Simulazione di uno stormo inseguito da un predatore

Mila Casali, Davide Fuda, Francesco Fonseca

Introduzione

Gli stormi di volatili sono in grado di spostarsi in modo coordinato nella stessa direzione, senza la presenza di un capo o di una figura di autorità che li guidi o organizzi. Dunque il nostro scopo è riuscire a simulare uno stormo che si comporti in modo ordinato e coeso partendo dall'implementazione del moto di un singolo volatile. Il problema di riuscire ad emulare tale comportamento venne affrontato da Craig Reynolds nel 1986 [1], che propose il modello al quale ci rifacciamo in questo progetto. Tale modello prende il nome dal software ideato da Reynolds che venne battezzato "boids".

1. Analisi del problema

La simulazione avviene in uno spazio tridimensionale toroidale e si basa sul fatto che ogni singolo ente segua tre regole fondamentali ideate dallo stesso Reynolds [2]: la separazione, che fa in modo che ogni uccello eviti di avvicinarsi eccessivamente ai vicini, l'allineamento, che spinge ogni volatile ad adattare la propria velocità a quella dei compagni e infine la coesione, che mantiene l'unità dello stormo facendo in modo che ogni uccello vada verso il centro dello stesso. In aggiunta a queste tre regole basilari abbiamo implementato la simulazione in modo tale che tenga conto della presenza di un predatore dal quale gli uccelli fuggono e in modo tale che il moto dello stormo sia influenzato dalla presenza del vento che spira in una specifica direzione, inoltre lo stormo tende a mantenersi a una determinata quota. Ognuna di queste indicazioni è coadiuvata da fattori limitanti sulla velocità e posizione dei singoli volatili in modo tale da avere un comportamento realistico, tali fattori potranno essere inseriti dall'utente o generati casualmente, essi includono: l'ampiezza dello spazio di simulazione, l'apertura alare, la velocità massima, la distanza minima, i fattori di allineamento, coesione, separazione e paura del predatore, la velocità del vento e la sua direzione, la velocità e il raggio di attacco del predatore. Inoltre saranno sempre a discrezione dell'utente la presenza del vento e dello spazio toroidale.

2. Progettazione dell'Algoritmo

Il problema è stato affrontato implementando un algoritmo che tenga conto dei dati relativi a uno stormo di boids e allo stesso tempo li utilizzi per determinare il comportamento di ogni singolo boid in ottemperanza alle regole sopracitate.

2.1 Scelte di Progetto

Come ausilio allo sviluppo del programma abbiamo utilizzato la repository gitHub riportata in appendice

2. Calcolo delle forze: per ogni boid, in base alle posizioni e alle velocità dei boids vicini, calcolo delle forze di separazione, allineamento e coesione.
3. Influenza ambientale: applicazione delle forze derivanti dal vento e dalla presenza del predatore.
4. Aggiornamento del movimento: aggiornamento delle velocità e delle posizioni dei boid.
5. Gestione dei bordi: applicazione delle regole toroidali o di rimbalzo ai confini dello spazio di simulazione.
6. Interazione con il predatore: il predatore cerca di catturare i boid, che a loro volta tentano di fuggire.
7. Estrazione delle statistiche: ad ogni unità di tempo (o multipli) vengono estratte una serie di misure che riassumono il comportamento dello stormo.

3. Implementazione

Il progetto è organizzato modularmente, suddiviso in un file sorgente e una serie di file di intestazione che contengono le definizioni delle classi e le funzioni ausiliarie. Per quanto riguarda le classi ausiliarie si è scelto di includere sia la dichiarazione sia l'implementazione nel file header per semplificare la gestione del codice. Il file principale, `**boids.cpp**`, funge da entry point per l'intero programma e coordina l'interazione tra le varie classi e funzioni definite negli altri file.
Il file principale deve essere compilato con il comando:

```
g++ -o boids boids.cpp -Wall -Wextra -Wpedantic -Wconversion
-Wsign-conversion -Wshadow -Wimplicit-fallthrough -Wextra-semi
-Wold-style-cast -D_GLIBCXX_ASSERTIONS -fsanitize=address -lsfml-graphics
-lsfml-window -lsfml-system
```

allo scopo di richiamare correttamente tutte la librerie grafiche.
Oltre al file principale, ci sono sei file header che contengono le classi e le funzioni fondamentali per il funzionamento del programma. Questi file sono:

1. **vec3.hpp**: Contiene la definizione della classe `Vec3`, che rappresenta i vettori tridimensionali utilizzati per gestire le posizioni e le velocità dei boids nello spazio. Questa classe include operazioni vettoriali essenziali per il movimento e l'interazione dei boids.

2. **boid.hpp**: Contiene la definizione della classe `Boid`, l'unità di base della simulazione, che rappresenta i singoli volatili dello stormo. Ogni boid ha come proprietà una posizione e una velocità. Sono inoltre presenti metodi volti all'aggiornamento delle velocità e delle posizioni dei boids.

6. **functions.hpp**: Include funzioni ausiliarie e utility che supportano la logica principale del programma. Queste funzioni possono includere calcoli matematici o altre operazioni necessarie per il corretto funzionamento della simulazione.

Ogni file `hpp` è progettato per concentrarsi su una specifica parte del programma, ma non è completamente autonomo. I file di intestazione sono strettamente interconnessi, con molte dipendenze reciproche. Ad esempio, la classe `Swarm` dipende dalla classe `Boid`, e il predatore definito in `predator.hpp` è una sottoclasse di Boid e riprende dei metodi di 'Swarm'. Inoltre la classe 'Vec3', definita nel header 'vec3.hpp' è inclusa in ogni file. Questa interdipendenza è necessaria per creare una simulazione coerente e funzionale, in cui ogni parte del sistema interagisce con le altre. La suddivisione del codice in questi file header migliora l'organizzazione e facilita la manutenzione, ma richiede una chiara comprensione delle relazioni tra le diverse componenti del codice. Di seguito una descrizione dettagliata delle 4 classi principali operanti nel codice:

3.1 Descrizione dettagliata delle classi

3.1.1 La classe "Vec3"

La classe `Vec3` rappresenta un vettore tridimensionale e costituisce una componente fondamentale del progetto, fornendo una struttura dati essenziale per la gestione delle operazioni geometriche nello spazio 3D. Questa classe implementa vari metodi per operazioni vettoriali di base, come la somma, la sottrazione, il prodotto scalare e vettoriale, la normalizzazione e il calcolo della norma del vettore. Inoltre, la classe include alcune operazioni necessarie al calcolo delle distanze in uno spazio toroidale. Ogni operazione è implementata tramite operatori. Ad esempio, la somma di due vettori `Vec3` può essere eseguita utilizzando l'operatore `+`, mentre la moltiplicazione per uno scalare è realizzata tramite l'operatore `*`. La classe include anche un metodo per accedere agli elementi del vettore tramite l'overload dell'operatore `[]`.

3.1.2 La classe "Boid"

La classe `Boid` rappresenta l'unità base del sistema di simulazione, modellando un singolo volatile che si muove nello spazio tridimensionale. Ogni boid è caratterizzato da una posizione (`position`) e una velocità (`velocity`), entrambe rappresentate da un vettore di classe `Vec3`. La classe fornisce metodi per ottenere e modificare questi attributi, consentendo di aggiornare la posizione e la velocità del boid in base alle interazioni con altri boids e con l'ambiente circostante. Il metodo `update_boid_velocity` aggiorna la velocità del boid tramite un'accelerazione e successivamente il metodo 'update_boid' ne aggiorna la posizione considerando la nuova velocità. Se la velocità supera un limite massimo, viene normalizzata e ridimensionata per rientrare in questo limite.

3.1.3 La classe "Predator"

La classe `Swarm` è il fulcro del progetto, essa rappresenta uno stormo di boid e gestisce la loro dinamica di movimento, regolata da vari parametri come la velocità massima, la distanza minima tra i boid, e fattori di separazione, coesione, allineamento e paura.

La posizione iniziale dei boid viene generata casualmente all'interno di un'area di schermo definita dal vettore `screen`. Per ogni boid, vengono generate tre coordinate casuali (x, y, z) che rappresentano la posizione iniziale del boid nello spazio tridimensionale definito dal vettore screen. La velocità iniziale del boid viene generata creando tre componenti (vx, vy, vz) calcolate come numeri casuali compresi tra -max_speed/√3 e max_speed/√3. Ciò garantisce che la velocità iniziale sia casuale ma limitata da 'max_speed'.
I boid tendono inoltre a mantenersi a una quota costante rispetto all'asse `z`, definita come i ⅔ dello schermo grazie ad una funzione 'keep_height'. Se un boid si sta allontanando dall'altezza desiderata, la funzione calcola una correzione basata sulla differenza tra l'altezza attuale e quella preferita, oltre all'attuale velocità del boid, modulata da un fattore di divisione ('division_factor') per evitare cambiamenti troppo bruschi.

La funzione `avoid_predator` permette ai boid di evitare un eventuale predatore: calcola un vettore di fuga basato sulla distanza tra il boid e il predatore. Se la distanza è zero, il boid continua a muoversi nella sua direzione attuale alla massima velocità. Se il predatore è entro un certo raggio (`sight_distance`), il boid si allontana nella direzione opposta al predatore, con un'intensità proporzionale alla vicinanza del pericolo. Il vettore di fuga risultante viene poi modulato da un fattore di paura (`fear_factor`). Quando un boid viene preso dal predatore, ossia il wingspan del predatore e del boid si sovrappongono, quest'ultimo viene eliminato dallo stormo.
La classe contiene una variabile cooldown che rappresenta il tempo passato dall'ultima uccisione.

Oltre alle regole di movimento, la classe gestisce anche altri fattori, come la presenza di vento (`wind`), che influenza il movimento dei boid. Il metodo principale `update_swarm` aggiorna la posizione e velocità di tutti i boid tenendo conto delle regole di movimento, del predatore, e delle condizioni ambientali. Se un boid si trova all'interno del raggio di azione del predatore (`wingspan`), viene rimosso dallo stormo, simulando la sua cattura. Per i boid non catturati, vengono calcolate nuove velocità basate sulle regole di comportamento: viene richiamato due volte il metodo di 'Boid', 'update_boid_velociy' passando come 'delta_v' prima la somma delle correzioni alla velocità dovute alle 4 regole, poi quella relativa alla fuga dal predatore, per aumentarne il peso. Viene poi richiamato il metodo 'update_boid' passando come 'delta_v' la correzione dovuta al vento, questo metodo oltre alla velocità modifica la posizione del boid tenendo in considerazione la velocità precedentemente modificata e la correzione dovuta al vento.

quelli che si trovano all'interno di una certa distanza (`sight_distance`) rispetto al boid attualmente considerato (`b`). Se il boid in esame non è troppo lontano (cioè se è entro la `sight_distance`), la sua posizione viene aggiunta al centro di massa percepito, e il conteggio (`count`) dei boid vicini viene incrementato. Se non ci sono boid vicini (ossia, `count` è zero), la funzione restituisce un vettore nullo (`vec3(0, 0, 0)`), indicando che non c'è alcun movimento verso il centro di massa. Se ci sono boid vicini, il centro di massa percepito viene calcolato facendo la media delle posizioni di questi boid vicini, e il boid corrente (`b`) viene spinto leggermente verso questo centro di massa percepito. La forza di questa spinta è scalata da un fattore di coesione (`cohesion_factor`).
La funzione gestisce anche il caso in cui il comportamento toroidale sia abilitato, in questo caso la distanza tra i boids è calcolata attraverso la funzione 'toroidal_distance', definita in 'functions.hpp', e la velocità è indirizzata di conseguenza. In entrambi i casi, la funzione restituisce un vettore che rappresenta la direzione e la forza con cui il boid corrente dovrebbe muoversi per avvicinarsi al centro di massa percepito dello stormo.

rule 2. La funzione `rule2` implementa la regola di "separazione" per i boid, che ha l'obiettivo di mantenere una distanza minima tra ciascun boid e i suoi vicini, evitando collisioni. Questa regola contribuisce a distribuire i boid nello spazio, impedendo che si aggreghino eccessivamente. La funzione scorre tutti i boid nello stormo. Se un boid è troppo vicino al boid in esame (`b`) e si trova a una distanza inferiore o uguale alla distanza minima (`min_distance`), viene calcolato un vettore correttivo. Questo vettore è diretto lontano dal boid vicino ed è proporzionale alla distanza tra i due boid, normalizzata per assicurare che il vettore correttivo sia più grande quanto più i boid sono vicini.
Se il comportamento toroidale è attivo, la funzione calcola la distanza toroidale tra i boid, tenendo conto della possibilità che i boid si "teletrasportino" attraverso i bordi dello schermo. Infine, la funzione restituisce il vettore correttivo `c`, scalato da un fattore di separazione (`separation_factor`) per modulare l'intensità della spinta.
La funzione evita anche la sovrapposizione dei boids, modificandone direttamente la posizione nel caso siano a distanza inferiore a wingspan.

rule 3. La funzione `rule3` implementa la regola di "allineamento" per i boid, che permette a ciascun boid di tendere a muoversi nella stessa direzione dei suoi vicini. Questa regola contribuisce a creare un comportamento coordinato all'interno dello stormo, con i boid che cercano di allineare la loro velocità con quella dei boid circostanti. La funzione esamina tutti i boid nello stormo. Se un boid è vicino a `b` e si trova entro una certa distanza di visuale (`sight_distance`), la velocità del boid vicino viene aggiunta al vettore `pv`, e il contatore `count`

prevede un comportamento di rimbalzo. Se il boid supera i limiti verticali, la sua velocità viene invertita e attenuata per simularne l'impatto, impedendogli di uscire oltre l'altezza massima o sotto il livello minimo. Infine, la nuova posizione e velocità del boid vengono aggiornate.

bounce. La funzione `bounce` gestisce il rimbalzo di un boid quando colpisce i confini dell'area di simulazione. Se un boid raggiunge i bordi dello schermo in una delle due dimensioni orizzontali (x, y o z) e la sua velocità lo porterebbe a uscire dai limiti, la funzione inverte la componente corrispondente della velocità, riducendola del 10% per simulare una perdita di energia al momento dell'impatto. Inoltre, la posizione del boid viene corretta per assicurarsi che rimanga all'interno dei confini, evitando che si sposti oltre i limiti. La funzione aggiorna quindi la velocità e la posizione del boid per riflettere queste modifiche.

3.2 Le funzioni ausiliarie

Nel file 'functions.hpp' sono definite diverse funzioni che si occupano di tre aspetti fondamentali dell'implementazione della simulazione: l'inizializzazione dei parametri dello stormo e l'interazione con l'utente

3.2.1 Inizializzazione dei parametri

La funzione `initialize_parameters` riveste un ruolo cruciale nella configurazione iniziale della simulazione dei boid. Questa funzione consente di stabilire i parametri di base che determinano il comportamento dello stormo e le condizioni ambientali in cui esso si muove. In particolare, l'utente ha la possibilità di abilitare o disabilitare la presenza del vento e di uno spazio toroidale, il che influenza significativamente il comportamento globale del sistema. La funzione supporta sia un'inizializzazione automatica dei parametri, con valori predefiniti, sia un'inizializzazione manuale, dove l'utente può specificare parametri all'interno di range predefiniti come la dimensione dello stormo, la velocità massima dei boid, la distanza minima tra essi, e fattori di separazione, coesione, e allineamento. Inoltre, è possibile impostare parametri relativi al predatore, come la velocità e il raggio d'attacco.
La funzione offre anche la possibilità di generare i parametri in modo casuale richiamando 'casual_parameters' che genera per essi valori casuali all'interno dei range di validità. Questa flessibilità consente di esplorare una vasta gamma di scenari simulativi, rendendo la funzione fondamentale per esperimenti e analisi del comportamento emergente dei boid.

3.2.2 Aggiornamento della simulazione

La funzione `update_simulation` rappresenta il cuore dinamico della simulazione. Essa si occupa di aggiornare le posizioni e le velocità sia del predatore che dei boid in base alle regole comportamentali implementate e agli eventuali input ambientali. Inoltre, questa funzione esegue una stampa periodica delle

3.3.1 Creazione e Gestione delle Finestre
La funzione `draw_windows` è responsabile della creazione delle finestre di visualizzazione. Utilizzando la risoluzione del desktop dell'utente (‘getDesktopMode’), questa funzione calcola le dimensioni appropriate per le finestre, tenendo conto delle dimensioni dello schermo e di eventuali elementi dell'interfaccia, come la barra delle applicazioni. Due finestre vengono create per visualizzare il volo dei boid da diverse prospettive: la proiezione XY e la proiezione XZ. Le finestre sono posizionate in modo tale da sfruttare al meglio lo spazio disponibile sullo schermo, facilitando così l'osservazione simultanea delle diverse proiezioni.

3.3.2 Disegno dei Boid e del Predatore
La funzione `draw_boids_on_plane` si occupa del disegno effettivo dei boid e del predatore all'interno delle finestre create. I boid vengono rappresentati come cerchi bianchi, con un raggio corrispondente all'apertura alare, mentre il predatore, più grande, è disegnato come un cerchio rosso per distinguerlo chiaramente dal resto dello stormo. La posizione dei boid e del predatore all'interno delle finestre è calcolata in base alle dimensioni dello schermo e alle coordinate relative del piano selezionato.

3.3.3 Gestione degli Eventi
La funzione `handle_events` gestisce l'interazione tra l'utente e le finestre della simulazione. Utilizzando un loop di eventi, questa funzione verifica se l'utente ha chiuso una delle finestre e, in tal caso, chiude la finestra corrispondente. Questo permette una gestione ordinata della simulazione e garantisce che il programma si chiuda in modo controllato quando l'utente decide di interrompere la simulazione.

3.5 Il file main (boids.cpp) e l’interazione con l’utente
Nel file principale ‘boids.cpp’, l'interazione con l'utente è gestita attraverso una serie di richieste di input definite nei file ‘functions.hpp’ e ‘swarm.hpp’. Questi file contengono messaggi predefiniti che guidano l'utente nella configurazione della simulazione, chiedendo se preferisce includere effetti come il vento o lo spazio toroidale, o se desidera impostare manualmente i parametri. Una volta raccolte le preferenze dell'utente, boids.cpp utilizza le classi e i metodi della libreria SFML per creare e gestire tre finestre che visualizzano le proiezioni ortogonali dello spazio di simulazione. Queste finestre vengono aggiornate in tempo reale tramite il metodo ‘update_simulation’, che incrementa un contatore intero ‘t’, permettendo così il progresso continuo della simulazione.

4. Testing del Programma
Per verificare il corretto funzionamento del programma sviluppato, sono stati eseguiti diversi test

2. Operatori di uguaglianza e disuguaglianza: I test hanno confermato che gli operatori `==` e `!=` funzionano correttamente.
   - Input: `*Vec3 v1(1, 2, 3); Vec3 v2(1, 2, 3); Vec3 v3(4, 5, 6);*`
   - Output atteso: `*v1 == v2; v1 != v3;*`

3. Operazioni di somma e sottrazione: Le operazioni di somma, sottrazione e assegnazione sono state testate per verificare che i risultati siano corretti.
   - Input: `*Vec3 v1(1, 2, 3); Vec3 v2(4, 5, 6); Vec3 v3 = v1 + v2;*`
   - Output atteso: `*v3(5, 7, 9);*`

4. Moltiplicazione e divisione scalare: Sono state testate la moltiplicazione e la divisione di un vettore per uno scalare, verificando la correttezza dei risultati.
   - Input: `*Vec3 v1(1, 2, 3); Vec3 v2 = v1 * 2;*`
   - Output atteso: `*v2(2, 4, 6);*`

5. Norma e normalizzazione: La funzione `norm()` è stata testata per calcolare correttamente la lunghezza del vettore, mentre `normalize()` è stata verificata per produrre un vettore unitario.
   - Input: `*Vec3 v1(3, 4, 0);*`
   - Output atteso: `*norm = 5; normalize = Vec3(0.6, 0.8, 0.0);*`

6. Prodotti scalare e vettoriale: Sono stati eseguiti test per verificare i risultati dei prodotti scalare (`dot`) e vettoriale (`cross`).
   - Input: `*Vec3 v1(1, 0, 0); Vec3 v2(0, 1, 0);*`
   - Output atteso: `*dot = 0; cross = Vec3(0, 0, 1);*`

4.2 Test della Classe `Boid`

1. Costruttore e accessori: Il costruttore di default e parametrizzato della classe `Boid` è stato testato per verificare che inizializzi correttamente le posizioni e le velocità dei boid.
   - Input: `*Boid b1; Boid b2(Vec3(1, 2, 3), Vec3(4, 5, 6));*`
   - Output atteso: `*b1.get_position() == Vec3(0, 0, 0); b2.get_velocity() == Vec3(4, 5, 6);*`

2. Operazioni sui boid: Sono state verificate le funzioni di aggiornamento della posizione e della velocità dei boid, compreso il rispetto della velocità massima.
   - Input: `*b.update_boid_velocity(delta_v, max_speed);*`
   - Output atteso: `*b.get_velocity().norm() <= max_speed;*`

2. Media e deviazione standard: Le funzioni `mean()` e `dev_std()` sono state testate con vettori di valori per verificare che i risultati siano corretti.
  - Input: `*std::vector<double> values = {1.0, 2.0, 3.0, 4.0, 5.0};*`
  - Output atteso: `*mean = 3.0; dev_std = std::sqrt(2.0);*`

4.4 Test della Classe `Swarm`

1. Inizializzazione dello stormo: Sono stati eseguiti test sull'inizializzazione della classe `Swarm` per verificare che i parametri predefiniti e personalizzati siano correttamente gestiti.
  - Input: `Swarm swarm;`
  - Output atteso: `*swarm.get_size() == 100;*`

2. Interazione tra boid nello stormo: È stato verificato che l'aggiornamento dello stormo con la presenza di un predatore modifichi correttamente il numero e la disposizione dei boid.
  - Input: `swarm.update_swarm(predator);`
  - Output atteso: `*swarm.get_size() <= 10;*`

3. Comportamento ai confini: Sono stati testati i comportamenti di confine per garantire che i boid rispettino i limiti dello spazio o si avvolgano correttamente in modalità toroide.
  - Input: `b.set_position(Vec3(-1, 50, 50)); swarm.border(b);`
  - Output atteso: `*b.get_position().x == 0;*`

4.5 Test della Classe `Predator`

1. Costruttore e comportamento: È stato verificato che il predatore sia inizializzato correttamente e che le sue funzioni di aggiornamento influenzino lo stormo come previsto.
  - Input: `Predator predator;`
  - Output atteso: `*predator.get_position() == Vec3(0, 0, 0);*`

2. Aggiornamento del predatore: I test hanno confermato che il predatore aggiorna la sua velocità e posizione per inseguire i boid nello stormo.
  - Input: `predator.update_predator(swarm);`
  - Output atteso: `*predator.get_velocity().normalize() == Vec3(1, 0, 0);*`

4.6 Simulazione al variare dei parametri

In questa sezione presentiamo alcuni risultati ottenuti variando i parametri della simulazione

4.6.1 Test 1: Simulazione senza vento e con i bordi

Input:

- Spazio toroidale: Disattivato
- Numero di boid: 120
-Velocità massima: 75
- Parametri di separazione, allineamento, coesione e paura: 82%, 86%, 67%, 98%
- Vento: Assente

Output:

Stormo che pian piano diventa coeso velocemente e poi si muove lentamente nello spazio tridimensionale senza disgregarsi. Nel momento in cui il predatore attacca lo stormo scappa.

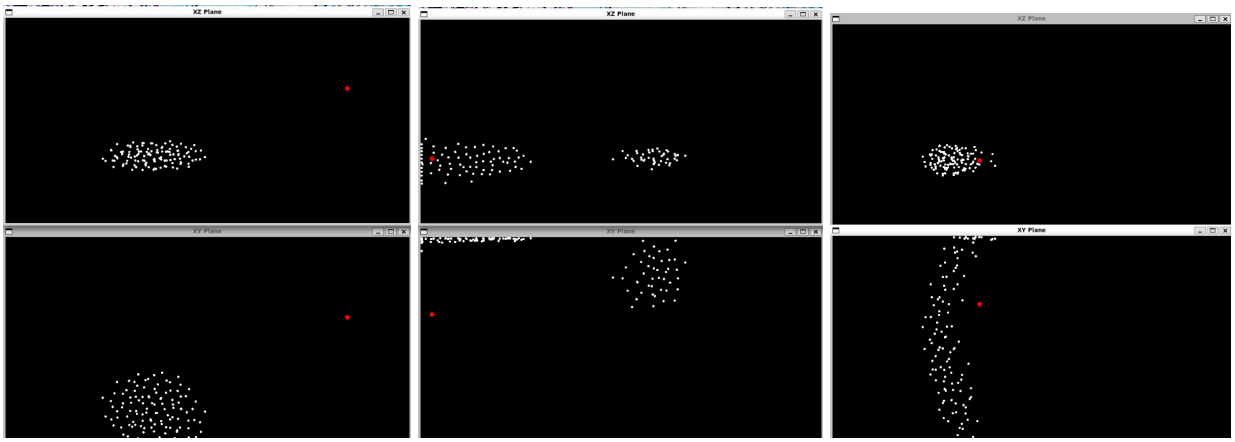


Figura 1: Screenshot della simulazione ai tempi: 4000, 6700, 8700.

4.6.2 Test 2: Simulazione con vento e spazio toroidale

Input:
- Spazio toroidale: Attivato
- Numero di boid: 70
-Velocità massima: 115
- Parametri di separazione, allineamento e coesione: 37%, 52%, 6%, 5%
- Vento: Presente

Output:
- Stormo che si muove in un ampio spazio con traiettoria influenzata dal vento, con alcuni boid sono catturati ma lo stormo reagisce lentamente alla presenza del predatore.

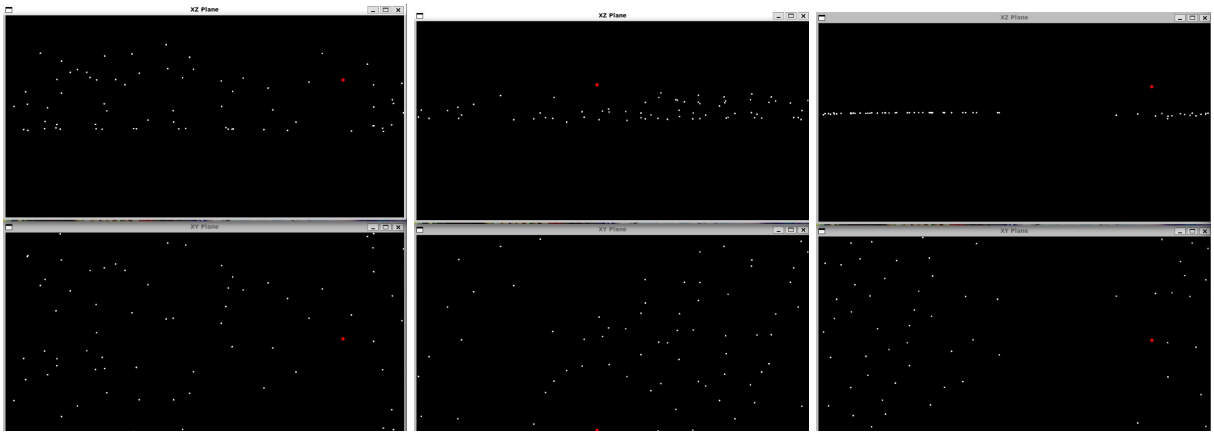


Figura 2: Screenshot della simulazione ai tempi: 700, 4100, 10400.

4.6.3 Test 3: Simulazione con vento e con bordi

Input:
- Spazio toroidale: Disattivato
- Numero di boid: 150
-Velocità massima: 164
- Parametri di separazione, allineamento, coesione e paura: 73%, 50%, 2%, 100%
- Vento: Presente

Output:
- Boid che arrivano ad una coesione lentamente, influenzati dal vento tendono verso un bordo ma non vi rimangono attaccati dal momento che presentano una forte reazione ogni qualvolta il predatore attacca lo stormo. Alcuni boid sono catturati e rimossi dallo stormo.

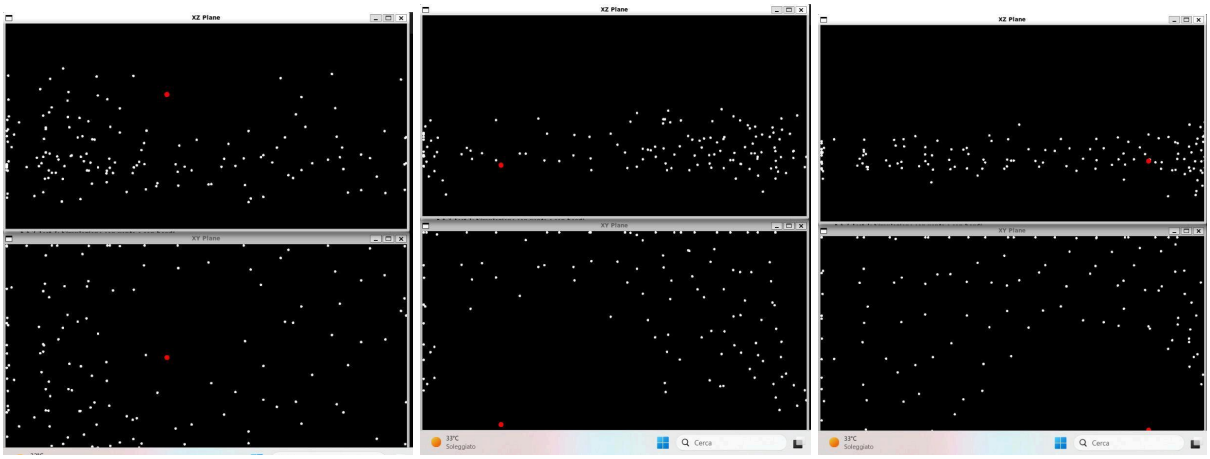


Figura 3: Screenshot della simulazione ai tempi 900, 2600, 4100

4.6.4 Test 4: Simulazione senza vento e con spazio toroidale

Input:
- Spazio toroidale: Attivato
- Numero di boid: 50
- Velocità massima: 179
- Parametri di separazione, allineamento, coesione e paura: 76%, 20%, 66%, 44%
-Vento: Assente

Output:
Boids molto statici, con centro di massa pressoché fisso posizionato “dietro” allo schermo nello spazio toroidale, stormo coeso ma con distanze medie alte e picchi di velocità al momento degli attacchi.

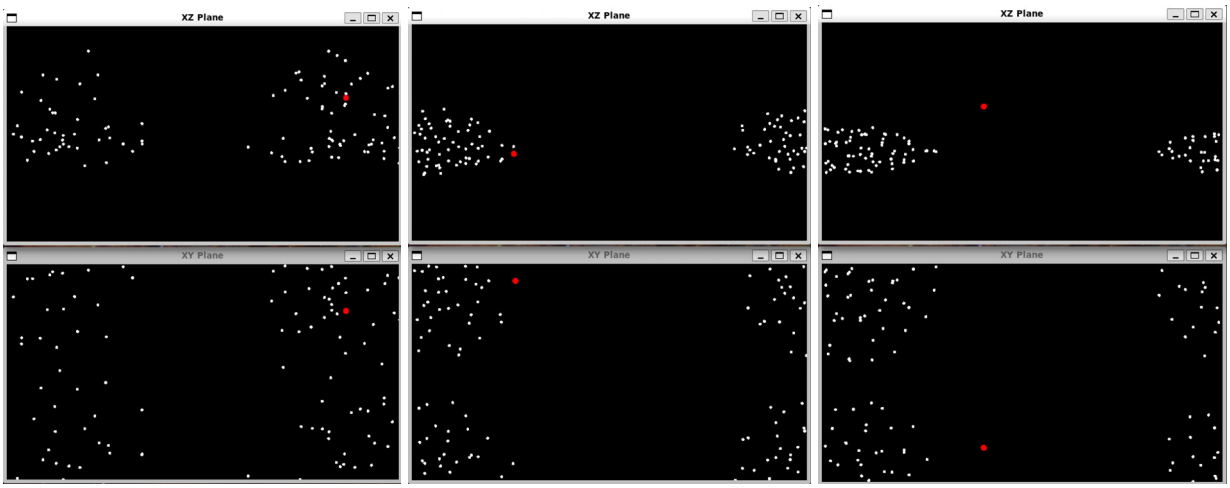
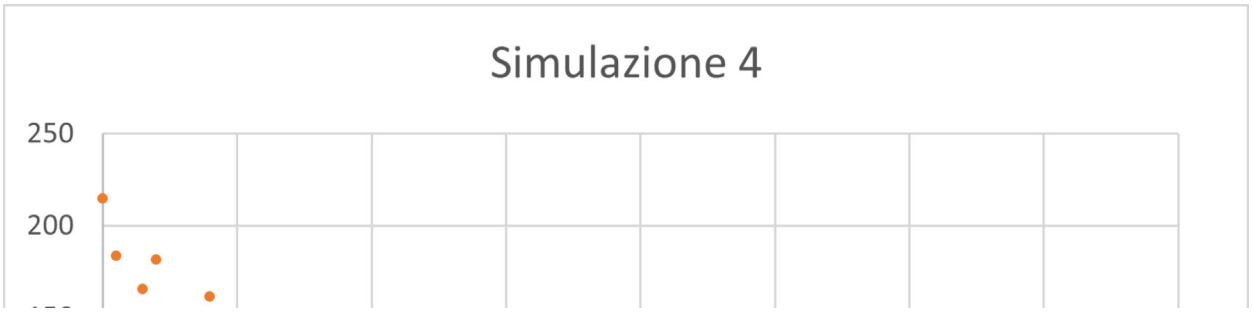


Figura 4: Screenshot della simulazione ai tempi 300, 2100, 5300



Conclusioni

La simulazione si svolge fluidamente con ogni combinazione di variabili provata e l'organizzazione basata sugli header files rende estremamente semplice la manutenzione. L'ampio uso di iteratori non appesantisce significativamente il programma fin quando lo stormo non supera i 150 boid. Lo stormo si comporta come previsto aggregandosi dopo poche iterazioni della funzione *'update_simulation',* sia in presenza di vento che non, e in entrambe le possibili configurazioni dello spazio, toroidale e non, inoltre è chiara l'influenza del predatore sul resto dello stormo.

Bibliografia

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

[2] https://vergenet.net/~conrad/boids/pseudocode.html

Appendice

[3] Repository github contenente il progetto:
https://github.com/Ciccio-Fonsy/Boids