Presentazione progetto d'esame: Simulazione di stormi

Sofia Berluti

16 ottobre 2024

Sommario

Questo progetto ha lo scopo di simulare, tramite un programma in C++, il comportamento di uno stormo di boids in uno spazio bidimensionale, sulla base di tre regole: separazione, allineamento e coesione. Sono inoltre state aggiunte delle regole finalizzate a gestire il comportamento ai confini dello schermo e il modulo della velocità dei boids.

1 Introduzione

La simulazione, resa nota da un software di intelligenza artificiale realizzato nel 1986 da Craig Reynolds, si basa sull'interazione tra agenti detti *boids* in uno spazio bidimensionale. Nel modello più elementare i boids determinano la loro traiettoria tramite tre regole di volo (separazione, allineamento, coesione), che fanno si che essi formino uno stormo coeso.

Le regole di volo dei boids vengono applicate tenendo conto solo dei boids "vicini". Nello specifico, dato un boid b_i, i suoi vicini sono tutti i boids b_j per cui:

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

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

$$\vec{v}_{b_i} = \vec{v}_{b_i} + \vec{v}_1 + \vec{v}_2 + \vec{v}_3 \tag{2}$$

$$\vec{x}_{b_i} = \vec{x}_{b_i} + \vec{v}_{b_i}\Delta t \tag{3}$$

La regola di separazione (Eq. 4) fa sì che il boid interessato si allontani dai boids vicini e ha quindi lo scopo di evitare che i boids collidano tra loro. La regola di allineamento (Eq. 5) fa si che il boid tenda ad allinearsi alle traiettorie dei boids vicini e che proceda quindi nella stessa direzione dello stormo, mentre la coesione (Eq. 6) fa si che il boid si muova verso il centro di massa (Eq. 7) dei boids vicini.

$$\vec{v}_1 = -s\sum_{j\neq i}(\vec{x}_{b_j} - \vec{x}_{b_i}) \quad \text{se} \quad |\vec{x}_{b_i} - \vec{x}_{b_j}| < d_s \tag{4}$$

$$\vec{v}_2 = a(\frac{1}{n-1}\sum_{j\neq i}\vec{v}_{b_j} - \vec{v}_{b_i}) \tag{5}$$

$$\vec{v}_3 = c(\vec{x}_c - \vec{x}_{b_i}) \tag{6}$$

$$\vec{x}_c = \frac{1}{n-1}\sum_{j\neq i}\vec{x}_{b_j} \tag{7}$$

Nelle equazioni sopra rappresentate compaiono i parametri *s, a, c.* Essi sono dei fattori di proporzionalità che determinano l'influenza di una certa regola sul moto dei boids: s determina l'intensità della repulsione, a la rapidità con cui il boid sterza e c la tendenza dei boids a restare uniti.

Compaiono inoltre i parametri n, numero di boids coinvolti nella simulazione, e d_s, distanza entro cui agisce la regola di separazione.

Oltre al criterio della distanza, nel programma è stato inserito anche un angolo di vista, motivo per cui i boids riconosceranno come propri vicini solo quei boids che rientrano nel raggio scelto e si trovano entro il loro angolo di vista.

Nella simulazione è anche stato aggiunto un controllo sulle velocità dei boids, in modo tale che esse siano sempre comprese tra due valori fissati.

Inoltre i boids possono muoversi all'interno di due diversi spazi 2D: uno rettangolare, tale per cui, una volta raggiunto uno dei bordi, essi tornano indietro con velocità opposta; e uno toroidale, tale per cui una volta raggiunto uno dei bordi, il boid riappare dalla parte opposta dello schermo con la stessa velocità.

2 Discussione del codice

É possibile trovare il codice al seguente link di GitHub: https://github.com/SofiaBerluti/ProgettoBoids

2.1 Compilazione ed esecuzione

2.1.1 g++

Per l'esecuzione del programma è necessario aver installato la libreria grafica SFML e l'Xserver MOBAXTERM.
Per compilare ed eseguire il codice principale e aprire la scheda grafica è necessario lanciare i seguenti comandi:

```
$ g++ -Wall -Wextra -fsanitize=address -lsfml-graphics -lsfml-window -lsfml-system
boids.cpp vector2D.cpp main.cpp -o boids
$ ./boids
```

Mentre, per quanto riguarda i due test sul funzionamento della classe Vector2D e sulle regole di volo dello stormo sono necessari questi comandi:

```
$ g++ -Wall -Wextra -fsanitize=address vector2D.cpp test_vector2D.cpp -o test_vector2D
$ ./test
$ g++ -Wall -Wextra -fsanitize=address -lsfml-graphics -lsfml-window -lsfml-system
boids.cpp vector2D.cpp test_rules.cpp -o test_rules
$ ./test_rules
```

2.1.2 CMake

Poichè il programma contiene più *translation units*, è possibile utilizzare il sistema di build *CMake* per rendere la fase di compilazione ed esecuzione più semplice. I dettagli e le opzioni sui comandi da eseguire sono riportati in un file *CMakeLists.txt*. Per creare la directory di lavoro e generare i file con le istruzioni specifiche si utilizza il seguente comando da terminale:

```
$ cmake -S . -B build -DCMAKE_BUILD_TYPE=Release
```

Per eseguire le istruzioni generate dal comando precedente si utilizza il comando:

```
$ cmake --build build
```

E infine:

`$ build/boids-sfml` (per quanto concerne il main del progetto)
`$ build/rules.t` (per quanto concerne i test sulle regole di volo)
`$ build/vector2D.t` (per i test sulla *struct* vector2D)

2.2 Struttura del codice

Il codice implementato è composto da otto *translation units*: tre source file *main.cpp*, *vector2D.cpp*, *boids.cpp*; tre header file *vector2D.hpp*, *boids.hpp* e *rules.hpp* e due file *test_vector2D.cpp* e *test_rules.cpp* responsabili del testing.

2.3 Classi e Strutture

2.3.1 Vector2D

La struct *Vector2D* contiene due attributi di tipo float: **x** e **y**, volti a rappresentare le due coordinate di un vettore sul piano bidimensionale. Vi sono inoltre il metodo **magnitude()**, che calcola il modulo di un vettore, e la *free function* **get_angle()**, che calcola, in radianti, l'angolo compreso tra due vettori tramite una formula derivata dall'algebra lineare.

Inoltre, sempre nel file Vector2D.hpp, sono stati dichiarati come free functions undici operatori volti a calcolare l'uguaglianza, la diversità, la somma, la differenza, il prodotto e la divisione tra due oggetti di tipo Vector2D.

Tutti i metodi e le free functions sono stati poi definiti nel file Vector2D.cpp

2.3.2 Bird

La struct *Bird* rappresenta le caratteristiche di un singolo boid e contiene quindi due attributi di tipo Vector2D: **position** e **velocity**. Quindi d'ora in poi si userà equivalentemente boid e Bird.

2.3.3 Flock

La classe *Flock* contiene undici attributi privati: un *vector* **flock_** finalizzato a contenere tutti gli oggetti di tipo Bird coinvolti nella simulazione, un int **number_** che rappresenta il numero di Bird, cinque double **separation_**, **alignment_**, **cohesion_**, **distance_separation_** e **viewangle_**, che rappresentano i parametri principali della simulazione, due double **window_width_** e **window_height_** e un *enum* di tipo **Space**.

Vi sono poi il costruttore, che prende come argomento un oggetto di tipo **Parameters**, e quattro metodi: **start()**, **evolve()**, **draw()**, **get_statistics()**.

Il metodo start() genera un numero number_ di oggetti di tipo Bird posizionati casualmente su tutto lo schermo.

Il metodo evolve(), facendo uso dell'algoritmo **for_each()**, applica le funzioni **get_neighbours()**, **separation()**, **alignment()** e **cohesion()** ad ogni Bird contenuto nel vector flock_, in modo da aggiornarne la velocità. Applica poi le funzioni **avoid_speeding()** e **boundaries_behavior()** e aggiorna la posizione di ogni Bird moltiplicando l'oggetto position ad esso associato per un intervallo di tempo espresso dalla variabile deltaTime di tipo float.

Il metodo draw() disegna sulla *window* tanti oggetti **triangle** di tipo sf::CircleShape, ognuno associato ad un Bird e alle sue coordinate, con l'attenzione a orientare l'inclinazione del triangolo in base alla velocità del Bird ad esso associato.

Infine, il metodo get_statistics() salva i dati ottenuti su un file **data.csv** e li stampa sulla window. Nello specifico, si ottengono la velocità media di tutti i Bird con relativa standard deviation e la distanza media, ricavata facendo la media di tutte le distanze prese tra un Bird e il Bird a lui più prossimo, con relativa standard deviation.

2.3.4 Space, Parameters, Settings

Per organizzare le variabili coinvolte nella simulazione sono state create due ulteriori struct: la prima, **Parameters**, contiene otto variabili legate alle regole di volo e utilizzate anche nel costruttore della classe Flock, mentre **Settings** contiene quattro double: **window_width**, **window_height**, **max_speed**, **min_speed**.

Inoltre è stato creato un enum **Space** contenente due variabili: **toroidal** e **rectangular**, in modo da associarvi due diversi comportamenti dei Bird ai bordi della window nella funzione boundaries_behavior().

2.4 Rules.hpp

Nel file rules.hpp sono state dichiarate e definite *inline* sei free functions.

get_neighbours() individua, dato uno specifico Bird, tutti gli altri Bird che si trovano entro uno spicchio di circonferenza di raggio distance e ampiezza pari a view_angle e li raggruppa all'interno di un vector chiamato neighbours.

Le tre funzioni **separation()**, **alignment()** e **cohesion()**, se il vector neighbours non risulta vuoto, ricavano le tre componenti aggiuntive di velocità, facendo uso delle formule riportate in Eq. 4, 5, 6.

Infine sono state implementate altre due funzioni. **boundaries_behavior()**, a seconda del tipo di spazio in cui ci troviamo, "teletrasporta" il Bird dall'altra parte della window riassegnando la sua posizione una volta che esso raggiunge uno dei bordi (*toroidal*), oppure riassegna un valore opposto alla sua velocità per farlo tornare indietro senza che oltrepassi i bordi (*rectangular*). **avoid_speeding()**

Item	Quantity
Number	150
Separation	0.4
Alignment	0.6
Cohesion	0.008
Distance	170
Distance Separation	30
View Angle	2.5
Space	toroidal

Tabella 1: Parametri di default

riassegna un valore costante (max_speed o min_speed) al modulo della velocità di un Bird se esso supera il valore di soglia max_speed o scende al di sotto del valore minimo min_speed.

2.5 Main

Per prima cosa vengono richiesti in input i parametri iniziali, attributi della struct Parameters, necessari allo sviluppo del modello. L'utente può scegliere se procedere con dei parametri di default (vedi Tab. 1) oppure inserirli manualmente. Dopodichè vengono settati anche i parametri della struct Settings. In particolare, window_width e window_height vengono impostati al novanta per cento della lunghezza e della larghezza dei desktop.

Si imposta uno sfondo caricando un file *.jpg* dopo aver creato una texture e un background tramite la libreria grafica SFML e si setta il *frame rate* a 60 frame al secondo.

Si creano quindi un oggetto clock di tipo sf::Clock e un testo **stats** di tipo sf::Text volto a riportare le quantità calcolate da get_statistics() in tempo reale sulla window. Successivamente si imposta il *font* di stats dal file *cmunnt.ttf* e si crea una **box** nera di tipo sf::RectangleShape che fungerà da sfondo per le stringhe stampate a schermo di stats.

Viene poi creato un oggetto flock di tipo Flock e inizializzato con gli attributi di parameters. Subito dopo, flock chiama la funzione start().

A questo punto si apre il ciclo principale del programma e al suo interno viene creata la variabile deltaTime, il cui valore viene mantenuto al di sotto di un millisecondo tramite una condizione *if*.

Flock chiama le funzioni evolve() e draw() e i risultati calcolati da get_statistics() vengono stampati sullo schermo all'interno della box dopo essere stati convertiti in delle stringhe.

2.6 Test

Per assicurare la correttezza e l'affidabilità del codice sviluppato, sono stati eseguiti alcuni test (vedi Fig. 1), suddivisi in due file differenti.

In test_vector2D.cpp è stato testato il corretto funzionamento della struct Vector2D e degli operatori, sia tramite dei CHECK, sia tramite dei CHECK_THROWS_WITH_AS per le eccezioni.

In test_rules.cpp sono invece state testate tutte le free functions di rules.hpp. In particolare si è testato il corretto funzionamento di get_neighbours(), delle tre regole di volo, della funzione avoid_speeding() per il controllo della velocità e poi si è verificato che la funzione boundaries_behavior() riassegnasse i valori corretti dell'attributo position qualora essi raggiungessero i limiti indicati.

2.7 Messaggi di Warning

Al momento dell'esecuzione il programma presenta una serie di warning (vedi Fig. 2), dovuti alla versione OpenGL utilizzata, che tuttavia non ostacolano il suo corretto funzionamento.

2.8 Presentazione e discussione dei risultati

Di seguito sono riportati alcuni screenshot della simulazione (vedi Fig. 3 e 4). Come ci si aspetterebbe, inizialmente i boids sono sparsi randomicamente su tutto lo schermo, mentre dopo un certo lasso di tempo iniziano a raggrupparsi e a formare uno stormo coeso.

```
sofia_berluti@LAPTOP-BDFCUJT8:~/Boids$ build/vector2D.t
[doctest] doctest version is "2.4.5"
[doctest] run with "--help" for options
===============================================================================
[doctest] test cases:  1 |  1 passed | 0 failed | 0 skipped
[doctest] assertions: 27 | 27 passed | 0 failed |
[doctest] Status: SUCCESS!
sofia_berluti@LAPTOP-BDFCUJT8:~/Boids$ build/boids.t
[doctest] doctest version is "2.4.5"
[doctest] run with "--help" for options
===============================================================================
[doctest] test cases:  2 |  2 passed | 0 failed | 0 skipped
[doctest] assertions: 20 | 20 passed | 0 failed |
[doctest] Status: SUCCESS!
```

Figura 1: Risultati dei test eseguiti

```
Warning: The created OpenGL context does not fully meet the settings that were requested
Requested: version = 1.1 ; depth bits = 0 ; stencil bits = 0 ; AA level = 0 ; core = false ; debug = false ; sRGB = fals
e
Created: version = 0.0 ; depth bits = 0 ; stencil bits = 0 ; AA level = 0 ; core = false ; debug = false ; sRGB = false
Setting vertical sync failed
sfml-graphics requires support for OpenGL 1.1 or greater
Ensure that hardware acceleration is enabled if available
OpenGL extension SGIS_texture_edge_clamp unavailable
Artifacts may occur along texture edges
Ensure that hardware acceleration is enabled if available
```

Figura 2: Warning

Al fine di osservare questo comportamento regolare, è necessario scegliere con cura i parametri. In particolare, mentre i valori di separation e alignment possono tranquillamente variare tra 0 e 1 (si consigliano valori intorno a 0.5), è importante porre attenzione su cohesion: si consiglia un numero compreso tra 0.001 e 0.01, per evitare che i boids si raggruppino eccessivamente arrivando a sovrapporsi. Inoltre, si consiglia di scegliere un valore di separation_distance, distanza entro cui agisce la regola di separazione, inferiore di circa un ordine di grandezza rispetto a distance, distanza entro la quale vengono individuati i vicini. Si ricorda che le distanze sono tutte espresse in pixel, e che la larghezza e l'altezza del desktop corrispondono rispettivamente a circa 860 e 1500 pixel.

Tuttavia, nello spazio toroidale, in prossimità dei bordi, incorrono alcuni problemi: infatti i boids situati in parti opposte dello schermo non si riconoscono come vicini, nonostante lo spazio dovrebbe essere continuo. Questo fatto crea delle momentanee interruzioni nella continuità dello stormo durante il volo ed è necessario che passi un po' di tempo prima che i boids tornino compatti. Sempre a causa del comportamento ai bordi, la standard deviation della distanza media può subire grandi fluttuazioni.

Dopo un certo lasso di tempo, quando si è ormai formato uno stormo compatto, i valori di distanza media e velocità media raggiungono un valore pressochè costante di approssimativamente 25 pixel e 14.8 pixel/s.

Figura 3: Inizio simulazione

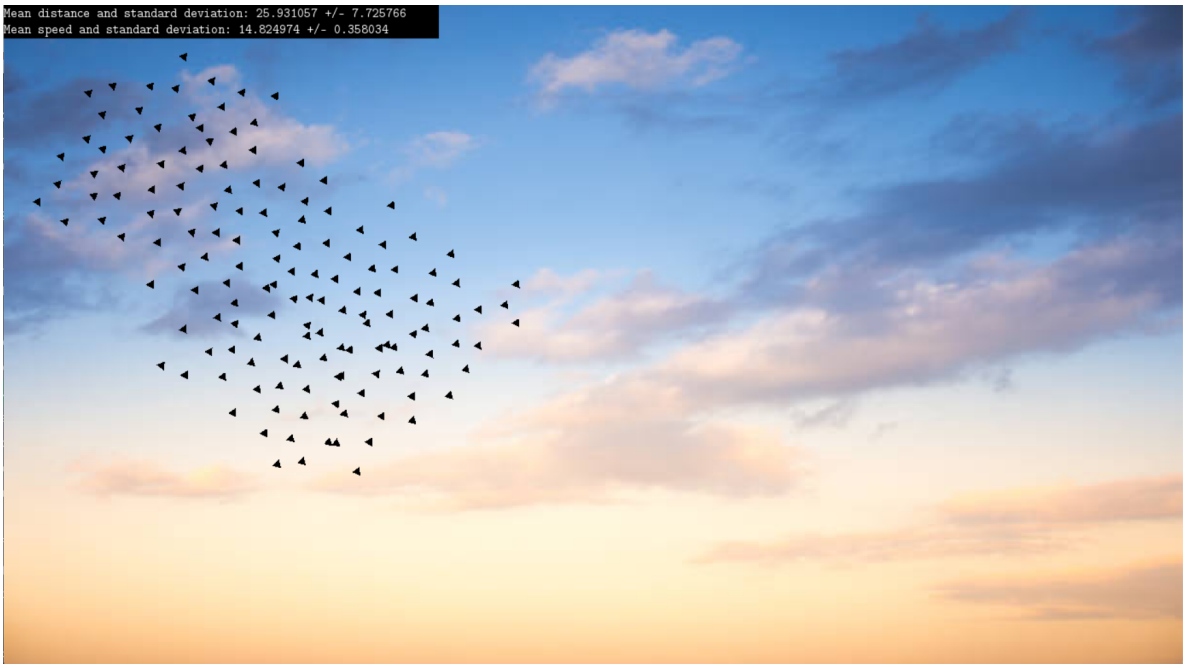

Figura 4: Simulazione avviata