Programmazione Modulo 2:
Dr. Mario

Il progetto consiste nella realizzazione di un clone (semplificato) del gioco *Dr. Mario* rilasciato da Nintendo nel 1990 per il *Nintendo Entertainment System*. Vi abbiamo fornito il template del progetto che implementa una grafica semplificata del gioco originale e la gestione dell'input da tastiera. Il vostro compito consiste nell'implementare la logica del gioco e ulteriori funzionalità descritte in questo documento. Esiste una versione del gioco che potete provare online al seguente indirizzo:

`https://jsnes.org`

1 Dr. Mario

Mario, nei panni di un medico, deve liberare il corpo di una persona infetto da dei virus utilizzando delle pastiglie colorate. Potete vedere un esempio di schermata del gioco in Figura 1.

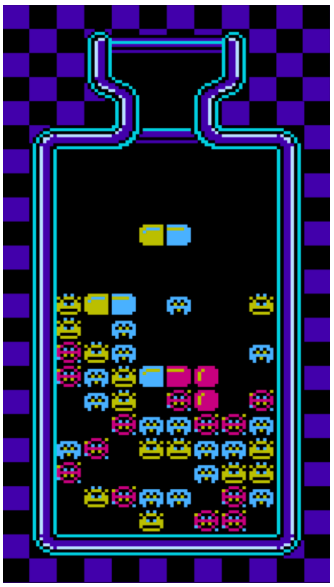


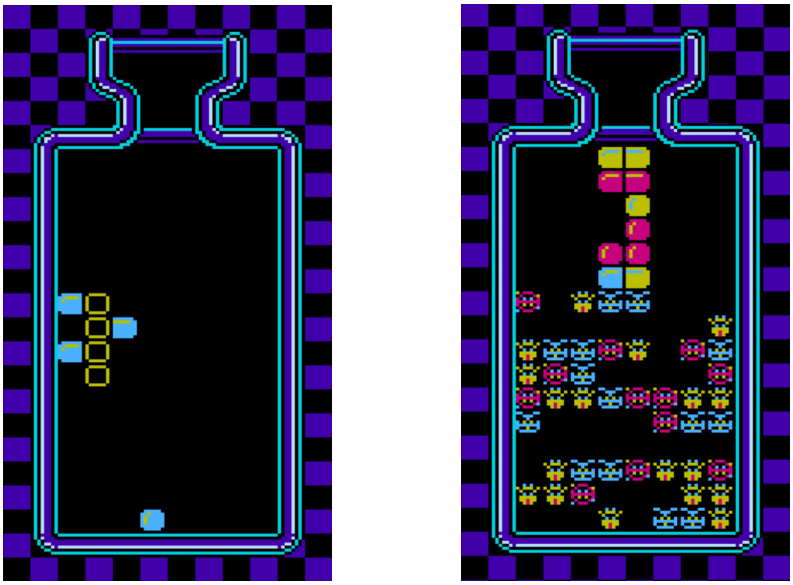
Figura 1: Schermata di Dr. Mario.

I virus possono essere di tre colori diversi: rosso, giallo e blu. Le pastiglie sono dei rettangoli di dimensione 2×1 e ogni metà di una pastiglia può avere uno dei tre colori dei virus.

Le pastiglie (di colore casuale) vengono inserite nella posizione centrale della prima riga della matrice che rappresenta il corpo. Possono essere spostate (a destra, sinistra e in basso) o fatte ruotare (di novanta gradi a destra o sinistra) finché non si appoggiano su un virus o sul fondo dello schema.

A questo punto, se quattro (o più) elementi (virus e parti di pastiglie) dello stesso colore sono allineati in orizzontale e/o in verticale, vengono rimossi dallo schema. Se in seguito a questa operazione sono presenti parti di pastiglie sospese del vuoto, queste devono cadere fino a che non si appoggiano su un virus o sul fondo dello schema, eventualmente eliminando ulteriori virus/pastiglie. Questo procedimento deve essere ripetuto finché non ci sono più elementi che stanno cadendo, quindi deve essere generata una nuova pastiglia. Un esempio del procedimento appena descritto è mostrato in Figura 2.

La partita continua fino a che tutti i virus vengono eliminati dallo schema oppure la posizione in cui inserire la nuova pillola è ostruita da altre pillole già presenti nello schema. Questi due casi sono mostrati in Figura 3.



(a) Vittoria!

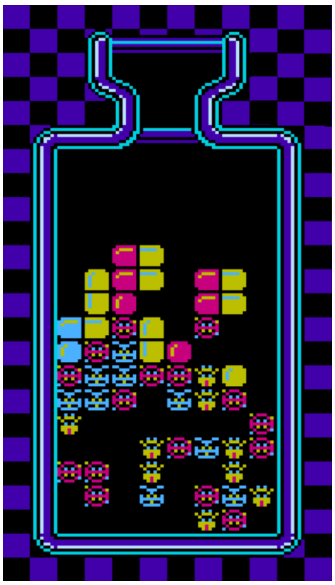
(b) Sconfitta :(

Figura 3: Fine della partita.

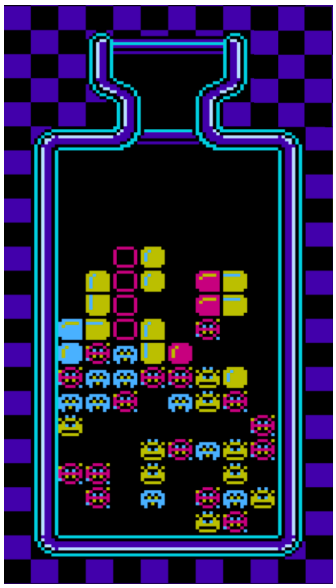
2 Struttura del progetto

Il template del progetto che vi abbiamo fornito consiste di cinque file, due header e tre file C. I file `drmauro_main.c`, `game.c`, `game.h` possono tranquillamente essere ignorati, a meno che non decidiate di implementare alcune delle funzionalità extra descritte in Sezione 6.

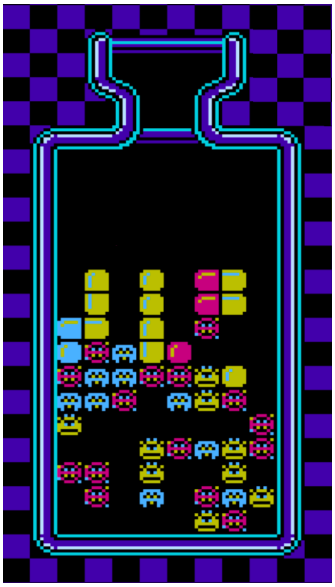
Il file `drmauro.h` contiene lo scheletro delle strutture dati fondamentali del programma e le firme delle quattro funzioni da implementare. Lo stato del gioco è salvato in una variabile di `struct campo` definito come segue:



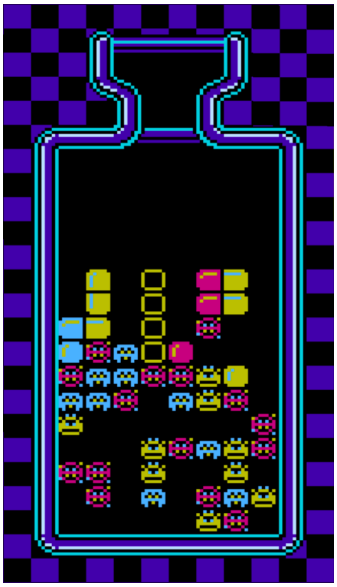
(a) Il virus rosso può essere eliminato.



(b) Ora le 2 pastiglie gialle cadono.



(c) Le 4 pastiglie gialle vanno rimosse.



(d) Si può inserire una nuova pastiglia.

Figura 2: Esempio di eliminazione.

```
struct gioco {
  struct cella campo[RIGHE][COLONNE];
  int punti;
  /* ---- */
  int active_id;
};
```

I campi `campo` e `punti` sono necessari per l'implementazione della grafica e non devono essere modificati. In particolare, il campo di gioco è una matrice di elementi `struct cella`, il cui tipo è definito come segue:

```
struct cella {
  enum contenuto tipo;
  enum colore colore;
  /* ---- */
  int id;
};
```

Una cella contiene informazioni sul tipo di componente del gioco (`VUOTO`, `MOSTRO`, `PASTIGLIA`) ed eventualmente il colore (`ROSSO`, `GIALLO`, `BLU`). Anche questi due campi non possono essere rimossi dalla definizione in quanto sono necessari per l'implementazione della grafica.

I campi `id` e `active_id` sono un suggerimento per implementare la logica del gioco. In particolare, se il contenuto di una cella del campo è parte di una pastiglia, `id` contiene l'identificatore univoco della pastiglia di cui fa parte. Il campo `active_id` contiene l'identificatore della pastiglia attualmente in movimento.[1]

3 Funzioni da implementare

3.1 Logica del gioco

Nella funzione `step` dovete implementare la logica del gioco descritta in Sezione 1. La funzione ha la seguente firma:

```
void step(struct gioco *gioco, enum mossa comando)
```

In particolare, dovete modificare lo stato del gioco in seguito all'esecuzione della mossa specificata dal parametro `comando` che può assumere i seguenti valori:

- `NONE`: nessuna mossa specificata;
- `DESTRA`: sposta la pastiglia a destra di una posizione;
- `SINISTRA`: sposta la pastiglia a sinistra di una posizione;
- `GIU`: fa cadere la pastiglia in basso;
- `ROT_DX`: ruota la pastiglia di 90 gradi a destra;
- `ROT_SX`: ruota la pastiglia di 90 gradi a sinistra.

Se non è stata specificata nessuna mossa oppure la mossa specificata non è possibile (ad esempio, ci si trova già nella colonna più a sinistra e viene dato il comando `SINISTRA`), la pastiglia deve essere spostata verso il basso di una posizione. In seguito alla mossa dovete controllare se è possibile eliminare alcuni dei mostri/pastiglie presenti nello schema.

[1] Se decidete di seguire questa strategia, gli identificatori dovete generarli voi in fase di creazione della pastiglia.

Fate attenzione al verificarsi dei seguenti casi:

- in seguito all'eliminazione dei mostri/pastiglie potrebbero esserci delle parti di pastiglie da far cadere: in tal caso, ad ogni successiva chiamata della funzione `step` queste parti devono essere fatte scendere di una posizione e l'eventuale mossa specificata in input alla funzione dev'essere ignorata;
- se non ci sono pastiglie che stanno cadendo, deve essere generata casualmente una nuova pastiglia.

La funzione `step` deve anche occuparsi del calcolo del punteggio della partita. Se con una mossa vengono rimossi contemporaneamente n virus, il punteggio attribuito è pari a

$$\sum_{i=1}^{n}(100 \cdot 2^i)$$

Se, a seguito di una mossa, si verifica la caduta di parte delle pastiglie e queste causano l'eliminazione di ulteriori n' virus, dovete applicare nuovamente la formula sopra e raddoppiare il valore ottenuto. Se si verificano ulteriori eliminazioni il punteggio viene moltiplicato per 4 e così via. Vediamo un esempio:

- dopo l'esecuzione di una mossa vengono rimossi 4 virus;
- cadono parti di pastiglie che causano l'eliminazione di altri 2 virus;
- cadono ulteriori parti di pastiglie che eliminano altri 3 virus.

Il punteggio totale è dato da:

$$\begin{array}{ll} \sum_{i=1}^{4}(100 \cdot 2^i) & \text{mossa} \\ +2 \cdot \sum_{i=1}^{2}(100 \cdot 2^i) & \text{prima combo} \\ +4 \cdot \sum_{i=1}^{3}(100 \cdot 2^i) & \text{seconda combo} \\ = 3000 + 2 \cdot 600 + 4 \cdot 1400 & \\ = 9800 & \end{array}$$

Nella funzione `vittoria` dovete controllare se la partita è terminata o meno, secondo quanto descritto nella Sezione 1. La funzione ha la seguente firma:

```
enum stato vittoria(struct gioco *gioco)
```

In particolare, la funzione deve restituire:

- `SCONFITTA` se la partita è stata persa;
- `VITTORIA` se la partita è stata vinta;
- `IN_CORSO` se la partita è ancora in corso.

3.2 Generazione di un campo da gioco

Dovete implementare la funzione `riempi_campo` con la seguente firma:

```
void riempi_campo(struct gioco *gioco, int difficolta)
```

Il parametro `difficolta` è un numero intero compreso tra 0 e 15 che determina il numero di virus che devono essere inseriti all'interno dello schema in base alla formula:

$$\#virus = 4 \cdot (\texttt{difficolta} + 1)$$

Il campo generato deve soddisfare i seguenti vincoli:

- le prime 5 righe del campo devono essere vuote;
- non possono essere presenti in fila più di due virus dello stesso colore.

3.3 Lettura di un campo da file

Dovete implementare la funzione `carica_campo` con la seguente firma:

```
void carica_campo(struct gioco *gioco, char *percorso)
```

Il parametro `percorso` è il path del file da cui caricare lo schema. Il file contiene tante righe quanto il campo da gioco. Ogni riga contiene esclusivamente i seguenti caratteri:

- `R` identifica un virus rosso;
- `G` identifica un virus giallo;
- `B` identifica un virus blu;
- lo spazio identifica una cella vuota.

Nel Codice 1 potete vedere il file corrispondente allo schema di Figura 1. Notate che viene utilizzata la convenzione Unix dove il ritorno a capo è codificato dal carattere `\n`: su Windows, invece, il ritorno a capo è codificato dalla coppia di caratteri `\r\n`, quindi se create dei file di test su questa piattaforma assicuratevi di configurare in maniera appropriata il vostro editor di testo. Per semplicità potete assumere che i file rappresentino schemi validi che soddisfano i vincoli descritti nella sezione precedente.

Codice 1: File di input che corrisponde allo schema di Figura 1.

```
G   B  G
G B
RGB    B
RBG
 BG R  R
  RBBRRB
BR GGBBG
R    BGG
 GRBB RB
   G RR
```

4 Compilazione ed esecuzione

La libreria utilizzata per realizzare la componente grafica del gioco e la gestione dell'input è *SDL* versione 2.0[2]. Per installarla su Ubuntu potete usare il seguente comando:

```
sudo apt install libsdl2-dev
```

[2] `https://www.libsdl.org`

Consultate il sito della libreria per informazioni relative all'installazione su altri sistemi.

Per compilare il progetto è sufficiente utilizzare il comando `make`. Se la compilazione ha successo, verrà prodotto un file eseguibile chiamato `drmauro`. Il programma supporta i seguenti argomenti da riga di comando:

- `-s` permette di modificare la velocità di caduta delle pastiglie;
- `-d` permette di specificare il livello di difficoltà;
- `-f` permette di specificare il file da cui caricare lo schema;
- `-h` mostra l'help.

Le opzioni `-d` e `-f` sono mutuamente esclusive. Ad esempio, si può invocare il programma chiedendo di generare uno schema di difficoltà 10 come segue:

```
./drmauro -d 10
```

I tasti da usare per giocare sono:

- le frecce direzionali per spostare la pastiglia;
- i tasti `Z` e `X` per ruotare la pastiglia a sinistra o a destra;
- il tasto `Q` per uscire dal gioco.

5 Elementi di valutazione

Sebbene il progetto sia un lavoro che può essere svolto in gruppo, la valutazione sarà individuale. Oltre alla discussione orale, saranno valutate:

- qualità del codice: oltre alla correttezza, saranno considerate l'eleganza della soluzione implementata, l'utilizzo di indentazione corretta e consistente, la suddivisione in sottoprogrammi, etc;
- qualità della documentazione: codice non commentato non sarà valutato;
- eventuali funzionalità extra.[3]

6 Funzionalità extra

Ci sono un sacco di possibili funzionalità extra da implementare. Ad esempio:

- modificare la gestione dell'input in modo che sia possibile eseguire più di una mossa prima che la pastiglia scenda verso il basso di una posizione;
- usare le apposite funzioni della libreria SDL per acquisire l'input tramite joypad;
- implementare la versione multiplayer del gioco (**difficile!**)

Non modificate le regole di base del gioco spiegate in Sezione 1. Proposte particolarmente interessanti o eleganti potrebbero convincere il Prof. Marin a darvi un bonus all'esame :)

[3] Non sono necessarie per ottenere il punteggio massimo se gli altri requisiti sono soddisfatti.