ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA
SCUOLA DI SCIENZE

CORSO DI LAUREA IN INGEGNERIA E SCIENZE INFORMATICHE

GAME DESING & PROGRAMMING: PROGETTAZIONE E SVILUPPO DI UN VIDEOGIOCO 3D CON UNITY

Tesi di Laurea in Programmazione

Relatore: **Dott. Mirko Ravaioli**

Presentata da: **Fabio Pazzini**

II Sessione di Laurea
Anno Accademico 2015/2016

A mia sorella, mia mamma e mio babbo
Alla nonna Ciana e al nonno Bigio, il quale non ho mai avuto la fortuna di conoscere

<<Talk is cheap. Show me the code.>>
Linus Torvalds

INDICE

Capitoli

INTRODUZIONE 9

MERCATO 10

1. Unity 11
    1.1 Unity Editor 12
    1.2 Unity Engine 15
        1.2.1 Fisica 15
            1.2.1.1 Colliders 15
            1.2.1.2 Rigidbody 15
            1.2.1.3 Raycast 16
        1.2.2 Rendering 16
            1.2.2.1 Shaders 16
            1.2.2.2 Materials 17
            1.2.2.3 Textures 17
        1.2.3 Grafica 15
            1.2.3.1 Illuminazione 17
            1.2.3.2 Skybox 17
        1.2.4 Scripting 15
            1.2.4.1 Mono 18
            1.2.4.2 Wrappers 18
            1.2.4.3 Caratteristiche degli Script 19
            1.2.4.4 Struttura degli Script 19
            1.2.4.5 Funzioni più utilizzate 21
            1.2.4.6 Coroutines 21
2. Progettazione 23
    2.1 Idea 23
    2.2 Team 24
    2.3 Gameplay 25
    2.4 Mappa 26
    2.5 Produzione 28
    2.6 Post-Produzione 28
3. Elementi principali 30
    3.1 Giocatore 30
    3.2 Game Manager 35
        3.2.1 Disposizione randomica degli oggetti 35

3.2.2 Terminazione della partita 37
3.3 Intelligenza Artificiale dei nemici 39
3.4 Comportamento del nemico principale 43
3.5 Interazione giocatore – oggeti 45
3.6 Animazioni 48
3.7 Menù iniziale 49
3.8 Immagini del gameplay 50

Test 52

Sviluppi futuri 53

Conclusioni 54

Sitografia 56

Bibliografia 57

Ringraziamenti 59

INTRODUZIONE

La seguente Tesi di Laurea tratta il processo di analisi, progettazione, sviluppo e test di un videogioco: in sostanza, si andrà a realizzare un intero gioco dal principio applicando le migliori strategie progettuali ed implementative affinché risulti privo di problemi critici per l'utilizzatore del software stesso, quali la perdita di frame per secondo e bug che influenzano, negativamente, l'esperienza utente.
Nel primo capitolo si descriverà innanzitutto il programma utilizzato per lo sviluppo, Unity, analizzandolo nel dettaglio prendendo in esame l'editor grafico che esso mette a disposizione denominato, appunto, Unity Editor, proseguendo con lo Unity Engine, ovvero il "motore" che si cela dietro la UI. Si andranno a studiare le funzionalità che questo possiede, tra cui il rendering, la gestione della fisica, la manipolazione dell'illuminazione ed infine il meccanismo di scripting.
Il secondo capitolo affronterà invece il tema della progettazione a partire dall'ideazione del gameplay, passando per la creazione delle mappe e concludendo con la vera e propria fase di produzione, in cui il gioco viene effettivamente realizzato .
Nel terzo capitolo si osserveranno nello specifico gli elementi principali del videogioco, le tecniche di design applicate, la progettazione effettuata per particolari algoritmi tra cui quello che determina l'Intelligenza Artificiale dei nemici ed il comportamento del nemico principale; verrà inoltre analizzata l'interazione con gli oggetti e anche alcuni dettagli secondari come le animazioni.
Verranno infine descritti eventuali sviluppi futuri del progetto.
Il gioco sviluppato per questa Tesi prende il nome di The Mad House: si tratta di un videogioco Horror in prima persona in cui il giocatore si trova a dover reperire degli oggetti particolari esplorando un'ampia mappa appositamente progettata per consentire, almeno, 15 o 20 minuti di gioco. Questa prevede un bosco all'interno del quale sono collocate due grandi case: in esse si dovranno ricercare oggetti utili per la sopravvivenza ma anche gli oggetti coi quali si vince la partita. Nell'apposito capitolo verranno meglio descritte le modalità di gioco e di gameplay.
The Mad House è giocabile sia su PC equipaggiati con Windows che su computer dotati di Mac OS; sono state escluse le piattaforme mobili e quindi sistemi operativi come iOS ed Android poiché dotate di un hardware meno potente, tuttavia con adeguate modifiche ed ottimizzazioni può essere giocato senza problemi anche su questo tipo di sistemi.

MERCATO

Il mondo dei videogiochi è da sempre stato, fin dalla sua nascita, in costante crescita; il primo caso di gioco elettronico interattivo risale a due anni dopo la fine della Seconda Guerra Mondiale, nel 1947. Questo primitivo videogioco consisteva in un simulatore di lancio di missili su un tubo a raggi catodici: non ci volle molto prima che altri esperti del campo si cimentassero in esperimenti di questo genere, infatti con le nuove linee di computer, più potenti, numerosi team si impegnarono nello sviluppo di quelli che si sarebbero poi rivelati gli antenati dei giochi di oggi. Nel 1981 venne diffuso il primo gioco 3D della storia per il Sinclair ZX81; due anni dopo venne alla luce il primo gioco online per computer IBM e dopo altri due anni fu creato il celebre NES da parte di Nintendo. A quel punto la storia era stata scritta, e il mercato dei videogiochi iniziò a crescere esponenzialmente. Ma poiché l'hardware è inutile senza il software, migliaia di sviluppatori iniziarono a creare videogiochi di qualsiasi genere, dai classici Arcade a giochi online più complessi, da giochi per la famiglia a giochi che sfruttano la realtà virtuale. Ai giorni nostri l'industria videloudica fattura miliardi di dollari all'anno (solo nel 2015, considerando anche il mercato delle App per dispositivi mobili, si parla di 61 miliardi di dollari) di conseguenza appare chiaro come questo sia un campo dove non solo si ha la possibilità di avere successo con le idee giuste, ma anche dove la creatività e la fantasia fanno da padroni; e con computer e console sempre più performanti e potenti, che consentono l'esecuzione di giochi che prevedono requisiti hardware molto elevati in quanto dotati di elementi grafici (e non solo) poco distanti dal realismo, sono stati creati capolavori in ogni categoria: un esempio è il genere dei GDR (Gioco Di Ruolo), dove si possono trovare delle vere e proprie opere quali "Skyrim", "The Witcher 3", tutta la saga di "Final Fantasy", e via dicendo. Un'altra tipologia di videogiochi che riscuote grande successo è il genere Horror, a cui appartengono lavori come "Amensia: The Dark Descent", "Silent Hill", "Alien Isolation", "Outlast", "Slenderman". Ed è da quest'ultimo che il gioco sviluppato per questa Tesi trae maggiormente ispirazione, riprendendo e realizzando anche elementi simili ad alcune caratteristiche di Amnesia come l'IA dei nemici.

CAPITOLO 1 - UNITY

Per la realizzazione di questa Tesi è stato utilizzato il game engine Unity. Esso è composto da due sottosistemi, lo Unity Editor e lo Unity Engine: entrambi verranno dettagliatamente analizzati nei prossimi capitoli. Di seguito viene invece fatta una breve presentazione della storia e delle proprietà caratteristiche di questo motore, che permette lo sviluppo di giochi di piccola ma anche media / grande dimensione.

La storia
Unity venne presentato per la prima volta al WWDC, su invito di Apple, ben undici anni fa, nel Giugno del lontano 2005. L'obiettivo degli sviluppatori era quello di realizzare un software che permettesse a chiunque di sviluppare programmi in 2D e 3D efficienti e cross-platform; esso avrebbe quindi messo a disposizione una suite di strumenti potenti ed intuitivi per poter creare giochi (e, più in generale, programmi) da distribuire su (quasi) qualsiasi sistema in circolazione.

Unity
Fondamentalmente Unity è un motore di gioco, disponibile sia in versione Free che in versione Pro, il cui scopo è quello di consentire lo sviluppo di programmi e giochi da poter distribuire ovunque in maniera efficiente, rapida, e senza rinunciare ad elevate performance, dettaglio che influenza maggiormente la UX (*user-experience*).

Cross-Platform
È con questa filosofia che Unity venne sviluppato, ed è questa la peculiarità del software. Infatti al momento Unity supporta l'esportazione per numerosissime piattaforme, tra cui Windows e Unix (Mac OS, Linux), PlayStation 4 e Vita, Xbox 360 e One, Wii e Wii U; inoltre, sono coperti anche i principali OS mobili quali iOS, Android e Windows Phone. Infine, si possono anche sviluppare giochi dedicati alla VR ed AR (Virtual Reality ed Augmented Reality).

Gli utilizzi di Unity oggi
Una domanda che sorge spontanea, riguarda il modo in cui Unity venga utilizzato ai giorni nostri. Essendo un game engine il suo utilizzo primario riguarda indubbiamente lo sviluppo di videogiochi: questo perché consente agli sviluppatori di evitare di costruirsi un proprio motore di gioco (e, di conseguenza, imporre una licenza su di esso) comprendente gestione efficiente del rendering, dell'audio, e della fisica. Ad ogni modo, gli utilizzi che si possono fare di Unity spaziano in molti più ambiti del semplice gaming, ad esempio viene sfruttato per realizzare simulazioni, dimostrazioni interattive, e virtualizzazioni di oggetti ed ambienti di varia natura.

1.1 - UNITY EDITOR

L'interfaccia grafica di Unity è molto intuitiva e costruita in modo da rendere veloci le operazioni che necessitano dell'interazione da parte dell'utente. Verranno ora analizzate le componenti principali della UI del programma, che prende il nome di *Unity Editor*.

Inspector

In Unity vi è il concetto di GameObject, che identifica ogni oggetto presente all'interno della scena: questi GameObjects sono costituiti dai cosiddetti *Components,* i quali ne definiscono le proprietà fisiche e comportamentali. Tutto ciò che viene attribuito ad un oggetto è un Componente, compresi gli script: ogni GameObject possiede, di default, il Componente Transform, che rappresenta la posizione, la dimensione e la rotazione dell'oggetto all'interno del mondo virtuale.
L'Inspector è il tool attraverso il quale si possono aggiungere nuovi Componenti, modificare variabili o parametri pubblici propri di un Componente o anche rimuovere quelli che non si vogliono più utilizzare.
Inoltre, con questo strumento si può assegnare anche un *tag* ad un oggetto, in modo tale da "categorizzare" gli elementi del gioco ed eventualmente fare uso del tag tramite il codice per gestire l'oggetto al quale esso è assegnato.

Hierarchy

Quando si istanzia un oggetto nella scena, ne viene contemporaneamente creato un riferimento nella Hierarchy del progetto. In essa si trovano elencati tutti i GameObjects aggiunti, sia quelli impostati come "inattivi" (e quindi non visibili) sia quelli attualmente visibili e con cui si può interagire, all'interno del mondo di gioco. Si possono anche creare gerarchie di parentela tra gli oggetti semplicemente trascinando uno di essi all'interno di quello che si vuole utilizzare come genitore; da quel momento, tutte le operazioni di ridimensionamento, traslazione o rotazione sul GameObject genitore si rifletteranno sul GameObject figlio.

Project

Tutti gli elementi che vengono aggiunti in Unity, compresi script, modelli, textures, musiche, ma anche le scene, vengono salvati all'interno della root directory del progetto. Al suo interno si possono, ovviamente, creare delle subdirectory per una gestione più organizzata dell'intero software che si sta realizzando.

Game

Figura 1.1 - Game View

La View denominata Game viene utilizzata quando il programma viene messo in esecuzione. È quindi possibile effettuare un test pratico di ciò che si sta realizzando; Unity dà anche la possibilità di modificare i parametri dei Componenti degli oggetti, tramite l'inspector (precedentemente analizzato), ed osservare in tempo reale le conseguenze di tali modifiche in questa schermata.

Scene

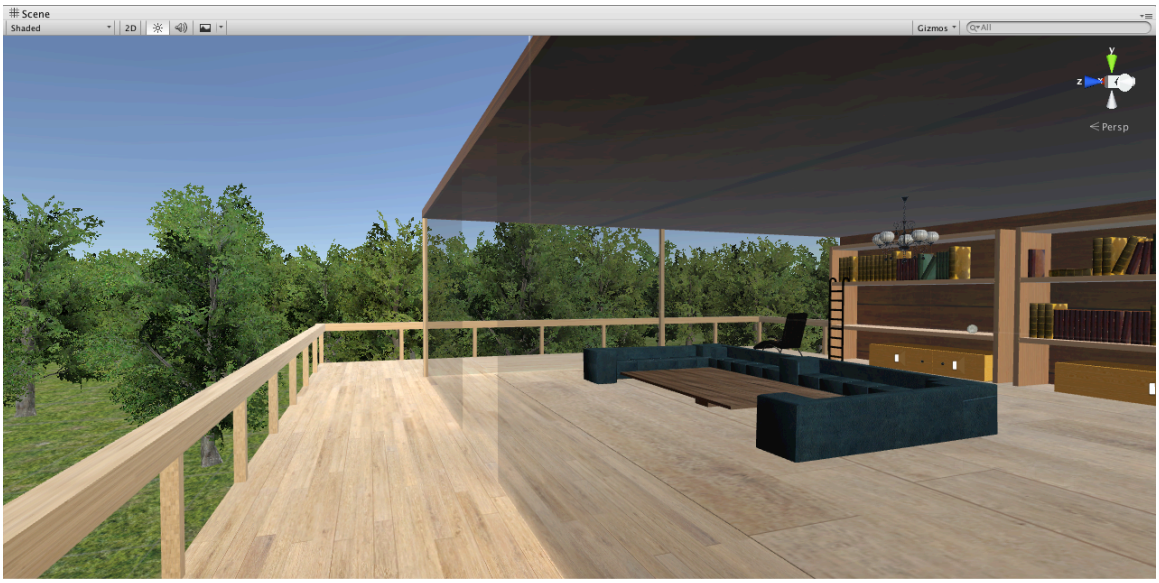

Figura 1.2 - Scene View

La Scene View rappresenta la scena corrente, ovvero il mondo che stiamo costruendo in Unity. Tutti i GameObjects sono manipolabili da questa finestra, e grazie agli strumenti di traslazione, rotazione e ridimensionamento si può modificare qualsiasi corpo attivo al suo interno. La Scene è completamente navigabile col semplice uso del mouse: con esso, infatti, si può ruotare la telecamera, zoomarla e spostarla con l'utilizzo di rapidi click e con l'ausilio della rotella centrale. Inoltre sono presenti dei comandi rapidi che si rivelano di importante utilità mentre si lavora col programma, un esempio è il pulsante F che, premuto, fa sì che la scena si focalizzi sull'oggetto selezionato.

Animation

Questo utile tool consente di creare animazioni direttamente all'interno della View Scene. Occorre per per prima cosa selezionare l'oggetto che si vuole animare, successivamente si seleziona nella Timeline il frame, nonché l'istante di tempo, in cui si vuole che l'animazione venga completata: ora basta effettuare l'animazione desiderata (traslazione, rotazione…), ed una volta terminato premere nuovamente il pulsante Record. A questo punto l'animazione è completata ed è pronta per essere assegnata al GameObject target.

1.2 - UNITY ENGINE

Nella sezione precedente è stato analizzato l'aspetto grafico ed interattivo di Unity, il cosiddetto Unity Editor; ora verrà invece preso in esame il motore di sviluppo che vi sta dietro, chiamato *Unity Engine*, del quale saranno analizzate la gestione della Fisica, del Rendering, della Grafica e dell'intero meccanismo di Scripting.

1.2.1 - Fisica

Quando si parla di Fisica in Unity si intendono molteplici aspetti legati alla fisica del mondo virtuale, che comprende componenti quali Colliders e Rigidbody, ma anche funzionalità come il Raycasting. L'intera gestione di queste risorse viene affidata all'NVIDIA Physx, un motore fisico estremamente versatile e potente realizzato da NVIDIA e che viene utilizzato in numerosi game engines, tra cui anche l'Unreal Engine 4.

1.2.1.1 - Colliders

I Colliders consentono la collisione e l'interazione tra gli oggetti. Da un punto di vista grafico possono essere visti come i contorni di un GameObject, ovvero definiscono la superficie tramite la quale l'oggetto effettua una collisione con altri corpi; dal punto di vista implementativo, invece, il Collider è la classe base estesa da numerose classi quali *BoxCollider, SphereCollider, MeshCollider* e *CapsuleCollider*. Un *BoxCollider* prevede un collider di forma scatolare, uno *SphereCollider* ne prevede uno di forma sferica mentre un *CapsuleCollider* ne crea uno a forma di capsula; tutte queste classi rappresentano un collider di tipo primitivo, poiché possiedono una semplice forma geometrica. Il discorso cambia quando si prende in esame un *MeshCollider*, infatti esso consente la creazione di un collider in base alla mesh applicata ad un oggetto e di conseguenza si ha la possibilità di applicare un collider geometricamente più complesso e preciso, richiedendo tuttavia una serie di calcoli molto impegnativi per il calcolatore.

1.2.1.2 - Rigidbody

Il Rigidbody è uno dei principali componenti in Unity, dal momento che permette ai GameObjects di agire in risposta all'azione della fisica. Affinché un corpo sia influenzato dalla gravità e sia soggetto a forze (che possono generarsi all'interno del mondo, come ad esempio a seguito di una caduta, oppure che possono essere dettate via scripting) deve essere necessariamente munito di questo componente. Selezionando un oggetto equipaggiato con un Rigidbody, all'interno dell'Inspector è possibile stabilire le forze che ne devono influenzare il comportamento.

1.2.1.3 - Raycasts

Il Raycasting è una funzionalità estremamente potente ed utile messa a disposizione dall'engine. Fondamentalmente consente di generare un raggio, a partire da un certo punto, verso una determinata direzione e con una certa lunghezza permettendo di sapere se ha colpito o meno qualcosa.
È possibile affermare che la sua incredibile utilità è direttamente proporzionale ai costi computazionali che richiede; è quindi necessario ridurre al minimo il numero di Raycasts all'interno della scena, ma soprattutto occorre minimizzare gli scontri tra un raggio e un mesh collider, poiché richiederebbero ulteriori ed elevati costi di calcolo.

1.2.2 - Rendering

Il rendering in Unity si basa sui cosiddetti *Rendering Paths,* che determinano la qualità grafica del gioco e sono strettamente dipendenti dalle caratteristiche hardware del calcolatore sul quale l'applicativo viene eseguito. Ognuno di essi offre diverse performance ed in particolare agiscono sulla resa grafica delle luci e delle ombre, le quali rappresentano uno sforzo significativo dal punto di vista computazionale affinché vengano appropriatamente calcolate e rappresentate. I principali *Rendering Paths* sono:

1. *Deferred Shading;*
2. *Forward Rendering;*
3. *Legacy Deferred;*
4. *Legacy Vertex Lit.*

Qualora la soluzione scelta non fosse adeguatamente supportata dalla GPU sottostante allora Unity provvederebbe a selezionare un'altra tipologia di *Path* che effettui un rendering meno curato nei dettagli, in modo da offrire all'utente, in ogni caso, un'esperienza di gioco fluida.
Il rendering va ad operare su tre elementi: *Shaders, Materials* e *Textures.*

1.2.2.1 - Shaders

Gli *Shaders* sono dei piccoli programmi che contengono istruzioni e calcoli con lo scopo di riprodurre la struttura materiale dell'oggetto al quale vengono applicati. È quindi possibile scrivere degli Shaders per qualsiasi tipo di materiale.

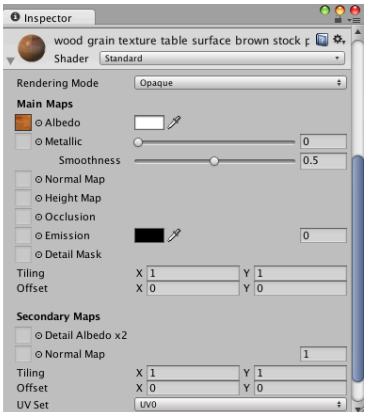

Figura 1.3 - Shader nell'Inspector

1.2.2.2 - Materials

I Materials definiscono un set di informazioni per la rappresentazione tridimensionale di un corpo, e la maggior parte di essi accetta una o più textures in ingresso. Inoltre, possono anche definire alcune proprietà fisiche del corpo. In Unity vengono gestiti tramite il pannello dei Materials, attraverso il quale si può osservare il risultato finale dell'applicazione delle immagini sul Material.

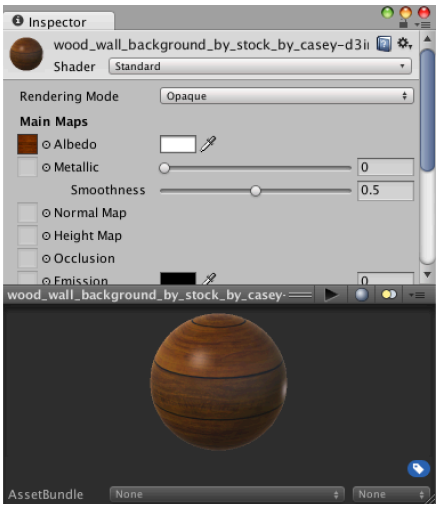

Figura 1.4 - Material nell'Inspector

1.2.2.3 - Textures

Le Textures sono semplicemente delle immagini che possono essere applicate sui Materials o direttamente sul GameObject target.

1.2.3 - Grafica

Per quanto concerne la grafica in Unity, si possono distinguere due macro-elementi principali: l'illuminazione e gli Skyboxes.

1.2.3.1 Illuminazione

Le luci sono un elemento di fondamentale importanza in qualsiasi videogioco si stia realizzando. Esse danno la possibilità di creare incredibili atmosfere e di dare maggiore profondità al gioco, permettendo una maggiore immersione da parte dell'utente, in particolare se ne vengono create delle combinazioni. In Unity esse sono estremamente personalizzabili, infatti se ne possono modificare (tramite l'Inspector) l'intensità, l'ampiezza di illuminazione, ed anche il colore emesso; inoltre vengono considerate e gestite come normali GameObejcts.

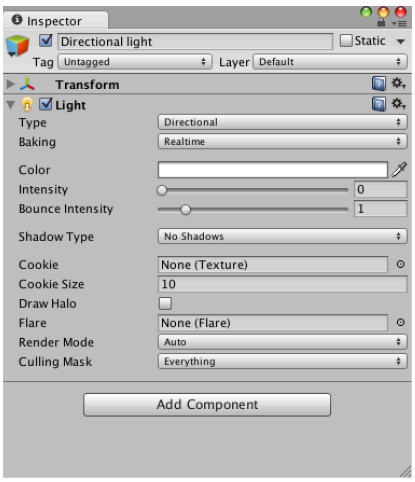

Figura 1.5 - Impostazione luce nell'Inspector

1.2.3.2 Skybox

Gli Skyboxes vengono utilizzati per simulare il cielo e l'orizzonte, e vengono renderizzati attorno all'intera scena.

1.2.4 - Scripting

Lo scripting in Unity è un argomento alquanto complesso, e in questo paragrafo verranno analizzate sia le modalità con cui si può programmare sia il framework su cui si basa.

1.2.4.1 - Mono

Mono è un'implementazione open source del framework .NET di Microsoft. L'idea con cui Mono è stato sviluppato consisteva nel fornire un'implementazione di .NET per sistemi Unix-like, dal momento che esiste solo per Windows, e che tale implementazione fosse open-source e libera.
Più in generale, ciò che Mono vuole realizzare è la possibilità di avere la massima portabilità del software consentendo un suo corretto funzionamento su tutti i sistemi operativi; questo, ovviamente, comporta una re-implementazione delle librerie specifiche di ogni sistema così che i programmi girino ovunque. Un esempio è la libreria System.Windows.Forms di Microsoft: questa libreria consente allo sviluppatore di creare e gestire i componenti grafici di un applicativo (finestre, pulsanti, ecc…). È evidente come su un sistema Mac OS o Linux questa libreria non sia presente, ed è qui che entra in gioco Mono: esso fornisce una re-implementazione delle WinForms in modo tale da poter utilizzare un programma che si affida ad esse anche su altri sistemi operativi.
Ad ogni modo, è importante sottolineare come Unity non sia stato realizzato al di sopra di Mono. Se così fosse, infatti, Unity sarebbe di natura cross-platform, mentre invece ne esistono molteplici versioni, una specifica per ogni sistema operativo; ciò che Unity fa è sfruttare Mono come framework su cui basare il proprio sistema di scripting grazie alle elevate performance da esso offerte.

1.2.4.2 - Wrappers

I linguaggi con cui si può programmare in Unity sono principalmente tre: C#, JavaScript e Boo. Come è ben noto questi sono linguaggi che offrono delle ottime performance, tuttavia, quando si tratta di realizzare dei software particolarmente complessi, soprattutto dal punto di vista grafico, essi si rivelano insufficienti. Di conseguenza sorge spontaneo il quesito su come Unity riesca a consentire l'esecuzione di programmi con delle performance elevatissime anche quando si fa uso di oggetti grafici molto complessi. Il "segreto" consiste nella presenza dei cosiddetti Wrappers: questi permettono di racchiudere classi e funzioni, alcune pubbliche ed altre private, all'interno di librerie normalmente accessibili ed utilizzabili. Essi vengono scritti in C#, tuttavia, dietro le quinte, fanno uso dei linguaggi di programmazione C e C++ in quanto estremamente potenti e performanti. Inoltre il cuore stesso di Unity è scritto in questi due linguaggi, e conseguentemente anche tutta la gestione della grafica, del suono e della fisica.

1.2.4.3 - Caratteristiche degli Script

Gli scripts in Unity vengono solitamente considerati come *Components* in quanto estendono dalla classe MonoBehaviour, la quale a sua volta estende da Behaviour che estende, appunto, da *Component*.

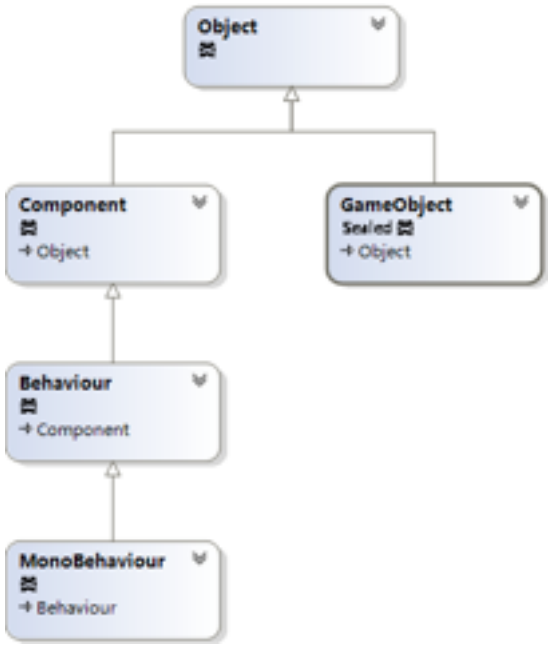

Figura 1.6 - Struttura gerarchica degli oggetti in Unity

Questa proprietà implica che un certo script possa essere assegnato ad un GameObject, e quest'azione viene semplicemente effettuata trascinando lo script sull'oggetto target. Si possono differenziare due tipologie di script:
- Comportamenti: questi script vanno ad avere un impatto sul comportamento degli oggetti, ne definiscono le azioni e li controllano; questa tipologia estende da MonoBehaviour.
- Dati: sono script che vengono utilizzati solamente come contenitori di dati e che non hanno alcuna influenza sul comportamento dei GameObjects; poiché queste classi non estendono da MonoBehaviour, e di conseguenza nemmeno da Component, non vengono assegnate agli oggetti.

1.2.4.4- Struttura degli Script

In Unity gli script che estendono da MonoBehaviour seguono una certa struttura, definita da una serie di funzioni ad ognuna delle quali è associato un preciso istante di tempo in cui deve essere eseguita da parte dello Unity Engine. In questo

paragrafo verranno analizzate le principali:

1. *Awake:* è la prima funzione ad essere richiamata, in particolare nel momento in cui l'oggetto al quale lo script è associato viene caricato. Questo metodo, in C# e Boo, sostituisce l'utilizzo del costruttore in quanto, proprio come quest'ultimo, viene richiamato una sola volta e proprio nell'istante di tempo in cui l'oggetto viene creato.
2. *Start:* è la funzione che viene invocata subito dopo il metodo *Awake*, in particolare nel preciso frame in cui lo script viene attivato; come quest'ultimo, viene invocato una sola volta nel lifecycle dell'oggetto.
3. *Update*: metodo invocato ad ogni frame, se lo script è attivato all'interno dell'Inspector.
4. *FixedUpdate:* metodo invocato un determinato numero di volte per frame.
5. *LateUpdate*: funzione invocata una volta completate tutte le chiamate alle funzioni *Update*.

I metodi appena descritti non sono sempre utilizzati nel medesimo script: è a discrezione del programmatore decidere quali utilizzare, dipendentemente da ciò che il suo script deve fare e in che modo esso va ad influire nel tempo sul comportamento dell'oggetto al quale è associato.
Una delle problematiche implementative più rilevanti riguarda l'utilizzo della funzione *Update* rispetto a *FixedUpdate.* Fondamentalmente, Update viene richiamata ad ogni frame: questo comporta che la sua frequenza dipende da quanto velocemente il computer sul quale il programma gira renderizza le immagini. Più il computer è lento, meno frequentemente Update viene invocata rispetto ad un computer veloce; se si volesse implementare un comportamento time-based, ovvero dipendente dallo scorrere del tempo e non dagli FPS (frame-per-second), occorre fare uso della variabile *Time.deltaTime* che rappresenta l'intervallo di tempo trascorso tra l'ultimo frame e quello corrente. Il metodo FixedUpdate invece si comporta in maniera differente: esso viene richiamato ad intervalli prefissati, senza dipendere dalla velocità del computer come il metodo Update, in particolare *1 / Time.fixedDeltaTime* volte al secondo, dove Time.fixedDeltaTime corrisponde al tempo trascorso tra due chiamate del metodo FixedUpdate (e mantiene sempre lo stesso valore).
Per quanto riguarda le situazioni concrete in cui occorre optare per una delle due soluzioni, la chiave della scelta riguarda il comportamento dello script nel tempo.
I task che eseguono un continuo aggiornamento dello stato fisico di un oggetto devono essere implementati attraverso l'uso del metodo FixedUpdate, poiché tutti i movimenti e la fisica devono dipendere dal tempo e non dal framerate: in questo caso i calcoli vengono effettuati dopo l'esecuzione di tale metodo. Inoltre, quando si calcolano i movimenti all'interno di FixedUpdate, non si deve moltiplicare per Time.deltaTime in quanto i calcoli sulla fisica sono già in *unità-per-secondo*.
I task che invece implementano Update vengono utilizzati per, ad esempio, rilevare input da tastiera o rilevare oggetti nello spazio con l'uso del metodo *Physics.Raycast*: l'esecuzione di queste azioni dipenderà dunque dal framerate.

1.2.4.5 - Funzioni più utilizzate

Verranno ora analizzate le funzioni di libreria, offerte da Unity, maggiormente utilizzate nello sviluppo di questo videogioco.

1. *OnTriggerEnter (Collider object):* metodo invocato quando un Collider entra nel trigger. Come per *OnTriggerExit* ed *OnTriggerStay*, affinché questi metodi vengano correttamente richiamati occorre che sia stata spuntata la voce "Is Trigger" nel componente relativo all'oggetto che si vuole rendere attraversabile.
2. *OnTriggerExit (Collider object):* metodo invocato quando un Collider esce dal trigger.
3. *OnTriggerStay (Collider object):* metodo invocato durante tutto il periodo di tempo in cui un Collider risiede all'interno del trigger.
4. *OnCollisionEnter (Collision object):* metodo invocato quando un corpo dotato di un componente di tipo Rigidbody entra in collisione con un altro oggetto. L'oggetto appartenente alla classe Collision passato come argomento contiene informazioni sui punti di contatto, sulla velocità di impatto ed altre riguardanti la collisione; i metodi *OnCollisionEnter, OnCollisionExit* ed *OnCollisionStay* funzionano correttamente solo se almeno uno dei due corpi possiede un componente Rigidbody impostato come *non-kinematic.*
5. *OnCollisionExit (Collision object):* metodo invocato quando un corpo esce dalla collisione con un altro oggetto.
6. *OnCollisionStay (Collision object):* metodo invocato per tutto il periodo di tempo in cui un corpo resta in collisione con un altro oggetto.
7. *OnGUI ():* funzione utilizzata per renderizzare e gestire le GUI.
8. *OnMouseDown ():* funzione richiamata quando l'utente clicca con il mouse sul Collider o su un GUIElement, che contiene tutte le funzionalità per le GUI utilizzabili in Unity.

1.2.4.6 - Coroutines

Uno strumento molto utile nella programmazione in Unity sono le Coroutines.
Prima di descriverne il funzionamento, occorre sottolineare come in Unity non vi sia concorrenza in termini di thread a meno che non si crei manualmente un thread a parte: infatti vi è un unico main loop e tutte le funzioni vengono invocate, ordinatamente, dal thread principale (a dimostrazione di ciò occorre semplicemente creare un loop infinito in una funzione e si può notare come Unity vada in crash); tutti gli altri threads quindi vanno solo ad aggiungere eventi alla coda degli eventi mantenuta dal thread principale il quale li risolve in maniera ordinata (questo principio viene anche chiamato produttore – consumatore).
Per quanto concerne le Coroutines, queste vengono avviate tramite il metodo

StartCoroutine(Coroutine()). Di base esse lavorano come normali funzioni fino all'istruzione *yield return X*, dove X identifica vari tipi di istruzioni, in quanto è da questo punto che iniziano a differire da un classico metodo: infatti, Unity mette in pausa la Coroutine salvando il valore delle variabili locali ed iniziando ad eseguire le operazioni richiamate, e quando è il momento di riprendere la sua esecuzione si ricomincia dalla riga di codice successiva a quella corrispondente all'istruzione *yield return X.*

CAPITOLO 2 - PROGETTAZIONE

In questo capitolo verranno analizzate le fasi di progettazione e design dell'intero gioco, e più in generale di qualsiasi videogioco si voglia realizzare, partendo dall'ideazione del gameplay e delle ambientazioni per arrivare poi alla vera e propria implementazione del codice. Occorre anche scegliere la piattaforma target per il proprio lavoro, in questo caso il gioco è stato ideato per sistemi Windows e Mac OS; tuttavia, importando le dovute librerie e implementando le specifiche funzioni per poter interagire dai vari controller (ad esempio, per Nintendo Wii, la libreria UniWii per permettere di giocare tramite il WiiMote ed il Nunchuck) è possibile giocare anche su altre piattaforme poiché, come già sottolineato, Unity consente l'esportazione per numerosissime console e sistemi.

2.1 - Idea

Quando si progetta un videogioco in primo luogo è necessario elaborare un'idea o un concept che ne definisca, in linea generale, il genere ed il tema.
In questa fase non occorre essere particolarmente specifici, occorre solamente scegliere la tipologia di prodotto che si andrà a sviluppare: per esempio, si può voler creare un gioco di corse, un gioco di avventura, o un gioco open-world (ovvero in cui il mondo è liberamente esplorabile); tuttavia si potrebbe voler anche realizzare un remake o un sequel di un gioco già sul mercato. Ed è quest'ultimo il caso applicato, in parte, per la creazione di The Mad House. Infatti, esso è stato particolarmente influenzato dal gameplay di un celebre videogioco Horror denominato *Slender: The Eight Pages*. Si tratta di un videogame in prima persona in cui il giocatore si trova in una foresta e deve raccogliere 8 pagine, sparse all'interno di essa, evitando di essere presi dallo Slenderman, una sorta di mostro dall'aspetto umanoide ma alquanto inquietante che insegue il personaggio e le cui apparizioni diventano sempre più frequenti mano a mano che ci si avvicina all'obiettivo della partita. È su questa idea che The Mad House è stato realizzato, mantenendo l'obiettivo di trovare una serie di oggetti fuggendo da un mostro (il modello dello Slenderman è stato riutilizzato per questo progetto come nemico principale) ma espandendo enormemente sia le meccaniche di gioco che l'ambientazione, i nemici e le funzionalità. L'idea elaborata ad inizio progetto prevedeva quindi un videogioco appartenente al genere di Avventura, a tema Horror.

2.2 - Team

Una volta collocato il gioco in un certo genere e definitone il relativo tema, è necessario costituire un team di persone specializzate nei singoli ambiti. Le figure che si devono cercare sono:

1. *Concept Artist* (o *Concept Designer)*: è colui che realizza sketch ed immagini degli elementi di gioco tra cui il giocatore, le armi, le mappe, e così via. È un professionista nel disegno 2D.
2. *Level Designer*: questa figura ha l'obiettivo di progettare e costruire le mappe del gioco e di definire il gameplay. Egli ha la responsabilità di creare ambienti profondi e consistenti, in linea con il tema del prodotto. È anche suo compito creare la giusta atmosfera combinando al meglio le luci della scena.
3. *Modeler*: è la figura professionale che si occupa di "tradurre" i concepts elaborati dal Concept Artist dal 2D al 3D; in sostanza, crea i modelli tridimensionali che verranno poi collocati all'interno del mondo di gioco.
4. *Texture Artist:* è colui che si occupa di realizzare le immagini e le texture da applicare ai modelli 3D. Questa figura deve avere una profonda conoscenza nel campo della grafica 2D.
5. *Animator:* il risultato del lavoro del Modeler è uno statico modello tridimensionale, associabile ad una sorta di "scultura". L'Animator, come lavoro, deve innanzitutto eseguire il rigging di tale modello: questa operazione consiste nel realizzare un rig del corpo, ovvero, tramite una serie di nodi e collegamenti che simulano le ossa, si costruisce uno scheletro da applicare al modello così da renderlo movibile. Fatto ciò, deve creare le animazioni richieste gestendo manualmente questi legamenti.
6. *Programmatore*: è la persona che si occupa di tradurre gli algoritmi in linee di codice. Egli deve avere conoscenza sia del linguaggio di programmazione che si vuole utilizzare per creare il videogioco, sia dell'Engine in cui si sviluppa.
7. *Project Manager*: il suo lavoro è prettamente gestionale e richiede ottime competenze organizzative e metodologiche. Il suo obiettivo primario prevede il controllo del rispetto dei tempi di sviluppo, dei costi e della qualità del software durante tutto il ciclo di vita del software.
8. *Web Designer*: anche se non necessaria come figura per un progetto di piccole dimensioni, si rivela il contrario per grandi ed ambiziosi progetti. Il suo compito prevede prima di tutto la realizzazione di un sito web dedicato al gioco, in seguito il suo mantenimento ed aggiornamento.

Nel caso di The Mad House, essendo questo un gioco sviluppato individualmente, è possibile affermare che la figura del Concept Artist, del Level Designer e del Programmatore siano rappresentate da una sola persona, tuttavia i modelli, le immagini ed i suoni sono stati presi da internet.

2.3 - Gameplay

Come precedentemente affermato, il gameplay è l'elemento che determina le meccaniche di gioco e quindi può essere considerato l'aspetto più rilevante in un prodotto di questo tipo. Il Level Designer normalmente si occupa di elaborarne uno ispirandosi ad un gioco già esistente oppure ideandone uno del tutto originale; in questo caso si è deciso di creare un gioco molto simile ad un videogame, pubblicato alcuni anni fa, che ha riscosso particolare successo.
Definito il gameplay bisogna anche scrivere il walkthrough, ovvero una sorta di guida che spiega come completare il gioco. Quello scritto per The Mad House è il seguente:
Il giocatore viene catapultato in una foresta buia, e l'unica cosa che può vedere sono piccole luci in lontananza ed un falò davanti a sé: vicino ad esso si trova una cassa all'interno della quale si potrà raccogliere una torcia e la batteria per alimentarla. Una freccia suggerisce il successivo punto in cui il giocatore si dovrà recare per trovare un oggetto utile ai fini della partita, e qui si raccoglierà un coltello. Questo potrà essere usato per uccidere i nemici, tuttavia dopo un determinato tipo di utilizzo (può infatti essere lanciato o usato per accoltellare) si romperà e dovrà essere ricostruito, presso una fornace collocata nella foresta stessa, spendendo un certo numero di monete raccoglibili nella mappa. A questo punto il giocatore dovrà recarsi all'interno della villa principale seguendo l'indicazione della freccia e una volta entrato si troverà bloccato al suo interno: qui dovrà in primo luogo riuscire a sopravvivere sfuggendo dai nemici che lo inseguono qualora lo dovessero individuare, inoltre dovrà esplorarla completamente così da trovare il maggior numero di sfere infuocate possibile e infine due chiavi, una per sbloccare l'ingresso principale e l'altra per poter aprire la porta della piccola casetta poco distante dalla casa principale. Una volta trovate tutte e sei le sfere infuocate egli dovrà andare al vulcano e saltarvi all'interno, a quel punto la partita termina con successo. Durante la partita il giocatore dovrà anche evitare lo sguardo dello Slenderman, e per scappare dai nemici che lo inseguono ha due possibilità: la prima prevede la loro uccisione con l'ausilio del coltello precedentemente raccolto, la seconda, invece, prevede di nascondersi in uno degli armadi collocati nella casa ed attendere che il nemico si allontani.

2.4 - Mappa

Per la realizzazione della mappa di gioco è stato disegnato un primo prototipo molto basilare, che prevedeva una piccola casa che è risultata poi essere il punto di partenza per la struttura definitiva. Dopo un'attenta fase di analisi dell'ambiente, che non deve essere troppo ostico da esplorare per il giocatore ma contemporaneamente nemmeno troppo lineare e semplice, la villa risultante prevede due ampi piani provvisti di numerose stanze, ognuna delle quali a proprio modo importante nella partita. Di seguito vengono mostrate due semplici planimetrie dei livelli, escludendo l'arredamento.

Piano terra:

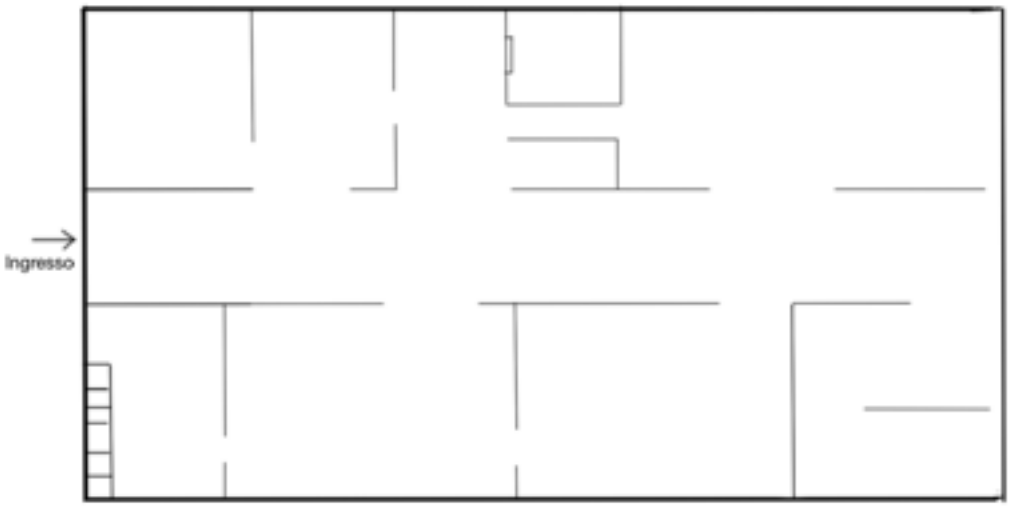

Figura 2.1 - Planimetria del piano terra della villa

Primo piano:

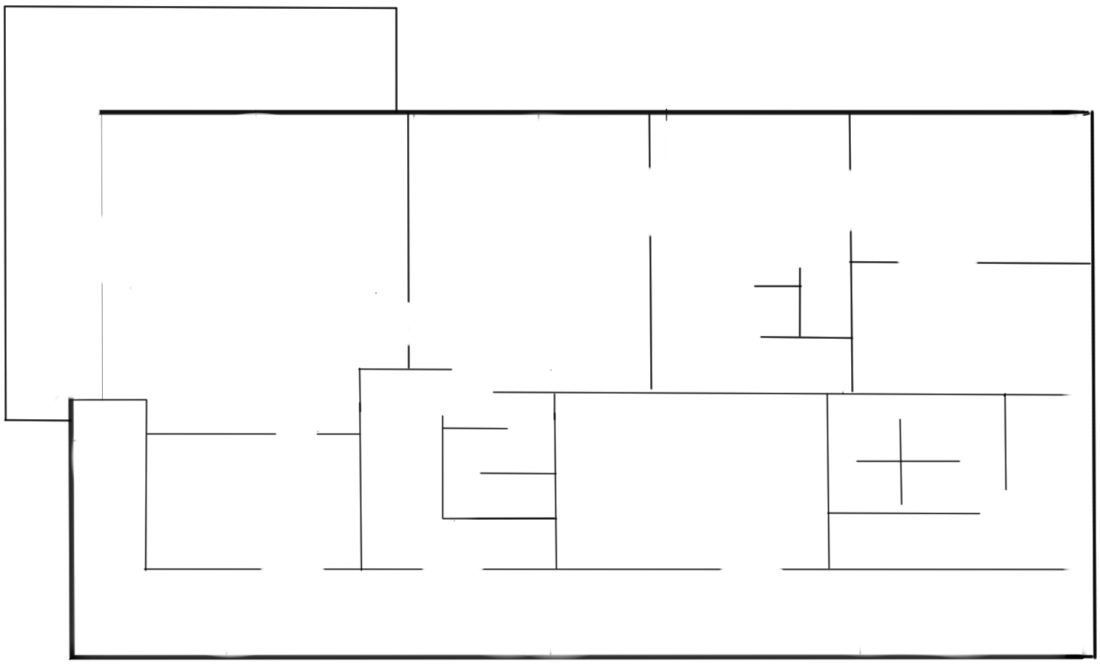

Figura 2.2 - Planimetria primo piano della villa

Di seguito invece la planimetria della seconda casa situata nella foresta, di dimensioni molto ridotte rispetto alla struttura principale:

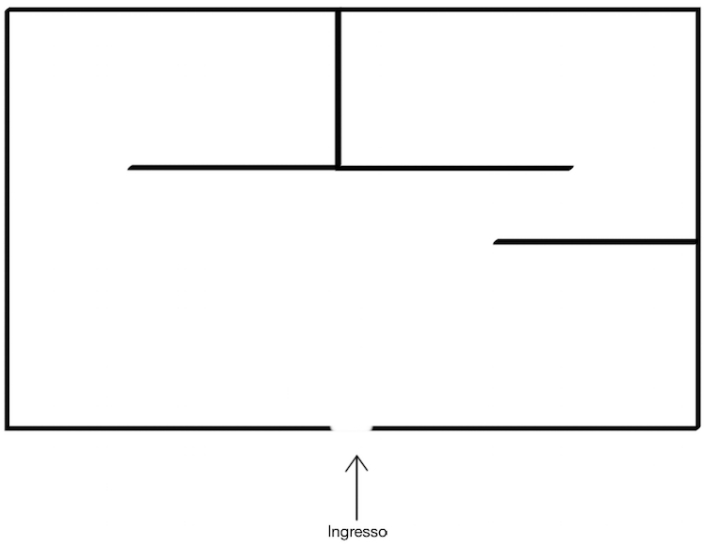

Figura 2.3 - Planimetria seconda casa

2.5 - Produzione

Questa fase inizia quando tutte le specifiche del gioco sono state chiaramente definite ed ogni componente del team sa precisamente quale sia il proprio ruolo all'interno del progetto.
Fondamentale in questa fase è la coordinazione tra i membri del gruppo in modo da rispettare le scadenze e verificando passo a passo che le specifiche vengano adeguatamente rispettate. In particolare, gli artisti si occuperanno della creazione delle animazioni dei modelli, mentre i programmatori lavoreranno su una eventuale creazione di un proprio motore di gioco (per la gestione della fisica, delle luci, delle collisioni e del movimento) e sull'IA (Intelligenza Artificiale).
Una volta che tutti i componenti del gioco sono stati realizzati il team si dedica all'ottimizzazione: questa è una fase estremamente delicata e che richiede profonde competenze. Una possibile ottimizzazione potrebbe consistere nel ridurre gli effetti di luce o gli effetti speciali, ma anche nel ridurre la qualità delle texture e riducendo al minimo il codice dell'IA; tutte queste accortezze possono fare la differenza poiché si può passare da poco più di 30 a 60 fps fissi. Questi ultimi, soprattutto negli ultimi anni, sono diventati un elemento molto importante per i videogiocatori in quanto che rendono l'esperienza di gioco estremamente fluida.
The Mad House è stato progettato per essere giocato su PC in modo tale da non avere limitazioni per quanto riguarda le performance e poter così sfruttare i potenti hardware di cui sono dotati i computer oggigiorno. La fase di produzione si è evoluta nel seguente modo: una volta progettate le case ed ideati gli ambienti si è proceduto con la relativa creazione tramite l'editor grafico integrato in Unity; tutti gli oggetti con cui il giocatore può interagire sono stati lasciati per ultimi, cosicché la scrittura degli appositi script venisse eseguita parallelamente e i test fossero effettuati nello stesso momento. Dopodiché si è passati all'implementazione del cuore del gioco, cioè allo sviluppo di tutti i singoli meccanismi che ne determinano il gameplay. Completata l'implementazione si è eseguito del refactoring in modo tale da ridurre il codice e riorganizzarlo così da essere più facilmente manutenibile; in conclusione è stata rivolta particolare attenzione all'ottimizzazione del gioco al fine di poterlo giocare (con qualità più bassa) anche su piattaforme meno performanti (tuttavia non si è testato il funzionamento su dispositivi mobili). Tutte queste fasi verranno analizzate più dettagliatamente nei prossimi capitoli.

2.6 - Post-produzione

Nella fase di post-produzione si testa tutto ciò che è stato realizzato. Per farlo, si crea una versione alpha del gioco e la si passa al team di testers: il loro compito è quello di individuare tutte le falle ed i bug e classificarli in base alla gravità.
A quel punto vengono inviati dei feedbacks al team di sviluppo il quale si

occuperà di risolverli e correggerli. Terminata questa fase il gioco è pronto per la pubblicazione e quindi per la messa in commercio.
Per ovvi motivi, tra cui la mancanza di un team che si divida i compiti e dalla mancanza di esperienza del creatore del gioco, The Mad House non risulta a lavoro concluso un prodotto vendibile; tuttavia, si è immaginato di doverlo mettere in commercio e di conseguenza si è richiesto il test a molteplici persone munite di PC di vario tipo, così da verificarne il corretto funzionamento. I bug sono stati classificati in base alla rilevanza e si è proceduto con la loro risoluzione passando poi a tutti gli errori secondari che influenzavano comunque l'esperienza utente e che necessitavano di essere risolti.

Capitolo 3 - Elementi principali

All'inizio del progetto è stato necessario individuare quelli che sarebbero poi stati gli elementi principali del gioco, capire come concretizzarli a livello implementativo e come interfacciarli gli uni con gli altri. Si possono individuare delle macro-parti che determinano il corretto funzionamento del gioco:

1. Giocatore
2. Game Manager
3. Intelligenza Artificiale dei nemici
4. Comportamento nemico principale
5. Interazione giocatore - oggetti (coltello, monete, medikit, porte, nemici, sfere…)
6. Animazioni
7. Menù iniziale

3.1 - Giocatore

Per inserire il giocatore all'interno della scena si è utilizzato un prefab pre-esistente in Unity, che possiede già gli scripts dedicati allo spostamento e alla rotazione della visuale rispettivamente tramite l'utilizzo della tastiera (sono supportati i tasti W, A, S, D e le frecce direzionali) e del mouse.
Questo prefab prevede un oggetto a forma di capsula per simulare il corpo di una persona, mentre la telecamera che consente di immergersi in una visione in prima persona è collocata ad altezza volto; in aggiunta, sono state inserite due capsule per simulare le braccia del giocatore: in una mano viene tenuto il coltello, nell'altra la torcia. Per quanto riguarda il primo, il player ha la possibilità di utilizzare due metodologie di attacco: una prevede il lancio in direzione frontale del coltello, l'altra invece prevede un colpo verticale sfruttando un'animazione appositamente creata per lo scopo. Di seguito un diagramma degli stati che ne spiega in breve la dinamica.

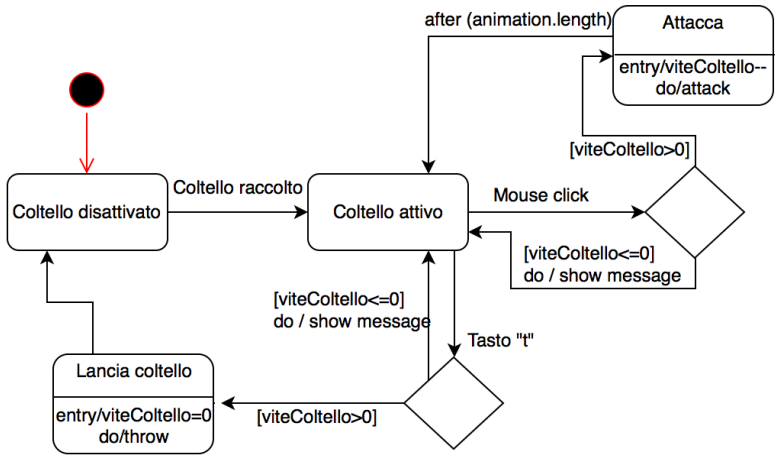

Figura 3.1 - Diagramma degli Stati rappresentante il meccanismo di utilizzo del coltello

Questo schema rappresenta gli stati attraversati dal coltello collocato nella mano del giocatore: inizialmente esso è disattivato, una volta raccoltane una copia presente nella mappa il primo viene attivato, ed il secondo distrutto, e a quel punto si resta in attesa di un'interazione da parte dell'utente (click del mouse o una pressione di un pulsante sulla tastiera) che comporta dei controlli e delle azioni che vengono ora meglio analizzate.

Pseudo-codice classe *KnifePicked.cs*:

```
private Transform knife;

void Update() {
    if (knife is active) {
        if (press t key) {
            if (knife.life > 0) {
                /* La traiettoria di lancio del coltello dipende
                dall'orientamento della telecamera */
                throw knife;

                // Si distrugge il coltello
                break knife;
            }
        } else if (right mouse click) {
            if (knife.life > 0) {
                /* Si esegue una azione di attacco verticale e la rispettiva
                animazione */
                attack;
            }
        }
    }
}
```

Il metodo *Update* si occupa innanzitutto di verificare se il coltello sia effettivamente attivo all'interno della scena: in caso positivo, si controlla se l'utente preme il pulsante "t" sulla tastiera oppure il tasto destro del mouse.
Nel primo caso si controlla lo stato di rottura del coltello, e se questo è intatto allora si eseguono una serie di operazioni che possono essere così riassunte:

1. Creazione di un clone del coltello;
2. Assegnazione della dimensione del coltello al clone;
3. Disattivazione coltello iniziale;
4. Lancio del clone dopo aver annullato sia il collegamento con l'oggetto padre sia le costanti di rotazione e posizione che caratterizzavano il coltello iniziale.

Una volta lanciato, il coltello può essere raccolto col semplice click del tasto sinistro del mouse, tuttavia essendo richiamato il metodo *Break* (che modifica lo stato dell'oggetto in "distrutto") occorrerà che il giocatore vada presso un determinato punto della mappa e ricostruirlo per poter continuare ad utilizzarlo.
Nel secondo caso, invece, viene verificato se il coltello sia rotto o meno e se non lo dovesse essere allora verrebbe avviata l'operazione di coltellata col metodo *Attacking*, implementato nella classe *Attack.cs*, del quale vediamo lo pseudo-codice:

```
private bool is_attacking;

void Attacking() {
        start (animation);
        start Coroutine (ExecuteAttack());
}

IEnumerator ExecuteAttack() {
        is_attacking = true;
        wait for (animation.length);
        is_attacking = false;
}
```

Dal punto di vista implementativo si ottiene il Component di tipo Animation assegnato al coltello e viene messa in esecuzione l'animazione. Successivamente si avvia una Coroutine che ha l'obiettivo di settare il valore booleano della variabile *is_attacking* a true per tutto il tempo di durata dell'animazione, e reimpostarlo poi a false nel momento in cui questa termina. Questo espediente consente di sapere se il giocatore stia tentando di accoltellare i nemici e quindi "registrare" il colpo solo se effettivamente l'animazione è in esecuzione. Infatti, quando avviene un contatto coltello - nemico si invoca il metodo *IsAttacking* per sapere se il colpo sia valido o meno.
Il motivo di questa scelta implementativa al posto di un semplice check sul contatto coltello - nemico è giustificato dal fatto che, in quest'ultimo caso, basterebbe che il giocatore tocchi il nemico tenendo il coltello fermo in mano per ucciderlo; questa situazione ovviamente è da evitare, di conseguenza è stata elaborata una strategia per permettere di capire quando il colpo debba essere considerato.

Per quanto riguarda invece la torcia, il suo ruolo è fondamentale all'interno del gioco ed occorre che sia sempre accesa per evitare di perdere. Una volta trovata, bisogna necessariamente rifornirsi di kit medici che fungono da pile per alimentarla.

Viene ora spiegato tramite un digramma degli stati il meccanismo di funzionamento della luce (si suppone che il GameObject della torcia sia attivo nella Scene):

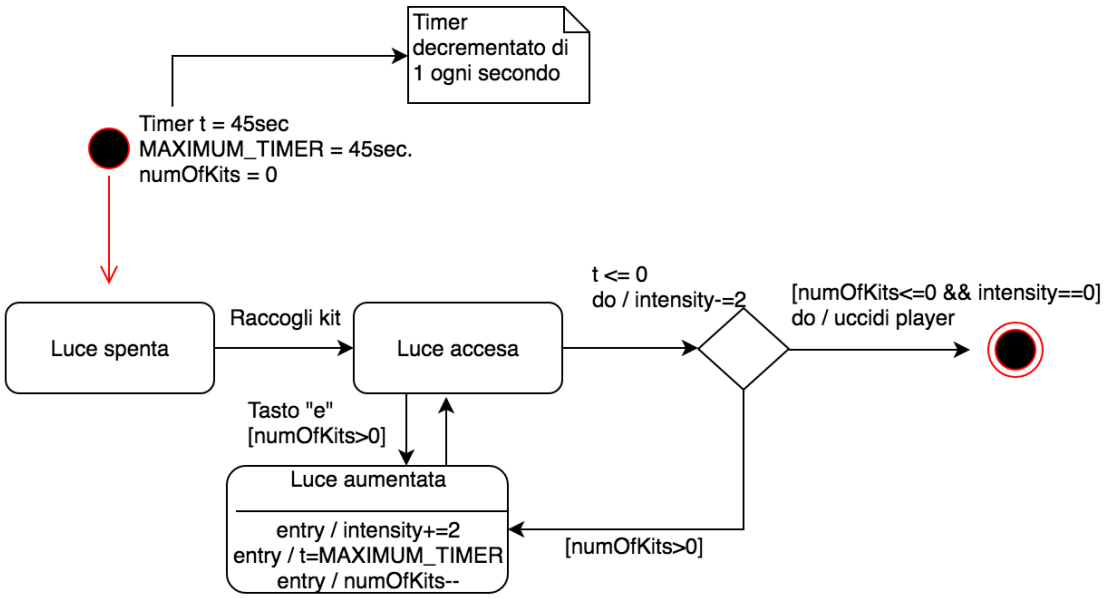

Figura 3.2 - Diagamma degli Stati del funzionamento della torcia

Si mantiene un timer, impostato ad un valore massimo assegnato ad una costante chiamata *MAXIMUM_TIMER*, che segnala al giocatore il tempo rimastogli a disposizione per una certa carica di luce; qualora il timer dovesse azzerarsi, si decrementa di 2 l'intensità della luce e si verifica se sono stati raccolti dei kit non ancora utilizzati: in tal caso il valore della luminosità della torcia viene incrementato di 2 unità e il numero di kit a disposizione decrementato di una unità. Inoltre, viene reimpostato il timer al valore massimo. In questa situazione, quindi, il kit viene automaticamente utilizzato senza l'intervento dell'utente. Tuttavia egli può effettuare questa operazione manualmente premendo il pulsante "e" della tastiera. Infine l'ultimo caso si ha quando il timer si azzera ed il giocatore non possiede alcun kit, e in questa situazione viene terminata la partita uccidendo il player.

Pseudo-codice del metodo Update nella classe *LightTimer.cs:*

private float timer;
private int lights_picked;
private Light light;
private static float MAXIMUM_TIMER = 45.0f;

```
void Update() {
	if (timer <= 0) {
		if (lights_picked > 0) {
			Recharge();
		} else if (lights_picked == 0 && light.intensity == 0) {
			manager.ExecuteDeath(DeathsEnum.BY_LIGHT, null);
		} else {
			light.intensity -= 2;
			timer = MAXIMUM_TIMER;
		}
	}

	if (press e key) {
		if (lights_picked > 0) {
			Recharge();
		}
	}

	if (light.intensity == 0) {		timer = 0;
	}

	timer -= 1 sec;
}

void Recharge() {
	lights_picked -= 1;
	light.intensity += 2;
	timer = MAXIMUM_TIMER;
}
```

3.2 - Game Manager

Il Game Manager è un'entità che gestisce molteplici aspetti del gioco. In primo luogo distribuisce in maniera randomica per ogni partita gli oggetti che l'utente può raccogliere (kit, chiavi, monete, sfere infuocate), inoltre gestisce le morti del giocatore; tutte le operazioni che hanno un impatto sul mondo o che riguardano funzionalità generali del gioco sono eseguite da questo GameObject.

3.2.1 - Disposizione randomica degli oggetti

La disposizione degli oggetti è un importante aspetto del gioco in quanto consente di rendere ogni partita diversa dalle altre. Per fare ciò si è deciso di collocarli casualmente ad ogni avvio, in modo che il giocatore non giochi mai una partita identica ad una precedente.
La strategia applicata prevede innanzitutto una struttura dati di tipo *SortedList*, che rappresenta insiemi di coppie (chiave, valore) ordinate secondo le chiavi.
Come chiavi vengono utilizzati tris di coordinate (x, y, z) che rappresentano le possibili posizioni all'interno del mondo di gioco ove possono essere posti gli oggetti, mentre come valori si mantengono dei valori booleani che indicano se la relativa chiave e quindi posizione è libera (e quindi disponibile per la collocazione di un nuovo oggetto) oppure se è già occupata.
Quando si vanno a creare gli oggetti si richiama un particolare metodo della classe *AllPositions.cs* che prende come parametro il numero di oggetti da creare di una certa tipologia e che, dopo una serie di istruzioni e controlli, restituisce una lista di tris di coordinate.

Pseudo-codice:

```
// Si ottiene la lista di tutte le posizioni in cui gli oggetti possono essere spawnati
private List<TrisCoordinates> coords = GetAllPositions();

// Numero di oggetti spawnati
private int num_set_objects;

// Indica se l'oggetto è una chiave (impostata tramite un apposito setter)
private bool is_key;

// size contiene il numero di oggetti di un certo tipo che devono essere creati
List<TrisCoordinates> GetRandomCoords (int size) {
    int random = 0;
    List<TrisCoordinates> list = new List<TrisCoordinates>();
    for (i = 0; i < size; i++) {
        if (num_set_objects < coords.size) {
```

```
            random = get random int from coords in range (0, coords.size);
            /* check sul valore booleano in corrispondenza della chiave in
        posizione randomica, se true allora la posizione è libera */
            if (!coords.Values[random]) {
                // se l'oggetto è una chiave
                if (is_key) {
                    if (coords.Keys[random].GetZ() > 167.00f) {
                        // ripeti ciclo
                        i -= 1;
                    } else {
                        /* aggiungo la posizione ottenuta alla lista
                    delle posizioni che la funzione restituisce */
                        list.add (coords.Keys[random]);
                        replace coords (coords.Keys[random], true);
                        num_set_objects += 1;
                    }
                } else {
                    list.add (coords.Keys[random]);
                    replace coords (coords.Keys[random], true);
                    num_set_objects += 1;
                }
            } else {
                i -= 1;
            }
        }  else {
            break;
        }
    }
    return list;
}
```

In sostanza si ottiene un indice randomico all'interno della lista, e si controlla il valore booleano dell'elemento in quella posizione; se esso è *false* allora la posizione è libera e si va a collocare l'oggetto che si sta creando (l'unica eccezione riguarda la collocazione delle chiavi, infatti in questo caso occorre che non siano posizionate al di fuori della villa principale altrimenti si rimarrebbe bloccati al suo interno), se invece è *true* si decrementa l'indice del ciclo *for* e si ripete il tutto fino a quando non sono stati collocati tutti gli oggetti. Per evitare la situazione in cui si rimane all'infinito all'interno del ciclo, cioè quando non vi sono più posizioni libere ma si tenta comunque di disporre ulteriori oggetti, è stata aggiunta una variabile che viene confrontata con la dimensione della lista ad ogni iterazione, e se sono uguali significa che tutte le posizioni di spawn degli oggetti sono piene.

3.2.2 - Terminazione della partita

Dal momento che vi sono più modi per concludere la partita è opportuno che ci sia un metodo per differenziarle e per eseguire specifiche operazioni in dipendenza dalla situazione in cui ci si trova.
Una partita termina quando:

1. Il giocatore guarda in faccia lo Slenderman
2. Il giocatore viene toccato tre volte da un nemico
3. La luminosità della torcia si azzera e non si hanno kit da utilizzare

Per gestire quindi le morti del giocatore si è assegnato un ID, impostato tramite una *enum*, ad ognuna di esse che viene controllato dal metodo ExecuteDeath: questo richiama specifiche operazioni in dipendenza dal valore letto. Inoltre questa funzione accetta come secondo parametro un'animazione che può essere richiesta per l'esecuzione di una determinata morte.

L'enum si presenta così:

```
enum DeathsEnum {
	BY_ENEMY = 1,
	BY_BOSS = 2,
	BY_LIGHT = 3
}
```

Pseudo-codice relativo al metodo *ExecuteDeath*:

```
// id è l'identificativo del tipo di morte, anim è l'eventuale animazione da riprodurre
void ExecuteDeath (int id, Animation anim) {
	if (id == (int) DeathsEnum.BY_BOSS) {
		// Morte causata dal nemico principale
		start Coroutine (ReproduceScream());
	} else if (id == (int) DeathsEnum.BY_ENEMY) {
		// Morte causata dall'attacco di un nemico secondario
		start death animation;
		start Coroutine (FallDeath());
	} else if (id == (int) DeathsEnum.BY_LIGHT) {
		// Morte causata dalla luminosità torcia = 0 e numero di kit = 0
		start Coroutine (ActivateDeathWall());
	}
}
```

Il seguente schema rappresenta ciò che accade in conseguenza della morte del giocatore in tutte e tre le casistiche:

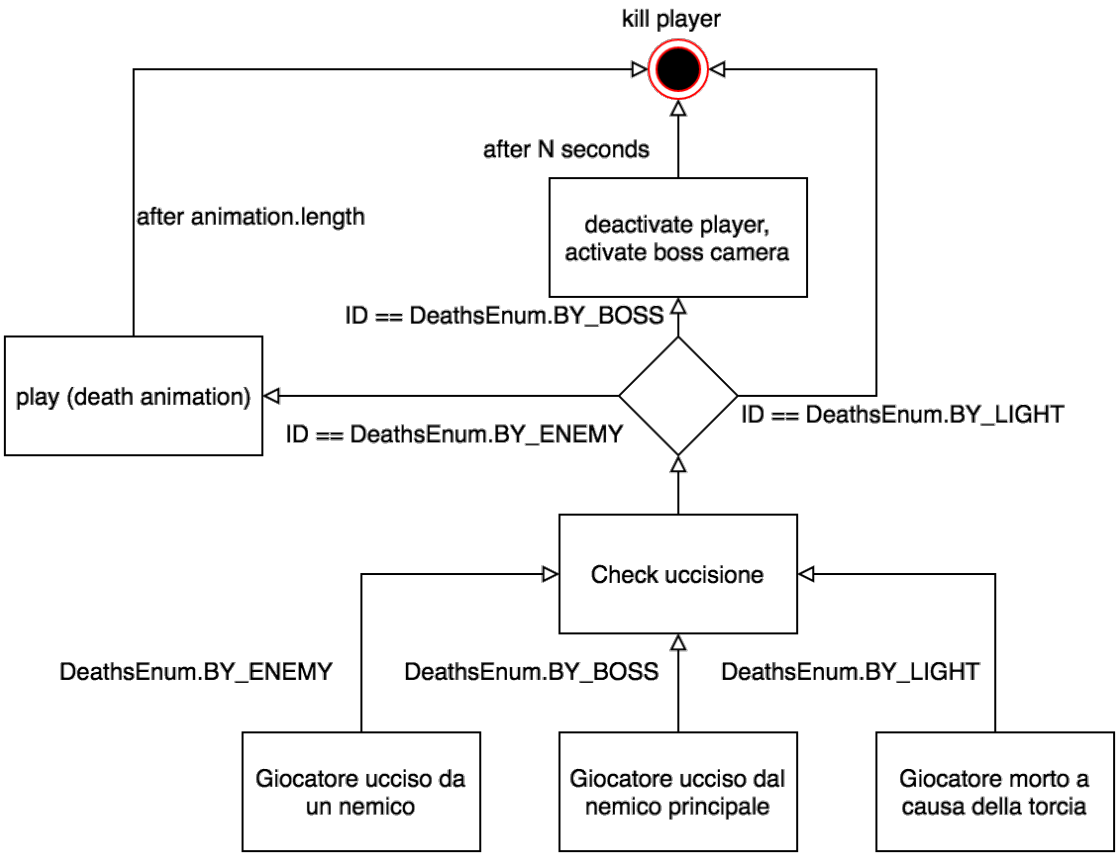

Figura 3.3 - Diagramma rappresentante le conseguenze di ogni tipo di morte

Come possibile asserire dal precedente schema ogni tipo di morte ha una o più azioni associate, difatti, se il giocatore risulta ucciso da un nemico, viene eseguita un'animazione appositamente creata, successivamente, dopo un tempo pari alla lunghezza dell'animazione, si termina la partita; nella circostanza invece di uccisione da parte del nemico principale, si attiva la telecamera posta davanti allo Slender e si disattiva il GameObject relativo al player e dopo un certo numero di secondi si conclude la partita, mentre nel caso di estinzione della luce emessa dalla torcia si andrebbe direttamente alla schermata di sconfitta.

3.3 - Intelligenza Artificiale dei nemici

Questo è, probabilmente, l'elemento più importante dell'intero gioco. Con lo scopo di rendere i nemici un ostacolo per il giocatore, si è creata un'intelligenza artificiale che li animasse e desse loro un determinato comportamento in relazione ai movimenti e alle azioni del giocatore stesso.
Tale IA non è particolarmente avanzata, tuttavia fa sì che i nemici si comportino in maniera perfettamente adeguata per un videogioco di questa tipologia.
L'obiettivo è trasmettere al giocatore un senso di inquietudine nel momento in cui un nemico inizia ad inseguirlo e renderlo quindi meno razionale nella fuga: per farlo si è aggiunto un suono di sottofondo che simula il battito cardiaco, il quale accelera quando viene inseguito e torna ad un ritmo normale quando il nemico si allontana. Questa situazione può verificarsi o quando ci si allontana molto dal nemico o quando ci si nasconde all'interno in uno degli armadi. Tuttavia in quest'ultima situazione il nemico può aprire autonomamente la porta dell'armadio e colpire il giocatore qualora quest'ultimo si fosse nascosto mentre il primo gli era vicino. Di seguito un diagramma degli stati che rappresenta l'IA sviluppata:

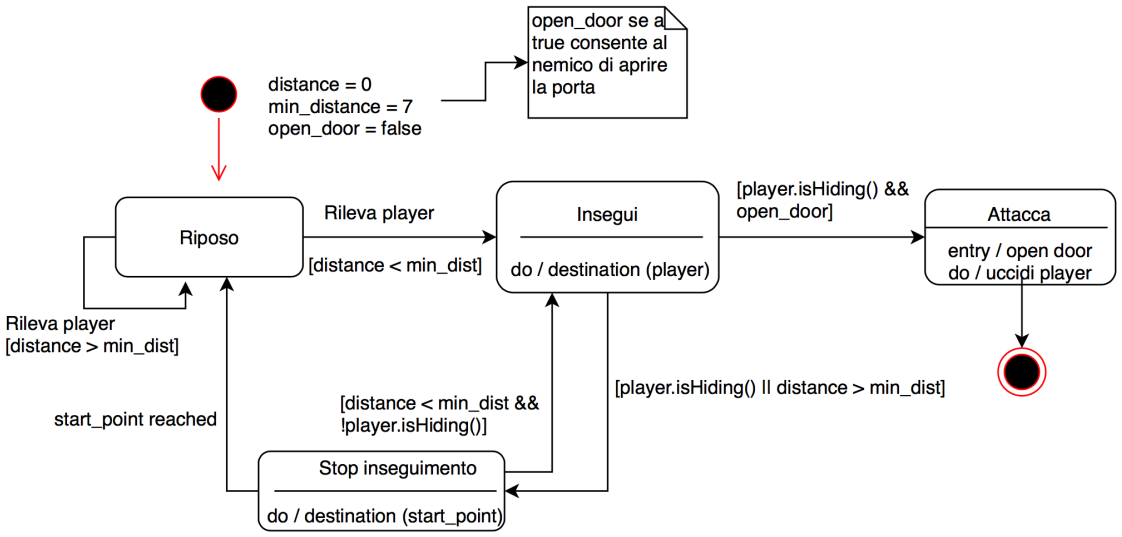

Figura 3.4 - Diagramma degli Stati che spiega il principio di funzionamento dell'IA dei nemici

Per analizzare l'implementazione occorre sottolineare come vi siano più oggetti che determinano il comportamento dei nemici. Considerando un certo armadio come nascondiglio, al suo interno è stato collocato un parallelepipedo, privo di Mesh, settato come *trigger*; il GameObject del giocatore reagisce ad un contatto

con esso e se si chiude l'anta del mobile egli è considerato come "nascosto".
A quel punto il nemico può comportarsi in due modi: se il giocatore si è nascosto mentre questo gli era vicino allora il nemico apre l'anta dell'armadio in autonomia ed uccide il giocatore, altrimenti considera il giocatore come "fuggito" e torna al proprio punto di partenza. Quando il giocatore esce dall'armadio può accadere che il nemico torni indietro se il primo rimane comunque nel suo raggio di azione, altrimenti può ricominciare ad esplorare la casa liberamente.
Dal punto di vista implementativo vi sono numerose strategie che cooperano al corretto funzionamento di questa rudimentale IA. La classe che gestisce il movimento e le dinamiche dei nemici deve essere in continua comunicazione con la classe che invece gestisce lo stato del personaggio. Per esempio, quando un nemico insegue il giocatore, occorre che si sappia quale sia tale nemico; inoltre, occorrono continui controlli sullo stato di visibilità del giocatore e sulla distanza player / nemico in modo da determinare o meno un eventuale inseguimento.

Implementazione metodo Update nella classe *EnemyMovement.cs*:

```
void Update() {
    // Si utilizza un Raycast per individuare i GameObject rilevati di fronte a sé
    if (player is seen) {
        // Variabile che indica se il nemico deve iniziare ad inseguire il player
        toFollow = true;
    }

    // Calcolo distanza nemico - player
    float distance = Distance (transform.position, player.position);
    /* distanceEnemy indica la distanza massima raggiungibile dal raggio
    d'azione del nemico */
    if (distance > distanceEnemy) {
        stop fast_heartbeat;  // Si accelera il battito cardiaco
        toFollow = false;             // Stop inseguimento
        open_door = false;
    }

    // Se il nemico deve inseguire il giocatore e questo non si sta nascondendo
    if (toFollow && !player.isHiding()) {
        set destination (player.position);       // Mando il nemico verso il player
        player.SetEnemyFollowing (transform);    // Setto il nemico corrente
        start fast_heartbeat;                    // Accelero il battito cardiaco
    }
    // Se il nemico non deve inseguire il giocatore
    if (!toFollow || (player.isHiding() && !open_door)) {
        player.SetEnemyFollowing (null);    // Reset del nemico corrente a null
```

```
            open_door = false;
            set destination (startPoint.position); // Mando il nemico al punto
iniziale
            timer -= 1;
            if (timer < 0) {          // Quando il timer si azzera
                  Rotate (0, 90, 0);      // Ruoto di 90° il nemico rispetto all'asse y
                  timer = 5;        // Reimposto il valore del timer
            }
      }
}
```

Analogamente, nella classe relativa al Player, è necessaria la presenza di alcuni metodi che impostino in modo appropriato le variabili in gioco. In particolare è da notare parte del metodo *Update* nella classe *RayCollision.cs*, della quale vediamo lo pseudo-codice per capirne il funzionamento:

```
void Update() {
      // Se il giocatore si chiude all'interno di un armadio si entra nell'if
      if (doorClosed && inWardrobe) {
            isHiding = true;
            // Si controlla se il player sia inseguito da un nemico
            if (enemy != null) {
                  // Calcolo distanza player - nemico
                  float distance = Distance (transform.position, enemy.position);
                  /* 7 è il raggio di azione del nemico, se la distanza è < 7 allora il
            nemico deve aprire la porta dell'armadio in cui si nasconde il
            giocatore ed ucciderlo */
                  if (distance < 7) {
                        enemy.OpenDoor (true);
                  } else {
                        enemy.OpenDoor (false);
                  }
            }
      } else {
            isHiding = false;
      }
      /* ... CODICE ... */
}
```

In questo pezzo di pseudo-codice si vede come venga richiamata una funzione *OpenDoor*, alla quale viene passato un valore booleano, propria del nemico. Questo metodo va ad impostare la variabile *open_door* del codice visto precedentemente dipendentemente dalla distanza nemico - player nel momento in cui quest'ultimo si nasconde all'interno dell'armadio. Inoltre, le variabili *doorClosed* e *inWardrobe* vengono rispettivamente settate quando viene chiusa l'anta dell'armadio e quando il giocatore collide con il GameObject Trigger collocato all'interno dello stesso, con lo scopo di capire quando ci si stia effettivamente tentando di nascondere.

Un'importante feature offerta da questa versione di Mono riguarda la possibilità di permettere ad un oggetto di spostarsi fisicamente all'interno della scena, da un punto ad un altro, semplicemente sfruttando un paio di metodi. Il meccanismo sfrutta le *NavMesh*, che rappresentano geometricamente la superficie calpestabile da un qualsiasi GameObject.
Per creare una *NavMesh* occorre prima eseguire il baking di tutta l'area impostata come navigabile, ovvero bisogna selezionare tutte le zone sulle quali si possono muovere autonomamente gli oggetti e settarle come "Navigation Static" dall'Inspector; a quel punto si deve aggiungere il Component *Nav Mesh Agent* al GameObject target e, via codice, invocare il metodo *SetDestination*, il quale richiede come argomento un oggetto di tipo *Vector3*, indicando la destinazione finale.
Le *NavMesh* vengono utilizzate per consentire agli agenti, entità autonome utilizzate nel campo dell'intelligenza artificiale, di trovare il percorso da un punto ad un altro in ambienti vasti e in modo intelligente. Fondamentalmente si suddivide l'ambiente stesso in poligoni, e per la ricerca del percorso attraverso di essi si applicano particolari algoritmi di ricerca nei grafi come l'A* o il D*.
Le *NavMesh* definiscono quindi l'aera percorribile dagli agenti, così da evitare tutti quei calcoli complessi e dispendiosi a livello di risorse che si avrebbero se si andassero ad eseguire continui check sul rilevamento di collisioni; grazie a questi potenti algoritmi, gli agenti possono spostarsi evitando qualsiasi tipo di ostacolo presente nella mappa.
La funzionalità delle *NavMesh* in Unity risulta sicuramente una delle più utili e salva-tempo, e poiché rappresentano un perfetto compromesso tra performance e semplicità di implementazione si sono rivelate la scelta più appropriata per questo tipo di progetto.

3.4 - Comportamento del nemico principale

Il nemico principale è tratto dal gioco cui questo progetto si ispira, ovvero *"Slender: The Eight Pages"*; il modello è dunque il medesimo ed il comportamento molto simile.
In sostanza ciò che questo nemico fa è essere spawnato casualmente nella mappa in punti precedentemente stabiliti, fino a quando non vede il giocatore ad una certa distanza: se ciò accade, esso inizia a spostarsi nel punto di spawn più vicino al player tentando di incrociarne lo sguardo. Infatti se il giocatore lo guarda in viso verrà catturato e la partita terminerà con una sconfitta.
Tuttavia, per aiutare l'utente a percepire il pericolo, si è simulata una interferenza nella telecamera aggiungendovi un suono appropriato.
In *"Slender: The Eight Pages"* il comportamento dello Slender è determinato da un algoritmo più particolare, ad ogni modo quello realizzato per questo progetto lo emula piuttosto bene.
Nel seguente pseudo-codice si può vedere il principio su cui si basa il comportamento del nemico principale:

```
void FixedUpdate() {
    timer -= 1 sec;
    if (player is seen) {
        if (distance player-enemy < 20) {
            seen = !seen;
        }
        if (distance player-enemy <= 7) {
            if (player.IsLookingAtEnemy()) {
                player.SetActive (false);          // Telecamera giocatore off
                enemy_camera.SetActive (true); //  Telecamera nemico on
                manager.ExecuteDeath (DeathsEnum.BY_BOSS, null);
            }
        }
    }

    if (timer < 0) {
        if (!seen) {
            TrisCoordinates tc = GetRandomPosition();
            instantiate boss at position tc;
        } else {
            // Trovo il punto di spawn più vicino al giocatore
            Vector3 min = new Vector3 (
                            coords[0].GetX(),
                            coords[0].GetY(),
                            coords[0].GetZ()
```

```
                    );
                Vector3 temp;
                for (i = 0; i < coords.size; i++) {
                    temp = new Vector3 (
                            coords[i].GetX(),
                            coords[i].GetY(),
                            coords[i].GetZ()
                        );
                    if (dist > Distance (temp, player.position)) {
                        min = temp;
                        dist = Distance (temp, player.position);
                    }
                }
                instantiate boss at position min;
            }
            Destroy (transform);
        }
}
```

Per poter uccidere il giocatore quando questo nemico viene guardato direttamente nel viso si è applicato un GameObject, privo di mesh ma dotato di un Box Collider e di un particolare tag, che coprisse la parte superiore del corpo dello Slender. Quando il giocatore guarda questa sezione si entra nell'if *if(player.IsLookingAtEnemy())* e si termina la partita.

3.5 - Interazione giocatore - oggetti

Dal momento che il player può interagire con una moltitudine di oggetti nel gioco si è deciso di fare uso dei principali metodi che la libreria di Mono mette a disposizione del programmatore, che necessitano di appropriate impostazioni per poter funzionare correttamente.
Fondamentalmente si possono individuare due tipi di corpi:

1. Collider
2. Trigger

La differenza sostanziale è che quando un Collider, ovvero un corpo dotato di questo Component, collide con un altro Collider si applica una forza come conseguenza di questa collisione. Un Trigger, d'altra parte, come un Collider registra la collisione ma non viene generata alcuna forza in risposta.

Pe quanto concerne l'uso dei Trigger, in The Mad House molte funzionalità dipendono da questo tipo di corpo. Una di esse consiste nell'apparizione di jumpscares (eventi improvvisi che spaventano il giocatore), e un esempio è dato da ciò che accade quando il player attraversa il Trigger collocato all'ingresso della casa principale:

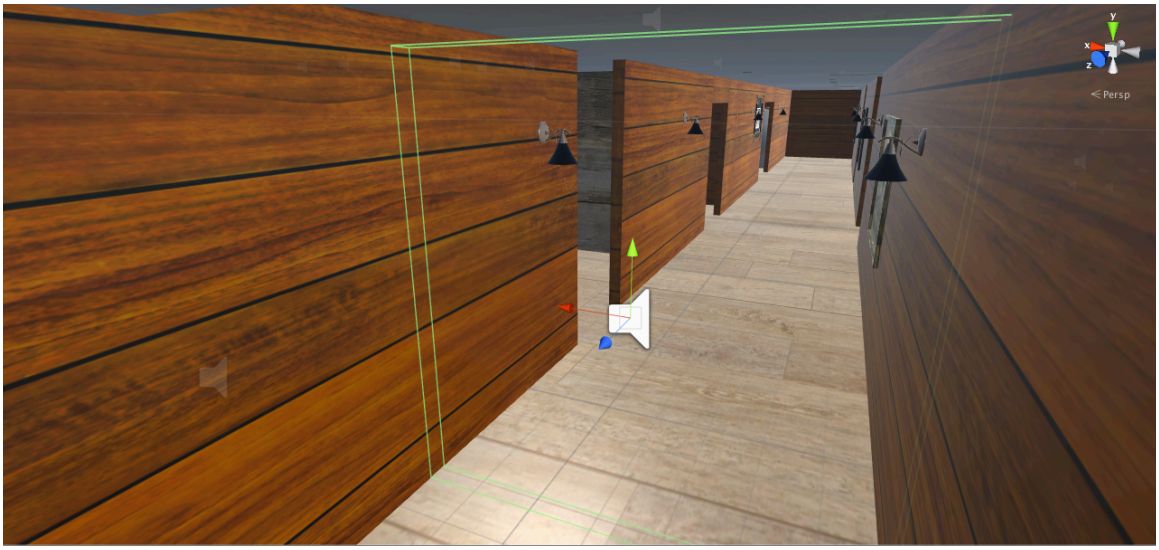

Figura 3.5 - Visuale dall'alto del corridoio principale: l'oggetto selezionato è il Trigger posto all'ingresso

Selezionato questo corpo, all'interno dell'Inspector è stato disattivato il Component *Mesh Render* il quale renderizzerebbe la Mesh applicatagli ed è stata settata l'opzione *Is Trigger* nel Component *Box Collider*. Affinché la collisione provochi una certa reazione, questa deve essere ovviamente definita

nell'implementazione della classe assegnata al Trigger. In questo caso l'azione in risposta alla collisione viene eseguita quando il giocatore esce dal contatto con il Trigger (pseudo-codice):

```
void OnTriggerExit (Collider coll) {
        if (coll.tag == "Player") {
                start Coroutine (Scare());
        }
}

IEnumerator Scare() {
        play (scream_sound);
        monster.SetActive (true);
        wait for (time);
        monster.SetActive (false);
}
```

Nella funzione *OnTriggerExit* si controlla che il GameObject uscito dal contatto abbia il tag *Player*, e in caso positivo si avvia la Coroutine che attiva all'interno della scena un mostro (il cui scopo è quello di spaventare l'utente) ed un suono di durata molto breve; dopo un arco di tempo impostato tramite una variabile pubblica denominata *time* si disattivano sia il mostro che il Trigger.

Anche i Collider si sono rivelati fondamentali per poter implementare numerose funzionalità, una delle principali prevede l'interazione con oggetti e porte. È infatti opportuno osservare la progettazione effettuata per l'interazione con queste ultime, poiché in tutta la mappa se ne trovano circa 40 e il giocatore è quasi obbligato ad interagirvi dal momento che appartengono a mobili all'interno dei quali ci si può nascondere e si possono trovare oggetti utili. Per questo motivo si è elaborata un'organizzazione stabile e

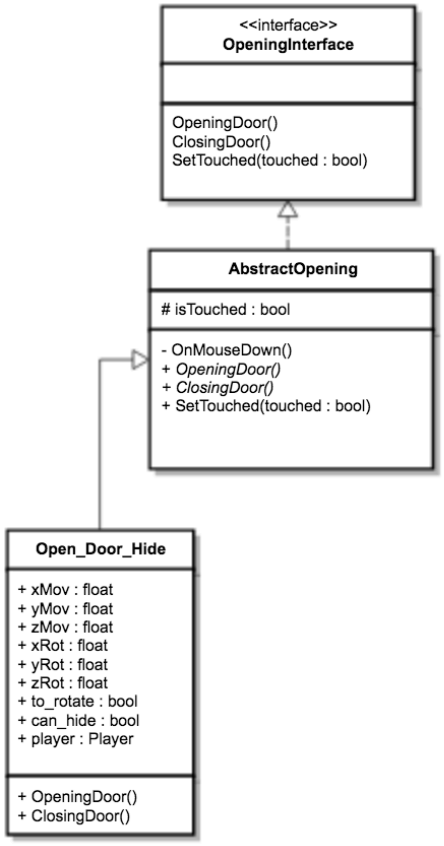

Figura 3.6 – Struttura realizzata per gestire le interazioni con le porte

facilmente estendibile.
La classe *Open_Door_Hide* è la classe concreta che implementa i metodi ereditati dalla classe astratta *AbstractOpening* i quali servono per aprire e chiudere le porte. Questa classe a sua volta implementa l'interfaccia *OpeningInterface* in modo tale da rendere la struttura più solida e poter interagire tramite essa.
Per quanto concerne l'effettiva apertura e chiusura delle porte, si fa uso di variabili pubbliche di tipo *float* e *bool* per determinare se una porta sia scorrevole o a battente e quindi consentire un'apertura dipendente da un solo spostamento di posizione oppure se da una traslazione più una rotazione. Nell'Inspector infatti si settano le variabili come nell'immagine:

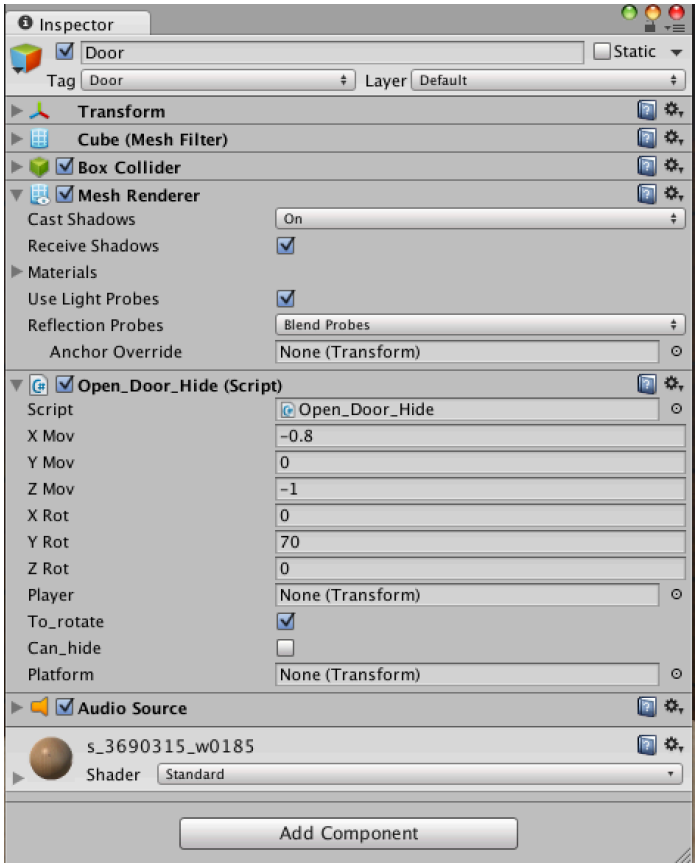

Figura 3.7 - Settaggio variabili pubbliche per l'apertura di una certa porta

La variabile pubblica *to_rotate* viene spuntata per quelle porte che oltre allo spostamento devono subire anche una rotazione, e tramite le variabili *xMov, yMov, zMov, xRot, yRot, zRot* si impostano i valori che verranno utilizzati nei calcoli vettoriali applicati per l'apertura. Infine la variabile *can_hide* indica se la porta in questione appartiene ad un armadio all'interno del quale ci si può nascondere. Un'ulteriore tipologia di interazione riguarda quella tra giocatore e nemici. Questa collisione può determinare la sconfitta e quindi il termine della partita, e per rilevarla si utilizza il metodo *OnCollisionEnter* messo a disposizione da Mono.

3.6 - Animazioni

Le animazioni sono un aspetto molto importante in qualsiasi videogioco, dal più banale al più complesso, poiché rendono l'esperienza di gioco più piacevole se realizzate in maniera discretamente adeguata.
Per questa tesi ne sono state realizzate alcune senza possedere un background in questo campo, di conseguenza il risultato non è un lavoro professionale ma sufficientemente adatto al tipo di progetto.
Come già detto precedentemente per poter creare e gestire le animazioni in Unity si fa uso del tool messo a disposizione dal programma stesso. Questo strumento si presenta nel seguente modo:

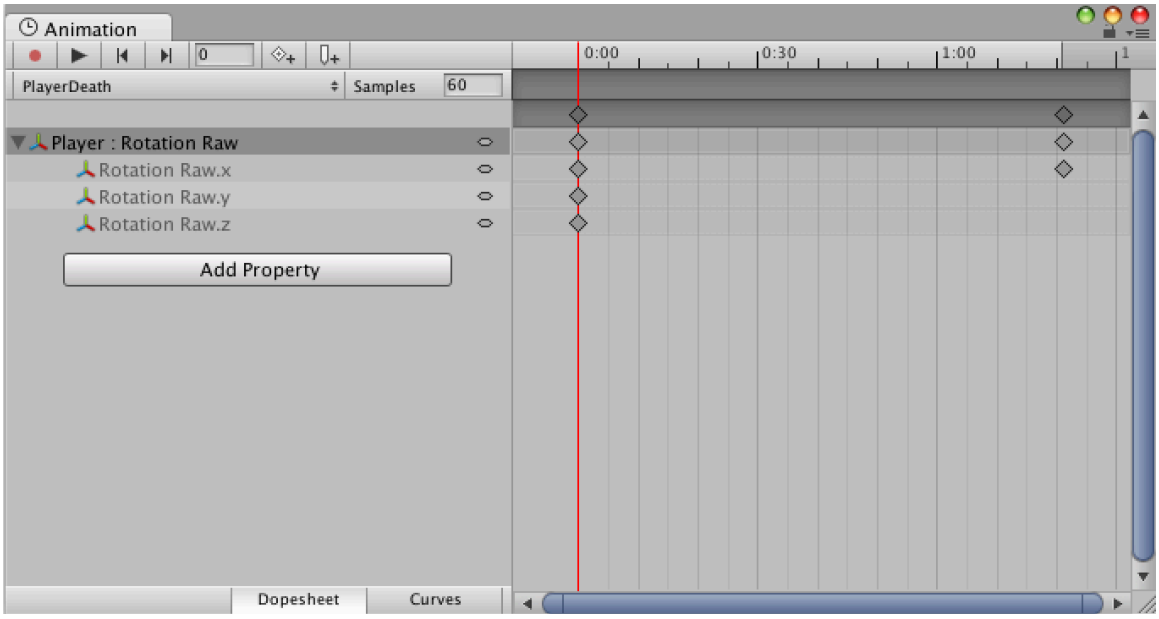

Figura 3.8 - Tool di animazione

Per quanto riguarda le animazioni realizzate per The Mad House, se ne possono individuare alcune interessanti:

1. Animazione del coltello per attaccare;
2. Animazione eseguita quando si viene uccisi da un nemico;
3. Pseudo-animazione realizzata per l'apertura della porta di ingresso di entrambe le case presenti nella mappa.

Chiaramente questi sono dettagli in più e funzionalità secondarie del progetto, elaborate per studiare ed apprendere l'utilizzo del tool che Unity mette a disposizione, il quale risulta intuitivo, di facile utilizzo e totalmente adatto per una eventuale necessità di creare animazioni di maggiore complessità.

3.7 - Menù iniziale

Come in qualsiasi altro videogioco anche in The Mad House è presente un menù iniziale che offre semplicemente due opzioni: Start ed Exit. Non sono state scritte istruzioni per giocare in quanto si vuole lasciare all'utente possibilità di capire le semplici meccaniche di gioco, senza influenzarlo nelle scelte.

Di seguito un'immagine di come si presenta il menù:

Figura 3.9 - Menù iniziale

Molto banalmente premendo Start si viene catapultati nel mondo di gioco, in un punto ben preciso della mappa, mentre premendo Exit il programma viene chiuso.

3.8 - Immagini del gameplay

Di seguito verranno mostrati alcuni screenshots di una partita, per rendere meglio l'idea di come The Mad House si presenta in varie fasi di gioco.

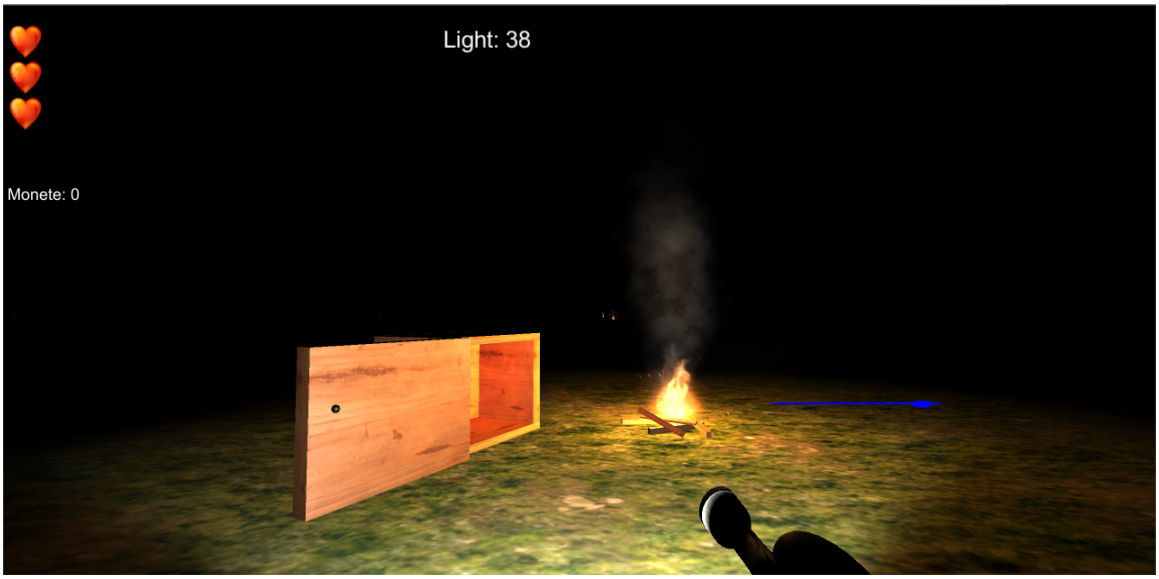

Figura 3.10 - Inizio del gioco: il giocatore ha appena raccolto la torcia ed il kit per alimentarla e la freccia punta al punto successivo.

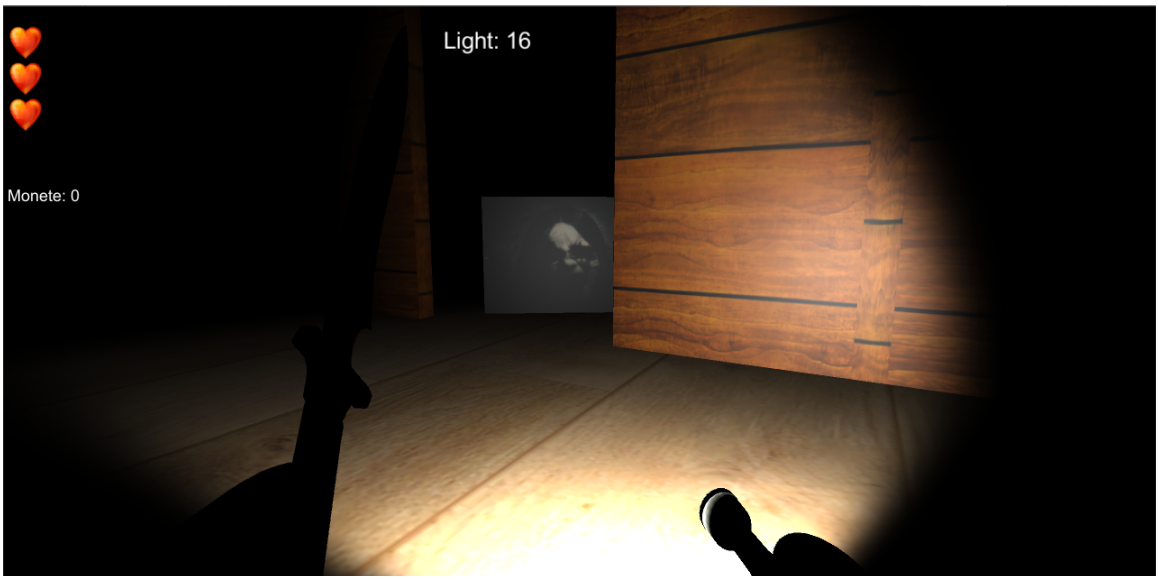

Figura 3.11 - Un nemico sta inseguendo il giocatore dopo averlo rilevato.

Figura 3.12 - Ritrovamento di due oggetti in una panca: il giocatore ha aperto le porte del mobile all'interno del quale si trovono una sfera infuocata ed un kit.

Figura 3.13 - Il giocatore viene rilevato dallo Slenderman: un punto di spawn è infatti davanti alla casa secondaria.

Test

La fase di testing di The Mad House è stata effettuata su sistemi Windows e Mac OS. Il gioco è risultato altamente stabile su tutti i terminali di cui si è fatto uso, compreso su un computer di fascia bassa.
Di seguito le configurazioni dei computer utilizzati:

OS X El Capitan → ~70 fps
i5 2.60 GHz
16 GB RAM
Intel Iris Graphics

Windows 8.1 → ~70 fps
i5 2.60 GHz
16 GB RAM
Intel Iris Graphics

Windows 7 → ~80 fps
i7-6700 3.40 GHz
16 GB RAM
Intel HD Graphics 530

Windows 10 Pro → ~70 fps
i5-4670K 3.40 GHz
8 GB RAM
Intel HD Graphics 4600

Windows 10 → ~55 fps
Intel Celeron CPU N2940 1.83 GHz
2 GB RAM
Intel HD

Windows 7 Pro → ~85 fps
i5-2500K 3.50 GHz
4 GB RAM
ATI Radeon HD 7770

Sviluppi futuri

The Mad House può essere considerato a tutti gli effetti un punto di partenza per un videogioco più ampio e profondo. Nonostante le funzionalità implementate non presentino gravi problematiche in tutti i casi testati, il gioco non è esente da bug e possibili migliorie: in particolare nel campo delle luci, della musica, delle textures e dei modelli 3D. Ad esempio i nemici necessitano di un modello tridimensionale adeguato poiché in questo progetto sono dei banali cubi ai quali è stata applicata una texture (a causa della mancanza di conoscenze in ambito di modellazione 3D).
Potrebbe anche essere necessaria un'ottimizzazione, in quanto in certe situazioni possono verificarsi cali considerevoli di FPS (perdite dai 13 ai 15 fps, con circa 65-80 fps costanti, dipendentemente dal PC) per qualche secondo.
Un altro aspetto sul quale ci si può concentrare per un ipotetico sviluppo futuro è quello della storyline, elemento del tutto assente in questo videogioco non essendo centrale per gli scopi prefissatisi, che potrebbe riguardare il salvataggio di qualcuno legato al personaggio e così via. Le possibili idee sono svariate e occorre quindi una figura che le riordini e che, insieme al Project Manager, scelga la più opportuna, o quella che si crede possa riscuotere più successo oppure ancora che possa rivelarsi più interessante.
Inoltre un possibile e molto stimolante sviluppo futuro potrebbe vedere il gioco portato su altre piattaforme ma soprattutto su visori per la Realtà Virtuale, tecnologia che si presta perfettamente per questo genere di prodotti.

Conclusioni

Questa tesi ha preso in analisi lo sviluppo di un generico videogioco, partendo dalla prima idea fino alla messa in commercio, applicando le varie fasi alla creazione di un gioco iniziando dal principio utilizzando il game engine Unity; esso è risultato la scelta migliore in quanto ha consentito di utilizzare il linguaggio di programmazione C#, il quale era già stato appreso per fini diversi, e contemporaneamente di mantenere performance elevate.
Poiché Unity è risultato un tool completamente nuovo per il tesista, si ha avuto la necessità di studiarne a fondo molti aspetti, soprattutto legati allo scripting e quindi si è dovuto analizzare approfonditamente l'enorme libreria che Mono mette a disposizione per capire il funzionamento di determinati meccanismi. Inoltre si è dovuto anche analizzare l'editor grafico e ciò che si può fare grazie ad esso, quindi l'uso dell'Inspector, il principio dei *Components*, e così via.
Il risultato del lavoro prende il nome di The Mad House, un videogioco Horror in prima persona: nel corso della Tesi ne sono state osservate le fasi di sviluppo dall'ideazione alla fase di testing finale, passando per la progettazione degli aspetti fondamentali (come la definizione del gameplay e la creazione delle mappe) e degli algoritmi e per la vera e propria fase di produzione e quindi di scrittura del codice.
La funzionalità più interessante è sicuramente l'IA dei nemici che, affiancata a tutte le altre funzionalità offerte, garantisce un tempo di gioco soddisfacente ed intenso.
Più in generale è possibile affermare che lo sviluppo di videogiochi risulta essere uno dei campi più ostici dell'informatica: affinché il prodotto risultante sia all'altezza del mercato, soprattutto negli ultimi anni, deve possedere caratteristiche che lo contraddistinguano in mezzo a tutti gli altri titoli, in particolar modo tra quelli del medesimo genere; per raggiungere questo obiettivo si necessita di un team solido, coordinato, ben costituito, equilibrato e con una volontà d'acciaio, ma anche di fondi da investire, senza i quali non si potrebbe procedere nella produzione. Servono persone qualificate, professionisti nel loro ruolo che impieghino la loro esperienza per creare un prodotto di elevata qualità ed innovativo.
Ad ogni modo, l'obiettivo che ci si era prefissati all'inizio è stato raggiunto con successo, e il risultato può essere considerato molto buono, nonostante le problematiche prima descritte, considerando la mancanza di conoscenze di base iniziali e le difficoltà incontrate.

Sitografia

[1] Unity Technologies, *Unity Documentation,* http://docs.unity3d.com/Manual/UnityManual.html

[2] Tom DiCristopher, http://www.cnbc.com/2016/01/26/digital-gaming-sales-hit-record-61-billion-in-2015-report.html , 26/01/2016

[3] Unity Technologies, http://answers.unity3d.com

[4] Matheus Amazonas, https://sometimesicode.wordpress.com/2014/12/22/how-does-unity-work-under-the-hood/ , 22/12/2014

[5] Listverse, http://listverse.com/2010/05/11/15-firsts-in-video-game-history/ , 11/05/2010

[6] Mike Geig, http://www.informit.com/articles/article.aspx?p=2031153 , 04/04/2013

[7] Guido Iodice, https://guiodic.wordpress.com/2009/06/13/due-parole-su-mono-e-sul-perche-non-e-necessariamente-il-male/ , 13/06/2009

[8] Mono Project, *Mono Documentation,* http://www.mono-project.com/docs/

[9] Pascal Bestebroer, http://www.gamasutra.com/blogs/PascalBestebroer/20151020/256765/The_four_phases_of_game_development.php , 20/10/2015

[10] StackOverflow, http://stackoverflow.com

Bibliografia

[1] The Process of Game Development, *Pearson*, https://www.pearsonhighered.com/samplechapter/0672326922.pdf

[2] “Beginning 3D Game Development with Unity 4, All-In-One, Multi-Platform Game Development” 2nd Edition, *Sue Blackman*

Ringraziamenti

In primo luogo ringrazio la mia famiglia, che mi è sempre stata vicina sin dal primo istante di questo percorso e che mi ha sempre incoraggiato. Senza di loro non sarei mai potuto riuscire ad arrivare fino in fondo, ci sono sempre stati e sono certo ci saranno sempre per me, indipendentemente dalla strada che sceglierò di intraprendere. Quindi grazie, vi voglio bene.
Vi ringrazio anche per il meraviglioso regalo di Laurea, non potrei davvero essere più felice di così.

Un grande ringraziamento va anche ai miei amici all'Università, coi quali ho passato tre bellissimi anni, in particolare Fabbro e Giulia: tutte le giornate passate insieme a studiare hanno reso questi anni incredibilmente piacevoli.

Tuttavia un grazie particolare va ad Aldo, non solo per avermi stimolato a migliorare e a dare il meglio nei progetti e nei lavori che abbiamo realizzato insieme, ma soprattutto per essermi stato vicino nei momenti difficili: con i suoi consigli e suggerimenti trovava sempre una soluzione e riusciva a tirarmi su il morale. Mi ha aperto gli occhi a tante cose e grazie a lui sono completamente cambiato rispetto al ragazzo che ero tre anni fa. Sono sicuro l'amicizia instaurata proseguirà nel tempo, ed è questa la cosa più importante, ma sono anche certo che riusciremo a portare a termine vari progetti futuri che abbiamo in mente.

Ringrazio anche il mio relatore, il Dott. Mirko Ravaioli, per essersi dimostrato interessato alla mia idea di progetto e per essersi sempre reso disponibile nel darmi indicazioni per poter realizzare un lavoro valido.