Simulazione di Volo di uno Stormo

Lorenzo d'Errico, Lorenzo Mariotti, Alessandro Pinto, Francesco Tori

1 Design del programma

1.1 Logica

L'apparato logico del programma è formato da: cinque header files, tre files.cpp, due classi e diverse free-functions, per un totale di tre translation units. La struttura è gerarchica, dove in cima sono posti gli header files "boids.hpp" e "rnumbers.hpp". Questi header files sono inclusi, direttamente o indirettamene, da tutte le TU, infatti vanno rispettivamente a definire: la classe Boid e le funzioni per la generazione di numeri casuali. In particolare, "rnumbers.hpp" include a sua volta diversi header files dallo standard. I due precedenti header files sono inclusi a loro volta dall'header file "flock.hpp", nel quale è definita la classe Flock, con i suoi dati membri e la definizione dei metodi minori. La classe Flock ha come punto centrale il dato membro "flock", il quale è un vettore di Boid generati casualmente tramite l'apposito metodo fill(). La definizione dei metodi più pesanti è riservata alle due translation unit: "velocity.cpp" e "statistics.cpp", le quali rispettivamente definiscono i metodi responsabili per il calcolo delle velocità d'interazione tra boids e il calcolo dei dati statistici dello stormo. L'unione di "velocity.cpp" e l'header incluso "flock.hpp", costituisce la translation unit dedicata al calcolo delle velocità d'interazione dei singoli boids tra di loro, o con agenti esterni come i bordi o gli eventuali scoppi dei petardi. In particolare, le funzioni responsabili per il calcolo delle velocità d'interazione interne allo stormo, sono la definizione del corpo di metodi dichiarati in "flock.hpp", mentre le funzioni dedicate agli agenti esterni, sono free-function definite direttamente in "velocity.cpp". La scelta di rendere funzioni membro i metodi per il calcolo delle velocità interne allo stormo, è stata fatta per via dei grandi vantaggi a cui porta l'accesso diretto ai dati membri dell'oggetto "stormo", ed in particolare al suo vettore "flock". Le free-funtions definite in "velocity.cpp" sono visibili dalle altre translation units, grazie alla loro dichiarazione in un apposito header file: "velocity.hpp". L'ultima translation unit dell'apparato logico è "simulation.cpp", che includendo gli header files "flock.hpp" e "velocity.hpp", fa uso di tutto ciò che è stato nominato fino ad ora, fatta meno la parte statistica. In "simulation.cpp" sono definite le due funzioni fondamentali per la corretta evoluzione dello stormo durante la simulazione. La prima è la free-function "Update", la quale accetta come parametri dei dati caratteristici della simulazione ed uno oggetto di tipo "Flock". "Update" si occupa di calcolare ed applicare le nuove posizioni e le nuove velocità di ogni singolo boid. Questo è

possibile tramite il richiamo delle numerose funzioni e metodi definiti in "velocity.cpp". In particolare, di questa funzione ne esiste anche una seconda versione commentata in fondo al file. Questa versione alternativa ha esattamente lo stesso effetto, ma fa uso del multithreading, argomento che chi ha sviluppato la parte della logica si è divertito ad esplorare. Tuttavia è solo una funzione sperimentale che non produce un aumento delle performance, ma al contrario, le diminuisce. L'altra funzione definita in "simulation.cpp" è la free-function "Orientation", la quale è strettamente legata alla parte grafica del progetto e si occupa di definire l'orientazione del boid a cui è applicata. Quanto definito nella translation unit relativa a "simulation.cpp" può essere utilizzato dalle altre translation unit, attraverso l'inclusione dell' header file "simulation.hpp", contenente la dichiarazione delle funzioni.

1.2 Grafica

Riguardo alla parte grafica, la totalità del codice è scritta in "graphics.cpp" e in "graphics.hpp" andando a formare un'unica translation unit relativa alla grafica. Lo sviluppo è avvenuto tenendo conto di due opzioni alternative: inizializzare pochi oggetti grafici, per poi all'interno del game loop andarne a modificare le caratteristiche, spostandoli nello schermo a seconda delle necessità, per infine disegnarle (ad esempio inizializzare una sola casella di testo, disegnarla nel punto 1, poi modificarla, spostarla nel punto 2, disegnarla di nuovo, ripetendo il processo per tutti gli n punti dove deve essere disegnato il testo). Oppure inizializzare subito tutti gli oggetti grafici necessari, modificarli secondo le necessità e andare soltanto a disegnarle durante il game loop. Per la rappresentazione dei boids la prima opzione opzione è stata quella vincente, ciò a causa del numero molto elevato di uccelli che compongono lo stormo. Mentre, per il resto dell'interfaccia (più "statica" e caratterizzata da meno elementi), la scelta è ricaduta sulla seconda opzione, più leggera a livello di calcolo, in quanto non bisogna rimodificare le caratteristiche di ogni oggetto ad ogni frame, ma basta definirle una volta sola. Tuttavia, per evitare centinaia di righe di codice in cui sostanzialmente si inizializzavano e modificavano oggetti quasi identici, si è preferito definire le 3 funzioni "initsetting" che permettono, una volta inizializzato l'oggetto, di definirne le caratteristiche con un'unica espressione.

Inoltre nell'header "graphics.hpp" è stata definita la classe Button, relativa ai pulsanti che fanno variare i parametri, i cui metodi sono stati definiti nel cpp. Vi è poi la definizione della funzione relativa all'aggiornamento dei dati statistici (che va a chiamare le funzioni definite in "statistics.cpp")che, tramite il reset di un apposito clock nel game loop, verrà chiamata ogni 3 secondi (non solo al fine di lasciare sufficiente tempo per leggere i dati a schermo, ma anche per questioni di ottimizzazione).

L'inizializzazione di tutti gli oggetti grafici tramite le funzioni sopra citate e il game loop avvengono nella funzione graphics, che sarà poi chiamata nel main per avviare l'interfaccia grafica. Al fine di rendere il codice più leggibile si è fatto uso di alcune enumerations, definite in cima a "graphics.hpp".

2 Istruzioni per eseguire il programma

2.1 Compilazione ed Esecuzione

Il programma è stato sviluppato stramite l'ausilio del Build System "CMake", nell'archivio consegnato è presente il file "CMakeLists.txt" per la sua configurazione. Il programma compila senza warning di alcun tipo sia in modalità "Debug" che "Release", tuttavia è notevole l'aumento delle performance nel secondo caso. Di seguito i comandi per:

- Generare la cartella "build", dove il programma tiene tutte le informazioni necessarie per la compilazione, a seconda della modalità desiderata:
  - Modalità "Debug":

    $$cmake\ -S\ .\ -B\ build\ -DCMAKE\_BUILD\_TYPE = Debug$$

  - Modalità "Release":

    $$cmake\ -S\ .\ -B\ build\ -DCMAKE\_BUILD\_TYPE = Release$$

- Compilare:

  $$cmake\ --build\ build$$

  L'eseguibile prodotto deve essere aperto da terminale e con un xServer disponibile nel caso l'esecuzione avvenga su una macchina virtuale.

- Eseguire il programma:

  $$build/simulation$$

- Eseguire i test:

  $$build/simulation.t$$

2.2 Note Aggiuntive

Il programma fa un forte utilizzo, sia nell'apparato logico che grafico, della macchina. Durante la fase di sviluppo si è fatto fatica a superare il centinaio di boids, questo ha spinto verso un raffinamento del codice, al fine di migliorare le performance, senza tuttavia sacrificare struttura e estestica. In realtà il grande stacco in quanto a prestazioni, è stato raggiunto risolvendo un setting errato (nel caso di chi scrive) di OpenGL, il quale all'avvio della simulazione stampava:

Warning: The created OpenGL context does not fully meet the settings that were requested

Il malfunzionamento, se presente, è risolvibile innanzitutto eseguendo il comando "glxinfo" nella cartella del progetto. In cima ai dati stampati tramite questo comando è possibile leggere un particolare della configurazione:

$$direct rendering: \ No \ (LIBGL\_ALWAYS\_INDIRECT \ set)$$

Se il direct rendering è settato su "no" è possibile attivarlo tramite il comando:

$$export \ LIBGL\_ALWAYS\_INDIRECT = 1$$

È quindi sufficiente cambiare il valore della variabile da 1 a 0 per attivare l'accelerazione hardware sulla macchina virtuale e risolvere il messaggio d'errore di OpenGl. In questo modo sono state aumentate sostanzialmente le prestazioni. Se il comando non dovesse avere alcun effetto sul parametro, si raccomanda l'installazione di tutte le dependencies citate alla pagina:

$$https://www.sfml-dev.org/tutorials/2.5/compile-with-cmake.php$$

in particolare dei pacchetti *x11* e *xrandr*.

È possibile inoltre che alla chiusura del programma vengano visualizzati dei messaggi di memory leaking, questi sono tuttavia dovuti a possibili problemi interni di SFML, che non hanno nulla a che fare con l'esecuzione del programma in questione.

3 Descrizione input/Output

3.1 Input con esempi

Il programma, quando eseguito, chiede in input il numero di boids da generare in un unico stormo. Il valore inserito verrà accettato o rifiutato secondo una serie di criteri:

1. Viene accettato se il valore inserito è un numero intero;
2. Viene rifiutato se il valore contiene numeri non interi o caratteri che non siano numeri;
3. Fanno eccezione alla regola precedente i caratteri: "+", "-" e " " se posti all'inizio o alla fine dell'input;
4. Viene rifiutato un input contenente spazi non nella prima o ultima posizione, anche se il resto delle cifre sono numeri interi.

I numeri di boids consigliati oscillano tra un minimo arbitrario ed un massimo dipendente da diversi fattori, tra cui: le specifiche della macchina, la modalità con cui si è costruito l'eseguibile, la presenza dell'accelerazione hardware e il collegamento ad una presa nel caso di un portatile. Simulazioni in fase di sviluppo hanno potuto mantenere una decente fluidità fino ai 300 boids circa.

Segue l'apertura di una finestra grafica dove viene mostrata l'evoluzione dello stormo e dei dati statistici. E' possibile interagire con lo stormo in due modi: modificando i parametri tipici dello stormo tramite gli appositi pulsanti, o clickando con il mouse direttamente sulla finestra dove sono rappresentati i boids. Nel primo caso si ottiene un'immediata risposta dello stormo alla modifica dei parametri che ne regolano le interazioni interne e la loro distanza di attivazione (tutti i parametri hanno un loro range e sono settati di default al loro valore intermedio). Nel secondo caso, il click causa un breve scoppio che spaventa i boids, facendoli fuggire in direzione opposta.

3.2 Interpretazione Output

Ogni tre secondi vengono stampati, sia sul terminale che nelle apposite caselle di testo nell'interfaccia, i dati statistici quali: distanza media tra i boids e media dei moduli delle velocità dei boids, con le relative deviazioni standard. Può essere interessante cambiare i parametri e vedere come ciò influenza i dati statistici.

4 Strategie di Testing

I test sono stati fatti tramite lo strumento terzo: "Doctest". Per verificare il corretto funzionamento del programma, è stato costrito un eseguibile a parte dalla simulazione. Questo eseguibile è formato dalle stesse componenti del codice sorgente principale, fatto meno del main e della translation unit relativa alla grafica. I test sono stati sviluppati seguendo l'ordine gerarchico dei file che compongono il programma, e progressivamente testano il corretto funzionamento delle maggior parte delle funzioni definite in tutto l'apparato logico. In totale i test sono 92 e sono tutti superati con successo. Inolte, sono stati inseriti diversi assert nel codice principale, al fine di evitare comporatementi anomali.

5 GitHub

Di seguito il link di accesso alla pagina di GitHub relativa al progetto:

$$https://github.com/Maruann/boids$$