Dungeon Arena

Davide Del Bimbo
davide.delbimbo@edu.unifi.it

Abstract

Dungeon Arena è un videogioco arcade 2D che offre un'ambientazione fantasy, il cui obiettivo è quello di uccidere i nemici. Nel corso dello sviluppo, abbiamo implementato diverse funzionalità e meccaniche di gioco, tra cui la gestione dei personaggi, l'intelligenza artificiale dei nemici, il sistema di combattimento, la gestione degli oggetti collezionabili e molto altro ancora. In questo documento, esamineremo nel dettaglio le varie componenti del gioco, illustrando come sono state progettate e implementate.

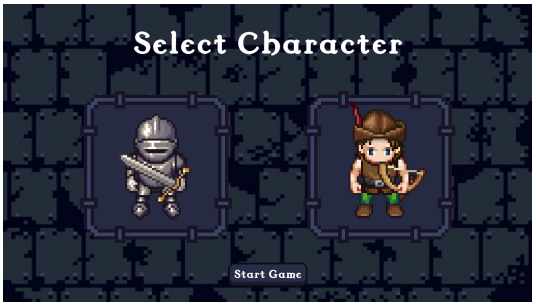

Figure 1: Scena di selezione del personaggio.

1. Schermata iniziale e scelta del personaggio

Il gioco si avvia mostrando una scena iniziale con due pulsanti: "Start Game" e "Exit Game". Premendo il primo pulsante, si viene rimandati ad una seconda scena di selezione dei personaggi, mentre il secondo pulsante permette di chiudere il gioco.

Dalla schermata di selezione del personaggio (figura 1 è possibile scegliere il personaggio con cui iniziare il gioco. In particolare, possiamo scegliere tra la tipologia "Knight" o "Archer". Il primo consente un combattimento ravvicinato corpo a corpo, ma con un movimento più lento. Il secondo consente un combattimento a distanza, ma con un movimento più veloce.

Queste due schermate sono state implementate agendo sulla User Interface. In particolare, sono stati definiti dei Canvas che si adattano correttamente al ridimensionamento della schermata. Sono stati aggiunti pulsanti e animazioni per effettuare le varie scelte.
I pulsanti che rappresentano la scelta dei personaggi sono stati realizzati attraverso un prefab. All'avvio del gioco, in base al numero di personaggi giocabili definiti, viene popolata la lista dei personaggi con le possibili scelte. Una volta scelto il personaggio, si abilita il pulsante "Start Game".

La transizione tra le scene viene gestita attraverso la classe `SceneManager`, che implementa il pattern Singleton. Questa classe espone metodi che consentono di navigare tra le varie scene del gioco o di uscire dal gioco. Inoltre, consente il caricamento asincrono della scena di gioco dopo aver selezionato il personaggio. In questo modo, possiamo mostrare una barra per il progresso del caricamento senza bloccare l'interfaccia utente.

La scelta del personaggio viene invece gestita attraverso la classe `SelectCharacterManager`, anch'essa un Singleton. Questa classe memorizza una lista dei personaggi giocabili e l'indice del personaggio che si sta scegliendo. Ogni volta che si modifica la scelta, questo indice viene aggiornato, fino a che non si preme il pulsante "Start Game".

2. Mappa

La mappa del gioco è statica e presenta diverse caratteristiche uniche. Il giocatore inizia al centro della mappa e può decidere di spostarsi in due aree speculari.

Queste aree sono collegate da un portale situato nella parte alta della mappa (figura 2). Il portale si attiva all'inizio del gioco e, una volta attraversato, si disattiva per un periodo. Questo meccanismo rende la mappa una sorta di toroide, permettendo al giocatore di muoversi ciclicamente tra le due aree.
In particolare, il portale è costituito da un `Collider2D` che rileva quando il personaggio collide con esso (impostato nel layer `Portals`) e da un punto di spawn per spostare il personaggio in un altro punto della mappa.
Inoltre, sono stati aggiunti ulteriori effetti grafici, come una porta che chiude il portale una volta che il personaggio vi entra e delle sprite che creano un effetto di luce sul pavimento quando il portale è aperto. La porta contiene un `Collider2D` impostato nel layer `Obstacles` per evitare

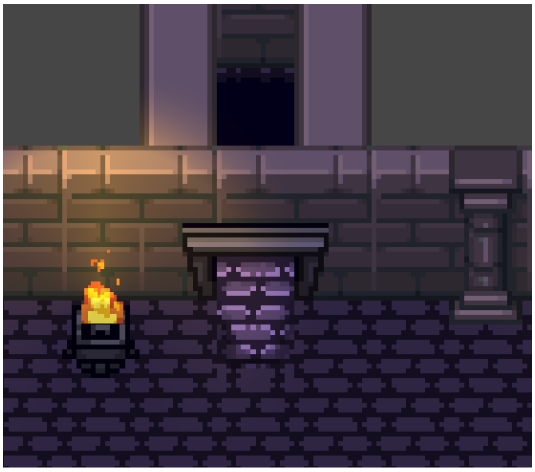

Figure 2: Portale per muoversi nelle aree della mappa.

che il personaggio entri nuovamente nel portale. Quando il giocatore accede al portale, si avvia una Coroutine che sostituisce le mattonelle e attiva le porte.

La mappa include vari ostacoli che impediscono il movimento del personaggio, come colonne, casse e fuochi. Inoltre, l'atmosfera del gioco è arricchita da un'illuminazione globale soffusa e da punti di luce, posizionati strategicamente per migliorare l'immersività e l'aspetto visivo del gioco. Questi punti luce sono stati definiti attraverso un unico prefab e dislocati all'interno della mappa come una rule tile.

3. Camera

La telecamera principale del gioco è gestita dalla classe `CameraFollow`, che si occupa di seguire il movimento del personaggio principale. La telecamera è impostata in modalità `Ortographic`, particolarmente adatta per i giochi 2D in quanto elimina la distorsione prospettica e mantiene le proporzioni degli elementi visivi costanti indipendentemente dalla loro distanza dalla telecamera.

La classe `CameraFollow` utilizza un sistema di smoothing per garantire che i movimenti della telecamera siano fluidi e non bruschi, migliorando così l'esperienza visiva del giocatore. La telecamera si adatta automaticamente alla posizione del personaggio, centrando il campo visivo su di esso e permettendo al giocatore di vedere una porzione ottimale della mappa circostante.

Inoltre, la telecamera è configurata per utilizzare una modalità di ordinamento sull'asse Y in modo da garantire una corretta rappresentazione della profondità all'interno del gioco. Questa configurazione consente di disporre gli sprite in modo che quelli con un valore Y inferiore (più vicini al fondo della mappa) vengano disegnati davanti a quelli con un valore Y superiore (più lontani dal fondo della mappa), creando un effetto di profondità visiva.

Per abilitare questa modalità di ordinamento, sono state modificate le impostazioni della `Transparency Sort Mode` della telecamera. Questa impostazione consente di ordinare i rendering degli sprite in base alla loro distanza lungo un asse personalizzato (in questo caso, l'asse Y). Per impostazione predefinita, il punto di ordinamento di uno sprite è rappresentato dal suo centro, ma è stato modificato per utilizzare la posizione del perno degli sprite, detto **pivot**.

4. Character

Tutti i personaggi nel gioco (sia quelli giocabili dal giocatore che quelli controllati dalla CPU) sono rappresentati dalla classe `Character`. Questa classe gestisce il controllo del personaggio e i suoi stati, operando come una macchina a stati. In particolare, ogni personaggio può trovarsi in uno degli stati: `Idle`, `Walk`, `Attack` o `Dead`. Le transizioni tra questi stati sono gestite in modo diverso a seconda che il personaggio sia controllato dal giocatore o dalla CPU.

In generale, se non viene rilevato alcun input, il personaggio si trova nello stato di `Idle`. Quando si rileva un input di movimento, il personaggio passa allo stato di `Walk`. Se viene rilevato un input di attacco, il personaggio passa allo stato di `Attack`. Al termine dell'attacco, il personaggio ritorna automaticamente allo stato di `Idle`.
Lo stato di `Dead` viene raggiunto quando il personaggio perde tutta la vita. Questo, è uno stato terminale dal quale non può uscire.

Ogni stato si occupa di avviare correttamente le animazioni basate sulla direzione verso cui è rivolto il personaggio. Per ciascuno stato esistono quattro sottostati corrispondenti alle direzioni: `Up`, `Down`, `Left` e `Right`. Questo consente una rappresentazione visiva coerente delle azioni del personaggio.

Le animazioni sono gestite dalla classe `Animated-Sprite`, che avvia una Coroutine per sostituire le sprite nel `Sprite Renderer` ad un frame rate specificato fino a quando l'animazione non viene fermata (ad esempio, quando si passa a un altro stato) o se l'animazione non è un loop (come nel caso dell'animazione di attacco).

Il movimento del personaggio è invece gestito dalla classe `Movement`, che modifica la direzione del movimento del `Rigidbody2D` in base agli input, muovendo il personaggio a una velocità fissata. Il movimento viene aggiornato nella funzione `FixedUpdate` per interagire correttamente con il motore fisico del gioco. Inoltre, sono consentiti solo movimenti lungo i quattro assi cardinali.

Per gestire in modo indipendente gli input, viene definita un'interfaccia chiamata `IInputHandler`. Questa interfaccia fornisce i metodi `GetMovement` per definire l'input di movimento e `GetFire` per determinare quando passare allo stato di attacco.

4.1. Player

La classe `Player` rappresenta i personaggi giocabili dal giocatore e gestisce diversi aspetti del loro comportamento e delle interazioni nel gioco. Questa classe implementa le interfacce `IAgent` e `IDamageable`, fornendo funzionalità per la gestione della salute, del danno e del movimento del personaggio.

In particolare, questa classe definisce gli stati del personaggio in base agli input della tastiera e del mouse, incorporando il componente `KeyboardInputHandler`.

4.2. Enemy

La classe `Enemy` rappresenta un personaggio controllato dalla CPU e definisce una macchina a stati per il nemico. Il nemico può trovarsi in uno degli stati `Wait`, `Patrol`, `Chase` o `Vulnerable`. Lo stato `Wait` rappresenta un periodo di inattività in cui il nemico non si muove e ha una durata predefinita. Al termine di questa durata, il nemico passa allo stato `Patrol`. Nello stato `Patrol`, il nemico si muove verso il suo punto di spawn e pattuglia punti casuali entro il raggio di pattugliamento del punto di spawn. Quando viene rilevato un giocatore nel raggio di rilevamento, il nemico passa allo stato `Chase` per inseguirlo. Invece, lo stato di `Vulnerable` è attivato da eventi esterni, come l'uso di un power-up da parte del giocatore. In questo stato, il nemico cerca una via di fuga per eludere l'attacco del giocatore. Al termine del potenziamento o se il nemico è sufficientemente lontano dal giocatore, torna allo stato di `Wait`.

Anche in questo caso, gli input vengono gestiti attraverso una classe `AIInputHandler`, che estende la classe base `IInputHandler`. In particolare, il nemico decide di iniziare a sparare tenendo conto della velocità di attacco, della posizione e della direzione del giocatore per determinare il momento e la direzione dell'attacco.
Per definire le diverse strategie di movimento in base allo stato del personaggio, viene utilizzato un pattern Strategy. Nello stato `Wait`, viene restituita una direzione di movimento `zero` per fermare il personaggio. Nello stato `Patrol`, il nemico si muove verso un punto casuale all'interno del raggio di pattugliamento. Durante lo stato `Chase`, l'obiettivo di movimento diventa il giocatore. Mentre, nello stato `Vulnerable` il nemico cerca un punto valido in una direzione che non sia quella da cui proviene il giocatore.

4.3. Pathfinding

Per calcolare i percorsi da seguire in base all'obiettivo di movimento, viene implementato l'algoritmo A*. A tale scopo, si definisce una griglia di nodi, in cui ogni nodo rappresenta una cella della mappa e contiene informazioni cruciali come la sua posizione nella griglia, la corrispondente posizione nella mappa, se è camminabile, il nodo genitore ed i costi G, F, H.

La griglia viene costruita specificando la risoluzione delle celle e assegnando a ciascuna di esse la proprietà di camminabilità, per determinare se il nemico può muoversi in quella direzione o meno. Ciò avviene verificando se la cella non è bloccata da ostacoli. Inoltre, viene implementato un metodo per trovare i vicini di un nodo ed un metodo per trovare il nodo corrispondente ad una data posizione sulla mappa.

L'algoritmo A* è un algoritmo di ricerca percorso più breve da un punto di partenza a un punto di arrivo all'interno di una griglia. La sua efficacia e la sua popolarità derivano dalla sua capacità di combinare la completezza (ovvero la capacità di trovare una soluzione se esiste) con l'ottimalità (ovvero la capacità di trovare la soluzione più breve) in molte situazioni:

1. Si ottengono i nodi di partenza e di destinazione dalla griglia. Quindi si inizializzano due liste: una contenente i nodi che devono ancora essere esaminati (lista aperta) ed una contenente i nodi già esaminati (lista chiusa). Quindi, si imposta il costo G ed H per il nodo di partenza e lo si aggiunge alla lista aperta.

2. Durante ogni iterazione del ciclo, si seleziona il nodo con il costo F minore tra quelli ancora da visitare. Se si raggiunge il nodo di destinazione, si ritorna il percorso trovato.

3. Si esplorano i vicini del nodo corrente. Per ogni vicino, si calcola il costo G, il costo H e si aggiorna il nodo genitore se viene trovato un percorso migliore. Se il vicino non è nella lista aperta, lo si aggiunge ad essa.

4. Una volta raggiunto il nodo di destinazione, si ricostruisce il percorso dalla fine (nodo di destinazione) al nodo di partenza seguendo i nodi genitori.

5. La distanza tra due nodi viene calcolata utilizzando la distanza di Manhattan, sommando le differenze assolute delle coordinate X ed Y tra i due nodi. Questa euristica consente di considerare solo gli spostamenti nelle direzioni cardinali.

4.4. Spawner dei nemici

Gli spawn point dei nemici (figura **??**) sono configurati attraverso un singolo prefab, contenente informazioni come la lista dei nemici che possono essere spawnati, il range di spawn ed il numero massimo di nemici attivi. L'obiettivo è individuare una posizione valida sulla mappa per l'istanziazione dei nemici con una frequenza prestabilita. Questa posizione deve essere libera da ostacoli e non deve essere occupata né dal giocatore né da altri nemici. Inoltre, lo spawn point tiene traccia dei nemici già spawnati.

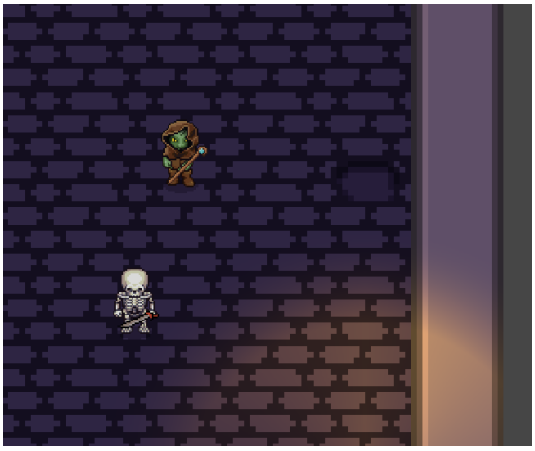

Figure 3: Spawn point dei nemici.

Quando un nemico viene eliminato, viene attivato un evento `OnDeath`, che consente allo spawn point di istanziare un nuovo nemico e rimuovere il nemico eliminato dalla lista.

Nello sviluppo del gioco sono inclusi due tipi di nemici distinti: lo "Skeleton" per il combattimento ravvicinato ed il "Wizard" per il combattimento a distanza. Durante la fase di spawn, si attiva un effetto visivo che coinvolge l'applicazione di un materiale bianco sulla sprite del personaggio mediante il metodo `Flash`.

5. Weapon

Sono state sviluppate due categorie di armi all'interno del gioco: quelle da combattimento ravvicinato e quelle da combattimento a distanza. Le prime sono progettate per attivare e disattivare un `Collider2D` che rappresenta l'area di impatto nella direzione dell'attacco, mentre le seconde generano proiettili nella stessa direzione. In particolare, la classe `Sword` estende la classe base `MeleeWeapon`, mentre le classi `Bow` e `Staff` estendono `RangedWeapon`. Una volta che l'hitbox di un personaggio viene colpito, queste armi infliggono danni.

Il rilevamento degli hitbox è gestito sia dai collider delle armi da combattimento ravvicinato sia dai raycast dei proiettili. Questi ultimi verificano che l'oggetto colpito non sia il personaggio stesso o altri personaggi appartenenti alla stessa classe `IAgent`. Inoltre, producono effetti visivi nel caso in cui colpiscono un oggetto fisico (come ostacoli o personaggi).

Le armi rilevano se l'oggetto colpito implementa l'interfaccia `IDamageable` e, in tal caso, applicano il danno all'agente. In questo contesto, viene verificato se il danno inflitto riduce la vita dell'agente al di sotto di zero, il che potrebbe comportare il passaggio allo stato di `Dead`. Quando un personaggio subisce un colpo, si attiva un effetto visivo che colora la sprite utilizzando il metodo `Flash`. Inoltre, il personaggio viene spinto all'indietro in base alla potenza dell'arma, subendo un effetto di `KnockBack`.

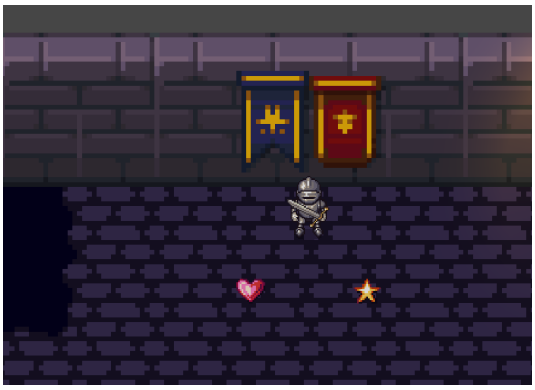

Figure 4: Oggetti rilasciati dai nemici.

6. Potenziamenti

Quando un nemico viene sconfitto, ha la possibilità di rilasciare degli oggetti con una certa probabilità. Tra gli oggetti rilasciabili vi sono quelli che ripristinano i punti vita, aumentano la velocità, moltiplicano il punteggio o inducono lo stato di vulnerabilità dei nemici (figura 4). Per gestire questo processo, è stata sviluppata una classe base chiamata `DroppableItem`. Questa classe si occupa di avviare un'animazione al momento dell'istanziazione dell'oggetto e di distruggerlo dopo un certo periodo di tempo, durante il quale il giocatore può raccoglierlo semplicemente passandoci sopra. Se ciò accade, gli effetti dell'oggetto, definiti nella classe `ItemEffect`, vengono applicati al giocatore.

Gli oggetti rilasciabili sono configurati come prefab e mantenuti dai nemici. Il nemico sceglie casualmente uno di questi oggetti e li rilascia basandosi sulle rispettive probabilità di drop. Inoltre, se il giocatore passa sopra all'oggetto droppato, viene attivato un effetto visivo, altrimenti l'oggetto si dissolve gradualmente.

7. Game Manager

La classe `GameManager` svolge un ruolo cruciale nel coordinare diversi aspetti del gioco. Una delle sue principali responsabilità è la gestione dei giocatori e dello stato del gioco. Una volta che il giocatore ha selezionato un personaggio dal menu di selezione, il `GameManager` si occupa di istanziare il `Player`. Successivamente, tiene traccia della vita del giocatore, dello score e della durata dei potenziamenti. All'inizio di ogni nuovo gioco, il GameManager ripristina lo stato del gioco.

Quando il giocatore muore, il GameManager rilascia le risorse necessarie per gestire la sua morte e aggiorna lo stato del gioco di conseguenza. Lo score del giocatore viene

modificato in base alle azioni compiute durante il gioco, come ad esempio l'uccisione dei nemici. Quando un nemico viene sconfitto, viene chiamato il metodo `AddScore`, che applica al punteggio del giocatore il valore previsto per quel nemico.

Inoltre, il `GameManager` gestisce i potenziamenti attivando e monitorando il timer per la loro durata attraverso il metodo `StartPowerUpTimer`.

8. User Interface

La UI (figura 5) permette di visualizzare diverse informazioni fondamentali durante il gioco, inclusa la vita corrente del personaggio, il timer del powerup e lo score attuale. Inoltre, include un pulsante di pausa che consente al giocatore di accedere al menu di pausa durante il gioco.

Il menu di pausa (figura 6) offre diverse opzioni per il giocatore. Può scegliere di riprendere il gioco dal punto in cui è stato interrotto, tornare al menu iniziale per selezionare un nuovo personaggio o uscire completamente dal gioco.

Quando il giocatore muore, compare una schermata di "Game Over" (figure 7) che riassume il punteggio finale ottenuto durante la partita. Da qui, il giocatore ha la possibilità di iniziare una nuova partita o di uscire dal gioco.

Figure 5: Interfaccia utente del gioco.

Figure 6: Menu di pausa.

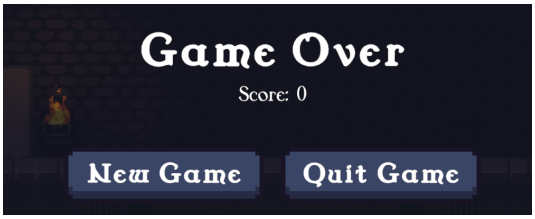

Figure 7: Schermata di Game Over.