Simulazione di uno stormo di uccelli

Davide Rizzi - Riccardo Neri - Beatrice Buzzi - Mattia Brunelli

3 Luglio 2025

1 Breve introduzione al tema

Il programma da noi scritto ha come obiettivo la simulazione del comportamento di uno stormo di uccelli, in una maniera simile al software scritto da Craig Reynolds nel 1986. La simulazione si basa sull'applicazione di tre regole fondamentali, cioé le regole di separazione, allineamento e coesione, sui cosiddetti "boids" cioé gli oggetti virtuali che prendono il posto degli uccelli.

1.1 Regola della separazione

La regola della separazione impedisce ai boids di entrare in collisione. Per farlo, è necessario determinare una distanza reciproca minima al di sotto della quale due boids iniziano ad allontanarsi tra di loro. Il contributo della regola di separazione da sommarsi alla velocitò del boid e dato da:

$$\overrightarrow{v_s} = -s \sum_{j \neq i} (\overrightarrow{x_{b_j}} - \overrightarrow{x_{b_i}}) \tag{1}$$

Dove i è il boid preso in considerazione, j sono tutti gli altri boids, s è il fattore di separazione (che è un numero che determina l'intensità della velocità di allontanamento).

1.2 Regola dell'allineamento

La regola dell'allineamento porta i boids ad assumere velocità parallele, in modo da muoversi in maniera uniforme. La regola influisce sulla velocità dei boids come:

$$\overrightarrow{v_a} = a \left(\frac{1}{n-1} \sum_{j \neq i} \overrightarrow{v_{b_j}} - \overrightarrow{v_{b_i}} \right) \tag{2}$$

Dove a è un fattore di allineamento che deve essere tale che $a < 1$ e che determina l'intensità con cui i boids tendono ad allinearsi coi loro vicini.

1.3 Regola della coesione

La regola della coesione induce i boids ad avvicinarsi ai vicini. Per farlo, ogni boid vede sommarsi alla sua velocità un contributo che lo indirizza verso il centro di massa dei vicini. Il centro di massa è dunque calcolato con:

$$\overrightarrow{x_{c_i}} = \frac{1}{N-1} \sum_{j \neq i} \overrightarrow{x_{b_j}} \quad \text{con N numero di boids} \tag{3}$$

Ottenuto il centro di massa, la velocità è ottenuta tramite:

$$\overrightarrow{v_c} = c(\overrightarrow{x_{c_i}} - \overrightarrow{x_{b_i}}) \tag{4}$$

Dove c è un fattore di coesione che determina quanto velocemente i boids si aggregano.

2 Principali scelte implementative

2.1 Implementazione degli oggetti

In quanto l'obiettivo del programma è quello di gestire un numero che può essere anche piuttosto grande di singoli oggetti tutti soggetti alle stesse regole, abbiamo deciso di implementare i *boids* come struct e il $flock$ come classe.
La struct *boid* è composta da 4 valori:
-$x\_position$: posizione su x, di tipo *double*;
-$y\_position$: posizione su y, di tipo *double*;
-v_x: velocità su x, di tipo *double*;
-v_y: velocità su y, di tipo *double*;

La classe $flock$ è inizializzata dai seguenti valori (variabili private):
-N: il numero iniziale di *boid* con cui viene inizializzata la classe, di tipo *int*;
-d: il valore della distanza massima oltre alla quale i *boid* non vengono più considerati vicini, di tipo *double*;
-ds: il valore della distanza minima, sotto la quale inizia ad avere effetto la regola della separazione, di tipo *double*;
-s: il valore del fattore di separazione, di tipo *double*;
-a: il valore del fattore di allineamento, di tipo *double*;
-c: il valore del fattore di coesione, di tipo *double*;

La classe comprende inoltre un vettore che contiene tutti i *boids* appartenenti al $flock$. Essendo un membro privato, l'utente non ci interagisce mai direttamente.
Il valore di N, al contrario degli altri, può cambiare tramite l'aggiunta di ulteriori *boid*. L'aggiunta o formazione di *boid* è possibile tramite le funzioni:
-$AddBoid(boid\&\ a)$: aggiunge un singolo boid preesistente alla classe, fino al valore di N. Se vi sono già N *boids*, il valore di N è incrementato di 1.
-$flock\_formation()$: rimuove tutti i *boid* nella classe, per poi inizializzarne N tramite il metodo $boid\_initialize()$, che li genera con posizione e velocità casuali.

2.2 Implementazione delle regole di volo e aggiornamento delle velocità

Le tre regole di volo già descritte sono state implementate separando ognuna di esse tra asse x e y. Si hanno perciò due metodi distinti per ogni regola, ognuno agente su un asse diverso. I metodi delle tre regole principali non modificano direttamente le velocità dei *boids*, bensì restituiscono il contributo da sommare alla velocità del *boid*.

L'aggiornamento delle velocità è fatto tramite la funzione $external\_effects(boid\&\ a)$, che raccoglie in un *module* (struct appositamente creata per salvare coppie di valori) la somma su x e y degli effetti delle tre regole sul *boid*. La funzione viene reiterata per ogni elemento della classe tramite la funzione $velocities\_update()$, che contiene inoltre un controllo della velocità risultante di ogni *boid* e la modifica in modo che rientri tra un valore massimo e un valore minimo. La posizione dei boids viene invece gestita dalla funzione $position\_update()$.
In fase di simulazione le funzioni sopra citate vengono richiamate ogni 15 ms.

Per quanto riguarda il comportamento ai bordi, abbiamo deciso di implementare una "forza repulsiva" che modifica la velocità dei *boids* che arrivano entro una determinata distanza dagli estremi della finestra, spingendoli verso il centro.

2.3 Implementazione grafica con SFML

Per la grafica, abbiamo utilizzato la libreria SFML. In quanto gli sprite di base ci sembravano poco visivamente stimolanti, abbiamo deciso di creare sia lo sprite per i *boid* che un semplice sfondo in pixel art. Abbiamo inoltre implementato un metodo che permette di generare un nuovo *boid* ogni volta che si clicca sulla finestra grafica.

3 Compilazione del codice

In quanto è stata usata la libreria SFML, è prima necessario installarla sulla propria macchina col semplice comando:

$ $sudo\ apt - get\ install\ libsfml - dev$

Il programma può essere buildato in due modalità: Debug e Release. Per iniziare la build, si esegue il primo comando per creare la cartella build:

$ $cmake\ -S.\ -B\ build\ -G\ "Ninja\ Multi - Config"$

A questo punto, si può buildare la versione di Debug. Iniziamo a fare ciò effettuando i test per questa funzione con i seguenti comandi:

$ $cmake\ --build\ build\ --config\ Debug$
$ $cmake\ --build\ build\ --config\ Debug\ --target\ test$

Per avviare la finestra grafica, e dunque il programma vero e proprio, si usa poi:

$ $./build/Debug/boids$

Se invece si vuole buildare e testare la versione di Release, i comandi necessari sono:
$ $cmake\ --build\ build\ --config\ Release$
$ $cmake\ --build\ build\ --config\ Release\ --target\ test$

Se si vuole eseguire la versione di Release, il comando da inserire è:

$ $./build/Release/boids$

Noterete che una volta lanciato l'eseguibile verranno richiesti alcuni dati da inserire prima di avviare la finestra grafica. Come precedentemente descritto, la classe necessita dei seguenti dati durante l'inizializzazione:
-N: di tipo *int*, il cui valore deve essere > 1;
-d: di tipo *double*, il cui valore deve essere $> 0.$;
-ds: di tipo *double*, il cui valore deve essere $0. < ds < d$;
-s: di tipo *double*, il cui valore deve essere $> 0.$;
-a: di tipo *double*, il cui valore deve essere $0. < a < 1.$;
-c: di tipo *double*, il cui valore deve essere $> 0.$;

Da quel che abbiamo potuto osservare, la scelta dei parametri iniziali può avere una notevole influenza sull'evoluzione della simulazione. Per questo motivo abbiamo pensato di fornire per ogni parametro un range consigliato per una migliore inizializzazione dello stormo:

$10 < N < 100$ Valore consigliato: 30
$10. < d < 15.$ Valore consigliato: 14.
$2. < ds < 4.$ Valore consigliato: 3.
$2. < s < 5.$ Valore consigliato: 1.
$0.2 < a < 0.5$ Valore consigliato: 0.3
$0.05 < c < 0.1$ Valore consigliato: 0.05

4 Interpretazione dei dati

Ogni 5 secondi, durante la simulazione, vengono stampati a schermo una serie di dati finalizzati a descrivere l'evoluzione dello stormo nel tempo. In primis viene fornita la velocità media dei boids, calcolata come la media aritmetica dei moduli dei vettori velocità, e la deviazione standard. Con l'avanzare della simulazione, quando lo stormo si comporta come previsto, è possibile osservare una leggera diminuzione della velocità media ma di maggior rilevanza è la diminuzione del valore della deviazione standard che, con l'uniformarsi delle velocità, tende a diminuire, salvo fluttuazioni casuali.
Il secondo dato di output è la distanza media che intercorre tra un boid ed i restanti, calcolata come la somma di tutte le distanze reciproche fratto il numero di distanze reciproche tra i boids. Il valore della distanza media, come previsto, diminuisce con il progressivo aggregarsi dei boids fino ad attestarsi attorno ad un valore che dipenderà dai parametri di input iniziali (in particolar modo ds e s).
Durante le varie sperimentazioni ci siamo resi conto che, in alcuni casi, lo stormo, dopo essersi aggregato, tendeva a diminuire drasticamente la propria velocità (probabilmente a causa di un fattore di coesione troppo elevato ed un numero eccessivo di boid nello spazio di simulazione) e i boids tendevano ad "oscillare" attorno a posizioni stazionarie. Proprio per via di questo effetto di oscillazione la velocità media risultava comunque elevata e pertanto poco utile a capire cosa stava succedendo. Per questo motivo abbiamo deciso di aggiungere un ulteriore parametro di output, ovvero $flock\_velocity$. Questa funzione somma algebricamente le componenti x e y di tutti i boid, divide i risultati per il numero di boid e calcola il modulo del vettore somma tra le componenti; così facendo è possibile ottenere una velocità qualitativa che tiene conto anche del verso delle velocità. Come previsto, nel momento in cui la simulazione viene avviata, il valore di $flock\_velocity$ è piccolo, poiché le velocità sono uniformemente distribuite in tutte le direzioni. Man mano che i boids tendono ad aggregarsi e procedere nella stessa direzione, esso aumenta visibilmente a patto che vi sia un solo gruppo compatto o più gruppetti che procedono nella medesima direzione.

5 Strategia di test

Per assicurarci di poter verificare il corretto funzionamento di ogni parte del programma, abbiamo suddiviso ogni funzione in numerose funzioni più piccole. Abbiamo testato le varie funzioni tramite il documento *doctest.h* e utilizzando la funzione $assert()$ nel $main$, in modo da assicurarci che il programma termini nel caso di input errati da parte dell'utente.

6 Dichiarazioni di utilizzo di strumenti AI

Abbiamo sfruttato strumenti AI per aiutarci nell'implementazione della libreria $std :: chrono$, che è utilizzata per il corretto funzionamento del loop degli eventi.
Sempre nei file di SFML, ci siamo avvalsi di strumenti AI per imparare la sintassi corretta delle classi interne di questa libreria, in particolare quelle per l'importazione delle texture, e nel setup dello spazio di simulazione, cioé in $sf :: View$ ed $sf :: Sprite$.
Infine, abbiamo utilizzato l'AI per risolvere un problema in cui la finestra veniva avviata all'inizio del programma ma non all'inizio del ciclo, tramite la linea di codice:
$extern\ sf :: RenderWindow\ window$;
(riga 8 del file $boids\_sfml.hpp$)

7 Possibili errori

Durante gli ultimi test prima della consegna, uno membro del gruppo ha ottenuto un errore relativo a MESA e a glx prima della richiesta degli input, oltre che dei "memory leak" subito dopo la fine del loop. Cercando su internet, pare che siano problemi relativi ad SFML e non al codice da noi scritto, anche perché un altro membro del nostro gruppo è riuscito a compilare senza che questi errori si presentassero. Noi pensiamo che potrebbe essere

un errore dovuto a conflitti tra WSL ed SFML, in quanto il membro del gruppo che non ha avuto questi errori non fa uso di WSL, bensì di un *dual boot*.

8 Link al github del progetto

Aggiungiamo il link alla pagina Github che abbiamo usato per coordinarci durante la stesura del programma.
`https://github.com/dvd-rizzi/progetto_esame`