Relazione Progetto Boids

Edoardo Graziani

25 Agosto 2025

1 Introduzione

Il progetto ha come obbiettivo quello di generare un eseguibile che contenga una simulazione di "boids". Questi oggetti così detti dall'abbreviazione di "bird-oid objects" simulerebbero il comportamento di stormi di uccelli, o enti simili, sotto diverse condizioni.

1.1 Prerequisiti

Il progetto fa uso della "Simple and Fast Multimedia Library" (SFML), versione 2.6.2. Ogni versione di SFML 2.6.x risulta però compatibile con il progetto. In quanto il progetto è stato costruito su sistema macOS, il pachetto SFML in uso è stato ottenuto tramite l'uso di `brew` da bash con il comando:

```
brew install sfml@2
```

Il file `CMakeLists.txt` ricerca SFML in base ad una versione 2.x qualsiasi ma risulta ottimale munirsi delle versioni 2.6.x.

1.2 Informazioni su aggiornamenti di SFML

In caso si fosse già in possesso di un pacchetto SFML 2.6.x risulta deleterio, per utenti macOS, aggiornare SFML attraverso il comando su bash:

```
brew update sfml
```

ciò sovrascrive il pacchetto con versioni più recenti come SFML 3.x, che fanno uso di funzioni definite in modo molto differente da quelle in uso nel progetto. Per prevenire l'aggiornamento dei pacchetti è consigliato installare sfml@2 e poi dare il comando:

```
brew pin sfml@2
```

Nel caso fossero presenti sia sfml (aggiornato al'ultima versione o non) e sfml@2, per prevenire problemi di building degli eseguibili si consiglia di applicare questi comandi:

```
brew unlink sfml
brew link sfml@2
```

2 Istruzioni per il building

2.1 Creazione della "build directory"

```
mkdir build
```

2.2 Debug Mode

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

2.3 Release Mode

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

Note: in Release Mode è possibile si presenti un "warning" in riferimneto ad una variabile "inutilizzata". Questo è dovuto all'uso della variabile solo all'interno di un "assert" adiacente.

2.4 Running dell'eseguibile

In seguito alla compilazione è possibile eseguire i comandi:

```
./build/debug/BoidSimulation
```

or

```
./build/release/BoidSimulation
```

Se si preferisce, è possibile impostare:

```
-DBUILD_TESTING=False
```

2.5 Running dei test

I test associati al progetto sono contenuti sotto due forme. La prima è quella di vari assert sparsi nelle funzioni definite nei file. La seconda è quelli di DOCTEST raggruppati all'interno del file `test_boid.cpp`. Questi ultimi sono eseguibili in due maniere (sia nel caso di debug mode che di release mode) dalla directory principale: - nel caso di debug mode:

```
`ctest --test-dir build/debug --output-on-failure`
`./build/debug/test_boid`
```

- nel caso di release mode:

```
`ctest --test-dir build/release --output-on-failure`
`./build/release/test_boid`
```

nel caso si usasse `ctest` si otterrebbe un "singolo test" la cui fallibilità è legata al fallimento di uno dei suoi componenti. Nel caso si esegua direttamente `test_boid` si otterrebbe il numero di test presenti, falliti e passati.

3 Regole di interazione

3.1 Controllo dei vicini

Tutte le interazioni tra boid tengono conto della "vicinanza" tra boids e determinano l'interazione da applicare in base a differenti "raggi" o "distanze". Dato un boid b_i, i suoi vicini (nel caso di una certa interazione voluta al massimo a distanza d) sono tutti i boids b_j per cui:

$$\left|\boldsymbol{x}_{b_i} - \boldsymbol{x}_{b_j}\right| < d$$

3.2 Regole di volo

Le regole di volo vengono usate per determinare ognuna una componente della variazione di velocità del boid a cui vengono applicate. Per ogni boid b_i:

$$\boldsymbol{v}_{b_i} = \boldsymbol{v}_{b_i} + \boldsymbol{v}_s + \boldsymbol{v}_c + \boldsymbol{v}_a + \boldsymbol{v}_e + \boldsymbol{v}_f$$

$$\boldsymbol{x}_{b_i} = \boldsymbol{x}_{b_i} + \boldsymbol{v}_{b_i}$$

dove $\boldsymbol{v}_s$, $\boldsymbol{v}_c$, $\boldsymbol{v}_a$, $\boldsymbol{v}_e$ e $\boldsymbol{v}_f$ sono le velocità ottenute dall'applicazione delle tre regole e dalle aggiuntive forze di allontanamento dagli ostacoli ("evasion" nel caso di `completeEvasion = true`) e di inseguimento del puntatore ("following", nel caso di `mouseFollowMode = true`). L'agiornamento della posizione non presenta l'ordinario "dt" perché autonomamente impostato sull'aggiornamento dei cicli con numero di frame limitato a 120.

Regola 1: separazione

La regola ha lo scopo di evitare che i boids collidano tra di loro.

$$\boldsymbol{v}_1 = -s \sum_{j \neq i} (\boldsymbol{x}_{b_j} - \boldsymbol{x}_{b_i}) \quad \text{se} \quad \left| \boldsymbol{x}_{b_i} - \boldsymbol{x}_{b_j} \right| < d_s$$

Usando la distanza come fattore per determinare la nuova velocità si ottiene una accelerazione graduale del boid che si allontana. Un opportuno fattore di separazione s determina l'intensità della repulsione. La distanza $d_s < d$ stabilisce il range di influenza della regola.

Regola 2: allineamento

Lo scopo della regola è fare in modo che i boids tendano a procedere nella stessa direzione dello stormo.

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

Viene sottratta alla media delle velocità dei boids (escluso il boid su cui è applicata la regola) la velocità attuale del boid b_i. Il tutto moltiplicato per un fattore di allineamento $a < 1$, che determina la rapidità con cui il boid sterza.

Regola 3: coesione

La regola induce il boid a volare verso il centro di massa dei boid vicini. Il centro di massa è dato da:

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

La velocità $\boldsymbol{v}_3$ si ottiene sottraendo la posizione attuale del boid alla posizione del centro di massa, moltiplicando poi il risultato per un fattore di coesione c opportuno:

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

Regola 4: evasione

Lo scopo della regola è evitare le collisioni tra un boid e gli ostacoli presenti nell'ambiente.

$$\boldsymbol{v}_4 = \begin{cases} r \dfrac{\boldsymbol{p}_{b_i} - \boldsymbol{p}_{\text{ostacolo}}}{\|\boldsymbol{p}_{b_i} - \boldsymbol{p}_{\text{ostacolo}}\|} & \text{se } \|\boldsymbol{p}_{b_i} - \boldsymbol{p}_{\text{ostacolo}}\| < R_s \\ \boldsymbol{0} & \text{altrimenti} \end{cases}$$

La forza viene calcolata solo se la distanza tra il boid b_i e l'ostacolo è minore di una soglia di sicurezza R_s, definita come la somma del raggio del boid e di una dimensione caratteristica dell'ostacolo. Se la condizione è soddisfatta, si genera una forza repulsiva $\boldsymbol{v}_4$ diretta lungo la retta che unisce il punto più vicino sull'ostacolo al boid, normalizzata e moltiplicata per un fattore di intensità, qui riferito come r (chiamato diversamente all'interno dei file) . In caso contrario, la forza è nulla.

Regola 5: inseguimento del puntatore

Questa regola ha lo scopo di dirigere i boid verso la posizione corrente del puntatore.

$$\boldsymbol{v}_5 = \begin{cases} m \dfrac{\boldsymbol{p}_{\text{mouse}} - \boldsymbol{p}_{b_i}}{\|\boldsymbol{p}_{\text{mouse}} - \boldsymbol{p}_{b_i}\|} & \text{se } \|\boldsymbol{p}_{\text{mouse}} - \boldsymbol{p}_{b_i}\| > \varepsilon \text{ e non colpito} \\ \mathbf{0} & \text{altrimenti} \end{cases}$$

La forza è diretta verso la posizione del puntatore del mouse, ed è normalizzata per mantenere direzione ma non dipendere dalla distanza. È moltiplicata per un coefficiente $m < 1$ (all'interno del progetto è stato sostituito con il valore fisso di 0.1f per questioni empiriche), che regola la rapidità con cui il boid si dirige verso il puntatore. La forza viene applicata solo se la distanza tra il boid b_i e il puntatore supera una soglia ε (per evitare oscillazioni quando sono troppo vicini ed evitare la creazione di agglomerati nei pressi del puntatore) e il boid non è in stato di *hit* (nello stato in cui si trova un boid che ha colliso con un ostacolo è costretto a rimanere fermo per un certo periodo per aggiornare il suo stato di "danneggiato"). In caso contrario, la forza è nulla.

4 Struttura del Programma

4.1 boid

I file `boid.cpp` e `boid.hpp` gestiscono le informazioni fondamentali dei singoli boid. Ogni boid contiene le informazioni riguardo ai raggi d'azione `r1, r2` ed `r3` ed il raggio che ne determina la dimensione `radius`; tutti come membri privati della classe `Boid`. Questi sono membri comuni a tutti i boid e quindi resi statici come anche la forma `b_bird` ed il numero di lati per disegnarlo, `sides`, legati alla grafica di SFML. La classe contiene poi elementi non statici che sono legati al cronometro interno di ogni Boid che viene usato nel corso delle collisioni: questi sono `damage, hitTimer, isHit` e `hitColorChanged`.
L'evoluzione del singolo boid dal punto di vista della posizone e della velocità sono dati da `Update_Position` e `SpeedChange`. La prima funzione somma ad ogni ciclo il vettore velocità a quello posizione. La seconda funzione aggiorna completamente e direttamente i valori del vettore velocità. Oltre ai vari "getters" e "setters" sono presenti anche le funzioni che aggiornano lo stato di collisione del singolo boid. Queste funzioni sono `MarkHit`, `UpdateHit` e

`ApplyDamage`. La prima da inizio al timer interno in base allo stato di collisione ed indica il danno subito ad `ApplyDamage`. La seconda aggiorna il timer interno e comunica la possibilità di dover eliminare il boid. L'ultima aggiorna solo il colore del boid in base al danno subito. All'interno del file sono presenti anche operatori necessari per facilitare i calcoli con i vettori: questi sono il prodotto scalare, la norma e la differenza al quadrato di due vettori.

4.2 flock

I file `flock.cpp` e `flock.hpp` contengono dichiarazione e definizione delle tre accelerazioni: separazione, coesione ed adesione. Il file `flock.hpp` contiene inoltre la dichiarazione delle tre costanti moltiplicative applicate alle tre forze.

4.3 obstacle

I file `obstacle.cpp` e `obstacle.hpp` definiscono gli ostacoli come sottoclasse dei boid, caratterizzati dall'override delle funzioni `Update_Position` e `SpeedChange` in modo che posizione e velocità vengano costantemente mantenuti al vettore zero. Le variabili per l'evasione completa, insieme alle dimensioni dell'ostacolo ed al numero di lati sono definiti in modo statico in quanto comuni a tutti gli ostacoli. La funzione `CollisionResponse` gestisce le interazioni boid-ostacolo quando `completeEvasion` risulta vera. In caso contrario le forze di repulsione sono date dalla funzione `RepelBoid`.

4.4 quadtree

I file `quadtree.cpp` e `quadtree.hpp` gestiscono la generazione del quadtree. La funzione principale del quadtree è di creare sotto-riquadri in cui sia possibile ridefinire quali boid considerare nel calcolo delle forze di interazione. In assenza di ciò ad ogni ciclo ogni boid dovrebbe tenere conto di ogni altro boid. Con il quadtree ogni boid fa riferimento solo ai boid all'interno dello stesso riquadro: questi boid vengono poi chiamati "neighbours" (vicini). Per definire il corretto funzionamento del quadtree le funzioni presenti gestiscono: la divisione del singolo riquadro in quattro riquadri (questa operazione si può ripetere entro certi limiti stabiliti), la definizione dei sotto-riquadri, il metodo d'inserimento dei boid e l'eliminazione di sotto-riquadri non più utilizzati. Per evitare problemi con l'eliminazione dei puntatori sono stati utilizzati gli "smart pointers".

4.5 evolution

I file `evolution.cpp` e `evolution.hpp` gestiscono l'aggiornamento della velocità e della posizione dei boid in seguito all'applicazione delle forze in atto. Queste possono essere le tre forze ordinarie (separazione, coesione ed allineamento) o le forze aggiuntive fornite dall'allontanamento dagli ostacoli (`completeEvasion`)o dall'inseguimento del puntatore dell'utente (`mouseFollowMode`). La funzione

`Evolution` presente all'interno dei file gestisce anche l'effetto pac-man dello spazio toroidale su cui si muovono i boid.

4.6 menu

I file `menu.cpp` e `menu.hpp` definiscono attraverso delle `struct` i quattro elementi che possono essere usati all'interno dei menù. Questi sono: `Slider`, `CheckBox`, `Button` e `Notification`. Ognuno di questi elementi contiene funzioni che generano, creano le grafiche e gestiscono le interazioni per ognuna delle loro parti. Questi elementi sono poi utilizzati all'interno della classe che definisce i menù. I vari menù sono definiti attraverso una `struct` che viene poi chiamata nella funzione `generateLayout`: questa genera i componenti dei menù in base alla loro tipologia. I menù contengono sliders, bottoni e checks per impostare entro certi limiti i raggi d'azione delle forze dei boid, la grandezza dei boid, il numero di boid generati e le varie modalità con cui funziona la simulazione. Queste ultime sono: l'allontanamento dagli ostacoli, il danno graduale e l'inseguimento del puntatore dell'utente.

4.7 main

Il `main.cpp` contiene definizione e dati della finestra in utilizzo. All'interno del `main.cpp` sono poi inizializzati i valori default dei moltiplicatori delle forze e dei vari raggi d'azione e quello che definisce la dimensione dei boid. Sempre all'interno del `main` sono poi definiti i limiti superiori per la presenza di boid ed ostacoli; questi sono stati scelti in modo arbitrario.
Sono poi utilizzate tutte le funzioni (`random_device`, `random_engine` e le varie distribuzioni) che permettono di generare casualmente posizioni e velocità dei boid iniziali e generati tramite mouse.
All'interno del `main` sono presenti tre cicli `while`. Il primo di questo mantiene aperta la finestra, che essa si trovi sulla simulazione che sui menù. Il primo sotto-ciclo `while` gestisce le interazioni con i menù e trasmette i valori inseriti attraverso i menù alle variabili della simulazione. Il secondo ciclo `while` gestisce tutti i tasti che possono essere usati all'interno della simulazione e la simulazione stessa. Questo implica che vi siano inseriti i cicli di inserimento dei boid ed ostacoli nel quadtree e tutti i cicli che gestiscono le interazioni boidboid e boid-ostacolo.

5 Risultati ed osservazioni

I boid generati rapidamente si riuniscono in gruppi/stormi di varie dimensioni che possono essere ben osservati a piccole dimensioni e gran numero di boid generati. A scapito delle forze imposte nel caso di inseguimento del puntatore si ottengono i comportamenti più disordinati. Spesso spinti da un numero elevato di boid limitrofi alcuni di questi riescono a collidere. Un altro comportamento

insolito studiato è quello osservabile nel corso delle collisioni con danno graduale. Il boid in questi casi subisce danno, ma ha la possibilità di allontanarsi dall'ostacolo. In molti casi però, soprattutto in presenza di stormi massicci, il boid viene rapidamente spinto contro lo stesso ostacolo. Si era infatti considerata la possibilità di uno stato di "semi-esistenza" dei boid in cui sarebbe stato permesso loro di allontanarsi senza subire interazioni boid-boid, per un certo periodo dettato dal timer interno.

6 Links

Link per raggiungere la repository di GitHub contenente il progetto:
`https://github.com/EdGr203/BoidSimulation.git`