Pcd-Assigment03 - part 1 Report

Concurrent Boids with Actors

Bedeschi Federica
0001175655 federica.bedeschi4@studio.unibo.it

Pracucci Filippo
0001183555 filippo.pracucci@studio.unibo.it

Anno accademico 2024-2025

Indice

1 Analisi **2**
1.1 Descrizione del problema 2
1.2 Visione concorrente 2

2 Design e architettura **3**
2.1 Interazioni 4
2.1.1 Interazioni per il funzionamento della simulazione . . . 4
2.1.2 Interazioni per la terminazione della simulazione 5
2.1.3 Interazioni per la sospensione della simulazione 6

3 Conclusioni **7**

Analisi

1.1 Descrizione del problema

Il problema dei Boids consiste in una simulazione del comportamento degli stormi di uccelli. Ogni boid aggiorna la propria posizione in maniera continuativa, solo dopo aver aggiornato la propria velocità in base a quelle degli altri boid. In questo modo si simula il comportamento di gruppo, che può essere condizionato dalle seguenti regole:

- separazione: il boid vira al fine di evitare il sovraffollamento locale (dunque si allontana dai boid vicini);
- allineamento: il boid vira al fine di allinearsi alle traiettorie di volo dei boid vicini;
- coesione: il boid vira al fine di muoversi verso la posizione media (baricentro) dei boid vicini.

1.2 Visione concorrente

Dal punto di vista concorrente gli aggiornamenti delle velocità e delle posizioni per ogni boid possono essere fatti concorrentemente. Tuttavia, bisogna tenere conto che tutte le posizioni possono essere aggiornate solo dopo aver aggiornato tutte le velocità, e viceversa.
A livello di interfaccia grafica, l'aggiornamento della vista deve essere eseguito a seguito dell'aggiornamento di tutte le posizioni dei boid.

Design e architettura

Abbiamo utilizzato il paradigma ad attori sfruttando il framework **Akka** in Scala, in particolare **Akka Typed** e lo stile funzionale coi **Behavior**. Abbiamo creato diversi attori sia per il model (quindi la business logic) che per la view (ovvero la visualizzazione e le interazioni con l'utente):

- model:
  - `BoidManager`: attore che crea e gestisce i `BoidActor`, ed interagisce con il `ViewActor`;
  - `BoidActor`: attore che rappresenta un singolo boid, gestendo la propria posizione e velocità;
- view:
  - `ViewActor`: attore responsabile dell'interfaccia grafica e degli eventi generati dalle interazioni con l'utente;
  - `DrawerActor`: attore che si occupa di "disegnare" i boid nell'interfaccia grafica;
  - `EnteringPanelActor`: attore responsabile del pannello di avvio e del conseguente inizio della simulazione.

Inoltre è presente un `MainActor`, che si occupa di creare il `BoidManager` e il `ViewActor` per avviare la simulazione, e rappresenta quindi la root della gerarchia.
Per la gestione della simulazione abbiamo optato per una soluzione centralizzata, in cui il coordinatore dei `BoidActor` (che sono in numero pari ai boid, grazie al disaccoppiamento degli attori dai thread fisici) è il `BoidManager`, fatta eccezione per lo scambio delle posizioni e delle velocità che avviene in maniera distribuita tra i `BoidActor`. Quest'ultima scelta è stata fatta perché per la nostra visione del paradigma ad attori abbiamo deciso di dare

più importanza ai `BoidActor`, assegnando loro la responsabilità di aggiornarsi e comunicare tra loro per ottenere le informazioni necessarie, nonostante questo impatti negativamente su prestazioni e scalabilità.

2.1 Interazioni

Per gestire le interazioni abbiamo utilizzato il pattern **Request-Response**, ad eccezione dei casi dove era necessario inviare messaggi a sè stessi per reagire ad un evento, in cui abbiamo sfruttato le **Future** e il `pipeToSelf`. Inoltre per gestire le operazioni bloccanti, ovvero `Thread.sleep` all'interno del `DrawerActor` per rispettare il frame rate, abbiamo utilizzato, oltre a **Future** e `pipeToSelf`, anche un dispatcher dedicato come contesto di esecuzione.

2.1.1 Interazioni per il funzionamento della simulazione

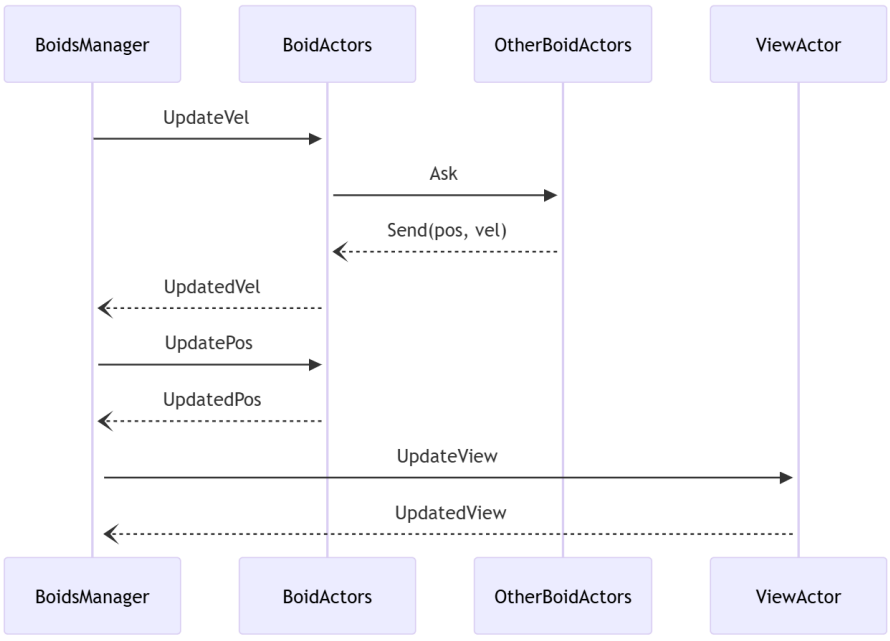

Figura 2.1: Diagramma di sequenza - simulazione

Il `BoidsManager` per eseguire la simulazione gestisce ciclicamente le interazioni mostrate nel diagramma in Figura 2.1. Innanzitutto manda ad ogni

`BoidActor` un messaggio `UpdateVel` per comunicargli di aggiornare la propria velocità. Ogni `BoidActor` per poterlo fare necessita delle velocità e delle posizioni di tutti gli altri `BoidActor` che sono considerati a lui vicini; per ottenerle invia un messaggio `Ask`. Una volta ricevute tutte le risposte (messaggio `Send`), aggiorna la propria velocità ed invia il messaggio `UpdatedVel` al `BoidsManager`. Quest'ultimo una volta ricevuti tutti i messaggi `UpdatedVel` richiede l'aggiornamento di tutte le posizioni inviando un messaggio `UpdatePos` ad ogni `BoidActor`, il quale, una volta ricevuto, risponderà con un messaggio `UpdatedPos` dopo aver aggiornato la propria posizione. Infine, il `BoidManager` una volta ricevuti tutti i messaggi `UpdatedPos` richiede l'aggiornamento della view mandando il messaggio `UpdateView` al `ViewActor`, il quale, una volta ricevuto, risponderà con un messaggio `UpdatedView` dopo averla aggiornata. Il `BoidManager`, ricevuto il messaggio `UpdatedView` ricomincerà la procedura dall'inizio.

2.1.2 Interazioni per la terminazione della simulazione

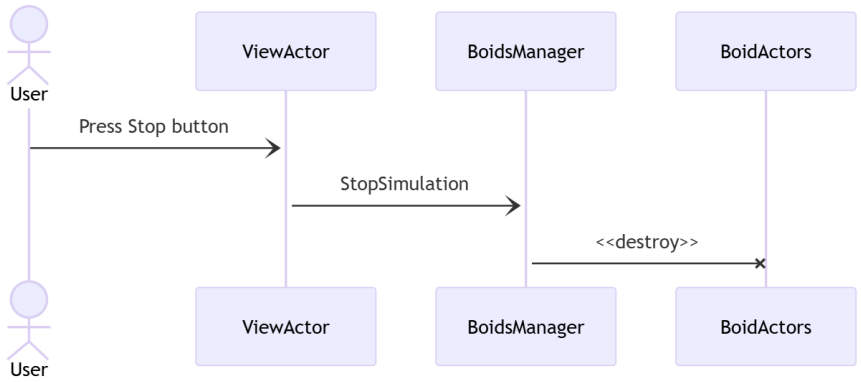

Figura 2.2: Diagramma di sequenza - terminazione della simulazione

Il `BoidsManager` termina la simulazione quando riceve il messaggio `StopSimulation`, inviato dal `ViewActor` in risposta all'evento generato dalla pressione del pulsante "Stop", come si vede nel diagramma in Figura 2.2.

Per fare ciò distrugge tutti i `BoidActor` e si mette in attesa del messaggio `Start` per avviare una nuova simulazione.
Interazioni analoghe sono utilizzate per notificare al `BoidManager` l'aggiornamento dei valori degli slider.

2.1.3 Interazioni per la sospensione della simulazione

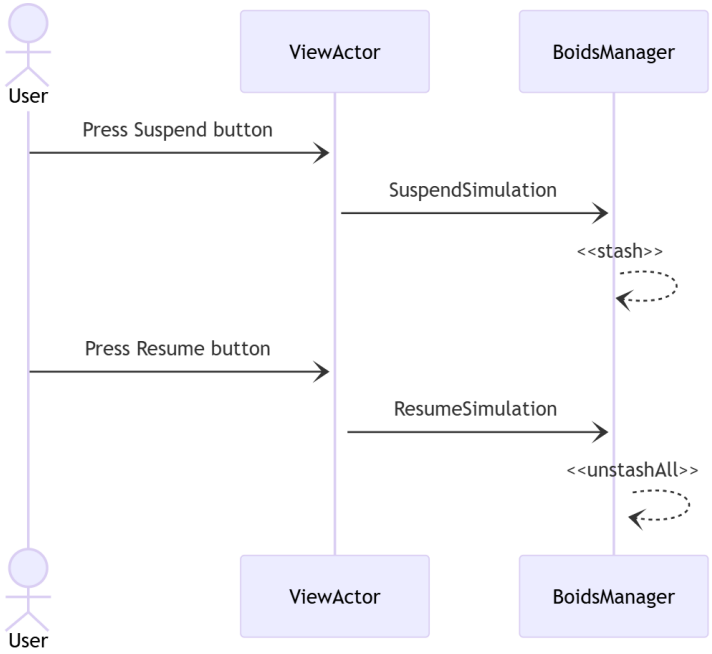

Figura 2.3: Diagramma di sequenza - sospensione della simulazione

Il `BoidsManager` sospende la simulazione quando riceve il messaggio `SuspendSimulation`, inviato dal `ViewActor` in risposta all'evento generato dalla pressione del pulsante "Suspend", come si vede nel diagramma in Figura 2.3. Per fare ciò si mette in uno stato in cui tutti i messaggi che riceve li inserisce in un buffer tramite *stashing*. Non appena il `BoidsManager` riceve il messaggio `ResumeSimulation`, inviato dal `ViewActor` in risposta all'evento generato dalla pressione del pulsante "Resume", riprende la simulazione recuperando i messaggi contenuti nel buffer tramite *unstashing*.

Conclusioni

Utilizzando il paradigma ad attori è stato possibile concentrarsi sulle singole entità e sulle responsabilità di ognuna, separando le computazioni interne dalla logica delle interazioni, la quale avviene tramite scambio di messaggi. Inoltre con gli attori non dobbiamo preoccuparci di gestire i thread fisici, ma grazie alla loro leggerezza possiamo creare un alto numero di attori; per questo motivo infatti abbiamo optato per creare un `BoidActor` per ogni boid. In questo caso, a differenza del primo assignment, non è stato necessario utilizzare le barriere, in quanto è sufficiente uno scambio di messaggi appropriato con il `BoidManager`, permettendo quindi una gestione più semplice ed intuitiva.