Simulazione Boids in C++ con SFML

Filippo Dalmonego
Riccardo Licata
Marta Cazzin

October 8, 2024

1 Introduzione

Questo progetto implementa una simulazione di Boids in C++ in uno spazio bidimensionale (toroidale o chiuso), utilizzando la libreria grafica SFML. L'obiettivo è simulare il comportamento collettivo di stormi di uccelli attraverso tre regole fondamentali:

- *separazione*: un boid si allontana dai boids vicini;
- *allineamento*: un boid tende ad allinearsi alle traiettorie dei boids vicini;
- *coesione*: un boid tende a muoversi verso il baricentro dei boids vicini. Il modello è basato sulla simulazione tramite intelligenza artificiale di Craig Reynolds del 1986.

Il link per la directory di Github è il seguente: https://github.com/licric/Boids.

2 Scelte Progettuali e Implementative

2.1 Architettura del Codice

Il progetto è strutturato in modo modulare, suddividendo le funzionalità principali in file header (`.hpp`) e implementazioni (`.cpp`). Le principali classi e strutture sono:
Vec2D : Classe per la gestione di vettori bidimensionali, con operazioni aritmetiche e geometriche. Le funzioni membro `rotate()`, `angleBetween()`, `magnitude()` e `dotProduct()` permettono di lavorare con i vettori 2D e operare trasformazioni come quella fatta da `naturalVeer()` in `Boid.cpp` su `sumCorr`.
Boid : Questa struttura rappresenta un singolo boid, includendo attributi come posizione, velocità e un numero identificativo. Inoltre, sono implementati diversi metodi per il calcolo e la gestione dei movimenti del boid:

- `boidCanSee()`: Verifica se un boid è in grado di "vedere" i suoi vicini.
- `distSquared()`: Calcola la distanza al quadrato tra il boid corrente e un altro boid.
- `limitVelMaxMin()`: Limita la velocità del boid entro un intervallo definito.
- `naturalSteer()`: Limita la velocità di sterzata, rendendo i movimenti del boid più fluidi e naturali. Per visualizzare meglio l'operato della funzione è possibile caricare su geogebra il file fornito nella cartella *others* , mentre in README.md sono presenti specifiche ulteriori riguardo questa ed altre funzioni.

Mentre le prime tre funzioni svolgono operazioni abbastanza intuitive, il metodo `naturalSteer()` richiede una spiegazione più approfondita. Il suo obiettivo è regolare i cambiamenti di direzione del boid, simulando un comportamento più naturale rispetto a bruschi cambi di rotta. Nella sezione successiva, verrà spiegato il funzionamento di `naturalSteer()` in dettaglio, insieme alle motivazioni che ne giustificano l'implementazione.

Flock: rappresenta lo stormo di Boids, i quali attributi privati sono i 3 coefficienti `a_`, `c_`, `s_` e le 3 quantità che controllano l'abilità di visione dei Boids nello stormo `radOfVision_`, `radTooClose_` e `angleOfVision_` (quest'ultima è fissata). Le variabili discusse rappresentano i parametri fondamentali della simulazione, i quali vengono chiesti all'utente al momento dell'esecuzione. Differentemente, il vettore di Boids resta un oggetto pubblico in quanto sarà necessario modificarlo con la funzione `evolve()` ad ogni frame. I metodi di questa classe sono:

- `compute()`: Calcola i 3 coefficienti di correzione della velocità e li somma in un solo vettore di vettori 2D (`sumCorr`) utilizzando la struct `Corrections`.
- `evolve()`: Aggiorna la velocità con `sumCorr` e le posizioni con le nuove velocità, prendendo come parametri l'incremento temporale `dt` e la misura della finestra di simulazione. Prima di aggiornare il vettore velocità di sogni boid viene chiamata `naturalVeer()` per rinormalizzare `sumCorr`, mentre dopo averla aggiornata viene chiamata `limitVelMaxMin()` che appunto limita la velocità tra un massimo e un minimo prima di aggiornare le posizioni. Infine vengono usati i parametri di dimensione della finestra di simulazione per gestire il comportamento ai bordi, i quali possono essere completamente aperti (toroide) oppure chiusi (Boids rimbalzano sulle pareti).

Neighbors: Struttura per ottimizzare il calcolo dei vicini visibili per ogni boid. Contiene tre attributi:

- `seen`: contiene gli identificatori (`N`) dei boid visibili, accumulati in un unico vettore per tutti i boid.
- `offset`: indica l'indice di partenza in `seen` per i vicini di ciascun boid, permettendo di sapere dove iniziano i dati pertinenti.
- `howMany`: specifica il numero di vicini per ogni boid, facilitando l'accesso ai dati nel vettore `seen`.

La funzione libera `findNeighbors()` calcola i vicini per ogni boid basandosi sull'angolo (`angleOfVision()`) e sul raggio di visione (`radOfVision()`). Per ciascun boid verifica quali altri boid sono visibili utilizzando il metodo `boidCanSee()`. Se un boid è visibile, il suo identificatore viene aggiunto a `seen`, e il conteggio dei vicini viene incrementato. Aggiorna infine `howMany` con il numero totale di vicini trovati.

3 Istruzioni per Compilare, Testare ed Eseguire

Per compilare il programma alcuni prerequisiti sono consigliati:

- **C++17**: Il codice richiede un compilatore che supporti lo standard C++17.
- **SFML 2.5**: Libreria grafica per la visualizzazione.
- **CMake**: Per gestire il processo di compilazione.

3.1 Compilazione

Abbiamo usato **CMake** per compilare il nostro programma. In seguito riportiamo i passi da seguire:
installare cmake e le librerie sfml (se non già installati)

```
Ubuntu: sudo apt install cmake libsfml-dev
Mac: brew install cmake sfml
```

Successivamente spostarsi nella cartella Boids e creare la cartella build, configurare l'ambiente cmake e compilare

```
mkdir build
cd build
cmake -DCMAKE_BUILD_TYPE=Release ../
cmake --build . -j4 --target main
```

Per compilare i test invece, sempre all'interno di /build:

```
cmake --build . --target test.t
```

3.2 Esecuzione

Entrare nella cartella build e per avviare il programma e i test, rispettivamente: ./main e ./test.t

4 Input e Output

4.1 Input

All'avvio, il programma richiede all'utente di inserire i valori di alcuni parametri. L'utente può anche decidere se lasciare i valori standard o modificarli. In seguito si trovano i valori standard che sono consigliati per un numero di Boids attorno ai 250-300:

- **Coefficiente di allineamento (a)**: 1.
- **Coefficiente di coesione (c)**: 2.
- **Coefficiente di separazione (s)**: 7.
- **Raggio di separazione** (d_s): 30.
- **Raggio di visione** (d): 300.

4.2 Output

Il programma visualizza una finestra grafica dove i Boids sono rappresentati come triangoli che si muovono nello spazio simulato, seguendo le regole impostate. Nel file Flock.cpp è possibile scegliere se lasciar volare i Boids in uno spazio chiuso con cornici oppure nello spazio libero, immaginando la finestra grafica come un toroide. La generazione casuale iniziale dei Boids viene fatta in due zone del display separate, ma è solo un tentativo di creare due stormi separati che poi andranno ad unirsi, non influisce sulla dinamica di volo. I parametri possono essere scelti a piacere per simulare comportamenti di gruppo differenti; ad ogni modo per osservare un buon comportamento consigliamo i parametri e il numero di Boids sopra riportati.

4.3 Eventuali problemi/errori

A causa di alcune librerie di SFML, può capitare di ottenere dei leak di memoria indiretti (di piccola entità) a seguito dell'esecuzione del programma. La finestra e la simulazione non è afflitta da questi errori, e la simulazione prosegue correttamente. Con i bordi chiusi è possibile inoltre che il programma si interrompa a causa di un errore derivate da un throw da noi messo nella funzione magnitude(), il quale controlla che non ci siano vettori 2D nulli in quanto con essi andiamo a operare delle divisioni. L'errore emerge quando una grande quantità di boids si scontra in un angolo della finestra di simulazione e non si presenta invece se lo spazio è toroidale.

5 Interpretazione dei Risultati

La simulazione mostra il comportamento emergente dei Boids che, a seconda dei coefficienti impostati, possono formare stormi coesi, allineati o dispersi. Usando i parametri standard e stando all'interno dei range consigliati, il programma simula come aspettato il comportamento di uno stormo di uccelli, rispettando le regole del modello usato. Inoltre è possibile osservare sullo schermo le velocità medie e la distanza media dei Boids con le rispettive deviazioni standard. Possiamo osservare la distanza media calare negli istanti iniziali quando lo stormo si forma, mentre può succedere che i Boids si dispongano in fila e continuino in una sola direzione se si sceglie la configurazione a schermo senza bordi. La durata di questo fenomeno "anomalo" , però, dura alcuni secondi, dopo i quali la fila si piaga su se stessa e da nuovamente origine ad un gruppo unito. Quando il numero di Boids va sopra i 400 la simulazione tende a rallentare sui nostri computer, sintomo del fatto che questo progetto richiede un grande numero di calcoli per frame, i quali aumentano all'aumentare dei soggetti in volo. Considerando che vengono fatti molti calcoli aggiuntivi rispetto al modello

base quali naturalVeer() e il calcolo delle statistiche di volo possiamo ritenerci soddisfatti con il risultato. E' comunque possibile provare la simulazione con 400/500 boids su laptop, semplicemente la dinamica risulterà più lenta. Un'interessante osservazione è che variando il numero di Boids, per ottenere lo stesso comportamento si devono cambiare i parametri. Ciò è sicuramente dato dal fatto che ogni Boid vedrà più vicini e il comportamento emergente risulterà più coeso in quanto il parametro di coesione peserà di più. Legato a questa considerazione è un provvedimento che è stato preso nel main.cpp quando il programma prende in input il parametro di separazione. Quest'ultimo è infatti limitato inferiormente, e il motivo di ciò è presto chiaro se ci spostiamo nella struct Neighbors. Al vettore seen viene riservata tramite seen.reserve() uno spazio definito che aiuta il programma a sorpassare i problemi di overhead di riallocazione quando il vettore in questione viene riempito. Contenendo i vicini di ogni Boid, infatti, il vettore seen risulta essere molto lungo e vogliamo essere sicuri che la gestione della memoria sia più efficiente possibile in termini di tempo. Inizialmente avevamo scelto un altro modo per salvare i neighbors che si basava su std::vector¡std::vector¡int¿¿, ma riflettendo sull'operato della CPU sulla memoria abbiamo optato per salvare i dati in modo contiguo in memoria utilizzando un vettore unico per i dati e altri due per permetterne la lettura. Non abbiamo svolto test quantitativi sulle differenze di tempo dei due approcci, tuttavia confidiamo che la scelta corrente sia migliore. Tornando al reserve(), abbiamo svelto di stimare il numero di vicini per ogni boids fissando un lower bound per il coefficiente separation e provando ad aumentare il numero di Boids. Il valore 150 è quindi il massimo numero di vicini che un Boid può avere se il numero totale di Boids è nel range consigliato (250-300).