Relazione sul progetto d'esame per il corso di Programmazione per la Fisica

Chiara Baldelli, Liam Cavini

11 Settembre 2023

Indice

1 Struttura del programma

Il programma realizzato simula il comportamento di uno stormo in uno spazio bidimensionale attraverso oggetti detti boid. Esso si compone di dodici file principali inseriti nella cartella `source`: alcuni di questi sono header file le cui funzionalità sono definite nei corrispondenti file sorgente. Tutte le funzioni e classi (al di fuori del main) sono contenute nel namespace `boid`.

I blocchi fondamentali presenti nel programma sono:

- **constants.hpp**: Si compone di un unico header file in cui sono definite le costanti usate nei restanti file del progetto. Si è scelto di definire le variabili in un header file invece che in un file .cpp per permettere al compilatore di usarle come costanti compile-time.

- **point.hpp/cpp**: Contengono la definizione della classe `Point`. Questa presenta due attributi privati, che rappresentano le coordinate di un vettore in due dimensioni. Gli oggetti `Point` possono essere sommati/sottratti fra di loro e moltiplicati per un double; si comportano rispetto a queste operazioni in modo analogo ai vettori. Inoltre vi sono funzioni per prendere il modulo del vettore (utile per il calcolo di distanze) e per ruotarlo di un certo angolo. La classe `Point` è usata in quasi ogni file del programma, il suo utilizzo permette di scrivere codice più sintetico e pulito.

- **boid.hpp/cpp**: `boid.hpp` contiene la definizione di tre classi, `Bird`, `Boid` e `Predator`. La classe `Bird` contiene i metodi e attributi condivisi da `Boid` e `Predator`, come `m_pos` e `m_vel`, due punti che rappresentano la posizione e la velocità rispettivamente. Dalla classe `Bird` ereditano `Boid` e `Predator`, che aggiungono metodi specifici alla loro classe. Di particolare importanza è il metodo `update` della classe `Boid`, che aggiorna la posizione del boid, e applica le forze di allineamento, coesione e separazione (chiamando i rispettivi metodi) in base ai boid che sono nel suo range. La verifica di quali boid siano nel range non viene fatta da `update`, spetta invece al programmatore fornire a questa funzione il vettore di boid rispetto ai quali vengono applicate le forze (chiamato `in_range`). Questo perché, per ottenere una performance migliore, il compito di popolare `in_range` è lasciato all'oggetto `Quad_tree`.

  La classe `Predator` ha il proprio metodo `update`, che trova il più vicino boid nel range del predatore e muove quest'ultimo verso di esso.

  I metodi della classe `Bird`, e quindi condivisi da predatori e boid, sono `turn_around`, che gestisce il rientro dei boid/predatori nello schermo in caso superino un certo margine, e `repel` che allontana il boid da un punto passato alla funzione. Quest'ultima funzione viene usata per allontanare i boid dai predatori e dal puntatore del mouse (quando si preme il pulsante sinistro).

- **quadtree.hpp/cpp**: Come precedentemente menzionato, il vettore `in_range` utilizzato come argomento nella funzione `update` dei boid viene popolato attraverso un'istanza della classe `Quad_tree`, definita in `Quad_tree.hpp`. L'obiettivo principale di questa classe è di costruire efficacemente tale vettore, evitando l'iterazione su n^2 boid, dove n rappresenta il numero di boid presenti sullo schermo.

  Per raggiungere questo obiettivo, l'oggetto `quad_tree` suddivide lo spazio in cui i boid risiedono.

  Un oggetto `Quad_tree` contiene un membro che rappresenta una cella nella partizione dello spazio, di tipo `Rectangle`. `Rectangle` è una struct definita in `Quad_tree.hpp`. Questa struct, in modo simile a `Point`, è utilizzata per rendere il codice più leggibile. Il membro rappresenta una cella nella partizione dello spazio. Per rappresentare i boid contenuti nella cella, viene usato un vettore di puntatori a boid, sempre membro di `quad_tree`. Altri membri rappresentano la capacità della cella, se la cella è divisa, e le celle figlie. Questi ultimi sono puntatori ad altri oggetti `Quad_Tree`. L'inizializzazione di questi puntatori avviene quando la dimensione del vettore di boid supera la capacità massima. Quando ciò accade la cella si divide in quattro celle figlie, e i boid vengono distribuiti tra le celle figlie. L'aggiunta e la distribuzione dei boid sono gestite dal metodo `insert`, mentre il processo di suddivisione della cella è gestito dal metodo `subdivide`.

Per costruire il vettore `in_range` di un boid, viene utilizzato il metodo `query`. Questo metodo identifica con quali celle il range del boid si sovrappone, chiamando il metodo `square_collide`. Successivamente, il vettore `in_range` viene popolato con i boid contenuti in queste celle che rientrano nel range specificato.

La classe `Quad_tree` alloca dinamicamente le celle figlie. Si è utilizzato puntatori intelligenti per gestire la deallocazione.

- **sfml.hpp/cpp**: contengono due funzioni, `vertex_update` e `display_circle`

  `vertex_update` è una funzione template che gestisce gli oggetti della libreria SFML chiamati vertici, usati per la rappresentazione grafica dei `Boid` e dei `Predator`. La funzione aggiorna la posizione dei vertici in modo che formino dei triangoli posizionati in corrispondenza dei boid, e direzionati nel verso del moto.

  `display_circle` disegna su schermo un oggetto cricle della libreria SFML, centrato sulla posizione del boid. Questa funzione viene utilizzata per visualizzare i vari range dei boid.

- **gui.hpp/cpp**:

  gestisce l'implementazione del pannello di controllo (la GUI) tramite la classe `Panel`. `Panel` raccoglie tutti gli elementi del pannello in una mappa fra questi e una enum class definita nel file `gui.hpp`. Inoltre gestisce la loro grandezza e posizionamento. La classe `Panel` permette di ridurre la quantità di codice necessaria per inizializzare e modificare gli elementi del pannello, e il numero di parametri da passare alle funzioni che lo fanno. Queste ultime sono definite in `gui.cpp`.

- **main.cpp**: contiene la funzione `main`. Nel main, viene inizializzato e popolato il vettore di boid e predator, così come l'array di vertici necessario per rappresentarli sullo schermo. Inoltre, vengono effettuate le inizializzazioni per la finestra SFML, il GUI, il pannello di controllo e le variabili che contengono i valori delle forze sui boid e il loro numero. Il main contiene anche il loop principale del programma, che gestisce gli input e chiama le funzioni per l'aggiornamento dei valori dal pannello di controllo, delle posizioni dei boid e dei predatori, oltre a disegnare gli oggetti sulla finestra grafica.

2 Istruzioni per compilazione, esecuzione e testing

Per eseguire il programma è necessario installare due librerie, SFML[1] (Simple and Fast Multimedia Library) e TGUI[2] (Texus' Graphical User Interface): quest'ultima è necessaria per costruire l'interfaccia grafica.

Per la compilazione del programma, è stato utilizzato il "meta" build system CMake. Lanciando dal terminale il comando

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

viene creata una cartella denominata `build`, contenente le istruzioni per compilare correttamente il codice. si compila quindi tramite il comando:

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

vengono creati il file eseguibile `boid` e il file con i relativi test `boid.t`.

Per eseguire il programma si lancia il comando

```
./build/release/boid
```

mentre con il comando

```
./build/release/boid.t
```

[1] `https://www.sfml-dev.org/index.php`
[2] `https://tgui.eu/tutorials/0.9/linux/`

si eseguono i test. Consigliamo di compilare in release mode per una miglior performance ma è possibile compilare ed eseguire il programma anche in debug mode lanciando nell'ordine

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

per creare una cartella `build`.

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

per creare gli eseguibili boid e boid.t

```
./build/debug/boid
```

per eseguire il programma

```
./build/debug/boid.t
```

per eseguire i test
Per non buildare i test è sufficiente sostituire il primo comando con

```
cmake -S . -B build/debug -DBUILD_TESTING=False -DCMAKE_BUILDING_TYPE=Release
```

oppure analogamente

```
cmake -S . -B build/debug -DBUILD_TESTING=False -DCMAKE_BUILDING_TYPE=Debug
```

3 Formato di Input e Output

Una volta avviato il programma si apre una finestra dove si può osservare l'evoluzione temporale del volo dello stormo. Non è necessario alcun dato in input da terminale: vengono inizializzati di default il numero di boid (pari a 300), il numero di predatori (pari a 0) e i tre coefficienti di separazione, coesione e allineamento che regolano il comportamento dello stormo.

l'utente dispone di diversi strumenti per interagire con la simulazione ed eventualmente influenzarne l'andamento. Cliccando in un qualsiasi punto della finestra si ha un effetto di repulsione tra i boid che si trovano in prossimità. Il pulsante `show cells` consente di visualizzare sulla finestra le celle con cui viene ripartito lo spazio con il metodo quadtree.

I primi tre slider presenti consentono all'utente di variare i coefficienti di coesione, allineamento e separazione; il quarto permette invece di variare il numero di boid (rappresentati in verde) tra 0 e 2500.

Inoltre lo slider `Range` offre all'utente la possibilità di modificare il raggio del range di allineamento e di coesione: a esso è associato un pulsante `Display Range` che consente di visualizzare su un boid tale range. Lo slider `separation range` e il pulsante `Display Separation Range` hanno le stesse funzioni per quanto riguarda il range di separazione. Uno slider consente poi di modificare il numero di predatori (rappresentati in rosso) tra 0 e 10. L'ultimo slider permette infine di modificare il raggio del range della preda, quindi la distanza entro cui un boid riconosce un predatore e se ne allontana. Il pulsante `Display Prey Range` consente di visualizzarlo su un boid.

In alto a sinistra sono visualizzati gli fps (frame per seconds).

3.1 Versione alternativa-Output delle statistiche

Si è scelto, seppur richiesto nella consegna, di non mostrare in output le statistiche dello stormo. Questo poiché, dovendo gestire un numero di boid molto elevato (fino a 2500), aggiungere il calcolo delle statistiche avrebbe rallentato estremamente il programma.

Si è tuttavia deciso di non ignorare completamente la consegna, realizzando un fork della repository github (utilizzata in fase di implementazione) dove è presente una versione alternativa del progetto con le statistiche implementate e visibili in alto a destra nella finestra grafica. Consigliamo, durante la visione di questa simulazione, di non aumentare eccessivamente il numero di boid in modo da permettere al

programma di essere efficiente. Il fork contiene gli stessi identici file del progetto originale con l'aggiunta di

- **statistics.hpp/cpp**: vi sono implementati i metodi statistici usati per il calcolo di media e deviazione standard di posizione e velocità dei boid.

Inoltre il file

- **main.cpp** è stato modificato per permettere la visualizzazione in alto a destra delle statistiche mediante la realizzazione di un oggetto label della libreria TGUI.

il fork si trova a `https://github.com/AbyssDelver/Boidventure-statistics`.

4 Interpretazione dei Risultati

Per interpretare i risultati, è necessario osservare la simulazione visualizzata attraverso la libreria SFML. Da un insieme inizialmente disordinato di boid, nel corso di alcuni secondi, si sviluppa una molteplicità di piccoli stormi coesi che tendono progressivamente a convergere in un singolo stormo. Quando compaiono i predatori, il volo dello stormo subisce un disturbo evidente, e i boid cercano di evitare i predatori disperdendosi. Utilizzando il pannello di controllo è possibile sperimentare con i parametri dei boid, ad esempio, è possibile notare una maggiore tendenza alla dispersione quando si riduce il valore dello slider della coesione. Il comportamento dei boid ottenuto variando questi parametri si conforma con le aspettative degli autori.

5 Strategia di Test

Il programma è stato sottoposto a test utilizzando il framework DOCTEST. Tutti i test sono implementati nel file `boid.test.cpp`, situato nella cartella `source/test`. I test possono essere suddivisi in due categorie principali: casi particolari e casi semplici.

I test dei casi particolari sono progettati per valutare il comportamento delle classi nei limiti estremi, ad esempio utilizzando 0 come coefficiente di forza. Questi test mirano a garantire che le classi gestiscono correttamente situazioni eccezionali.

D'altra parte, i test dei casi semplici sono finalizzati a verificare la corretta implementazione dei metodi utilizzando input di base, consentendo così una previsione attendibile degli output.

Inoltre, è stato condotto un test di memory leak sul quadtree per garantire che non vi siano perdite di memoria.

Per quanto riguarda le parti del codice più visibili, sono stati scritti meno test poiché è stato possibile verificarle eseguendo il programma e controllando la finestra grafica. Inoltre, nel codice sono stati inseriti vari assert per evitare input errati nelle funzioni.

6 links

- **github**: `https://github.com/Chiarass/Boidventure/tree/main`
- **github statistics fork**: `https://github.com/AbyssDelver/Boidventure-statistics`