Alma Mater Studiorum - Universitá di Bologna

Campus di Cesena – Scuola di Scienze

Corso di Laurea in Scienze e Tecnologie Informatiche

Game Programming: Sviluppo di un Gioco

Tesi di Laurea in Programmazione

Relatore

DOTT. MIRKO RAVAIOLI

Presentata da

MICHELE SASSO

III° Sessione

Anno Accademico 2012 - 2013

Indice

Introduzione

Nella tesi in questione si vogliono analizzare e descrivere metodi e strumenti utilizzati per la realizzazione di un videogame.
Il progetto consiste in un *Puzzle / Horror Game* con meccaniche di gioco caratteristiche di un Puzzle Game, come la risoluzione di enigmi tramite l'utilizzo della logica accompagnate da un'ambientazione cupa e misteriosa caratteristica di un Horror Game. Il progetto è stato realizzato per Oculus Rift, un recente dispositivo utilizzato per la realtà virtuale.
Il contesto della tesi è quello dello sviluppo di applicazioni grafiche interattive.
Si parlerà di:

- Strumenti utilizzati per la creazione del progetto
- Come è stato ideato il progetto e la sua progettazione
- Come è stato implementato il videogioco
- Quali possono essere degli sviluppi futuri

Il nome scelto per l'applicazione è "Catacomb", infatti il giocatore si troverà ad esplorare delle catacombe. "Catacomb" è un gioco 3D ed è giocabile su PC con sistema operativo Windows. In questo gioco l'utente tramite l'Oculus Rift potrà, camminare, guardarsi intorno ed interagire con gli oggetti. In questa tesi verrà mostrata la progettazione del gioco e come è stata creata la struttura per l'interazione tra giocatore e oggetti presenti nella scena.
L'ambiente di sviluppo utilizzato è Unity3D, un software completamente dedicato alla creazione di videogiochi 3D e 2D con un renderer grafico incluso ed un supporto per il multipiattaforma (PC / Mac / iOS / Android / Linux etc..).

La tesi è così raggruppata: il **capitolo 1** tratta principalmente di Unity3D la piattaforma di sviluppo utilizzata per la creazione del gioco. Verrà inoltre fatta un'introduzione all'interfaccia del programma e verranno illustrati i principali componenti indispensabili per la creazione di un qualsiasi videogioco. Verrà presentato l'Oculus Rift e verrà spiegato a grandi linee il suo funzionamento.

Nel **capitolo 2** viene trattato tutto il processo di progettazione che è stato utilizzato per rendere l'esperienza di gioco la più piacevole possibile. La fase di Design del livello e del gameplay.

Nel **capitolo 3** verrà mostrata anche con l'ausilio di esempi di codice la struttura principale che si occupa della gestione delle interazioni tra giocatore ed oggetti. Si mostrerà anche come è stata creata la scena del livello su Unity.

Nel **capitolo 4** si parlerà di come sia stato creato l'audio, dei possibili sviluppi del videogioco e verranno fatte delle considerazioni sul risultato finale ottenuto.

Capitolo 1

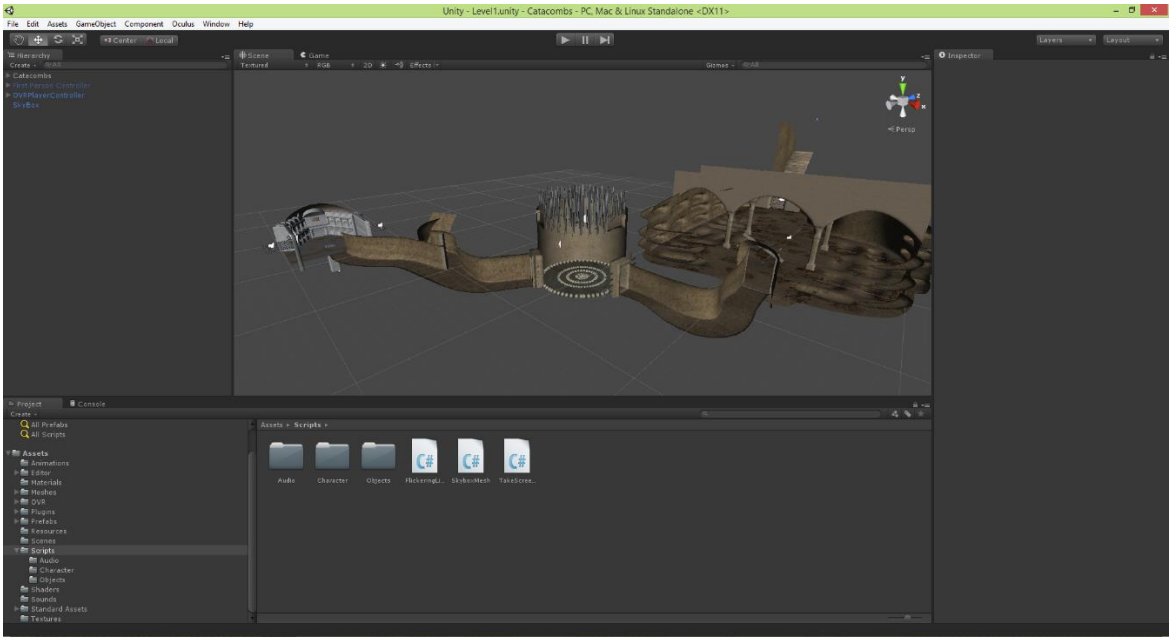
Introduzione a Unity3D

Unity3D è un ambiente di sviluppo per videogiochi, il suo workflow è molto semplice infatti è adatto per tutte le tipologie di Team, piccoli e grandi.
All'interno di Unity possiamo trovare tutti i principali tool che sono necessari per la creazioni di un videogioco come l'animation tool, il gestore di materiali, il tool per la creazione di un terreno, alberi etc…
Grazie al supporto di diverse librerie grafiche quali Direct3D, OpenGL e OpenGL ES, Unity permette di sviluppare applicazioni cross-platform. E' possibile creare applicazioni per Windows, Mac, Linux, Web, iOS, Android, Windows Phone, Wii, Xbox e Playstation. Ovviamente alcune delle licenze sono a pagamento o necessitano della versione PRO.
La versione base di Unity è completamente free ed è anche possibile pubblicare sui vari store senza comprare la versione PRO. La versione PRO differisce da quella free per alcune funzioni come la riflessione dell'acqua, ombre dinamiche, il profiler, utile per l'ottimizzazione del codice, gli effetti di postprocess e tante altre funzioni. C'è da chiarire che anche senza l'utilizzo di tutte queste funzioni è possibile creare un gioco di qualità, ma se si vuole passare alla versione PRO bisognerà pagare 1500$.

Interfaccia di Unity3D

L’area di lavoro di Unity è suddivisa in finestre:

- **Hierarchy** è la finestra che elenca gli oggetti presenti nella scena, grazie ad essa è possibile organizzare gli oggetti e selezionarli. Un oggetto può contenere altri oggetti al suo interno.
- **Scene** è la finestra che contiene la vista sulla scena attualmente in utilizzo, in questa finestra è possibile collocare, spostare, ruotare e scalare i vari oggetti nello spazio creando l’ambiente dove il giocatore andrà a muoversi. Questa finestra è impostata di default sulla vista 3D quindi si andrà a lavorare su x, y e z ma tramite un bottone in alto indicato con il simbolo 2D è possibile impostare la scena per una visualizzazione 2D nel caso in cui si voglia creare un gioco a due dimensioni.
- **Game** è la finestra che simula il gioco vero e proprio. Tutto quello che è visualizzato dalla camera viene reso visibile in questa finestra. Durante il testing di una scena si utilizza questa finestra.
- **L’Inspector** visualizza tutte le proprietà ed i componenti di un game object selezionato
- **Project** è la finestra dove è possibile visualizzare tutte le risorse del nostro progetto e organizzarle a nostro piacimento.
- La **Console** è la finestra dove vengono visualizzati tutti i messaggi di sistema o gli eventuali errori/warnings di compilazione.



Concetti fondamentali

Ogni progetto creato con Unity deve contenere tutte le risorse di cui l'applicazione ha bisogno, al suo interno possono essere definite nuove scene, scripts e prefabs.

Una scena è un ambiente virtuale isolato, in essa sono definiti tutti gli oggetti che compongono un livello. Implicitamente quando viene caricata una scena l'engine alloca tutti gli oggetti in memoria, nel caso in cui si cambia da una scena ad un'altra tutti gli oggetti della scena precedente vengono deallocati, esistono comunque diverse tecniche per mantenere dei game object tra una scena e l'altra.

In Unity prevalgono tre concetti fondamentali, *game object*, *component e prefab*. Il **game object** è la primitiva fondamentale dell'engine a cui si possono aggiungere dei componenti che ne specificano dei comportamenti e delle proprietà. Ogni game object ha un Transform, che definisce la posizione, rotazione e grandezza dell'oggetto secondo un sistema di coordinate xyz.

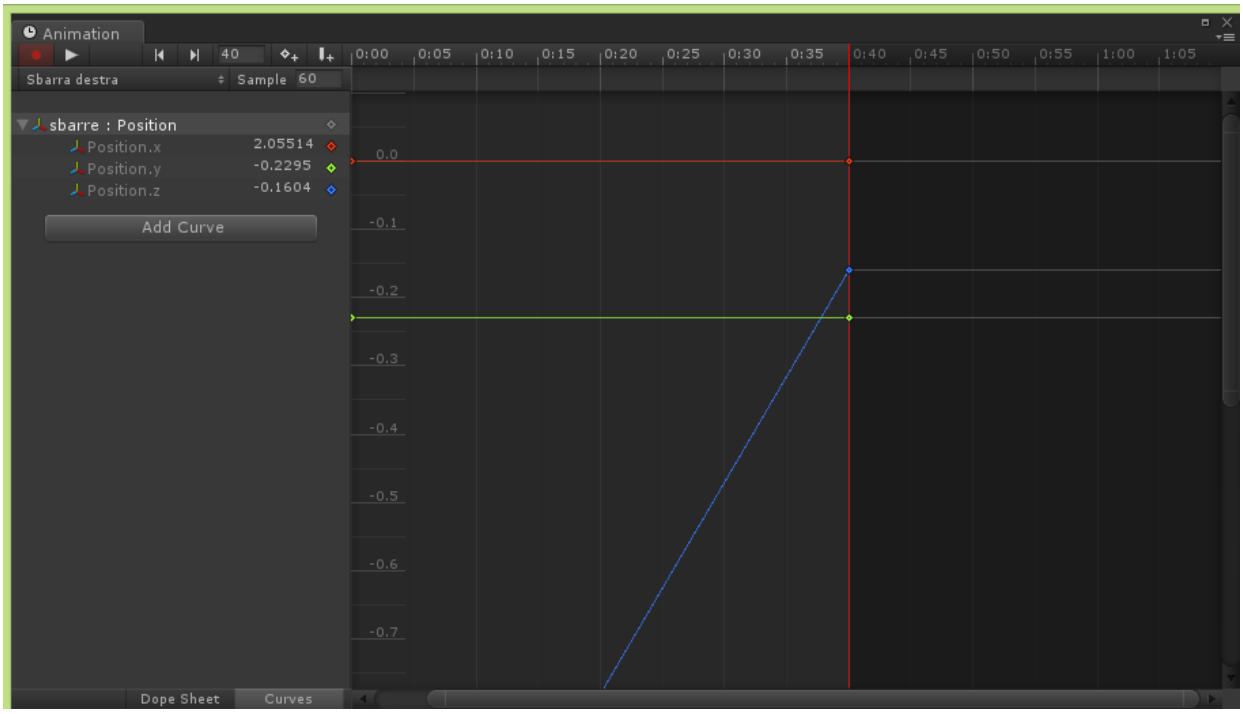
Il **component** è appunto un componente che va aggiunto ad un game object in base alle nostre esigenze andando così ad aggiungere proprietà specifiche

al nostro oggetto o comportamenti che il nostro game object deve avere all'avvenire di un qualche evento. I component più importanti che andremo ad utilizzare sono: Audio Source, Animation, Mesh Collider, Box Collider e Scripts (anche gli Scripts vengono considerati dei component).
E' possibile aggiungere ad un oggetto uno script semplicemente trascinando lo script dalla finestra project alla finestra hierarchy sull'oggetto desiderato. Dall'inspector poi è possibile abilitare, disabilitare o eliminare un component, come anche impostare il valore di una variabile all'interno di uno script.
Unity permette il salvataggio di oggetti creati nella scena tramite l'utilizzo dei **prefab**, elementi creabili dalla finestra project che formano un collegamento tra tutti i game object dello stesso tipo. Il vantaggio di utilizzare un prefab risiede non solo nel poter salvare all'interno del progetto i game object più comuni, ma anche nel permettere che una modifica effettuata su di esso si ripeta su tutte le copie di quell'oggetto, rendendo molto più facile la gestione delle copie.

Animation Tool

Nella creazione di Catacomb, il tool che è stato usato maggiormente è senz'altro l'Animation Tool. Questo tool permette di creare animazioni da assegnare poi ai vari gameobject. Con la parola animazione non si intende solamente la traslazione e la rotazione ma anche qualsiasi cambio di proprietà dell'oggetto: è possibile far cambiare il colore di un oggetto, è possibile cambiare la variabile pubblica di uno script e tante altre cose.
L'animation tool si presenta con principalmente due fogli di lavoro: il *Dope Sheet* e *Curves*. Il Dope sheet è semplicemente un foglio dove creando delle Key è possibile memorizzare dei cambi di proprietà nel tempo. Una volta create delle Key, e quindi dei cambi di proprietà, su Curves possiamo osservare la curva che è stata generata. In questo foglio di lavoro è possibile

modificare l'andamento dell'animazione nel tempo tramite l'utilizzo di strumenti come: smooth, flat e broken.



Scripting in Unity

Javascript, C# e Boo sono i tre linguaggi con cui è possibile lavorare su Unity. Nello stesso progetto è possibile utilizzare script con linguaggi differenti ma in questo progetto si è scelto di utilizzare solo script in C#. La parte di scripting di Unity è definita su MonoDevelop, un'implementazione open-source del .NET Framework. Nell'installazione di Unity è inclusa una versione proprietaria di MonoDevelop da poter utilizzare per scrivere il codice.

Funzioni Base

Ogni script per funzionare deve essere utilizzato come component e quindi "assegnato" ad un game object presente nella scena. Molte funzioni all'interno di uno script sono richiamate solo al verificarsi di un particolare evento relativo a quell'oggetto.

Quando si va a creare uno script nel progetto Unity crea una classe che deriva da MonoBehaviour. MonoBehaviour comprende tutte le funzioni più importanti e maggiormente utilizzate per la creazione di un videogioco:

- *Update:* Chiamata prima che un frame proceda nella fase di rendering, al suo interno è spesso specificata la parte di codice che definisce il comportamento dell'oggetto e/o la gestione degli input.
- *FixedUpdate:* Come per Update, ma viene chiamata prima di ogni step dell'engine fisico. Utilizzata per oggetti Rigidbody ossia quelli che sono affetti dalla fisica.
- *OnGUI:* Richiamata più volte per frame, utilizzata per disegnare a schermo l'interfaccia utente e gestire gli eventi relativi alla GUI (Graphics User Interface)
- *Start:* E' chiamata nel frame in cui lo script viene abilitato prima di qualsiasi altro metodo. La funzione Start viene chiamata una sola volta per tutta la durata dello script.
- *OnTriggerEnter:* Richiamata quando il Collider del game object a cui è "attaccato" lo script entra dentro un Trigger.
- *OnTriggerExit:* Come la precedente ma viene chiamata quando i collider non si intersecano più.
- *OnTriggerStay:* Richiamata ogni volta che è verificato che i due collider si intersecano. Si ripete ad ogni frame.
- *OnCollisionEnter:* Chiamata quando il collider/rigidbody ha iniziato a toccare un altro collider/rigidbody.
- *OnCollisionExit:* Come la precedente, chiamata quando il collider/rigidbody non tocca più un altro collider/rigidbody.
- *OnCollisionStay:* Chiamata ogni frame per il quale il collider/rigidbody sta toccando un altro collider/rigidbody.

Le variabili al di fuori delle funzioni sono inizializzate all'abilitazione dello script. Ogni variabile resa pubblica è resa visibile nell'inspector ed è possibile assegnare i valori di quest'ultime tramite l'interfaccia di Unity.

Componenti principali

Unity mette a disposizione tanti componenti allo sviluppatore molti di questi sono indispensabili per la creazione di un gioco altri servono per creare particolari effetti grafici. Da codice si possono raggiungere componenti attraverso la funzione *GetComponent* derivata da *GameObject*.

Alcune proprietà sono comuni a tutti i componenti come la proprietà *enabled,* che permette di attivare/disattivare un determinato componente. Alcuni dei componenti più importanti sono accessibili direttamente dal game object come fossero delle proprietà ad esempio: renderer, audio, animation, collider, rigidbody, etc…

Ora andremo ad analizzare più a fondo le componenti fondamentali:

- Transform: Come detto in precedenza, il Transform si occupa di mantenere tutte le informazioni riguardanti rotazione, posizione e dimensione. La sua classe comprende funzioni che permettono di effettuare trasformazioni sugli oggetti (Translate, Rotate, etc..).
- Renderer: Il renderer è la componente che rendere l'oggetto visibile sullo schermo disattivando questo component renderemo l'oggetti invisibile.
- AudioSource: E' la componente che simula le fonti di suoni in un mondo 3D. AudioSource è attaccato al game object per riprodurre suoni 3D e suoni 2D. I suoni 2D sono costanti: in qualsiasi luogo il giocatore si trovi nella scena il volume di riproduzione sarà sempre lo stesso. I suoni 3D invece simulano i comportamenti del suono nella vita reale: più ci si avvicinerà alla fonte del suono e più questo sarà forte, più ci si allontanerà e più tenderà ad attenuarsi.
- Camera: La Camera è il component che crea una finestra per guardare il mondo. Lo spazio dello schermo è definito in pixel: l'angolo in basso a sinistra corrisponde alle coordinate (0,0)mentre quello in alto a destra alle coordinate (pixelWidth, pixelHeight). La z indica la distanza dell'oggetto dalla camera.

- Collider: Attiva il collision detection per quell'oggetto permettendo di utilizzare funzioni come OnCollisionEnter. Ad ogni collider è assocata una mesh Unity di default utilizza un cubo ma si possono utilizzare anche sfere, cilindri etc.. Mettendo la spunta su isTrigger renderemo il collider un "interruttore" che non è affetto dalla fisica e che può essere gestito esclusivamente tramite le funzioni OnTrigger.
- RigidBody: Questo componente assegna il game object all'engine fisico. Aspetti quali gravità, accellerazione e attrito saranno quindi simulati per questo oggetto. I Rigidbody sono completamente personalizzabili: è infatti possibile definire il centro di massa, velocità, modalità di collisione, massa etc…

Se si vuole una lista completa dei components con le loro funzioni e proprietà è consigliabile guardare le Unity Scripting Reference.

Comunicazione tra Scripts

Per comunicare da uno script ad un altro esistono diversi metodi che adesso andremo ad analizzare. Uno dei metodi più comuni è quello dell' utilizzo del metodo *GetComponent.* E' possibile accedere ad un component e quindi accedere a tutte le funzioni e proprietà pubbliche utilizzando appunto il GetComponent che è un metodo derivato da game object. Ogni qualvolta si crea uno script nel progetto questo diventa un "tipo" identificato dal nome datogli. Basterà quindi semplicemente inserire il nome dello script come parametro al metodo GetComponent e saremo in grado di accedervi. Questo metodo può essere utilizzato per accedere a tutti i tipi di componenti.
In Unity esiste una funzione che permette di mandare messaggi ad un determinato game object inviado come parametro una stringa che contiene il nome di una funzione: se questa esiste verrà eseguita. Nel caso in cui siano presenti più scripts con la stessa funzione queste vengono eseguite contemporaneamente.

Esistono anche funzioni che permettono di ottenere i riferimenti di gameobject presenti nella scena. Find e FindGameObjectWithTag sono due funzioni della classe Gameobject che permettono uno di trovare un gameobject tramite il nome (Find) e l'altro tramite il tag (FindGameObjectWithTag).

Oculus Rift

L'Oculus Rift è uno schermo da indossare sul viso per la realtà virtuale. Le sue caratteristiche principali sono la bassa latenza e un ampio campo di visuale. Sviluppato da Oculus VR ha ottenuto un finanziamento di 16 milioni di dollari di cui 2,4 milioni dalla campagna di Kickstarter. L'Oculus Rift garantisce un'immersione a 360° nel gioco, soprattutto se accompagnato da un ottimo paio di cuffie: infatti è possibile ruotare la testa a 360° per potersi girare attorno. In sostanza tutti i movimenti fatti con la testa vengono perfettamente captati e riprodotti nell'Oculus.

L'Oculus è ancora un device in costruzione e vengono rilasciate periodicamente delle versioni aggiornate esclusivamente per gli sviluppatori. Il primo prototipo utilizzava uno schermo da 5.6 pollici ma dopo, la campagna avvenuta con successo su KickStarter, è stato deciso di passare ad uno schermo da 7 pollici. Lo schermo LCD ha una profondità di colore di 24 bit per pixel ed è abilitato alla stereoscopia 3D.

Il campo di visione è di oltre 90 gradi in orizzontale (110 gradi di diagonale), più del doppio rispetto agli altri dispositivi in competizione. L'attuale risoluzione è di 1280x800 (16:10 aspect ratio), quindi 640x800 per occhio (4:5 aspect ratio). Per la versione commerciale si vuole arrivare ad una risoluzione di 1920x1080.

Essendo l'Oculus ancora una tecnologia in sviluppo esistono dei piccoli problemi. Il problema principale è quello del "mal di mare": l'utilizzo dell'oculus soprattutto per le prime volte e per periodi lunghi può causare nausea, giramento di testa e vertigini. Questi

sintomi sono più o meno forti in base al soggetto ma col tempo svaniscono. Il Team di Oculus VR studiando un modo per ridurre al minimo la sensazione di malessere. Questa sensazione viene causata proprio dall'immersione a 360° nel gioco: quando si indossa l'oculus e nel gioco si cammina e ci si guarda intorno al cervello arriva l'input come se effettivamente la persona si stesse muovendo e questo crea questa sensazione di malessere.

E' previsto (ma non confermato) che la versione per il consumatore sia disponibile tra fine 2014 o inizio 2015.

Capitolo 2

Sviluppo dell'idea

Catacomb è principalmente una rivisitazione fatta appositamente per Oculus Rift di "Idol Hunter", un gioco già sviluppato dalla 48h Studio di cui faccio parte. Idol Hunter è un Horror/ Puzzle game dove il giocatore si trovava a prendere le parti di un esploratore in ricerca di tesori all'interno di templi maya. Il movimento del giocatore è a caselle con la possibilità di ruotare la visuale in 4 direzioni (nord, sud, ovest ed est) utilizzando un'interfaccia grafica che permette il movimento del personaggio.

Su Catacomb quello che si è voluto fare è cercare di far immedesimare al 100% il giocatore facendolo sentire un vero esploratore a caccia di tesori all'interno di catacombe pronte a difendere il loro tesoro.

Si è cercato soprattutto di creare una grafica quasi realistica che dia il senso di profondità su ogni piccolo particolare e di utilizzare degli effetti sonori che aumentino l'immedesimazione. L'idea di gioco è come detto prima quella di un esploratore alla ricerca di tesori all'interno di catacombe (ispirate a quelle italiane) cercando di risolvere i vari puzzle e di sopravvivere alle varie trappole posizionate in difesa della reliquia.

Il gioco è ambientato nelle catacombe, il movimento del giocatore è libero, l'interfaccia grafica è stata completamente rimossa e la grafica è completamente nuova e molto migliorata. Sostanzialmente si è preso il concept di Idol Hunter e si è creato un gioco quasi del tutto nuovo.

Inizialmente c’è stata una fase di Design del livello dove: è stata creata una mappa del livello, è stato definito il game play ed è stato creato il walkthrough. Poi si è scelta l’ambientazione e lo stile grafico da utilizzare, con l’ausilio di foto di reali catacombe.

Infine si è scelta la piattaforma di sviluppo. Abbiamo deciso di utilizzare Unity perché molto efficace e free.

Quanto appena descritto è un processo di sviluppo dell’idea di gioco di Catacombs. Diverse sono le domane che ci si deve porre prima dello fase implementativa ed una delle più importanti è quella riguardante la capacità del team; è effettivamente il team è in grado di sviluppare un’idea del genere e di mantenere un livello qualitativo elevato.

Specifiche del progetto

Nonostante Unity dia la possibilità di creare applicazioni per più piattaforme è stato scelto di sviluppare Catacombs solo per PC per due motivi: l'utilizzo dell'Oculus Rift e per non avere limitazioni per quanto riguarda le risorse.

Il Giocatore si troverà nei panni di un esploratore che dovrà superare le varie stanze sfuggendo alla morte per infine risolvere un enigma che lo farà arrivare al suo obbiettivo: il crocifisso d'oro. Il livello di gioco comprende 3 stanze: la prima è una stanza di introduzione, nella seconda il giocatore si ritroverà bloccato e dovrà cercare di scappare mentre il soffitto con degli spuntoni scende piano piano, nella terza stanza avrà una possibilità per risolvere l'enigma ed in caso di errore spunteranno degli spuntoni dal muro che determineranno la fine del gioco.

Nella Hierarchy della scena sono presenti 3 oggetti padre:

- Catacombs l'oggetto contenente tutte le mesh del livello
- OVRPlayerController: Questo oggetto è fornito dal dev kit dell'Oculus, ed include tutto il necessario per predisporre la scena per un gioco in prima persona dove si vuole utilizzare l'Oculus. Quindi gestisce il movimento e la rotazione della camera. Più avanti approfondiremo questo oggetto.
- SkyBox: E' un oggetto che simula il cielo.

Design & Gameplay

Una fase importantissima prima di iniziare la fase di sviluppo è quella della definizione del gameplay, ossia pianificare fin nei minimi dettagli cosa deve fare il giocatore per arrivare alla fine del livello e cosa può fare. Questa fase serve principalmente in progetti più complessi per capire se c'è dinamicità nel gioco e se è sufficiente il materiale progetto o bisogna ampliarne qualche aspetto. Questa fase nell'ambito dei videogiochi a livello commerciale può essere determinante per il successo ma spesso viene sottovalutata.

La definizione del Gameplay è una parte di un intero processo chiamato *Level Design* e viene svolto da una persona, il *Level Designer*.

Level Designer

Il compito del level designer è appunto quello di progettare su carta o tramite l'utilizzo di appositi strumenti il livello o i livelli, indicando minuziosamente cosa il giocatore DEVE e PUO' oltre allo stile grafico (se necessario) o la presenza di un determinato oggetto. Nel caso in cui non vengano indicati dal Level Designer stili grafici o oggetti sta alla bravura del grafico creare il luogo più adatto al livello corrente.
Il Level Designer potrebbe anche essere portato in alcuni casi alla creazione del walkthrough, documento contenente quello che il giocatore può/deve fare però in maniera più narrativa e descrittiva.

Ecco un esempio di walkthrough utilizzato per Catacombs:

> *L'esploratore apre gli occhi e si trova all'ingresso della catacomba, si volta a destra e nota un giaciglio incavato nel muro con uno scheletro vestito da frate. La Stanza è piena di piccole tombe incavate nel muro, per terra si vedono ossa e teschi che spuntano dal terreno. Nella catacomba pervade l'oscurità, l'esploratore è in grado di vedere a malapena a qualche metro di distanza. L'esploratore si inoltra nella stanza e intravede un piccolo corridoio che lo porta ad una stanza circolare*

ma più piccola della precedente. Una volta all'interno l'esploratore viene bloccato, le porte vengono sbarrate e non c'è via d'uscita. Si sentono rumori di ingranaggi, l'esploratore alza gli occhi e nota il soffitto pieno di spuntoni scendere piano piano; si guarda intorno e nota che sta camminando su un pavimento fatto di ossa e di teschi come se qualcuno volesse far capire all'esploratore che in tanti hanno tentato di ottenere quel crocefisso ma nessuno c'è riuscito. L'esploratore con tutta calma inizia ad analizzare la stanza e nota un mattone nel muro più sporgente di altri. Lo preme e il rumore di ingranaggi si blocca e si sente una porta aprirsi. Un altro corridoio che porta ad un bivio, a destra la strada si interrompe per colpa di una frana. Il corridoio lo porta in una stanza che sembra importante, nota delle piastrelle ed una serie di tombe in pietra incavate nel muro, sicuro di qualcuno di importante. Nota anche nel pavimento un'incisione con un simbolo su una lastra di pietra. Nel muro sei simboli con sotto dei mattoni sporgenti a mò di bottoni e sotto ancora dei buchi. 5 simboli errati 1 simbolo giusto capisce l'esploratore, nota che uno dei simboli è lo stesso proposto nella lastra di pietra per terra con l'incisione sopra. Preme il mattone sotto quel simbolo e sente che qualcosa si muove. La lastra per terra inizia a salire fino a che non si nota un'incavatura con il crocefisso appoggiato.

Questo processo ha più senso ovviamente in progetti più complessi di quello trattato in questa tesi.

Problematiche principali

In un gioco 3D ci sono 3 principali problematiche che necessitano di una struttura logica di funzionamento: il movimento del giocatore, l'interfaccia grafica (GUI) e le interazioni tra oggetti/oggetti, oggetti/giocatore e giocatore/oggetti.

La GUI in questo gioco è stata completamente eliminata perché andrebbe a diminuire l'effetto di realtà virtuale e poi anche perché è sconsigliabile l'utilizzo di un'interfaccia grafica con l'oculus a causa dei due schermi.

Come detto in precedenza il Developer kit dell'Oculus mette a disposizione un prefab preimpostato per un gioco in prima persona in grado di camminare, correre e ruotare la testa. Per il nostro progetto abbiamo utilizzato questo oggetto che poi andremo ad analizzare.

Per quanto riguarda invece le interazioni tra oggetti e giocatore si è cercato di creare una struttura il più generica possibile che fosse in grado di "generalizzare" un'interazione.

Con "generalizzare" un'interazione si intende che ogni tipo di input (in questo caso il Click, StepIntoCollider e StepOverCollider) funzioni allo stesso modo. Capiremo meglio più avanti nel capitolo.

Movimento del giocatore

Il giocatore può muoversi tramite l'utilizzo della tastiera con i bottoni W,A,S,D oppure con le freccette direzionali. Tenendo premuto il tasto Shift il giocatore sarà in grado di correre. Inoltre nel caso non si voglia muovere la testa, si possono utilizzare i bottoni Q ed E per ruotare la camera a destra e a sinistra. Ora andiamo ad analizzare i punti principali dello script.

Gestione degli input

```
// WASD
if (Input.GetKey(KeyCode.W)) moveForward = true;
if (Input.GetKey(KeyCode.A)) moveLeft    = true;
```

```
if (Input.GetKey(KeyCode.S)) moveBack          = true;
if (Input.GetKey(KeyCode.D)) moveRight         = true;
// Arrow keys
if (Input.GetKey(KeyCode.UpArrow))    moveForward = true;
if (Input.GetKey(KeyCode.LeftArrow))  moveLeft          = true;
if (Input.GetKey(KeyCode.DownArrow))  moveBack       = true;
if (Input.GetKey(KeyCode.RightArrow)) moveRight   = true;
```

GetKey è un metodo della classe Input e il suo compito è quello di riscontrare la pressione di un bottone che viene identificato con il KeyCode. Nel caso in cui uno di questi bottoni venga premuto, una variabile booleana viene messa a true per capire in che direzione il giocatore vuole andare. Per capire la direzione basta semplicemente un if che controlli quali variabile sia a true.

```
if ( (moveForward && moveLeft) || (moveForward && moveRight) ||
        (moveBack && moveLeft)    || (moveBack && moveRight) )
                    MoveScale = 0.70710678f;
```

La velocità massima raggiungibile dal giocatore nelle varie definizione è definita come segue

---

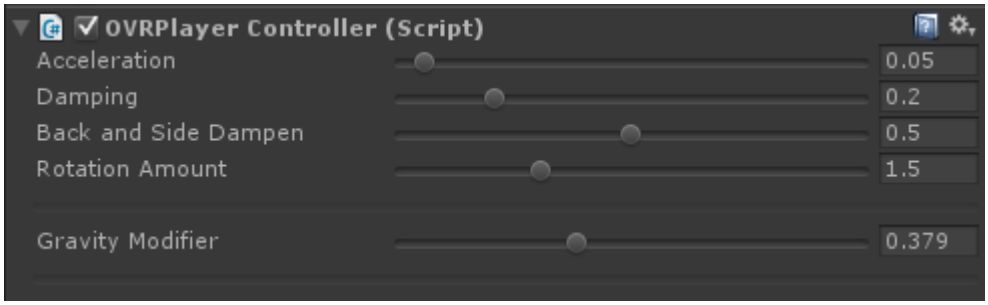
```
if (moveForward)
MoveThrottle += DirXform.TransformDirection(Vector3.forward * moveInfluence);
if (moveBack)
MoveThrottle+=DirXform.TransformDirection(Vector3.back*moveInfluence)* BackAndSideDampen;
if (moveLeft)
MoveThrottle += DirXform.TransformDirection(Vector3.left * moveInfluence) * BackAndSideDampen;
if (moveRight)
```

MoveThrottle+=DirXform.TransformDirection(Vector3.right * moveInfluence) * BackAndSideDampen;

Il MoveThrottle ossia la velocità massima è data dalla direzione che si sta seguendo * per un fattore moveInfluence che è dovuto dall'accelerazione moltiplicati per il MoveScale che vale 1 se si va dritti e 0.70 se si va a destra, sinistra e indietro. Se si va a destra o a sinistra o indietro bisogna moltiplicare il tutto per BackAndSideDampen che sarebbe un moltiplicatore che andrà a ridurre la velocità di un 50%.

Il movimento è completamente personalizzabile infatti ci sono dei valori che possiamo aggiustare in base alle nostre preferenze:

- *Acceleration*: è un moltiplicatore che più è grande più il giocatore andrà veloce.
- *Damping*: è un valore che simula l'attrito. Più questo valore sarà alto più il terreno farà resistenza.
- *BackAndSideDampen*: quanto vogliamo(in percentuale) che il giocatore si rallenti camminando lateralmente e all'indietro.
- *JumpForce*: indica la potenza del salto, quindi quanto arriverà in alto. (in Catacombs il giocatore non può saltare)
- *RotationAmount*: la velocità di rotazione utilizzando i tasti Q ed E
- *GrafityModifier*: Un fattore che influenza il salto, simulando la gravità terrestre.



Interazione tra oggetti e giocatore

Per l'interazione tra oggetti (il giocatore rimane un oggetto) ho creato due classi principali, una classe Trigger derivata da MonoBehaviour e una classe ActionBase anch'essa derivata da MonoBehaviour. La struttura studiata per l'interazione funziona come segue.
Un oggetto è definito oggetto Trigger quando l'attivazione di quest'ultimo provoca l'attivazione di uno o più oggetti (se stesso incluso). Con attivazione si intende un qualsiasi cambio di proprietà di un oggetto, come ad esempio un'animazione, una riproduzione di un suono, un cambio di posizione etc…
Un oggetto trigger può essere attivato tramite:

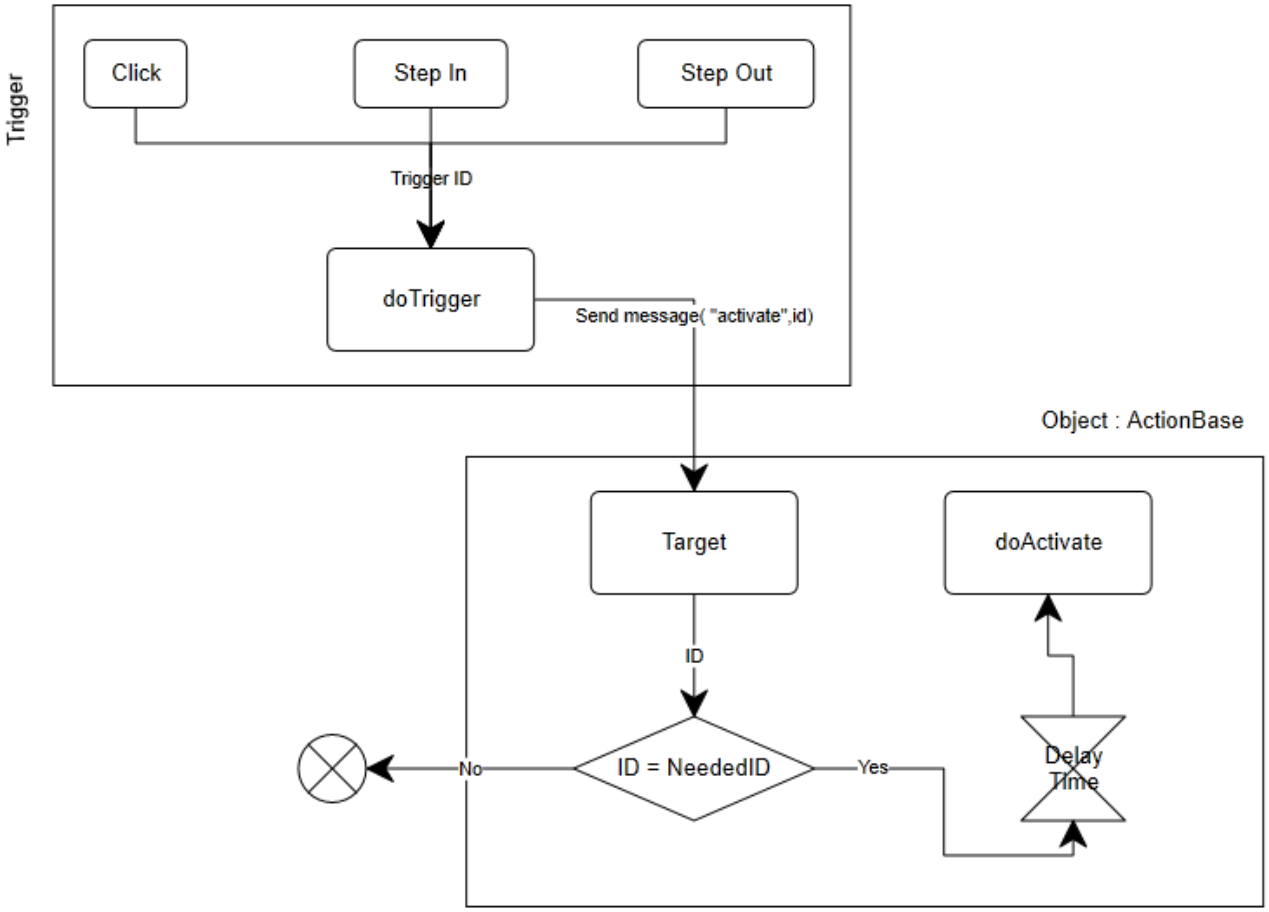
- *Click*: Quando il giocatore clicca l'oggetto.
- *StepIn*: Quando un oggetto entra dentro il collider del trigger. (L'oggetto deve avere un collider attaccato)
- *StepOut*: Quando un oggetto esce dal collider del trigger.

Ogni oggetto trigger ha un attributo pubblico, il TriggerID, che serve per identificare il Trigger. Una volta attivato il Trigger questo invia un messaggio a tutti gli oggetti target passando come attributo il TriggerID.

Quanto appena detto è in sostanza il funzionamento generale di un oggetto Trigger.
La classe ActionBase invece l'unica cosa che fa è quella di controllore se l'ID del trigger di cui ha ricevuto il messaggio equivale a quello di cui l'oggetto necessita: in caso positivo l'oggetto attende x secondi (se ce ne sono da aspettare) e poi effettua il doActivate, ossia la funzione che determina le azioni da compiere.

Gli oggetti che devono compiere un'azione dopo l'attivazione di un trigger hanno attaccato una classe che è derivata da ActionBase e non ActionBase stessa.

Di seguito un piccolo diagramma a blocchi che ci può aiutare per la comprensione della struttura progettata per l'interazione con gli oggetti.



Capitolo 3

Implementazione del progetto

Classe HitObject

HitObject è una classe molto semplice che si occupa di controllare se nella direzione in cui il giocatore sta guardando ci sia o meno un oggetto con un collider attaccato, il tutto quando il giocatore preme il tasto sinistro del mouse.

Per l'implementazione di questa classe abbiamo usato il metodo RayCast della classe Physics. Il RayCast è un raggio che viene sparato da una origine verso una direzione per una distanza x ed il suo compito è quello di determinare se colpisce o meno dei collider. Questo metodo può prendere in ingresso svariati parametri, nel nostro caso diamo in pasto al metodo RayCast un Vector3 origine, un Vector3 destinazione, una variabile hit di tipo RayCastHit dove verranno immagazzinate tutte le informazioni del collider colpito (se ne viene colpito uno) e la distanza.

Tutto questo ovviamente fatto all'interno della funziona Update perché abbiamo la necessità di controllare ad ogni frame se il giocatore ha premuto o meno il tasto del mouse.

```
public class HitObject : MonoBehaviour {
void Update ()
{
        if(Input.GetButtonDown("Fire1")) //Se click sinistro
        {
                RaycastHit hit;
                if(Physics.Raycast(this.transform.position,this.transform.forward,out hit,1))
                {
                        hit.collider.gameObject.SendMessage("clickedWorld"); //sendmessage a hit
                }
        }
}}
```

Nel caso in cui un oggetto con un collider attaccato venga colpito dal RayCast tutte le sue informazioni vengono salvate nella variabile hit per poi utilizzarle per inviare un messaggio all'oggetto per notificarlo che è stato attivato dal giocatore.

Classe Trigger

Un oggetto Trigger può essere attivato in 3 modi Click, StepOn e StepOff. Il compito di questa classe è quello di accorgersi quando viene attivata e inviare a tutti i suoi oggetti target un messaggio di attivazione con l'id.

In questa classe ci sono 3 importanti variabili pubbliche:

- TriggerID di tipo int che contiene l'identificatore del trigger.
- Targets, una lista di tipo GameObject che contiene le reference di tutti gli oggetti che devono essere attivati all'attivazione di questo trigger.
- TriggerMode è una variabile di tipo enumTriggerModes che contiene la modalità di attivazione supportata.

```
public int triggerID;
public List<GameObject> targets;
public enumTriggerModes TriggerMode;
public enum enumTriggerModes
{
        Click,
        StepOn,
        StepOff
}
```

Una variabile enum è una variabile di tipo dato che consiste in un insieme di valori chiamati elementi, membri o enumerators. In questo caso il nostro TriggerMode definisce

il comportamento del nostro trigger: se questo può essere attivato con un click o passandoci sopra o uscendo dal trigger.

Ecco come le variabili pubbliche si presentano nell'inspector:

Essendo Targets una lista, bisogna prima definire la lunghezza e poi trascinare negli appositi spazi i GameObject target. TriggerMode invece se viene selezionato si aprirà una tendina con appunto i 3 valori della modalità di trigger (Click, StepOn, StepOff).

```
public void clickedWorld()
{
        if (TriggerMode == enumTriggerModes.Click)
                activation();
}
public void OnTriggerEnter(Collider col)
{
        if (TriggerMode==enumTriggerModes.StepOn)
                activation();
}
public void OnTriggerExit(Collider col)
{
        if(TriggerMode==enumTriggerModes.StepOff)
                activation();
}
```

Quanto riportato sopra sono le 3 funzioni che vengono richiamate quando uno delle 3 modalità si verifica. Come visto in precedenza quando il giocatore clicca e colpisce un oggetto con un collider questo manda un messaggio di "clickedWorld" e se l'oggetto colpito è un Trigger(e quindi ha lo script Trigger attaccato a se stesso) questa funzione viene richiamata. Nel caso del click, si controlla se la modalità di attivazione di questo oggetto è uguale alla modalità che effettivamente lo ha attivato: in caso positivo si richiama la funzione activation che andra ad inviare a tutti gli oggetti target il messaggio di attivazione.

Per lo StepOn e StepOff vengono utilizzate invece delle funzioni della classe MonoBehaviour. Come detto nell'introduzione OnTriggerEnter e OnTriggerExit sono due funzioni che vengono eseguiti quando un collider (qualunque esso sia) entra in collisione con un trigger o esce dalla collisione con un trigger.

Per questo motivo viene fatto il controllo della modalità di attivazione con la modalità effettiva, visto che può capitare che un oggetto attivabile solo con il click entri in collisione con il giocatore e quindi viene eseguita la funzione OnTriggerEnter, cosa che non vogliamo che accada.

```
void doTrigger(int id)
{
	if(targets.Count<=0) //Se non ci sono Target si invia un messaggio di attivazione a se stesso
	{
		SendMessage("activate",id);
	}
	foreach(GameObject a in targets) //Se ci sono Targets si invia un messaggio ad ogni oggetto
	{
		if(a)
		{
			a.SendMessage("activate",id);
		}
	}
}
```

```
public void activation()
{
        doTrigger (triggerID); //Richiama la funzione Activation inviando come parametro l'ID
}
```

Nel caso in cui non ci siano target nella lista l'oggetto invierà un messaggio di attivazione a se stesso, questo perché come detto in precedenza un oggetto può essere un trigger ma allo stesso tempo dover compiere un'azione.

Classe ActionBase

Il Trigger invia un messaggio di attivazione a tutti i target e questi ultimi per eseguire un'azione hanno bisogno di uno script derivato da ActionBase. ActionBase è la classe base che definisce se all'arrivo di un messaggio di attivazione da un trigger l'oggetto deve fare un'azione oppure no. In sostanza quello che fa la classe base è controllare se il messaggio arrivato è da un trigger consono, controllare se l'oggetto può essere attivato in base al numero massimo di attivazioni, aspettare x secondi nel caso in cui la variabile DelayTime sia maggiore di zero ed infine richiamare la funzione doActivate.

```
public class ActionBase : MonoBehaviour {
        public float DelayTime;
        public int triggerID=-1;
        public int maxActivations=-1;
        private int currentActivations=0;

        public void activate(int ID)
        {       //Controllo ID e MaxActivation se si Aspetto DelayTime e  chiamo doActivate
        if((triggerID<0||triggerID==ID)&&(maxActivations<0||currentActivations<maxActivations))
        {
                if(DelayTime>0)
                        StartCoroutine(delayedAction());
                else
                        doActivate();
```

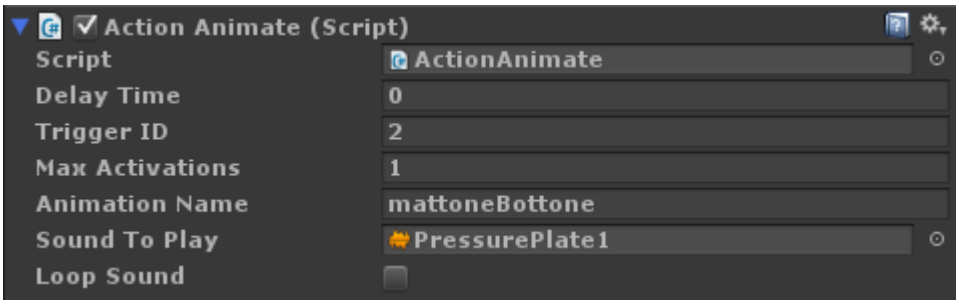
```
            currentActivations++;  //Incremento il numero di attivazioni
            }
        }

        IEnumerator delayedAction()
        {
            yield return new WaitForSeconds(DelayTime);  //Aspetto DelayTime secondi
            doActivate();
        }
        protected virtual void doActivate()
        {
            Debug.Log("Activated "+transform.name);
        }
}
```

IEnumerator delayedAction() è una Coroutine che viene eseguita per aspettare x secondi: in questo lasso di tempo è come se lo script si fermasse. Tutti gli oggetti che hanno uno script che è derivato da ActionBase ereditano come valori:

- DelayTime: è un valore float che indica i secondi da attendere prima di compiere un'azione
- MaxActivation: Il numero massimo di attivazioni per quell'oggetto, dopo di che l'oggetto non si attiverà più
- TriggerID: è l'ID del trigger abilitato all'attivazione di questo oggetto. Se il triggerID è -1 questo significa che qualsiasi trigger che manda un messaggio a questo oggetto lo attiva.

Ecco come si presenta uno script derivato da ActionBase nell'inspector:



Un esempio di script derivato da ActionBase è quello di ActionAnimate, uno script che è stato utilizzato più volte nella creazione di Catacomb.

Action Animate

ActionAnimate è una classe derivata da ActionBase che gestisce la riproduzione di un animazione e la riproduzione del suono dell'animazione.

```
public class ActionAnimate : ActionBase {
        public string animationName;
        public AudioClip soundToPlay;
        public bool loopSound = true;
        private bool _playing;
        protected override void doActivate()
        {       animation.Play(animationName);
                _playing = true;
                if (soundToPlay && audio ) {
                        audio.clip = soundToPlay;
                        audio.loop = loopSound;
                        audio.Play (); }
        }}
```

Questo semplice script avvia un'animazione ed il suo suono accedendo alla componente audio e animazione già attaccate all'oggetto.

Classe StepManager

Considerato il fatto che abbiamo creato un gioco per realtà virtuale abbiamo cercato di aumentare il più possibile il realismo e quindi è stato deciso di creare una classe che simulasse il suono dei passi di una persona quando cammina o corre.
Con la classe StepManager abbiamo cercato di avvicinarci il più possibile alla realtà ecco come funziona la classe:

```
public class StepManager : MonoBehaviour {

    float stepTimer=0.0f;
    float stepCool=0.6f;
    public AudioClip stepSound;
```

Le variabili pubbliche in questa classe sono:

- stepTimer: è un timer che se ha valore 0 indica che può essere riprodotto il suono di un passo, altrimenti significa che è appena stato riprodotto un passo.
- stepCool: indica quanti secondi devono passare prima che sia possibile riprodurre un altro passo. In questo caso è stato scelto di riprodurre un passo ogni 0.6 secondi.
- stepSound: è una variabile che mantiene al suo interno le reference al suono da riprodurre.

Nella funzione Update dobbiamo andare a fare vari controlli. Il primo e più importante è quello di capire se il giocatore è vivo o morto. Lo script OVRPlayerController è stato appositamente modificato per fare in modo che una volta che il giocatore muore questo non sia più in grado di muoversi. Questo è stato fatto inserendo un controllo prima dei

calcoli per il movimento con una variabile booleana che determina lo stato del giocatore: in caso questa variabile sia impostata a true il giocatore non sarà più in grado di muoversi. Quindi è bastato utilizzare il metodo GetComponent per poter accedere alla variabile pubblica “morto” e controllare il valore di essa.

Fatto ciò prima di poter riprodurre un suono bisogna ovviamente controllare se il giocatore si sta muovendo oppure no: questo è stato fatto controllando tramite la classe Input. Come è possibile notare dal codice qui sotto oltre alla condizione in OR c’è un’altra condizione che deve essere verificata, ossia se in quel frame è già in riproduzione un suono. Questo evita di riprodurre per lo stesso passo due suoni.

```
if(!this.gameObject.GetComponent<OVRPlayerController>().morto)
{
if((Input.GetButton( "Horizontal" ) || Input.GetButton(“Vertical")&&!audio.isPlaying)
{
        audioStep();
}
```

La funzione audiostep() riproduce il suono del passo e resetta lo stepTimer nel caso in cui quest’ultimo abbia valore 0

```
void audioStep()
{
        if(stepTimer == 0)
        {
                audio.PlayOneShot(stepSound);
                stepTimer=stepCool;
        }
}
```

Ovviamente come è possibile intuire la riproduzione del suono di un passo è più frequente nel caso in cui si corra e meno frequente nel caso in cui si cammini. La condizione proposta prima trattava solo il caso della camminata e in caso positivo riproduceva il suono con un StepCool con valore 0.6. Qua sotto si ha invece la condizione che gestisce la corsa (è possibile correre semplicemente tenendo premuto il tasto ShiftSinistro).

```
if((Input.GetButton("Horizontal")||Input.GetButton(“Vertical"))&&Input.GetKey(KeyCode.LeftShift )
&& !audio.isPlaying)
{
        stepCool=0.4f;
        audioStep();
}
else
{
        stepCool=0.6f;
}
```

Nel caso in cui si corra, per rendere più frequente la riproduzione, viene abbassato lo stepCool a 0.4.

Qua sotto si può vedere invece la gestione del timer. La funzione Time.deltaTime restituisce semplicemente il valore in secondi che si impiega a completare un frame che va sottratto a stepTimer. Sottraendo ad ogni frame Time.deltatime arriveremo ad un momento in cui Steptimer arriverà a 0 e quindi sarà possibile riprodurre il suono.

```
if(stepTimer > 0)
        stepTimer-=Time.deltaTime;
if(stepTimer < 0)
        stepTimer=0;
```

Per maggior chiarezza propongo la classe StepManager per intero:

```
void Update () {
if(!this.gameObject.GetComponent<OVRPlayerController>().morto) {
        if((Input.GetButton( "Horizontal" ) || Input.GetButton( "Vertical" ))&& !audio.isPlaying) {
                audioStep();
        }
        if ( (Input.GetButton( "Horizontal" ) || Input.GetButton( "Vertical" ))&&
        Input.GetKey(KeyCode.LeftShift ) && !audio.isPlaying) {
                stepCool=0.4f;
                audioStep();
        }else{
                stepCool=0.6f;
        }
        if(stepTimer > 0)
                stepTimer-=Time.deltaTime;
        if(stepTimer < 0)
                stepTimer=0;
}}

void audioStep(){
if(stepTimer == 0){
                audio.PlayOneShot(stepSound);
                stepTimer=stepCool;
        }
}
```

Classe TriggerDeath

La classe TriggerDeath è quella che notifica al giocatore che è stato ucciso. Tramite la funzione OnTriggerEnter si controlla se il collider con cui il giocatore ha colliso ha il tag impostato a Trap.

```
public class TriggerDeath : MonoBehaviour {
        public AudioClip deathSound;
        public void OnTriggerEnter(Collider col) {
                if(col.transform.tag=="Trap") {
                        SendMessage("die");
                        audio.PlayOneShot(deathSound);
                        StartCoroutine(delay());
                }
        }
}
IEnumerator delay() {
        yield return new WaitForSeconds(2.0);
        Application.loadedLevel();
}
```

E' possibile accedere alle informazioni del collider con cui siamo entrati in collisione tramite l'utilizzo di col che è un parametro in ingresso della funzione OnTriggerEnter. Nel caso in cui col abbia il tag impostato a Trap. Significa che siamo appena entrati in collisione con una trappola e quindi notifichiamo la morte del giocatore. Questo è stato

fatto semplicemente inviando un messaggio con parametro die che andrà appunto a richiamare la funzione die all'interno dell'OVRPlayerController che imposterà il valore di morto a true bloccando così ogni possibilità di movimento del giocatore. La coroutine delay, dopo che è stata lanciata, aspetta 2 secondi e poi ricarica il livello che si stava giocando.

Classe FlickeringLight

L'esploratore entra nella catacomba con una torcia e FlickeringLight è la classe che ne simula lo sfarfallio. Questo perché nella realtà la luce emessa da una fiamma non è costante.

Il componente che viene utilizzato per la generazione della luce è una pointLight. Questo componente può essere posizionato nello spazio e genera luce dal suo punto di origine fino ad un range massimo. La pointlight è completamente personalizzabile: infatti possiamo cambiare colore, range, intensita e moltre altre cose.

Lo script si presenta con diverse variabili pubbliche che servono appunto per personalizzare la luce nella maniera più opportuna.

Min & Max Range sono il raggio d'azione minimo e massimo della luce, Min & Max Intensity sono l'intervallo di valori che può assumere l'intensità della luce, Max Movement è di quanto si sposta la luce e Update Time è il valore in secondi di ogni quanto si devono generare nuovi valori.

Ora andiamo ad analizzare come lavora lo script:

```
void Start () {
        myLight=light;
        myTransform=transform;
        startingPosition=transform.localPosition;
        app=Vector3.zero;
```

```
StartCoroutine(flicker());    }
IEnumerator flicker () {
        while(true)
        {
                myLight.range=Random.Range(minRange,maxRange);
                myLight.intensity=Random.Range(minIntensity,maxIntensity);

                app.x=maxMovement*Random.Range(-1f,1f);
                app.y=maxMovement*Random.Range(-1f,1f);
                app.z=maxMovement*Random.Range(-1f,1f);

                myTransform.localPosition=startingPosition+app;

                yield return new WaitForSeconds(updateTime);
        }
```

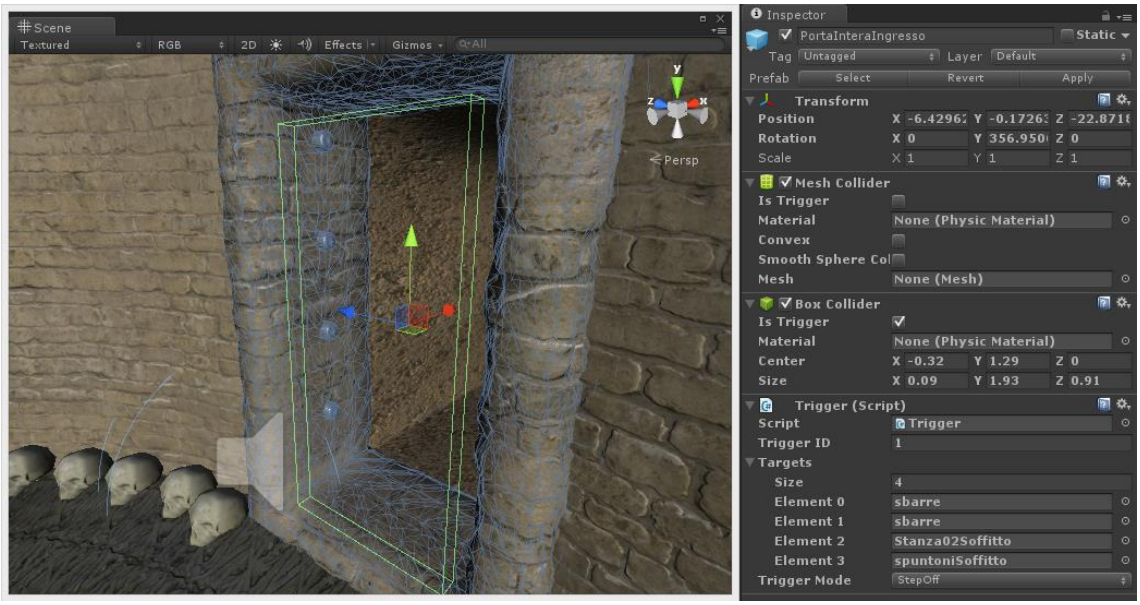
Nella funzione Start vengono inizializzate tutte le variabili necessarie per fare i calcoli. La coroutine che a noi ci interessa particolarmente è flicker(). Tutte le operazioni vengono fatte all'interno di un while(true) perché vogliamo che si continui a generare valori fino a che l'oggetto viene distrutto. Tutte le operazioni che vengono fatte sono semplicemente delle generazioni casuali di valori all'interno di un rage ed assegnate al component light.

Impostazione della scena

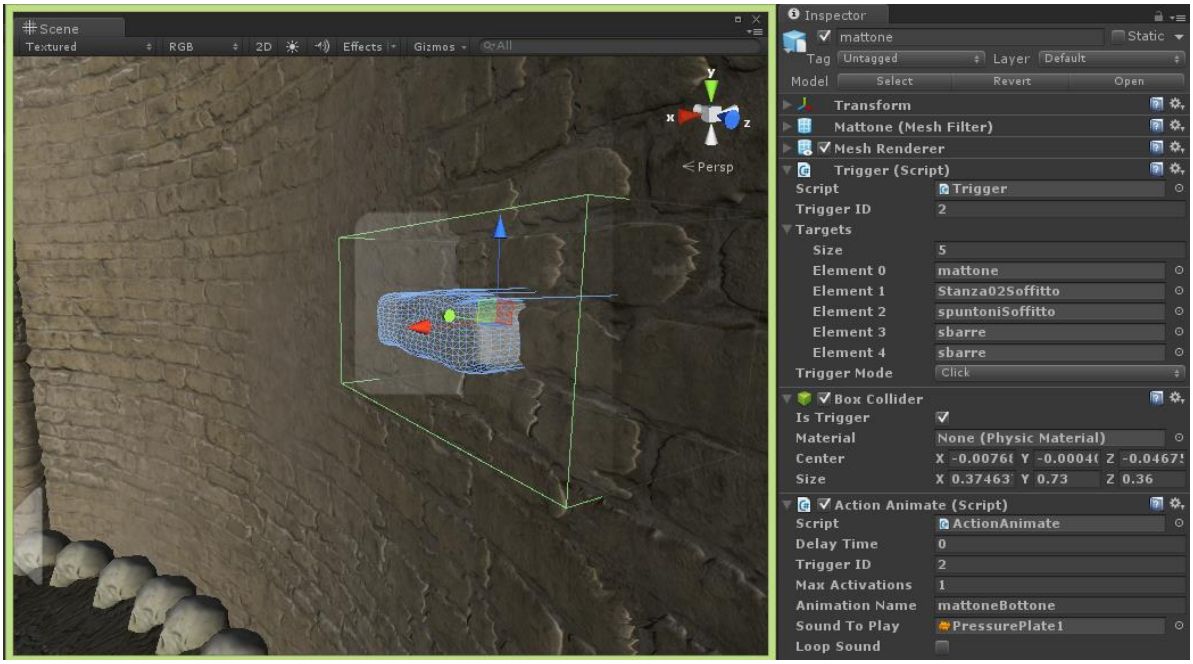
Una volta che gli assets grafici sono stati costruiti è bastato semplicemente trascinare la mesh all'interno della scena per posizionarla. Inizialmente è stato aggiunto a tutte le mesh un meshCollider per far in modo che il giocatore non le attraversi e successivamente sono stati attaccati agli oggetti trigger ed agli oggetti action tutti i vari componenti necessari, inclusi gli script. Per gli oggetti che interagiscono con il giocatore è stato cambiato il meshCollider con un boxCollider: questo perché il meshCollider è ottimo per le mesh come pavimenti, colonne, muri e soffitti ma per oggetti piccoli con forme un po' particolari potrebbe creare problemi.

Nella seconda stanza, quando l'esploratore passa attraverso la porta, quello che deve succedere è che: la porta alle spalle dell'esploratore deve chiudersi ed il soffitto con gli spuntoni deve iniziare a scendere verso il pavimento e alla pressione di un mattone particolare nel muro il soffitto deve fermarsi e la porta deve aprirsi.

Per fare tutto ciò abbiamo iniziato con l'inserire un boxCollider grande tutta la porta di ingresso dove sopra ci abbiamo attaccato il componente Trigger con triggerMode=StepOff. In questo modo quando l'esploratore esce dal trigger invia un messaggio di attivazione alla porta ed al soffitto con gli spuntoni.



Una volta impostata la porta ci dobbiamo preoccupare di programmare il mattone che blocca il soffitto. Quello che dobbiamo fare è aggiungere un BoxCollider, un componente Trigger ed un ActionAnimate. Il collider serve per interagire con il giocatore, il Trigger per inviare un messaggio a se stesso, al soffitto con gli spuntoni ed alla porta, infine l'ActionAnimate serve per far animare il mattone come se fosse un bottone.



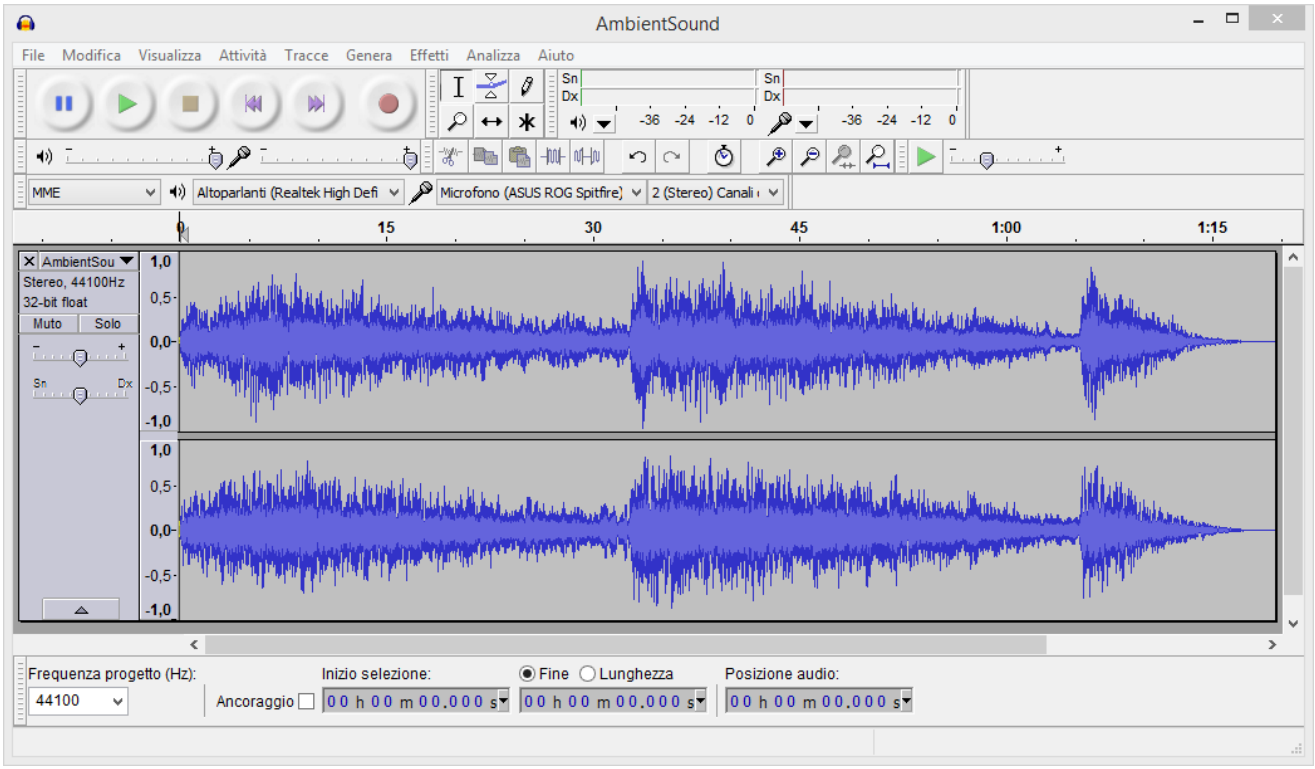
Per quanto riguarda il soffitto con gli spuntoni, essendo un oggetto action, è stato necessario innanzi tutto mettere il tag dell'oggetto a Trap, poi si è aggiunto il BoxCollider e due ActionAnimate: questo perché il primo ActionAnimate attiva l'animazione del soffitto e viene attivato dal Trigger della porta di ingresso mentre il secondo ferma l'animazione e viene attivato dal Trigger del mattone.

Grafica e Suono

La parte di grafica è stata trattata dal mio collega Tommaso Pantaloni.
Tutti i suoni utilizzati, ambientali e non, sono stato presi da una sito online "FreeSound". Questa piattaforma mette a disposizione a tutti gli sviluppatori di videogiochi musiche,

suoni ambientali, effetti sonori e quant'altro. Ovviamente è tutto gratis: unica cosa che si deve fare è che nel caso di una pubblicazione del gioco si deve scrivere nei crediti che i suoni sono stati scaricati da questa piattaforma.

I suoni scaricati sono stati tutti adeguatamente tagliati ed equalizzati. Il tutto è stato fatto con l'ausilio di un programma free: Audacity.



Audacity è un prorgamma di editing audio che nonostante sia un programma free, offre tutti gli strumenti necessari per la creazioni di effetti adeguati alle nostre esigenze.

Sviluppi Futuri

"Catacomb" può essere definito un prototipo a tutti gli effetti, giocandoci riusciamo a capire che tipo di gameplay avrà e quali sono i suoi punti forti ed i suoi punti deboli. In genere a livello professionale un prototipo come quello presentato viene utilizzato per trovare terze persone disposte a finanziare il progetto, i publisher.
Nonostante sia un prototipo si possono comunque arricchire le meccaniche di gioco, la grafica e l'atmosfera.

Livelli Aggiuntivi

La creazione di livelli aggiuntivi è praticamente d'obbligo perché per un gioco come questo bisogna garantire un minimo di ore di gioco. I livelli dovranno differenziare tra loro in ambientazione e meccaniche di gameplay in modo tale da non annoiare il giocatore, potranno essere aggiunte nuove trappole come delle seghe circolari che fuoriescono dal muro o un grande masso che precipita dal soffitto o un pavimento finto che crolla e così via. Per quanto riguarda le meccaniche di gioco invece non è banale come per la creazione di nuove trappole perché bisogna trovare una meccanica di gioco che non vada ad influenzare troppo il gameplay.

Menù

Sicuramente una delle prime cose da aggiungere è il menù dando la possibilità all'utente di accedere alla sezione opzioni, alla selezione livelli ed ai crediti.
Un'idea carina per la creazione di un menù accattivante è quella di creare un menù 3D invece di un classico 2D.

Per “Catacomb” si è pensato di creare una scena con l’ingresso di una catacomba nello sfondo a destra una tenda con un tavolo e a sinistra una grande roccia.
Il giocatore potrà quindi camminare in questa scena, ovviamente in una zona limitata e potrà vedere i nomi degli sviluppatori incisi nella roccia ed una mappa sopra il tavolo con indicate le varie locazioni delle catacombe.
La mappa può essere un modo originale per fare una selezione del livello dove il giocatore cliccando in una delle catacombe caricherà il livello desiderato.

Meccaniche aggiuntive

Una delle possibili meccaniche da aggiungere può essere quella di inserire un quaderno dove il giocatore potrà annotarsi varie informazioni che troverà in giro per il livello come disegni, simboli o delle note personali. Le note personali possono essere aggiunte tramite la tastiera, invece se si vuole annotare un simbolo o un’immagine presente sul livello si potrà fare tramite l’utilizzo dei trigger e Action item.

Inventario

Sarebbe un’ottima idea quella di inserire un inventario con un’interfaccia 3D per articolare un po’ l’interazione tra oggetti. L’inventario si presenterebbe come una struttura che premendo un pulsante fa fluttuare gli oggetti posseduti dal giocatore dando la possibilità di equipaggiare uno o più oggetti. Equipaggiando per esempio uno scudo, il giocatore sarà in grado di salvarsi la vita nel caso in cui uno spuntone lo colpisca distruggendo ovviamente lo scudo.

Parte Narrativa

Attualmente in “Catacomb” è praticamente assenta la parte narrativa ossia quella parte di un gioco in cui tramite dialoghi, libri, video o delle semplici scritte su schermo, spieghi al giocatore cosa sta succedendo fornendo un indizio su quale è la prossima cosa da fare.
Un videogioco per essere di qualità non per forza deve avere una trama bella e complicata, anzi un gioco può anche non avere una trama e rimanere comunque un gioco di qualità.
Però tra parte narrativa e gameplay deve comunque esistere una sorta di bilanciamento, se la parte narrativa in un gioco è molto complessa ed articolata allora il gameplay può essere semplice e viceversa.
“Catacomb” essendo un horror/puzzle game deve avere una parte narrativa che giustifichi le varie catacombe che stiamo esplorando e soprattutto il perché le stiamo esplorando, quindi in “Catacomb” una delle cose essenziali da fare nel futuro è quella di costruire una trama che colleghi insieme tutte le catacombe per un unico motivo, che può essere per esempio:

> *Per fermare l’imminente invasione aliena il nostro esploratore deve andare in tutte le catacombe a recuperare le reliquie per poi eseguire un antico rito che proteggerà la Terra dall’invasione*.

Conclusione

Il lavoro dietro un videogioco risulta essere lungo e a volte difficile. Infatti per produrre un prodotto di qualità sono necessarie doti che vanno ben oltre la logica della semplice programmazione. Si tratta principalmente di originalità, capacità artistiche e sonore e conoscenza del campo dei videogiochi. Esiste una netta differenza tra gioco di qualità e gioco di successo. Abbiamo già dato una definizione di gioco di qualità precedentemente, per quanto riguarda invece gioco di successo significa avere un gioco di qualità a cui è stato applicato un ottima strategia di marketing. Il Successo di un gioco è dato da una moltiplicazione: qualità del gioco * strategia di marketing.

Quanto presentato sulla tesi è solo parte del lavoro dietro la creazione di un videogioco: principalmente ci siamo concentrati sull'implementazione (lato programmazione) e sulla progettazione (concettuale e pratica) di "Catacomb". Per capire come funziona il mondo del game development abbiamo prima introdotto Unity3D, la sua interfaccia ed i vari tool messi a disposizione, abbiamo anche dato dei concetti base per quanto riguarda lo scripting in Unity. Successivamente si è parlato in parte della progettazione di "Catacomb", come è stata creata l'idea ed è stata mostrata la struttura di interazione tra gli oggetti. Per quanto riguarda la parte implementativa abbiamo mostrato e commentato tutte le classi principali necessarie per il funzionamento del gioco e come è stata impostata parte della scena.Importante ai fini del progetto è saper pianificare il lavoro: se il team di sviluppo è composto da più o meno persone si deve subito pensare a quali feature della nostra idea vogliamo implementare per prime, considerando che tempi di lavoro troppo lunghi possono compromettere la riuscita del lavoro.

In conclusione si è voluto presentare il lavoro dietro allo sviluppo di "Catacomb", un horror/puzzle game realizzato per Oculus Rift. Si spera che le informazioni all'interno della tesi siano sufficienti a chi si approccia per la prima volta allo sviluppo di un applicazione con Unity o di chi semplicemente vuole approfondire le sue conoscenze, ma soprattutto si vuole dare l'idea della quantità di lavoro dietro un videogioco, sia questo semplice o complesso, sperando di suscitare curiosità per questo stimolante ambiente di lavoro.

Bibliografia

[1] Unity Technologies, *Unity Documentation*, http://unity3d.com/learn/documentation

[2] Sue Blackman, *Beginning 3D Game Development with Unity4: All-in-one, multi-platform game development,* Apress, 2013