Universitità di Cagliari

Facoltà di Scienze MM.FF.NN.

Tesina Corso SO1 - Parte 2, Threads

Davide Gessa (45712)

A.A. 2011 - 2012

28 Dicembre 2011

Indice

1 Struttura del programma **3**
1.1 Suddivisione files 3
1.2 Elenco funzioni 3
1.3 Il main 4

2 Architettura del programma **5**
2.1 Comunicazione dei Task 6
2.2 Tasks 6
2.2.1 Controllo 7
2.2.2 Alieno 7
2.2.3 Bomba 8
2.2.4 Navicella 8
2.3 Sincronizzazione 8
2.3.1 Buffer 'objects' 8
2.3.2 Coda 'queue' 9

Capitolo 1

Struttura del programma

1.1 Suddivisione files

Il codice del programma è stato ripartito in vari file cercando di dividere le varie funzionalità dei diversi oggetti della scena:

- alien.c alien.h - navicella aliena
- bomb.c bomb.h - bomba aliena
- control.c control.h - controllo delle iterazioni tra i vari oggetti e rendering della scena
- main.c - starter dell'applicazione
- missile.c missile.h - missile lanciato dall'astronave
- scores.c scores.h - gestione dei punteggi
- space_ship.c space_ship.h - navicella giocatore
- utility.c utility.h - funzioni varie utili per il funzionamento del programma
- space_invaders.c space_invaders.h - definizioni globali e funzioni comuni

1.2 Elenco funzioni

- alien_task() : gestione navicella aliena
- bomb_task() : gestione bomba aliena
- clear_quad() : cancella un area quadrata dello schermo

- control_check_collision() : controlla se c'è una collisione tra due oggetti
- control_task() : gestione della scena
- missile_task() : gestione di un missile
- render_string_array() : renderizza uno sprite nello schermo
- scores_add() : aggiungi un punteggio alla lista punteggi
- scores_load() : carica i punteggi da un file
- scores_save() : salva i punteggi in un file
- space_ship_task() : gestione navicella giocatore
- timevaldiff() : calcola la differenza tra due strutture timeval
- queue_add() : aggiunge un elemento alla coda di update
- queue_get_first() : restituisce il primo elemento della coda di update
- queue_init() : inizializza la coda di update
- queue_exit() : deinizializza la coda di update
- get_free_object_index() : restituisce la prima posizione libera nel buffer condiviso degli oggetti
- control_set_collision() : imposta ad un oggetto lo stato delle collisioni
- get_collision_state() : restituisce ed azzera lo stato delle collisioni di un oggetto

1.3 Il main

La funzione main si preoccupa di inizializzare l'applicazione (coda di update, dimensione schermo), creare i thread per gli oggetti iniziali con le relative funzioni di gestione.

1. Creazione thread navicella
2. Creazione dei threads degli alieni
3. Avvio della funzione di controllo

Capitolo 2

Architettura del programma

Ogni oggetto segue un funzionamento simile agli altri oggetti, e viene creato nel medesimo modo; ogni oggetto ha una funzione di gestione così strutturata:

1. Inizializza i dati iniziali e li inserisce in una struttura di tipo object_-data_t
2. Esegue un loop (dal quale esce al verificarsi di alcune condizioni) che ad ogni iterazione produce le nuove informazioni dell'oggetto (ad esempio, tramite la tastiera nel caso della navicella del giocatore), le inserisce nella sua struttura e le invia al controllo tramite una coda di update. Inoltre, il controllo può utilizzare il buffer condiviso per segnalare ad un determinato oggetto che è avvenuta una collisione (vedi sezione 2.1).

Le informazioni di un oggetto sono memorizzate nella seguente struttura:

```
///> Struttura contenente le informazioni relative all'oggetto
typedef struct
{
int  x; ///< Posizione x dell'oggetto
int y; ///< Posizione y dell'oggetto
int  size; ///< Dimensione dell'oggetto (sia x che y)

object_type_t type; ///< Tipo di oggetto
int life; ///< Vita rimanente all'oggetto

pthread_t thread; ///< Thread assocciato all'oggetto
int id; ///< Posizione nell'array degli oggetti

direction_t dir; ///< Direzione oggetto (usata per i marzianetti)
```

```
pthread_mutex_t coll_mutex; ///< Mutex per aggiornare le collisioni
object_type_t coll; ///< Stato delle collisioni dell'oggetto
} object_data_t;
```

Strutturare in questo modo il software, mi ha consentito di convertire la versione che utilizzava i processi/pipe, nella versione corrente che utilizza i threads in meno di un ora; mi è bastato aggiungere delle funzioni thread-safe per la gestione della comunicazione, e sostituirle alle vecchie chiamate per la scrittura/lettura delle pipe.

2.1 Comunicazione dei Task

Per quanto riguarda la comunicazione tra il task di controllo e gli oggetti, ho strutturato il software in modo da diminuire il numero di accessi in memoria ed i tempi di attesa ai semafori; tutti gli oggetti della scena condividono:

- objects[OBJECTS_MAX] : Un buffer contenente tutti gli oggetti della scena con le relative informazioni, nel quale ci scrive il thread di controllo per segnalare agli oggetti eventuali collisioni. Il controllo utilizza la funzione control_set_collision(object_data_t *obj, object_type_t t) per segnalare ad un oggetto 'obj' che è avvenuta una collisione con un oggetto di tipi 't'; l'informazione verra inserita nell'apposito slot dell'array 'objects'. Il thread dell'oggetto, potrà verificare se sono avvenute collisioni con la funzione object_type_t get_collision_state(int id).
- queue[QUEUE_SIZE] : Una coda di update nella quale gli oggetti inseriscono la loro struttura object_data_t per essere renderizzata dal controllo. E' possibile accedere alla coda 'queue' tramite delle apposite funzioni che permettono di sincronizzare gli accessi: queue_add(object_data_t ob) : aggiunge un oggetto alla coda (usata dai thread degli oggetti) queue_get_first(object_data_t *ob) : preleva un oggetto dalla coda (usata dal controllo per ottenere il prossimo oggetto da renderizzare)

Una schematizzazione della comunicazione tra i task dell'applicazione è visibile in figura 2.1.

2.2 Tasks

Come stabilito nelle specifiche, alieni, controllo, bombe, navicella e missili utilizzano un thread separato ciascuno.

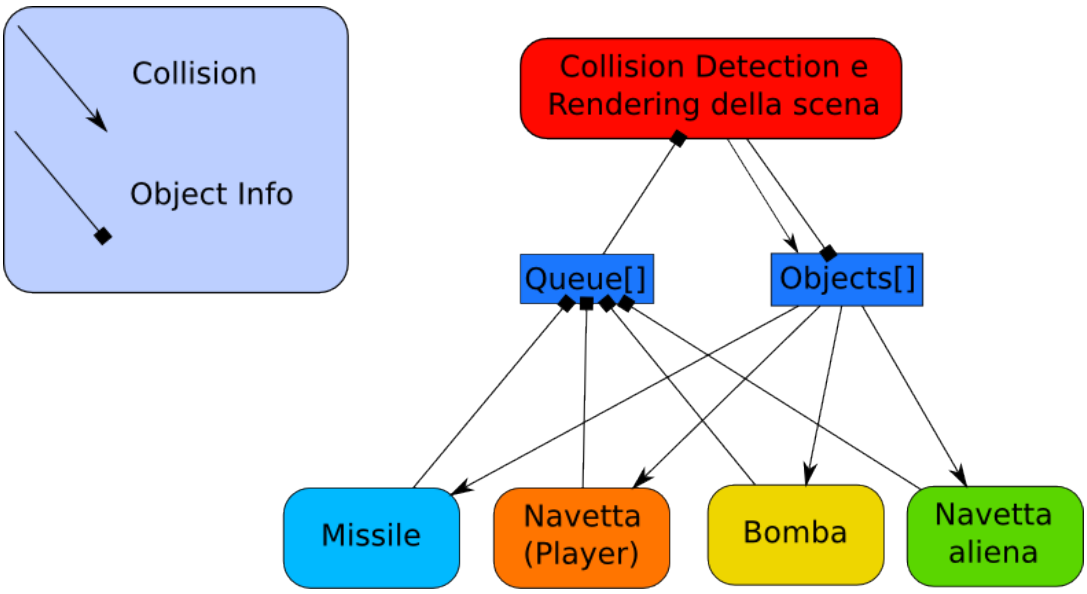

Figura 2.1: Schema di comunicazione fra i tasks

2.2.1 Controllo

Il thread di controllo esegue un loop, ed ad ogni iterazione si occupa di:

1. Ricevere dalla coda 'queue' le informazioni relative ad un oggetto
2. Salvare nell'array 'objects' le informazioni ricevute
3. Controllare se l'oggetto è in collisione con un altro oggetto, ed in tal caso, eseguire una control_set_collision per informare l'oggetto
4. Cancellare l'area di schermo occupata dall'oggetto nell'iterazione precedente
5. Ridisegnare l'oggetto nella nuova posizione

Il ciclo viene interrotto quando si verifica una condizione di gameover, o quando il giocatore vince. Quando il gioco finisce, il controllo invia a tutti i threads ancora in vita (missili, bombe, alieni, navicella) un informazione (tramite buffer condiviso) per avvisare che il programma deve chiudersi e che i threads devono terminare.

Finito il gioco, viene aggiunto il punteggio nella lista punteggi salvata in un file, e viene visualizzata la classifica.

2.2.2 Alieno

Si muove seguendo un percorso destra->giu->sinistra->giu come nel gioco originale; ad ogni iterazione invia al controllo la nuova posizione tramite

la coda, e legge dal buffer condiviso degli oggetti le informazioni relative alle collisioni: se collide con un altro alieno, cambia direzione, altrimenti decrementa la vita, e nel caso abbia finito le vite disponibili, si distrugge e si ricrea di livello superiore.

Ad intervalli regolari l'alieno sgancia una bomba, generando un nuovo thread.

2.2.3 Bomba

Il thread bomba ad ogni iterazione, scende di una posizione, invia al controllo la nuova posizione tramite la coda, e legge dal buffer condiviso degli oggetti le informazioni relative alle collisioni.

2.2.4 Navicella

Ad ogni iterazione, il thread della navicella attende un input da tastiera per comandare il movimento o sparare dei missili; il movimento può essere solo orizzontale. Il lancio dei missili è limitato da un timer, che permette un certo numero di spari ogni secondo. Premuto il tasto di sparo, la navicella crea due nuovi threads (missile destro e missile sinistro) che avviano la funzione di gestione del missile.

Missile

Il thread missile ad ogni iterazione, sale di una posizione in verticale, si sposta di un'unità in orizzontale (a seconda che sia un missile destro od un missile sinistro), invia le info al controllo tramite la coda, e attende un tempo predefinito; come per la bomba, il loop termina quando il processo di controllo segnala un avvenuta collisione con un altro oggetto.

2.3 Sincronizzazione

Avendo due buffer condivisi da tutti i threads, ho dovuto sincronizzare gli accessi per non generare inconsistenze nei dati.

2.3.1 Buffer 'objects'

E' possibile accendervi per modificare lo stato delle collisioni; ogni elemento del buffer contiene un mutex 'coll_mutex' ed una variabile intera 'coll'; per modificare la variabile intera 'coll' è necesssario fare un lock del mutex. Ho realizzato due funzioni threadsafe per modificare e leggere il contenuto della variabile coll, control_set_collision() per impostare una collisione ad un oggetto e get_collision_state() per verificare lo stato delle collisioni di un oggetto e azzerarne il valore.

2.3.2 Coda 'queue'

E' possibile accedervi per inserire un oggetto in coda o per prelevare il primo inserito tramite le due funzioni queue_add() e queue_get_first(); ho utilizzato l'algoritmo per il buffer condiviso, con n produttori ed un consumatore; le due funzioni sopra citate utilizzano tre semafori per regolare l'accesso al buffer.