Five nights at DIFA

Lorenzo Redighieri
Luca Zoppetti

12 Giugno 2023

Sommario

Five nights at DIFA è un gioco 2D sviluppato come progetto d'esame per il corso di Programmazione per la Fisica dell'anno accademico 2022-2023. L'obiettivo del protagonista è fuggire dai sotterranei di un edificio in cui imperversa un terribile virus. All'interno del progetto sono implementati il modello SIR e semplici algoritmi utili alla realizzazione di giochi.

Indice

1 Il progetto

Il gioco è stato realizzato interamente in `C++` con l'ausilio di alcuni strumenti di sviluppo che verranno trattati più in dettaglio nella sezione 2. Sono state utilizzate due librerie esterne:

- `SFML`, per il rendering grafico degli elementi di gioco e per la gestione dei movimenti delle entità all'interno del gioco;
- `tmxlite`, una libreria per la lettura di mappe in formato `.tmx`.

Le mappe `.tmx` (quella di gioco e quella di test) che si trovano nell'archivio del progetto sono state disegnate con il programma esterno Tiled, che non è necessario per la compilazione e l'esecuzione del codice.

L'archivio del progetto è strutturato in diverse sotto-cartelle per tenere organizzati il codice e le risorse utili all'esecuzione del programma:

- `assets`, contenente le mappe e tutte le *texture* usate nel gioco;
- `modules`, utilizzata in combinazione con `CMake` per rilevare la libreria esterna `tmxlite`;
- `src`, contenente tutto il codice sorgente suddiviso in cartelle contenenti una classe ciascuna, la quale è separata in un *header file*, un file sorgente e un file di test;
- `test`, utilizzata per creare un eseguibile contenente i test con `doctest`.

1.1 Istruzioni per la compilazione

La procedura descritta in seguito fa riferimento alla piattaforma Ubuntu `22.04` (o equivalentemente a un sistema Windows con `WSL`), su cui devono essere già installati `CMake, git` e `g++`. Per compilare il programma è necessario installare anche le due librerie esterne. `SFML` è disponibile tramite il package manager di Ubuntu, quindi può essere installato con il seguente comando:

```
$ sudo apt install libsfml-dev
```

Al contrario, `tmxlite` dev'essere scaricato e compilato dalla sua *repository* di GitHub. È possibile farlo seguendo questi passaggi:

```
$ git clone https://github.com/fallahn/tmxlite
$ cd tmxlite/tmxlite
$ mkdir build && cd build
$ cmake .. -DTMXLITE_STATIC_LIB=TRUE
$ sudo make install
```

Estraendo l'archivio contenente il codice, è ora possibile proseguire con la compilazione. Il codice può essere compilato in *Debug* mode, che abilita l'*address sanitizer*, o in *Release* mode. In entrambi i casi bisogna seguire la medesima procedura di compilazione dalla cartella principale del progetto:

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

Per eseguire i test si può lanciare il seguente comando:

```
$ build/all.test
```

Per far partire il gioco si utilizza invece il seguente:

```
$ build/fnad
```

In *Debug* mode vengono segnalati dei *memory leak* dall'*address sanitizer*. Essi sono dovuti a codice presente nelle librerie esterne e non sono dovuti a codice scritto direttamente nel progetto.

1.2 Obiettivo del gioco

Il protagonista si trova catapultato nei sotterranei del Dipartimento di Fisica e Astronomia dell'Università di Bologna. Al suo interno imperversa una feroce epidemia che dilaga seguendo il modello SIR[1] e che rende i personaggi presenti nel gioco aggressivi nei confronti del protagonista. Il suo obiettivo è quello di raccogliere tre chiavi entro un tempo limite per poter fuggire da una delle porte presenti nei sotterranei prima che i nemici lo colpiscano tre volte.

1.3 Comandi

Per selezionare il livello all'inizio del gioco si possono utilizzare i tasti da [1] a [5] della tastiera o i tasti da [1] a [5] del tastierino numerico (con [NumLock] attivato). Per muoversi all'interno della mappa sono disponibili i tasti [W], [A], [S], [D] o le freccette ([↑], [←], [↓], [→]). Non è disponibile alcun altro comando[2], le chiavi vengono raccolte automaticamente e per vincere è sufficiente passare sopra una porta dopo aver raccolto tre chiavi. Per chiudere il gioco dopo aver visto la schermata con il risultato si può premere [INVIO].

2 Strumenti di sviluppo

Per migliorare la qualità del prodotto finale sono stati utilizzati diversi software che vengono spiegati dettagliamente nel seguito.

2.1 CMake

`CMake` è uno strumento di gestione del processo di *build* che è stato fondamentale per la realizzazione del progetto. Esso ha permesso di organizzare tutti i comandi necessari alla compilazione in un semplice file di configurazione che è riportato nell'archivio del progetto, `CMakeLists.txt`. In breve, il file di configurazione di `CMake` esegue questi passaggi:

[1] Per la descrizione del modello si rimanda alla sezione 3.1
[2] In realtà nel gioco è presente anche un piccolo *easter egg*.

- imposta delle avvertenze da mostrare durante la compilazione per prevenire alcuni errori comuni;
- abilita l'*address sanitizer* in *Debug* mode per segnalare eventuali *memory leak*;
- cerca le librerie esterne `SFML` e `tmxlite`;
- crea un eseguibile per il programma contenente tutti i file sorgente e collega le librerie esterne;
- costruisce un eseguibile di test unendo tutti i file di test e i rispettivi file sorgente, collegando anche in questo caso le librerie esterne.

2.2 Clang Format

Clang Format è uno strumento di formattazione del codice che è stato utilizzato per applicare regole di formattazione comuni a tutti i file del progetto. È stata utilizzata la configurazione "google" fornita da Clang Format senza apportarvi alcuna modifica. Tutto il codice `C++` del progetto è stato formattato con questo strumento.

2.3 Git e GitHub

Per tenere traccia delle modifiche effettuate nel corso del progetto è stato utilizzato Git, un sistema di *version control* che permette di consolidare insiemi di modifiche apportate al codice in unità dette *commit*. Per la condivisione della *repository* creata con Git è stato utilizzato GitHub, un servizio online per la condivisione di codice. GitHub offre diversi strumenti per facilitare la collaborazione, che sono stati usati ampiamente per il completamento del progetto. Ogni modifica è stata effettuata tramite un'apposita *Pull Request*, all'interno della quale sono stati sfruttati gli strumenti di discussione e di revisione del codice offerti dalla piattaforma. Inoltre, fino al momento dell'aggiunta delle *texture* dei personaggi è stata utilizzata una *GitHub Action* apposita per la compilazione del programma e l'esecuzione dei test automatica ad ogni *push* e ad ogni *Pull Request*. Dopo aver aggiunto le *texture* non è stato più possibile adottare questa strategia di testing perché le macchine virtuali di GitHub su cui vengono eseguite le *Actions* non sono dotate di scheda grafica.

3 Strategie utilizzate

3.1 Il modello SIR

Il modello SIR è stato implementato nei file `epidemic.hpp` ed `epidemic.cpp`. Sono state create due `struct`: `SIRState`, contenente tre `double`, e `SIRParams`, contenente due `double`. `SIRState` indica lo stato del modello SIR considerando S, I e R come variabili continue, mentre `SIRParams` indica i parametri β e γ che

regolano l'andamento dell'epidemia.
È stata infine creata una classe, chiamata `Epidemic`, avente i seguenti dati membri:

- `sir_state_` (di tipo `SIRState`), indica lo stato in tempo reale dell'epidemia;
- `sir_params_` (di tipo `SIRParams`), indica i parametri che regolano l'andamento dell'epidemia;
- `enemies_` (di tipo `std::vector<fnad::Enemy>`), contiene la popolazione di nemici a cui viene applicato il modello SIR;
- `days_per_second_` (di tipo `double const`), indica quanti giorni del modello SIR passano ogni secondo di gioco (il suo valore è 0.5).

Il costruttore di `fnad::Epidemic` non accetta parametri e quando viene chiamato istanzia un'epidemia di default con i seguenti valori:

- `sir_state_ = {1., 1., 0.}`,
- `sir_params_ = {0.7, 0.05}`.

Questa scelta permette di impostare l'epidemia in seguito alla scelta del livello di gioco da parte dell'utente. La creazione vera e propria di un oggetto di tipo `fnad::Epidemic` avviene infatti tramite il metodo pubblico `resetSIRState`, il quale prende in input un oggetto `fnad::SIRState`, da cui estrae i valori da assegnare a `sir_state_`, e un oggetto `fnad::Map`, dal quale ottiene i dati necessari per generare i nemici nei punti corretti della mappa. L'oggetto `fnad::Map` possiede un dato membro di tipo `std::vector<fnad::Spawner>` (chiamato `spawners_`), il quale contiene una collezione di oggetti `fnad::Spawner` (che verranno chiamati *spawners*). Ogni *spawner* rappresenta una certa regione della mappa ed è dotato di un metodo pubblico (`getSpawnPoint`) che restituisce un punto casuale all'interno di essa. A ciascun nemico viene dunque associato casualmente uno *spawner* con probabilità proporzionale all'area di quest'ultimo, il quale assegna una posizione sulla mappa al nemico tramite il metodo `getSpawnPoint`.

L'evoluzione dell'epidemia è gestita dal metodo pubblico `evolve`. Questa funzione prende in input un oggetto di tipo `sf::Time` (che chiameremo `dt`) e un altro di tipo `fnad::Character` (che chiameremo `character`). `dt` indica il tempo passato nel gioco, che viene utilizzato per capire di quanto deve evolvere l'epidemia seguendo il modello SIR. Invece `character` serve per determinare il movimento dei nemici infetti. Infatti `evolve`, oltre a gestire l'evoluzione logica della pandemia, ne gestisce anche quella fisica, stabilendo lo spostamento grafico da applicare ad ogni oggetto contenuto nel vettore `enemies_`: i nemici infetti che "vedono"[3] il personaggio si muovono verso di lui, mentre tutti gli altri (tranne

[3]Un nemico "vede" il personaggio se il segmento che li congiunge non interseca alcun muro della mappa. Questo aspetta verrà trattato nella sezione 3.3

i rimossi) compiono un moto casuale.
Ogni volta che viene chiamato il metodo `evolve` (quindi ogni *frame* di gioco) viene aggiornato il dato membro `sir_state_` tramite le seguenti equazioni:

$$
\begin{aligned}
S_i &= S_{i-1} - \beta \frac{S_{i-1}}{N} I_{i-1} \Delta T \\
I_i &= I_{i-1} + \left(\beta \frac{S_{i-1}}{N} I_{i-1} - \gamma I_{i-1} \right) \Delta T \\
R_i &= R_{i-1} + \gamma I_{i-1} \Delta T
\end{aligned}
$$

dove S_i, I_i e R_i indicano, rispettivamente, i valori delle variabili `sir_state_.s`, `sir_state_.i` e `sir_state_.r` nell'i-esimo frame di gioco, S_{i-1}, I_{i-1} e R_{i-1} indicano i valori posseduti dalle stesse variabili nel frame precedente, N indica il numero totale di nemici (pari alla dimensione di `enemies_`), β e γ indicano i valori delle variabili `sir_params_.beta` e `sir_params_.gamma` (costanti durante l'intera partita) e ΔT indica il valore di `dt`.
Il dato membro `sir_state_` rappresenta lo stato continuo dell'epidemia (`s`, `i` e `r` sono `double`), ma ai fini del gioco serve una rappresentazione discreta di tale stato (avrebbe poco senso infettare un numero non intero di nemici). È dunque necessario tradurre questi `double` in tre `int`. A tale scopo, all'interno di evolve, ciascun dato membro di `sir_state_` viene arrotondato all'intero ad esso più vicino e, confrontando l'`int` così ottenuto con il numero corrente di nemici suscettibili, infetti o rimossi si è in grado di calcolare il numero di nemici da infettare e da rimuovere per aggiornare lo stato dell'epidemia.

3.2 Gestione delle collisioni

Per gestire le collisioni con i muri, assieme alla mappa grafica è stata creata una "mappa logica" con il programma esterno Tiled. Essa consta di quattro livelli di oggetti rettangolari che vengono interpretati dal programma per creare il mondo di gioco. In particolare, il primo livello è quello contenente i muri, il secondo gli *spawner* di nemici, il terzo le uscite e il quarto le chiavi. Per contenere tutte queste informazioni è stata creata la classe `fnad::Map`, che tiene al suo interno quattro vettori contenenti elementi rispettivamente di tipo `fnad::Wall` (semplicemente un alias per il tipo di `SFML sf::FloatRect`), `fnad::Spawner`, `fnad::Exit` (anch'esso un alias di `sf::FloatRect`) e `fnad::Key`.

L'oggetto di tipo `fnad::Map` viene creato all'interno della classe `fnad::Game`, responsabile della gestione dei loop di gioco, e viene poi passato a ciascuna entità che viene creata dal gioco. La classe `fnad::Entity` conserva una referenza alla mappa e la utilizza ogni volta che viene chiamato il metodo `safeMove` per applicare il movimento richiesto tenendo conto della presenza dei muri. Il metodo sopra citato funziona analizzando le collisioni separatamente sull'asse x e sull'asse y, per evitare che un movimento in diagonale possa permettere a un'entità di superare l'angolo di un muro senza essere bloccata. L'algoritmo cerca i muri con cui è avvenuta una collisione tramite il metodo di `SFML intersects` e seleziona fra questi il più vicino al punto di partenza dell'entità. Successivamente, calcola la correzione da applicare su ciascuno dei due assi cartesiani per

riportare l'entità al bordo del muro. Il metodo `safeMove` ritorna un oggetto di tipo `fnad::Collision` che è poi utilizzato dal metodo di movimento casuale della classe `fnad::Enemy` per invertire la direzione nel caso in cui ci sia stato un contatto con un muro.

3.3 Visione dei nemici

Come accennato in precedenza, per determinare il movimento dei nemici infetti è necessario verificare che essi "vedano" il personaggio. Se non lo vedono si muovono casualmente, altrimenti si dirigono verso di lui.
Per verificare se il personaggio rientra nel campo visivo di un nemico è stato scritto un metodo pubblico di `fnad::Enemy` chiamato `sees`, che prende in input un oggetto di tipo `fnad::Character` (il personaggio) e restituisce un `bool` (`true` se vede il personaggio, `false` se non lo vede). Per determinare questo risultato, viene innanzitutto ricavata l'equazione della retta (che indicheremo con r) che congiunge il personaggio e il nemico. Successivamente, per ogni muro della mappa vengono calcolate le equazioni delle rette contenenti i suoi quattro lati al fine di trovare i punti di intersezione tra la retta r e le quattro rette del muro. Se almeno uno di questi punti appartiene al segmento che congiunge il personaggio e il nemico, allora la funzione restituisce `false`, altrimenti restituisce `true`.

3.4 Rendering grafico

3.4.1 Lo sfondo

Per ottimizzare il rendering grafico di un componente molto elaborato come lo sfondo è stata utilizzata la classe di `SFML` `sf::RenderTexture`. Essa funziona in modo molto simile a una finestra, permettendo di disegnare qualsiasi elemento si desideri su di essa. Tuttavia, al contrario di una finestra, essa non viene mostrata immediatamente, ma può essere tenuta in memoria ed essere utilizzata in seguito a seconda della necessità, ad esempio per essere disegnata proprio su una finestra.

La logica della rappresentazione dello sfondo è stata implementata nella classe `fnad::Background`. Il suo costruttore accetta come input una mappa disegnata con Tiled (di tipo `tmx::Map`) e ne legge tutti i *tileset* (delle particolari immagini che contengono tutti i blocchi per costruire il mondo disegnato con Tiled) per caricare le corrispondenti *texture* in una mappa privata all'interno della classe. In seguito, il costruttore procede a leggere tutti i livelli a partire dal quinto in poi. I primi quattro livelli sono infatti riservati alla costruzione logica della mappa, con i muri, gli *spawner* di nemici, le chiavi e le uscite. Ciascuno di questi livelli viene passato alla funzione `drawLayerToBackground` insieme ai *tileset* letti in precedenza. Il livello viene letto a partire dall'angolo in alto a sinistra, scorrendo tutti i blocchi verso destra e scendendo poi alla riga di blocchi successiva. Ciascun blocco viene mappato così a un corrispondente oggetto di tipo `fnad::Tile`, che indica il *tileset* di riferimento e un quadrato utilizzato per "ritagliare" la *texture* intera del *tileset* all'immagine necessaria per

rappresentare il singolo blocco. Per ciascun blocco viene successivamente creato un `sf::Sprite`, ovvero un oggetto dotato di una *texture*, su cui viene caricata la *texture* del blocco e che viene disegnato nella posizione corretta all'interno della `sf::RenderTexture`. In seguito al processo di lettura, la *texture* completa viene stampata su un apposito `sf::Sprite`, che viene poi richiamato dal metodo `Background::draw` per essere disegnato sulla finestra all'interno del *loop* di gioco.

3.4.2 I personaggi

Per animare i personaggi sono state utilizzate delle sequenze di sei immagini per ognuna delle quattro direzioni in cui può muoversi ciascuna entità. Le classi `fnad::Character` e `fnad::Enemy` sono state dotate di un metodo pubblico `animate`, dentro al quale è stata gestita la logica delle animazioni.
Per minimizzare gli sprechi di risorse sono state caricate solo tre *texture*: due per il personaggio (in movimento e in posizione statica) e una per i nemici, che non sono mai fermi. Dato che in una partita viene istanziato un solo oggetto di tipo `fnad::Character` e molti oggetti di tipo `fnad::Enemy`, a quest'ultimo è stato aggiunto un dato membro `static` di tipo `sf::Texture`, così da tenere in memoria un'unica *texture* condivisa fra tutti i nemici. Sono stati aggiunti due dati membri di tipo `sf::Texture` anche a `fnad::Character`, tuttavia non sono stati messi `static` perché in ogni caso viene sempre istanziato un unico oggetto di questo tipo per ogni partita.

Per l'animazione di `fnad::Character` serve innanzitutto decidere se utilizzare la *texture* statica o quella dinamica. In seguito, in base alla direzione del personaggio, viene scelta una regione della *texture* all'interno della quale vi sono i sei disegni con cui costruire l'animazione. Una volta determinata tale regione di interesse, con l'aiuto di un dato membro di `fnad::Character` di tipo `sf::Clock` e della funzione `setTextureRect`, viene creata l'animazione ridimensionando ogni decimo di secondo la *texture* del personaggio in modo che vengano mostrate in sequenza le sei immagini contenute nella regione di interesse.

L'animazione di `fnad::Enemy` funziona in maniera pressoché identica a quella di `fnad::Character`, con l'unica differenza che non è necessario controllare se il nemico si stia muovendo o se sia fermo, poiché i nemici sono costantemente in movimento.

4 Testing e debugging

Ciascun componente del gioco è stato accuratamente testato, ove possibile tramite l'utilizzo di *unit tests*. Per scrivere i test è stato utilizzato `doctest`, uno strumento di testing *single-header* il cui codice è scritto nel file `doctest.h` dell'archivio. In alcuni casi, soprattutto per i componenti grafici del gioco, non è stato possibile testare le funzionalità tramite codice, quindi sono state adottate altre strategie.

4.1 I test sulle classi di gioco

I test scritti mirano a verificare che tutta la logica richiamata dalla classe `fnad::Game` per gestire gli eventi di gioco si comporti come previsto. Per quanto riguarda le entità è stata verificata soprattutto la correttezza dei movimenti: il protagonista deve muoversi correttamente secondo l'input dell'utente, i nemici devono spostarsi in modo differente a seconda del loro stato SIR, e tutte le entità in generale non devono collidere con i muri. Inoltre, sono stati scritti dei test per verificare le collisioni fra i nemici e il protagonista, che devono portare a una diminuzione dei punti vita di quest'ultimo.

Per la classe `fnad::Epidemic` è stato verificato che l'evoluzione di una pandemia d'esempio ritornasse i risultati attesi e che, in seguito a un arbitrario numero di iterazioni, il vettore di nemici rimanesse sincronizzato con lo stato SIR continuo. Per le altre classi di gioco (`fnad::Background`, `fnad::Map`, `fnad::Key`, `fnad::Spawner`) è stato verificato che la loro creazione producesse gli oggetti attesi e che essi interagissero correttamente con il protagonista.

4.2 Metodologie di debug grafico

Per studiare il comportamento del programma dal punto di vista grafico sono state adottate altre strategie. Ad esempio, sono state create diverse mappe con più livelli grafici per verificare che ogni oggetto disegnato con Tiled venisse effettivamente disegnato a schermo. Inoltre, la mappa è stata esplorata più volte per verificare il corretto posizionamento dei muri. Durante il corso del progetto sono stati anche creati piccoli programmi separati per studiare il comportamento di alcune funzionalità specifiche di `SFML`, come i testi e le texture.