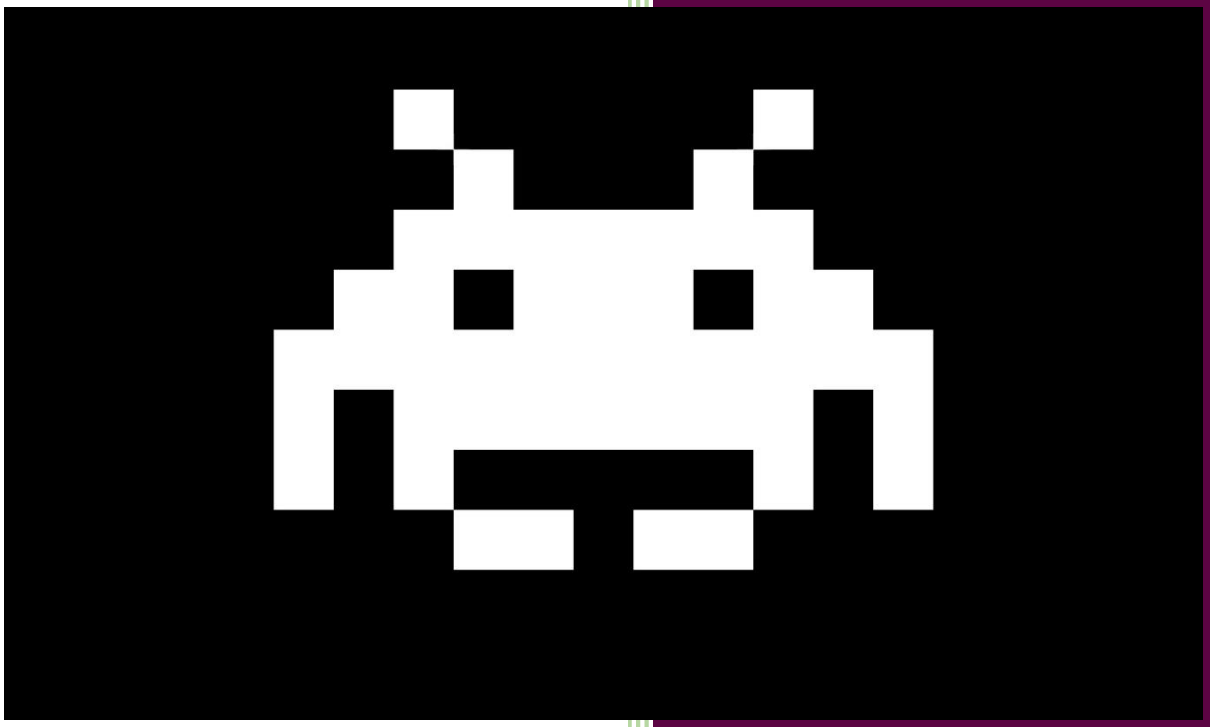
SISTEMI OPERATIVI

SPACE DEFENDERS



MATTEO TUZI 60/61/66042
DAVIDE UCCHEDDU 60/61/66026
11/01/2022

Sommario

Analisi funzionale:

La consegna richiede lo sviluppo di un programma ispirato ai famosi giochi cabinati *Space Invaders* e *Defenders*.

Il progetto deve prevedere l'implementazione del menù, della difficoltà (facile, difficile, custom) e della fase di gioco.

Attraverso il menù, sarà possibile giocare, scegliere la difficoltà o uscire dal gioco.

Modifiche progettuali:

- I proiettili della navicella alleata rimbalzano
- La quantità di nemici è personalizzabile dal menù
- La navicella alleata ha più vite
- La difficoltà è personalizzabile

Queste modifiche sono atte a rendere il gioco più fluido, in particolare la meccanica che gestisce il rimbalzo dei proiettili rende il gioco più divertente e dinamico da giocare.

Funzionamento del gioco:

Una navicella alleata controllata dall'utente può muoversi solo verticalmente nella parte sinistra dello schermo. Parte dal centro e con vite variabili in base alla modalità scelta dal giocatore: 3 vite nel caso di difficoltà facile, una se difficile (nel caso sia custom, se il numero di navicelle è superiore a 3, viene impostata automaticamente ad una sola vita altrimenti a 3).

La navicella spara due proiettili in diagonale, i quali vengono generati alla pressione del tasto "spacebar"; quando i proiettili raggiungono i bordi superiori e inferiori rimbalzano, cambiando direzione; i proiettili scompaiono se colpiscono una navicella nemica o se toccano il bordo destro.

Alla destra dello schermo sono presenti N navicelle nemiche, dove N è 3 nel caso di difficoltà facile, 9 in caso di difficoltà difficile, altrimenti personalizzabile fino ad un massimo di 12 navicelle, 3 per ogni colonna.

Le navicelle si muovono in modo sincrono, in particolare ogni 5 spostamenti (prima verso il basso e poi verso l'alto) vanno a sinistra di una posizione. Inoltre, ogni volta che *rand() % 10* è uguale a 0, sparano una bomba che si muove orizzontalmente fino a toccare il bordo sinistro o la navicella alleata.

Quando una navicella nemica viene colpita, genera una navicella di secondo livello, che è lo stesso oggetto, con la differenza che ai bordi dello sprite ci sono delle “x” che danno l’idea di essere una navicella piccola collegata ad una più grande; richiede 2 colpi per essere uccisa; quindi, la navicella in totale ha 3 vite.

Condizioni di fine del gioco:

Il gioco finisce se:

- Una navicella nemica arriva nell’area della navicella alleata (sconfitta)
- Una bomba collide n volte la navicella alleata, dove n indica il numero di vite (sconfitta)
- Tutte le navicelle nemiche sono distrutte (vittoria).

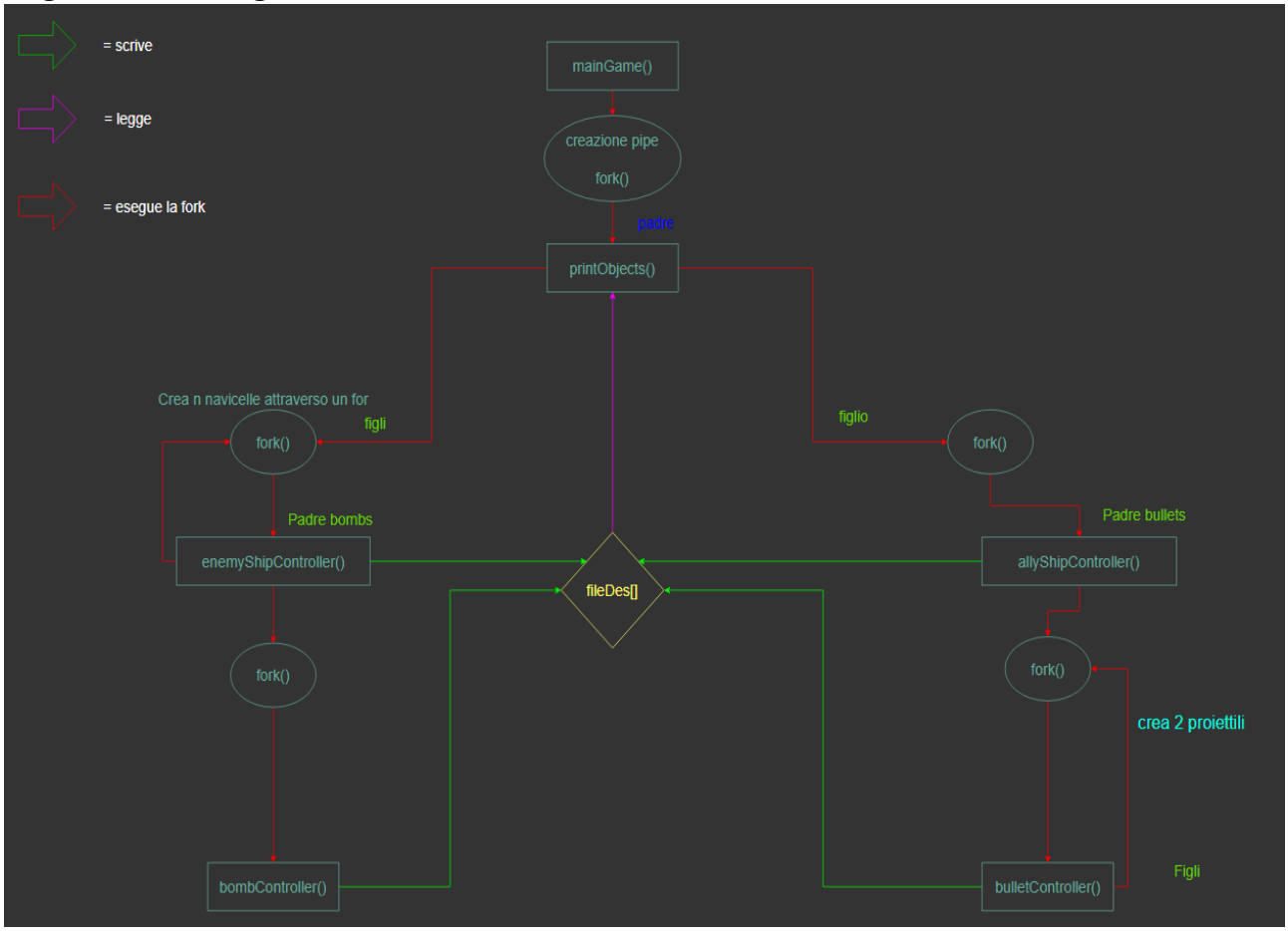
Analisi tecnica versione processi:

La versione processi è stata implementata attraverso l’utilizzo delle pipe, un canale di comunicazione unidirezionale che permette a due processi appartenenti allo stesso PPID di comunicare tra di loro. Il nostro progetto è stato strutturato in maniera tale che la funzione “printObjects” utilizzi la system call read per leggere gli oggetti (consumatore) che scrivono i figli attraverso le procedure (produttori):

- void allyShipController(WINDOW* win, Point p, int pipeOut), il cui scopo è quello di gestire il movimento verticale della navicella e lo sparo; quando l’utente preme “spacebar” genera due proiettili e setta due variabili (isBulletShot e canShoot) a true, implementate al fine di permettere all’utente di sparare solo due proiettili alla volta; tali variabili vengono settate a false solo nel caso in cui il processo proiettile termini. La procedura gestisce la pulizia dei processi zombie attraverso una waitpid con option “WNOHANG”. Tutto l’algoritmo è inserito all’interno di un while(true), all’interno del quale è presente una write per la scrittura nella pipe della posizione della navicella.
- void bulletController(WINDOW* win, Point posShip, Direction direction, int pipeOut), il cui scopo è quello di scrivere nella pipe la posizione del proiettile e di gestire la collisione con i bordi; quando raggiunge il bordo destro, il processo corrispondente viene terminato; quando raggiunge i bordi superiore e inferiore, il proiettile cambia direzione, da “DOWN_DIRECTION” ad “UP_DIRECTION” o viceversa.

- void enemyShipController(WINDOW* win, Point p, int pipeOut, int idNumber, Difficulty difficultyMode), il cui scopo è quello di gestire lo spostamento della navicella nemica e la sua scrittura nella pipe; quando *rand()%10* è uguale a 0, viene effettuata una fork per generare il processo bomba. Come per la navicella alleata, anche quella nemica gestisce la pulizia dei processi zombie con la waitpid. In modo simile alla funzione allyShipController viene usata la flag generateBomb per evitare che siano presenti più di una bomba per alieno vivo.
- void bombController(WINDOW* win, Point p, Point posAlien, int pipeOut), gestisce lo spostamento della bomba: in particolare ogni 50 ms si sposta verso sinistra di una posizione e scrive nella pipe. Quando la bomba raggiunge il bordo sinistro o collide con la navicella alleata, il processo corrispondente viene terminato.

Una rappresentazione grafica di quanto scritto sopra è visionabile attraverso la seguente immagine:



Analisi tecnica versione thread

La versione thread prevede la creazione dei thread attraverso la funzione pthread_create.

Abbiamo deciso di sfruttare le variabili globali per la gestione delle varie entità, in quanto la memoria è condivisa tra i vari thread, ma questo genera un problema di sincronizzazione, per questo motivo entra in gioco la necessità di gestire le sezioni critiche.

Le sezioni critiche sono delle particolari sezioni di codice che modificano non atomicamente una variabile condivisa tra più thread.

Un esempio di sezione critica che è possibile trovare nel nostro progetto sono la modifica delle coordinate da parte dei thread e della loro stampa nella funzione di stampa (printObjects).

Durante il corso abbiamo studiato vari metodi di sincronizzazione, dai lock mutex ai monitor; per questo progetto abbiamo voluto implementare un solo lock mutex che gestisse la sincronizzazione tra i thread.

A tutti i thread, dopo la creazione, viene applicata la funzione di detach la quale ci permette di liberare le risorse dei thread quando terminano la loro esecuzione.

Funzioni principali:

- void mainGame(WINDOW* win, Difficulty difficultyMode): Questa procedura gestisce l'avvio del gioco. Crea i thread della navicella alleata e delle N navicelle nemiche scelte sulla base del contenuto della variabile "difficultyMode.numAliens". La procedura inizializza inoltre tutti i buffer, le variabili di stato e le variabili globali che ci servono per gestire la sincronizzazione tra i thread. Infine, una volta che viene terminata la funzione di stampa, si occupa di deallocare tutte le risorse usate globalmente e di terminare eventuali thread ancora in esecuzione.
- void* allyShipController(), questa funzione viene assegnata ad un thread, ha lo scopo di gestire lo spostamento della navicella alleata e di gestire lo sparo. Al suo interno viene utilizzato un mutex durante l'inizializzazione dell'entità per evitare inconsistenza tra i vari thread. Quando l'utente preme "spacebar" vengono creati due thread, i quali vengono associati alla funzione bulletController(). Anche qui viene implementata la gestione con flag di controllo per evitare la creazione di proiettili multipli (come nella versione con i processi).

- void* bulletController(void* directionV), questa funzione si occupa di gestire le coordinate dei proiettili. Viene passato come argomento una direzione, che viene utilizzata come indice per accedere al buffer condiviso “bullets”. Ogni aggiornamento delle coordinate viene gestito attraverso il mutex lock e unlock.
- void* enemyShipController(void* idNumberT), questa funzione viene assegnata ad n thread, dove n è il numero di navicelle aliene. idNumberT è un identificatore che ci serve per accedere alla cella specifica del buffer globale. Come per la navicella alleata, l’inizializzazione è gestita all’interno di un mutex per evitare inconsistenza tra gli altri thread. Lo spostamento dell’alieno è gestito come per i processi, con la differenza che ogni modifica di posizione è inserita all’interno di un mutex lock e unlock. Quando rand()%10 è uguale a 0 e non sono presenti altri oggetti bomba attivi con medesimo idNumberT, viene creato un thread, associato alla funzione bombController(). Come per i bullets, viene chiamata la funzione pthread_detach per il thread bomba.
- void* bombController(void* idNumberT), questa funzione viene assegnata al thread che gestisce la bomba. Viene passato idNumberT che identifica l’alieno che ha sparato la bomba. Come abbiamo fatto per le altre entità, anche per la bomba effettuiamo un mutex lock e unlock quando inizializziamo e quando modifichiamo le coordinate.
- EndGame printObject(), questa funzione gestisce la stampa e le collisioni dei vari oggetti gestiti da thread. Al suo interno è presente un ciclo che verifica che la variabile “gameStatus” sia impostata a “CONTINUE”, se è vero significa che il gioco deve continuare, altrimenti verrà impostata a “VICTORY” nel caso di vittoria dell’utente, e a “DEFEAT” nel caso di sconfitta dell’utente, restituendo al chiamante tale variabile.
- void checkCollision(), questa procedura verifica le collisioni e quindi gestisce i casi di vittoria e sconfitta. Nel caso di collisione tra alieno e proiettile, termina il thread del proiettile, modifica la vita dell’alieno decrementandola di uno e, se quest’ultima arriva a 0, termina anche il thread dell’alieno e imposta il suo stato a “OBJ_DEAD”. Nel caso di collisione tra navicella alleata e bomba, effettua la exit sulla bomba e modifica la vita della navicella alleata decrementandola di uno. Nel caso di collisione tra navicella alleata e navicella aliena, viene impostata gameStatus a “DEFEAT”.

- EndGame isGameOver(int nAlienAlive), questa funzione verifica se la partita deve finire o meno, restituendo “VICTORY”, “DEFEAT” o “CONTINUE” dopo un controllo sul numero di nemici e sulla vita della navicella.

Di seguito, una rappresentazione grafica di quanto descritto sopra: