POLITECNICO DI MILANO

Facoltà di Ingegneria

Corso di Laurea Magistrale in Ingegneria Informatica

SVILUPPO DI UN VIDEOGIOCO PLATFORM DALL'IDEA ALLA REALIZZAZIONE

Relatore: Prof. Pierluca LANZI

Tesi di Laurea di:Michele CLABASSI
Matr. 766656

Anno Accademico 2011 / 2012

Indice

1. Sommario.....3
2. Elementi formali nei videogiochi.....4
  2.1. Conflitto e risoluzione.....5
3. Il Game Design Document.....6
4. Dal concept al Game Design Document finale.....7
5. Analisi degli elementi formali.....11
  5.1. Giocatore.....11
  5.2. Obbiettivo.....11
  5.3. Procedure.....11
  5.4. Regole.....12
    5.4.1. Regole che definiscono oggetti e concetti.....12
    5.4.2. Regole che restringono le azioni possibili.....12
    5.4.3. Regole che determinano degli effetti.....12
  5.5. Risorse.....13
6. Glossario di Unity.....15
  6.1. Gameobject.....15
  6.2. Mesh.....15
  6.3. Animazioni.....15
  6.4. Texture.....15
  6.5. Materiale.....16
  6.6. Motore fisico.....16
  6.7. Rigidbody.....16
  6.8. Collider e trigger.....16
  6.9. Rilevamento delle collisioni.....16
  6.10. Emettitore di particelle.....17
  6.11. PlayerPrefs.....17
  6.12. Script.....17
    6.12.1. Awake.....17
    6.12.2. Start.....18
    6.12.3. Update.....18
    6.12.4. LateUpdate.....18
    6.12.5. Eventi del mouse.....18
    6.12.6. Rilevazione di collisioni – Trigger.....18
    6.12.7. Rilevazione di collisioni – Collision.....18
    6.12.8. Linecast.....19
    6.12.9. RaycastHit.....19
7. Implementazione.....21
  7.1. Schermata iniziale.....21
  7.2. Stato del gioco.....21
  7.3. Personaggi - Dumb.....21
    7.3.1. Dumb – GameObject.....21
    7.3.2. Dumb – Movimenti.....22
  7.4. Personaggi - Mostri.....24
    7.4.1. Mostri – GameObject.....24

7.4.2. Mostri – Percezione....24
7.4.3. Il tag “DeadEnd”....24
7.4.4. Mostri – Il nucleo dell'IA....25
7.4.5. Mostri – Movimento....25
7.5. Personaggi – M.E.A.N....29
7.5.1. M.E.A.N. - GameObject....29
7.5.2. I checkpoint....29
7.5.3. M.E.A.N. - Percezione....30
7.5.4. M.E.A.N. - Il nucleo della IA....31
7.5.5. M.E.A.N. - Movimento....33
7.6. Informazioni sullo schermo....42
7.6.1. Tempo rimanente....42
7.6.2. Salute....42
7.7. Telecamera....42
7.8. Piattaforme....43
7.8.1. Piattaforme rotanti....44
7.8.2. Piattaforme sganciabili e a due posizioni....44
7.8.3. Piattaforme mobili....45
7.9. Punteggio....46
7.9.1. Fine del livello....46
7.9.2. Visualizzazione del punteggio....47
8. Strumenti usati....48
8.1. Strumenti per l'organizzazione del lavoro....48
8.1.1. Google Docs....48
8.2. Strumenti per l'implementazione....48
8.2.1. Unity 3D....48
8.2.2. JavaScript....49
8.2.3. C#....49
8.2.4. Mono e MonoDevelop....49
8.3. Grafica & audio....50
8.3.1. Blender....50
8.3.2. Adobe Photoshop....50
8.3.3. Audacity....51
9. Conclusioni....52
9.1. Possibili sviluppi futuri....52
10. Riferimenti bibliografici....54

1. Sommario

Lo sviluppo di un videogioco è un processo che richiede come punto di partenza un concept, e il prodotto finale è raggiunto solo attraverso una serie di processi, una parte dei quali è solo remotamente legata all'aspetto tecnologico.

Questo elaborato di tesi ripercorre le fasi di un progetto legato al corso di Videogame Design and Programming, e sviluppato a partire da un'idea di Luca Pinato (Università degli Studi di Milano).

Partendo da una premessa generale sugli aspetti caratterizzanti dei videogiochi, si analizzerà l'evoluzione dal concept al Game Design Document, e da qui al gioco nelle sue fasi intermedie e finale.

Saranno messi in rilievo prima gli aspetti formali e drammatici del gioco, e in seguito le soluzioni implementative adottate, con particolare attenzione all'intelligenza artificiale dei personaggi.

2. Elementi formali nei videogiochi

I videogame sono descrivibili nelle loro componenti essenziali tramite i cosiddetti elementi formali, quelli che caratterizzano la struttura del gioco.

Il primo elemento è il **giocatore**, o i giocatori. I videogiochi sono progettati per giocatori, che accettano volontariamente le regole e le restrizioni che il gioco comporta. I giocatori sono caratterizzati da un numero (che può andare dal singolo giocatore ai grandi numeri dei giochi detti "massively multiplayer") e dal ruolo che questi assumono all'interno del gioco. Il ruolo è molto spesso lo stesso per tutti i giocatori, e nel caso in cui non sia così (es. World of Warcraft, in cui i personaggi hanno peculiarità, caratteristiche, e obbiettivi diversi) è richiesta una particolare attenzione all'equilibrio tra i diversi ruoli.

Vi sono anche numerosi modelli di interazione dei giocatori, tra cui i più comuni nel mondo videoludico sono:

- Single Player vs Game, cioè quando è presente un solo giocatore che compete con il gioco stesso per raggiungere i propri obbiettivi
- Player vs Player, in cui due giocatori si affrontano uno contro l'altro
- Competizione multilaterale, o detta in parole semplici, tutti contro tutti
- Gioco cooperativo, nel quale più giocatori collaborano al raggiungimento di obbiettivi comuni
- Competizione di squadra, in cui due o più gruppi di giocatori si affrontano tra di loro

Il successivo elemento formale sono gli **obbiettivi**. Questi dovrebbero essere idealmente impegnativi e al contempo raggiungibili, così da coinvolgere il giocatore. Possono essere anche molteplici, parziali, mini-obbiettivi, e vanno considerati sotto questa categoria anche gli obbiettivi determinati dai giocatori stessi. Un'implicazione degli obbiettivi è il tono del gioco stesso, che può variare notevolmente anche a seconda della scelta fatta; conviene quindi considerare attentamente che obbiettivi porre al giocatore, poiché ciò non rappresenta un elemento puramente formale, bensì anche un aspetto "drammatico" del gioco.

Alcuni generici esempi di obbiettivo possono essere la soluzione di un rompicapo, una gara tra i giocatori o tra il giocatore e altri personaggi controllati dal computer, una fuga o un inseguimento, il salvataggio di un personaggio, l'esplorazione, e via dicendo.

Altro elemento formale sono le **procedure** del gioco, in altre parole il modo in cui il giocatore può agire, le azioni che può svolgere per raggiungere gli obbiettivi. Per la maggior parte dei giochi si distingue tra:

- preparazione del gioco
- proseguimento dell'azione
- azioni speciali (che possono avvenire solo in determinati momenti del gioco)
- azioni risolutive

Quando si parla di videogame, tali procedure sono solitamente integrate nei controlli del gioco. Sono legate ai dispositivi utilizzati per l'input e l'output, e al tipo di ambiente in cui il

giocatore userà il gioco.

Vi sono poi le **regole**, che definiscono ciò che il giocatore può fare e quali sono le sue limitazioni; nell'ambito videoludico sono spesso implicite nel gioco stesso.

È necessario bilanciare le regole e il modo in cui esse sono memorizzate dai giocatori: un eccesso di regole rende complicato il loro apprendimento, per contro se esse non sono esplicitate o sono poco chiare questo può creare confusione.

Altro elemento rilevante sono le **risorse**, che si possono usare nel gioco per raggiungere un obbiettivo. Gli esempi più tipici sono le vite, la salute, molteplici unità, denaro o risorse equivalenti, azioni, power-up, tempo, e via dicendo.

2.1. Conflitto e risoluzione

Come all'interno di un racconto, i videogiochi si articolano attraverso un processo che attraversa una o più fasi di conflitto, seguite da una risoluzione.

Il **conflitto** è introdotto nel gioco attraverso regole, procedure e situazioni, le quali forzano il giocatore a ricercare una soluzione indiretta al raggiungimento dei propri obbiettivi. I classici elementi di conflitto sono ostacoli, avversari e dilemmi.

La **risoluzione**, ovvero l'esito del gioco, deve essere incerta per mantenere l'attenzione del giocatore. Un esito scontato, ad esempio un videogame in cui la trama continua in maniera invariata a prescindere dalle azioni e dal successo del giocatore, rende poco interessante l'intera esperienza di gioco.

Tale esito non si identifica necessariamente con il concetto di vittoria, e nemmeno con quello di fine del gioco, ma può essere rappresentato da un diverso tipo di ricompensa per il raggiungimento di alcuni obbiettivi.

3. Il Game Design Document

Il game design document è un documento che segue l'evoluzione del gioco, a partire dalla sua fase embrionale di concept, fino alle fasi finali di sviluppo. Rappresenta un punto di riferimento per tutti i membri del team che lavorano sul videogame, permettendo la suddivisione dei compiti e la successiva integrazione dei lavori di ciascun componente.

Esso comprende una presentazione generale del gioco, e definisce tutti i presupposti che stanno alla base del design, tra cui il pubblico di destinazione, le piattaforme per cui si sceglie di sviluppare, i requisiti di sistema, e via dicendo. Entra poi nel dettaglio delle meccaniche di gioco, sia dal punto di vista più astratto del gameplay, sia più nello specifico spiegando i controlli che il giocatore userà.

La parte successiva del documento presenta una versione più o meno sintetica della storia che fa da premessa e da filo conduttore al gioco, e descrive l'eventuale mondo di gioco: in altre parole per i giochi più complessi dà le linee guida per chi si deve occupare di rappresentare o implementare le caratteristiche dell'ambiente in cui il gioco si svolge. Sono indicati anche gli eventuali media utilizzati (video, musiche, ecc.) che potrebbero essere protette da copyright.

Un'ultima parte si occupa dell'aspetto tecnico, entrando nel dettaglio delle tecnologie adottate per ogni aspetto della realizzazione del videogame, dagli strumenti usati nelle fasi iniziali di ideazione agli elementi maggiormente correlati con l'aspetto informatico.

Sono incluse nello stesso documento eventuali puntualizzazioni sugli aspetti legali concernenti il materiale usato e la possibilità di utilizzo del prodotto da parte altrui. Nel caso in esame non vi sono specifici aspetti legali degni di nota, poiché ogni contenuto è auto-prodotto oppure reperito da fonti gratuite e open-source.

Per comodità d'uso da parte del team di sviluppatori, è incluso anche uno storico delle modifiche apportate al Game Design nel corso dell'evoluzione del videogioco, a partire dalla semplice presentazione dell'idea di base, fino a giungere allo stato finale del prodotto.

In definitiva, il Game Design Document è l'equivalente del manuale di istruzioni che talvolta si trova incluso nelle confezioni dei videogame.

4. Dal concept al Game Design Document finale

L'idea iniziale era relativamente semplice: invertire la normale situazione, in cui il giocatore riveste il ruolo di un personaggio, e deve muoversi all'interno di un livello interamente controllato dall'intelligenza artificiale del gioco. Il giocatore avrebbe dunque rivestito i panni del livello, e i personaggi sarebbero stati tutti controllati da intelligenza artificiale. Alcuni di questi avrebbero dovuto essere ostacolati, altri aiutati, nel loro tentativo di raggiungere la fine dei diversi livelli.

Nella pagina seguente in fig.1 è riportato il concept per Run Level Run.

RUN LEVEL RUN

Introduction

Run Level Run is a platform game that sees the classic roles reversed, the “runner”, who is facing the path is the device on which you are playing, and the “level”, the path to face, the player. The goal is to help the runner to beat the level, or to prevent it from reaching the end if the trail.

Description

The game is a “revenge” against all those levels that in the past you could not overcome or that put you in doubt to the point of questioning your abilities.

Putting yourself in the shoes of the level to face, you can, using several commands, put the runner in trouble and this time it's not you, it's your device!

Slow down or speed up the runner at the right times, you can make him lose the time to take the leap that would enable him to escape, as well as to shake his world or flip it, you can upset him and make him miss the bonuses scattered accross the path.

But it will not be easy, especially if you choose instead of helping him!

Run Level Run offers three different game types. A typical campaign mode, in which you have to decide whether to help or hinder the runner, a challenge mode where you face a smarter runner and additional difficulties (such as the inability to use a certain type of command) and a third multiplayer mode where level and runner are both humans.

Key Features

- Increasingly difficult levels and increasingly skilled runners will test your skills
- Ability to edit and share the multiplayer levels

Genre

Platform side scrolling game

Platform

Though the chosen platform is PC, it is possible to commercialize the game on every available platform, with particular attention to new generation controllers such as Kinect, PSMove and WII controllers, and accelerometers in mobile devices.

Fig.1 – il Game Concept

A partire da questa semplice idea, si è lavorato inizialmente sugli *elementi drammatici* del gioco. Ci si è chiesto, cioè, come rendere più invitante il gioco a partire dal solo aspetto narrativo.

Il primo mezzo attraverso cui si è tentato di raggiungere tale obbiettivo è l'introduzione di **personaggi** con cui il giocatore potesse sviluppare un qualche tipo di legame, sia in senso positivo che in senso negativo. È così che è nato DUMB, l'incarnazione del "casual player", non molto abile ma benvoluto dai livelli di ogni videogioco: insomma, un personaggio con cui ogni videogiocatore si può identificare se gli è capitato di ritrovarsi ad affrontare una volta dopo l'altra lo stesso livello, diventando sempre più frustrato perché non riesce a superarlo. Dalla parte opposta c'è M.E.A.N., un'associazione di arroganti personaggi, abili nel superamento di ogni ostacolo, dediti a impedire ai personaggi come Dumb di riuscire laddove solo loro ritengono di avere il diritto di arrivare.

Oltre ai personaggi in sé, si è anche articolato il tutto in una **premessa** che desse un contesto agli elementi formali, coinvolgendo il giocatore fin dall'inizio. Tale premessa è in un certo senso una spiegazione in termini narrativi delle meccaniche di gioco che costituivano il concept. I livelli, stufi del predominio di M.E.A.N., decidono di prendere l'iniziativa e aiutare Dumb a superarli, finalmente, e al contempo fare il possibile per evitare che i membri di M.E.A.N. possano avere la meglio.

Con l'aggiunta di questi elementi, si cerca di attirare l'attenzione del giocatore proprio per la sua natura di videogamer. L'aspetto narrativo infatti, fa leva sulle esperienze di frustrazione verso il gioco stesso o altri giocatori che a molti capita di provare ripetutamente.

La **storia** occupa un ruolo di secondo piano, ed è spesso identificata con lo sviluppo stesso del gioco. Essa cambia in accordo con gli esiti raggiunti dal giocatore. Ad esempio a seconda della prestazione in un livello, l'atteggiamento e le capacità di Dumb e dei membri di M.E.A.N. potrebbero cambiare nella prosecuzione del gioco.

Nella pagina successiva in fig.2 è possibile vedere, come esempio, la prima pagina del Game Design Document finale.

RUN LEVEL RUN
Game Design Document

Index

1.Design History
2. Vision Statement
    2.1 Game Logline
    2.2 Gameplay Synopsis
3. Audience, platform and marketing
    3.1 Target audience
    3.2 Platforms
    3.3 System requirements
    3.4 Top performers
    3.5 Feature comparison
4. Legal analysis
5. Gameplay
    5.1 Overview
    5.2 Gameplay description
    5.3 Game elements
6. Controls
    6.1.1 Interface
    6.2 Rules
    6.3 Score
    6.4 Winning Condition
    6.5 Modes and Features
    6.6 Levels
7. Story
8. Game World
9. Media List
10. Technical Spec
    10.1 Technical Analysis
        10.1.1 New Technoligies
        10.1.2 Major Software Development Tasks
        10.1.3 Risks
        10.1.4 Estimated Resources Required
    10.2 Development Platforms and Tools (SW and HW)
    10.3 Delivery
    10.4 Game Engine
    10.5 Interface Technical Specs
    10.6 Controls' Technical Specs

1.Design History

- 23/01/2012 - Final revision of the document.
- 20/01/2012 - TS adjustments.

Fig.2 – estratto dal Game Design Document

5. Analisi degli elementi formali

5.1. Giocatore

Il giocatore è unico, non è prevista alcuna modalità multiplayer.

Il modello d'interazione del giocatore è una variazione rispetto al “single player versus game”. Con questa classificazione si intende, generalmente, quel tipo di giochi in cui un giocatore singolo affronta il videogioco, che svolge un ruolo da antagonista. In questo caso abbiamo però una ulteriore componente: il giocatore deve sì competere contro i personaggi di M.E.A.N., ma al tempo stesso vi è una componente di collaborazione con il gioco stesso. Infatti lo scopo principale è di aiutare Dumb ad arrivare al termine del livello, ed in alcune situazioni il giocatore si troverà a collaborare con il videogioco, piuttosto che competere contro di esso.

5.2. Obbiettivo

L'obbiettivo primario è aiutare Dumb ad attraversare il gioco fino ad arrivare al livello finale, facendo in modo che ne raggiunga la fine prima dei propri rivali.

Per fare questo vi sono degli obbiettivi parziali a ogni livello: fare in modo che Dumb raggiunga per primo la fine, evitare che venga catturato dai mostri o da M.E.A.N. o che questi ultimi arrivino al traguardo prima di lui.

È possibile completare il gioco anche senza raggiungere ogni obbiettivo parziale con successo, ma il raggiungimento di tutti gli obbiettivi permette di raggiungere un punteggio più elevato. Al contrario, fallire in alcuni obbiettivi parziali comporta un aumento nella difficoltà del livello successivo.

Per definirlo tramite gli obbiettivi più comuni presenti nei videogiochi, si tratta allo stesso tempo di una *gara* per il raggiungimento della fine del gioco, e di una *fuga* dai personaggi antagonisti.

5.3. Procedure

L'ambiente per cui Run Level Run è pensato è il personal computer, con l'utilizzo del solo mouse per tutte le azioni che è possibile effettuare. Per la natura di tali azioni, sarebbe anche immediato il passaggio a dispositivi mobili dotati di touchscreen, sostituendo le operazioni effettuate con il mouse con le corrispondenti azioni fatte con il semplice tocco di un dito.

Nelle schermate iniziali e nelle schermate intermedie tra i livelli, l'interazione è molto immediata ed intuitiva, l'unica procedura è:

- Click: selezione di un'opzione o sotto-opzione dei menu

Nel corso dei livelli, le operazioni sono facilmente riassumibili in due categorie generali di

procedure:

- Click: attivazione [o disattivazione] di molti tipi di risorsa
- Click ripetuto: taglio/rottura di alcuni oggetti
- Click & Drag: manipolazione delle piattaforme rotanti, spostamento di oggetti mobili

Si tratta di modelli di interazione molto comuni e facilmente apprendibili. È presente a inizio gioco una serie di istruzioni sulle procedure da usare con i diversi tipi di oggetti controllabili dal giocatore. Assieme alla spiegazione testuale dell'operazione, sono presenti alcune immagini estratte da situazioni di gioco che la illustrano graficamente. Le operazioni associate a ogni tipo di risorsa sono le più intuitive possibili, rendendo così semplice il gioco anche senza la precedente consultazione delle istruzioni.

5.4. Regole

5.4.1. Regole che definiscono oggetti e concetti

Sono presenti diversi tipi di personaggio all'interno dei livelli. Ognuno di essi ha delle statistiche associate, che determinano il modo in cui il personaggio si muove: velocità, capacità di saltare, area di percezione, numero di vite, e via dicendo. Alcune di queste caratteristiche si applicano a tutti i personaggi, altre soltanto a una parte di essi.

Sono anche presenti dei valori globali, che definiscono la velocità globale e la gravità del livello. Queste possono entrambe essere modificate dall'effetto di particolari poteri.

5.4.2. Regole che restringono le azioni possibili

Non esistono vere e proprie regole che rientrino in questa categoria.

5.4.3. Regole che determinano degli effetti

Si tratta di un insieme di regole, implicito nel gioco, che non vengono esplicitamente spiegate al giocatore. Esse comprendono:

- le regole costituenti la semplice intelligenza artificiale che muove Dumb, ad esempio fermarsi quando viene attivata una piattaforma mobile, e ricominciare a muoversi quando essa viene riattivata
- le regole dell'intelligenza artificiale dei mostri, più complessi di Dumb, in base alle quali essi reagiscono alla presenza di personaggi circostanti
- le regole dell'intelligenza artificiale di M.E.A.N., che permettono a questi personaggi di seguire il percorso per completare il livello, fuggire dai mostri, inseguire Dumb, evitare ostacoli, ecc.
- le regole legate ai power-up (o power-down), che alterano i valori delle variabili

globali o riferite a un personaggio (vedi 4.4.1.)

- le regole dei poteri speciali, che modificano i valori delle variabili, o producono effetti sull'intero livello, o ancora alterano il modo in cui il giocatore vede il gioco sul proprio schermo

5.5. Risorse

Rispetto al documento di design, la demo sviluppata comprende solo una parte delle risorse, quindi ci si limiterà ad analizzare quelle effettivamente presenti.

Innanzitutto possiamo elencare le *vite* a disposizione di Dumb e la sua *salute*. Le prime rappresentano il numero di volte che può morire senza fallire il livello, o in altre parole le possibilità che ha il giocatore di affrontare nuovamente lo stesso schema. La salute, invece, rappresenta la resistenza di Dumb agli ostacoli che può incontrare lungo il cammino: la maggior parte di questi non determina immediatamente la sua morte, bensì riduce la vita di una certa quantità.

Il *tempo* è altrettanto importante, poiché il livello (e dunque il giocatore) ha a disposizione un tempo limitato per portare Dumb a raggiungere il traguardo posto alla fine.

Possono essere classificati tra le risorse, in un certo senso, anche i mostri, non veri e propri personaggi, ma nemmeno oggetti come gli altri, che il giocatore deve evitare e sapere usare a proprio vantaggio.

Altro tipo di risorsa sono i *poteri speciali*, più nel dettaglio:

- Flip: potere “negativo”, inverte la visione del livello, cioè il gioco è visualizzato sullo schermo al contrario
- Blur: potere “negativo”, rende sfocata e pertanto difficile la visione del livello sullo schermo
- Terremoto: scuote l'intero livello, causando uno spostamento dei personaggi e di tutti gli oggetti mobili
- Teletrasporto: permette di spostarsi velocemente tra due parti distanti del livello, oppure di accedere a punti del livello non raggiungibili altrimenti (dove spesso si può trovare dei power-up).

Vi sono poi i power-up:

- Cuore: ripristina il livello di salute di Dumb
- Medikit: aggiunge una vita a Dumb

Lungo il percorso sono presenti alcuni ostacoli, più specificamente delle bombe, attivate

quando un personaggio vi entra in contatto. Queste causano a M.E.A.N. una diminuzione del livello di salute.

Vi è un numero di piattaforme di diverso tipo, che costituiscono il nucleo fondamentale del game-play:

- Piattaforme rotanti: vengono controllate con il Click & Drag e possono essere usate per indirizzare i personaggi nella direzione desiderata tra varie possibili
- Piattaforme a due posizioni: simili alle piattaforme rotanti, e attivate attraverso un semplice clic, possono essere alzate o abbassate per indirizzare i personaggi su una di due direzioni, o anche solo per bloccare loro la strada
- Piattaforme sganciabili: cliccando su di esse, si staccano dal punto a cui sono ancorate e cadono, si può così fare in modo che ai personaggi manchi il terreno sotto ai piedi
- Piattaforme mobili: si attivano e disattivano con un clic, e si spostano tra due punti del livello, spesso accorciando la strada o permettendo l'accesso a zone non raggiungibili in altro modo; l'unico aspetto di vera “intelligenza” di Dumb è fermarsi quando una o più di queste piattaforme sono attive

Ultima risorsa è il *punteggio*. I punti sono guadagnati in base al tempo e alla vita e salute rimanenti. I punti accumulati possono essere usati nel prosieguo del gioco per l'acquisto di poteri e oggetti, utilizzabili nei livelli seguenti per rendere più semplice affrontarli.

6. Glossario di Unity

Si darà in questo capitolo una breve spiegazione dei termini riferiti a Unity necessari ad avere una più chiara comprensione dell'implementazione.

Innanzitutto occorre sapere che Unity è un game engine 3D, che dà la possibilità di creare delle *scene* in cui inserire una serie di *gameobject* a cui è poi possibile assegnare componenti di vario tipo, tra cui gli script che hanno la possibilità di accedere alle diverse variabili collegate al gameobject.

6.1. Gameobject

Un gameobject è un oggetto presente sulla scena, che ha dei valori che ne definiscono posizione, rotazione e scalamento. Può inoltre avere un numero variabile di componenti e fare parte di una gerarchia di gameobject, avendo quindi 0 o 1 nodo genitore, e da 0 a un numero indefinito di figli. La componente più comune per un gameobject è la mesh, in un ambiente 3D.

6.2. Mesh

Una mesh è la rappresentazione di un oggetto 3D come insieme di poligoni che hanno una specifica posizione nello spazio l'uno rispetto all'altro. Una mesh può essere "riggata", cioè fornita di una sorta di scheletro virtuale che viene poi sfruttato per creare delle animazioni, oppure si possono creare delle animazioni con altri metodi, rappresentando quindi il movimento di oggetti nello spazio. A una mesh può anche essere assegnata una o più texture e uno o più materiali.

6.3. Animazioni

Le animazioni sono realizzabili sia con strumenti esterni, sia direttamente dal codice. Nel primo caso esse saranno una componente associata al gameobject e gestibili attraverso apposite funzioni fornite dalle librerie di Unity. Nel secondo caso invece ogni movimento dell'animazione viene determinato tramite il codice che sarà associato direttamente o indirettamente al gameobject.

6.4. Texture

Una texture è un'immagine associata a una mesh, che permette di sovrapporre a un modello tridimensionale "piatto" un'immagine. L'associazione tra i punti della mesh e quelli dell'immagine avviene tramite la cosiddetta UV map, che può essere calcolata da tool appositi oppure adattata manualmente per facilitare il compito di chi si occupa della grafica.

6.5. Materiale

Il materiale di una mesh influenza il modo in cui essa è visualizzata. Ogni materiale ha un suo diverso modo di reagire alla luce, dunque a seconda del materiale applicato l'aspetto di una mesh appare diverso.

6.6. Motore fisico

Uno degli strumenti messi a disposizione da Unity è il motore fisico che può essere usato direttamente per eseguire molte operazioni senza la necessità di implementare una soluzione ad hoc. Il motore è personalizzabile, inserendo i valori desiderati per la gravità, una soglia per dare dei valori minimi alle velocità dopo cui un corpo si ferma, smette di rimbalzare, e via dicendo. È possibile assegnare un *rigidbody* a un corpo ed eseguirvi operazioni con diverse grandezze fisiche, oltre a leggerne semplicemente il valore dagli attributi.

6.7. Rigidbody

È un componente, applicabile ai gameobject, che permette di sfruttare le funzionalità del motore fisico. È possibile modificare o leggere il valore di grandezze come la velocità, l'attrito, le forze presenti, applicare forze di diverso tipo e in generale attivare o disattivare l'effetto delle varie grandezze fisiche sull'oggetto.

Un rigidbody può essere cinematico (kinematic): se tale proprietà è attiva, le forze, le collisioni, e i vincoli non hanno più alcun effetto sul rigidbody in esame, pertanto questo è completamente controllato da script e animazioni. Esso continua comunque a influire sul movimento degli altri rigidbody, non cinematici, attraverso collisioni e vincoli.

6.8. Collider e trigger

I collider sono componenti che consentono agli oggetti solidi di interagire tra di loro invece di attraversarsi semplicemente. Un collider può avere forme diverse, anche quelle di mesh complesse, ma normalmente si preferisce anche per i personaggi collider più semplici, per avere prestazioni maggiori. Un collider può anche essere contrassegnato come trigget: la differenza è che il trigger non è causa di effetti fisici quando due oggetti entrano in contatto, mentre il collider normale evita che due oggetti solidi si compenetrino tra sé.

6.9. Rilevamento delle collisioni

Unity tiene sotto controllo le interazioni tra collider di vario tipo, secondo regole precise a seconda del tipo di gameobject. Una serie di funzioni serve a reagire ai vari eventi possibili: OnCollisionEnter e OnTriggerEnter servono a rilevare l'inizio di un contatto con il collider o trigger qui il codice si riferisce, OnCollisionExit e OnTriggerExit ne rilevano la fine. Inoltre ci sono due ultime funzioni, attivate quando il contatto era già presente e continua a esserlo, OnCollisionStay e OnTriggerStay.

6.10. Emettitore di particelle

Un emettitore di particelle è un gameobject che funge da origine per un sistema di particelle. Quest'ultimo è in parole povere la simulazione di un fenomeno fisico, ad esempio fuoco, fumo, acqua, ecc., attraverso l'uso di piccoli oggetti, le particelle, che vengono riutilizzati molte volte per creare l'effetto finale, con il vantaggio di non richiedere un'eccessivo carico computazionale.

6.11. PlayerPrefs

Serve a memorizzare e accedere alle preferenze del giocatore tra le diverse sessioni in cui il gioco viene eseguito. Salva i dati in percorsi diversi a seconda del supporto sul quale si sta giocando.

6.12. Script

Un tipo particolare di componente per i gameobject è lo script. Esso può essere in tre diversi linguaggi, di cui i più noti JavaScript e C#. Nel corso di questo progetto è stato usato prevalentemente JavaScript, anche se alcuni elementi, nel dettaglio le bombe, sono parzialmente realizzati con C# laddove si è presa ispirazione per il codice a qualcosa di già esistente.

Gli script sono in una versione particolare dei linguaggi citati, comprensiva di librerie specifiche per l'uso su Unity. Essi possono interagire con altri script se ne hanno un riferimento a disposizione. Il gameobject su cui è applicato lo script può essere facilmente acceduto, infatti scrivere transform è sufficiente a ottenere un riferimento all'oggetto.

Negli script sono poi presenti alcune funzioni, di cui l'utilizzo dettagliato è spiegato nel seguito, ereditate dalla classe "genitore" di ogni script, MonoBehaviour. Queste funzioni sono chiamate in alcuni particolari momenti per tutti gli script, ad esempio la funzione Update avviene a ogni aggiornamento del frame, pertanto se in uno script è presente l'override di Update, l'algoritmo in essa contenuto verrà eseguito a ogni aggiornamento del frame.

Nel seguito sono elencate le funzioni ereditate da MonoBehaviour più utilizzate nel progetto, e maggiormente menzionate in questo elaborato.

6.12.1. Awake

Awake viene chiamata quando l'istanza dello script è caricata. Questo significa che viene sempre eseguita all'avvio del gioco, subito prima che questo inizi. È invocata solo una volta nel corso della vita dell'istanza, dopo che gli altri oggetti sono stati caricati: in questo modo è possibile usare riferimenti ai gameobject presenti sulla scena già nel corso dell'esecuzione della funzione.

6.12.2. Start

Start è invocata una volta sola nel corso di vita dell'istanza dello script, appena prima della prima esecuzione di Update dello stesso script, e subito dopo che ogni Awake è stata eseguita. La differenza rispetto ad Awake è che Start viene eseguita soltanto se l'istanza dello script è attivata. Questo consente di ritardare l'inizializzazione del codice quando questa non è richiesta fin da subito, ottenendo così tempi di avvio del gioco minori, e potendo ordinare l'inizializzazione degli script secondo le proprie necessità.

6.12.3. Update

È chiamata a ogni nuovo frame, rappresentando così il modo più semplice di gestire il cambiamento di stato di ogni elemento del gioco al passare del tempo.

6.12.4. LateUpdate

È eseguita dopo la terminazione dell'esecuzione di tutte le funzioni Update. La sua utilità è la possibilità di imporre un ordine nell'esecuzione degli script. Nel progetto, l'esempio migliore di questo si ha nello script che controlla la telecamera. Infatti il movimento di Dumb, ovvero l'oggetto che viene seguito dalla telecamera, viene aggiornato nel ciclo Update: questo significa che se la telecamera fosse aggiornata anch'essa nell'Update, poiché l'esecuzione dei vari script avviene in ordine casuale, non ci sarebbe nessuna garanzia di fare riferimento al valore già aggiornato.

6.12.5. Eventi del mouse

Nel dettaglio si tratta di OnMouseOver, OnMouseExit, OnMouseDown e OnMouseUp. Rispettivamente sono eventi eseguiti quando il mouse si sposta sopra un elemento della GUI o un collider, oppure se ne sposta al di fuori, è premuto sull'elemento o collider, o viene rilasciato. Questi eventi sono usati soprattutto nei menu iniziale e intermedi, e nell'interazione con le piattaforme.

6.12.6. Rilevazione di collisioni – Trigger

Le funzioni OnTriggerEnter, OnTriggerExit e OnTriggerStay sono chiamate ogni volta che un Collider rispettivamente entra nel trigger, ne esce, o lo sta già toccando nel frame attuale. Gli eventi trigger sono inviati solamente se uno dei due collider coinvolti dal contatto ha un rigidbody associato.

6.12.7. Rilevazione di collisioni – Collision

Le funzioni sono stavolta OnColliderEnter, OnColliderExit e OnColliderStay. Ci sono alcune differenze rispetto al caso dei trigger. Innanzitutto, invece di un collider come nelle funzioni relative ai trigger, viene passata alla funzione un'istanza della classe Collision, che

contiene informazioni a proposito dei punti di contatto e della velocità d'impatto oltre ai riferimenti all'oggetto colpito. Il contatto è rilevato soltanto quando almeno uno dei due collider è associato a un rigidbody non cinematico.

Collision detection occurs and messages are sent upon collision						
	Static Collider	Rigidbody Collider	Kinematic Rigidbody Collider	Static Trigger Collider	Rigidbody Trigger Collider	Kinematic Rigidbody Trigger Collider
Static Collider		Y				
Rigidbody Collider	Y	Y	Y			
Kinematic Rigidbody Collider		Y				
Static Trigger Collider						
Rigidbody Trigger Collider						
Kinematic Rigidbody Trigger Collider						
Trigger messages are sent upon collision						
	Static Collider	Rigidbody Collider	Kinematic Rigidbody Collider	Static Trigger Collider	Rigidbody Trigger Collider	Kinematic Rigidbody Trigger Collider
Static Collider					Y	Y
Rigidbody Collider				Y	Y	Y
Kinematic Rigidbody Collider				Y	Y	Y
Static Trigger Collider		Y	Y		Y	Y
Rigidbody Trigger Collider	Y	Y	Y	Y	Y	Y
Kinematic Rigidbody Trigger Collider	Y	Y	Y	Y	Y	Y

Tab.1 – interazione di collider e trigger

6.12.8. Linecast

Linecast è un metodo di Physics, nucleo del motore fisico di Unity, con cui è possibile verificare la presenza di collider lungo il percorso tra due punti, specificati come argomenti in input. Ritorna true in caso questo avvenga, e rende accessibili ulteriori informazioni sull'oggetto trovato in una struttura RaycastHit.

Questa funzione, assieme all'uso dei trigger, è alla base delle intelligenze artificiali dei personaggi, poiché permette di rilevare la situazione del percorso tra la posizione attuale e quella da raggiungere, che si tratti di un altro personaggio o di un checkpoint.

6.12.9. RaycastHit

RaycastHit è una struttura che memorizza i dati dell'oggetto colpito dal Linecast (o da un

raycast, che però non è altrettanto importante nell'ambito dell'implementazione analizzata). Oltre a una serie di riferimenti all'oggetto colpito (collider, rigidbody, transform, ecc.) è possibile avere il punto esatto nello spazio in cui avviene l'incrocio, la normale al punto colpito, la distanza tra esso e l'origine, e altri dati ancora.

7. Implementazione

Lo strumento usato per lo sviluppo del gioco è Unity, un game engine multipiattaforma che mette a disposizione alcuni utili strumenti, che non è stato dunque necessario implementare in prima persona. In particolare, oltre al motore grafico, è stato usato il motore fisico di Unity ed il suo sistema di rilevamento delle collisioni tra oggetti.

Il gioco è stato sviluppato in JavaScript, impiegando le librerie specifiche di Unity.

7.1. Schermata iniziale

La schermata iniziale è una semplice interfaccia grafica attivabile con il mouse. È possibile visualizzare delle semplici istruzioni sui controlli del gioco, per i quali non è possibile alcun tipo di interazione. Si può inoltre, naturalmente, avviare il gioco, scegliendo tra tre diverse difficoltà.

Con la difficoltà inferiore, l'unico obbiettivo del giocatore è di fare in modo che Dumb raggiunga la fine del livello. All'aumentare della difficoltà entrano in gioco anche i mostri e M.E.A.N.

7.2. Stato del gioco

All'avvio del gioco, lo stato viene inizializzato in base alla difficoltà selezionata: due variabili memorizzano se mostri e M.E.A.N. siano attivi o meno.

Viene poi memorizzata globalmente la velocità dell'intero gioco, e il numero di vite rimanenti al giocatore per portare Dumb alla fine del livello.

Un'ultima variabile che viene mantenuta durante tutta la durata del gioco è il punteggio, aggiornato di volta in volta in base al raggiungimento o al fallimento degli obbiettivi parziali.

Questo stato è aggiornato solo da eventi specifici, e i dati vengono altrimenti mantenuti costanti, ogni volta che viene caricato un livello successivo o una schermata intermedia.

7.3. Personaggi - Dumb

7.3.1. Dumb – GameObject

L'oggetto che rappresenta Dumb sulla scena del gioco è semplice. Ha una mesh che serve a visualizzarlo sullo schermo, assieme al materiale e alla texture, e delle animazioni la cui riproduzione può venire attivata con un'invocazione da uno script. È associato a un rigidbody (cioè sfrutta il motore fisico di Unity, attraverso il quale può essere manipolato). Ha inoltre associato un collider, che permette di identificare le collisioni di Dumb con altri oggetti dotati della stessa funzionalità.

Al GameObject è associato uno script che permette gli spostamenti del personaggio.

7.3.2. Dumb – Movimenti

A inizio livello, ed ogni volta che muore e deve riprendere dall'inizio, il personaggio viene posizionato in un punto preciso del livello, definito da un GameObject: chiamiamo questo punto di respawn.

Il movimento di Dumb avviene in un'unica direzione lungo l'asse orizzontale dello schermo, e segue l'andamento del terreno. Non è possibile saltare, accelerare, o in altro modo alterare direttamente il comportamento del personaggio.

Lo spostamento avviene impostando a un valore, parti a quello definito come velocità di Dumb, all'intero oggetto, per questo motivo esso è in grado di camminare anche sulle piattaforme disposte obliquamente. Raggiunto un certo grado di pendenza, non gli è più possibile proseguire poiché la velocità è contrastata dalla forza di gravità in direzione verticale, e dalla forza di attrito in direzione orizzontale.

Lo script che muove Dumb ha poi un controllo sulle piattaforme mobili: se una di esse risulta attivata, la velocità viene impostata a zero, interrompendo il movimento. Una volta che non è più presente alcuna piattaforma attiva, il personaggio è di nuovo libero di muoversi.

Una funzione *ApplyDamage* è presente, e può essere invocata da tutti gli oggetti in grado di causare danni (o anche, al contrario, di ripristinare la salute). La vita di Dumb è ridotta del quantitativo di danni associato all'oggetto che invoca la funzione, o aumentato del numero di punti di salute che è possibile recuperare grazie all'oggetto. Dopodiché si verifica che il valore non sia inferiore allo 0 né maggiore di 6, poiché tali valori determinerebbero errori, sia nella gestione dello stato della salute, sia nella visualizzazione della salute sullo schermo.

Se la salute è pari a zero, il numero di vite viene ridotto di uno. Se dopo questo aggiornamento le vite restanti sono inferiori a zero, viene caricata la schermata di Game Over. Altrimenti, avviene il respawn all'inizio del livello, resettando tutti gli oggetti in esso contenuti.

Un'altra funzione invocabile da oggetti esterni è *Teletrasporto*, chiamata dai gameobject relativi al potere del teletrasporto. Questa imposta a zero le velocità orizzontale e verticale di Dumb, e ne modifica la posizione nello spazio, portandolo al punto predeterminato.

Nella fig.3 è possibile vedere un grafo che visualizza il semplice automa a stati finiti corrispondente a quanto appena descritto.

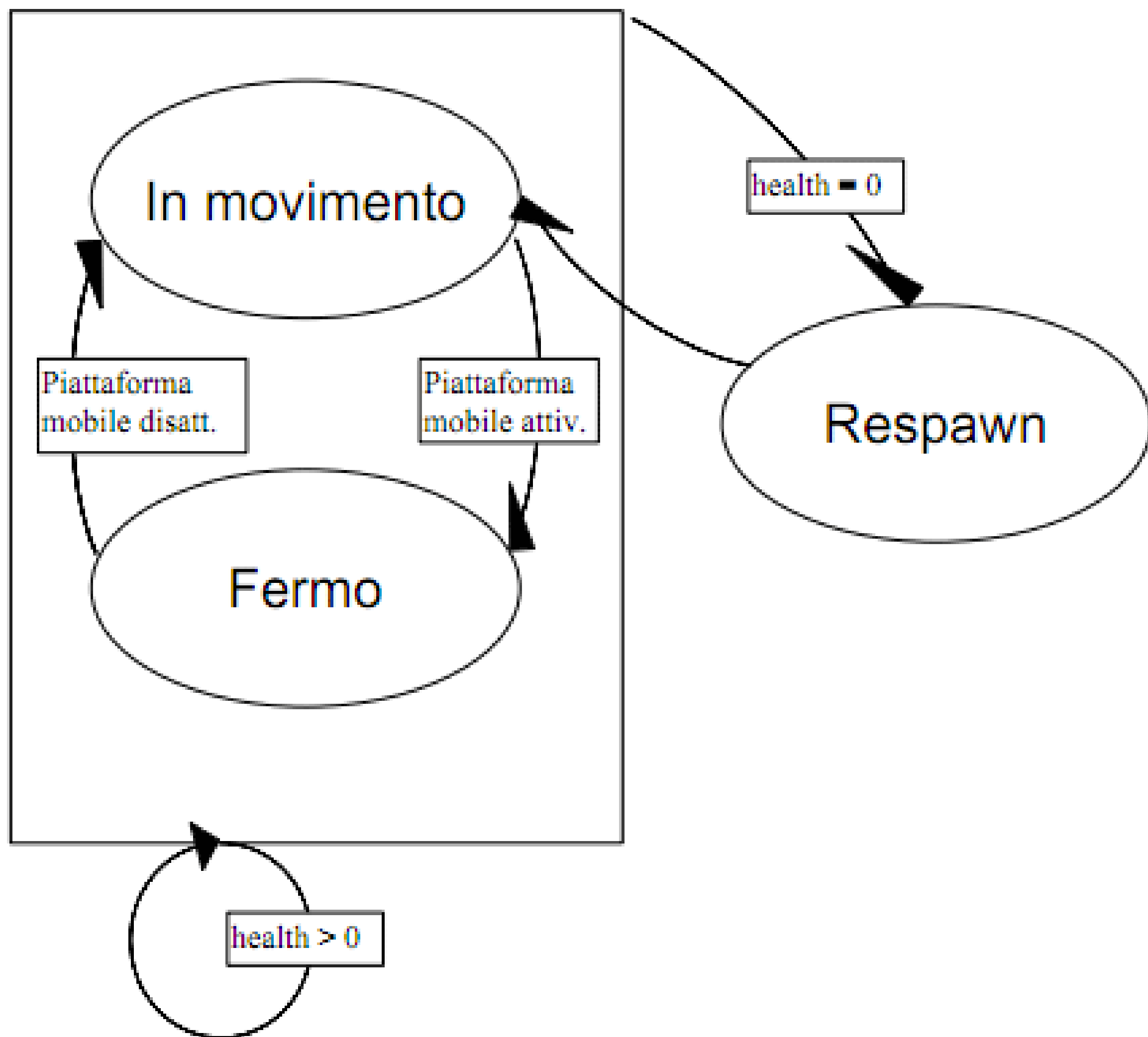

Fig.3 – Automa a stati finiti che rappresenta approssimativamente l'intelligenza artificiale di Dumb

7.4. Personaggi - Mostri

7.4.1. Mostri – GameObject

L'oggetto che rappresenta i mostri ha un grado di sofisticazione maggiore: mesh, materiale e texture lo rendono visibile, ed è un rigidbody, quindi come Dumb usa il motore fisico di Unity. Ha un collider applicato alla mesh che consente di usare gli algoritmi di determinazione delle collisioni.

In aggiunta a questo, tramite un gameobject vuoto inserito nella gerarchia, dispone di un altro collider, di dimensioni maggiori, che viene utilizzato per rappresentare l'area di percezione del mostro. Questo collider è un trigger, cioé il suo contatto con altri oggetti dotati di collider non causa effetti fisici, bensì solamente il lancio di un evento correlato.

I mostri hanno poi tre script associati: uno è applicato alla SensingArea (il trigger menzionato sopra) e si occupa di simulare la percezione visiva/uditiva del personaggio, gli altri due sono associati direttamente al gameobject del mostro e si occupano rispettivamente di gestirne il semplice automa a stati finiti di alto livello che costituisce l'intelligenza artificiale e di governarne i movimenti. Alcune di queste competenze sono parzialmente sovrapposte, ad esempio è lo script relativo alla percezione a dare origine al cambiamento di stato, sebbene poi sia lo script della IA ad occuparsene.

7.4.2. Mostri – Percezione

Lo script che si occupa della percezione di altri personaggi sfrutta il rilevamento delle collisioni tra il trigger e i collider di altri oggetti. Il numero di contatti con altri personaggi è rappresentato da una variabile inizializzata a 0: ogniqualvolta un personaggio entra o esce dal trigger il contatore è rispettivamente aumentato o decrementato.

Se il mostro stava semplicemente aggirandosi per il livello, inizierà ad inseguire il personaggio, se invece stava già inseguendo un altro personaggio confronterà la distanza dei due e selezionerà il più vicino come obbiettivo da inseguire.

Quando il numero dei contatti torna a zero, l'inseguimento viene interrotto e il mostro riprende a muoversi casualmente nel livello.

7.4.3. Il tag “DeadEnd”

Gli oggetti che vengono classificati come *DeadEnd* sono usati dai mostri e da M.E.A.N. come punti di riferimento. Vengono inseriti manualmente dal designer del livello per identificare i percorsi non convenienti ai personaggi guidati da intelligenza artificiale più avanzata di quella di Dumb. Più nello specifico vengono usati per indicare i punti da non superare perché il personaggio non muoia oppure si trovi bloccato in un punto da cui non ha modo di uscire.

Qualsiasi oggetto che non abbia un altro tag può essere indicato come DeadEnd, la soluzione più semplice è usare come DeadEnd oggetti invisibili privi di rigidbody.

7.4.4. Mostri – Il nucleo dell'IA

In questo script è salvato l'eventuale personaggio attualmente inseguito dal mostro, e lo stato corrente del livello più alto dell'automa a stati finiti che governa l'intelligenza artificiale. Questo può avere due stati: *roam* e *chase*. Il cambiamento di stato è invocato dallo script relativo alla percezione, poiché è quello a tenere traccia dei personaggi percepiti. Se lo stato è chase, la direzione del movimento è di volta in volta aggiornata calcolando la posizione del personaggio inseguito rispetto a quella del mostro, altrimenti se lo stato è roam viene invocata una funzione dell'ultimo script, quello responsabile del movimento, a cui è dato il compito di rendere casuali gli spostamenti.

Inoltre se lo stato è chase si verifica di volta in volta che il percorso dal mostro all'obbiettivo dell'inseguimento non passi attraverso nessun oggetto DeadEnd. La verifica avviene attraverso un linecast: Unity traccia una linea da un gameobject in direzione dell'altro, e restituisce l'oggetto incontrato. Se si riscontra la presenza di un DeadEnd, lo stato è modificato in roam.

Lo script contiene anche un metodo per disattivare il mostro quando appropriato. La funzione attraversa l'albero dei figli del gameobject rappresentante il mostro e li disattiva tutti.

7.4.5. Mostri – Movimento

Il primo ruolo di questo script è di aggiornare di frame in frame l'orientamento del personaggio in base alla direzione in cui si sta muovendo e di selezionare l'animazione corretta da riprodurre in base alla posizione corrente relativamente alle piattaforme.

Viene poi tenuta traccia delle collisioni con piattaforme e DeadEnd. La variabile aggiornata in base alle prime serve a definire di volta in volta se il mostro si trova in aria oppure in piedi. Le seconde servono invece a prevenire che il mostro salti al di fuori del livello o finisca in una zona da cui non ha la possibilità di uscire.

Le collisioni con altri personaggi sono tenute sotto controllo, e nel caso avvengano è invocata la funzione associata al personaggio toccato che ne causa la morte.

Se lo stato corrente è roam, prima di effettuare il movimento, c'è una piccola probabilità di effettuare un cambiamento di direzione, altrimenti la direzione non cambia (è già stata calcolata dalla IA). Il movimento viene implementato applicando una velocità nella direzione desiderata, grazie al motore fisico.

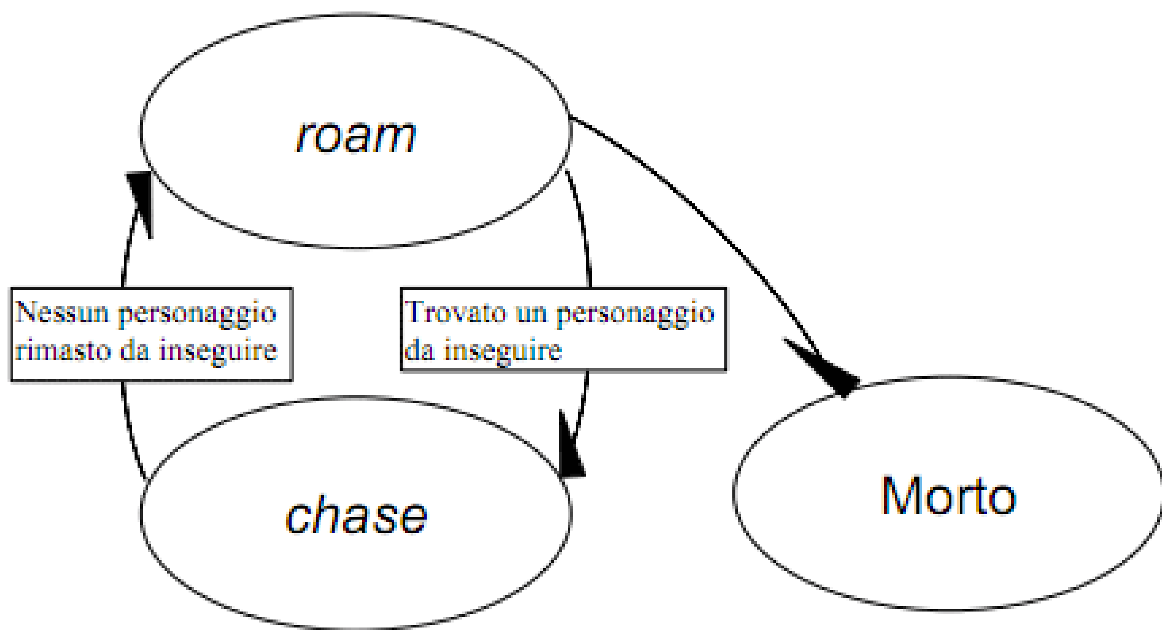

Fig.4 – Automa a stati finiti che rappresenta approssimativamente l'intelligenza artificiale di un mostro

Cod.1: Il codice del nucleo della IA del mostro

```
var movementScript : MonsterMove;
static var state;
static var currentlyChasing : GameObject; // Il
gameObject che il mostro sta inseguendo

function Start () {
Carica lo script dei movimenti per accedere alle sue
funzioni
	state = "roam";
}

function Update () {
	if (state=="roam") {
		movementScript.walk();
	}
	if (state=="chase") { // Se il personaggio sta
inseguendo un nemico si muove nella sua direzione


movementScript.changeDirection(Mathf.Sign(transform.posi
tion.x-currentlyChasing.transform.position.x));
		movementScript.walk2();
		if (!noWalls(currentlyChasing)) stopChasing();
// Se è presente un muro tra il mostro e l'inseguito,
l'inseguimento termina
	}
}

// Metodo per fare morire il personaggio
function Die () {
		var B=transform;
		if (B!=null) {
			for (var i=0;i<B.GetChildCount();i++){
				B.GetChild(i).active=false;
			}
			B.active=false;
		}
}

// Interrompe l'inseguimento
function stopChasing(){
	state="roam";
}
```

```
// Verifica se non c'è nessun muro (tag "DeadEnd") tra
il mostro e il gameObject inseguito
function noWalls(chased : GameObject){
     var hitvalue : RaycastHit;

Physics.Linecast(transform.position,chased.transform.pos
ition,hitvalue);
     if (hitvalue.collider!=null &&
(hitvalue.collider.gameObject.tag=="DeadEnd")) return
false;
     else return true;
}

function ToggleVisibility() {
    // toggles the visibility of this gameobject and all
its children
    var renderers =
gameObject.GetComponentsInChildren(GameObject);
    for (var r : Renderer in renderers) {
        r.enabled = !r.enabled;
    }
}
```

7.5. Personaggi – M.E.A.N.

7.5.1. M.E.A.N. - GameObject

Il gameobject associato a M.E.A.N. è essenzialmente simile a quello dei mostri. Esso comprende mesh, materiale e texture per visualizzare il personaggio, un collider per il rilevamento di collisioni del personaggio con altri oggetti, e un trigger associato alla SensingArea. Il trigger per la percezione ha in questo caso un uso più estensivo: l'intelligenza artificiale di M.E.A.N. è infatti più complessa e fa grande affidamento sull'abilità del personaggio di "vedere" alcuni punti di riferimento presenti nel livello, anche se invisibili.

Anche in questo caso vi sono tre script: uno per la percezione di altri personaggi ed elementi ausiliari alla IA, uno per la gestione degli stati della IA, ed uno per il movimento.

7.5.2. I checkpoint

I checkpoint, o CP, sono dei punti di riferimento usati dall'intelligenza artificiale di M.E.A.N. per permettergli di trovare la strada verso la fine del livello. L'idea di base è che se non si trova ad inseguire Dumb per catturarlo, o a fuggire dai mostri presenti nel livello, M.E.A.N. cercherà di raggiungere il traguardo prima che il proprio avversario riesca a farlo, aiutato dal giocatore.

L'intento era imitare da vicino il possibile stile di un giocatore umano, invece di puntare al percorso ottimale, sia che si trattasse di precalcolarlo o inserirlo manualmente, sia che si trattasse di utilizzare un algoritmo che prevedesse l'esito delle varie combinazioni di azioni possibili. Occupandosi del design del livello si può dunque includere, oltre ai DeadEnd, anche i checkpoint che, inseriti in punti critici del percorso, guidano M.E.A.N. nella direzione corretta. Quando non sta inseguendo Dumb o scappando da un mostro, questi si muoverà casualmente sul livello fino a quando percepirà un checkpoint. A quel punto inizierà a muoversi in direzione di esso. Una volta passati, i checkpoint forniscono informazioni sulla direzione successiva in cui muoversi. Questo nella demo rappresenta la possibilità di proseguire nella stessa direzione oppure quella di tornare nella direzione di provenienza. È anche prevista la possibilità di far sì che M.E.A.N. si fermi fino a quando riceve un'indicazione contraria, e quella di dargli l'ordine di arrampicarsi.

La realizzazione dei checkpoint li rende in realtà adatti anche all'utilizzo in un contesto 3D. Sono rappresentati, seppure invisibili nel gioco, da frecce orientate in un verso preciso: nel 2D questo si traduce in destra o sinistra, in un ambiente tridimensionale sarebbe possibile indirizzare il personaggio in qualsiasi direzione in base alla rotazione del checkpoint rispetto all'asse verticale (direzione orizzontale). Questo si traduce nella possibilità di guidare il personaggio indicandogli qualsiasi direzione nel raggio di 360°, senza richiedere di memorizzare alcun tipo di dato ulteriore in una variabile. La decodifica del significato dell'orientamento di un CP è gestita dagli script, e la direzione è impostata semplicemente uguagliando la direzione in cui punta il checkpoint.

7.5.3. M.E.A.N. - Percezione

Lo script tiene traccia della percezione di oggetti di diverso tipo tramite il trigger e applica la reazione adeguata, direttamente oppure invocando una funzione dallo script dell'intelligenza artificiale.

In caso di checkpoint individuato, è invocata la funzione *foundCP* dello script dedicato alla IA. Se a essere individuato è un altro personaggio, viene invocata la funzione *chaseEnemy* nel caso si tratti di Dumb, mentre è chiamata la funzione *escapeFrom* se si tratta di un mostro.

Nel caso che una bomba, o un altro oggetto pericoloso, si trovi nel campo di percezione, si verifica se M.E.A.N. si sta muovendo in direzione di essa e se una collisione tra i due oggetti è possibile

La prima verifica è immediata, basta confrontare la direzione di spostamento, che può avere valore +1 o -1, con il segno della differenza tra la posizione della bomba e quella di M.E.A.N.: se i due valori coincidono lo spostamento è in direzione dell'oggetto.

Il controllo sulla collisione possibile avviene in base alla posizione verticale dei due oggetti, ed è anch'esso piuttosto semplice. Si calcola:

- differenza tra il valore di y del punto più in basso della bomba e di quello più in alto di M.E.A.N.
- differenza tra il valore di y del punto più in alto della bomba e di quello più in basso di M.E.A.N.

Se questi due valori sono diversi, ciò significa che le due posizioni verticali sono almeno in parte coincidenti, in caso contrario non c'è rischio immediato di collisione, e si può quindi ignorare l'oggetto pericoloso. In fig.5 sono illustrati i vari casi possibili.

Se il rischio di collisione esiste, è calcolato un punto situato al di sopra della bomba, e viene invocato il metodo *jumpTo* dello script relativo al movimento, usando tale punto come obbiettivo del salto.

Viene tenuta traccia anche degli oggetti che escono dall'area di percezione. In base a questi eventi si esce dagli stati *chase* ed *escape* per tornare allo stato *roam* (vedi paragrafo successivo).

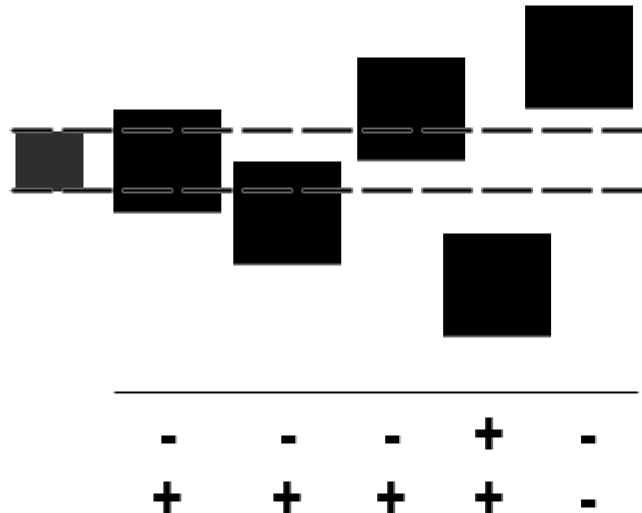

Fig.5 – varie possibili posizioni relative di bomba e M.E.A.N.

7.5.4. M.E.A.N. - Il nucleo della IA

Lo script comprende diverse variabili:

- state: lo stato dell'automa a stati finiti che costituisce la base dell'intelligenza artificiale
- spawnlocation: il punto di respawn, dove il personaggio viene posizionato quando muore e deve riprendere dall'inizio del livello
- maxhdiff: la massima differenza di altezza tra M.E.A.N. e il checkpoint perché quest'ultimo sia raggiungibile senza dover saltare
- CPspassed: un array che memorizza i checkpoint già attraversati, è usato per evitare che M.E.A.N. ritorni sui propri passi
- cpResetInt, cpResetWait: valori usati per resettare la lista di CP che sono già stati attraversati
- currentCP: il checkpoint che il personaggio sta attualmente cercando di raggiungere
- previousCP: quando un checkpoint risulta non raggiungibile, viene temporaneamente memorizzato in questa variabile per evitare di tentare inutilmente e in continuazione di raggiungerlo
- currentlyEscaping, currentlyChasing: i personaggi, rispettivamente, da cui M.E.A.N. sta scappando e che sta inseguendo
- health, maxhealth: servono a gestire la vita del personaggio

Gli stati possibili sono *roam*, *chase*, *escape*, *gotoCP*, *spawn* e *finish*. Lo stato roam è usato quando nei paraggi non sono presenti altri personaggi o checkpoint da seguire. Gli stati chase, escape, e gotoCP sono usati rispettivamente in caso di presenza di Dumb, mostri o checkpoint. Lo stato spawn è attivato quando M.E.A.N. muore ed è necessario effettuarne il respawn, e finish solo quando M.E.A.N. raggiunge la fine del livello.

All'evento di spawn, lo stato è impostato come roam, e in questo stato il personaggio si limita a camminare sulle piattaforme seguendo una direzione casuale.

Quando lo script che gestisce la percezione segnala la presenza di Dumb, la funzione chaseEnemy aggiorna lo stato a chase e aggiorna l'oggetto obbiettivo dell'inseguimento. In questo stato a ogni frame si verifica se sono presenti ostacoli tra i due personaggi: se ciò è verificato lo stato diventa roam nonostante Dumb sia ancora presente nell'area di percezione e l'inseguimento termina, in caso contrario la direzione viene aggiornata in base alla posizione reciproca di M.E.A.N. e Dumb, e il primo continua a spostarsi in direzione del secondo.

La verifica della presenza di ostacoli avviene tramite la funzione *noObstacles*. Questa traccia una linea tra M.E.A.N. e l'oggetto specificato come argomento tramite l'operazione di Linecast, che restituisce i gameobject da essa intersecati. Se uno di questi è un muro oppure un pavimento, sono presenti degli ostacoli, altrimenti il percorso tra il personaggio e l'oggetto analizzato è libero.

Se viene segnalata la presenza di un mostro, la funzione escapeFrom aggiorna lo stato a escape e aggiorna l'oggetto da cui fuggire. Nello stato escape, in ogni frame si verifica se è presente nei dintorni un checkpoint verso cui M.E.A.N. possa saltare: difatti i mostri non hanno questa capacità, ed è il metodo più efficace per riuscire a fuggire da essi. Se questa possibilità non esiste, la direzione viene aggiornata in base alla posizione reciproca dei due personaggi, e M.E.A.N. continua a spostarsi in direzione contraria rispetto al mostro.

Quando viene trovato un checkpoint, è invocata la funzione *foundCP*. Tale funzione verifica innanzitutto in che stato ci si trova: se lo stato corrente è escape, viene semplicemente salvato il checkpoint come currentCP.

Se invece lo stato è roam, si controlla che non siano presenti ostacoli tra M.E.A.N. ed il checkpoint, che il CP non sia presente nella lista di quelli già attraversati, e che non si tratti di un CP precedentemente scartato e salvato in previousCP perché irraggiungibile. Se tutti tre i controlli hanno successo, lo stato è modificato in gotoCP, la direzione del movimento è settata a quella che porta verso il checkpoint, ed esso è salvato in currentCP.

Una volta che M.E.A.N. è effettivamente passato attraverso un checkpoint, è chiamata la funzione *cpReached*. Di norma questo avviene con lo stato impostato a *roam* o più probabilmente *gotoCP*: in questa occasione si invoca la funzione *followCP* dello script responsabile dei movimenti del personaggio e roam diventa il nuovo stato. La funzione provvederà a far sì che M.E.A.N. prosegua il suo cammino nella direzione corretta.

A seguito di questa operazione, eseguita solo in certi casi, si verifica se il checkpoint fosse tra quelli non ancora visitati e se l'attuale direzione di movimento corrisponda a quella

indicata dal CP. In caso questo sia vero, il CP è aggiunto alla lista di quelli già visitati. Altrimenti, si verifica se il controllo precedente sia fallito a causa della direzione di movimento: se questo è il caso, ciò significa che M.E.A.N. sta muovendosi in direzione opposta al percorso per raggiungere la fine del livello, dunque il checkpoint è rimosso dalla lista di quelli attraversati.

Lo script contiene anche una semplice funzione per l'applicazione di danni al personaggio. Viene applicata la quantità di danni (oppure ripristinata una certa quantità di punti vita) specificata dall'argomento della funzione, se si è raggiunto lo zero viene invocata la funzione *Die*.

Quest'ultima imposta lo stato a *spawn*. Con stato impostato come spawn, si disattiva temporaneamente la gravità per M.E.A.N., la sua velocità è settata a zero, il numero di contatti col terreno, di cui si tiene traccia per controllare il movimento, e la lista di checkpoint passati sono azzerati, e la vita riportata al valore iniziale. Il gameobject viene quindi spostato nel punto di respawn, lo stato è cambiato a *roam* e la gravità è riattivata.

La transizione a *finish*, per concludere, avviene solamente quando M.E.A.N. raggiunge il termine del livello. È sottinteso che questa non deve essere stata prima raggiunta da Dumb.

7.5.5. M.E.A.N. - Movimento

Lo script responsabile del movimento contiene le variabili *currentcontacts* e *previouscontacts*, che tengono traccia della posizione rispetto al pavimento, *currentDirection*, che memorizza l'attuale direzione di movimento del personaggio, *charspeed* e *charjump*, rispettivamente la velocità e la massima capacità di saltare di M.E.A.N., quest'ultima intesa come la massima distanza in linea d'aria che può essere raggiunta con un salto.

Ad ogni frame, in base alla direzione di movimento e alla posizione rispetto alla piattaforma sottostante, sono gestiti l'orientamento del gameobject e le animazioni da riprodurre. Si tiene traccia tramite il collider del numero di contatti con il terreno, in modo da poter determinare quando il personaggio si trova su una piattaforma, e può quindi correre o saltare. Sempre tramite il collider, si verifica eventuali contatti con Dumb, che in questo caso muore direttamente, perdendo una vita.

Nel caso M.E.A.N. entri in contatto con un checkpoint, è questo script a invocare la funzione cpReached precedentemente descritta. Inoltre, come era anche per il caso dei mostri, il contatto con un oggetto DeadEnd determina il cambiamento di direzione del movimento, per evitare di dirigersi verso zone dove il personaggio potrebbe morire o restare bloccato.

La funzione *walk* semplicemente applica una velocità al gameobject per comportarne uno spostamento nella direzione attuale.

Vi sono poi le funzioni relative al salto. Esse sfruttano le seguenti equazioni tratte dalla fisica:

$\frac{1}{2} m v_y^2 = mgh$ uguaglianza tra energia potenziale e cinetica

$v_y = g\,t$ moto uniformemente accelerato

$v_x = \frac{s}{t}$

La chiamata a *jumpTo* usa le coordinate di M.E.A.N. e quelle passate come argomento per tentare di fare effettuare un personaggio verso il punto obbiettivo. Il salto è eseguito solo se la magnitudine del vettore tra i due punti, ovvero la sua lunghezza, è minore della massima capacità di salto del personaggio. Si calcola le componenti verticale e orizzontale in base alle formule:

$v_y = \sqrt{2\text{hg}}$ con cui si calcola la componente verticale

e poiché $t = \frac{v_y}{g}$ allora la componente orizzontale è $v_x = \frac{s}{t}$

Se la componente orizzontale è maggiore della velocità massima possibile camminando, il salto non è effettuato.

In aggiunta a diverse funzioni utili alla gestione della direzione di movimento e dei checkpoint, è compresa nello script anche una funzione *Teletrasporto*, invocata dall'oggetto Teleport con il punto di destinazione come argomento. Quando viene invocata, la funzione setta a zero la velocità, sia orizzontale che verticale, e aggiorna la posizione di M.E.A.N. in base all'argomento ricevuto.

In fig.6 è possibile vedere una rappresentazione grafica dell'automa a stati finiti alla base dell'IA

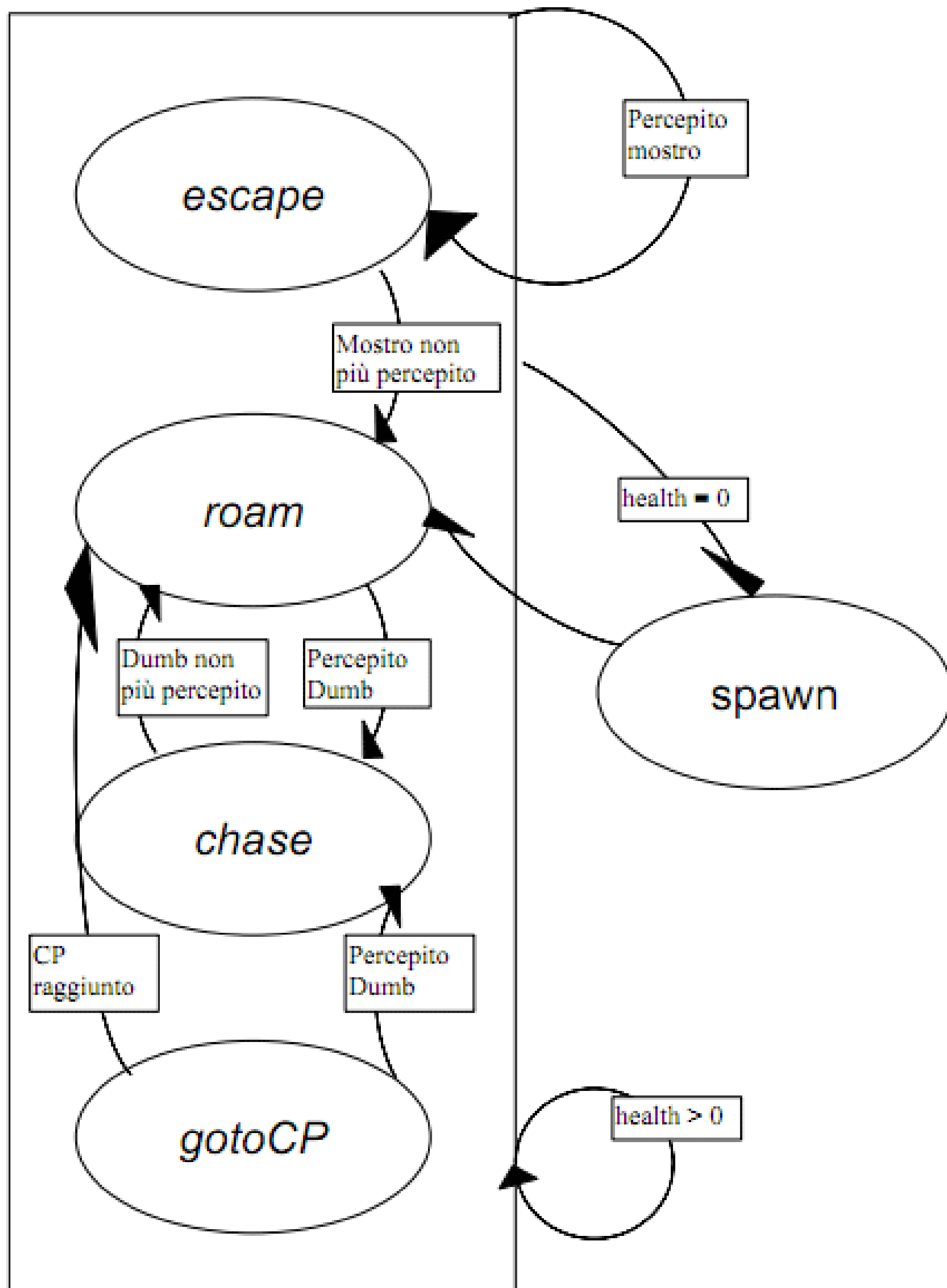

Fig.6 – Automa a stati finiti che rappresenta approssimativamente l'intelligenza artificiale di M.E.A.N.

Cod.2: Il codice del nucleo della IA di M.E.A.N.

```
var movementScript : MEANMove;
var camScript : cameraMove;
static var state;
var spawnpoint : GameObject;// Assegnato dall'Inspector!
In questo punto avviene il respawn
var spawnlocation;
var CPspassed = new Array();
var cpResetWait = 1000;
var cpResetInt = 0;
static var currentCP : GameObject;
var maxhdiff = 1;
var currentlyEscaping; // Il gameObject da cui MEAN sta
scappando
var currentlyChasing; // Il gameObject che MEAN sta
inseguendo
var maxhealth = 3.0; // La vita massima di MEAN
(variabile usata
var health : float = maxhealth;
var previousCP; // usato per evitare che MEAN "insista"
a cercare di raggiungere lo stesso CP che è per lui
irraggiungibile

function Start () {
     spawnlocation = spawnpoint.transform.position;
     movementScript = GetComponent(MEANMove); // Carica
lo script dei movimenti per accedere alle sue funzioni
     camScript =
Camera.mainCamera.GetComponent(cameraMove);
     state = "roam";
}

function LateUpdate () {
     // Se trascorre il tempo cpResetWait è azzerata la
lista di checkpoint visitati (in modo che MEAN possa
"ritrovare la strada")
     if (state!="gotoCP"){
          cpResetInt++;
          if (cpResetInt==cpResetWait) CPspassed = new
Array();
     }
}
// Nell'Update sono gestiti i diversi stati in cui si
può trovare il personaggio
```

```
function Update () {
     if (state=="finish") {
          transform.rigidbody.velocity=Vector3(0,0,0);
     }
     if (state=="escape") { // Se il personaggio sta
scappando da un mostro, si muove nella direzione opposta
al pericolo
          if (currentCP!=null && noObstacles(currentCP))
movementScript.jumpTo(currentCP.transform.position);
          else {


movementScript.changeDirection(Mathf.Sign(currentlyEscap
ing.transform.position.x-transform.position.x));
               movementScript.walk();
               }
     }
     if (state=="chase") { // Se il personaggio sta
inseguendo un nemico (Dumb), verifica se è ancora in
grado di raggiungerlo, se sì si muove nella sua
direzione, altrimenti smette di inseguirlo
          if (!noObstacles(currentlyChasing))
stopChasing();


movementScript.changeDirection(Mathf.Sign(transform.posi
tion.x-
currentlyChasing.gameObject.transform.position.x));
          movementScript.walk();
     }
     if (state=="spawn") { // Posiziona il personaggio
nel punto di respawn e lo fa tornare "vivo"
          transform.rigidbody.useGravity=false;
          rigidbody.velocity = Vector3(0,0,0); // Ferma
il personaggio
          transform.position = spawnlocation;
          MEANMove.currentcontacts=0;
          health=maxhealth;
          CPspassed = new Array();
          state="roam";
          transform.rigidbody.useGravity=true;
     }
     if (state=="roam") {
          movementScript.walk();
     }
     if (state=="gotoCP") { // Se il personaggio ha
trovato il Checkpoint "CP" successivo, si muove in
direzione di esso
          // Se il CP è raggiungibile saltando, ed è
```

```
troppo in alto per poterlo passare senza saltare, il
personaggio fa un salto in direzione del CP
          if
(movementScript.jumpToCalculateMagnitude(currentCP.trans
form.position) < MEANMove.charjump
               && (currentCP.transform.position.y-
transform.position.y) > maxhdiff &&
noObstacles(currentCP)) {


movementScript.jumpTo(currentCP.transform.position);
          /*} else if (noObstacles(currentCP) &&
Mathf.Abs(transform.position.x-
currentCP.transform.position.x)>5) { // Se il CP non è
raggiungibile saltando, ma non ci sono ostacoli tra esso
e il personaggio, quest'ultimo cammina in direzione del
CP


movementScript.changeDirection(Mathf.Sign(transform.posi
tion.x-currentCP.transform.position.x));
               movementScript.walk();*/ //sostituito con
cambiamento di direzione nel momento in cui viene
cambiato lo stato a "gotoCP"
          } else { // Altrimenti, il personaggio
continua semplicemente a camminare
               movementScript.walk();
          }
          if (!noObstacles(currentCP)) {
               state="roam";
               previousCP=currentCP;
               currentCP=null;
          }
     }
}

// Metodo per fare morire il personaggio
function Die() {
     state = "spawn";
}

// Aggiunge il checkpoint corrente a una lista di CP
passati
function addToList(cpToAdd : GameObject){
     CPspassed.Push(cpToAdd.GetInstanceID());
}

// Rimuove il checkpoint corrente e tutti i precedenti
dalla lista
```

```
function removeFromList(cpToRemove : GameObject){
     for (var i=0;i<CPspassed.length;i++){
          if (CPspassed[i]==cpToRemove.GetInstanceID())
CPspassed.removeAt(i);
     }
}

// QUando è stato trovato un checkpoint...
function foundCP(checkpoint : GameObject) {
     if (state=="roam"){ // Se lo stato attuale è "roam"
          if (newCP(checkpoint) &&
noObstacles(checkpoint) && checkpoint!=previousCP){ //
Se il CP non è presente nella lista e non ci sono
ostacoli tra esso e il personaggio
               state = "gotoCP"; // Lo stato cambia
               if (noObstacles(checkpoint))
movementScript.changeDirection(Mathf.Sign(transform.posi
tion.x-checkpoint.transform.position.x));
               cpResetInt = 0; // Azzero il conteggio
per il reset della lista di CP visitati
               currentCP=checkpoint;
          }
     } else if (state=="escape")
currentCP=checkpoint; // Se MEAN sta scappando da un
mostro mi limito a salvare il CP appena identificato
}

// Verifica se il CP non è presente nella lista dei CP
già "visitati"
function newCP(checkpoint : GameObject){
     for (var i : int in CPspassed){
          if (i==checkpoint.GetInstanceID()) return
false;
     }
     return true;
}

// Verifica che tra il personaggio e il checkpoint non
siano presenti ostacoli
function noObstacles(checkpoint : GameObject){
     var hitvalue : RaycastHit;

Physics.Linecast(transform.position,checkpoint.transform
.position,hitvalue); // Calcola una linea tra
personaggio e CP
     if (hitvalue.collider==null ||
(hitvalue.collider.gameObject.tag=="Jumpable" ||
hitvalue.collider.gameObject.tag=="NonEsplosivo")) {
         Debug.DrawLine (transform.position,
```

```
checkpoint.transform.position,Color.red);
          return false; // Se è presente un muro o un
pavimento, sono presenti degli ostacoli!
     } else {
     Debug.DrawLine (transform.position,
checkpoint.transform.position,Color.green);
          return true; // Altrimenti, il percorso è
libero
     }
}

// Quando il personaggio passa attraverso un CP
function cpReached(checkpoint : GameObject){
     if (state=="gotoCP" || state=="roam") { // Nel caso
"base"...
          movementScript.followCP(checkpoint); // Segue
l'indicazione data dal CP (destra, sinistra, fermo,
sali, fine...)
          state="roam";
     }
     if (newCP(checkpoint) &&
MEANMove.currentDirection==movementScript.getCPDir(check
point)) addToList(checkpoint); // Se il CP non era
ancora stato visitato, ed il movimento del personaggio
dopo averlo passato è quello indicato dal CP, lo
aggiunge alla lista
     else if (MEANMove.currentDirection!
=movementScript.getCPDir(checkpoint)) {
     // Altrimenti se il movimento attuale del
personaggio è diverso da quello indicato dal CP (ad
esempio sto scappando, cadendo...)
     // il CP e tutti quelli visitati successivamente
sono rimossi dalla lista
          var removeFrom=false;
          for (var k = 0;k<CPspassed.length;){
               if
(CPspassed[k]==checkpoint.GetInstanceID())
removeFrom=true;
               if (removeFrom==true)
CPspassed.RemoveAt(k);
               else k++;
          }
     }
     if (checkpoint.tag=="Finish" ||
checkpoint.tag=="FinishMEAN") state="finished";
}


function ApplyDamage( damage : float ) {
```

```
     health -= damage;
     if(health < 0) health = 0;
     if(health > maxhealth) health = maxhealth;
     if(health == 0) Die();
}

// Inizia a scappare dal mostro
function escapeFrom(monster : GameObject){
     state="escape";
     currentlyEscaping=monster;
}

// Smette di scappare dal mostro
function stopEscaping(){
     state="roam";
     currentlyEscaping=null;
}

// Inizia ad inseguire Dumb
function chaseEnemy(dumb : GameObject){
     currentlyChasing=dumb;
     state="chase";
}

// Smette di inseguire Dumb
function stopChasing(){
     state="roam";
}

function ToggleVisibility() {
    // toggles the visibility of this gameobject and all
it's children
     if
(transform.FindChild("rootJoint").renderer.enabled==true
)
transform.FindChild("rootJoint").renderer.enabled=false;
     else
transform.FindChild("rootJoint").renderer.enabled=true;
}
```

7.6. Informazioni sullo schermo

Lo script *CameraScreen*, applicato alla telecamera principale presente sulla scena, si occupa della gestione e visualizzazione del tempo, e della visualizzazione della salute e delle vite rimanenti.

7.6.1. Tempo rimanente

Il tempo restante è inizializzato al valore 120. Ad ogni aggiornamento, si verifica se esso ha raggiunto lo zero, in questo caso Dumb perde una vita e viene effettuato il respawn. Se invece non è questo il caso, il valore è decrementato della quantità di tempo trascorsa dal frame precedente. Il testo usato per visualizzare il tempo restante è poi aggiornato con il nuovo valore.

7.6.2. Salute

La visualizzazione della salute restante è un po' più complessa. Si usa una texture fissa, più un array composto di altre 6 texture che rappresentano la salute come un grafico a torta.

La texture fissa di sfondo è disegnata tramite GUI.skin, che modifica la skin globale usata. Viene poi calcolata una matrice TRS (traslazione, rotazione, scalamento) usata per adattare la GUI dalla risoluzione nativa alla risoluzione usata sul computer del giocatore.

Viene poi selezionata la texture corretta per rappresentare la salute restante, e tale texture viene disegnata sopra alla skin già presente, allineata nella posizione corretta.

I danni inflitti a Dumb sono aggiornati tramite questo script, che invoca la relativa funzione *ApplyDamage* presente nello script descritto precedentemente e, una volta aggiornate le informazioni sulla salute del personaggio, invoca nuovamente la funzione per il disegno della skin.

7.7. Telecamera

La telecamera è gestita tramite lo script *CameraScrolling*, applicato sulla telecamera principale. Questa è concentrata su di un unico obbiettivo, che segue lungo il livello, definito nella variabile *target*. Le altre variabili importanti sono:

- distance: la distanza dall'obbiettivo
- springiness: o elasticità, cioè quanto è rapida la telecamera nello spostarsi assieme all'oggetto che sta seguendo; valori bassi corrispondono a una telecamera meno reattiva
- heightOffset: lo sfasamento tra l'effettiva posizione dell'oggetto e il punto seguito
- distanceModifier: un modificatore alla distanza della telecamera dall'oggetto obbiettivo

- velocityLookAhead: è il numero di secondi di anticipo con cui viene stimata la posizione futura dell'obbiettivo, quando questo si sta muovendo
- maxLookAhead: un limite in orizzontale e verticale al massimo valore accettabile per la stima sulla posizione futura, questo evita che la telecamera inquadri una porzione di livello dove l'obbiettivo non è ancora presente

All'inizio dello script, viene settato l'obbiettivo da seguire: se è necessario è possibile in questo momento saltare a un'altra parte dello schermo senza usare il normale metodo che verrà poi eseguito nella funzione *LateUpdate*.

Passata questa fase, lo spostamento è gestito dalla funzione appena citata: a ogni frame si calcola il punto su cui la telecamera deve essere fissata. Questo avviene calcolando innanzitutto la posizione dell'obbiettivo, modificata in base alle variabili sopra citate: più nello specifico, si aggiunge alla posizione appena calcolata il valore di *heightOffset* sull'asse y e quello, cambiato di segno, di *distance* moltiplicato per *distanceModifier*. Una volta ottenuto tale valore, si ottiene la velocità attuale dell'oggetto che stiamo seguendo, tramite il suo rigidbody; da questa velocità calcoliamo l'offset della posizione stimata del personaggio tra *velocityLookAhead* secondi, tramite la semplice formula fisica:

$s=vt$ da cui deriva la formula che usiamo $lookAhead=targetVelocity\cdot velocityLookAhead$

Effettuiamo poi il clamp (limitazione del valore tra un minimo e un massimo) basato sull'offset appena calcolato e sul valore di *maxLookAhead*, sia sull'asse x che sull'asse y. Il risultato ottenuto viene infine aggiunto a quello calcolato precedentemente.

L'ultimo passaggio consiste nel clamp del valore attuale rispetto ai limiti superiore destro e inferiore sinistro del livello. Una volta arrivati al risultato definitivo si effettua un'interpolazione lineare tra le due posizioni prima e dopo l'aggiornamento, in base al valore dell'elasticità moltiplicato per il tempo trascorso dal frame precedente. Questo implica, considerato il valore scelto per l'elasticità, 4, che nei casi in cui tra un frame e l'altro trascorra un quarto di secondo (Time.deltaTime pari a 0.25) o un tempo superiore, si ignori completamente l'interpolazione. Grazie a questo accorgimento si riesce a mantenere un movimento fluido della telecamera, ma senza rischiare di perdere le tracce di Dumb nel momento in cui le prestazioni del gioco dovessero scendere considerevolmente.

7.8. Piattaforme

Le piattaforme sono la componente centrale del gameplay nella demo sviluppata per Run Level Run. Ogni tipo di piattaforma è costituita da un gameobject comprensivo di mesh, materiale e texture, e di un rigidbody per permettere agli oggetti dotati di collider di rilevare i contatti.

7.8.1. Piattaforme rotanti

Queste piattaforme sono contrassegnate con il colore verde, e danno al giocatore la possibilità di manipolarle tramite un'interazione di tipo click & drag. La rotazione avviene attorno a una delle estremità della piattaforma e attorno all'asse z, cioè interamente sul piano bidimensionale.

La piattaforma è evidenziata quando il mouse si trova sopra di essa, aggiungendo al materiale il colore bianco e quando il mouse è premuto, una variabile *selected* assume il valore true. Una volta selezionato, la rotazione attorno all'asse z è modificata a ogni frame, ruotando la piattaforma in senso antiorario in corrispondenza di movimenti del mouse verso sinistra in direzione orizzontale e verso il basso in direzione verticale. La rotazione avviene invece in senso orario muovendo il mouse verso l'alto in verticale e verso destra in orizzontale.

Il valore delle due componenti verticale e orizzontale è calcolato di volta in volta e moltiplicato per un fattore rappresentante la velocità di spostamento della piattaforma, e per un secondo valore rappresentante l'attrito. Il nuovo valore della rotazione è calcolato con i due valori appena citati, applicando poi l'interpolazione dal valore precedente a quello appena trovato.

7.8.2. Piattaforme sganciabili e a due posizioni

Questi due tipi di piattaforma sono governati dallo stesso script. La differenza si ha per via del fatto che le piattaforme a due posizioni dispongono di un Hinge Joint, un perno, che fissa una delle due estremità mentre l'altra è libera di muoversi.

A ogni piattaforma di questi due tipi, sono associate due variabili: *lastRot* e *attivata*. Quest'ultima è inizializzata a true, indicando la piattaforma fissata alla propria posizione iniziale. La prima variabile invece memorizza la rotazione di partenza della piattaforma.

All'avvio del livello, le piattaforme sganciabili sono colorate di giallo, quelle a due posizioni di rosso, per renderle facilmente distinguibili. Anche esse, come quelle rotanti, al passaggio del mouse vengono evidenziate aggiungendo il bianco al colore del loro materiale. Il rigidbody delle piattaforme, e dunque le piattaforme stesse, sono bloccate in posizione e la gravità è disattivata.

Quando il giocatore clicca su di essa, la piattaforma viene disattivata: le vengono tolti i vincoli che erano applicati al rigidbody e la gravità è attivata. Questo permette alle piattaforme sganciabili di cadere liberamente, e a quelle a due posizioni di lasciare cadere l'estremità libera.

Cliccando nuovamente su una piattaforma a due posizioni inattiva, si riporta il valore di *attivata* a true e la gravità è disabiliata. Inoltre durante ogni esecuzione della funzione Update viene eseguita un'interpolazione tra la rotazione attuale della piattaforma e quella originaria, riportandola alla posizione iniziale.

7.8.3. Piattaforme mobili

Le piattaforme mobili sono segnalate dal colore blu, e come tutte le altre vengono evidenziate in colore più chiaro quando il mouse vi passa sopra. Tra i propri componenti hanno due diversi script: uno responsabile del comportamento della piattaforma e l'altro adibito alla gestione degli effetti grafici che ne accompagnano il movimento.

Il primo script comprende alcune variabili:

- targetA, targetB: i due punti tra cui si muove la piattaforma
- enablePlatform: lo stato attivo o inattivo della piattaforma
- speed: la velocità a cui la piattaforma si muove
- timer: un oggetto che quando la piattaforma è attiva viene incrementato a ogni frame ed è usato nell'implementazione del moto tra i due punti targetA e targetB

Inoltre viene conservato un riferimento all'oggetto rappresentante Dumb, per il motivo che si vedrà tra breve.

Il moto della piattaforma è un moto armonico, che calcola un peso*weight* tramite la seguente funzione:

$$weight = 0.5 \cdot \cos(timer \cdot speed \cdot 2\pi) + 0.5$$

Si tratta dunque di un coseno scalato sull'asse x, e poi scalato e traslato sull'asse y per essere compreso tra 0.5 e 1. Questo valore viene poi usato per calcolare la posizione corrente, con la formula:

$$p = targetA \cdot weight + targetB \cdot (1 - weight)$$

In questo modo si ha un moto armonico con periodo pari a $speed^{-1}$ secondi, nel caso dei valori effettivamente usati, 2 secondi.

In fig.7 si può vedere un'esemplificazione di come potrebbe cambiare la posizione in relazione al tempo.

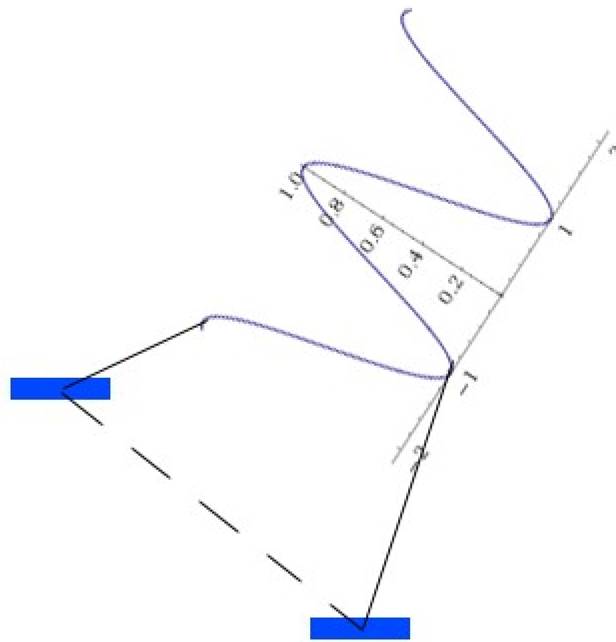

Fig.7 – origine del moto armonico delle piattaforme mobili

Il secondo script serve invece a gestire gli emettitori di particelle che si trovano sotto alle piattaforme mobili. Tali emettitori vanno attivati se la velocità della piattaforma è maggiore di zero in verticale, oppure se supera il valore di soglia memorizzato in *horizontalSpeedToEnableEmitters* in orizzontale. Questa velocità è calcolata nella funzione LateUpdate come differenza tra la nuova e la vecchia posizione, ed è usata nella funzione Update al frame successivo.

Nell'Update, si salva il valore di *areEmittersOn* in una variabile temporanea *wereEmittersOn*. La prima viene poi aggiornata, diventando true se uno dei due test citati precedentemente sulla velocità è verificato. Lo stato degli emettitori è poi aggiornato solo se è cambiato rispetto al frame precedente, cioè se fallisce il confronto tra le due variabili. Se il cambiamento è richiesto, è cambiato il valore dell'attributo *emit* a ogni elemento figlio della componente ParticleEmitter della piattaforma.

7.9. Punteggio

7.9.1. Fine del livello

Alla fine del livello è presente un gameobject con un collider che reagisce ai contatti con Dumb e M.E.A.N.; nel secondo caso viene causata la morte di Dumb, con il conseguente respawn o game over, a seconda delle vite rimanenti. Nel caso sia Dumb a raggiungere il termine del livello, invece, si calcola il punteggio finale prima di caricare la schermata che lo visualizza.

Il calcolo del punteggio è una combinazione di vite e tempo restanti. Il punteggio, e assieme a esso anche un riferimento al successivo livello da caricare, sono salvati tra le preferenze del giocatore. Dopo questa operazione, viene caricato un livello speciale il cui unico scopo è la visualizzazione del punteggio, prima dell'avvio del nuovo livello.

7.9.2. Visualizzazione del punteggio

La visualizzazione del punteggio avviene in un livello unicamente dedicato a questo. Si recupera dalle preferenze del giocatore il punteggio e il livello successivo, e nella funzione Update viene fatta aumentare la stringa che mostra i punti totalizzati fino a raggiungere il valore finale. Allo stesso tempo, in base al punteggio, viene calcolato il numero di stelle, da 1 a 3, che rappresentano la prestazione del giocatore in questo livello. Una volta alla fine dell'animazione viene evidenziato il testo “Next level”: cliccando su questo il livello successivo viene caricato.

8. Strumenti usati

8.1. Strumenti per l'organizzazione del lavoro

8.1.1. Google Docs

Google Docs è un servizio messo a disposizione da Google che permette di scrivere e apportare modifiche a documenti online, salvarli online e volendo scaricarli sul proprio computer o caricare documenti su Google Docs. Si può così scrivere documenti Word da qualunque computer e saranno salvati direttamente sul proprio account Google. In pratica è l'equivalente del pacchetto Microsoft Office su internet.

Google Docs offre la possibilità di condividere i documenti con utenti specifici oppure anche tramite la condivisione dell'URL del documento, rendendoli leggibili e modificabili a più persone. Questo permette di lavorare in gruppo sullo stesso documento, per necessità di brainstorming o per lavorare in gruppo sugli stessi documenti.

Nel contesto del progetto è stato usato nella collaborazione con gli altri componenti del team, in particolar modo nella stesura collaborativa del Game Design Document, come linea guida all'implementazione del gioco prima e in seguito come documentazione al termine del progetto.

8.2. Strumenti per l'implementazione

8.2.1. Unity 3D

Unity 3D è un game engine presente sia in versione gratuita che a pagamento, che permette di creare giochi 2D e 3D semplificando fortemente il lavoro degli sviluppatori per quanto riguarda grafica, fisica, interazione degli oggetti e via dicendo. È multipiattaforma, consentendo di sviluppare con lo stesso strumento applicazioni che potranno essere eseguite su Windows, Mac, Linux, via browser, e su tutti i sistemi più diffusi per dispositivi mobili.

Unity permette di lavorare a partire da un'interfaccia grafica tramite cui è possibile costruire la scena per il proprio videogioco senza ricorrere in maniera massiccia a scripting. Per ogni gameobject è possibile manipolare le variabili tramite il pannello Inspector, che visualizza anche tutti i componenti, e le variabili relative, a esso associati.

Dispone di un motore grafico proprio, dando inoltre la possibilità di sviluppare i propri shader se quelli forniti non soddisfano i requisiti desiderati dallo sviluppatore. Ha un motore fisico che permette la simulazione delle leggi fisiche e di applicare, modificare e leggere forze, velocità lineari e angolari, attrito, ecc.

Dal punto di vista grafico alcune funzioni particolari sono la possibilità di usare texture procedurali, che riducono il peso delle applicazioni realizzate, il Tree Generator, che consente la creazione procedurale di oggetti complessi quali sono gli alberi, e un'ulteriore funzione particolarmente apprezzabile è l'asset store, un market integrato dove è possibile acquistare gli asset grafici da usare nel proprio progetto.

Altra caratteristica molto utile è la possibilità di alterare il gioco mentre esso è in

esecuzione, senza la necessità di creare una build, per mettere a punto le impostazioni, provare nuove soluzioni, e in generale verificare in corso d'opera l'effetto delle modifiche sul gioco.

Unity usa tre linguaggi di scripting: JavaScript, C# e Boo: di questi, nel progetto sono stati usati i primi due, con particolare predilizione per il più intuitivo JavaScript. In tutti i casi, oltre al normale linguaggio, sono presenti delle librerie proprie di Unity grazie alle quali molti aspetti sono fortemente semplificati.

8.2.2. JavaScript

JavaScript è un linguaggio di scripting object oriented, usato molto spesso nello sviluppo di siti web. Nato con il nome Mocha, fu standardizzato per la prima volta tra il 1997 ed il 1999. La sua caratteristica principale è il fatto di essere, nonostante la sintassi simile a quella dei linguaggi compilati, un linguaggio interpretato, ciò significa che il codice non è compilato, ma eseguito a run-time da un interprete.

È inoltre debolmente tipizzato, il che consente una maggiore elasticità nel trattamento delle variabili, e debolmente orientato agli oggetti.

Altra particolarità degna di nota è che solitamente viene eseguito sul client: questo implica che la mole di lavoro richiesta al server è molto ridotta, anche nel caso di algoritmi molto complessi.

Un ultima peculiarità vantaggiosa è la possibilità di usare gli eventi, che permette di associare l'esecuzione di un dato frammento del codice in relazione con l'accadere di un evento, il quale può essere dovuto ad un azione dell'utente o allo stato della pagina e dei suoi elementi.

8.2.3. C#

C# è un linguaggio di programmazione object oriented, sviluppato da Microsoft specificamente per la programmazione nel framework .NET, ed è stato approvato come standard ECMA (European Computer Manufacturers Association). È considerato come il linguaggio che meglio riflette le linee guida di ogni programma .NET, i suoi tipi primitivi coincidono con quelli .NET e le sue astrazioni sono particolarmente adatte a gestire il framework.

Sono realizzati in C#, nell'ambito di questo progetto, gli script relativi alle bombe, sulle linee guida di un codice open source disponibile su internet.

8.2.4. Mono e MonoDevelop

Mono è un progetto open source coordinato da Novell per creare un insieme di strumenti compatibili con il Framework .NET, secondo gli standard ECMA (Ecma-334 e Ecma-335).

I più importanti di questi strumenti sono il compilatore C# e il Common Language Runtime.

La macchina virtuale di Mono contiene un motore JIT per vari processori: x86, SPARC, PowerPC, ARM, s390 (in modalità a 32 bit) e x86-64 e SPARC a 64 bit. La VM può eseguire una compilazione just-in-time o può pre-compilare il codice in codice nativo. Per altre architetture hardware esiste solo un interprete.

MonoDevelop è un ambiente di sviluppo parte del progetto Mono, realizzato principalmente per C# e altri linguaggi .NET. Tra le sue varie caratteristiche di MonoDevelop vi sono il supporto allo sviluppo multipiattaforma (Linux, Windows e Mac OSX), l'ambiente di lavoro configurabile, il supporto a vari linguaggi (C#, Visual Basic, C/C++ e altri), debugger e IDE integrati e personalizzabili e il designer grafico per GTK+.

8.3. Grafica & audio

8.3.1. Blender

Blender è un programma libero di modellazione, rigging, animazione, compositing e rendering di immagini tridimensionali. Dispone inoltre di funzionalità per mappature UV, simulazioni di fluidi, di rivestimenti, di particelle, altre simulazioni non lineari e creazione di applicazioni/giochi 3D. Richiede poco spazio per essere installato e può essere eseguito su molte piattaforme. Anche se è spesso distribuito senza documentazione o esempi è ricco di caratteristiche tipiche di sistemi avanzati di modellazione.

Alcune di queste funzionalità, che sono state esplorate durante la realizzazione del progetto, sono:

- Conversione da e verso numerosi formati per applicazione 3D, come Wings 3D, 3D Studio Max, LightWave 3D e altri.
- Strumenti per gestire le animazioni, come la cinematica inversa, le armature (scheletri) e la deformazione lattice, la gestione dei keyframe, le animazioni non lineari, i vincoli, il calcolo pesato dei vertici e la capacità delle mesh di gestione delle particelle.
- Caratteristiche interattive, come la collisione degli ostacoli, il motore dinamico e la programmazione della logica, permettendo la creazione di programmi stand-alone o applicazioni real time come la visione di elementi architettonici o la creazione di videogiochi.

8.3.2. Adobe Photoshop

Photoshop è un software proprietario prodotto dalla Adobe Systems Incorporated specializzato nell'elaborazione di fotografie e, più in generale, di immagini digitali.

Questo programma è in grado di effettuare ritocchi di qualità professionale alle immagini, offrendo enormi possibilità creative grazie ai numerosi filtri e strumenti che permettono di emulare le tecniche utilizzate nei laboratori fotografici per il processamento delle immagini, le tecniche di pittura e di disegno.

Un'importante funzione del programma è data dalla possibilità di lavorare con più "livelli",

permettendo di gestire separatamente le differenti immagini che compongono l'immagine principale.

Questo software è stato usato per la manipolazione delle texture applicate agli oggetti tridimensionali del gioco e per gli elementi dell'interfaccia grafica presente nel livello e nei menu iniziale e intermedi.

8.3.3. Audacity

Audacity è un editor di file audio multipiattaforma, rilasciato sotto la GNU General Public License. Il programma di base permette registrazione, riproduzione, modifica e mixaggio di un file audio. Una serie di operazioni aggiuntive sono possibili grazie a plugin già inclusi, con i quali è possibile intervenire su diversi parametri tra cui il volume, la velocità, l'intonazione, la compressione e la normalizzazione.

Il suo utilizzo nel progetto è stato volto all'utilizzo di file audio reperiti su repository di risorse open source per gli effetti sonori e le musiche del gioco.

9. Conclusioni

Il progetto si poneva il fine di seguire tutti i passi che portano dall'ideazione alla creazione di un videogioco indie. Il primo scopo è stata la realizzazione di un game concept appetibile, che attirasse l'interesse di sviluppatori e potenziali giocatori, e che rappresentasse un'idea originale. Successivamente si è ampliata l'idea iniziale, dando corpo progressivamente al gioco fino a raggiungere l'idea per cui è stata realizzata una demo.

Si è realizzata quindi una documentazione del gioco, il Game Design Document, che ne ha accompagnato lo sviluppo e ne documenta l'idea iniziale, lo stato attuale, e le possibilità di una futura espansione del gioco.

Successivamente si è studiata una suddivisione del lavoro nel gruppo, in base alle capacità, la disponibilità e gli interessi personali di ciascuno, definendo una serie di obbiettivi parziali da raggiungere. Nel corso dello sviluppo del progetto, sono state ampliate le capacità individuali di lavoro in team e di comunicazione, per coordinare il lavoro di più persone.

Nella fase di implementazione si sono cercate soluzioni originali per realizzare l'obbiettivo designato:

- è stata realizzata da zero un'intelligenza artificiale appositamente studiata per il gioco ideato, tentando di simulare il più possibile il comportamento di un ottimo giocatore umano in un caso (M.E.A.N.) e di semplici mostri dai semplici istinti nell'altro
- si sono realizzate tutte le risorse presenti nella demo, ognuna con le sue caratteristiche che influenzano il resto dell'ambiente e aggiungono una novità al gameplay, cercando di bilanciarle l'una con l'altra
- si sono affrontate numerose revisioni del sistema di gioco, cercando di rimuovere i bug, aumentare la semplicità d'interazione e la giocabilità
- soprattutto, si è cercato di creare la demo di un possibile prodotto nuovo, originale sia dal punto di vista dell'idea che da quello dell'implementazione e, soprattutto, divertente

9.1. Possibili sviluppi futuri

Il gioco per ora consiste in una demo formata da un limitato numero di livelli. La creazione di nuovi livelli rappresenta, dal punto di vista temporale, un collo di bottiglia per l'espansione del gioco: richiede infatti non solo l'ideazione del livello e la verifica che sia effettivamente possibile per Dumb raggiungerne la fine, ma anche l'importazione sulla scena di tutti gli oggetti necessari. Solo una volta fatto tutto questo, è possibile aggiungere i riferimenti per l'IA e arrivare al livello completo.

Una possibile soluzione a questo problema potrebbe essere la creazione di un piccolo tool che crea in maniera parzialmente casuale nuovi livelli, in base a criteri regolabili dall'utente. Tali livelli potrebbero essere poi messi a punto manualmente, risparmiando una considerevole quantità di tempo.

Vi è inoltre l'aspetto della grafica, slegato dall'implementazione: nella realizzazione della demo è stata evidente la mancanza di una persona sufficientemente abile graficamente, che potesse dare al gioco un aspetto veramente originale e accattivante.

Se questo non rappresenta un vero ostacolo a una demo, o a un progetto di tesi sviluppato nell'ambito di una facoltà di Ingegneria, se si volesse giungere a un prodotto completo e alla sua distribuzione (gratuita o a pagamento) sarebbe necessario rivedere la grafica del gioco cercando un aiuto più competente in materia.

10. Riferimenti bibliografici

- https://developer.mozilla.org/en/JavaScript : documentazione sul JavaScript del developer center di Mozilla

- http://msdn.microsoft.com/en-us/vcsharp/aa336809.aspx : specifiche C#

- http://docs.unity3d.com/Documentation/ScriptReference/ : documentazione dello scripting in Unity

- http://answers.unity3d.com/ : comunità di Unity in cui è possibile porre domande relative a problemi e reperire una notevole quantità di informazioni dalle domande già risolte

- http://stackoverflow.com/questions/ : comunità "Q&A" di sviluppatori software

- http://games.ws.dei.polimi.it : materiale relativo agli aspetti teorici

- Tracy Fullerton. *Game Design Workshop, Second Edition: A Playcentric Approach to Creating Innovative Games.* Gama Network Series. Morgan Kaufmann, 2008 : libro di testo per il corso di Videogame Design and Programming

- http://it.wikipedia.org/wiki/ : dettagli relativi ai vari strumenti utilizzati