Relazione Tecnica: Progetto Frogger (Clone)

1. Obiettivi del Progetto

L'obiettivo era la riproduzione fedele delle meccaniche del classico arcade "Frogger" (1981) in ambiente **Linux**, utilizzando il linguaggio **C**. Il focus principale non è stato solo la logica di gioco, ma la gestione di un sistema **Real-Time concorrente**, dove ogni entità (veicoli, tronchi, proiettili) agisce indipendentemente, simulando un sistema distribuito su scala locale.

2. Architettura del Sistema

Il software adotta un'architettura a **micro-processi** basata sulla system call `fork()`. A differenza di un approccio multi-thread, qui ogni entità di gioco è un processo distinto con il proprio spazio di memoria, garantendo robustezza e isolamento.

Il sistema è composto da:

- **Processo Madre (Game Engine/Renderer):** Gestito dalla funzione `gameArea()`. È il "consumatore". Si occupa di:
  - Disegnare l'interfaccia (UI) frame per frame.
  - Gestire la logica delle collisioni.
  - Riprodurre l'audio (tramite SDL2).
- **Processi Figli (Produttori):**
  - **Veicoli:** Vengono generati `nlanes * nVehicles` processi indipendenti. Ogni veicolo calcola la propria posizione e la scrive sulla pipe.
  - **Tronchi:** Ogni tronco è un processo autonomo che gestisce il proprio movimento nel fiume.
  - **Rana (Input Handler):** Un processo dedicato (`fun_frog`) che ascolta l'input dell'utente e invia i comandi al motore grafico.
  - **Proiettili:** Gestione dinamica dei processi. Quando si spara, viene effettuata una `fork()` on-demand per gestire la traiettoria del proiettile.

3. Comunicazione e Sincronizzazione (IPC)

La comunicazione tra i processi avviene tramite **Pipe Unnamed** (canali di comunicazione unidirezionali). Il progetto utilizza tre canali principali definiti nel `main.c`:

1. `pipevec`: Canale per i dati dei veicoli (Automobili/Camion).
2. `pipefrog`: Canale per i comandi di movimento della rana.
3. `pipeobj`: Canale condiviso per oggetti generici (Tronchi e Proiettili).

Gestione della Concorrenza: Per evitare che il motore grafico si blocchi in attesa di dati da un processo lento, le pipe sono state configurate in modalità **Non-Bloccante** (`O_NONBLOCK` tramite `fcntl`).

- *Tecnica:* Il motore grafico cicla continuamente (`while(!collision)`). Se una `read()` sulla pipe restituisce dati, aggiorna la posizione dell'oggetto; se non c'è nulla (return -1), continua il rendering mantenendo fluido il gioco (evitando freeze dell'interfaccia).

4. Stack Tecnologico e Librerie

- **Linguaggio:** C standard (C99/GNU).
- **Gestione Processi:** Librerie di sistema UNIX (`unistd.h`, `sys/types.h`, `sys/wait.h`) per `fork`, `pipe`, `kill`.
- **Interfaccia Grafica (TUI):** Libreria **ncurses** (`ncurses.h`).
    - Utilizzata per il rendering testuale avanzato (gestione finestre `newwin`, colori `COLOR_PAIR`, e posizionamento cursore efficiente).
    - Double Buffering simulato tramite `wrefresh()` per evitare lo sfarfallio.
- **Audio:** Libreria **SDL2** con estensione **SDL_mixer**.
    - Utilizzata per gestire musica di sottofondo e effetti sonori (spari, collisioni, salti) in parallelo alla logica di terminale.

5. Sfide Tecniche Risolte

Gestione delle Collisioni

La logica di collisione è centralizzata in `gameArea`. Invece di usare mutex complessi, il motore grafico possiede la "verità assoluta" sullo stato del gioco.

- *Esempio:* Quando la rana è su un tronco, il sistema aggancia le coordinate della rana a quelle del tronco (`frog.x += frog_on_log`), simulando la fisica del trascinamento.

Gestione delle Risorse (Processi Zombie)

Essendo un'applicazione multi-processo, c'è il rischio di creare processi orfani o zombie.

- *Soluzione:* Nel `main.c`, alla chiusura del gioco (Game Over o Vittoria), viene eseguito un ciclo di pulizia che invia segnali di terminazione (`kill(pid, 1)`) a tutti i PID dei figli registrati negli array `pidv[]` e `pidlogs[]`, garantendo il rilascio corretto delle risorse al sistema operativo.

Logica "Nemici" Dinamica

Il sistema implementa una logica avanzata per i nemici sui tronchi.

- Un array `log_map[]` mappa lo stato di ogni tronco (libero, occupato da nemico, occupato da rana).
- I nemici hanno un timer di fuoco autonomo (`start_timer_bullets`) e possono generare nuovi processi proiettile dinamicamente, aumentando la complessità algoritmica rispetto al Frogger classico.