Progetto di Programmazione 2021/2022

0 – Componenti del gruppo e ruoli

- Tiziano Rigosi: stanze, *Blueprint* e struttura del livello.
- Cono Cirone: generazione livello e gestione schermate di gioco.
- Luca Gabellini: sistema delle *Entity*; *Player* e relative componenti (artefatti, proiettili, etc).
- Davide Filippi: nemici.

1 – Game loop

L'esecuzione del gioco avviene nella funzione *main( )*. Il menu del gioco è implementato tramite un ciclo *while* che attende input da tastiera per cominciare la partita. In seguito ad alcuni passaggi di inizializzazione comincia il vero e proprio loop di gioco *(fig.1)*, strutturato nel modo seguente:

1) elaborazione input del giocatore;
2) aggiornamento degli elementi in gioco;
3) aggiornamento dello schermo;
4) *sleep( ).*

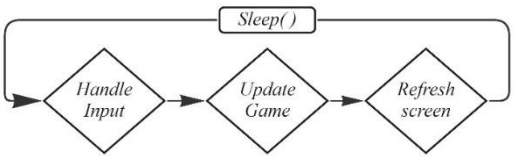

Figura 1: schema del game loop

La scelta di introdurre un meccanismo di *sleep* è dovuta alla necessità di limitare la velocità d'esecuzione del programma: infatti, senza di esso eseguirebbe troppo velocemente per essere perfino visibile agli occhi di chi gioca. In questo modo, invece, è possibile regolarizzare il *game loop* e assicurare una esecuzione corretta e consistente del programma.

2 – Gestione dello schermo

La schermata iniziale viene gestita dalla classe *PrimaSchermata*. La funzione *disegna_titolo( )* mostra a schermo il titolo del gioco. Poi ci vengono mostrate delle scelte, in particolare possiamo:

- iniziare una nuova partita (avviando il *game loop* descritto sopra);
- visualizzare il menu dei comandi;
- uscire dal gioco (terminando il *while*) e ritornare al terminale.

La schermata finale viene gestita dalla classe *GameOver*. Questa visualizza il punteggio ottenuto dal giocatore nella partita e permette di tornare al menu principale.

La gestione dello schermo durante la partita avviene mediante la classe *GestioneSchermo*. In particolar modo lo schermo viene diviso, grazie all'utilizzo della funzione *partition( )*, in tre finestre:

1) finestra in cui si svolge la partita;

2) finestra dove si possono visualizzare le informazioni sulla partita corrente: vita rimanente, numero di bombe, artefatto corrente e punti esperienza;
3) finestra dei dialoghi, in cui, in base a quello che sta succedendo al giocatore in quel momento nella partita mostra un messaggio diverso.

3 – Elementi di gioco: *Entity* e le sue sottoclassi

La classe astratta *Entity* definisce un oggetto presente a schermo (quindi dotato di *Shape* e *Location*) con il quale è possibile interagire durante la partita. Per definizione di classe astratta non è possibile creare direttamente oggetti di tipo *Entity* ma solo di sue sottoclassi che ne implementino i metodi. Tutte i principali elementi di gioco derivano da questa specifica classe *(fig.2).*

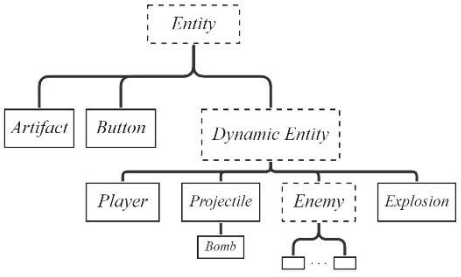

Figura 2: gerarchia delle sottoclassi di Entity. Le classi astratte sono rappresentate con il tratteggio.

Le *Entity* sono legate tra loro mediante la classe *Entity_List*, una lista bidirezionale presente in ogni stanza di gioco. Ad ogni ciclo del *main loop,* viene scansionata la lista della stanza in cui si trova attualmente il giocatore per aggiornare lo stato di tutte le *Entity* presenti; inoltre, si rilevano eventuali collisioni avvenute. La classe si occupa solamente di notificare alle *Entity* che una collisione si è verificata: l'effettiva gestione della stessa è poi delegata tramite *Entity_List* agli oggetti coinvolti. Se una *Entity* viene distrutta, la si rimuove dalla lista.

Riguardo la gestione delle collisioni: ogni *Entity* memorizza ad ogni passo la posizione da cui proviene e i caratteri che occupavano la posizione in cui si trova al momento. Quando avviene una collisione generica (ad es. con un ostacolo), la *Entity* è riportata nella posizione precedente e i caratteri vengono ripristinati. Il fatto che questo procedimento avvenga prima che lo schermo sia aggiornato crea l'illusione di "solidità" degli ostacoli.

Per i tipi *Artifact* e *Projectile*, si è scelto di distinguere l'istanza *in-game* dell'oggetto (la *Entity*) dal suo descrittore (strutture *Artf_Type* e *Prj_Type*). Se quindi, ad esempio, si vuole posizionare l'artefatto *Armor*, è sufficiente dichiarare un *Artifact* dotato di puntatore al *Artf_Type* dell'armatura. Analogamente, il *Player* mantiene solo un riferimento ai tipi di artefatto e proiettile al momento equipaggiati piuttosto che all'istanza.

La classe *Enemy* è stata creata come classe template di tutti i nemici: il metodo caratteristico della classe è *select_direction( )* che sceglie la direzione del nemico in base a quella del giocatore. Oltre ai punti vita, è presente una variabile *wait* che genera una possibilità randomica di saltare il turno (per non rendere il nemico monotono) e una variabile *speed* per definire ogni quante azioni del giocatore il nemico può muoversi, simulandone quindi la velocità.

Le sottoclassi di *Enemy* implementano i singoli nemici, andando ad alterare in maniera significativa comportamenti ereditati dalla superclasse. Ad esempio, nemici come *Wasp* o *Big* inseguono il giocatore, mentre *Turret* resta immobile e ruota su sé stesso.

4 – Livelli

La classe *Blueprint* definisce in modo statico l'aspetto delle stanze e il posizionamento delle entità al loro interno. Ciascuna istanza di *Blueprint* è composta da una *mappa* (ovvero un array di stringhe che definisce l'aspetto della stanza) e un array di *BP_Entity,* una semplice struttura dati che contiene il nome di una *Entity* e le sue coordinate. Tutte le istanze di *Blueprint* sono inserite in una lista ad anello da cui il livello sceglie di volta in volta, in modo semi casuale, quale stanza andare a creare.

La classe *Room* e le rispettive sottoclassi definiscono lo spazio in cui il giocatore supera ostacoli, raccoglie oggetti e sconfigge nemici. Una volta entrato in una stanza, il giocatore dovrà sconfiggere i nemici al suo interno, se presenti, prima di poter aprire le porte della stanza e poterne uscire: nel momento in cui la relativa *Entity_List* è svuotata, viene chiamata la funzione *open_doors( ).*

Il parametro *id* varia tra la classe padre e le varie sottoclassi, e determina, oltre al loro contenuto, anche l'aspetto e le condizioni di apertura delle porte che conducono ad una data stanza: la stanza speciale e la stanza del boss richiedono rispettivamente di aver completato la metà e tutte le stanze del livello; la stanza segreta richiede l'utilizzo di una bomba.

La generazione del livello avviene grazie alla funzione *generate( )*. La generazione avviene in maniera casuale, infatti, il tutto viene gestito da un *int dado* inizializzato tramite *rand( ).* Inizialmente l'idea era quella di generare tutte le stanze con il dado che decideva la casella della matrice in cui andare a disporre la nuova stanza. Il problema di questo approccio è che, data la poca affidabilità di *rand( ),* in alcuni casi, portava a creare delle mappe che si sviluppavano lungo un'unica direzione. Per evitare questo problema si è introdotta la funzione *generate2( )*: una volta generata una stanza nella posizione scelta dal dado, *generate2()* andrà a creare stanze attorno a quella appena creata (quando possibile). In questo modo si riescono ad avere più stanze collegate tra loro ma allo stesso tempo mantenere una generazione casuale della mappa.

La stanza del boss, la stanza speciale e la stanza segreta vengono generate in posizioni specifiche del livello per garantire che il giocatore non venga bloccato da porte che è incapace di aprire.

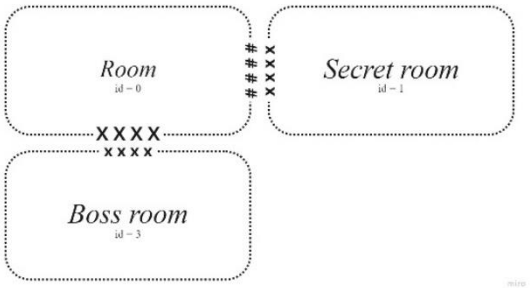

Figura 3: I caratteri delle porte dipendono dagli id delle stanze collegate.

Dopo aver generato la mappa tutte le porte di accesso alle stanze vengono chiuse con la funzione *close_all_doors( )* che effettua un ciclo *for* sulla matrice delle stanze e posiziona le porte con le funzioni *mettiporta( )* in base all'id della stanza adiacente *(fig.3).* Il meccanismo è stato pensato in questo modo per far comprendere meglio al giocatore a che tipo di stanza può accedere.