Relazione boids

Ilaria Antonellini, Camilla Berti, Maria Blasi Toccaceli

Agosto 2022

Indice

Introduzione

Il progetto Boids è volto alla simulazione del comportamento di un branco di pesci in movimento in uno spazio bidimensionale. La traiettoria che viene seguita dai boids, determinata dall'interazione tra vicini, viene stabilita sulla base di tre regole: separazione, allineamento e coesione. Ognuna di queste fornisce una componente della variazione di velocità del boid, e grazie ad opportune formule è possibile prevedere, dopo un certo intervallo di tempo, le nuove coordinate e velocità di ogni boid e la sua tendenza a formare con gli altri un gruppo coeso.

1 Scelte progettuali e implementative

Per la simulazione sono state create classi e funzioni implementate in modo da poter interagire tra loro in maniera efficiente.

1.1 BoidState

Ad ogni istante di tempo t, ogni pesce che costituisce il branco è caratterizzato da una certa posizione x, y, e da una certa velocità v_x, v_y. Si è quindi definita una struct denominata `BoidState` che contiene queste informazioni. Al fine di un completo e corretto funzionamento di questa classe sono stati implementati alcuni operatori simmetrici in termini di funzioni membro, tipo `@=`, e altri come free functions, sfruttando l'operator overloading. Si è scelto di implementare fuori dalla classe gli operatori non simmetrici per non appesantire più del necessario la classe stessa. Inoltre, sempre inerenti a questa classe, si sono definite due free functions, `norm` e `velocity_norm`, che calcolano rispettivamente la distanza tra due boids e il modulo della loro velocità.

1.2 Components, neighborscontrol

Lavorando in due dimensioni e avendo quindi a che fare con grandezze di tipo vettoriale, è stata definita anche la struct `Components` che permette di tenere traccia delle componenti vettoriali sia della velocità che della posizione (è stata utilizzata un'unica struct per entrambe le grandezze per evitare di appesantire il codice). In modo analogo a quanto fatto per BoidState, sono stati poi implementati gli operatori per operazioni algebriche e booleane.
Sono state poi definite anche due funzioni `size` che restituiscono rispettivamente la dimensione di un vettore di BoidState e di double come tipo intero. Come accennato nell'introduzione, il moto dei pesci è determinato da tre regole che influenzano la traiettoria solo se i boids sono considerati vicini. Per stabilire la vicinanza, è stata implementata la funzione `neighborscontrol`, che ritorna un vettore i cui elementi sono i boids vicini a quello considerato, determinati dalla regola:

$$|\vec{x}_{b_i} - \vec{x}_{b_j}| < d$$

1.3 Regole di nuoto

Si è scelto di rappresentare le regole di nuoto come classi per poi implementare il loro utilizzo come function objects. Si sono quindi definite tre classi:
`SeparationRule`, `AlignmentRule` e `CohesionRule`, in ciascuna delle quali è stato definito l'operatore (). La scelta di implementare le regole di nuoto come oggetti funzione al posto che funzioni vere e proprie, in cui passare anche i parametri s, a e c, che rimangono costanti durante ogni simulazione, deriva dalla volontà di tener separati gli oggetti su cui queste funzioni operano, cioè posizione e velocità dei boids, da ciò che si è usato per cambiare le componenti, per rendere più espressivo il codice e facilitare l'applicazione di queste regole ove necessario. Ogni regola è caratterizzata da una certa costante che determina la propria influenza sul moto dei boids. L'operatore () restituisce un oggetto

di tipo `Components`, che rappresenta il contributo dato alla velocità del boid-iesimo determinato da quella particolare regola. Inoltre, è stata implementata la funzione booleana `check_ownership` da poter utilizzare all'interno delle tre classi al fine di controllare che il boid a cui si applicano le regole appartenga al vettore passato come parametro.

1.3.1 Separation

Determina lo spostamento del boid da quelli vicini qualora la distanza sia inferiore a un certo d_s fissato (molto inferiore a quello definito prima per stabilire i boids vicini). Qui le componenti della velocità sono date da:

$$\vec{v}_1 = -s \sum_{j \neq i} (\vec{x}_{b_j} - \vec{x}_{b_i})$$

L'operatore () sfrutta `std::transform` per calcolare le differenze tra i boids e il boid-iesimo, per poi farne la somma e restituire v_1, dopo aver moltiplicato per il fattore s.

1.3.2 Alignment

Lo scopo dell'allineamento è di modificare la direzione della velocità per far sì che il boid si allinei alle traiettorie di nuoto dei vicini. La componente della nuova velocità data da questa regola è della forma:

$$\vec{v}_2 = a(\frac{1}{n-1} \sum_{j \neq i} \vec{v}_{b_j} - \vec{v}_{b_i})$$

in cui la condizione $a < 1$ rappresenta l'invariante di classe. Anche in questo caso, all'interno di operator() si usa `std::accumulate` per la sommatoria e la funzione `size` precedentemente descritta per ricavare n, calcolando così v_2.

1.3.3 Cohesion

Cohesion garantisce la formazione di un branco compatto, inducendo il boid a sterzare verso il centro di massa dei boids vicini. Come prima cosa è necessario calcolare il centro di massa dato da:

$$\vec{x}_c = \frac{1}{n-1} \sum_{j \neq i} \vec{x}_{b_j}$$

che viene usato per calcolare il terzo e ultimo contributo della nuova velocità utilizzando la seguente formula:

$$\vec{v}_3 = c(\vec{x}_c - \vec{x}_{b_i})$$

È stata quindi implementata prima la funzione `centre_of_mass` che fornisce il primo vettore posizione, $\vec{x}_c$, e successivamente l'operatore () che restituisce $\vec{v}_3$. È stato deciso di separare il calcolo del centro di massa in una funzione a parte sia per poter alleggerire l'implementazione dell'operatore () e sia per avere la possibilità di testare la corretta implementazione delle due formule.

1.4 Boids class

È stata definita un'ulteriore classe denominata `Boids` che permette di costruire un oggetto in grado di descrivere il branco e la sua evoluzione. Per alleggerire la definizione della classe `Boids` sono state create alcune free functions: `velocity_limit` e `borders`. La prima impone un limite massimo e minimo per la velocità dei boids, la seconda impone dei vincoli sulle posizioni assumendo che il moto dei boids avvenga in uno spazio toroidale, facendo sì che se un pesce raggiunge il bordo, viene teletrasportato dall'altra parte. Si procede poi alla definizione della classe Boids. Nei dati membri di questa classe sono presenti: `n_`,il numero di boids che si vogliono nella simulazione, che deve essere maggiore di 1 e rappresenta l'invariante di classe, `d_`, la distanza usata per determinare la vicinanza tra boids, i parametri `s_, a_` e `c_` per costruire i tre oggetti funzione delle regole di nuoto e un vettore di BoidState, `boids_`, che rappresenta il branco. Come funzioni membro sono state implementate:

- `singleboid`: utilizza la legge oraria per stabilire i valori di posizione e velocità dopo un certo Δt di un singolo pesce: $\vec{v}_{b_i} = \vec{v}_{b_i} + \vec{v}_1 + \vec{v}_3 + \vec{v}_3$, $\vec{x}_{b_i} = \vec{x}_{b_i} + \vec{v}_{b_i}\Delta t$.

  La funzione viene applicata solo nel caso in cui la dimensione del vettore risulti maggiore di 1 (in caso contrario non avrebbe senso applicare le regole, il boid continuerebbe a muoversi di moto rettilineo uniforme mantenendo costante la sua velocità, non avendo boids vicini con cui interagire).

- `evolution`: metodo che permette di calcolare l'evoluzione dell'intero branco dopo un certo intervallo di tempo. L'intento era di rendere le istruzioni che contiene questa funzione il più chiare e lineari possibile, si può notare infatti che è costituita da chiamate di funzioni già definite precedentemente: inizialmente per ogni elemento viene chiamata `neighborscontrol` e `singleboid` e successivamente viene fatto il controllo sui limiti della velocità e sui bordi.

- `totalboids`: restituisce il vettore contenente tutti i BoidState dei pesci che costituiscono il branco.

- `pushback`: permette di inserire all'interno del vettore nuovi boids (solo se all'interno del vettore non è già presente un boid con le medesime coordinate, cosa che viene controllata dalla funzione booleana `same_pos_check`). Questa funzione è stata implementata solamente al fine di semplificare la creazione di test per il programma.

Le funzioni membro che sono state definite descrivono in maniera esaustiva e minimale il comportamento di un oggetto di tipo Boids. Si è scelto di implementare il numero minimo necessario di metodi all'interno della classe per rendere l'utilizzo delle varie funzionalità il più versatile e trasversale possibile. La funzione `same_pos_check` chiamata in `pushback` è stata implementata come free function. Di quest'ultima viene fatto l'overloading: viene creata una free

function analoga in comportamento che prende tuttavia parametri differenti, usata negli assert e nella generazione casuale dei boids. Entrambe ritornano vero se non viene trovato nessun elemento con la stessa posizione.

1.5 Statistic

Sono state definite altre free functions relative a `Boids`. La prima è `statistic` che, dopo aver fatto evolvere il branco di un certo intervallo di tempo, calcola il valore medio della distanza tra i boids e la loro velocità media con le relative deviazioni standard della media. La seconda è `state` che ritorna sotto forma di stringa tali valori. La scelta di far ritornare questi valori come stringa è dovuta a necessità legate all'implementazione della parte grafica. È stato deciso di spezzare in due funzioni diverse l'operazione di calcolo e di stampa per poter testare i calcoli effettuati da `statistic`. Prima di implementare quest'ultima funzione, è stata definita un'altra struct, `Stats`, al fine di rendere più ordinato il codice e poter racchiudere i dati statistici in un unico oggetto. Grazie all'utilizzo di cicli for e algoritmi è stata implementata la media aritmetica per calcolare i valori medi di distanza e velocità e la seguente formula per la deviazione standard della media.

$$\sigma_{\bar{x}} = \sqrt{\frac{\bar{x^2} - \bar{x}^2}{N}}$$

La scelta di implementarle come free functions deriva dal fatto che completano e impreziosiscono la classe `Boids` ma non influiscono sul suo significato profondo, quindi si è preferito alleggerire la definizione di quest'ultima.

1.6 Asserts ed eccezioni

Sono stati aggiunti asserts ed eccezioni all'interno del codice. Mediante gli asserts si sono espresse condizioni booleane che si prevedono essere verificate in punti strategici del programma, che permettono di stabilirne la correttezza. Sono state inserite asserzioni in punti particolari del codice, come, per esempio, all'interno della definizione degli operatori () nelle classi dedicate alle regole di nuoto, riguardo la size del vettore. Quando l'oggetto funzione viene chiamato, in `singleboid`, è certo che la dimensione del vettore sia maggiore di 1.
Le eccezioni sono state inserite in punti strategici del codice per impedire che si verifichino comportamenti indesiderati e, nel caso in cui ciò avvenga, per notificare all'esterno che si è verificato un errore in fase di esecuzione del codice. In questo programma si è scelto di utilizzare eccezioni di tipo `std::runtime_error`, in modo che, ogni volta che si verifica un errore, viene restituita la stringa che contiene il motivo per cui l'eccezione è stata sollevata. Per esempio viene sollevata un'eccezione in `AlignmentRule` qualora il valore della costante di allineamento inserito in input sia maggiore di 1.

2 Implementazione della grafica e main

Per simulare i boids nel main è stato fatto uso della libreria grafica SFML. Il programma accetta in input il numero di boids e i parametri `s, a` e `c` e costruisce un oggetto di tipo `Boids`. La distanza che determina i boids vicini e il raggio di influenza della regola di separazione sono fissati rispettivamente a 150 e 15 pixel. Grazie all'algoritmo `generate` i boids vengono generati con posizione e velocità casuali. Dato che i numeri sono generati uniformemente in maniera pseudo-casuale, quindi hanno alla base un algoritmo matematico, al fine di ottenere sempre sequenze diverse è necessario fornire un seme sulla base di un valore, naturalmente casuale. Per questo si usa un oggetto di tipo `std::random_device`. Dopo aver generato oggetti di tipo `BoidState` fino a n (il numero di boids totali), viene effettuato un controllo, per verificare che non siano stati generati boids con la stessa posizione e nel caso ne vengano trovati, vengono cancellati e rigenerati. Il programma effettua anche una simulazione non grafica di 120 secondi, stampando a schermo sul terminale i valori statistici di come si muovono i boids ogni due secondi, attraverso la funzione `simulate`, che considera le distanze e le velocità medie del gruppo, con rispettive deviazioni standard. Successivamente vengono creati e personalizzati degli elementi grafici, tutti forniti da SFML:

- La finestra, ovvero l'area di simulazione.
- Una sprite, un oggetto che permette di poter impostare una texture, in questo caso lo sfondo.
- Un triangolo, che rappresenta un singolo boid.
- Un oggetto di tipo testo, per vedere ad ogni istante di tempo i dati statistici sulla finestra.
- Un rettangolo, che fa da sfondo al testo.

Dopodiché inizia il game loop, il cuore della parte grafica. Quest'ultimo viene girato 30 volte al secondo. Al suo interno viene chiamata la funzione `evolve`, definita all'inizio del main, che ritorna il vettore dell'oggetto `Boids` dopo che si è evoluto di un certo Δt, fissato a 0.001 secondi. Per far sì che la simulazione dei boids vada di pari passo con il game loop, `evolve` prende come parametro anche la variabile `step_evolution`, che calcola quante volte deve esser chiamata `evolution` (che fa evolvere il sistema al Δt fissato), per star dentro a 30 fps. Per rendere la simulazione più veloce, è stato stabilito un fattore di conversione tra il tempo di tipo `sf::Time`, pari a un millesimo di secondo e il parametro Δt di tipo double da passare a evolution, in modo che un millesimo di secondo corrisponda a 0.1 double. Il rapporto 1 s = 100 double è presente nel codice ogni qual volta si deve fare la conversione, come nella funzione `simulate` o quando viene chiamata `state` nel main.
Dopo aver pulito la finestra con il metodo `clear`, si imposta la posizione di ogni triangolo e lo si disegna. Dopo aver raffigurato anche il box della statistica, viene chiamato il metodo `display` e il game loop ricomincia.

3 Physical design

Al fine di rendere il codice più ordinato, il progetto è stato suddiviso in più file: un header file, quattro file sorgente e un file con i test. In questo modo ogni file non supera le 150/200 righe di codice (esclusi i test), la lettura e quindi la comprensione risultano ancora più immediate grazie anche alla scelta di mantenere parti del codice finalizzate allo stesso scopo all'interno dello stesso file.

- boids.hpp: è l'header file, file autonomo contenente tutte le definizioni delle classi e dei rispettivi costruttori e tutte le dichiarazioni delle funzioni, escluse quelle definite nel main. Questo file sarà poi incluso nelle varie translation unit in modo da garantire che le varie definizioni e dichiarazioni siano identiche in ognuna di queste.
- operators.cpp: si definiscono qui gli operatori inerenti alla struct `BoidState` e `Components` oltre alle definizioni delle funzioni `norm` e `velocity_norm`.
- rulesofswim.cpp: sono qui implementate le varie classi inerenti alle regole e tutte le free functions che vengono utilizzate solo nelle classi delle regole, come ad esempio la funzione che calcola il centro di massa.
- boids.cpp: in questo file sono definite tutte le funzioni che permettono di prevedere quelle che saranno le nuove posizioni e velocità di ogni boid dopo un certo intervallo di tempo, tenendo conto di tutti i limiti che sono stati imposti, oltreché le funzioni che restituiscono medie e relative incertezze dopo il medesimo intervallo di tempo. In sostanza, qui si trovano tutti gli elementi utili a descrivere il comportamento del branco di pesci.
- main.cpp: viene qui implementato il main del programma contenente la parte grafica e tutte le funzioni inerenti a quest'ultima, insieme alla definizione di `simulate`.
- boids.test.cpp: questo file contiene tutti i test realizzati durante la scrittura del programma che hanno permesso di stabilire se il funzionamento di quest'ultimo fosse coerente con le aspettative.

Tutto il codice è stato formattato utilizzando lo strumento clang-format, un formatter, configurato nel file `.clang-format` secondo lo stile standard di Google. Per la configurazione del meta build system cmake è stato creato il file `CMake_Lists.txt` che verrà approfondito nel paragrafo `5.1`. Per quando riguarda i test è stato necessario avvalersi dell'header file `doctest.h` che permette la creazione delle unità di testing.
Durante lo sviluppo del progetto è stato usato Git, che ha permesso di tracciare le versioni dei file sorgenti.

4 Strategia di test

Parallelamente all'implementazione di classi e funzioni, al fine di verificare la correttezza di queste, e soprattutto verificare che i valori ritornati fossero uguali

a quelli attesi, sono stati scritti alcuni test. Il codice è stato testato sia in situazioni che simulano il comportamento normale, che in casi più scomodi, ovvero con valori che avrebbero potuto far fallire il programma. Inizialmente è stato provato il corretto funzionamento degli operatori implementati per le varie classi che sono state definite. È stato poi testato il corretto funzionamento delle regole di nuoto, di conseguenza il meccanismo dei function objects, e analogamente per tutte le free functions e i metodi utilizzati nel programma. Inoltre, sono state testate le eccezioni inserite nel codice utilizzando degli appositi check case.

5 Istruzioni su compilazione ed esecuzione[1]

Il progetto Boids prevede la produzione di due eseguibili finali, il primo contenente i test relativi agli oggetti con cui è stato implementato il programma e il secondo contenente le istruzioni per poter osservare il risultato finale: la simulazione di un branco di pesci. Per questo programma si utilizza il compilatore GCC (g++) che è utilizzabile in ambiente Unix o Unix-like. In fase di sviluppo si è compilato il codice sorgente per i test con le seguenti istruzioni dalla linea di comando: `g++ -Wall -Wextra -fsanitize=address boids.cpp operators.cpp rulesofswim.cpp boids.test.cpp`. In questo modo si sono abilitati i warnings e il sanitizer che segnala problemi legati alla gestione della memoria (per esempio i memory leaks). Per compilare il file main.cpp bisogna includere anche le librerie necessarie alla parte grafica. Per compilare si utilizzano le istruzioni: `g++ -Wall -Wextra -fsanitize=address boids.cpp operators.cpp rulesofswim.cpp main.cpp -lsfml-graphics -lsfml-window -lsfml-system`. Quelli precedentemente presentati sono i comandi minimali utilizzati per compilare in fase di sviluppo e generano un file binario denominato `a.out` ed eseguibile sulla linea di comando con: `./a.out`. Si suggerisce di inserire però alcune varianti utili. In particolare per rinominare l'eseguibile che viene creato al termine della compilazione si può aggiungere come opzione: `-o boidstest` per quanto riguarda i test e `-o main` per quanto riguarda il file main.cpp. Questa opzione è utile per poter avere due file eseguibili contemporaneamente e non dover sovrascrivere il file a.out ogni volta che si vuole compilare una nuova modifica. Infatti, per eseguire è sufficiente scrivere rispettivamente: `./boidstest` e `./main`. Successivamente alla fase di sviluppo è stato compilato il progetto disabilitando gli assert presenti nel codice. Questa opzione è utile per compilare solo il file main.cpp in maniera ottimizzata. Questo è possibile scrivendo dalla linea di comando: `g++ -Wall -Wextra -fsanitize=address -DNDEBUG boids.cpp operators.cpp rulesofswim.cpp main.cpp -lsfml-graphics -lsfml-window -lsfml-system -o mainoptimized`. Per eseguire il binario che è stato generato viene usato: `./mainoptimized`. Un'altra variante possibile per avere ottimizzazioni una volta terminata la fase di sviluppo è utilizzare l'opzione: `g++ -Wall -Wextra -fsanitize=address -O3 -DNDEBUG boids.cpp operators.cpp rulesofswim.cpp main.cpp -lsfml-graphics -lsfml-window`

[1] I comandi sotto illustrati sono pronti all'uso: è sufficiente copiarli e incollarli sul terminale.

`-lsfml-system -o mainfullyoptimized`. Per eseguire è sufficiente scrivere sulla linea di comando: `./mainfullyoptimized`.

5.1 CMake

Dal momento che il progetto Boids prevede più translation units, di conseguenza è composto da più file, si è scelto di utilizzare il meta build system cmake per buildare il programma e rendere la fase di compilazione e linking più semplice. Le opzioni principali specificate nel file CMakeLists.txt volte alla configurazione delle directories di build sono analoghe a quelle sopra utilizzate ovvero: l'abilitazione di warnings e l'utilizzo del sanitizer in modalità di debug (ovvero in fase di sviluppo), mentre sono stati disabilitati gli assert e utilizzate le ottimizzazioni necessarie in modalità release, mantenendo abilitati i warnings. In fase di sviluppo è stata creata la directory build configurando cmake in debug mode, utilizzando il comando: `cmake -S . -B build -DCMAKE_BUILD_TYPE=Debug`. Qui sono stati creati due target: il primo riguardante i test e il secondo contenente il main del progetto. Per effettuare il build in debug mode bisogna eseguire dalla linea di comando: `cmake --build build`. Per eseguire i targets in questa modalità è sufficiente scrivere: `build/boids.t` per quanto concerne i test e `build/boids-sfml` per quanto riguarda il main. Un modo alternativo per eseguire il test è: `cmake --build build --target test`. Terminata la fase di sviluppo è stata creata una seconda directory, denominata build_release, configurata in release mode, utilizzando: `cmake -S . -B build_release -DCMAKE_BUILD_TYPE=Release`. Per la fase di build in questo modo si utilizza: `cmake --build build_release`. Invece per l'esecuzione in questa modalità, ovvero quella ottimizzata, si esegue solo il target riguardante il main con il comando: `build_release/boids-sfml`.

6 Descrizione del formato di input e output

Il programma prevede l'inserimento in input di quattro valori. Il primo che deve essere inserito è il numero di boids, in questo caso pesci, dei quali si vuole calcolare distanza e velocità media con relativa deviazione standard. Il numero da inserire è un intero positivo maggiore di uno. Se questa richiesta non è soddisfatta il programma si interrompe e non è più possibile proseguire. Qualora il numero inserito non sia un intero, a causa delle conversioni implicite, viene presa la parte decimale come parametro di separation. Si suggerisce di inserire un numero cospicuo di boids ma non troppo alto, altrimenti è più difficile osservare il movimento dei pesci a livello grafico. Si consiglia di provare inizialmente con valori come 20 o 30 e provare ad aumentare il numero di boids per osservare come cambia il loro comportamento. In ogni caso è sconsigliato oltrepassare la soglia di circa 100 individui all'interno del campione. Se questo viene fatto il programma non si interrompe ma è peggiore la prestazione grafica. I restanti parametri da inserire sono i valori numerici delle costanti di separazione, allineamento e coesione. I valori di queste variabili vanno inseriti in questo ordine al

fine del corretto funzionamento del programma. Il primo parametro può assumere un qualsiasi valore nei reali ma si consiglia di inserire un numero compreso nell'intervallo $[0, 2[$ per osservare la formazione di branchi di pesci. Il valore della costante di allineamento deve necessariamente essere minore di 1, se ciò non viene rispettato il programma si interrompe e lancia un messaggio di errore. Il terzo e ultimo parametro è utile se assume un valore compreso tra $[0, 0.1[$, altrimenti non si osserva la formazione di branchi. Con valori maggiori infatti la regola di coesione diventa preponderante e i pesci tendono a sovrapporsi. Se ciò accade, vista la formula usata per implementare separation, per cui la repulsione diventa nulla se due boids assumono la stessa posizione, i boids non si separeranno più. Per evitare ciò, il parametro s deve essere due ordini di grandezza maggiore rispetto a c. Secondo le prove avvenute in fase di sviluppo i range migliori per la simulazione sono i seguenti:

- **s**: un numero compreso tra 0.5 e 1.
- **a**: sebbene possa assumere qualunque valore tra 0 e 1, si consiglia un numero più vicino a 1
- **c** è consigliabile un numero al millesimo, che non superi 0.01

In fase di sviluppo sono stati usati i seguenti valori: $n = 30$, $s = 0.5$, $a = 0.8$, $c = 0.003$.
Per quanto riguarda l'output, il programma restituisce il valore della distanza media tra i boids con relativa deviazione standard e il valore della velocità media sempre con relativa deviazione standard. Si nota che all'inizio la distanza è di qualche centinaio di pixel e dopo che si è formato il branco raggiunge circa 80. Ovviamente la distanza dipende fortemente dal numero di boids della simulazione e potrebbe subire un aumento dopo che si è formato il branco a causa del comportamento ai bordi. La deviazione standard della distanza media dipende dalle posizioni dei boid che sono generate in maniera pseudocausale, se si formano più branchi di pesci, il suo valore sarà maggiore. La deviazione standard varia sensibilmente anche in base ai parametri inseriti in input. Per quanto riguarda la velocità media si ha tendenzialmente un valore di qualche unità espresso in pixel al secondo (calcolato considerando i parametri precedenti assume valori variabili tra 0.5 e 1.5 circa). In ogni caso il valore massimo della velocità raggiungibile è fissato a 5. Analogamente al caso della deviazione standard associata alla distanza media, anche la deviazione standard associata alla velocità varia in base ai parametri inseriti, come conseguenza presenta valori molto variabili. Infine, si può notare che dopo la chiusura della finestra grafica, viene segnalato un errore di memory leak. Quest'ultimo riguarda SFML, quindi non è dovuto alla mancata correttezza del codice.

7 Interpretazione dei risultati ottenuti

L'implementazione della parte grafica ha permesso di stabilire visivamente se il programma fosse stato implementato in maniera corretta per poter simulare il

comportamento del branco. Utilizzando i valori sopra consigliati si può ben osservare tale fenomeno, quindi come i pesci, inizialmente con velocità e posizioni differenti in quanto generate casualmente, si avvicinino e tendano ad uniformare la loro velocità. Si può inoltre vedere l'influenza che ogni regola ha sul moto mettendo a zero le costanti relative alle altre. Un'ulteriore conferma viene fornita anche dai valori in output: la distanza media tra i vari pesci, infatti, diminuisce notevolmente con lo scorrere del tempo a conferma del progressivo ravvicinarsi di questi ultimi nella loro traiettoria. Si possono tuttavia osservare delle eccezioni rispetto a questa decrescita che sono però naturali. Sono infatti dovute allo spostamento di una parte del branco da una parte all'altra della finestra grafica. In altri casi è possibile notare la formazione di un branco che raggiunge una posizione di "equilibrio", ovvero resta unito ma senza muoversi nell'area di simulazione.