SPACE DEFENDER

Di
Marco Monni
Giovanni Antonio Sprega

VERSIONE PROCESSI

I file

Per la stesura di questo progetto abbiamo deciso di distribuire l'implementazione su più file, ognuno dei quali racchiude uno specifico gruppo di funzioni.

main.c: Contiene la funzione principale del gioco e si occupa:

- dell'inizializzazione della pipe (versione processi);
- della visualizzazione della schermata iniziale del gioco;
- della creazione dei processi relativi alla nave del giocatore;
- della creazione del processo relativo alla flotta nemica;
- dell'invocazione della funzione contenente il loop di gioco
- dell'inizializzazione e della chiusura della finestra di gioco

global.h: Contiene tutte le include delle librerie utilizzate, tutte le macro, la struttura utilizzata per la rappresentazione degli oggetti di gioco, tutti i prototipi delle funzioni suddivisi per file di appartenenza.

player.c: Il source che gestisce gli oggetti del giocatore. Contiene le funzioni:

- playerShip
- playerShotInit
- shot

enemies.c: Il source che gestisce gli oggetti nemici. Contiene le funzioni:

- fleetEnlister
- enemyShip
- enemyShot

gameplay.c: Il source che gestisce il gioco. Contiene le sprite delle astronavi amica e nemiche e la sola funzione:

- gameArea

global.c: Contiene:

- La variabile globale rocketFrame
- Le funzioni:
    - printSprite
    - deleteSprite
    - isRocket
    - checkCollision
    - resetItem
    - timeTravelEnemyRocket
    - rocketAnimation
    - printLives

start-over.c: Il source che si occupa delle schermate di inizio e fine gioco. Contiene:

- Le matrici Ascii-art per la stampa del titolo del gioco
- Le funzioni:
    - countdownPrint
    - startGame
    - gameOver

La Logica:

La logica del programma segue un flusso piuttosto elementare. Riprendendo quanto menzionato nella sezione precedente alla voce riguardante il file main.c, possiamo seguire passo a passo l'implementazione del gioco.

1. Operazioni preliminari:

La prima operazione che viene svolta è l'inizializzazione del seed random, che verrà utilizzato più avanti per dei particolari scopi, e della finestra di gioco. Gioco che inizia ora con la visualizzazione di una schermata di presentazione con un titolo che, ad ogni secondo per tre secondi, cambia colore. Trascorsi questi tre secondi, la prima cosa che avviene è l'inizializzazione della pipe che si occuperà del dialogo tra i processi.

Dopodiché vengono creati due processi figli. Il primo è il processo relativo al giocatore, mentre il secondo quello dei nemici. In entrambi i processi, la pipe viene chiusa in lettura e lasciata *aperta in*

scrittura, in quanto ciò che faranno sarà comunicare *verso* il processo padre, quindi scrivendo dati che verranno da esso letti successivamente.

A questo punto, sempre dal main, viene invocata la funzione che si occuperà di gestire il *loop di gioco*. Ad essa, al contrario di quanto avvenuto nei due processi di cui sopra, passeremo la pipe *aperta in lettura* e chiusa in scrittura, in quanto il loop di gioco dovrà occuparsi dell'elaborazione dei dati *ricevuti* dai vari processi relativi agli oggetti di gioco.

2. La struttura Object

La struttura, di cui si può trovare la dichiarazione nel file *global.h*, è fondamentale per il programma. Essa, infatti, si occupa di contenere le informazioni di ogni singolo oggetto che verrà creato (navi e razzi). I suoi parametri sono:

- Identifier : fondamentale per il riconoscimento del singolo oggetto (rif. righe 26-31 global.h)
- X e Y : le coordinate dell'oggetto
- Lives : il numero delle vite rimaste all'oggetto (utilizzato per le navi)
- Pid : il pid del processo che si occupa del singolo oggetto, per gestirne la terminazione
- Serial : il numero di serie dell'oggetto, utilizzabile come indice.

3. Il Giocatore

3.1 Funzione playerShip

La prima funzione che analizziamo è la playerShip che, come anticipato, riceve in ingresso la pipe aperta in scrittura e chiusa in lettura. La funzione si occupa dell'inizializzazione, della gestione del movimento e della gestione dello sparo, il tutto attraverso un loop virtualmente infinito il quale, al termine di ogni iterazione, scrive nella pipe i dati aggiornati del giocatore. Il movimento consentito è verticale, e ogni pressione del tasto *spazio* genera un controllo che porta ad eseguire lo sparo di due proiettili diagonali.

3.2. La funzione playerShotInit

La funzione viene invocata dalla *playerShip* e riceve in ingresso la pipe, le coordinate attuali della nave del giocatore e il serial. All'interno di questa funzione vengono creati due processi, uno per ogni direzione di sparo, e ognuno di questi due processi invoca la funzione *shot*.

3.3. La funzione Shot

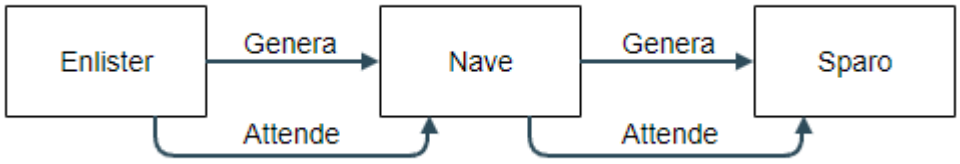
La funzione viene (doppiamente) invocata dalla funzione *playerShotInit* e riceve in ingresso gli stessi parametri e, in aggiunta, il parametro *direction*. Questo parametro è fondamentale per differenziare il tipo di sparo (verso l'alto o verso il basso) e, a seconda di esso, il parametro *identifier* viene inizializzato di conseguenza. Un loop, ad ogni iterazione e con un ritardo di 30000 microsecondi, si occupa di aggiornare la posizione del razzo e di scrivere i dati aggiornati sulla pipe. I proiettili

rimbalzano e cambiano direzione ad ogni impatto con il limite superiore o inferiore dell'area di gioco.

4. Il Nemico

4.1. La funzione fleetEnlister

La funzione *fleetEnlister* ha, come suggerisce il nome, lo scopo di generare la flotta delle navi nemiche. Un ciclo for, avvalendosi della macro *ENEMIES* per conoscere il numero di nemici, creerà tanti processi quante sono le navi da generare. Ad ogni iterazione viene creato un nuovo processo, inizializzata una nuova nave, chiamata la funzione che si occuperà di gestire il movimento della nave, e generate le coordinate della nave *successiva*. Una volta conclusa la generazione, il processo attenderà la terminazione di tutti i processi figli (e nipoti), in un effetto a cascata dove i processi delle navi attendono la terminazione dei processi degli spari e il processo che ha chiamato la fleetEnlister attende la terminazione dei processi delle navi, come in figura.



4.2. La funzione enemyShip

La funzione *enemyShip* ha il medesimo scopo della *playerShip*, ma con qualche differenza. In primis, il movimento non è frutto di un input da parte dell'utente ma avremo un movimento verticale con un singolo movimento orizzontale verso la nave del giocatore e dunque un cambio direzione (su – giù). Il movimento è ottenuto grazie alla variabile *flag*, settata di default al valore *VERTICAL*, ma che prende il valore della macro *HORIZONTAL* ogni volta che raggiunge il limite superiore o inferiore dell'area di gioco. La successiva valutazione della *flag* ci porta dunque al *case HORIZONTAL*, il quale gestisce prima lo spostamento orizzontale, poi il cambio direzione e il ripristino del valore di default della *flag*. La velocità di movimento è variabile, infatti viene incrementata del 20% ogni sette secondi.

Per quanto riguarda lo sparo, la funzione valuta costantemente il tempo trascorso tra uno sparo e l'altro proveniente dalla stessa nave. La quantità di tempo necessaria è pari al tempo massimo di percorrenza, in secondi, di tutta l'area di gioco di un proiettile generico, quantità di tempo che viene calcolata da una semplice funzione che tratteremo più avanti: *timeTravelEnemyRocket*. Per ora basti sapere che il valore restituito dalla funzione viene utilizzato a tale scopo. Questa implementazione fa sì che ogni nave non possa sparare un nuovo razzo finché il precedente non ha concluso il suo ciclo vitale (uscita dall'area di gioco o impatto con la nave del giocatore).

4.3. La funzione enemyShot

La funzione *enemyShot* è del tutto simile alla sua controparte relativa allo sparo del giocatore. La differenza sostanziale è che i proiettili viaggeranno orizzontalmente in linea retta, senza alcun tipo di deviazione.

5. Il gameplay

5.1. Le strutture dati

Il programma utilizza prevalentemente dei semplici array statici per l'immagazzinamento dei dati degli oggetti di gioco:

- data : variabile che viene utilizzata per estrarre i dati dalla pipe
- player : array che contiene i dati della nave del giocatore
- enemy : array che contiene i dati delle navi dei nemici
- rocketUp : array che contiene i dati degli spari del giocatore (alti)
- rocketDown : array che contiene i dati degli spari del giocatore (bassi)
- enemyRockets: array che contiene i dati degli spari dei nemici

Gli array prendono dimensioni diverse a seconda della destinazione d'uso. Ad esempio, ovviamente, l'array *enemy* avrà dimensione pari al numero dei nemici, così come l'array *enemyRockets*, visto il discorso del paragrafo 4.2 riguardo l'autorizzazione allo sparo delle navi nemiche.

5.2. Il loop di gioco

5.2.1. Prima del loop

Prima di entrare nel loop il programma inizializza alcune variabili.

- score : utilizzata per l'assegnazione o la decurtazione dei punti al giocatore
- enemyCounter : il contatore dei nemici ancora in vita
- gameResult : una flag che verrà poi passata alla funzione *gameOver*

Successivamente vengono assegnati i valori ai parametri *lives e pid* del player e di tutti i membri dell'array *enemy*. A questo punto, dopo aver iniziato anche la riproduzione del sottofondo musicale, il programma può entrare nel loop di gioco.

5.2.2. Ingresso nel loop

La prima, fondamentale, operazione che viene eseguita è l'estrazione di un dato dalla pipe e il suo salvataggio nella variabile apposita: *data*. Quindi viene assegnato il suo *serial* alla variabile *id*.

Questo si rivelerà utile successivamente, per l'assegnamento degli oggetti nemici e razzi nelle giuste posizioni degli array.

5.2.3. Il primo switch: assegnamento

Il primo *switch* si preoccupa proprio di questo, l'assegnazione del dato alla giusta variabile di destinazione. A tal proposito, il case *player* si limita ad assegnare il pid alla prima lettura e le coordinate in tutte le letture, prima compresa. Stessa cosa avviene nel case *enemy*, dove in aggiunta viene anche verificata la posizione della nave nemica rispetto al giocatore. Qualora la nave avesse raggiunto il giocatore, la partita terminerebbe con la *sconfitta* del giocatore per mezzo dell'azzeramento diretto delle sue vite. Nei due case non vengono assegnate anche le vite degli oggetti: esse sono state inizializzate in questa stessa funzione per evitare una sovrascrittura dei dati che avrebbe reso le navi, sia nemiche che del giocatore, immuni ai razzi. Passando ai case dei razzi, *rocket_up*, *rocket_down* ed *enemy_rocket*, le gestioni sono tra loro identiche. In questi case, l'assegnazione di *data* è completa: non abbiamo infatti alcun problema relativo alla gestione delle vite, in quanto il ciclo vitale dei razzi non ne è condizionato. Un successivo controllo si assicura che il razzo si trovi ancora all'interno dei confini dell'area di gioco: se così non fosse uccide il processo interessato sfruttando il campo *pid* della struttura e resetta i valori del dato contenuto nell'array. Lo scopo del *reset* diverrà più chiaro successivamente: sarà importante per la terminazione di eventuali processi ancora in corso alla fine del gioco.

5.2.4. Il secondo switch: eventi di gioco

Il secondo *switch* è il cuore del programma: si occupa infatti delle stampe degli elementi sullo schermo e dei controlli fondamentali del gioco. Il case *player* produce esclusivamente la stampa della sprite del giocatore. Il case *enemy* è analogo al case precedente, ma si presenta una scelta sulla sprite da stampare, dipendente dalle vite residue della nave. Tre vite corrispondono ad una nave di primo livello, due vite ad una nave di secondo livello, una vita ad una nave di secondo livello danneggiata. Così come nello switch precedente, anche qui i case dei razzi sono piuttosto simili l'un l'altro. I case *rocket_up* e *rocket_down* sono caratterizzati da un ciclo for che si occupa di scorrere l'array *enemy* nella sua interezza. Ad ogni iterazione viene valutata l'eventuale collisione tra il razzo e la nave contenuta nella posizione dell'array *enemy* data dall'indice del ciclo for, mediante la funzione *checkCollision*. Se la collisione è avvenuta, il processo del razzo viene ucciso e i suoi dati resettati come abbiamo già visto precedentemente. Inoltre, le vite della nave colpita vengono decrementate di un'unità. A questo punto avviene la valutazione delle vite residue dell'oggetto colpito.

Se il valore è pari a zero attende eventuali processi figli: ciò è reso possibile dal fatto che l'inizializzazione dei razzi nemici prevede che il serial del razzo generato sia lo stesso della nave che l'ha generato. Fatto ciò, il processo della nave colpita può essere ucciso e gli oggetti interessati dall'evento cancellati dall'area di gioco, il punteggio del giocatore viene incrementato di 300 punti ed *enemyCounter* ridotto di un'unità.

Se il valore è pari a uno oppure a due, viene cambiata la sprite visualizzata e il punteggio del giocatore viene incrementato di 100 punti.

Il case *enemy_rocket* non necessita di alcun ciclo for, in quanto la collisione può avvenire con la sola nave del giocatore. Se la collisione è avvenuta, il processo del razzo viene ucciso, il carattere eliminato, e i dati del razzo resettati. A questo punto si decrementano le vite del giocatore di un'unità e vengono decurtati 500 punti al giocatore.

5.2.5. La fine del loop

Le ultime operazioni eseguite dal ciclo sono la stampa a schermo delle vite del giocatore mediante la funzione *printLives* e la stampa del punteggio, e l'eventuale riavvio del sottofondo musicale. Viene quindi controllato che esistano ancora navi nemiche attive. Se così non fosse, o se le vite del giocatore fossero esaurite, il loop di gioco termina.

5.2.6. Terminazione dei processi

Usciti dal processo, diventa fondamentale terminare tutti i processi relativi agli oggetti di gioco ancora in vita.

Un primo ciclo, ad ogni iterazione, scorre gli array *enemyRockets* ed *enemy*, verificando che i processi siano ancora attivi e uccidendo quindi prima il processo del razzo e poi quello della nave nella posizione data dall'indice. L'ordine in questo caso è molto importante, dal momento che uccidendo prima il processo della nave lasceremmo il processo del razzo senza un padre, quindi in status zombie.

Un secondo ciclo uccide i processi dei razzi del giocatore ed infine viene ucciso il processo della nave del giocatore. Si noti l'applicazione dello stesso principio adottato in precedenza.

Oltre ai processi, viene anche terminata la riproduzione del sottofondo musicale. Successivi suoni verranno riprodotti dalla funzione *gameOver*, ora invocata.

6. Gamestart & Gameover

6.1. La schermata iniziale

La schermata iniziale è piuttosto semplice da spiegare. La funzione *startGame* è richiamata dal *main* e altro non fa se non stampare a schermo il titolo del gioco ed occuparsi del countdown dopo il quale il gioco avrà effettivamente inizio. Il countdown è scandito dalla riproduzione di un classico "bip" ad ogni secondo, dal cambio dei colori del titolo e da una stampa al centro della "stella" che informa l'utente, in numeri romani, dei secondi rimanenti prima dell'inizio del gioco.

6.2. La schermata finale

La funzione *gameOver* svolge il compito di rappresentare la schermata finale del gioco. Un controllo iniziale riproduce la musica adeguata a seconda che il giocatore abbia ottenuto una vittoria o subito una sconfitta. Un ciclo for, invece, si occupa di scandire il tempo di visualizzazione della schermata, che è pari a 15 secondi. In caso di vittoria viene visualizzato un messaggio positivo, con uno sfondo di stelle gialle, mentre in caso di

sconfitta abbiamo un messaggio negativo e uno sfondo di “ufo” rossi. In entrambi i casi verrà visualizzato comunque il punteggio ottenuto dal giocatore. Al termine dei 15 secondi interrompe eventuali riproduzioni ancora in corso della musica finale e il controllo passa nuovamente al *main*. L’ultima operazione che viene svolta è la chiusura della finestra di gioco e dunque il ritorno al terminale standard.

7. Funzioni utili

7.1 Le stampe

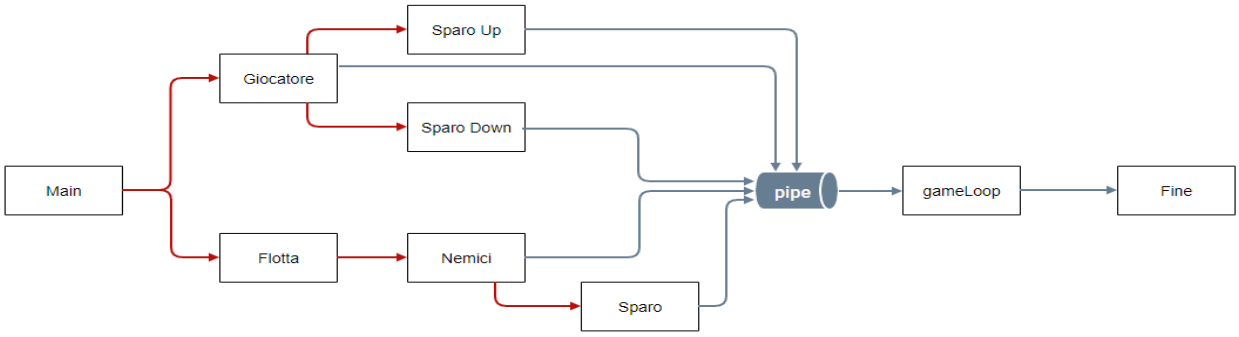
Le funzioni di stampa e cancellazione sono piuttosto semplici. *printSprite* esegue un doppio ciclo for che scorre la matrice sprite e la stampa a schermo, *deleteSprite* utilizza lo stesso principio ma per la cancellazione. La *printLives*, invece, stampa le vite in alto a sinistra mediante uno switch. Ogni case stampa il giusto numero di cuori, per un massimo di 3. Abbiamo inoltre la funzione *rocketAnimation*, che da al razzo un effetto visivo di rotazione durante il suo viaggio verso l’obiettivo. Anche qui, un banale switch si occupa della stampa del carattere di turno.

7.2. La collisione

La funzione *checkCollision* si occupa di valutare se sia o meno avvenuta una collisione tra un oggetto razzo a ed un oggetto nave *b*, con un principio simile a quello delle stampe. La differenza sta nel fatto che vengono confrontate le coordinate dei due oggetti coinvolti, e se la collisione viene rilevata la funzione restituisce il valore 1, altrimenti 0. In appoggio, abbiamo la funzione *isRocket* che controlla l’*identifier* dell’oggetto passato come parametro. La conclusione del ciclo vitale di un oggetto comporta sempre la chiamata a *resetItem*, che assegna a tutti i campi dell’oggetto un valore negativo.

7.3. Time travel

La funzione *timeTravelEnemyRocket* prende come parametro un valore in *microsecondi* e restituisce il tempo di percorrenza di tutta la schermata di gioco di un razzo, espresso in secondi. La formula è molto semplice: (tempo * distanza) / 1000000.



Un piccolo schema. In rosso la generazione dei processi.

VERSIONE THREAD

In questa sezione parleremo principalmente delle differenze rispetto alla versione processi, avendo abbondantemente trattato il resto precedentemente.

1. Pipe vs Buffer

La differenza fondamentale tra le due versioni riguarda la comunicazione. Se nella versione processi abbiamo utilizzato la "*Pipe*", nella versione thread abbiamo implementato un concetto se vogliamo abbastanza simile: il *Buffer*. Questo buffer altro non è se non un array sul quale i thread produttori scriveranno e dal quale un consumatore leggerà. Per fare ciò, stavolta, non avremo strumenti già pronti a disposizione (*write* e *read*), ma avremo bisogno di soluzioni "*custom*".

1.1. Le funzioni insert ed extract

La funzione insert è quella che ci permetterà di *scrivere* nel buffer un oggetto di gioco passato come parametro, mentre la funzione *extract* ci permetterà di *leggere* un dato dal buffer e di restituirlo ad un'apposita variabile. Eventuali conflitti che potrebbero sorgere vengono gestiti mediante l'uso di due *semafori* e di un *mutex*.

- Semaforo empty : utile per verificare che il buffer non sia pieno
- Semaforo full : utile per verificare che il buffer non sia vuoto
- Mutex : utile per la mutua esclusione dall'accesso alla risorsa.

Il flusso delle due funzioni è molto simile:

per prima cosa si verifica che il buffer non sia pieno, nella insert, o vuoto, nella extract. Non appena la funzione avrà il "via libera", il mutex si occuperà di *bloccare* l'accesso concorrente al buffer, in modo da non avere conflitti con altri thread che vorrebbero a loro volta scrivere nella stessa area di memoria. A questo punto i dati possono essere elaborati senza timore di interferenze esterne, quindi la insert sarà libera di scrivere il suo dato nel buffer, la extract viceversa sarà libera di estrarlo. Dopo l'aggiornamento degli indici per il successivo accesso all'array il mutex *sbloccherà* l'accesso concorrente al buffer e si aggiornerà il semaforo, segnalando l'aggiunta o la rimozione di un dato agli altri thread che attendono l'accesso al buffer.

2. Altre differenze

2.1. Processi vs Thread

Per il passaggio da una versione all'altra, pur mantenendo la stessa struttura, si è reso necessario l'adattamento di alcune funzioni per renderle compatibili coi thread. Le differenze rimangono comunque minime e non meritano particolari approfondimenti, trattandosi principalmente di

variazioni del tipo di ritorno e variazioni dei parametri in ingresso. Le differenze più evidenti sono relative ai razzi, sia del giocatore che dei nemici.

2.2. La gestione degli spari

L'implementazione degli spari di entrambe le fazioni ha subito qualche modifica, in particolar modo per quanto riguarda la terminazione dei thread ad essi associati. Si è deciso di sfruttare la rilevazione del tempo e lasciare la terminazione del thread a sé stesso. Sostanzialmente, un controllo verifica quanto tempo è passato dalla creazione del thread e compara questo valore al tempo massimo di percorrenza di tutta l'area di gioco. Una volta raggiunto, esce fuori dal loop di gestione del razzo e il thread termina naturalmente.