Relazione sul progetto di simulazione del comportamento di stormi

Elisa Barilari, Alida Castagnoli, Ilaria Core

Gennaio 2025

Indice

1 Regole di volo

Il programma realizzato simula il comportamento di uno stormo in uno spazio bidimensionale attraverso oggetti detti *boid*, che seguono precise regole di volo che li portano a formare un gruppo coeso e ad allineare le proprie velocità. La simulazione permette di osservare anche il comportamento dello stormo in presenza di predatori, oggetti simili ai boid che seguono regole di volo diverse. Per tutti gli uccelli le regole di volo sono determinate dal comportamento degli uccelli vicini, a prescindere dal fatto che siano boid o predatori.

Tale vicinanza è definita da

$$\left|\vec{x}_{b_i} - \vec{x}_{b_j}\right| < d \tag{1}$$

dove $\vec{x}_{b_i}$ indica la posizione dell'uccello target, preso come riferimento e $\vec{x}_{b_j}$ quella dei suoi vicini, mentre d identifica il raggio entro il quale è possibile trovare dei vicini. Al fine di rendere la simulazione più realistica, è stato introdotto un angolo di vista, pertanto la regione dello spazio in cui si trovano gli uccelli effettivamente vicini al target, risulta essere un settore circolare di raggio d che sottende un angolo pari all'angolo di vista. Il parametro d e l'angolo di vista sono differenti a seconda che il target sia un boid oppure un predatore.

Per simulare l'andamento dello stormo è necessario aggiornare periodicamente posizione e velocità dei singoli uccelli b_i. La posizione viene aggiornata attraverso la formula:

$$\vec{x}_f = \vec{x}_i + \vec{v}_f \cdot \Delta t \tag{2}$$

Il primo contributo alla nuova velocità è determinato dalla regola di bordo, che impedisce agli uccelli dello stormo di uscire al di fuori della schermata della simulazione. Tale regola agisce quando un uccello si trova in prossimità del bordo della finestra, entro un margine fissato, e somma alla velocità un contributo al fine di indirizzare l'uccello verso bordo opposto.

I restanti contributi alla velocità $\vec{v}_{b_i}$ aggiornata si diversificano per boid e predatori, in quanto sono soggetti a regole di volo diverse.

Boid

La velocità aggiornata per il singolo boid presenta i seguenti contributi:

$$\vec{v}_f = \vec{v'} + \vec{v}_{sep} + \vec{v}_{all} + \vec{v}_{coh} + \vec{v}_{rep} \tag{3}$$

Con il termine $\vec{v'}$ a secondo membro dell'Eq.3 si intende la velocità del boid corrente, eventualmente corretta con la regola di bordo, mentre con il vettore $\vec{v}_{sep}$ si indica il termine di separazione, che ha lo scopo di evitare la collisione con altri boid,

$$\vec{v}_{sep} = -s \cdot \sum_{j \neq i} (\vec{x}_{b_j} - \vec{x}_{b_i}) \quad \text{se} \quad \left|\vec{x}_{b_i} - \vec{x}_{b_j}\right| < d_s \quad \text{con} \quad d_s < d \tag{4}$$

Con il vettore $\vec{v}_{all}$ si indica il termine di allineamento, che ha lo scopo di far procedere i boid nella stessa direzione e risulta pari a

$$\vec{v}_{all} = a \cdot (\frac{1}{n-1} \sum_{j \neq i} \vec{v}_{b_j} - \vec{v}_{b_i}) \tag{5}$$

Con il vettore $\vec{v}_{coh}$ si indica il termine di coesione, che ha lo scopo di direzionare il boid corrente verso il centro di massa dello stormo e si ricava con la formula

$$\vec{x}_c = \frac{1}{n-1} \sum_{j \neq i} \vec{x}_{b_j} \tag{6}$$

$$\vec{v}_{coh} = c \cdot (\vec{x}_c - \vec{x}_{b_i}) \tag{7}$$

Con il vettore $\vec{v}_{rep}$ si intende un termine di repulsione con lo scopo di allontanare il boid corrente dai predatori. Tale termine si ottiene con la stessa formula che si usa per la separazione (Eq.4) moltiplicata per un coefficiente pari a 6, dove $\vec{x}_{b_j}$ indica la posizione dei soli predatori vicini.

Predatori

La velocità aggiornata per un predatore, la velocità aggiornata $\vec{v}_{b_i}$ si ricava dall'espressione:

$$\vec{v}_f = \vec{v'} + \vec{v}_{sep} + \vec{v}_{cha} \tag{8}$$

Sono presenti tre contributi: il primo $\vec{v'}$ identifica la velocità del predatore corrente, eventualmente corretta con la regola del bordo; il secondo $\vec{v}_{sep}$ è valutato in modo equivalente all'analogo contributo per un boid e ha lo scopo di allontanare il predatore corrente da altri predatori, per evitarne la collisione; il terzo $\vec{v}_{cha}$ ha lo scopo di favorire l'inseguimento del predatore del centro di massa dei boid vicini, la formula è analoga all'Eq.4 moltiplicata per un fattore (-2), in quanto si vuole avvicinare il predatore.

Inoltre, se necessario, sulle velocità aggiornate degli uccelli sono applicate delle regole di volo che servono a rimodulare la velocità, aumentandola se si arriva a un modulo di velocità minimo, diminuendola se si arriva a un modulo di velocità massimo. Tali valori di minimo e massimo sono differenti per boid e predatori.

2 Struttura del programma

Il programma si compone di cinque file di intestazione, contenuti nella cartella `/include`, le cui implementazioni sono riportate in altrettanti file sorgente, contenuti all'interno della cartella `/src` insieme al main file e al file contenente i test.

Le classi e le funzioni implementate nel progetto sono ripartite tra i seguenti namespace: `graphic_par`, `triangles`, `bird`, `point`, `statistics` e `flock`.

I blocchi fondamentali presenti nel programma sono:

- **point.hpp/.cpp**: Contengono rispettivamente la dichiarazione e la definizione della classe `Point`, racchiusa all'interno del namespace `point`. Questa presenta due attributi privati `x_` e `y_` che rappresentano le coordinate di un vettore in due dimensioni. Gli oggetti `Point` possono essere istanziati mediante l'ausilio del costruttore di default, che inizializza a zero gli attributi, e del costruttore parametrico. Inoltre, grazie a degli operatori opportunamente definiti, possono essere sommati o sottratti fra di loro e moltiplicati o divisi per un *double*; si comportano rispetto a queste operazioni in modo analogo ai vettori. É stato inoltre implementato l'`operatore==` per confrontare due oggetti di tipo `Point`.

  La classe `Point` contiene poi un metodo `module()` per ottenere il la distanza del vettore dall'origine, un metodo `distance()` per ottenere la distanza da un altro oggetto di tipo `Point` e un metodo `angle()` per ottenere l'angolo tra il vettore corrispondente all'oggetto stesso e il semiasse verticale positivo, rivolto verso l'alto.

- **bird.hpp/.cpp**: Contengono rispettivamente la dichiarazione e la definizione di tre classi - `Bird`, `Boid` e `Predator` - all'interno del namespace `bird`.

- `Bird`: è la classe madre, da cui ereditano pubblicamente le classi `Boid` e `Predator`, le quali identificano rispettivamente gli uccelli dello stormo e i loro predatori. `Bird`

contiene dunque gli attributi condivisi da queste ultime: `position_` e `velocity_`, oggetti di tipo `Point` che rappresentano i vettori posizione e velocità dell'uccello, e `sight_angle_`, un *float* che rappresenta invece l'angolo di vista dell'uccello. `Bird` contiene inoltre due metodi con implementazione: `separation()` per gestire la separazione da altri uccelli, `border()` per gestire il comportamento ai bordi. Sono poi presenti tre metodi dichiarati *virtual*: `friction()` e `boost()`, puramente virtuali che rendono la classe astratta, per assicurarsi che il moto avvenga tra la velocità minima e quella massima stabilite e il distruttore ∼`Bird()` per la corretta deallocazione della memoria nello *heap*.

Le classi figlie ereditano gli attributi privati della madre, i costruttori e il distruttore, `separation()` e `border()`, mentre implementano separatamente i metodi `friction()`, `boost()` eseguendo un *override* dei metodi della madre.

- `Boid`: implementa dei metodi specifici quali `alignment()` e `cohesion()`, che consentono ai boid di formare uno stormo coeso, e `repel()`, che li porta ad allontanarsi dai predatori presenti nel loro range visivo.

- `Predator`: implementa la funzione `chase()`, che accelera i predatori nella direzione dei boid vicini.

- **graphic.hpp/.cpp**: Contengono le informazioni utili per la realizzazione della grafica della simulazione, realizzata mediante SFML. In particolare sono presenti due namespace:

  - Il namespace `graphic_par` racchiude parametri relativi alla simulazione e alla sua rappresentazione grafica, come larghezza e altezza della finestra su cui si visualizza lo stormo e il tempo che regola le grandezze cinematiche. Presentano inoltre la dichiarazione e definizione della funzione `getPositiveInteger()` che permette all'utente di settare tramite input il numero di `Boid` e `Predator` della simulazione.

  - Il namespace `triangles` racchiude i parametri geometrici relativi ai triangoli, quali altezza e larghezza di base, accessibili mediante le funzioni `getHeight()` e `getBaseWidth()`, e posizione relativa dei vertici rispetto al centro. Ad ogni triangolo è associato un uccello della simulazione, pertanto è presente anche una funzione per la generazione dei triangoli a partire dal vettore dello stormo: `generateTriangles()`. Infine, tali triangoli vengono adattati al moto degli uccelli mediante una funzione `rotateTriangle()` che consente di ruotarli lungo la direzione del moto.

- **statistics.hpp/.cpp**: Contengono la dichiarazione della struct `Statistics`, inserita nel namespace `statistics` e la definizione dei due costruttori. La struct presenta come attributi quattro variabili di tipo *double* che rappresentano la media e la deviazione standard rispettivamente della distanza tra coppie boid e del modulo delle velocità, ad un tempo fissato.

- **flock.hpp/.cpp**: Contengono la dichiarazione e la definizione, rispettivamente, della classe `Flock` racchiusa all'interno del namespace `flock`. Sono qui contenute informazioni generali sullo stormo: sono presenti come attributi privati della classe il numero di boid e predatori, i coefficienti di allineamento, coesione e separazione, le distanze entro cui applicare le regole di volo e i moduli delle velocità massime e minime sia per i boid che per i predatori. Tra gli attributi privati figura inoltre `flock_`, un vettore di *shared pointer* a oggetti di tipo `Bird`.

All'interno di questa classe è presente il metodo `generateBirds()`, che genera casualmente le posizioni e le velocità iniziali degli uccelli utilizzando un generatore di numeri casuali (`std::default_random_engine`) configurato con una distribuzione uniforme (`std::uniform_real_distribution<double>`). I valori generati vengono utilizzati per creare oggetti di tipo Boid e Predator, che sono poi organizzati in container distinti. Successivamente, tali oggetti vengono convertiti in `std::shared_ptr<Boid>` e `std::shared_ptr<Predator>` per essere aggiunti al vettore `flock_`.

Al fine di aggiornare le velocità e le posizioni dei singoli uccelli, è implementato il metodo `updateBird()` che prende come argomento il singolo elemento di `flock_`. Nel dettaglio, identifica i vicini del boid o predatore corrente utilizzando i metodi ausiliari: `findNearBoids()` e `findNearPredators()`. Calcola le nuove posizione e velocità sulla base delle regole di volo applicabili e aggiorna la rappresentazione grafica associata all'uccello, utilizzando un array di triangoli (`sf::VertexArray`) del tipo `sf::Triangles`. Il triangolo associato all'uccello viene ruotato per allinearlo alla nuova direzione della velocità tramite la funzione `triangles::rotateTriangles()`.

Il metodo `evolve()` applica iterativamente `updateBird()` a tutti gli elementi del vettore `flock_`, aggiornando così le posizioni e le velocità di ogni componente dello stormo.

Infine il metodo `statistics()` calcola statistiche utili per analizzare il comportamento dello stormo. In particolare: calcola, a tempo fissato, la media della distanza tra ogni coppia di boid e del modulo delle velocità, con le relative deviazioni standard. I risultati vengono restituiti come oggetto di tipo `Statistics`.

- **main.cpp** Contiene la funzione `main()`, all'interno della quale viene inizializzato un oggetto di tipo `Flock` con il numero di boid e di predatori ricevuto in input, mediante la funzione `graphic_par::getPositiveInteger()`; l'attributo privato `flock_` è popolato con oggetti di tipo `Boid` e `Predator`, le quali posizioni e velocità sono generate casualmente mediante il metodo `flock::generateBirds()`. Viene quindi chiamata la funzione `triangles::generateTriangles()` per generare l'array di vertici necessario a rappresentare boid e predatori sullo schermo come triangoli. Il file contiene anche il loop principale, dove vengono aggiornate le posizioni dei componenti dello stormo, tramite la funzione `flock::evolve()` e vengono gestiti gli eventi sulla finestra SFML, come la stampa a schermo dei triangoli e delle statistiche o la chiusura della stessa, con la conseguente interruzione del programma. In particolare, la stampa delle statistiche viene chiamata ogni 15 iterazioni del loop principale.

3 Compilazione, esecuzione e testing

La compilazione del progetto avviene tramite CMake, uno strumento per la generazione di build system che facilita la gestione delle dipendenze del progetto e consente di creare un unico eseguibile finale, rendendo il processo di configurazione e compilazione più flessibile e portabile. Per visualizzare l'interfaccia grafica, è necessario installare la libreria grafica SFML (Simple and Fast Multimedia Library). Questa libreria fornisce strumenti semplici ed efficienti per la gestione della grafica.

Al fine di compilare in *Release mode* eseguire i seguenti comandi:

```
cmake -S ./ -B build/release -DBUILD_TESTING=True -DCMAKE_BUILD_TYPE=Release
```

```
cmake --build build/release
```

Viene così costruito l'eseguibile *Boids* in una nuova cartella `/build/release/`.

Al fine di evitare problemi con il caricamento del font (arial.ttf), che potrebbe dipendere dalla directory da cui viene avviata l'esecuzione, si consiglia di utilizzare il seguente comando:

```
./build/release/Boids
```

Per l'esecuzione dei test:

```
./build/release/Boids.t
```

In seguito si riportano i comandi per una compilazione in *Debug mode*:

```
cmake -S ./ -B build/debug -DBUILD_TESTING=True -DCMAKE_BUILD_TYPE=Debug
```

```
cmake --build build/debug
```

```
./build/debug/Boids
```

```
./build/debug/Boids.t
```

Per disabilitare la compilazione dei test configurando Cmake con la flag:

```
-DBUILD_TESTING=False
```

4 Input e output della simulazione

Al momento dell'esecuzione della simulazione, viene chiesto all'utente di inserire in input da terminale il numero di uccelli e il numero di predatori; una volta impostati, questi numeri non possono essere modificati. Per il corretto funzionamento della simulazione si suggerisce di impostare un numero di boid non superiore a 400 e un numero di predatori non superiore a 5. Gli altri parametri sono fissati. La finestra sarà divisa in due parti: una colonna a sinistra di colore grigio, dove vengono presentate le statistiche (aggiornate ogni 15 iterazioni del loop principale), e un secondo riquadro di colore nero su cui si può vedere la simulazione vera e propria, dove i triangoli blu e rossi rappresentanto rispettivamente gli uccelli dello stormo e i predatori.

5 Interpretazione dei risultati

Osservando la rappresentazione grafica della simulazione, si nota che i boid formano diversi piccoli gruppi che tendono a convergere verso un unico stormo, allineando le proprie velocità. Quando un predatore entra nel range visivo di uno o più boid si ha un chiaro effetto dispersivo e tendono ad aumentare la velocità media dello stormo e la deviazione standard delle velocità, specialmente per stormi più ridotti. Il comportamento dello stormo rispecchia quanto atteso.

6 Test

Il file dei test, denominato **test.cpp**, si trova nella cartella `/src`. I test sono stati implementati utilizzando la libreria `DOCTEST`, che ne consente la stesura direttamente nel codice sorgente. La libreria genera autonomamente un main e permette l'esecuzione dei test quando includendo, all'inizio del file sorgente, la seguente direttiva:

```
#define DOCTEST_CONFIG_IMPLEMENT_WITH_MAIN
```

I test sono stati progettati per verificare la correttezza dell'implementazione, concentrandosi principalmente su casi semplici per assicurare un comportamento atteso delle funzionalità di base.

Anche la componente grafica ha svolto un ruolo fondamentale durante la fase di debugging: in diversi casi, infatti, la visualizzazione grafica ha agevolato l'individuazione di errori, rendendoli più evidenti rispetto all'analisi esclusivamente testuale o logica, che è stata comunque supportata dall'uso di `GDB`.