Documentazione Progetto (Daniele Venafro)

Per questo test ho deciso di sviluppare un gioco fantasy in terza persona in cui bisogna impersonare un soldato alla ricerca di alcune reliquie sacre, con l'intenzione di ricevere una benedizione divina con la quale sconfiggere le forze del male. All'interno del gioco sarà quindi possibile raccogliere oggetti(**pozioni**, **monete**, **spada sacra**, **scudo sacro**), **comprare** pozioni parlando con un **npc** in cambio di monete, **utilizzare le pozioni** per curare la propria vita, **risolvere puzzle** ambientali muovendo oggetti in una posizione precisa, inserendo le reliquie nelle zone apposite o uccidendo tutti i **nemici** in una determinata area per liberare un passaggio. Ho dato **molta importanza** anche ai menu e la **UI** in generale, è possibile giocare il gioco completamente **sia con mouse che gamepad** e dalle impostazioni è possibile **modificare settaggi** come livelli audio, risoluzione e scegliere tra vari profili grafici preconfigurati o adottarne uno "custom" personalizzabile in ogni opzione. Il gioco **salverà le impostazioni** correnti e le ricorderà al prossimo avvio. Il gioco è stato sviluppato completamente in **inglese** (UI, **dialoghi**), un po' per mia abitudine e un po' per un mio tentativo di non perdere questa abitudine.

Player Controller: per il player ho utilizzato un rigidbody con un capsule collider, in combinazione con PlayerInput e il NewInputAction. Avrei potuto utilizzare il character controller che già implementa molte funzionalità non presenti per i rigidBody, ma all'inizio non avevo ancora chiaro cosa avrei fatto di preciso, per cui avere il personaggio influenzabile dalla fisica sarebbe potuto essere utile. Per la rilevazione del terreno ho usato uno SphereCast sotto i piedi del modello, mentre per la rilevazione degli ostacoli una combinazione di raycast: uno orizzontale per verificare se di fronte al player ci fosse un ostacolo e successivamente un altro verticale per verificare che l'ostacolo non fosse troppo alto. Per rendere il tutto più preciso ho ripetuto lo stesso non solo di fronte al player, ma anche a +-45 gradi, in modo da rilevare gli ostacoli in modo più preciso. Lo sphereCast inoltre è servito anche per rilevare la normale del terreno colpito, in modo da proiettare il vettore di movimento se necessario. Per il movimento infatti non ho usato un approccio legato a forza ed accelerazione, ma sulla posizione, pertanto in presenza di discese il personaggio avrebbe camminato dritto finendo con il non toccare il terreno e cadere. Proiettando il vettore di movimento rispetto alla normale del terreno è stato possibile garantire un movimento più realistico. Il personaggio può saltare(applicazione di un impulso) e nuotare. Per realizzare il nuoto ho semplicemente disattivato la gravità del rigidbody utilizzando un trigger lungo tutta la superficie marina, trigger attivato nel momento in cui un gameobject empty che ho inserito nella gerarchia del player vi ci entra. Regolando la posizione di questo oggetto in altezza ho potuto fare in modo che il nuoto si attivasse solo quando il personaggio si trova immerso entro una certa soglia. Ho anche implementato il suono dei passi del player, grazie allo sphereCast individuo il tag della superficie e in base a quello identifico quale suono utilizzare.

Impostazioni/UI: È possibile accedere alle impostazioni sia dal menu principale che dal menu inGame. Per la UI ho creato una classe astratta da usare come base in modo da avere una interfaccia comune, dei metodi già implementati, variabili inizializzate e metodi da implementare già definiti. In questo modo sono riuscito a velocizzare e semplificare lo sviluppo delle nuove interfacce utente. Per la realizzazione grafica ho utilizzato la UI Toolkit, non solo è più leggera dalla UI tradizionale di Unity, ma è anche molto comoda da utilizzare grazie ad UIBuilder. All'interno della classe astratta ho definito un metodo per inizializzare il navigationPath tra i VisualElement usando gamepad o tastiera, dato che la UI Toolkit non sempre gestisce bene questi eventi

nativamente, definirli in maniera esplicita è stato più efficace e definendo questo metodo ho potuto configurare il tutto più velocemente ad ogni nuova vista implementata. Ho creato anche una classe UIRoot per gestire i menu a comparsa, questo componente tiene traccia delle viste aperte in una lista, in questo modo basta scorrere indietro la lista per sapere sempre qual è stata la vista precedente. Utile ad esempio per tornare dalle opzioni all'inGameMenu semplicemente, anche perché questa logica funziona sempre e non è necessario implementare della logica nuova per tornare alla vista precedente quando si sviluppa una nuova UI.

Singleton: All'interno del progetto ho inserito 3 singleton per definire eventi o salvare/gestire dati di cui avevo necessità avere visibilità globale nel progetto senza l'uso di riferimenti:

1. **CollectableItemManager**: si occupa di identificare che tipo di oggetto collezionabile è stato raccolto e gestire di conseguenza caso per caso;
2. **HealthManager**: permette di leggere quanta vita è rimasta al player, applicare un danno o una cura. Gestisce anche la visibilità della vignetta rossa usata per indicare al giocatore che il player ha poca vita.
3. **PlayerInputState**: utile per salvare informazioni sull'input device attualmente in uso, l'actionMap o lo schema corrente. Permette di capire quando attivare il cursore del mouse e quando rendere visibile il focus per tastiera e gamepad.

Constants: una classe statica al cui interno salvo il valore di stringhe costanti che possono essermi utili nel progetto. In questo modo non posso dimenticare una stringa perché l'ide me la suggerisce, non possono esserci errori di battitura e se voglio cambiarne una mi basta modificare il valore della stringa e quel cambiamento viene applicato all'intero progetto.

Nemici/Combat: Per i nemici ho utilizzato un NavMeshAgent e ho realizzato una classe EnemyAI all'interno della quale ho definito gli stati dell'enemy con un enum. All'interno dell'update controllo in valore dell'enum con uno switch e richiamo la logica adeguata in base allo stato. I nemici individuano auotomaticamente il player se si avvicina entro una certa soglia, altrimenti hanno un campo visivo definito da un trigger. Se il player entra nel trigger, loro cominciano a lanciare un raycast verso la direzione in cui sono rivolti. Se il player viene toccato dal raycast, i nemici lo attaccano. In questo modo vengono consumate risorse per il raycast solo se il player è effetivamente a portata. Se non vedono il player o lo perdono di vista tornano in patrol e si muovono in maniera randomiche con delle pause tra un movimento e l'altro. Per gestire l'attacco ho utilizzato un animation event, in modo da lanciare l'evento in un punto specifico dell'animazione. A quel punto viene valutato il posizionamento di player e nemico per capire se il colpo ha avuto successo o meno. Tutti i valori necessari per il funzionamento dei nemici sono esposti nell'inspector e possono essere personalizzati. Per quanto riguarda l'attacco del player funziona tutta in maniera molto simile.

Conclusioni: Sviluppare tutto nell'arco di una settimana è stato complesso, era da tanto che non usavo Unity e questo non ha aiutato. Gestire le impostazioni e tutto ciò che è legato al combat system è stato impegnativo e calibrare per bene il playerController ha richiesto tempo e continue modifiche e/o aggiunte e ancora adesso è migliorabile. Probabilmente alcune classi si sarebbero potute spacchetare ulteriormente e sicuramente ci sono bug che non ho notato, però sono soddisfatto del risultato finale. Spero lo siate anche voi.