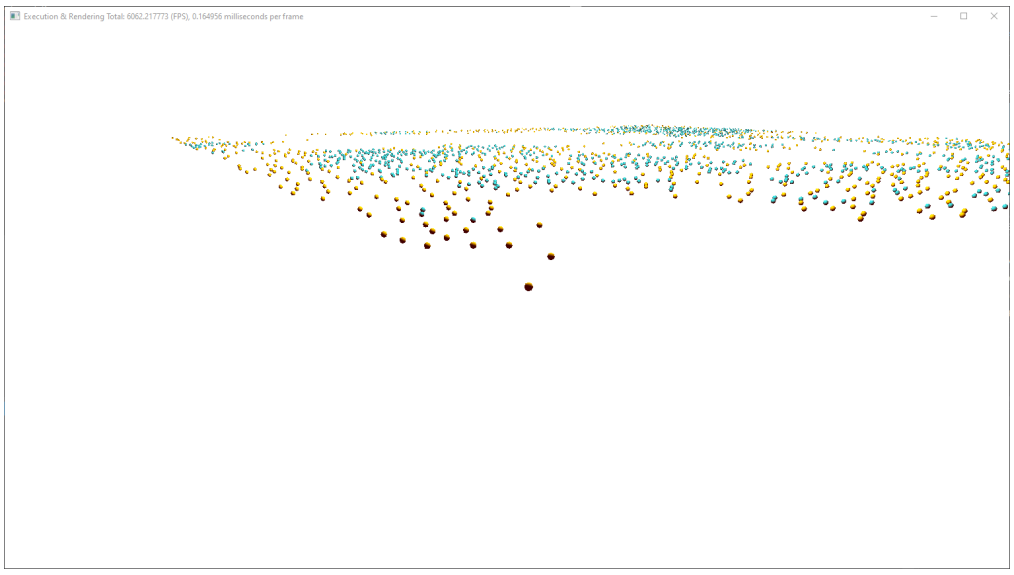

Figura 4.1: Aspetto della visualizzazione predefinita.

In modalità visualizzazione non è possibile muoversi nello spazio, in quanto la telecamera avrà sempre l'origine $(0, 0, 0)$ al centro della finestra. Si può solamente cambiare lo zoom, ruotare il mondo in orizzontale o inclinare la vista in verticale. Inoltre le sfere sono molto meno indicative della direzione rispetto ai piccoli triangolini usati da NetLogo. Per risolvere questi problemi l'intera

visualizzazione predefinita è stata riscritta. Sono stati modificati i modelli 3D degli agenti, i controlli disponibili e il funzionamento della telecamera. Gli agenti nella visualizzazione personalizzata sono ora dei coni che puntano nella direzione definita dalla variabile `heading` (Figura 4.2). Ora è possibile spostarsi liberamente nelle tre dimensioni dello spazio tramite le frecce direzionali della tastiera e il tasto destro del mouse. Si può ancora ruotare la visuale ma è stato rimosso lo zoom. In compenso è possibile vedere gli agenti da vicino spostandosi verso di loro. Sono stati inoltre aggiunti nuovi controlli per la simulazione, ora è infatti possibile impostare la velocità di simulazione tramite i tasti numerici da 1 a 9. Il codice della visualizzazione personalizzata è stato

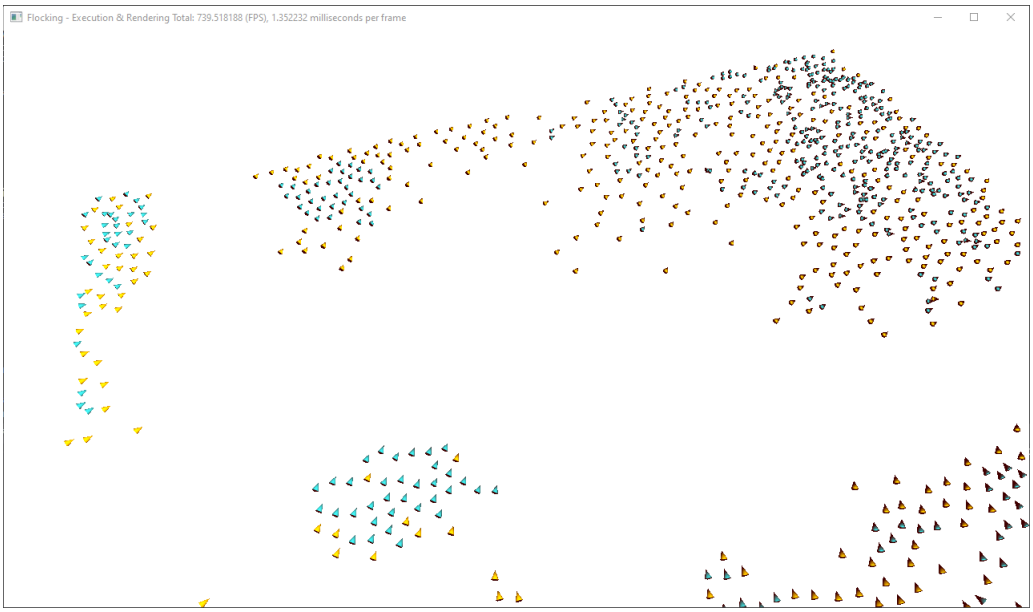

Figura 4.2: Aspetto della visualizzazione personalizzata.

scritto in OpenGL sulla base di quello generato dinamicamente della visualizzazione predefinita. Sono state fatte tuttavia pesanti modifiche che però non riducono le performance rispetto alla versione originale. I coni hanno meno vertici rispetto alle sfere e dato che la variabile che contiene la direzione viene salvata all'interno dei vettori per la posizione non è aumentato il requisito di memoria. Infatti, questi vettori erano già presenti nella visualizzazione di base, che non usava la coordinata `z` in quanto non era inclusa nel modello degli agenti. Il colore e la rotazione dei coni sono calcolati all'interno della *shader* dei vertici, che è un programma eseguito dalla GPU per ogni vertice da disegnare.

In aggiunta alla visualizzazione con OpenGL sono state implementate una modalità verbosa, attivabile definendo il simbolo `FLOCKING_VERBOSE`, e una

modalità che stampa su file i dati sulle azioni intraprese ad ogni iterazione, attivabile con `FLOCKING_PLOT`. Grazie a questi dati, che vengono scritti all'interno di un file di testo chiamato *out*, è possibile eseguire lo script *plot.gp* che si trova all'interno della cartella principale del progetto. Questo file viene interpretato dal programma *gnuplot* [25] che creerà un grafico a partire dai dati, come quelli visibili in figura 5.6 e 5.9, di cui si parlerà successivamente.

4.4 Flocking Vee Formations su FLAME GPU

L'altro modello implementato con FLAME GPU è Flocking Vee Formations. L'algoritmo implementato è basato sul modello omonimo presente nella libreria NetLogo, che è stato pensato per creare delle formazioni a "V" negli stormi, come già visto nella sezione 1.6. In questo caso, le costanti di simulazione sono quelle indicate dalla tabella 4.3. Per permettere agli agenti di

nome variabile	**tipo**	**info**
population	int	Dimensione della popolazione
vision	float	Distanza massima di vista per un agente
speed	float	Velocità iniziale di ogni gente
minimum_separation	float	Distanza oltre la quale due agenti sono "separati"
speed_change	float	Fattore di accelerazione o decelerazione degli agenti
obstruction_angle	float	
max_turn	float	Massima rotazione per una singola iterazione, in gradi
bounds	float	Dimensione del lato del quadrato che costituisce il mondo. Dovrebbe corrispondere a quello specificato nei limiti del partizionamento dei messaggi.
updraft_distance	float	Un agente cercherà di accelerare per raggiungere un altro agente che si trova ad una distanza superiore a quella definita da questa costante.
FOV	float	Angolo, in gradi, corrispondente all'ampiezza della vista frontale di un agente.

Tabella 4.3: I parametri della simulazione, il loro tipo e significato.

controllare la propria velocità è stata aggiunta una variabile chiamata *speed* all'interno della loro memoria, che viene controllata dalla funzione `move` per decidere la posizione dell'agente al termine di un'iterazione. Una differenza dal modello Flocking sta nel fatto che la funzione `flock` è in grado di generare numeri pseudocasuali. Flocking Vee Formations, infatti, non è un modello deterministico. In altre parole, dato uno stato iniziale ed un numero n di iterazioni, lo stato che si raggiunge dopo n iterazioni non è prevedibile, salvo una manipolazione dei numeri pseudocasuali.

Per quanto riguarda il codice che implementa le funzioni, è stato possibile riutilizzare le funzioni di Flocking corrispondenti ad alcune utili primitive di

NetLogo. All'interno della funzione `flock` è stato implementato l'algoritmo di Flocking Vee Formations della libreria NetLogo. Il colore di un agente in Flocking Vee Formations rappresenta l'azione intrapresa dall'agente nell'iterazione precedente, secondo lo schema di tabella 4.4.

Colore	**Azione**
ciano	accelerazione per raggiungere il vicino
rosso	movimento laterale per evitare una visuale ostruita
verde	decelerazione per allontanarsi da un agente
giallo	allineamento della propria velocità a quella del vicino
nero	nessun altro agente visibile

Tabella 4.4: Tabella del significato dei colori assunti dagli agenti nella visualizzazione.

Capitolo 5

Analisi dei Risultati

5.1 Osservazioni sui modelli

Il processo di analisi della correttezza dei modelli consiste nell'osservazione delle simulazioni e nel confronto con l'output della versione implementata in NetLogo. Questo processo, di carattere qualitativo, è stato fatto sia per Flocking che per Flocking Vee Formations. Alcuni casi di studio sono stati codificati in appositi stati iniziali ed osservati un'iterazione per volta, in modo da evidenziare il momento di comparsa di un possibile problema e comprendere come risolverlo. Molti comportamenti assunti dagli agenti, però, dipendono dall'insieme di parametri dati alla simulazione.

Purtroppo, a differenza di NetLogo, con FLAME GPU l'esplorazione dei parametri è più laboriosa. Il modo più semplice disponibile, lo stesso suggerito nella documentazione [21], è quello di creare un file di stato iniziale per ogni valore che si desidera provare e di osservare l'output. Questa operazione richiede molto tempo, sia per creare i file di stato sia per osservare i risultati. Inoltre non è possibile, in questo modo, modificare i parametri a tempo di esecuzione.

Un altro problema dell'esplorazione dei parametri in Flocking sono le modifiche alle dimensioni del mondo e al raggio di vista degli agenti. Infatti, siccome il partizionamento dei messaggi deve essere organizzato secondo dei parametri che dipendono da questi valori, ogni modifica ad almeno uno dei due richiede la ricompilazione dell'intero modello.

5.1.1 Flocking

Con i parametri specificati nel listato 5.1 è possibile osservare gli stessi comportamenti globali visibili nella simulazione di Flocking in NetLogo, lasciando i parametri ai loro valori predefiniti (figure 5.1 e 5.2). Grazie all'aggiunta di

una visuale limitata in ampiezza, rispetto alla versione di Flocking di NetLogo, è possibile osservare che alcuni agenti non sanno di avere dei compagni alle spalle (figura 5.3).

```
<states><itno>0</itno>
<environment>
  <population>200</population>
  <vision>5.0</vision>
  <minimum_separation>1.0</minimum_separation>
  <max_align_turn>5.0</max_align_turn>
  <max_cohere_turn>3.0</max_cohere_turn>
  <max_separate_turn>1.5</max_separate_turn>
  <speed>0.1</speed>
</environment>
</states>
```

Listato 5.1: Parametri simili a Flocking di NetLogo.

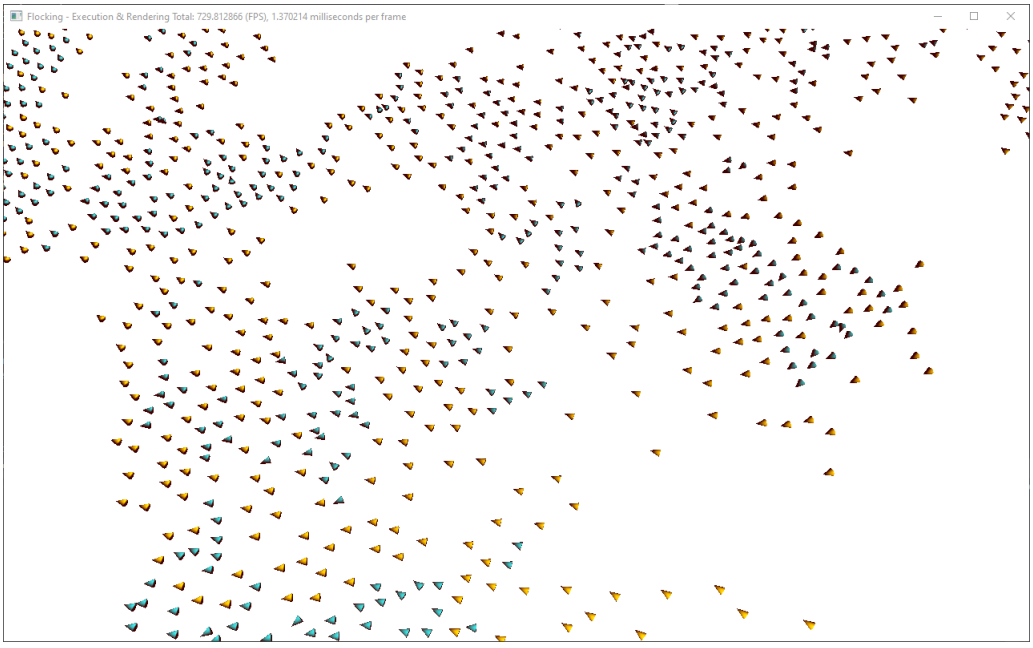

Figura 5.1: Dopo diverse iterazioni gli agenti si muovono quasi tutti nella stessa direzione.

Sperimentando con i parametri è possibile far emergere comportamenti particolari. Rimuovendo la regola della separazione, ovvero impostando la variabile `minimum_separation` a 0, si osserva una costante coesione e allineamento che produce un "serpente" di agenti, come in figura 5.4.

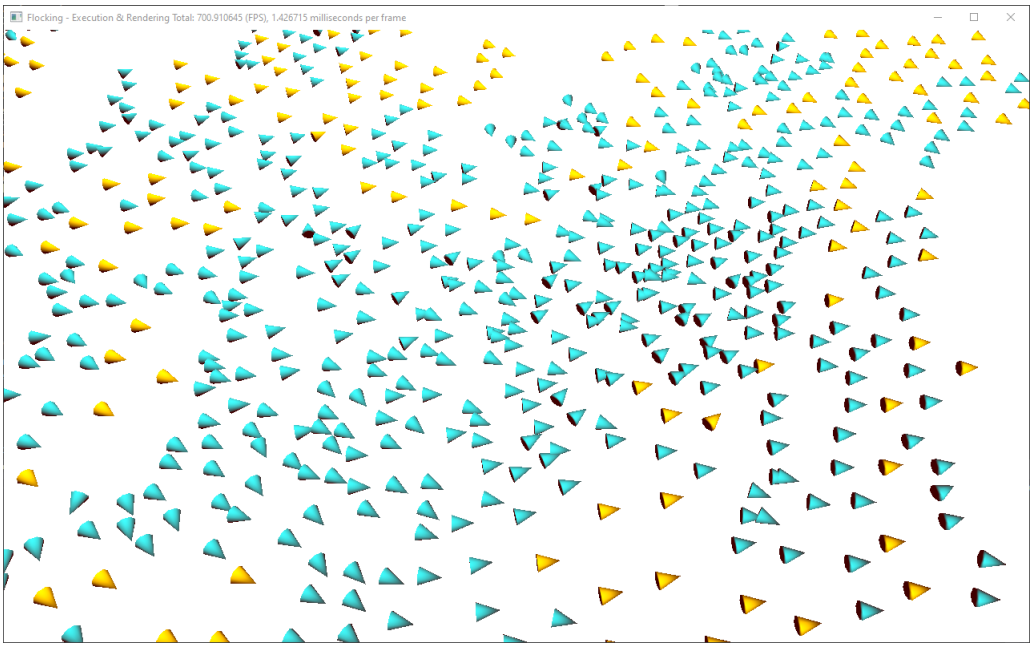

Figura 5.2: Quando due stormi si incontrano avvengono comportamenti apparentemente caotici.

```
<states><itno>0</itno>
<environment>
  <population>2000</population>
  <vision>2.0</vision>
  <minimum_separation>1.0</minimum_separation>
  <max_align_turn>5.0</max_align_turn>
  <max_cohere_turn>3.0</max_cohere_turn>
  <max_separate_turn>1.5</max_separate_turn>
  <speed>0.1</speed>
  <FOV>280</FOV>
</environment>
</states>
```

Listato 5.2: In questo altro insieme di parametri è stata ridotta la distanza di vista e ampliati i limiti del mondo.

Con una minor distanza di vista, invece, gli agenti formano stormi più piccoli e più facili da disgregare. Un esempio del genere è il listato 5.2 che ha prodotto l'immagine 5.5.

Analizzando i dati sulle azioni intraprese dagli agenti si nota che non si raggiunge una perfetta stabilità. Piccoli disturbi nel movimento dello stormo possono causare reazioni a catena che inducono l'intero gruppo a cambiare

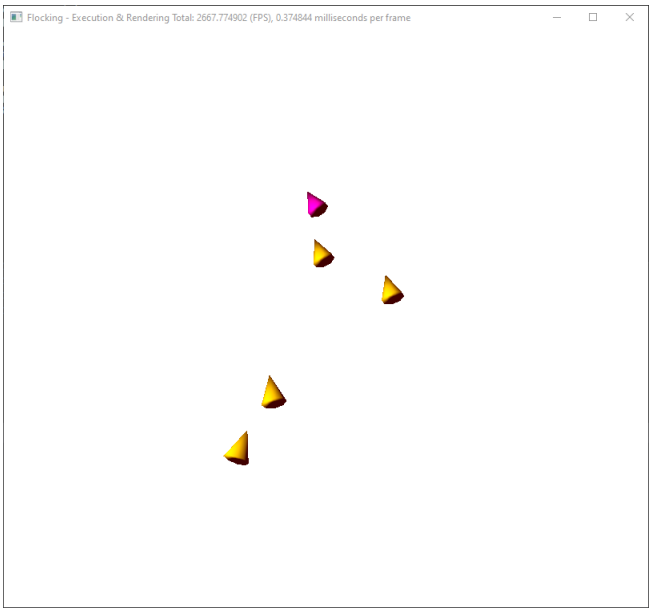

Figura 5.3: L'agente in magenta non vede i compagni alle spalle e pensa di essere isolato. I compagni invece lo stanno seguendo, infatti il giallo rappresenta l'allineamento e la coesione.

leggermente direzione. In figura 5.6 si osserva come dopo poche iterazioni non vi siano quasi più agenti isolati, e la maggior parte degli agenti che si trova nello stormo sia occupata a mantenere il proprio spazio libero allontanandosi dai vicini, che fanno la medesima cosa. Questo equilibrio viene facilmente spezzato da piccoli disturbi. Un evento del genere causa un aggiustamento a catena delle direzioni dello stormo, che fa si che la maggior parte degli agenti eseguano l'azione di allineamento, in giallo nel grafico. I punti in cui avvengono questi disturbi sono visibili in quanto la linea ciano, che rappresenta gli agenti in separazione, viene sostituita da quella gialla, incrociandosi durante lo scambio. Stabilita la nuova direzione, lo stormo torna ad avere in prevalenza agenti in separazione.

Figura 5.4: La rimozione della regola della separazione produce la formazione di un serpente di agenti. In questa immagine il serpente non è ancora del tutto unito.

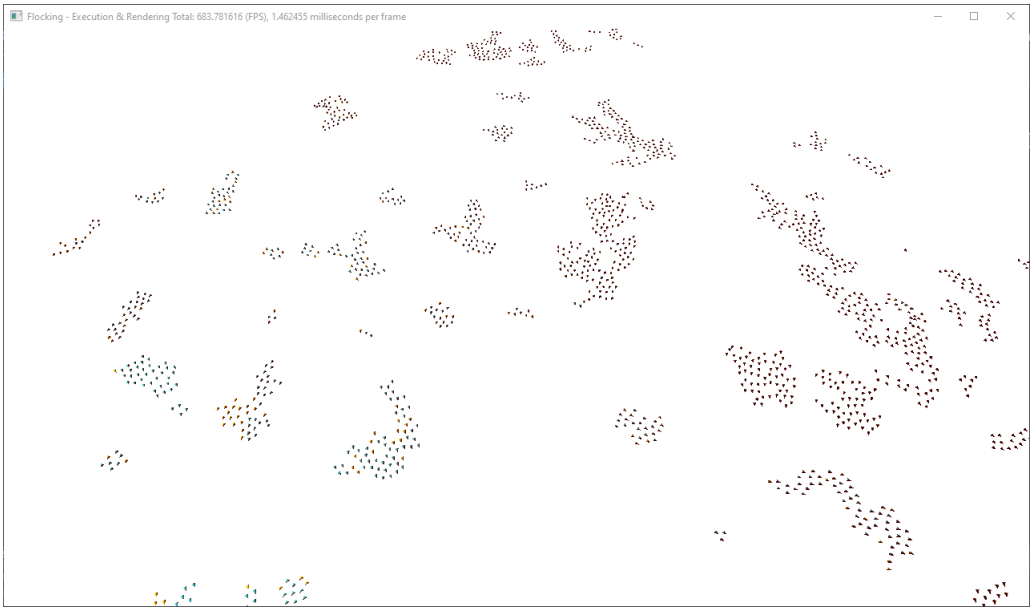

Figura 5.5: Una minor distanza di vista produce stormi più piccoli, in quanto più facili da disgregare.

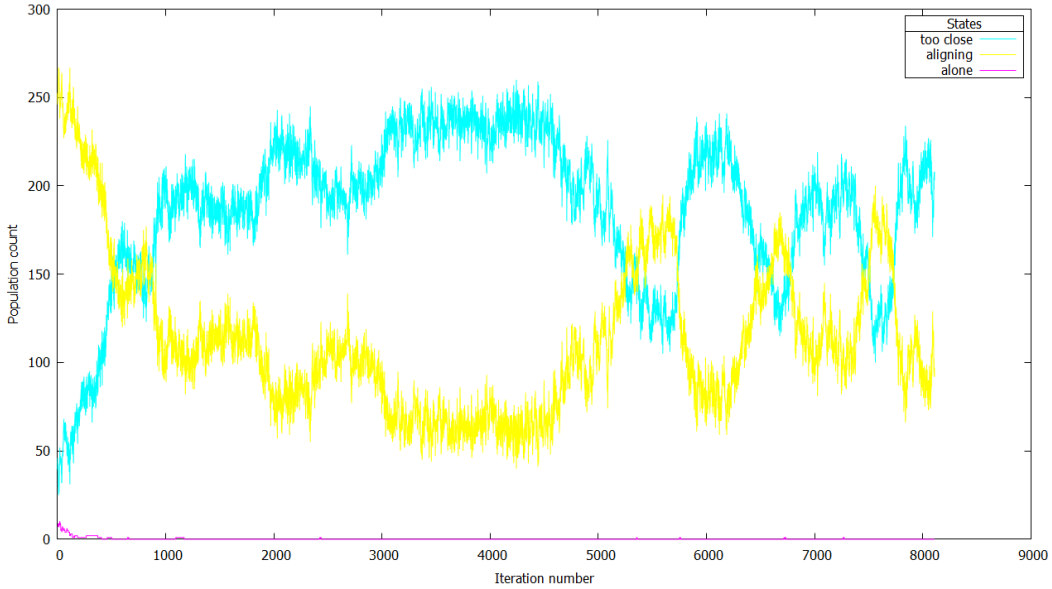

Figura 5.6: Grafico che mostra il conteggio degli agenti per ogni azione, rispetto ad una popolazione di 300 individui con i parametri di simulazione ai valori predefiniti, usando il modello Flocking.

5.1.2 Flocking Vee Formations

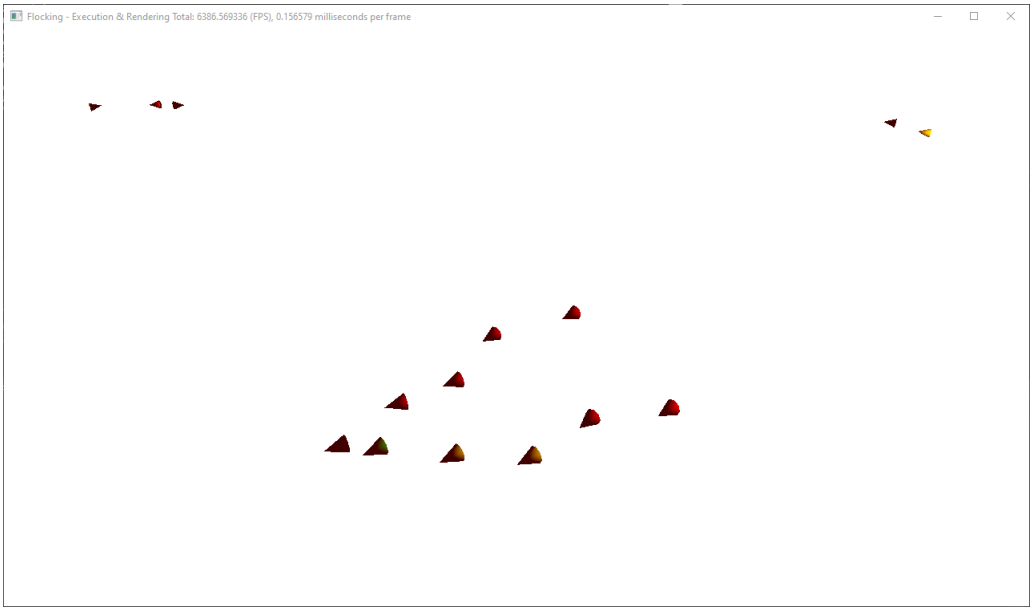

Figura 5.7: Esempio di formazione a V.

Il modello di Flocking Vee Formations non converge come avviene per Flocking. Lo spostamento laterale causato dalla vista ostruita ha la precedenza su tutte le altre azioni tranne l'avvicinamento (vedi sezione 1.6). Questo produce dei movimenti caotici quando due stormi si incrociano, in quanto la maggior parte degli agenti si trova con la vista ostruita e lo spostamento laterale è l'azione che adotta comportamenti che dipendono dal caso. Controllare il valore di `obstruction_angle` permette inoltre di stabilire l'ampiezza della "V" degli stormi. Il suo valore è 45° in figura 5.7 e 30° in figura 5.8.

Dai dati raccolti sulle simulazioni, si osserva come le azioni intraprese dagli agenti ad ogni iterazione in Flocking Vee Formations mantengano una certa proporzione lungo tutta la durata della simulazione (vedi figura 5.9). Questo è dovuto principalmente al fatto che, anche quando fanno parte di uno stormo, gli agenti si trovano spesso la vista ostruita e tentano di spostarsi per liberarsi la visuale, talvolta senza successo.

5.2 Studio delle prestazioni

Misurare le performance di Flocking non è un compito semplice. Ovviamente non è possibile usare le versioni di debug in quanto molto più lente delle versioni di release. Per quanto riguarda la versione console è possibile

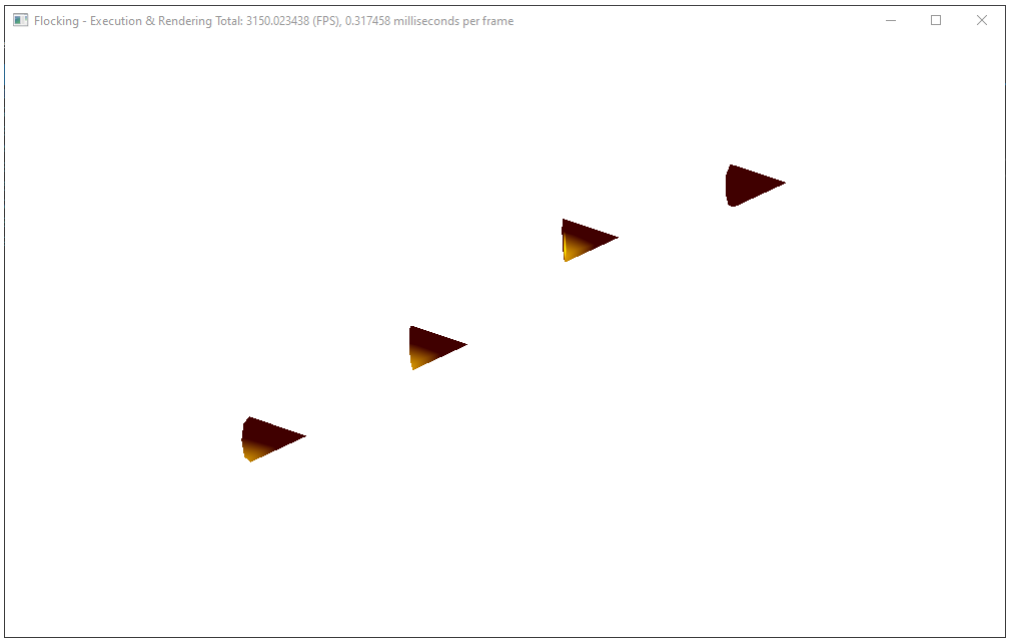

Figura 5.8: Esempio di formazione a linea obliqua.

ottenere il tempo totale di esecuzione della simulazione per un certo numero di iterazioni. Per ognuna di queste il programma scrive un nuovo file XML con l'intero stato del modello, rallentando l'esecuzione a causa dei tempi di I/O. Nella modalità visualizzazione è possibile misurare la frequenza di aggiornamento e la durata della creazione di un frame, per avere un indizio relativo del tempo impiegato con un determinato insieme di parametri piuttosto che un altro. Per superare queste difficoltà è stata creata una visualizzazione che non crea finestre ma semplicemente esegue un numero fisso di iterazioni del modello, stampa il tempo di esecuzione ed esce.

5.2.1 Prestazioni grafiche

Tramite un'analisi qualitativa sulla frequenza dei fotogrammi è possibile osservare come il fattore determinante per la velocità di computazione sia la densità di agenti rispetto all'area (figure 5.10 e 5.11). In particolare si intende la densità massima di agenti presente in una zona, in quanto significa un maggior numero di iterazioni sui vicini per ogni agente nei paraggi del punto critico.

Oltre un certo limite di popolazione però si osserva un forte calo di fotogrammi al secondo nonostante la densità rimanga praticamente invariata, come visibile in figura 5.12. Questo può essere dovuto sia alla presenza di sbalzi di densità causati da uno stato iniziale casuale su un'area molto vasta, sia alle

Figura 5.9: Grafico che mostra il conteggio degli agenti per ogni azione, rispetto ad una popolazione di 300 individui con i parametri di simulazione ai valori predefiniti, usando il modello Flocking Vee Formations.

limitate capacità dell'hardware che è messo sotto sforzo da queste simulazioni.

5.2.2 Misurazione delle prestazioni

Modificando il codice della visualizzazione è stato possibile eseguire un numero prefissato di iterazioni di seguito e stampare in output i dati sul tempo di esecuzione, per studiare come questo cambi in base ad alcuni parametri. In particolare sono stati studiati il numero di agenti, le dimensioni dello spazio e il raggio di vista degli agenti. Per ogni input sono state fatte più misurazioni ed è stata considerata la media dei tempi, in modo da evitare sbilanciamenti dovuti ad altri programmi in esecuzione, L'hardware usato per i test è una macchina dotata di GPU *Nvidia GeForce GTX 1060* con 3 GB di memoria dedicata, su un sistema operativo a 64 bit Windows, 16GB di RAM e CPU *Intel i7-7700K* a 4.4GHz.

In figura 5.13 si osserva come il numero di agenti influenzi negativamente le prestazioni, come previsto. Un risultato interessante è visibile nel grafico di figura 5.14, anch'esso estratto dai dati di tabella 5.1, in cui la massima "quantità di lavoro" non si ha in corrispondenza del minimo numero di agenti. Tale quantità, infatti, comincia a decrescere solamente sopra i 40.0000 agenti

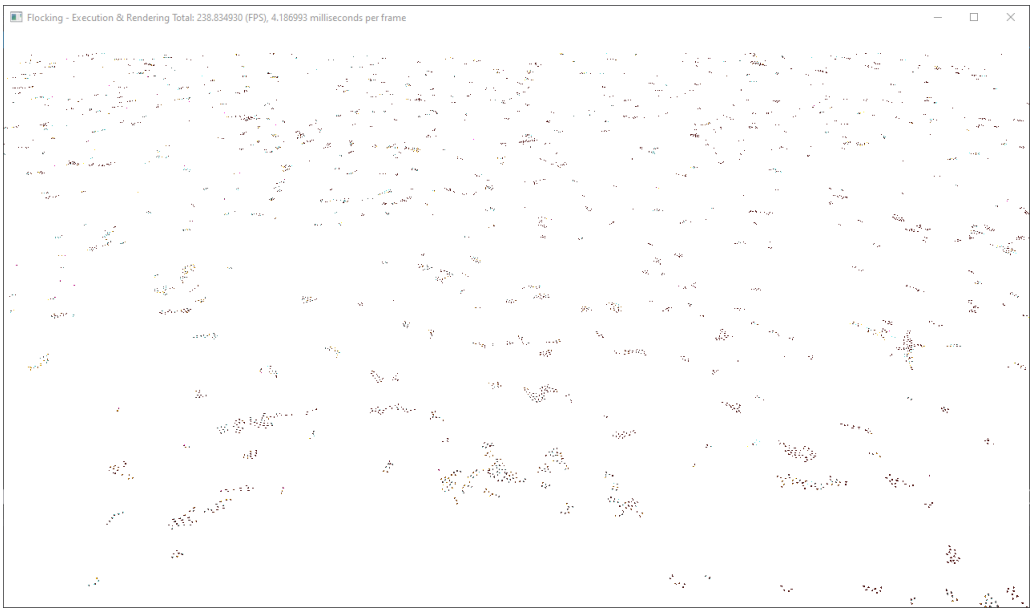

Figura 5.10: 20.000 agenti di Flocking in uno spazio con bordi ampliati, a 238 fps (frame al secondo), 4.1 millisecondi per ogni frame. Il raggio visivo e il partizionamento hanno lunghezza 2,0.

circa. Questa soglia corrisponde, probabilmente, alla massima densità oltre la quale gli agenti sono spesso occupati ad allontanarsi dai vicini, che sono spesso in numero alto.

Per quanto riguarda le prestazioni rispetto alle dimensioni del dominio è possibile osservare, dalla figura 5.15 e dalla tabella 5.3, che si ha un massimo intorno al valore di 5000. Questo valore è la lunghezza, in `float`, di un lato del quadrato che costituisce lo spazio in cui si muovono gli agenti. L'area di questo spazio, e quindi anche l'inverso della densità di agenti, cresce in modo quadratico rispetto al lato. Nonostante ciò si verifica un calo di prestazioni sotto una certa soglia di densità. Questo può essere dovuto ad un aumento delle sezioni del partizionamento dei messaggi, che deve coprire un'area sempre più grande.

In figura 5.16 e in tabella 5.3 vengono mostrati alcuni dati delle esecuzioni in cui viene modificata la distanza di vista degli agenti. In questo caso, per mantenere fissa la dimensione dello spazio, è stato possibile utilizzare solo valori che fossero dei sottomultipli, anche frazionari ma non periodici, di 1000. Le massime prestazioni si hanno con valori bassi di distanza di vista, come ci si poteva aspettare, ma la forma della curva suggerisce un particolare valore attorno al quale le prestazioni migliorano notevolmente. Questa soglia potrebbe essere il valore col quale, mediamente, un agente si vede isolato.

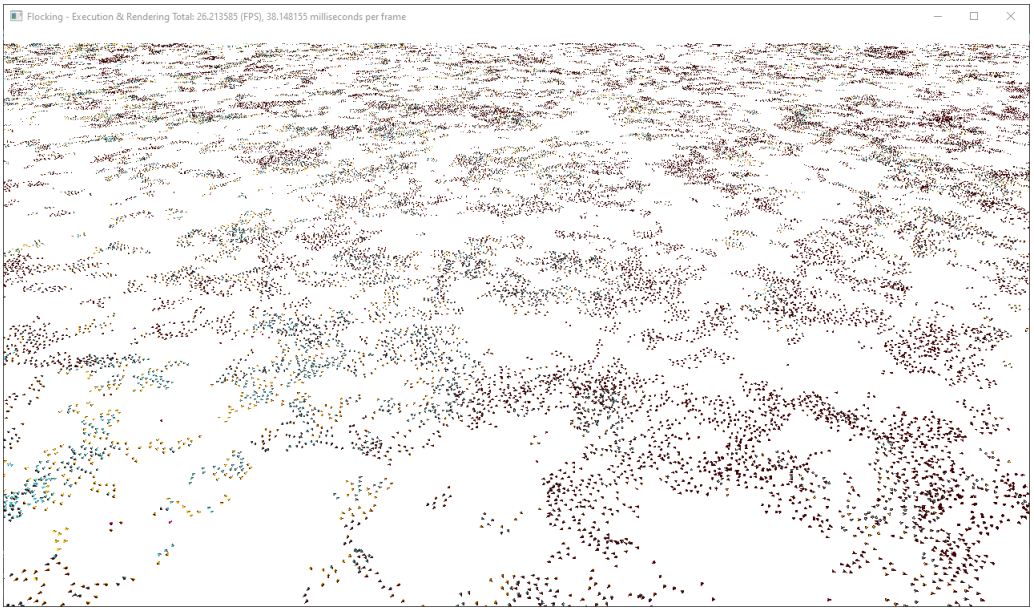

Figura 5.11: 200.000 agenti nello stesso spazio di figura 5.10 a 26 fps con 38 millisecondi per frame. Il modello è sempre Flocking; il raggio visivo e il partizionamento hanno lunghezza 2,0.

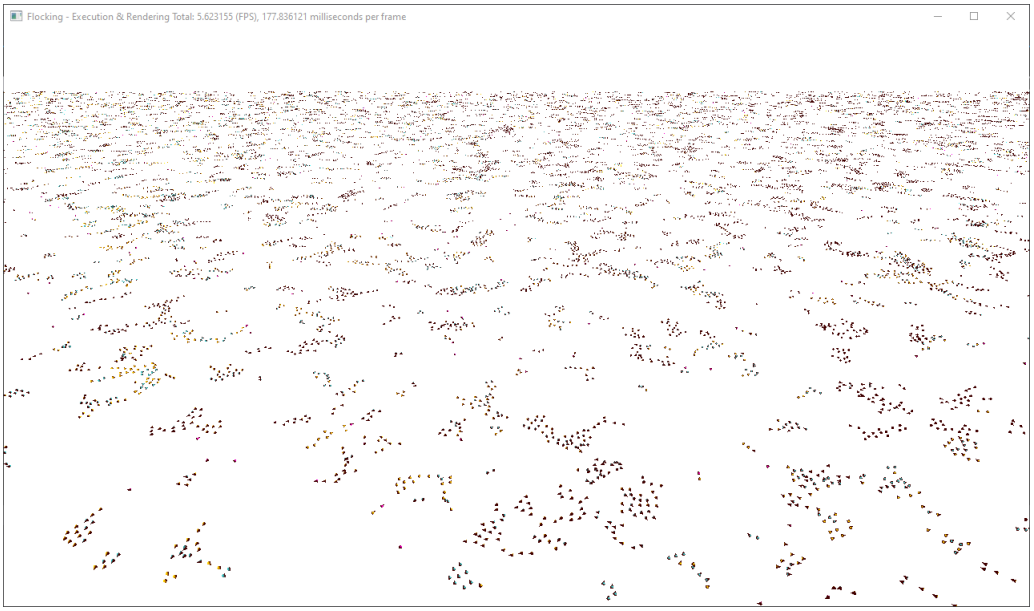

Figura 5.12: Un milione agenti immersi in uno spazio quadrato di lato 50.000, a 5.6 fps con 177 millisecondi per frame. Il raggio visivo e il partizionamento hanno lunghezza 2,0.

Agenti	View	Bounds	Tempo medio (s)	Iterazioni	Iterazioni/s	Iterazioni*agenti/s
100.000	5,00	10.000	149,69	10.000	66,80	6.680.330,53
90.000	5,00	10.000	115,22	10.000	86,79	7.810.897,79
80.000	5,00	10.000	85,38	10.000	117,12	9.369.350,90
70.000	5,00	10.000	61,77	10.000	161,90	11.332.807,14
60.000	5,00	10.000	44,60	10.000	224,20	13.452.152,21
50.000	5,00	10.000	32,76	10.000	305,22	15.260.787,94
40.000	5,00	10.000	24,00	10.000	416,73	16.669.042,70
30.000	5,00	10.000	19,03	10.000	525,50	15.764.864,73
20.000	5,00	10.000	14,47	10.000	691,03	13.820.612,50
10.000	5,00	10.000	8,87	10.000	1.127,96	11.279.551,58

Tabella 5.1: Esecuzioni di Flocking con FLAME GPU in cui viene modificato il parametro della popolazione. Il valore `iterazioni * agenti / s` può essere considerato un indice della quantità di lavoro svolta al secondo.

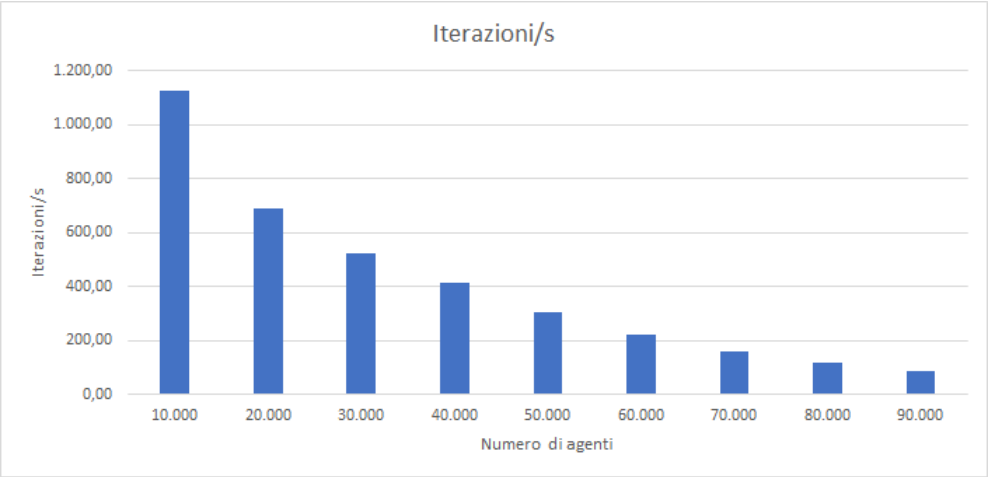

Figura 5.13: Grafico ricavato dalla tabella 5.1 che mostra come variano le iterazioni per secondo rispetto al numero di agenti.

Agenti	View	Bounds	T medio	Iterazioni	Iterazioni/s	Iterazioni*agenti/s
50.000	5,00	1.000,00	32,41852	10000	308,47	15423281,9
50.000	5,00	2.000,00	25,42835	10000	393,26	19663092,8
50.000	5,00	3.000,00	21,51674	10000	464,75	23237726,0
50.000	5,00	4.000,00	20,28639	10000	492,94	24647067,1
50.000	5,00	5.000,00	18,85698	10000	530,31	26515384,7
50.000	5,00	6.000,00	19,20737	10000	520,63	26031668,4
50.000	5,00	7.000,00	19,91614	10000	502,11	25105264,3
50.000	5,00	8.000,00	20,75277	10000	481,86	24093170,8
50.000	5,00	9.000,00	21,43342	10000	466,56	23328058,2
50.000	5,00	10.000,00	22,44128	10000	445,61	22280372,0

Tabella 5.2: Esecuzioni di Flocking con FLAME GPU in cui viene modificato il parametro delle dimensioni del lato dello spazio.

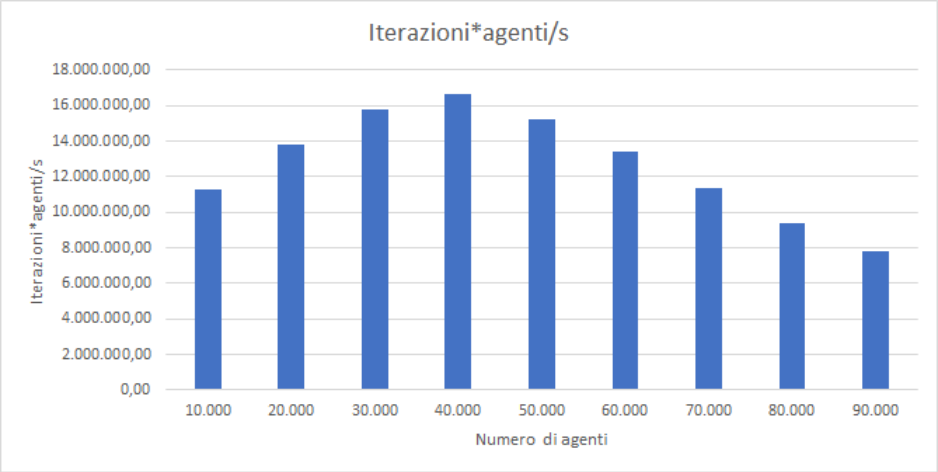

Figura 5.14: Grafico ricavato dalla tabella 5.1 che mostra come variano le iterazioni per secondo, moltiplicate per il numero di agenti, rispetto al numero di agenti.

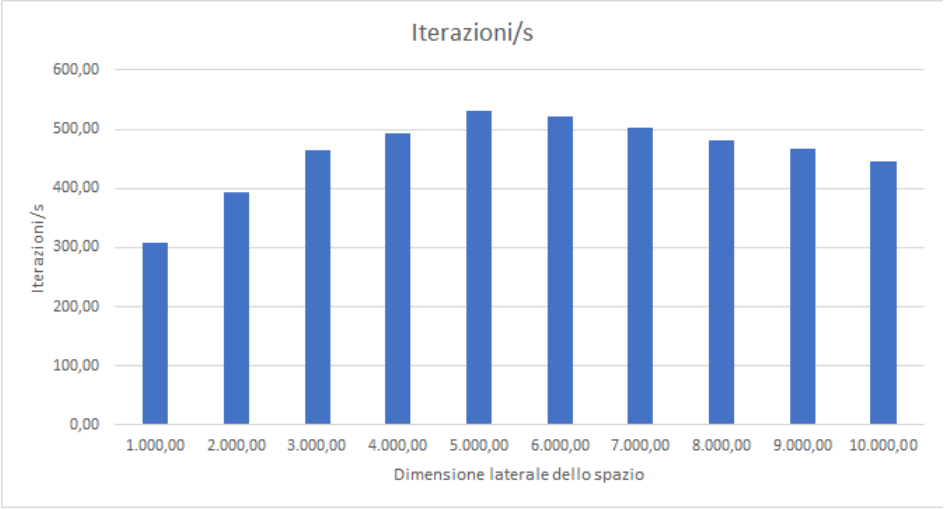

Figura 5.15: Grafico ricavato dalla tabella 5.2 che mostra come variano le iterazioni per secondo rispetto alle dimensioni del dominio.

Agenti	View	Bounds	T medio	Iterazioni	Iterazioni/s	Iterazioni*agenti/s
50.000	10,00	1.000,00	191,03417	10000	52,3466572	2.617.332,9
50.000	8,00	1.000,00	135,38986	10000	73,8607769	3.693.038,8
50.000	5,00	1.000,00	62,72803	10000	159,418375	7.970.918,7
50.000	4,00	1.000,00	45,84018	10000	218,149226	10.907.461,3
50.000	2,50	1.000,00	30,74913	10000	325,212471	16.260.623,5
50.000	2,00	1.000,00	29,01459	10000	344,654221	17.232.711,1
50.000	1,00	1.000,00	27,90489	10000	358,360173	17.918.008,6

Tabella 5.3: Esecuzioni di Flocking con FLAME GPU in cui viene modificato il parametro del raggio di vista degli agenti.

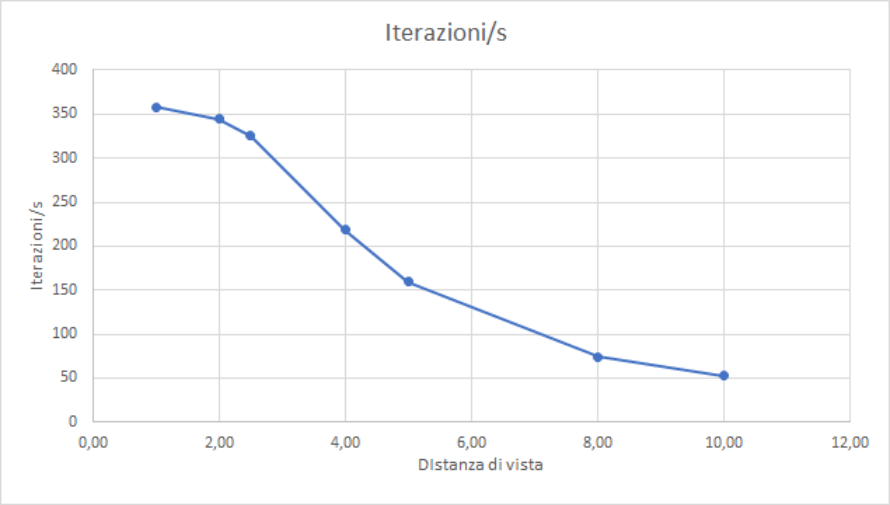

Figura 5.16: Grafico ricavato dalla tabella 5.3 che mostra come variano le iterazioni per secondo rispetto al raggio di vista degli agenti.

Conclusioni

5.3 Risultati

Implementando Flocking e Flocking Vee Formations con FLAME GPU è stato possibile osservare come gli algoritmi comportamentali di questi modelli si comportino su larga scala, cosa impossibile per una versione seriale della simulazione. Per larga scala si intende un dominio più ampio e con popolazioni molto numerose. L'aver realizzato il modello a partire da quello presente nella libreria NetLogo ha inoltre permesso la creazione di un insieme di primitive implementate in FLAME GPU e riutilizzabili per altri modelli simili a Flocking. Allo stesso modo è possibile riutilizzare il codice della visualizzazione in OpenGL creata per migliorare l'esperienza d'uso e per visualizzare meglio la direzione e il moto degli agenti.

5.4 Sviluppi futuri

Grazie al lavoro svolto sono possibili i seguenti sviluppi:

- Approfondire l'esplorazione dei parametri, creare nuove regole comportamentali o più in generale sperimentare col modello Flocking o Flocking Vee Formations per ottenere nuovi risultati;
- Implementare nuovi modelli della libreria NetLogo in FLAME GPU riutilizzando parte del codice, in particolare la porzione che implementa il movimento degli agenti e le primitive;
- Utilizzare la visualizzazione di Flocking e Flocking Vee Formations in sostituzione a quella predefinita di FLAME GPU per visualizzare meglio la direzione degli agenti se questi si muovono nello spazio bidimensionale con coordinata z costante;
- Modificare il modello per applicare l'algoritmo di Flocking o Flocking Vee Formations in tre dimensioni;

- Implementare una nuova visualizzazione che mostri la direzione degli agenti nelle tre dimensioni, per un eventuale modello in cui gli agenti possiedono movimento e direzione arbitrarie nello spazio;
- Modificare la visualizzazione per poter cambiare i parametri della simulazione, ovvero le costanti, durante l'esecuzione, tramite dei controlli grafici simili a quelli posseduti da NetLogo. A livello teorico questo potrebbe essere fattibile se si programmassero dei controlli visuali con OpenGL che impostino delle variabili presenti in memoria. Una funzione eseguita dalla CPU ad ogni iterazione può controllare i valori di queste variabili e sovrascrivere le corrispondenti costanti sulla GPU se rileva delle modifiche, tramite le API fornite da FLAME GPU.

Ringraziamenti

Vorrei ringraziare il mio relatore, il professor Marzolla, che ha reso possibile questo progetto ed è stato di fondamentale aiuto per la stesura di questa tesi. Un grazie anche a tutti gli altri docenti che mi hanno seguito, supportato e aiutato; grazie anche al personale di ateneo che ha contribuito al miglioramento della mia esperienza universitaria sotto molti aspetti. Ringrazio gli amici, i colleghi e i parenti che mi hanno sostenuto in qualunque modo, dalla compagnia alla collaborazione. In particolare ringrazio lo Spaceteam, un gruppo affiatato di colleghi di questo corso di laurea che si è formato nel corso della triennale e mi ha accompagnato nella quotidianità da studente universitario.

Bibliografia

[1] Paul Richmond, Dawn Walker, Simon Coakley, and Daniela Romano. High performance cellular level agent-based simulation with FLAME for the GPU. *Briefings in Bioinformatics*, 11(3):334–347, 02 2010.

[2] Lana and Lilly Wachowski (directors). The Matrix, 1999.

[3] H. Van, H. Van Dyke Parunak, Robert Savit, and Rick L. Riolo. Agent-Based Modeling vs. Equation-Based Modeling: A Case Study and Users' Guide, 1998.

[4] U Wilensky. NetLogo Flocking model, 1998.

[5] C. W. Reynolds. Flocks, Herds, and Schools: A Distributed Behavioral Model. *Computer Graphics: SIGGRAPH '87 Conference Proceedings*, 21(4):25–34, 1987.

[6] NetLogo homepage. `https://ccl.northwestern.edu/netlogo/`.

[7] Michael Friendly. *Advanced Logo: A Language for Learning*. Computer Science for the Behavioral Sciences Series. Lawrence Erlbaum Associates, 1988.

[8] NetLogo user manual. `https://ccl.northwestern.edu/netlogo/docs/NetLogo%20User%20Manual.pdf`.

[9] Perché NetLogo non è parallelizzato. `https://github.com/NetLogo/NetLogo/wiki/Why-Isn't-NetLogo-%22Parallel%22%3F`.

[10] FLAME homepage. `http://flame.ac.uk/`.

[11] D.J. Worth M. Holcome S. Coakley C. Greenough, L.S. Chin. The Exploitation of Parallel High Performance Systems in the FLAME Agent-Based Simulation Framework. Technical report, Science and Technology Facilities Council, June 2008.

[12] FLAME user manual. `http://flame.ac.uk/docs/user_manual.html`.

[13] Rod Smallwood Simon Coakley and Mike Holcombe, editors. *A General Framework for Agent-based Modelling of Complex Systems*, North Campus, Broad Lane, Sheffield, S3 7HQ, UK, 2006. Department of Computer Science,University of Sheffield.

[14] Doxygen. `http://www.doxygen.nl/`.

[15] GCC. `https://gcc.gnu.org/`.

[16] Nvidia programming. `https://www.nvidia.it/object/gpu-programming-it.html`.

[17] FLAME GPU homepage. `http://www.flamegpu.com/home`.

[18] Introduzione a CUDA. `https://developer.nvidia.com/cuda-zone`.

[19] FLAME GPU repository. `https://github.com/FLAMEGPU/FLAMEGPU`.

[20] Python homepage. `https://www.python.org/`.

[21] Documentazione online di FLAME GPU. `http://docs.flamegpu.com/en/master/index.html`.

[22] MPI documentation. `https://www.mpi-forum.org/docs/`.

[23] Repository di Flocking con FLAME GPU. `https://github.com/ldeluigi/Flocking-with-FLAMEGPU`.

[24] Sito web Nvidia OpenGL. `https://developer.nvidia.com/opengl`.

[25] gnuplot homepage. `http://www.gnuplot.info/`.