Progetto “X-Tetris” – Corso IaP

Studente: Maksim Kovalkov

X - Tetris. Il progetto consiste nell’implementazione di una versione modificata del gioco classico Tetris, che includa la possibilità di giocare in due sia contro un altro giocatore che contro un algoritmo “AI”.

1 Struttura del progetto

1.1 Rappresentazioni di dati

Per rappresentare il campo da gioco è stato scelto un array multidimensionale statico di byte, per semplificare la gestione della memoria perlomeno nella parte della gestione della logica del gioco. Sapere che un blocco all’interno del campo occupa 1 `char` permette di ragionare più facilmente sulle operazioni che devono accadere durante il gioco (posizionamento dei blocchi, cancellazione di righe...).

La rappresentazione dei tetramini è anch’essa scelta per maggiore comodità: sono array 4×4 (il rettangolo più piccolo che può contenere tutti i tetramini in tutte le loro rotazioni) di byte contenenti valori 0 e 1 usati per definire la loro forma.

La rappresentazione del *game state* fa uso del concetto di *state machine* per tenere conto delle situazioni distinte e ben definite che si possono verificare durante la partita: si sta scegliendo un tetramino, si sta scegliendo la posizione, stanno venendo cancellate delle righe, ecc..

1.2 Scelte/ipotesi sul gioco

Rappresentazione grafica dei blocchi. Si è scelto di tenere conto nella rappresentazione del campo da gioco anche del tipo di tetramino da cui “proviene” ogni blocco. Questo corrisponde al diverso colore dei blocchi che si può trovare nelle versioni “canoniche” del gioco Tetris, informazione usata dalla componente che gestisce la parte grafica per visualizzare i blocchi con caratteri diversi (come `##`, `@@` etc.), permettendo forse all’utente di percepire meglio la configurazione del campo da gioco.

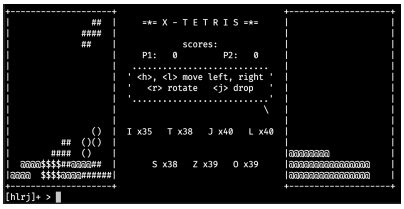

Figura 1: Esempio di svolgimento del gioco: si può vedere come si è scelto di rappresentare i blocchi.

Inoltre, a un singolo blocco corrispondono due caratteri affiancati su una stessa riga: si tratta di nuovo di una scelta estetica motivata dal fatto che di norma negli emulatori di terminale i caratteri sono allungati in verticale, per cui due caratteri affiancati approssimano meglio un quadrato.

Posizionamento blocchi. La selezione della posizione del pezzo è anch’essa ispirata ai giochi di Tetris originali: il pezzo viene spostato di un passo alla volta con comandi “sinistra”/“destra”/“ruota”; e fin dall’inizio fa parte del campo da gioco e non deve sovrapporsi a blocchi già esistenti.

1.3 Algoritmo per l'avversario

L'algoritmo usato per valutare le mosse è costituito da due parti: `heuristic`, una funzione che assegna un punteggio a una particolare configurazione del campo da gioco, basandosi su un'insieme di fattori quali l'altezza totale, la presenza di buchi ecc.; e un'altra funzione ricorsiva che prova tutti i possibili posizionamenti del pezzo, calcolando il punteggio di ognuno in base anche alle configurazioni che potrà assumere nelle N mosse future, e scegliendo quello con il punteggio più alto.

1.4 Approccio alla gestione dello stato

Gestione memoria e *lifetime*. L'uso della memoria dinamica è relativamente limitato, e poiché non c'è un modo di recuperare la normale esecuzione nel caso di un'allocazione che fallisce, il programma esce immediatamente se è a corto di memoria. Le componenti che usano memoria dinamica forniscono delle funzioni "costruttore" e "destruttore", da essere chiamate esplicitamente, per gestire l'acquisizione e il rilascio della memoria.

Suddivisione delle parti. Il codice è suddiviso in moduli responsabili ognuno di un compito specifico e (con alcune eccezioni) indipendenti dai dettagli di implementazione degli altri, con lo scopo di permettere una migliore gestione del sistema e sperabilmente ridurre la quantità di informazioni da tenere a mente per ragionare su ognuna delle singole parti. La suddivisione è la seguente:

- `tetris` – Gestione della logica del gioco
- `iohandler` – Gestione dell'input e dell'output
- `opponentai` – Strategia dell'avversario

Di seguito uno schema che rappresenta, ad alto livello, il flusso delle informazioni all'interno del programma.

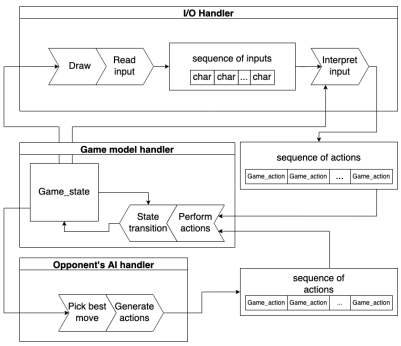

Figura 2: Suddivisione logica del progetto.

2 Problemi riscontrati

Interazione tra componenti. Si potrebbe sostenere che in alcune parti del programma, specie quelle che riguardano la comunicazione tra le varie componenti, si stia svolgendo del lavoro ridondante e che in generale il livello di complessità sia più alto del necessario. Soprattutto, usare le sequenze di `enum Game_action` per comunicare al modello di gioco la mossa scelta dall'IA dell'avversario potrebbe sembrare inefficiente: si copia l'intero campo da gioco per decidere la mossa migliore invece di operare sul campo originale; e dopo aver deciso la mossa si genera una sequenza di input che

manderebbe il pezzo nella posizione corretta, invece di comunicare direttamente le coordinate e la configurazione del tetramino. Si tratta di un *trade-off* che presenta i suoi vantaggi e svantaggi: da un lato, si applicano le solite considerazioni riguardanti la maggiore complessità e la ridondanza dello stesso lavoro eseguito più volte; dall'altro lato si ha un'interfaccia unica usata sia per interpretare gli input da tastiera che per eseguire la mossa scelta dal computer (in particolare il controllo della validità delle mosse si applica a entrambe) e ciò permette di evitare che il sistema entri in uno stato non previsto, causando bug sottili e difficili da rintracciare.

In ogni caso, ho cercato di fare in modo che anche le parti più complesse siano perlomeno abbastanza circoscritte e isolate: ovvero che si possa capire come funziona *solo* la lettura da tastiera, oppure *solo* le transizioni di stato del gioco che possono verificarsi, ecc.

Differenze tra single- e multiplayer. Ho deciso di utilizzare la stessa struttura dati per rappresentare sia il gioco a giocatore singolo che quello a due giocatori: nel primo caso, semplicemente, alcuni campi della *struct* non vengono utilizzati o non cambiano mai valore (come `current_player` o `board[1]`). Si tratta di nuovo di un *trade-off*, poiché a costo di utilizzare più memoria e di richiedere qualche *branch* in più nelle funzioni che gestiscono il gioco, si unifica entrambe le modalità evitando di introdurre casi particolari e di duplicare il codice, che potrebbe portare a errori di implementazione.

Variabili globali. Ho cercato di evitare il più possibile l'utilizzo di variabili globali, in quanto creano dipendenze implicite tra componenti del sistema più disparate, aumentando notevolmente la difficoltà di fare *debugging* e di apportare modifiche.

Nonostante siano presenti delle variabili globali, si può vedere che in tutti i casi i dati su cui le funzioni operano, incapsulati in opportune strutture, sono determinati esclusivamente dai parametri che ad esse vengono passate.

3 Altri commenti

Naming convention. La convenzione adottata è simile a quella preferita da Bjarne Stroustrup (creatore di C++): si usa `snake_case` ovunque possibile, preferendo però di cominciare con una maiuscola nei nomi dei tipi definiti dal programmatore (`Upper_snake_case`); si usano tutte maiuscole solo per macro introdotte da direttive `#define`. All'inizio, avevo valutato di adottare invece la convenzione di aggiungere il suffisso `_t` ai tipi, ma ho deciso diversamente per non incorrere in potenziali problemi di compatibilità nel caso dovessi includere altri header, poiché l'intero 'namespace' `*_t` è riservato secondo lo standard POSIX.

Controllo versioni. Qualunque progetto non del tutto banale può beneficare di un sistema di controllo versioni: ho utilizzato `git`, e pur non avvalendomi di tutte le funzionalità offerte da questo strumento, ho trovato molto utile la modalità di sviluppo generalmente più ordinata derivante dalla necessità di scrivere un messaggio per ogni commit.

Ispirazioni. Per alcuni aspetti ho usato, come esempio per conoscere le tecniche standard di organizzazione dei progetti in linguaggio C, il progetto `st` (*st.suckless.org*), un emulatore di terminale.