



UNIVERSITÀ  
DELLA CALABRIA

DIPARTIMENTO DI INGEGNERIA INFORMATICA,  
MODELLISTICA, ELETTRONICA E SISTEMISTICA  
(DIMES)

CORSO DI LAUREA TRIENNALE IN INGEGNERIA  
INFORMATICA

*Tesi di Laurea dal titolo:*

**TECNICHE DI PROGETTAZIONE E  
SVILUPPO DI VIDEOGAMES**

CANDIDATO

Lorenzo Morelli

Matricola: 169153

RELATORE

Professore

Sergio Flesca

A.A. 2016-2017



*A zio Gerardo, sempre sorridente sulla sua vespa azzurra.  
A zio Angelo che mi ha regalato un pizzico della sua follia.  
“Mira alle stelle, arriverai almeno alle montagne”*



# Indice

<b>Elenco delle figure</b>	<b>v</b>
<b>Introduzione</b>	<b>vii</b>
<b>1 Videogames</b>	<b>1</b>
1.1 Storia . . . . .	1
1.2 Definizione . . . . .	6
1.3 Mercato . . . . .	8
1.4 Classificazione e tipologie . . . . .	11
1.5 Componenti . . . . .	19
<b>2 Fasi di progettazione e sviluppo</b>	<b>23</b>
2.1 Figure chiavi nello sviluppo . . . . .	23
2.1.1 Professioni legate al Design . . . . .	24
2.1.2 Professioni legate alla programmazione . . . . .	26
2.2 Fasi di progettazione . . . . .	28
2.3 Metodologie di sviluppo . . . . .	29
2.3.1 Metodologie Agili . . . . .	30
2.3.2 Game-Scrum . . . . .	33
<b>3 Tecniche e strumenti di progettazione</b>	<b>45</b>
3.1 Architettura del videogioco . . . . .	45
3.2 Motori grafici: . . . . .	48
3.2.1 Unity 3D . . . . .	51
3.2.2 Unity Assets Store . . . . .	74
3.2.3 Platform Dependent Compilation . . . . .	81
3.3 Modellazione e Design . . . . .	82
3.3.1 Blender . . . . .	86

---

<b>4 Progetti “Star Naitum” e “HologramRun!”</b>	<b>91</b>
4.1 Pre-Produzione . . . . .	91
4.2 Star Naitum . . . . .	94
4.2.1 Progettazione . . . . .	94
4.2.2 Produzione e Script . . . . .	96
4.2.3 Play-Testing . . . . .	111
4.3 Hologram Run . . . . .	113
4.3.1 Progettazione . . . . .	113
4.3.2 Produzione e Script . . . . .	116
4.3.3 Play-Testing . . . . .	118
4.4 Post-Produzione . . . . .	118
<b>Conclusioni</b>	<b>123</b>
<b>Bibliografia</b>	<b>125</b>
<b>Ringraziamenti</b>	<b>127</b>

# Elenco delle figure

1.1	CathodeRay Tube Amusement Device . . . . .	1
1.2	Tennis for Two . . . . .	2
1.3	Pong . . . . .	3
1.4	8° generazione console . . . . .	5
1.5	Chris Crawford . . . . .	7
1.6	Newzoo . . . . .	8
1.7	AESVI . . . . .	10
2.1	Team di sviluppo multidisciplinari . . . . .	24
2.2	Metodologie Agili . . . . .	31
2.3	Opportunità lavorative EA Sports Career . . . . .	34
2.4	Step e SprintCycle Game-Scrum . . . . .	38
2.5	Esempio Kanban Board . . . . .	40
3.1	Architettura videogames . . . . .	46
3.2	Unity . . . . .	52
3.3	Licenze Unity . . . . .	53
3.4	Unity Interface . . . . .	54
3.5	Project View . . . . .	55
3.6	Hierarchy View . . . . .	56
3.7	Toolbar . . . . .	56
3.8	Transform Tools . . . . .	56
3.9	Transform Gizmo Toggles . . . . .	56
3.10	Play/Pause/Step . . . . .	57
3.11	Layers . . . . .	57
3.12	Layout . . . . .	57
3.13	Scene View . . . . .	57
3.14	Gizmo . . . . .	58
3.15	Trasformazioni del GameObject . . . . .	58

3.16 Scene View control bar . . . . .	59
3.17 Inspector . . . . .	59
3.18 Game View . . . . .	60
3.19 Pulsanti Play Mode . . . . .	60
3.20 GameView Control Bar . . . . .	60
3.21 La classe Object . . . . .	61
3.22 Pannello Material . . . . .	64
3.23 Infinite, 3D Head Scan creato da Lee Perry-Smith . . . . .	67
3.24 Other Settings . . . . .	67
3.25 Componente Camera . . . . .	68
3.26 Esempio dell'effetto Sun Shafts . . . . .	70
3.27 Il pannello Lens Flare Inspector . . . . .	71
3.28 Animation Tools . . . . .	73
3.29 Animator Tools . . . . .	74
3.30 Star Naitum Prefabs . . . . .	74
3.31 Esempio di Assets . . . . .	75
3.32 Cartelle base di un progetto Unity . . . . .	75
3.33 Unity project browser . . . . .	76
3.34 Assets workflow . . . . .	76
3.35 Proprietà Immagine . . . . .	77
3.36 Proprietà Audio . . . . .	78
3.37 Unity Assets store . . . . .	79
3.38 Pannello di importazione Assets . . . . .	80
3.39 StandardAssets ParticleSystem . . . . .	81
3.40 Pannello di Build Unity . . . . .	81
3.41 Classificazione e tipologie di modellazione . . . . .	83
3.42 Blender . . . . .	87
4.1 Questionario . . . . .	93
4.2 Pagina Play Store Star Naitum . . . . .	94
4.3 Sprite Astronauta . . . . .	95
4.4 Star Naitum Script . . . . .	97
4.5 Script EnemyAI . . . . .	98
4.6 Griglia PathFinding . . . . .	99
4.7 A* PseudoCode . . . . .	100
4.8 Esempio A* ottimale . . . . .	101
4.9 Esempio A* sub-ottimale . . . . .	102
4.10 PathFinding a RunTime . . . . .	103
4.11 Classe Audio . . . . .	103

---

4.12 Script AudioManager . . . . .	104
4.13 Script Camera2Dfollow . . . . .	105
4.14 Layer Parallaxing . . . . .	105
4.15 Script Parallaxing . . . . .	106
4.16 Tiling . . . . .	107
4.17 Script Tiling . . . . .	107
4.18 Script Platform2DController . . . . .	108
4.19 Script Weapon . . . . .	110
4.20 Script Game Manager . . . . .	111
4.21 PlatformDependent Architetture Compatibili . . . . .	112
4.22 Pagina Play Store HologramRun . . . . .	113
4.23 Prisma olografico . . . . .	114
4.24 Modello 3D Prisma e Riproduzione in Plexiglass . . . . .	115
4.25 Script HologramRun . . . . .	116
4.26 Script SpawnCubic . . . . .	117
4.27 Posizionamento multicamera HologramRun . . . . .	118
4.28 Versioni Build Android . . . . .	120
4.29 Progetto Star Naitum . . . . .	121
4.30 Progetto HologramRun! . . . . .	122



# Introduzione

La tesi di laurea è incentrata sullo studio delle tecniche di progettazione e sviluppo di software videoludico atto ad intrattenere una vasta e diversificata gamma d'utenza. L'argomento centrale è il videogames che verrà trattato da ogni punto di vista, partendo dal lato puramente artistico, sino ad arrivare alla parte tecnico-progettuale dove verranno osservate le tecniche, gli approcci e le metodologie che vengono utilizzate per sviluppare tali prodotti. Il tutto parte dagli albori del videogioco, iniziando dalla storia e proseguendo con la classificazione e la descrizione del mercato in cui tale prodotto si colloca, arrivando anche a definire quali sono le tecniche di programmazione più usate, i linguaggi, i paradigmi, le metodologie e concludendo il lavoro di tesi con la realizzazione dei progetti “Star Naitum” e “HologramRun!”. Tali progetti sono stati realizzati per mettere in pratica tutte le nozioni apprese durante la realizzazione della tesi e per approcciarsi al software Unity3D spesso usato da piccoli team per sviluppare prodotti videoludici. Il progetto “Star Naitum” si pone come un gioco Platform 2D in cui l’utente deve annientare orde di nemici cercando di totalizzare il maggior numero di punti. Il progetto è stato sviluppato e pensato per essere un prodotto multipiattaforma in grado di essere eseguito su:

- Windows
- MacOS
- Linux
- Android

Il secondo progetto denominato “HologramRun” sfrutta le leggi dell’ottica per creare un ologramma dove viene proiettata l’immagine del gioco. Il videogioco è molto semplice in quanto la struttura a prisma dell’ologramma deve essere relativamente piccola e quindi si è optato per un videogioco Platform in cui l’obiettivo è quello di rimanere, per il maggior tempo possibile, su una piattaforma mentre il cubo luminoso viene governato dall’utente al fine di evitare

gli ostacoli. Per la realizzazione dell'ologramma c'è stato bisogno di creare una struttura apposita da porre sullo schermo in modo da sfruttare il fenomeno di riflessione e creare così l'effetto olografico; la struttura è semplice da realizzare e di materiale accessibile a tutti proprio per dare la possibilità a chiunque di testare il prodotto. Il porting sui vari ambienti è stato possibile grazie al software di sviluppo Unity che, con le dovute attenzioni e settaggi, è in grado di generare dallo stesso progetto più output (denominate "build") per le diverse architetture supportate. La fase di Porting quindi, grazie all'astrazione fornita dal software Unity, è stata dedicata ad adattare gli input, che prima venivano catturati da tastiera e mouse, a dispositivi mobili quali tablet e smartphone. La tesi è così organizzata: il capitolo 1 è dedicato a tutti i vari aspetti del videogames, si descrive la storia e si prosegue andando a definire quali sono le componenti principali di qualsiasi videogames, si descrive e analizza il mercato e le opportunità derivanti da esso, si descrivono i vari tipi e classificazioni di videogiochi andando anche a fornire una precisa definizione di cosa sia un videogioco. Nel capitolo 2 vengono trattate le tematiche progettuali dello sviluppo quali le figure professionali all'interno di un'azienda che sviluppa questo tipo di prodotti, le fasi di progettazione e quindi i vari step che vanno a delineare il workflow del progetto e si conclude con l'analisi e la descrizione delle metodologie più usate per lo sviluppo di tali prodotti andando anche ad analizzare nel dettaglio la "Game-Scrum" methodology. Il capitolo 3 è dedicato agli strumenti utilizzati maggiormente per lo sviluppo di videogiochi e quindi segue un'analisi di tali applicazioni come i software di modellazione 3D ed i motori grafici. Tali strumenti verranno approfonditi andando a prendere in esempio Unity3D 2017.1 che è stato utilizzato per lo sviluppo dei progetti nel capitolo 4. Quest'ultimo è dedicato allo showcase dei progetti "Star Naitum" e "HologramRun" dove verranno descritte tutte le fasi di sviluppo del videogioco partendo dall'idea e andando a descrivere tutti gli aspetti incontrati nella realizzazione degli stessi quali:

- Scelta del motore grafico
- Applicazione della metodologia "Game-Scrum"
- Progettazione
- Implementazione
- Sviluppi futuri del progetto
- Studio e creazione del dispositivo Hologram

- Pubblicazione su Store Android

Seguono in fine i capitoli di Conclusioni, Bibliografia e Ringraziamenti.



# Capitolo 1

## Videogames

### 1.1 Storia

La storia dei videogiochi comincia negli Stati Uniti alla fine degli anni '40. Nel 1947 Thomas Toliver Goldsmith Jr ed Estle Ray Mann crearono un simulatore di lancio di missili chiamato CathodeRay Tube Amusement Device.

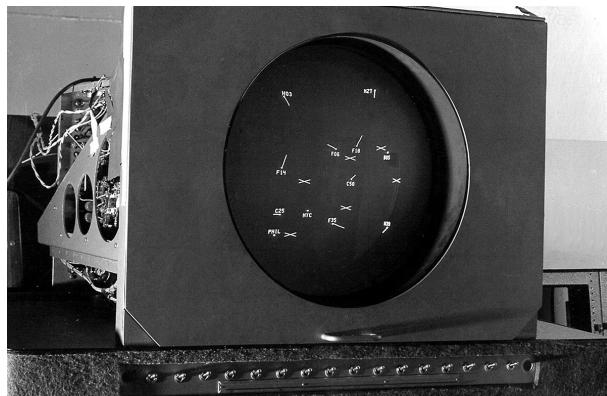


Figura 1.1: CathodeRay Tube Amusement Device

Il giocatore tramite una manopola doveva tentare di colpire dei bersagli, posti con degli adesivi sul monitor di un radar. Questo gioco fu brevettato nel 1948, tuttavia non fu mai commercializzato e rimase un prototipo. Nel 1952 il professor Alexander Shafto "Sandy" Douglas dell'Università di Cambridge creò Noughts and Crosses, un simulatore del classico gioco del tris. Questo programma funzionava solamente sui computer EDSAC, utilizzati solamente dall'Università di Cambridge e permettevano al giocatore di sfidare il computer.

Nel 1958, il fisico americano William Higinbotham creò Tennis for Two per intrattenere i visitatori del Brookhaven National Laboratory. Questo fu il primo gioco ad essere pensato per più di una persona e consisteva in un simulatore di tennis dotato di una manopola per parte per controllare la traiettoria della palla e da un tasto per lanciarla.

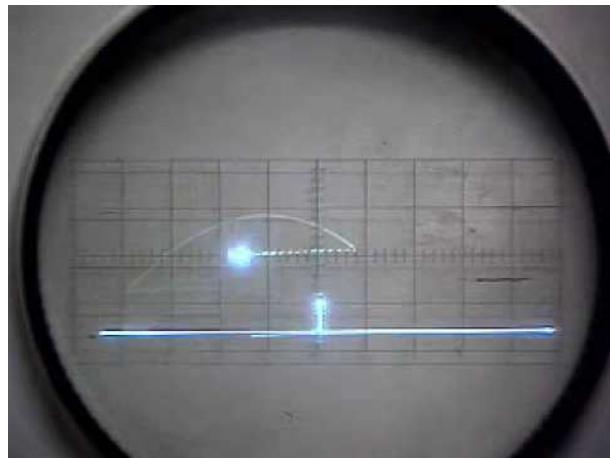


Figura 1.2: Tennis for Two

Inoltre questo fu il primo caso in cui venne introdotta una simulazione della forza di gravità tramite un apposito algoritmo. Dopo i primi videogiochi, nati più come applicazioni didattiche che come veri e propri giochi, nel 1961 Steve Russell, allora studente del MIT (Massachusetts Institute of Technology), creò Spacewar!, gioco per due giocatori che presentava un mondo dotato di regole fisiche e che si modificava per la prima volta in tempo reale. Questo gioco consisteva in una battaglia tra due navicelle spaziali in cui il primo giocatore che colpiva la navicella avversaria con un missile vinceva. Inizialmente i controlli del gioco erano affidati ai tasti del PDP-1, computer con i quali veniva distribuito, ma a causa della scomodità degli stessi fu inventato da Alan Kotok e Bob Saunders il primo vero e proprio joystick della storia. Grazie al successo ottenuto da Spacewar!, molti studiosi si interessarono al nuovo tipo di intrattenimento. Nel 1967 l'ingegnere americano Ralph Baer creò il primo videogioco visualizzabile su un televisore, Bucket Filling Game, e l'anno successivo inventò il Brown Box, prototipo del Magnavox Odyssey, prima vera e propria console, la quale entrò in commercio nel 1972 e permetteva l'utilizzo di diversi videogiochi. Nel 1975 essa fu tuttavia tolta dal mercato. Le vendite, infatti, si rivelarono insufficienti, a causa dell'alto prezzo e del fatto che questo dispositivo poteva essere utilizz-

zato solo con alcune tipologie di televisori Magnavox. Oltre a rappresentare un traguardo importante per aver dato il via al mercato delle console, il Magnavox Odyssey portò con se anche la prima periferica esterna, il Light Gun, fucile utilizzabile con alcuni giochi, ed il primo utilizzo nella storia degli sprite, figure bidimensionali in grado di spostarsi rispetto allo sfondo. In contemporanea, nel 1971, nasce anche il primo videogioco da sala giochi, Galaxy Game, inspirato a Spacewar!, che funzionava con l'inserimento di una moneta da 10 centesimi nella macchina. Questo gioco fu in seguito migliorato e riprogrammato da Nolan Bushnell e Ted Dabney e ne vennero prodotti 1500 esemplari. Il successo, tuttavia, non fu grande a causa dell'elevata difficoltà del gioco. Bushnell allora cambiò il nome della sua azienda in Atari, e nel 1972 inventò l'omonima console. Contemporaneamente viene rilasciato Pong, simulazione di Ping Pong in bianco e nero rilasciato in versione coin-op (ossia come cabinato da sala giochi in cui è necessario l'inserimento di una moneta per giocare).

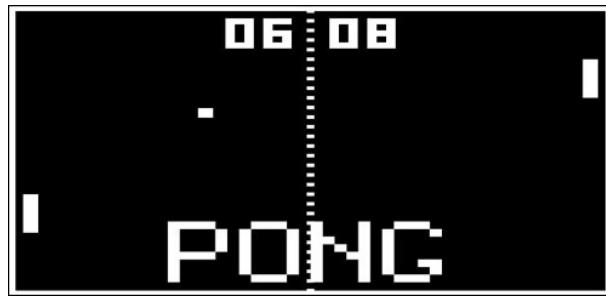


Figura 1.3: Pong

Pong ebbe un successo enorme, portando alla produzione di quasi 20.000 cabinati. Il 3 agosto del 1975 Pong viene rilasciato anche per la console casalinga di Atari. Il 1976 segnò l'inizio della seconda generazione di console, caratterizzata dall'avvento dei sistemi a 8-bit, che si concluse nel 1983 con la crisi dell'industria videoludica. Il Fairchild Channel F ed il 1292 Advance Programmable Video System diedero inizio a questa nuova fase introducendo le ROM, immagini del gioco contenenti vari elementi dello stesso quali, per esempio, la grafica e i dialoghi, che resero molto più facile la programmazione. L'anno seguente fu rilasciato l'Atari VCS 2600, seguito nel '78 dall'Intellivision di Activision. Nel 1979 venne rilasciata la prima console portatile dotata di cartucce intercambiabili, l'MB Microvision. Nel 1982 vide la luce Vectrex, prima console al mondo ad introdurre la grafica tridimensionale. Molte altre console casalinghe vennero create in questo periodo, fino alla crisi mondiale del 1983, che portò alla chiusura e

alla bancarotta di numerose case produttrici, soprattutto in Stati Uniti e Canada. Con l'ingresso nel 1983 della giapponese Nintendo nel mercato videoludico, inizia la terza generazione di console, caratterizzate ancora da sistemi ad 8-bit. Nel '83 viene lanciato il Nintendo Entertainment System (NES), console che ebbe un enorme successo di cui vennero vendute 61,91 milioni di copie, secondo i dati ufficiali. Proprio in questi anni nacquero saghe di titoli che hanno ancora oggi un grandissimo successo. Nel 1985 nasce il SEGA Master System, che con 13 milioni di copie vendute fu l'unico vero concorrente del NES, riscuotendo un buon successo in Europa ed in Brasile. Sempre in questo periodo ci fu una vera e propria esplosione del mercato delle console portatili, guidata dal Game Boy di Nintendo, rilasciato del 1989 e dismesso nel 2003, che unitamente al successivo modello, Game Boy Color, conta circa 118,69 milioni di unità vendute. Con l'avvento dei sistemi a 16-bit iniziò la quarta generazione di console. Sega fu la prima a lanciare una console a 16-bit con il Sega Mega Drive nel 1988. Due anni più tardi anche Nintendo rilasciò il suo sistema a 16-bit, il Super NES, che si affermò come console di maggior successo vendendo circa 49 milioni di copie, contro le 29,5 di Sega. Sega tentò anche di affermarsi sul mercato delle console portatili lanciando il Game Gear, il quale tuttavia non riuscì ad affermarsi ed a scalzare il Game Boy, vendendo solamente 11 milioni di unità. La quinta generazione di console ebbe inizio nel 1993, con l'avvento della grafica tridimensionale. I primi sistemi ad utilizzare questa novità furono l'Amiga CD32, il 3DO di Panasonic, l'Atari Jaguar ed il Sega Saturn. La definitiva consacrazione del 3D tuttavia arrivò nel 1995, quando Sony lanciò la Playstation, console che ebbe un enorme successo, arrivando a piazzare oltre 102 milioni di copie. Nintendo l'anno successivo lancia il Nintendo 64, passando così direttamente dalla console a 16-bit a quella a 64-bit, ma, nonostante nelle generazioni precedenti, non riuscì a raggiungere le vendite di Playstation, vendendo circa 33 milioni di copie. Sul fronte del mercato portatile tuttavia Nintendo mantiene la leadership, lanciando nel 1998 la nuova versione del Game Boy, il Game Boy Color. La sesta generazione di console fu quella dei sistemi a 128-bit. A dare il via a questa generazione fu Sega, che con il suo Dreamcast, rilasciato nel 1998, tentava di riconquistare un ruolo da protagonista sul mercato. Il successo iniziale fu buono, fino a quando nel 2000 Sony lanciò Playstation 2. Fu proprio questa console a dominare la sesta generazione, con più di 150 milioni di copie vendute. Nintendo l'anno successivo lanciò il Game Cube, mentre Microsoft entrò nel mercato con la sua Xbox. Entrambe le console ebbero un discreto successo, vendendo rispettivamente 21 milioni di pezzi la prima e 24 milioni di unità la seconda, restando tuttavia molto lontane dalla console marchiata Sony.

Sul fronte delle console portatili invece è ancora Nintendo a dominare la scena, rilasciando nel 2001 Game Boy Advance. Nel 2003 Nokia tentò di entrare nel mercato delle console portatili con l'NGage, una via di mezzo tra una console portatile ed un telefono cellulare, senza però grande successo con solo 3 milioni di unità vendute. Nel 2004 Nintendo lanciò il Nintendo DS, dando il via alla settima generazione di console. Questa console portatile era caratterizzata da due schermi LCD, di cui uno tattile. L'anno seguente arrivò sul mercato la seconda console di casa Microsoft, Xbox 360, seguita l'anno successivo da Playstation 3 di Sony e Wii di Nintendo. Sempre nel 2005, Sony debuttò anche sul mercato delle console portatili con Playstation Portable (PSP). La console più venduta di questa generazione è stato il DS di Nintendo, con 154 milioni di unità vendute, seguita da Wii (101), Playstation 3 (57), Xbox 360 (84) e PSP (67,8). L'ottava generazione delle console per videogiochi è iniziata nel 2011 con l'uscita del Nintendo 3DS ed è tuttora in corso. Questa generazione è caratterizzata principalmente dalla diffusione di console con architetture simili a quelle usate dai PC, che dovrebbero facilitare le operazioni di porting dei vari giochi da una piattaforma all'altra. Questa generazione vede inoltre la diffusione di console basate su Android, il cui debutto è cominciato nel 2013 con Ouya, e delle Steam Machine, mini-PC basati su sistema operativo Steam. Esistono alcuni tablet Android o Windows ai quali si possono collegare delle particolari docking station che li rendono ibridi tra console per videogiochi e tablet, come ad esempio il Wikipad, l'iBen e il Razer Edge Pro. Nel corso di questa era sono state introdotte varie innovazioni tecnologiche: Console che supportano lo standard di risoluzione video 4K: Playstation 4 Pro (2016), X-Box One X (2017), supporti per la realtà virtuale: Playstation VR (2016), console ibride tra console casalinghe e portatili: Nintendo Switch (2017).



Figura 1.4: 8° generazione console

Questa generazione inoltre vede profondamente cambiato il concetto stesso di generazione di console. Secondo i maggiori produttori infatti non avremo più generazioni ben distinte ogni sette anni circa, ma un'evoluzione più frequente dove le console verranno via via potenziate, ma resteranno retrocompatibili con le precedenti, come succede per i PC o per i dispositivi mobili (per esempio le citate Playstation 4 Pro e X-Box One X).

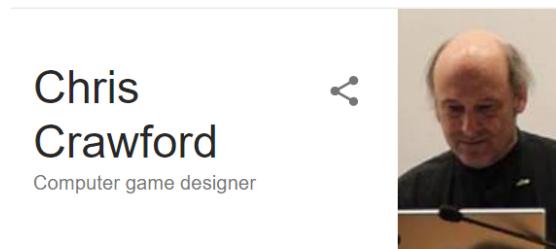
## 1.2 Definizione

Il videogioco è un gioco gestito da un dispositivo elettronico che consente di interagire con le immagini di uno schermo. Il termine generalmente tende ad identificare un software, ma in alcuni casi può riferirsi anche ad un dispositivo hardware dedicato ad uno specifico gioco. Colui che utilizza un videogioco viene chiamato “videogiocatore” e si serve di una o più periferiche di input quali il joystick, la tastiera, il gamepad. Questa può essere la definizione informale di cosa sia un videogioco ma alla domanda “cos’è esattamente un videogioco?” la risposta più autorevole che si può trovare è proprio di chi i videogiochi li sviluppa, li progetta e li manipola a proprio piacimento. Il famoso Computer Game Designer Chris Crawford prova a definire il termine videogioco usando una serie di dicotomie (divisione logica di un concetto in nuovi concetti distinti e contrapposti), riportando l’intervista in lingua originale quello che ne esce è il seguente elenco numerato di frasi che definisce un videogioco.

1. Creative expression is art if made for its own beauty, and entertainment if made for money.
2. A piece of entertainment is a plaything if it is interactive. Movies and books are cited as examples of non-interactive entertainment.
3. If no goals are associated with a plaything, it is a toy. If it has goals, a plaything is a challenge (Crawford specifica, per questo punto, anche che:  
(a) un giocattolo può diventare un gioco se il giocatore può creare delle regole da seguire, e (b) The Sims e The SimsCity sono giocattoli, non giochi).
4. If a challenge has no “active agent against whom you compete,” it is a puzzle; if there is one, it is a conflict. (Crawford ammette che questo è

un test soggettivo. I videogiochi con un'intelligenza artificiale chiaramente percepibile come algoritmi possono essere giocati come puzzle; questi includono i modelli usati per eludere i fantasmi in Pac-Man).

5. Finally, if the player can only outperform the opponent, but not attack them to interfere with their performance, the conflict is a competition. However, if attacks are allowed, then the conflict qualifies as a game.



## Chris Crawford

Computer game designer

Christopher Crawford is a computer game designer and writer. He designed and programmed several important computer games in the 1980s, including Eastern Front and Balance of Power. [Wikipedia](#)

**Born:** June 1, 1950 (age 67), [Houston, Texas, United States](#)

**Known for:** Game Developers Conference

**Video games:** Trust & Betrayal: The Legacy of Siboot, [MORE](#)

**Education:** University of California, Davis, University of Missouri

### Books

[View 5+ more](#)



Figura 1.5: Chris Crawford

Questa definizione, per quanto bizzarra possa essere, riesce a rispecchiare quello che è un videogioco: Un software complesso, interattivo, orientato agli obiettivi, con agenti attivi contro cui giocare, in cui i giocatori (compresi gli agenti attivi) possono interferire tra loro. Esiste una netta distinzione nel mondo videoludico

che divide i giochi “Casual” dai “Competitive”. I titoli Casual vengono proposti a quella fetta di utenza che vuole un passatempo senza doversi sforzare né alla comprensione delle meccaniche di gioco e né all’apprendimento delle stesse; i videogiochi “Competitive” sono invece finalizzati ad un pubblico più ristretto e molto più tecnico e di conseguenza esigente. Per questi ultimi titoli citati ci possono volere giorni, mesi o anni per approfondire tutte le meccaniche e quindi comprendere a pieno ciò che il prodotto offre, spesso un singolo titolo contiene più modalità di gioco per accontentare entrambe le utenze. In sostanza il videogioco è un prodotto mutevole: riesce a porsi in maniera differente in base alle priorità dell’utente che in quel momento si sta interfacciando con il videogioco stesso, è anche un prodotto software molto complesso in quanto deve riuscire a gestire una moltitudine di eventi, tutto in tempo reale e con prestazioni sufficienti al godimento del gioco stesso. Oltretutto deve essere in grado di girare bene in tutte le piattaforme possibili (console Microsoft, console Sony, Pc, console portatili, mobile) ma deve anche avere una forte infrastruttura di rete su cui basare le meccaniche multiplayer come server dedicati, meccanismi di sincronizzazione tra utenti, misure di sicurezza, accounting e gestione dei profili, messaggistica e localizzazione.

### **1.3 Mercato**

In questo paragrafo verrà presentato il mercato dei videogiochi. Inizialmente si cercherà di offrire una panoramica su quelli che sono, in termini di fatturato, i più grandi mercati mondiali.



Figura 1.6: Newzoo

Nell’ultimo anno si è registrata una grandissima crescita del mercato cinese, tanto da arrivare a contendere il primato con gli Stati Uniti. In Europa, invece,

la classifica resta piuttosto stabile, con Germania, Inghilterra e Francia ad occupare il podio. Dopo aver fatto questo, si passerà ad un excursus su quella che è oggi la situazione italiana. Dopo essere diventato il quarto mercato europeo nel 2011, il fatturato italiano ha subito una contrazione scendendo al di sotto del miliardo di euro. Oggi tuttavia il mercato torna a crescere, riportandosi sui livelli del 2009. A 70 anni dalla loro nascita, l'industria dei videogames rappresenta oggi il più grande settore nel mercato dell'intrattenimento. Questo primato è garantito da un fatturato di circa 86,3 miliardi di dollari, con una previsione di crescita stimata in un ulteriore 9,4%, toccando quota 91,5 miliardi di dollari. Gli Stati Uniti rappresentano oggi il mercato più vasto, con un fatturato che si aggira intorno ai 22 miliardi di dollari annui, secondo dati forniti da Newzoo, società specializzata in ricerche di mercato e analisi predittive con focus sul mondo dei videogiochi. Secondo i dati forniti dall'ESA (Entertainment Software Association 35, società di categoria americana) sono circa 155 milioni i gamer americani, di cui il 42% dichiara di giocare regolarmente (almeno tre ore a settimana). Un dato interessante riguarda la divisione per genere dei consumatori. Se da un lato comunemente si tende a credere che i videogiocatori siano quasi tutti maschi, i sondaggi dall'altro dimostrano una divisione piuttosto equa, con il 56% di giocatori di sesso maschile e il 44% di genere femminile. Un altro luogo comune che, sempre secondo il report dell'ESA, viene sfatato è quello sull'età media dei gamer. Se, infatti, ci si potrebbe attendere che la maggior parte di questi siano giovani e adolescenti, i dati dell'ESA collocano al primo posto tra i consumatori, persone di età compresa tra i 18 ed i 35 anni (30% dei consumatori), seguiti dagli over 50 (27%) e solo al terzo posto si trovano gli under 18, che rappresentano il 26% degli utenti. Sempre secondo Newzoo, il ruolo di mercato leader nel 2018 potrebbe tuttavia non appartenere più agli Stati Uniti. Se per questi, infatti, si prevede una crescita del 3% circa, per la Cina questo dato si stima sarà intorno al 23%, portando la Cina ad un fatturato annuo di circa 22,2 miliardi di dollari. È interessante notare che la Cina da sola ha quasi tanti giocatori quanti Stati Uniti ed Europa messi insieme, circa 446 milioni. A questo dato si aggiunge anche quello relativo ai giocatori paganti: la Cina conta, infatti, oltre 156 milioni di giocatori paganti, quasi il 50% in più rispetto ai circa 109 milioni degli Stati Uniti. Il terzo mercato per dimensione è quello Giapponese. Sede di alcuni dei più grandi colossi mondiali del settore, tra cui Sony e Nintendo, il Giappone vanta un giro di affari di circa 12,3 miliardi di dollari, con una crescita stimata tra il 2016 ed il 2018 del 1,2%. Questo Paese, inoltre, detiene un record: in Giappone, che conta 66,5 milioni di gamer su una popolazione totale di 126,8 milioni, si ha la più alta spesa

media annua per giocatore, che ammonta a 312,97 dollari l'anno. Per quanto riguarda l'Europa, il primo mercato è quello tedesco. Nel 2018 viene stimata per la Germania una crescita del fatturato del 2% circa, che dovrebbe portare il volume d'affari intorno ai 3,7 miliardi di dollari, collocandosi al quinto posto, preceduta dalla Corea del Sud con circa 3,8 miliardi di dollari. Al sesto posto mondiale, e secondo europeo, troviamo il Regno Unito, con un fatturato di 3,5 miliardi di dollari e una crescita dell'1,7%. Particolarità di questo mercato è la percentuale dei gamer "paganti". Su 36,4 milioni di giocatori britannici, il 61% spende denaro in videogames, la percentuale più alta in Europa.



**AESVI**  
ASSOCIAZIONE EDITORI SVILUPPATORI VIDEOGIOCHI ITALIANI

Figura 1.7: AESVI

Analizzando la documentazione fornita dall'AESVI (Associazione Editori Sviluppatori Videogiochi Italiani) riguardante gli ultimi dati di mercato, significativo è anzitutto il dato complessivo che fotografa le dimensioni del mercato, tornato nel 2016 è sopra quota un miliardo di euro (precisamente €1.029.928.287) con un'impennata del 8,2% rispetto al risultato pur positivo del 2015. Di questo miliardo di euro, il 62% è rappresentato dalle vendite di videogiochi (copie fisiche e digitali), mentre il 30% è stato dato dall'hardware console e il restante 8% dagli accessori. È particolarmente interessante guardare alla composizione dei €636.908.554 fatturati dai videogiochi che si dividono in oltre 346 milioni di euro derivanti dal mercato "fisico" con un trend in de-crescita dell'1,1% laddove invece il segmento dei videogiochi in "digital delivery" esplode del +32,8% superando i 290 milioni di euro in valore. Un altro numero atteso è il traguardo raggiunto dalle vendite di console in Italia nel 2016, ovvero 1.153.113 unità che si dividono in 910.455 console da salotto (e il 98% di esse sono state tutte di ultima generazione: Wii U, Xbox One o PlayStation 4) in crescita del 12,9% rispetto allo scorso anno e 242.658 console portatili anch'esse in crescita di un robusto 8,7% e ancora capaci di ritagliarsi un ruolo importante in un segmento

“mobile” pervaso dal videogiocare su Smartphone e Tablet. Osservando la distribuzione per fasce d’età, si nota una significativa concentrazione nelle fasce più adulte: circa il 62% dei videogiocatori ha un’età compresa tra i 25 e i 55 anni (nella fascia 25-34 abbiamo il 18,4%, nella fascia 35-44 il 22,4% e nella fascia 45-54 il 20,6%). Dato interessante, gli over 65 sono di poco più numerosi degli adolescenti: rappresentano il 7,9% dei giocatori, mentre nella fascia 14-17 troviamo il 7,2% del totale. Sul fronte della distribuzione per genere si nota invece una presenza maschile e femminile equivalente: il 50% dei videogiocatori sono uomini e il 50% donne. I dati confermano che stiamo assistendo ad un allargamento importante del target di videogiocatori. Da un lato i nati negli anni ’80, cresciuti giocando ai videogiochi, diventati a loro volta genitori stanno trasmettendo questa passione ai loro figli. Parallelamente il gioco mobile e online rendono il medium sempre più accessibile alla popolazione più adulta, in particolare agli over 65. Il pubblico di videogiocatori si allarga quindi sia verso le nuove generazioni che verso le popolazioni più adulte. Nonostante l’Italia occupi una posizione di rilievo tra i mercati europei, il numero di aziende e di persone impiegate nel settore è piuttosto esiguo, seppure in crescita. Sul suolo nazionale, infatti, si contano circa un centinaio di aziende, la maggior parte delle quali sono start-up molto giovani (solo il 20% di esse è in attività da più di 8 anni) strutturate come microimprese. Tuttavia dal 2011 è aumentato il numero delle aziende più strutturate, con circa il 40% di esse che oggi conta più di sei lavoratori. In totale nel 2013 le aziende italiane hanno fatturato circa 20 milioni di euro, un dato relativamente basso se comparato agli altri grandi paesi europei, ma si tratta comunque di un dato incoraggiante se si considera che questo rappresenta un aumento del 15% rispetto al 2011. Per quel che riguarda la distribuzione di queste aziende, la maggior parte sono collocate al Nord, con circa il 30% delle aziende situate in Lombardia ed il 12% in Piemonte. Nel Centro Italia la regione con la maggior presenza di aziende è il Lazio, in cui si trovano circa il 10% delle imprese nazionali, mentre al Sud le regioni di maggior rilievo sono Sicilia e Campania, che ospitano ognuna circa il 7% delle software house. Per ulteriori informazioni consultare il sito <http://www.aesvi.it/>

## 1.4 Classificazione e tipologie

Dalla loro nascita fino ad oggi, i videogiochi si sono costantemente evoluti creando man mano dei generi completamente diversi tra loro. In questo capitolo andremo quindi ad analizzare il tipo di situazione ambientale in cui il videogiocatore si trova rispetto al videogioco in quanto oggetto, cercando inoltre di spie-

gare come i videogames vengono attualmente raggruppati e classificati in base ai loro generi. Le principali piattaforme o luoghi dove è possibile videogiocare sono:

- sale giochi
- computer
- console
- internet
- cellulari/ smartphone / tablet

La moda dei videogiochi esplose principalmente nelle sale giochi, luoghi in cui era possibile socializzare e allo stesso tempo giocare con gli amici. I videogiochi Arcade o Coin-op (cioé Coin-operated, che funzionano mediante inserimento di una moneta), hanno una durata contenuta che solitamente va dai cinque ai dieci minuti in modo da invogliare i videogiocatori ad inserire altre monete. In passato questa situazione portò il mondo videoludico a far guadagnare molto denaro. Oggi però, l'attività videoludica praticata nelle sale giochi e nei bar è in forte calo; questa situazione è stata causata soprattutto dall'avvento delle console che permette l'utilizzo del videogioco comodamente da casa propria senza un eccessivo dispendio economico. Le console sono ormai diventate un successo dell'intrattenimento. Un videogioco che viene creato per console è probabile che sia più venduto rispetto allo stesso titolo proposto per computer. Questo è dovuto a due motivi fondamentali, il primo è perché non tutti i computer hanno sufficiente potenza di calcolo per gestire un videogioco moderno, ci sono dei componenti hardware che sono di vitale importanza per la fluidità del titolo (CPU, Scheda Video, Ram ecc). La maggior parte di questi offrono una miriade di scene create in computer-grafica che normalmente un pc, se non dotato di una opportuna scheda hardware dedicata alla grafica, non riesce a gestire. Il secondo motivo è che le console costano meno rispetto ad un pc, perché solitamente sono prive di monitor (fatta eccezione per quelle portatili), possiedono un hard-disk non troppo performanti e svolgono meno funzioni rispetto ad un computer pur offrendo un eccellente qualità audio e video, un sistema di collegamento online per le partite multiplayer, la navigazione sul web e la riproduzione di film e contenuti multimediali. Un altro modo per usufruire dei videogames è tramite internet. I giochi online sono in continua espansione, il successo può essere spiegato sia dall'interazione sociale dovuta all'opportunità di giocare e sfidarsi con altre persone, sia la maggior parte dei titoli presenti in rete è gratuita. Gran parte del successo del gioco online nasce grazie all'avvento dei browser-games,

videogiochi multi-player fruibili tramite il proprio browser di navigazione, che non richiede nessun tipo di installazione, e che quindi ha permesso di poterne usufruire tranquillamente sia sui computer di casa, che su quelli d'ufficio. Adesso molti di questi videogames si trovano anche all'interno di importanti social networks, ad esempio Facebook, che sono popolati da un numero enorme di persone. Per finire, un altro metodo è quello che permette di videogiocare tramite cellulare, smartphone o tablet. Negli ultimi anni il continuo miglioramento della tecnologia li ha resi abbastanza potenti da supportare diversi videogames, a tal punto da vederli come i protagonisti della nostra vita quotidiana.

*I generi dei videogames:* Per genere di videogame si intende la tipologia di canoni interni al gioco, così come avviene per i generi letterari o cinematografici. In tal senso le tipologie possibili sono molteplici. Secondo Francesco Carla, che è stato fra i primi giornalisti italiani ad occuparsi di videogiochi attraverso una rubrica sulle pagine di "Mcmicrocomputer", è possibile suddividere i videogiochi in:

- Azione ed avventura: Un genere successivamente suddivisibile in diversi filoni: punta e clicca il cui termine è principalmente usato per definire la classica azione eseguibile tramite il mouse che permette di interagire con oggetti, persone o luoghi come ad esempio in Monkey Island; survival horror il cui termine significa horror di sopravvivenza e definisce una categoria di videogames basati sulla sopravvivenza del personaggio giocato in un'atmosfera di suspense e paura. In questo filone i nemici del giocatore sono solitamente zombies, fantasmi, mostri oppure esseri umani impazziti. L'ambientazione è solitamente costituita da spazi chiusi oppure luoghi aperti ma particolarmente bui, tra gli esempi di questo genere c'è il famoso Resident Evil; nasconditi e spara (Metal Gear Solid), combattimenti spaziali (Space Invaders) e adventure: un genere di giochi in cui è presente sia l'esplorazione che l'azione come ad esempio in Tomb Raider;
- Sport: Questi giochi possono essere principalmente di due tipi diversi: individuali, nei quali il giocatore impersona lo stesso soggetto come ad esempio nei videogiochi di golf, o collettivi dove il giocatore si può identificare con i vari personaggi della sua squadra (tutti i capitoli di FIFA);
- Guida e gare: Si riferisce ai videogiochi di corse d'auto come i classici Pole Position e Gran Turismo;

- Sparatutto in prima persona: Detto anche first person shooter, FPS, come Doom, ciò significa che il giocatore si identifica in un personaggio con i cui occhi vede l'azione e si ritrova in una ripresa in soggettiva;
- Platform e puzzle: Il termine piattaforma è adottato per indicare i videogames dove la meccanica di gioco implica l'attraversamento di livelli costituiti da piattaforme a volte disposte su più piani. Tradizionalmente i platform sono strutturati in modo da avere il personaggio che si muove da sinistra verso destra dello schermo. L'eroe può saltare, salire o scendere scale, combattere nemici e collezionare oggetti. Solitamente è anche in grado di aumentare progressivamente le proprie capacità, per via del passaggio di livello o per aver trovato alcuni oggetti particolari. Basta pensare ai videogiocchi come Super Mario Bros che è un famoso esempio di platform. Il genere dei puzzle è invece piuttosto complicato da descrivere, solitamente viene richiesto al giocatore di spostare e far incastrare tra loro delle forme geometriche in modo da raggiungere un determinato obiettivo. In questo genere il colore e le forme giocano un ruolo fondamentale. Per avere successo non hanno bisogno di una trama ma devono essere semplici ed immediati. Il videogioco più famoso di questa categoria è senza ombra di dubbio Tetris;
- GDR o RPG: Sono gli acronimi di Gioco di Ruolo o Role Playing Game, sono videogames in cui la narrazione è molto sviluppata e derivano da giochi come ad esempio la saga di Ultima di Richard Garriott;
- Strategia o STR: Che è l'acronimo di strategia in tempo reale in inglese diventa RTS ovvero Real Time Strategy. Più che di un genere di videogioco si tratta di una modalità ludico-narrativa per la quale il giocatore deve svolgere una serie di operazioni complesse; è una tipologia di gioco nel quale le capacità di prendere decisioni di un giocatore hanno un grande impatto nel determinare il risultato finale. Questo tipo di genere può essere particolarmente complicato per un videogiocatore alle prime armi, ma se portato avanti può regalare soddisfazione e divertimento, un esempio particolarmente famoso di questo è Starcraft;
- Simulazioni: È una categoria di videogioco che cerca di simulare un aspetto della realtà e in genere richiede un mix di abilità, fortuna e strategia. Si cerca per quanto possibile di riprodurre l'esperienza reale come se il giocatore si trovasse realmente nella situazione rappresentata. Il videogioco può anche essere ambientato in un mondo fantasioso, ma il tema del

gioco è affrontato nel dettaglio come se fosse reale. I simulatori più famosi sono quelli di volo e quello di guida come ad esempio Microsoft Flight Simulator ed Assetto Corsa. Esistono inoltre anche dei famosi simulatori di vita reale come ad esempio The Sims e Second Life;

- God game: Con questo termine viene indicato un gioco dove il personaggio ha il ruolo di un dio. Ci si riferisce ai videogiochi strategici in forma di simulazione di un ambiente o talvolta di interi mondi e popolazioni, spesso di stampo fantasy, che fanno assumere al giocatore il ruolo di un'entità dai poteri divini o addirittura soprannaturali come ad esempio Populous o Black and White che hanno come obiettivo il dominio di enormi territori. La maggioranza dei god games è in tempo reale, mentre alcuni sono basati su una meccanica a turni;
- Picchiaduro: È il termine adottato per indicare i videogames dove lo scopo principale è quello di affrontare i nemici tramite incontri di lotta di vario genere, sia a mani nude che attraverso le cosiddette armi da mischia.

Questa classificazione dei videogiochi non è considerata universale, ce ne sono molte altre alla quale fare riferimento e che differiscono da questa appena descritta. Sono stati effettuati diversi studi sulla classificazione dei videogames e ognuno risulta essere valido perché viene sempre aggiornata e deriva anche dalle categorie che ci propone la televisione attraverso la sua pubblicità. Vanno quindi segnalati altri studi in cui vengono presentate classificazioni simili ma che differiscono da quella precedente per vari particolari e tra queste c'è la tassonomia proposta da Alan e Frdric Le Diberder del 1993 e 1998 che è articolata in tre grandi famiglie di videogiochi:

- Di riflessione
- Di simulazione
- D'azione

Alla prima categoria appartengono i generi classici come Monopoly, strategia come Chessmaster e i giochi di ruolo e d'avventura come ad esempio The Legend of Zelda. Alla seconda categoria appartengono i sistemi complessi come Sim City, i giochi sportivi come Need for speed o Gran Turismo, i simulatori come Assetto Corsa, e i giochi di strategia militare come Commandos o Men of war. Fanno parte invece della terza categoria i giochi di sport distinguendoli in giochi individuali e di squadra come FIFA; i giochi di combattimento come Tekken;

i giochi di tiro come ad esempio Doom e Wolfenstein; i giochi di riflessione riferendosi ai puzzle come Tetris e i giochi platform come Super Mario Bros. Questa particolare suddivisione gode dell'originalità in quanto ha adottato la scelta di definire i generi con una terminologia diversa rispetto a ciò che si trova ovunque. Un'altra categorizzazione, dei videogiochi, semplice e facile da comprendere, in cui inserire alcune specifiche tipologie, è quella proposta da Cantoia, Romeo e Besana del 2011. Essi considerano i videogiochi riassumibili in quattro macrocategorie:

- videogiochi d'azione
- videogiochi d'avventura
- videogiochi strategici
- videogiochi di simulazione

I *videogiochi d'azione* sono caratterizzati da interazioni rapide e continue, in cui un personaggio viene guidato dal giocatore attraverso un ambiente più o meno complesso; stimolano a pensare e ad agire in fretta e garantiscono un alto coinvolgimento a livello percettivo-motorio. Il personaggio solitamente è guidato in prima persona in un contesto di una visuale in soggettiva, cioè il giocatore vede sullo schermo le mani davanti a se, o in terza persona, ovvero il giocatore vede davanti a se l'intero corpo del personaggio muoversi nell'ambiente. Validi rappresentanti di questa categoria sono i platform, caratterizzati dal movimento complesso del personaggio attraverso ambienti formati da piattaforme e livelli. Questo è tipico degli sparatutto chiamati anche shooter, dove il personaggio affronta moltissimi nemici ferendoli a colpi di arma da fuoco; allo stesso modo funzionano i videogiochi hack and slash (Diablo) che si differenziano dallo sparatutto perché si basano su combattimenti ravvicinati all'arma bianca. Il videogioco free roaming è anch'esso basato sul movimento, in modo particolare sull'esplorazione: mentre il platform si svolge in ambienti limitati, in questo genere è possibile girovagare liberamente all'interno dei luoghi. Spazi realmente limitati sono tipici dei giochi picchiaduro, incentrati sui combattimenti corpo a corpo, quando è a scorrimento si tratta di un avvicendarsi di avversari che combattono contro il protagonista e segue un determinato percorso.

I *videogiochi d'avventura*, secondo gli autori, pongono l'attenzione su una narrazione di alto livello. Con il termine avventura grafica si intendono tutti i giochi pieni di enigmi, sfide di indagine e ragionamento che se ben integrate

con storie complesse e piene di mistero, queste possono avere modalità di interazione simili ai giochi d'azione anche se molto più spesso sono caratterizzate dall'interagire solo con il mouse, cliccando nell'ambiente sui punti di interesse dove possono nascondersi indizi ed enigmi, i cosiddetti punta e clicca. Un sottogenere spesso caratterizzato da elementi action ridotti è quello del survival horror, dove i protagonisti risolvono gli enigmi con lo scopo di mettersi in salvo in tempo da situazioni pericolose come case infestate da fantasmi o da zombie. Gli autori inseriscono in questa categoria anche gli RPG, questa sigla può essere preceduta dagli MMO nel caso in cui il videogioco sia accessibile da internet e preveda la possibilità di incontrare i personaggi guidati da altri giocatori in giro per il mondo. Il termine strategico si riferisce principalmente ad ambientazioni di guerra e conflitti, in cui bisogna sconfiggere eserciti e popoli avversari.

Nei *videogiochi strategici* il giocatore non guida un solo personaggio ma bensì interi gruppi. Il genere strategico, normalmente, può essere in tempo reale, cioè i giocatori agiscono contemporaneamente sia nello sviluppo delle loro forze che nei momenti di scontro, quindi il tempo diventa un fattore determinante in quanto passa allo stesso modo per tutti i giocatori, o a turni, dove la disposizione delle forze e la gestione del conflitto concedono un tempo preciso ad ogni giocatore, il giocatore può fare le sue mosse soltanto quando il concorrente precedente ha completato le proprie e decide di passare il turno o compie un'azione che in automatico passa il comando al giocatore successivo.

Secondo gli autori i *videogiochi di simulazione* prevedono la rappresentazione di eventi, dinamiche o attività della vita. I racing game ad esempio riproducono gare di corsa, gli sports game offrono la possibilità di giocare partite virtuali di ogni sport. Esistono anche videogiochi musicali che permettono di suonare strumenti veri tramite specifiche periferiche che ne riproducono il suono, così come i giochi di danza invitano il fruitore ad eseguire passi di ballo (rilevati dal computer per stabilire i punteggi) mostrati da uno o più personaggi sullo schermo. In questa categoria si inseriscono anche gli Arcade game: con questo termine si era soliti identificare le macchine installate nei bar, che davano la possibilità di interagire con ambienti semplici e precise attività. Molte di esse simulano attività ludiche reali, come i giochi di carte o altri giochi da tavolo. È da precisare che numerosi Arcade non sono simulativi, bensì ricreano attività particolari non presenti nella realtà. Esiste inoltre anche una classificazione per i giochi di ruolo online; gli MMOG o MMO (Massively Multiplayer Online Game) sono giochi per computer che sono in grado di supportare centinaia o migliaia di videogiocatori contemporaneamente, e si gioca su internet. Questo tipo di gioco è ambientato in un mondo virtuale. Gli MMO permettono ai giocatori di

competere o combattere con altre persone che si possono trovare anche dall'altra parte del mondo. Nella maggior parte dei MMO è necessario che il giocatore impieghi molto tempo nel suo videogame quindi è sicuramente più adatto a chi ha più tempo libero. Quando parliamo di videogiochi online siamo costretti ad inserire anche altre caratteristiche:

- MUD: È l'acronimo di Multy User Dungeon anche se spesso è considerato l'acronimo di Multy User Dimension o Domain, identifica una categoria di giochi di ruolo eseguiti su internet, attraverso il computer da più utenti. Si tratta di videogiochi testuali, dove ogni giocatore interagisce con il mondo e con gli altri utenti semplicemente digitando dei comandi da tastiera. Ogni utente controlla un personaggio che si muove all'interno di un mondo virtuale, organizzato in stanze e zone (una zona è un raggruppamento di più stanze), e può interagire coi personaggi degli altri utenti o con quelli gestiti dal computer. Dopo che il personaggio di un giocatore ha raggiunto il livello massimo diviene immortale oppure assume le caratteristiche di una divinità acquisendo i poteri che hanno gli amministratori. Solitamente, gli immortali usano questa loro capacità proponendo delle sfide agli altri giocatori, dette Quest, organizzando delle gare con delle ricompense finali. Alcuni MUD dispongono di comandi per consentire l'accesso anche ai giocatori non vedenti, garantendogli la possibilità di giocare ed interagire con gli altri superando i loro handicap;
- MUSH: È l'acronimo di Multy User Shared Habitat, deriva dal programma TinyMUD e risale ai primi anni del Novanta. Sono più orientati all'interpretazione rispetto ai MUD classici. I giocatori sono molto attenti a creare i propri personaggi il più accuratamente possibile;
- MOO (MUD Object Oriented): È un sofisticato programma informatico che permette a più utenti di collegarsi via Internet ad un ambiente condiviso, che contiene stanze ed altri oggetti, e di interagire fra loro e con l'ambiente simultaneamente. Questi vengono utilizzati per la realizzazione di giochi di ruolo, sistemi per conferenze, in ambiente educativo per la formazione a distanza, ma la loro natura è prevalentemente sociale oppure ludica. Un MOO è costituito da un database e al suo interno vi sono differenti livelli di utenti con differenti mansioni. Al primo livello si trovano i Wizard cioè gli amministratori che hanno accesso completo a tutte le funzionalità. Il livello inferiore è costituito dai Programmer che hanno la possibilità di creare nuovi oggetti, modificare quelli già esistenti ma, a differenza dei Wizard, non hanno accesso alla parte amministrativa.

tiva del sistema. Subito dopo troviamo il Builder che può solo clonare gli oggetti già esistenti senza poterne creare di nuovi. L'utente base è il player che ha solo la caratteristica di interagire con l'ambiente. Esiste per finire un utente di servizio chiamato Guest, solitamente utilizzato da chi si connette per la prima volta al sistema, che ha la funzionalità del Player ma con delle restrizioni e tutti i suoi comandi possono essere letti su un particolare canale di comunicazione, in modo da garantirgli aiuto da parte di un giocatore di livello superiore.

## 1.5 Componenti

Un software videoludico è molto più articolato di quanto possa sembrare semplicemente interagendo con esso, può essere diviso in molteplici componenti che fusi tra loro riescono a dar vita al videogioco, una possibile divisione in componenti potrebbe essere la seguente: grafica, narrativa, storia, IA e interazioni ma se cerchiamo di raggruppare le componenti di un videogames in un insieme minimale che copra ogni aspetto ci rendiamo conto che la suddivisione più corretta può essere quella in macro-componenti:

- Parte software
- Parte artistica

La *parte software* è, a sua volta, divisa in tantissime sotto-porzioni ma in sostanza è il lavoro tecnico che viene effettuato dagli sviluppatori e dal team che prende le specifiche e le traduce in chiamate, script, collisioni ed in generale nel codice del gioco.

La *parte artistica* è anch'essa divisa in ulteriori sotto-componenti ma in sostanza è la parte più creativa del progetto dove vengono prese decisioni inerenti alla storia, architettura, dialoghi, sceneggiatura, musica e vengono realizzati tutti gli artefatti artistici quali sprite, animazioni ecc.

In un buon videogioco queste due macro-componenti devono essere in simbiosi e i rispettivi team, o figure professionali, devono essere legati in maniera profonda per riuscire a creare un prodotto che sia eccellente sia dal punto di vista prettamente tecnico ma anche, e soprattutto, da un punto di vista artistico. I videogiochi riescono a creare un connubio perfetto tra il divertimento e quello che può essere un bel film o un libro quindi un prodotto che riesce a trasmettere all'utente un'emozione o raccontare una situazione cercando di rimanere il più possibile interessante e coinvolgente ma, a differenza di quest'ultimi, esso riesce

a sfondare quella barriera che, per ovvi motivi, gli altri prodotti d'intrattenimento non possono, cioè quella dell'interazione. L'intento è quello di porre l'utente come attore principale e attivo, cosa che non riesce a fare un prodotto come un film o un libro in quanto prodotto statico dove l'utente ha un ruolo passivo, seppur coinvolgente, ma pure sempre passivo rispetto al prodotto stesso. In questo caso il videogioco riesce a rompere questa barriera e riesce a trasmettere le stesse sensazioni di un film ma dando la possibilità all'utente, che in questo caso si trasforma in videogiocatore, di prendere delle scelte, scelte che possono portare la trama in una direzione piuttosto che un'altra e successivamente ad uno dei molteplici finali che scaturisce proprio dalle scelte prese in precedenza. Il lavoro che è richiesto da una software house è un lavoro molto difficile ma che può portare ad enormi successi e di conseguenza ad enormi ricavi andando così a perseguire l'obiettivo dell'azienda produttrice. Le due componenti principali devono riuscire a svilupparsi in contemporanea altrimenti si può assistere, come successo in tanti giochi del passato e non, a prodotti che riescono ad eccellere in alcune meccaniche quali la storia, la sceneggiatura o i dialoghi ma peccato in altre componenti quali la grafica, il gameplay, l'emotività dei personaggi o peggio di tutti le prestazioni del gioco stesso obbligando l'utente a non poter giocare per problemi di ottimizzazione. Il videogioco è equilibrio, deve essere un giusto mix tra tecnologia e arte e quando questa fusione riesce nel migliore dei modi si assistono a prodotti di ottima fattura sia narrativa, sia tecnologica che rimangono nella testa dei videogiocatori per anni facendo la storia di questo enorme mondo che è quello videoludico. Di seguito sono elencate e definite in dettaglio tutti gli aspetti chiave di un videogioco che è possibile valutare in un prodotto di qualsiasi dimensione:

- Qualità grafica: consiste nella qualità con la quale è riprodotto tecnicamente l'apparato grafico del videogioco che comprende: qualità e risoluzione delle texture, effetti di luce e shader, fedeltà degli oggetti riprodotti a schermo, qualità dei modelli, quantità di poligoni usati e tutti quelli che sono i tecnicismi dietro alla riproduzione di contenuti multimediali quali i videogames.
- Stile artistico: può essere molto soggettivo ma in generale consiste nello stile artistico scelto per il videogame che può spaziare dall'ultra reale (con paesaggi, persone, oggetti riprodotti in maniera maniacale e il più possibile vicine alla realtà) allo stile 8 bit (stile che richiama molto i videogames datati, appunto sviluppati su architetture ad 8 bit) o a quello più minimale (che consiste nel minor numero di dettagli, colori ed effetti richiamando una grafica più pulita).

- Storia: si valuta quella che è in generale la trama del videogioco, l'originalità e la presenza di momenti memorabili o di colpi di scena.
- Narrazione: consiste nel modo in cui la storia viene raccontata e quindi dei dialoghi, la sceneggiatura e tutte le tecniche che riguardano la narrativa.
- Sviluppo del personaggio: si valuta l'evoluzione del personaggio principale sia in termini statistici quali punti abilità, tecnica, forza, skills ma a volte può comprendere anche l'evoluzione caratteriale del personaggio che deve essere coerente con la storia e gli avvenimenti che avvengono attorno al personaggio stesso.
- Personalizzazione: si valuta il grado di personalizzazione del personaggio: creazione, indumenti, armi, movimenti, approcci al gameplay; ma può comprendere anche la personalizzazione del gioco in termini di editor interni che permettono di creare strutture, movimenti, effetti, oggetti.
- Rigiocabilità: fattore molto importante che valuta l'esperienza di gioco nelle "Run" successiva alla prima o in generale al numero di ore di intrattenimento che il gioco fornisce (con il termine "Run" si intende una sessione di gioco che accompagna l'utente fino alla fine di esso).
- Gameplay: anch'esso è molto soggettivo ma in generale valuta quello che è il "fattore divertimento" di un titolo; il gameplay può essere riassunto come la modalità di gioco oppure come l'effettiva interazione che l'utente ha con il videogames. La valutazione è estremamente collegata alla tipologia di videogioco sviluppato.
- Transizioni tra narrazione e gameplay: valuta la naturalezza (o in caso contrario la forzatura) del collegamento tra momenti di gameplay e momenti di narrazione della storia quali filmati, dialoghi e cutscenes.
- Difficoltà del gioco: valuta il bilanciamento del gameplay e quindi il senso di frustrazione o soddisfazione dell'utente quando si approccia al gameplay.
- Doppiaggio: valuta il livello e la presenza (o meno) dei doppiaggi nelle varie lingue messe a disposizione dal team di sviluppo.
- Musica e suoni: valuta la qualità e la coerenza delle musiche, e dei suoni ambientali in determinati contesti del videogame.
- Senso di immersione: valuta la sensazione di immersione che il titolo deve suscitare nell'utente. Questo fattore è molto soggettivo e spesso è collegato

alla narrazione, alla musica e ai suoni ambientali che cercano di suscitare determinate emozioni al videogiocatore.

- Ottimizzazione: valuta le prestazioni del titolo rapportate all'hardware disponibile. Nei videogames tutto ruota intorno all'FPS (Frames Per Second) che governa la fluidità del gioco (altre ottimizzazioni grafiche impediscono alcuni effetti spiacevoli alla vista come popup, aliasing o tearing). Il numero di frames consigliati per qualsiasi gioco è 60 frames al secondo perché, da determinati studi sull'occhio umano, oltre i 60 fps l'occhio umano incomincia ad avvertire la perfetta fluidità dei movimenti. Nell'ottimizzazione si valuta anche la quantità di risorse utilizzate dal titolo per ottenere la resa grafica voluta.
- Prezzo del gioco: valuta il prezzo del titolo in relazione alla quantità di ore di gioco, alla tipologia del titolo e in generale alle qualità precedentemente citate.
- Compatibilità: valuta il numero di piattaforme per cui il videogame è stato sviluppato ma può anche comprendere la compatibilità con gli strumenti di input e/o output quali tastiere, volanti, joystick, altoparlanti e visori3D.
- Numero di patch e DLC: si valuta il numero e la grandezza, in termini di GB, delle patch (in italiano “Toppa”, software che viene rilasciato gratuitamente per “coprire” eventuali bug o meccaniche poco curate spesso dopo il rilascio ufficiale) e dei DLC (Downloadable content, software che viene rilasciato spesso a pagamento e che mira ad aggiungere nuove funzionalità e/o nuove porzioni di gioco riuscendo così ad allungare la vita dello stesso).

## **Capitolo 2**

# **Fasi di progettazione e sviluppo**

### **2.1 Figure chiavi nello sviluppo**

Lo sviluppatore di videogiochi è colui che realizza applicazioni videoludiche, ovvero software interattivi di intrattenimento. Una società di sviluppo di videogiochi è un gruppo di “sviluppatori” con una comune ragione sociale, che sviluppa tali prodotti. Quando ci si riferisce ad una azienda che realizza software videoludici il termine corretto è “sviluppatore di videogiochi”, mentre la figura del publisher in Italia viene identificata come editore di videogiochi. Nel mercato si aggiunge un terzo fattore economico, chiamato distributore, società che si occupa della distribuzione delle confezioni complete del prodotto all’interno dei propri mercati di riferimento, sia nella GDO sia nei singoli punti vendita. Nell’ambito dei videogiochi, il termine sviluppatore di videogiochi può essere attribuito indistintamente a diverse figure professionali: programmatore, grafico 3D, grafico 2D, illustratore, direttore artistico, autore, caposviluppatore, produttore e project manager.

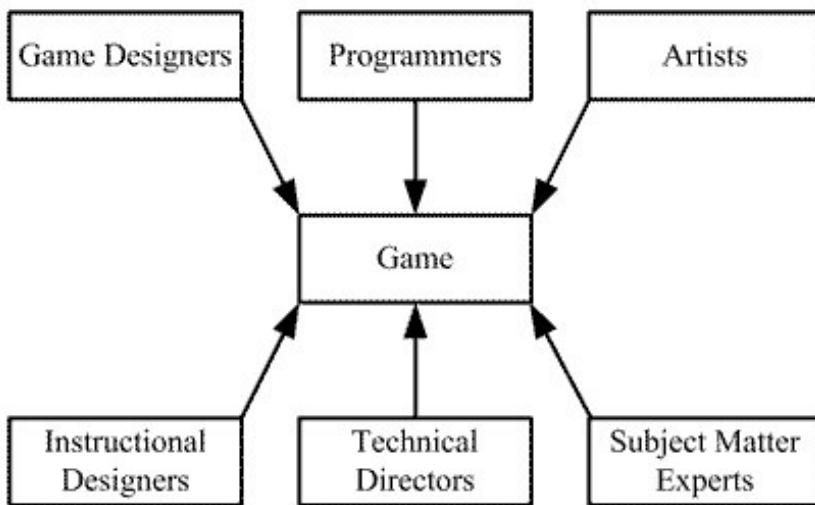


Figura 2.1: Team di sviluppo multidisciplinari

### 2.1.1 Professioni legate al Design

Queste professioni non sono strettamente legate alla scrittura di codice complesso o alla progettazione delle componenti (classi, script, documenti) di gioco ma si interessano più alla parte tecnico/artistica, che il titolo deve ottenere, grazie all’esperienza videoludica accumulata negli anni. L’esempio più “famoso” nell’ambito videoludico è il Game Designer che effettua le scelte sulle meccaniche e sulla tipologia di gioco da sviluppare più che dedicarsi ad implementare tali scelte. Analizzando nello specifico le diverse figure che sono legate al Design ritroviamo:

- **Game Designer:** Il game Designer è la figura principalmente responsabile del gameplay e del “fattore divertimento” del gioco. Sfruttando la propria esperienza e le proprie abilità, deve creare il miglior gioco sulla base di determinate circostanze, come la piattaforma, il genere e il pubblico. Comincia scrivendo e diagrammando il progetto all’interno di un documento di Design, utilizzando strumenti come screenshot e diagrammi d’interfaccia, tabelle e modelli di script. Nel corso dello sviluppo, il game Designer aggiorna i documenti in modo che il resto del gruppo sia sempre a conoscenza dell’attuale stato del gioco. Il documento di Design, tuttavia, non resta solo un’idea iniziale: nel corso del progetto potrebbe essere soggetto a svariate modifiche e aggiunte, come nuovi personaggi, mondi, schemi di

controllo, sistemi, interfaccia, trama ed enigmi. Intanto che il progetto su documento prende forma, è buona prassi per un game Designer giocarlo costantemente, in modo da assicurarsi che vi sia il giusto livello di bilanciamento, difficoltà e divertimento. informato costantemente dei risultati di playtesting, così da capire in quali aree il gioco richiede particolari attenzioni. Il game Designer, oltre a lavorare assieme a un gruppo di altri Designer, collabora strettamente con le altre aree del team, in modo da supervisionare tutti gli elementi sviluppati. Non è raro che un progetto abbia più di un game Designer, i quali si dividono le responsabilità in base alla loro esperienza e ai propri interessi.

- **Lead Designer:** Un lead Designer esegue molti degli incarichi menzionati per il game Designer ma soprattutto rappresenta il punto di riferimento per gli altri membri del gruppo di Design e del progetto per quanto concerne la sua area, in modo da raggiungere gli obiettivi di produzione prefissati. Assieme al producer, ha potere decisionale riguardo alle scelte di Design, specialmente nei casi in cui queste non implichino nessun cambiamento radicale negli obiettivi e tempi di sviluppo. Il lead Designer è responsabile della selezione degli altri Designer del team e talvolta si occupa di presentare il gioco ai media.
- **Level Designer:** Nello sviluppo di un gioco 3D, un level Designer si occupa di costruire l'architettura interattiva (strutture e terreno naturale) per un segmento del gioco. Ciò significa che egli implementerà specifici aspetti del gameplay in una determinata parte del progetto, dal momento che molti giochi tridimensionali usano la struttura del mondo come base del gameplay. L'equivalente del level Designer di un gioco che non si occupa dell'interazione tra spazi 3D è il mission (o campaign) Designer. Tale ruolo è appropriato a generi videoludici come GdR o RTS, in cui i Designer usano degli editor specifici per posizionare terreno e risorse. In ogni tipo di gioco, il level Design può includere obiettivi, abilità e comportamenti dei nemici.
- **Sceneggiatore:** Gli sceneggiatori o screenwriter o fiction writer ricercano e creano la trama (quando presente) alla base del mondo di gioco, presentata attraverso il testo su schermo, i dialoghi dei personaggi e le scene d'intermezzo. In base al tipo di gioco, la realizzazione può essere tanto basilare (semplice testo su menu, voce fuoricampo) quanto complessa (conversazioni a bivi). Gli sceneggiatori lavorano a stretto contatto con i Designer in modo da comprendere e proporre ogni possibile percorso attraverso il gioco. Collaborare con i Designer è inoltre cruciale per riuscire

a mantenere il giusto bilanciamento tra meccaniche e trama, attraverso ciascun potenziale percorso del gioco. Dal momento che i propositi e lo stile variano ampiamente da gioco a gioco, il contratto di uno Screenwriter è spesso limitato a una sola opera.

### **2.1.2 Professioni legate alla programmazione**

Un videogioco, in quanto prodotto complesso, ha bisogno di una più ampia gamma di programmatore che devono rendere concreto il lavoro effettuato al livello superiore. In questa parte troviamo molte tipologie di programmatore, in quanto il videogioco presenta al suo interno una moltitudine di conoscenze e tecnologie che devono sapersi collegare in maniera perfetta affinchè il tutto funzioni, che vanno a ricoprire tutte le necessità di implementazione scritte nei documenti di progetto (Audio, Grafica, Interazione, AI ecc). Andando ad analizzare tali figure troviamo:

- Programmatore: Il programmatore è chi scrive il codice macchina del videogioco in sviluppo. A seconda dell'anzianità e delle conoscenze, viene classificato come Junior o Senior della propria area di competenza. Agli inizi della propria attività lavorativa all'interno di uno studio di sviluppo, deve apprendere le regole di programmazione degli strumenti di lavoro utilizzati dall'azienda stessa. Nello sviluppo su console, è richiesta una conoscenza di linguaggi di programmazione come C/C++. Di norma, il programmatore, in base al suo percorso di studi e alle capacità personali, si specializza in diversi ambiti quali l'IA, gli effetti grafici, la fisica, la programmazione del suono, i sistemi di collisione e molte altre aree dello sviluppo.
- Capo programmatore: La figura del capo programmatore (lead programmer) unisce incarichi prettamente manageriali con gli impegni di un tradizionale programmatore. Il suo compito non sarà limitato alla scrittura di codici complessi, ma egli dovrà anche disporre del know-how necessario all'organizzazione e alla gestione di un team, poiché rappresenta un punto di congiunzione tra il Producer e il team di programmazione. Il capo programmatore dovrà guidare il gruppo nella scelta della tecnologia da usare (es. Direct3D o OpenGL) e decidere quali incarichi affidare a ciascun elemento. Oltre a supervisionare il lavoro del team e a contribuire enormemente al processo di programmazione, egli necessita di lavorare a stretto contatto con il team artistico e di Design, partecipando alla defi-

nizione delle milestone. Non di rado, il capo programmatore è un esperto in una o più specialità di programmazione, come intelligenza artificiale, rendering 3D, animazione 3D, fisica, multiplayer/networking o audio. Ovviamente, la posizione ricoperta lo porta a essere responsabile dell'intera struttura e implementazione del codice nel gioco.

- Programmatore di engine e tool: Un programmatore di engine (engine programmer) si occupa di realizzare le fondamenta del codice alla base del gioco, scrivendo il codice di programmazione dietro al rendering e alle funzionalità dello stesso. Per gran parte delle piattaforme di gioco, all'engine programmer è richiesta la conoscenza del linguaggio C/C++, talvolta di Assembly, di concetti matematici, grafici, rilevamento di collisioni e gestione di database. Un programmatore di tool (tools programmer) crea invece gli strumenti in grado di agevolare il lavoro degli artisti e dei Designer con l'engine (come plug-in per software di grafica in grado di aiutare a integrare texture o sfondi nel gioco). Migliore è lo strumento, più rapidamente possono lavorare Designer e artisti, velocizzando sensibilmente i tempi di produzione.
- Programmatore di grafica ed effetti speciali: Generalmente, a tutti i programmatori è richiesta una minima esperienza nella programmazione di grafica, ma nel caso del programmatore di grafica (graphics/special effects programmer) è essenziale che egli disponga di tutte le conoscenze tecniche per realizzare oggetti tridimensionali oltre a immagini in 2D. Un'enorme cultura matematica (specialmente riguardante l'algebra lineare e i calcoli avanzati) è un'abilità cruciale per questo ruolo. Come programmatore grafico c'è bisogno di comprendere le complessità dello skinning (il coprire) modelli tridimensionali, dell'importare file da programmi di animazione 3D e dell'unione delle animazioni. Tuttavia, egli deve anche possedere l'occhio artistico che gli permetta di implementare realistici e affascinanti effetti particolari (come fuoco ed elettricità), o almeno l'abilità di lavorare con il team di artisti per raggiungere l'effetto desiderato. Inoltre, è necessario che sia in grado di ottimizzare i suoi lavori, in modo da capire come visualizzare tutto in tempo reale con il più alto frame rate.
- Programmatore AI: Un programmatore di intelligenza artificiale (artificial intelligence programmer) scrive essenzialmente le regole che governano il comportamento delle entità all'interno del gioco. Un AI programmer alle prime armi deve apprendere gli algoritmi di base e i concetti dietro l'IA, come il path finding, i pattern e gli alberi di decisione. Tra i concetti più

avanzati nel campo dell'IA figurano le reti neurali, l'A-Life, gli algoritmi genetici e le macchine a stati finiti. L'abilità principale dell'AI programmer sta nell'implementare complessi comportamenti di gioco, che funzionino in tempo reale senza eccessivi sforzi del processore.

- Programmatore di rete/multiplayer: Quella di programmatore multiplayer (multiplayer/networking programmer) è una tra le figure più richieste in un team di sviluppo, anche grazie al recente successo dei giochi online. I giochi online hanno come unica sfida l'essere basati su un'enorme lista di variabili: tra queste figurano la potenza del sistema dell'utente, le capacità in continuo cambiamento delle reti, l'architettura dei server di gioco, il sistema di pagamento, la gestione dell'esperienza in-game e alcuni sistemi di sicurezza. I giochi online sono minacciati da gruppi di hacker, che possono drasticamente influenzare la soddisfazione e gratificazione dell'utente. Dal momento che questo tipo di giochi offre denaro in base alla quantità di tempo che i giocatori spendono online, il programmatore di rete è responsabile del successo a lungo termine o del fallimento di un prodotto e, talvolta, di quello dell'intera compagnia. Oltre alle conoscenze sopraccitate, egli deve disporre di un background composto dai campi standard della programmazione: architettura client/server, sicurezza di rete, protocolli di base (es. TCP/IP o UDP), sincronizzazione, creazione e gestione di database e interfacce di rete come DirectPlay e Winsock.

## 2.2 Fasi di progettazione

La creazione di videogame è un processo assai complesso che coinvolge diverse figure professionali nell'ambito del Design, del marketing e ovviamente della programmazione in codice. Per riuscire a realizzare un prodotto che sia allo stesso tempo valido dal punto di vista tecnologico e appetibile sotto il profilo strettamente commerciale, è fondamentale fare in modo che tutte queste professionalità collaborino in maniera coesa e coordinata andando a creare un team multidisciplinare. In tal senso è buona norma predisporre un piano di lavoro ben definito a partire dalla fase di progettazione che può essere suddivisa in diversi "momenti". Progettazione e creazione di videogame in 4 fasi:

1. La prima fase progettuale per la creazione di videogame consiste nella definizione dell'idea, ovvero del Concept su cui si realizzerà il videogioco. Si tratta di un momento molto importante in cui si mettono sul tavolo diverse opzioni narrative, diverse storie da "raccontare" potenzialmente

appetibili per il pubblico che si vuole intercettare. In tal senso è fondamentale in questa fase l'apporto di copywriter ed esperti di marketing e comunicazione in grado di leggere le “pulsioni” del mercato e individuare la storia giusta da raccontare.

2. La seconda fase di progettazione è quella dello Storyboard, ovvero la creazione di una sequenza di disegni che mostrano i livelli del gioco e le diverse scene con gli obiettivi da raggiungere per il giocatore. Ogni “tavola” dello Storyboard deve essere accompagnata da un paragrafo o due in cui si descrive in modo non troppo dettagliato ciò che avviene ad ogni livello del gioco. In questo fase illustratori, Designer e copywriter lavorano a braccetto per una rappresentazione coerente e sensata dell’idea di partenza.
3. Arriva quindi il momento in cui si scende nei particolari della “storia”. Così, dopo aver creato lo Storyboard, si iniziano a descrivere tutti i dettagli sia a livello di Design che di narrazione (i due aspetti a questo punto sono strettamente interconnessi tra loro). Il processo creativo dunque si complica perché occorre pensare ad ogni possibile variabile ed opzione che possa intervenire nel gioco. Si tratta di una fase importantissima in cui deve emergere la qualità del prodotto finale.
4. Infine la progettazione si conclude con un documento finale in cui si registrano e si elaborano tutte le idee prodotte in un formato che assomiglierà molto a una sceneggiatura cinematografica. La creazione di questo script è un passo conclusivo fondamentale che offre tuttavia ai progettisti la possibilità di correggere alcune imperfezioni o talvolta anche di cambiare idea su alcune scelte prese in precedenza. In quest’ultimo caso si procederà a una revisione complessiva del progetto.

Una volta conclusasi la fase progettuale si passerà alla creazione del videogame vero e proprio e in questo caso entreranno in gioco le figure tecniche precedentemente descritte e nella maggior parte dei casi lo sviluppo avviene con metodologie denominate “Agili”.

## 2.3 Metodologie di sviluppo

Una metodologia di sviluppo software o una metodologia di sviluppo di sistemi ingegneristici software è un framework utilizzato per strutturare, pianificare e controllare il processo di sviluppo di un sistema informativo. Esistono molte

plici metodologie che si differenziano per l'approccio allo sviluppo e la quantità di documentazione:

- Agile Software Development
- Crystal Methods
- Dynamic Systems Development Model (DSDM)
- Extreme Programming (XP)
- Feature Driven Development (FDD)
- Joint Application Development (JAD)
- Lean Development (LD)
- Rapid Application Development (RAD)
- Rational Unified Process (RUP)
- Scrum
- Spiral
- Systems Development Life Cycle (SDLC)
- Waterfall (a.k.a. Traditional)

### **2.3.1 Metodologie Agili**

Lo sviluppo di software agile è una struttura concettuale per intraprendere progetti di ingegneria del software. Esistono numerose metodologie di sviluppo software.

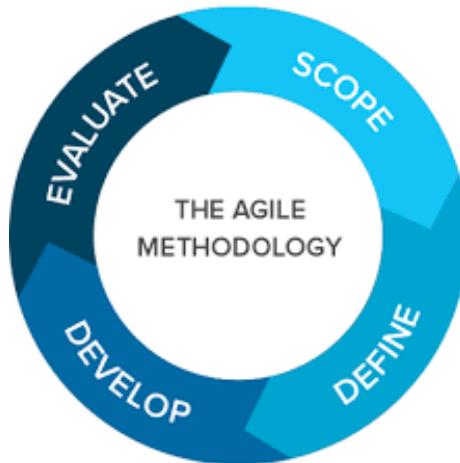


Figura 2.2: Metodologie agili

In generale le metodologie agili tentano di minimizzare il rischio sviluppando software in intervalli di tempo brevi, chiamati iterazioni, che in genere durano da una a quattro settimane. Ogni iterazione è come un proprio progetto software in miniatura e include tutte le attività necessarie per rilasciare il mini-incremento di nuove funzionalità: pianificazione, analisi dei requisiti, progettazione, codifica, test e documentazione. In altre metodologie l'iterazione potrebbe non aggiungere abbastanza funzionalità per garantire il rilascio del prodotto, mentre un progetto software agile vuole essere in grado di rilasciare un nuovo software al termine di ogni iterazione. Alla fine di ogni iterazione, il team rivaluta le priorità del progetto e ricalcola il percorso ottimizzato da effettuare che porterà alla prossima iterazione. I metodi agili enfatizzano la comunicazione in tempo reale, preferibilmente faccia a faccia, rispetto ai documenti scritti. I team agili organizzano spesso dei briefing (brief meeting) che includono tutte le persone necessarie per completare il software. Come minimo, questo include i programmati e le persone che definiscono il prodotto come i product manager, gli analisti di business oppure i clienti effettivi. Il briefing può anche includere tester, progettisti di interfacce, scrittori tecnici e management. Le metodologie Agili pongono il lavoro sul software come principale misura di progresso, combinato con la preferenza della comunicazione faccia a faccia, i metodi agili producono pochissima documentazione scritta rispetto ad altre metodologie. XP è una metodologia per la creazione di software in un ambiente molto instabile (dove le persone coinvolte nel progetto possono variare in numero, esperienza e skill). Permette la flessibilità all'interno del processo di modellazione. L'obiettivo principale di XP è ridurre il costo di cambiamento dei requisiti software. Con le metodologie

di sviluppo tradizionali, come la metodologia Waterfall, i requisiti per il sistema sono determinati e spesso “congelati” all’inizio del progetto di sviluppo. Ciò significa che il costo di cambiamento dei requisiti in una fase successiva del progetto, che è qualcosa di molto comune nel mondo reale, può risultare elevato. I concetti fondamentali dell’ Extreme Programming, descritte nella prima edizione di “Extreme Programming Explained” possono essere raggruppati in punti elencati come segue:

- Sviluppo guidato dai test
- Pianificazione contenuta
- Lavoro di tutto il team
- Programmazione a coppia
- Integrazione continua
- Miglioramento della progettazione
- Rilasci contenuti
- Conoscenza condivisa
- Design semplice
- Responsabilità del codice collettivo
- Standard di codifica o convenzioni di codifica
- Andatura sostenibile (vale a dire quaranta ore settimanali)

Nella seconda edizione di “Extreme Programming Explained” sono elencate una serie di pratiche di corollario in aggiunta alle pratiche primarie. I principi di base derivano da buoni principi, generalmente accettati nell’ambito di sviluppo software, che sono portati agli estremi (da questo il nome Extreme):

- L’interazione tra sviluppatori e clienti è buona norma. Pertanto, un team di XP dovrebbe avere un cliente nel proprio team, che specifica e dà la priorità al lavoro per il team e che può rispondere alle domande non appena si presentano.
- Se l’apprendimento è buono, portalo agli estremi: ridurre la durata dei cicli di sviluppo e feedback. Eseguire test più frequenti.

- Il codice semplice ha più probabilità di funzionare. Pertanto, i programmati XP scrivono il codice solo per soddisfare le esigenze attualmente presenti nel progetto e fanno di tutto per ridurre la complessità e la duplicazione nel loro codice tramite refactoring continuo.
- Il codice semplice è buona norma, riscrivi il codice quando diventa complesso.
- Revisionare il codice è buona norma. Pertanto i programmati XP lavorano in coppia, condividendo uno schermo e una tastiera (che migliora anche la comunicazione) in modo che tutto il codice venga revisionato come minimo da due persone.
- Testare il codice è buona norma. Pertanto, in XP, i test vengono scritti prima della scrittura del codice. Il codice è considerato completo quando supera i test (ma poi ha bisogno di refactoring per rimuovere la complessità). Il sistema viene periodicamente o immediatamente testato utilizzando tutti i test automatici pre-esistenti per garantire che funzioni. Sviluppo basato sui test.

Si pensava che la programmazione estrema potesse funzionare solo in piccoli gruppi con meno di 10-15 persone. Tuttavia, XP, Scrum e le metodologie agili sono tutt'oggi le metodologie più utilizzate per lo sviluppo di videogames da parte di grandi aziende di sviluppo con più di 1000 dipendenti tra cui anche EA Sports, Ubisoft, Konami.

### 2.3.2 Game-Scrum

L'uso di metodologie agili per lo sviluppo di giochi è diventato molto comune negli ultimi anni.

## 2. Fasi di progettazione e sviluppo



Figura 2.3: Opportunità lavorative EA Sports Career

Tuttavia, tali metodologie devono essere adattate alla realtà del team e alle particolarità dello sviluppo di un videogioco. Poiché esistono poche metodologie agili che affrontano in modo specifico i problemi riscontrati nello sviluppo di videogiochi, questa parte analizza una metodologia nata proprio per far fronte a queste piccole incongruenze e pensata apposta per lo sviluppo di videogiochi. La metodologia è chiamata Game-Scrum. I recenti progressi nella tecnologia hanno permesso lo sviluppo di giochi sempre più complessi e realistici, ma il costo di produzione di un gioco di alto livello ha raggiunto i milioni di dollari, lasciando gli editori sempre più avversi al rischio. Ciò creò una domanda nell'uso di tecniche e metodi per garantire che un gioco possa essere sviluppato con il minor numero possibile di ritardi ed errori. Tecniche e metodi dovrebbero anche facilitare il processo di scoperta del "divertimento", un fattore importante per il successo di un gioco. Sebbene alcune persone possano già avere conoscenze nello sviluppo di software, ci sono caratteristiche specifiche dello sviluppo dei videogiochi che possono intaccare il successo di grandi giochi, ciò si traduce in gravi problemi nella gestione dei progetti, aumentando i costi già elevati e causando ritardi. L'uso di una metodologia incentrata sullo sviluppo del gioco può aiutare a evitare questi problemi. L'uso di metodologie iterative sta aumentando in questo settore e le metodologie agili sono diventate popolari tra lo sviluppo di videogame. Sfortunatamente, ci sono pochi esempi di metodologie agili orientate allo sviluppo del gioco, e quelle precedenti non sono state efficaci nella pratica. L'obiettivo del Game-Scrum è presentare un approccio di metodologia agile basata su Scrum e Programmazione eXtreme per lo sviluppo di giochi e rivedere le due metodologie agili esistenti specializzandole nell'ambito dei videogame. Le principali caratteristiche delle metodologie agili sono: coo-

perazione, semplicità, adattabilità ed essere incrementali. All'interno di queste caratteristiche, il Manifesto Agile riunisce i valori principali di varie metodologie agili, che sono importanti per comprendere l'approccio allo sviluppo di videogiochi basato su una metodologia agile. Attraverso questi valori possiamo notare che l'attenzione dei metodi agili è sul prodotto, piuttosto che sul processo utilizzato. Quindi evitiamo la documentazione non necessaria, c'è la necessità di accettare e reagire ai cambiamenti (essere flessibili), e anche la collaborazione con il cliente nel monitoraggio dello sviluppo. Uno dei più noti sviluppi del software agile è l'eXtreme Programming (o XP), che viene proposto da Beck nel 1999 e le cui convinzioni fondamentali si basano su cinque principi [Teles 2006]: comunicazione, coraggio, feedback, rispetto e semplicità. Le sue pratiche sono state progettate per soddisfare questi valori, con completa libertà di usarli o meno. Chi definisce cosa e come le pratiche saranno applicate nel progetto è il coach, nonché capo del progetto; è sua la responsabilità di aiutare a mantenere le dinamiche (tempi e consegne) e di facilitare il processo per il team, fornendo risorse e attività di coordinamento, assicurando così che il suo team applichi correttamente le pratiche definite. Scrum [Schwaber e Beedle 2001] è un'altra metodologia agile anch'essa basata su brevi iterazioni chiamate sprint. Negli sprint i backlog devono essere approvati, essi sono una serie di requisiti atomici progettati per essere completati ad ogni iterazione. I backlog dovrebbero essere progettati secondo le priorità del cliente, costi di produzione, rischio e conoscenza necessari per produrlo o altri parametri. Lo Scrum Master è visto come un leader che aiuta il Team, che è responsabile dello sviluppo. C'è anche il Product Owner, che dà il suo feedback al progetto. Le iterazioni sono accompagnate da quattro tipi di incontri: uno per la pianificazione, uno per l'accompagnamento, uno per rivedere ciò che è stato fatto e uno per ispezionare il processo, le persone e gli strumenti. Scoprendo presto cosa nel gioco fornirà il fattore "divertimento" si consente al team di concentrarsi sullo sviluppo e sul miglioramento di una parte molto più ampia del progetto, aumentando notevolmente la probabilità di successo del gioco. Una scelta utile per definire, in maniera rapida, il fattore di divertimento è adottare metodologie iterative. Queste metodologie consentono di ricevere un feedback iniziale delle funzionalità implementate, migliorandole se necessario in modo facile. Un altro vantaggio è quello di facilitare la comunicazione e la cooperazione tra tutti coloro che sono coinvolti nella creazione del gioco. I problemi riscontrati nello sviluppo del software in generale sono ben noti, ma si sa molto poco dei veri problemi che riguardano l'industria del videogioco. Secondo Fabio Petrillo, professore specializzato in approcci di sviluppo di software videoludico al Dipartimento di Computer and Software Engineering

del politecnico di Montréal, i problemi principali sono da attribuire alla scelta dell'ambito, pianificazione e crunch time (termine usato per periodi di sovraccarico di lavoro estremo, che si verificano di solito più vicino alle scadenze). Dopo una analisi su vari team di sviluppo e prodotti videoludici di varie dimensioni, è stato scoperto che la grande maggioranza dei problemi incontrati era legata a problemi di gestione. Parte di questi problemi sono spesso associati alla presenza di un team multidisciplinare. Sebbene la multidisciplinarità crei un ambiente che incoraggia la creatività, potrebbe finire per creare una divisione tra “gli artisti” e “gli sviluppatori”, con conseguente difficoltà a ottenere una comunicazione efficace tra tutta la squadra. Un altro punto è la difficoltà nel definire elementi come il divertimento del gioco, che non ha un metro di giudizio oggettivo e rende difficile capire quando questo obiettivo è raggiunto o meno [Petrillo et al. 2008]. Alcune tecniche per risolvere questi problemi sono suggerite da Kanode e Haddad [Kanode e Haddad 2009], come l'uso di metodi iterativi, la costruzione di prototipi e la creazione di una pipeline per gli artefatti multimediali. Una parte dei problemi comunemente riscontrati può essere risolta applicando la metodologia Game-Scrum. Si trova davvero poco materiale relativo alle metodologie agili specificamente focalizzate sullo sviluppo di videogame. La maggior parte delle iniziative esistenti utilizza alcune derivazioni del metodo waterfall [Flood 2003] o, più recentemente, alcune derivazioni del metodo iterativo [Gregory 2008]. Alcune proposte che coinvolgono metodologie agili esistenti includono lo sviluppo di giochi eXtreme [Demachy 2003] e Game Unified Process [Flood 2003]. L'Extreme Game Development [Demachy 2003], o XGD, è un adattamento di XP per lo sviluppo di videogiochi. Secondo Demachy, il suo focus è su come adattare XP alla progettazione e alla creazione di videogiochi e/o contenuti multimediali e come testare automaticamente elementi specifici del gioco, come il fattore “divertimento”. Alcuni adattamenti di XP includono l'aggiunta di contenuti multimediali in continua integrazione; fare test per i contenuti multimediali; considerare il team nel suo complesso e usare UML per descrivere gli elementi del game Design, l'interazione e la comunicazione tra game Designer, programmatore e persino artisti; La metodologia chiarisce alcuni adattamenti da XP a XGD ma non affronta le problematiche di lavorare in team multidisciplinari. Consiglia solo la visione del team nel suo complesso e suggerisce il lavoro appaiato per artisti come la programmazione in coppia che è utilizzata dagli sviluppatori per ottenere il medesimo risultato. Inoltre, non discute i risultati dell'applicazione della metodologia nei progetti menzionati. Game Unified Process Game [Flood 2003] è una metodologia che combina il RUP (Rational Unified Process) alle tecniche di XP ed è stato

sviluppato principalmente per affrontare problemi di sviluppo evidenziati dall'utilizzo del modello WaterFall. Sebbene sia considerato un processo pesante, RUP adotta una metodologia che permette di apportare un minor numero di modifiche nel software, mentre l'XP offre modi pratici per evitare un'ulteriore documentazione. Delle conclusioni che Flood ha ottenuto dall'applicazione della sua metodologia, se ne sottolineano due: l'attenzione su un ciclo rapido di XP con un focus sui lunghi cicli di RUP che ha portato benefici al team di sviluppo, nonostante le difficoltà della squadra, al fine di adottare lo stile di sviluppo di una metodologia agile; il riconoscimento che la creazione di contenuti nello sviluppo del gioco è iterativo, che porta Flood a suggerire questa metodologia anche per artisti e Designer. Nonostante il feedback positivo, i risultati del team comparati con il metodo waterfall, non viene raccomandato come metodo per lo sviluppo del gioco, come commentato da Flood nella sua proposta. Inoltre, non è stato discusso come gestire gli artisti nelle iterazioni. Scrum è un framework incentrato sulla gestione progettuale cioè come dividere e coordinare i tasks in modo che tutto possa essere fatto senza impedimenti, in base al quale è possibile utilizzare qualsiasi altra pratica agile. Da questo punto di vista, XP sarebbe più focalizzato sull'ingegneria del progetto e su quali tecniche sono le migliori per completare le attività in modo efficiente. Game-Scrum utilizza queste due metodologie come base, adattandole all'esperienza di professionisti e concentrandosi su persone con poca o nessuna esperienza nello sviluppo del videogioco. La gestione del progetto è un importante punto nello sviluppo del videogioco, è importante che il team capisca le iterazioni e gli incontri di Scrum, questi dettagli non saranno trattati approfonditamente in questa parte, ma rimangono importanti per il successo del progetto. Durante le fasi di sviluppo di una metodologia agile, tutte le iterazioni hanno praticamente gli stessi passaggi fondamentali da effettuare per lo sviluppo di un software. Ma secondo Keith [Keith 2010], la distribuzione del lavoro nello sviluppo di un videogioco non è equamente distribuita durante le sue iterazioni. Per supportare questa realtà, GameScrum è diviso in tre fasi.



Figura 2.4: Step e SprintCycle Game-Scrum

*Pre-produzione:* In questa fase il gioco deve essere “scoperto”, cioè quali sono realmente gli obiettivi del gioco e quale sarebbe il fattore “divertimento” in esso. Qui il Concept (meccaniche e idee generali del gioco) sarà migliorato, si sceglie il linguaggio di programmazione, la piattaforma e vengono effettuare altre scelte sugli strumenti da utilizzare. Come Kanode e Haddad affermano [Kanode e Haddad 2009], “una buona fase di pre-produzione riduce la necessità di trovare quell’elemento sfuggente di “divertimento” durante la fase di produzione, e consente al team di concentrarsi sull’implementazione del gioco, piuttosto che sperimentarlo”. Questa fase definisce la direzione che prenderà la fase di produzione, senza soffocare la creatività che potrebbe emergere lì. L’obiettivo sarà quello di trovare il Concept e il Design ideale per il gioco, spesso attraverso tentativi ed errori. Affinché ciò avvenga viene raccomandato il brainstorming per sviluppare idee e crearne di nuove (Sinteticamente consiste, dato un problema, organizzare una riunione in cui ogni membro del team propone liberamente soluzioni di ogni tipo al problema, senza che nessuna di esse venga minimamente censurata; la critica ed eventuale selezione interverrà solo in un secondo tempo, terminata la seduta di brainstorming). Lo sviluppo di un prototipo aiuterà anche ad avere un’anteprima del fattore “divertimento” che il gioco o una parte di esso può offrire. Il prototipo deve fornire una navigazione di base e semplificata per l’utente e le caratteristiche richieste per effettuare test. Di solito, il codice prodotto non viene più utilizzato e il prototipo viene scartato. La creazione del documento di Design del gioco è un passo importante da compiere nella fase di pre-produzione, essendo responsabile della guida al Concept del progetto e dell’intero sviluppo e test del gioco. Un gioco povero di documento di Design potrebbe comportare incertezze sullo sviluppo delle funzionalità e, di conseguenza, potrebbero verificarsi anche ritardi e perdita di milestones. Tutt’ora non esiste uno standard per la creazione del documento di Design, Schuytema [Schuytema 2008] afferma che il documento deve avere una descrizione comprensiva e accurata di tutti gli aspetti del gioco oltre a descrivere

gli oggetti, le interazioni e i personaggi nel gioco, dovrebbe essere documentato non solo quello che fanno, ma anche ciò che influenzano, come interagiscono e il loro ruolo e comportamento nel gioco. Nonostante gli sforzi per rendere il documento abbastanza completo, il documento potrebbe comunque cambiare quindi anche in questo caso non rimane congelato. Però, si dovrebbero valutare i rischi dei cambiamenti e se le scadenze possono ancora essere soddisfatte. Poiché questo documento verrà successivamente tradotto in un Backlog in fase di produzione, per i giochi di piccole dimensioni può essere facoltativo, traducendo i requisiti direttamente come Backlog. Ciò potrebbe far risparmiare tempo al team man mano che procede più rapidamente alla produzione, ma potrebbe anche aumentare i rischi di implementare funzionalità scadenti o rendere il gioco meno interessante.

*Produzione:* In questa fase, si dovrebbe già aver un percorso ben definito del progetto, e quindi una buona idea di cosa sia realmente il gioco e cosa dovrebbe effettivamente essere fatto. Qui il documento di progettazione del gioco deve essere tradotto in un documento di Backlog. E ad ogni iterazione i più importanti backlog rimanenti dovrebbero essere divisi in pezzi più piccoli e avere i relativi compiti definiti. Per i team composti da molti artisti, è necessario pensare in un approccio migliore. Molto spesso un task artistico è svolto da diversi specialisti che lavorano in una sorta di linea di produzione. Ad esempio, il modellatore consegna il suo lavoro all'animatore, che lo passa al sound designer e così via. Quindi si producono degli artefatti che oltrepassano una sorta di catena di montaggio. Keith [Keith 2010] suggerisce l'uso di Kanban (Kanban è un approccio al cambiamento del processo per le organizzazioni che utilizzano una visualizzazione con un kanban board, permettendo una migliore comprensione del lavoro e del workflow) per i responsabili della creazione di contenuti artistici di gioco. Questo permette che, alla fine di uno sprint, ci sia ancora del lavoro in fase di sviluppo, a differenza di Scrum, che richiede che tutto il lavoro venga eseguito alla fine di un'iterazione.

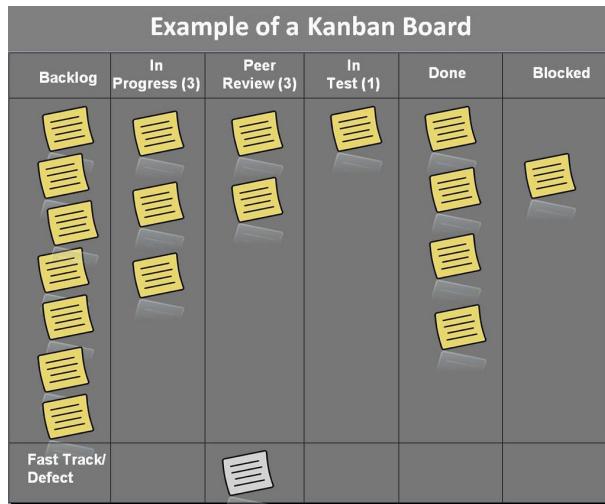


Figura 2.5: Esempio Kanban Board

L’obiettivo è ottenere una creazione di contenuti continua. Un’altra pratica suggerita da Keith è il time-boxing, che limita il tempo a disposizione per completare un compito particolare, tenendo conto dell’importanza del manufatto prodotto e della sua utilità nel gioco. Per i team composti da soli sviluppatori o con pochi artisti si consiglia di applicare le tecniche di gestione Scrum e XP che meglio si adattano alla realtà del team.

*Post-produzione:* Una volta completato il gioco, il playtest aiuta a garantire la qualità e il divertimento del titolo. Ma il focus della metodologia rimane il concentrarsi sul feedback fornito dall’intero processo e della creazione di un post-mortem (esperienza/feedback acquisito durante il lasso di tempo dopo il rilascio del titolo). Le considerazioni postmortem sono importanti perché consentono di conoscere i punti di forza e di debolezza del processo di sviluppo utilizzato, i problemi verificatisi e i suggerimenti per migliorare il processo. Attraverso questo feedback si può ottenere una stima migliore per i progetti futuri e si potrebbero apportare le modifiche necessarie al processo con maggiore esperienza. Myllyaho [Myllyaho et al. 2004] descrive i benefici e i vantaggi dell’analisi postmortems e parla anche dei postmortem disponibili nel sito web di Gamasutra (<http://www.gamasutra.com/>). Come descritto nel loro lavoro, di solito i postmortems forniscono una sintesi della descrizione del progetto, identificando le lezioni apprese nei tre aspetti.

- Il buono, contenente un elenco di buone soluzioni, miglioramenti, strumenti adeguati comprese le aree di gestione dei progetti, pratiche di co-

municazione, marketing e ingegneria;

- Il cattivo, con una lista di problemi e problematiche chiavi e la causa principale della loro presenza all'interno del processo;
- Il brutto, contenente una lista di questioni critiche che devono assolutamente essere corrette.

Inoltre, i postmortem in genere includono le seguenti metriche di progetto di gioco oltre alle esperienze apprese: editore e sviluppatore, numero di sviluppatori a tempo pieno, part-time sviluppatori e contractors; durata dello sviluppo e data di rilascio; piattaforme di gioco e hardware e software di sviluppo usato. Sebbene l'approccio proposto da Game-Scrum non porti nuovi elementi allo sviluppo del gioco, la sua innovazione consiste nell'unire insieme diversi suggerimenti con lo stesso obiettivo. Questo permette di creare un metodo sistematico per sviluppare videogioco, avendo non solo riferimenti pratici da parte di coloro che lavorano nell'area, ma anche il supporto di ricercatori che confrontano le esperienze con le conoscenze disponibili nelle documentazioni.

*Esempio di applicazione Game-Scrum:* in questa sezione verrà descritto il lavoro dei ragazzi della facoltà di Scienze Matematiche e Informatiche di São Carlos (SP), in Brasile, che hanno applicato il Game-Scrum per lo sviluppo di EngGame, un gioco educativo composto da diversi mini giochi nel campo dell'ingegneria del software, scrivendo un articolo che riportasse tutte le informazioni sulla metodologia. Il primo di questi mini giochi, chiamato EngReq, mira ad aiutare a conoscere il documento contenente i requisiti per gli studenti di ingegneria del software dell'omonima facoltà. Sebbene non funzionasse abbastanza, l'idea di EngReq esisteva già, ma era necessario sviluppare il gioco da quell'idea. Di seguito vengono riportate gli step della metodologia:

*La pre-produzione:* nel primo brainstorming è stato discusso l'idea di presentare diversi mini giochi a tema differente per le diverse età di videogiocatore, ognuna delle quali affronta una parte specifica o un concetto dell'ingegneria del software. EngReq si ambienta nell'età della pietra, e l'obiettivo del giocatore sarebbe quello di aiutare le persone della sua comunità a raccogliere i requisiti necessari per un particolare compito che la comunità desidera concludere e rifiutarli se non sono un requisito valido. Successivamente è stato realizzato un prototipo per testare le idee discusse durante il brainstorming, e se fosse stato possibile creare un gioco sull'ingegneria del software che non avvenga all'interno di un ufficio. Nel prototipo, è stato creato un disegno che rappresenti l'idea dello scenario del gioco, e alcune transizioni dello schermo e la presentazione

di requisiti per il giocatore. Si è scoperto che, sebbene all'inizio sembrasse una buona idea, l'elaborazione dei compiti con requisiti simili di un sistema software nell'età della pietra erano molto difficili, così fu deciso in un altro brainstorming che il gioco sarebbe stato ambientato nel medioevo. Siccome il prototipo sviluppato era relativamente semplice si è deciso di omettere la documentazione per risparmiare tempo, era un semplice mini gioco che sarebbe stato sviluppato in un periodo relativamente breve. Il gioco è stato composto da mini giochi che pur avendo una certa cronologia storica non riusciva a collegare tra di loro le varie storie. Quindi, siccome l'idea del mini gioco era anche molto chiara al team, i requisiti di questo sono stati descritti direttamente come un backlog di prodotto.

*La produzione:* alla fine della pre-produzione, EngReq era un mini gioco molto simile a un quiz, in cui gli abitanti di un villaggio medievale presentano, tramite le loro opinioni, quello che un sistema determinato richiederebbe. Il ruolo del giocatore è, in accordo ai requisiti indicati, classificarli tra requisiti funzionali o non funzionali o rifiutarli se non è affatto un requisito. Un sistema di upgrade è stato adattato al mini gioco e c'era la possibilità di aggiungere nuovi sistemi attraverso l'aggiunta di nuovi file di testo in una sottocartella specifica del gioco. EngReq è stato sviluppato in C++ con SDL per visualizzare contenuti multimediali sullo schermo. Per aiutare lo sviluppo di questo progetto è stato usato il sito [www.xp-dev.com](http://www.xp-dev.com), un sito web con repository gratuite e tools focalizzati sullo sviluppo agile. Il gioco è stato sviluppato da un piccolo team senza abilità artistiche. Per questo motivo, la progettazione di scenografie e personaggi è stata esternalizzata, ma anche così è stato applicato il modello delle iterazioni, in cui il progettista ha incontrato regolarmente gli sviluppatori e entrambe le parti hanno cercato di essere le più chiare possibile nel presentare le loro idee.

*La post-produzione:* con EngReq pronto per il beta test, un questionario è stato scritto per essere distribuito agli stakeholders: studenti universitari e laureati, professori di ingegneria del software alla fine di analizzare e valutare le potenzialità del videogioco. Quindi, attraverso le loro risposte, si potrà analizzare l'efficacia del meccanismo a mini giochi. Il questionario utilizzava una scala Likert con cinque livelli in cui la risposta "A" corrisponde a "totalmente d'accordo" e "E" corrisponde a "totalmente in disaccordo". Sono state scritte 11 domande chiuse e ha anche 4 domande aperte, dove gli stakeholder possono dare la loro opinione sul mini gioco, i suoi punti di forza e di debolezza e suggerire miglioramenti. Il questionario è stato applicato a cinque studenti, sei laureati e due insegnanti. Dalle risposte, era chiaro che il pubblico credeva

che il gioco funzionasse al suo scopo, ma il gameplay risultava un pò stancante e ripetitivo. È stato suggerito di aumentare il numero di requisiti disponibili e di mostrarne un numero limitato a caso. Le risposte hanno anche mostrato che il mini gioco ha i concetti corretti riguardo l'ingegneria del software, anche se non sembra essere su questo argomento per via del Concept medievale. Un punto da migliorare è stato il feedback dato al giocatore, dove i partecipanti al questionario suggerivano di rendere più dinamico il meccanismo di feedback visualizzando la risposta corretta subito dopo la scelta e non alla fine del mini gioco. Un postmortem di EngReq è stato prodotto con i suggerimenti sul progetto.

*Il post-mortem:* Come prima applicazione di questa metodologia, ci sono stati diversi problemi che potrebbero essere risolti con più pratica. Ma alcuni di loro, inclusi i diversi orari delle lezioni tra la squadra, hanno ritardato il progetto e non potevano essere risolti facilmente. Comunque, l'applicazione della metodologia Game-Scrum ha portato i suoi benefici riuscendo a fornire un approccio sistematico allo sviluppo di videogames.

L'ambiente di sviluppo del gioco con l'aumentare dei costi e l'alta mutabilità dei requisiti, rende essenziale sapere il prima possibile se l'investimento tornerà in quanto il tutto gira intorno a degli investimenti fatti dalle aziende. Sapere cosa porta "divertimento" al gioco e lavorarlo bene è un pre-requisito per il suo successo del titolo stesso. Ma nello sviluppo tradizionale, la maggior parte di questa scoperta avviene solo in fase di sviluppo tardivo, quando molto tempo e denaro sono già stati investiti e i rischi nell'apportare modifiche sono elevati. Una metodologia iterativa consente di avere pronte funzioni presto e quindi di scoprire e lavorare il "divertimento" del gioco in maniera più rapida e precisa, mentre un processo agile ti consente di concentrarti più sull'implementazioni delle funzionalità. Esistono poche alternative alle metodologie agili focalizzate sulle esigenze di un ambiente di sviluppo come quello videoludico, in cui i team multidisciplinari sono comuni e c'è la necessità di trovare il fattore "divertimento", che non è facilmente quantificabile. Il Game-Scrum cerca di risolvere specifici problemi di sviluppo del gioco provando ad assimilare i suggerimenti dei professionisti del settore e confrontarli con i ricercatori dell'area. Alcuni degli approcci sviluppati da Game-Scrum possono essere riassunti come segue:

- Contenuto artistico: consente di avere lavoro non finito alla conclusione di un'iterazione, e l'assemblare una catena di produzione dove il primo Assets deve essere completato in questa iterazione e i prossimi nelle iterazioni successive.
- Scopo del progetto: avvalersi di tecniche come il brainstorming, la pro-

totipazione e il documento di progettazione del gioco come parte della pre-produzione facilita la scoperta del “divertimento” del gioco e le iterazioni consentono di evolvere questo fattore. Anche queste tecniche possono ridurre la quantità di funzionalità superficiali presenti nel gioco.

- Gestione del progetto: utilizzando il framework di Scrum per sviluppare giochi si ottiene un processo forte come base di Game-Scrum. Come nello stesso Scrum, la costante valutazione del feedback del progetto aiuta a migliorarlo ogni volta che viene applicato.
- Organizzazione del team: suggerendo alcune alternative per migliorare l’organizzazione, in base al quale il team può scegliere secondo i suoi punti di forza e di debolezza quelli che meglio si adattano alla cultura e alle preferenze della squadra.

Anche dimostrando buone soluzioni durante lo sviluppo del gioco, Game-Scrum non è in grado di risolvere tutti i problemi che sorgono nello sviluppo di video-games. Potrebbe ancora esserci delle funzionalità superflue, dal momento che alcune idee emergono solo in seguito. Inoltre, nonostante il brainstorming sia usato per aggregare le idee e per evitare l’inserimento di funzionalità inutili, può portare al problema opposto cioè quello di tagliare le funzionalità basilari e ingigantire il contesto del gioco se non è ben fatto. Sebbene Game-Scrum sia stato testato con lo sviluppo di EngReq, è necessario maturare la metodologia suggerita. Il postmortem del mini gioco ha suggerito che l’approccio ha un potenziale, e l’esperienza acquisita può migliorare i risultati del prossimo mini gioco. I lavori futuri su questo argomento includono l’applicazione Game-Scrum nei progetti della Fellowship of the Game (<http://fog.icmc.usp.br/>), un gruppo di studenti dell’Università di San Paolo che si è concentrato sullo sviluppo di giochi. Con diversi progetti in corso, il primo passo sarà quello di allenare la squadra, utilizzando la metodologia Game-Scrum ai suoi progetti e analizzare l’esperienza dei postmortem. La Raccolta di questi feedback aiuterà a perfezionare e migliorare Game-Scrum, fino a quando non sarà in grado di risolvere la maggior parte dei problemi nello sviluppo del gioco con ogni tipologia di team, da quelli con più esperienza a quelli più inesperti.

# **Capitolo 3**

## **Tecniche e strumenti di progettazione**

### **3.1 Architettura del videogioco**

Il videogame è uno dei software tra i più complessi da sviluppare in ambito informatico. Il game development è una disciplina in cui non si incontra solo la bruta tecnica informatica bensì anche arti visive, musica, narrazione, Design e un vero oceano di materie diverse. Anche restando solo nel livello informatico, la programmazione richiede un insieme eterogeneo di materie informatiche, dalla programmazione grafica, all’architettura di sistemi, all’IA, al data management e così via. Per progettare un gioco gli sviluppatori indipendenti e gli aspiranti tali devono quindi aver ben chiaro come si struttura l’architettura di un programma complesso quanto un videogame. Cosa che in realtà non è così semplice e che questo capitolo cercherà di chiarire almeno nei suoi punti fondamentali.

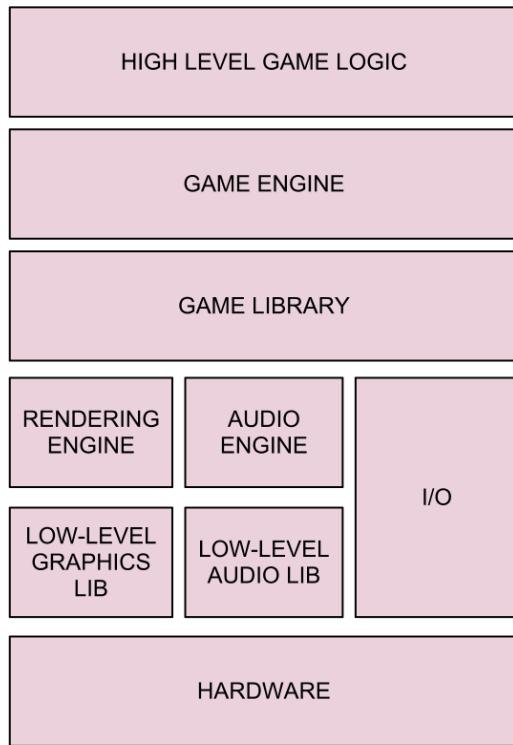


Figura 3.1: Architettura del Videogames

Questa è, a grandi linee, l'architettura di un videogioco. Nella pratica la divisione fra livelli non è sempre così netta ed è possibile trovare librerie che ne coprono due (o più) insieme. Andando a descrivere in dettaglio le varie parti troviamo:

*Librerie Grafiche di Basso Livello:* Tralasciando l'hardware (la cui spiegazione è più che ovvia), il primo strato che si incontra riguarda le librerie di basso livello. Rientrano in questa categoria OpenGL e le Direct3D (non le DirectX di cui le Direct3D sono la parte grafica). Le librerie grafiche di basso livello sono la struttura più vicina all'hardware ed offrono una grande libertà al prezzo di un elevata complessità. Esse spesso fungono da interfaccia fra la CPU e la GPU tramite linguaggi di shading ad-hoc come HLSL o GLSL. Raramente si sviluppa un gioco partendo da tali librerie. Guardando il diagramma precedente la motivazione è intuitiva: partendo dalle librerie di basso livello bisogna ricostruirsi tutti gli strati superiori. Cosa che per dei programmatore indipendenti è un compito lungo, faticoso e quasi mai portato a termine.

*Audio Engine e Librerie Audio di Basso Livello:* Si parlerà di questi due

livelli assieme poiché nella pratica è difficile trovarli completamente separati. Queste librerie, come ad esempio le OpenAL, si occupano della gestione della parte audio del gioco. A seconda del livello di astrazione, tali librerie non si limitano solamente a leggere e riprodurre un file audio ma offrono funzionalità più avanzate quali ad esempio aggiungere effetti d'ambiente, riverbero, eco o la simulazione della spazialità del suono e dell'effetto doppler. La programmazione dell'audio è solitamente considerata meno complessa della programmazione grafica e l'uso di librerie come OpenAL è decisamente comune. Tuttavia come vedremo i livelli superiori spesso e volentieri integrano tali librerie al loro interno.

*Input:* Il layer di input gestisce i comandi in arrivo dalle periferiche dell'utente come mouse, tastiera o gamepad. Librerie corrispondenti a questo livello sono ad esempio le OIS e parte di FreeGLUT. L'uso di queste librerie è solitamente piuttosto intuitivo e non necessita molte spiegazioni. I layer superiori di solito si limitano a wrappare semplicemente le librerie di input.

*Rendering Engine:* I motori di rendering sono molto simili alle librerie grafiche di basso livello ma aggiungono funzionalità importantissime quali: importazione di texture e modelli, gestione dello scenegraph e gestione delle animazioni (oltre a nascondere tutti i dettagli tecnici dei livelli inferiori). Oltre a questo spesso i motori di rendering sviluppano due backend separati e permettono quindi di scrivere codice che si poggia indistintamente su OpenGL o Direct3D in modo da facilitare la portabilità del codice su sistemi diversi. Un motore di rendering molto noto è Ogre3D.

*Multimedia/Game Library:* Salendo di un livello troviamo le game library. I motori di rendering infatti si limitano a gestire la grafica ma non offrono ad esempio nessun supporto per l'audio e l'input. A mettere tutto assieme sono proprio le game library, librerie che non solo uniscono e mescolano tutte le librerie degli strati inferiori (grafica, audio e input) ma aggiungono altre caratteristiche importantissime per i videogame come gestione dell'output su disco e l'integrazione di librerie fisiche (quali ad esempio Bullet o Box2D). Rientrano nella categoria librerie come XNA, Panda3D, DeltaEngine, jMonkeyEngine (in gran parte) e per quanto riguarda il 2D SFML. Le game library sono solitamente un buon punto di partenza per chi vuole sviluppare un videogame perché offrono un ottimo compromesso fra alto e basso livello.

*Game Engine:* La prima cosa da fare quando si sviluppa un videogame è scrivere il game engine. Oppure lo si può prendere già pronto. I game engine offrono un livello di astrazione ancora maggiore delle game library, implementano costruttori di mappe, gestione degli Assets, generatori di terreni, acqua

e condizioni climatiche e molto altro, lasciando allo sviluppatore solo la parte divertente: la logica di alto livello, la costruzione dei livelli e così via. I game engine commerciali sono vere e proprie applicazioni, una specie di IDE dei videogame e offrono quasi sempre tools di modellazione e Design visuali all'avanguardia. Game engine molto noti sono Unity3D, Unigine, CryEngine, Source Engine, Unreal SDK e molti altri ancora. I game engine sono molto potenti e permettono in breve tempo di tirare fuori qualcosa di interessante. Il rovescio della medaglia è che costano. Per usufruire di versioni complete o della possibilità di vendere il proprio gioco si può arrivare a sborsare centinaia o migliaia di euro. Molti game engine inoltre offrono delle API di backend per poter scrivere i propri personali moduli di basso livello (ad esempio per gestire la rete, il multiplayer o un personale sistema di input).

*Logica di Alto Livello:* Arrivando alla punta dell'iceberg troviamo il livello più interessante. Arrivati a questo punto si programma il gioco vero e proprio, si decide la grafica, si impostano le strutture dati, si progetta l'IA e poi si lancia il tutto sul game engine. Spesso questo livello è implementato con linguaggi di alto o altissimo livello (proprietari come l'Unigine Script oppure generici come Lua, C# o Python). Se si arriva a questo punto vuol dire che si è nella parte puramente “creativa” del game developing.

## 3.2 Motori grafici:

Il motore grafico è il nucleo software di un videogioco o di qualsiasi altra applicazione con grafica in tempo reale. Esso fornisce le tecnologie di base, semplifica lo sviluppo, e spesso permette al gioco di funzionare su piattaforme differenti come le console o sistemi operativi per personal computer. La funzionalità di base fornita tipicamente da un motore grafico include un motore di rendering (“renderer”) per grafica 2D e 3D, un motore fisico o rilevatore di collisioni, suono, scripting, animazioni, intelligenza artificiale, networking, e scene-graph. Come altre applicazioni di transizione, i motori grafici sono spesso indipendenti dalla piattaforma, permettendo allo stesso gioco di girare su più piattaforme, incluse le console quali PlayStation, Xbox, Nintendo e su sistemi operativi come Microsoft Windows e Mac OS, con nessuno o qualche piccolo cambiamento al codice sorgente del gioco. Motori grafici avanzati come l'Unreal Engine 3, il Source Engine, l'id Tech 5, il RenderWare e il Gamebryo forniscono una suite di strumenti di sviluppo visuali in aggiunta alla componente software riutilizzabile. Questi strumenti vengono forniti generalmente all'interno di un ambiente di sviluppo integrato (dall'inglese integrated development environment

o più comunemente IDE) affinchè permettano lo sviluppo semplificato e rapido (RAD) dei giochi secondo un metodo di progettazione data-driven. Questi motori grafici vengono chiamati spesso “game middleware” perché, in accordo con il significato commerciale del termine, forniscono una piattaforma flessibile e riutilizzabile che fornisce tutte le funzionalità chiave necessarie esternamente per sviluppare un’applicazione ludica riducendo i costi, la complessità e il tempo impiegato; tutti fattori critici e altamente competitivi nell’industria di videogiochi e computer. Spesso i middleware sono progettati con un’architettura modulare che permette di sostituire o estendere parti del motore con soluzioni più specializzate (e magari costose), come il software Havok per la fisica, FMOD per il suono, o SpeedTree per il rendering. Alcuni motori grafici come il RenderWare sono addirittura concepiti come una serie di middleware indipendenti che possono essere combinati a piacere per creare un motore personalizzato, piuttosto che modificare un motore già esistente. In qualunque modo si ottenga la flessibilità, questa rimane comunque una forte priorità, a causa della grande varietà di applicazioni cui i motori grafici devono rispondere. Nonostante l’idea di motore grafico rimandi immediatamente al concetto di videogioco, in verità essi sono usati in molti altri tipi di applicazioni interattive che richiedono grafica in tempo reale, come dimostrazioni commerciali, progettazioni architettoniche, simulazioni e ambienti di modellazione. Alcuni motori forniscono solo capacità grafiche, invece della vasta gamma di funzionalità richieste da un videogioco. Questi motori affidano agli sviluppatori il compito di inserirle, anche tramite altri middleware, magari da altri giochi. A motori come questi calza veramente la definizione di “motore grafico” in senso stretto. Non c’è comunque una convenzione condivisa sul termine: per il senso generale si può usare il termine “motore 3D”. Esempi di motore grafico in senso stretto sono: Irrlicht, Axiom, OGRE, Power Render, Crystal Space e Genesis 3D. Molti motori moderni forniscono lo scene-graph, una rappresentazione del mondo 3D orientata agli oggetti che spesso semplifica lo sviluppo del gioco e che può essere usata per un rendering più efficiente di mondi virtuali molto vasti. I motori grafici sono costruiti nella maggior parte dei casi su delle API grafiche, come Direct3D o OpenGL, che offrono un’astrazione software della GPU o della scheda video. Sono spesso usate anche librerie di basso livello, come DirectX, SDL e OpenAL, le quali offrono un accesso diretto ad altro hardware (ad esempio mouse, tastiera, joystick, scheda di rete, e scheda audio). Prima dell’avvento della grafica 3D accelerata, si usavano render software come le API Glide. Soluzioni software come questa sono ancora utilizzate in alcuni strumenti di modellazione o di rendering statico, in cui la precisione è più importante della velocità, o quando l’hardware del computer

non soddisfa i requisiti (come il supporto degli shader o delle Direct3D 10, come nel caso di Windows Vista). Lo sviluppo di motori grafici è un progetto comune tra gli studenti di informatica, appassionati, e sviluppatori di videogiochi. Può richiedere grandi conoscenze interdisciplinari che spaziano dalla geometria alla teoria del colore. Dal momento che il settore ha grande visibilità, comunque, questi sviluppatori lo trovano divertente e remunerativo. Un Motore 3D è un software progettato con lo scopo di rappresentare su una superficie 2D (come lo schermo) una scena 3D composta da elementi sintetici. Esistono diverse categorie di motori 3D, ma essenzialmente si possono distinguere in due tipi, a seconda dello scopo per il quale sono preposti: motori 3D in tempo reale e motori 3D per la produzione di immagini fotorealistiche.

- Motori 3D in tempo reale: Dall'inglese “real time 3D engine” sono impiegati laddove sia necessario produrre immagini tridimensionali “al volo” o “Run Time”. Con questa espressione si indica la capacità di calcolare, e quindi di mostrare a schermo (oppure tramite altri dispositivi ottici, ad esempio occhiali per la realtà virtuale) le immagini in brevissimo tempo, tale da ottenere un certo numero di immagini al secondo, tipicamente 30-60 immagini al secondo (abbreviato spesso come fps, dall'inglese “Frames Per Second”). Questi requisiti di velocità possono essere raggiunti con varie tecniche, evolutesi nel tempo grazie soprattutto all'invenzione di dispositivi hardware dedicati allo scopo: gli acceleratori grafici 3D, Questi dispositivi, costituiti in sostanza da un coprocessore matematico e da una certa quantità di memoria RAM, svolgono certe funzioni matematiche estremamente ottimizzate e consentono allo sviluppatore di sgravare la CPU da un'enorme quantità di calcoli, permettendo quindi di realizzare motori grafici più raffinati e più veloci. L'introduzione degli acceleratori grafici ha decretato la formazione di due sottocategorie di motori in tempo reale: i motori grafici software e i motori accelerati in hardware. Nonostante il nome possa trarre in inganno, si parla in entrambi i casi di software. La differenza consiste nel fatto che i primi sfruttano esclusivamente la CPU (ed eventualmente la FPU e le istruzioni SIMD come MMX, SSE, 3DNow! ecc) per effettuare i calcoli geometrici necessari, mentre i secondi relegano molte delle funzioni primarie (come la trasformazione, l'illuminazione, l'applicazione delle texture ecc) all'acceleratore hardware. Ovviamente entrambi gli approcci portano dei vantaggi e degli svantaggi: i motori software renderizzano le immagini esattamente nel modo previsto dal programmatore ma risultano lenti, quindi non possono produrre immagini di elevata qualità per l'eccessiva quantità di calcoli necessari; i

motori accelerati, invece, sono estremamente veloci e producono immagini di elevata qualità, ma richiedono la presenza di hardware dedicato e l'accuratezza delle immagini è soggetta al particolare acceleratore utilizzato. I motori 3D in tempo reale trovano largo impiego nella realizzazione di videogames, simulatori, interfacce grafiche, realtà virtuale.

- Motori 3D fotorealistici: Si definiscono così quei motori grafici 3D che producono immagini di qualità prossima o addirittura paragonabile a immagini di scene reali. I motori grafici di questa categoria sono esclusivamente di tipo software, cioè non si appoggiano su hardware di accelerazione 3D. Nelle applicazioni in cui vengono sfruttati i motori 3D fotorealistici, la precisione e la qualità delle immagini renderizzate è prioritaria rispetto alla velocità di calcolo. I motori grafici di questa categoria sfruttano algoritmi molto sofisticati per simulare fedelmente gli effetti ottici di diffusione, rifrazione, riflessione, pulviscolo, proiezione di ombre, aberrazioni cromatiche e altri effetti che contribuiscono a rendere la scena estremamente realistica. Molti di questi algoritmi non possono essere implementati nei motori in tempo reale per la loro estrema complessità, oppure vengono implementati in forma semplificata e approssimativa. Alcuni di questi algoritmi sono il Ray Tracing, il Photon Mapping, e altri. Data la mole di calcoli necessaria, generalmente i motori 3D fotorealistici sono progettati per essere eseguiti su macchine multiprocessore e su cluster. Questi motori grafici sono usati per la realizzazione di opere artistiche, progettazione architettonica e meccanica, Design, produzioni cinematografiche (per effetti speciali o per interi film d'animazione).

La maggior parte della matematica implicata nella realizzazione di un engine è quella dei vettori e delle matrici.

### 3.2.1 Unity 3D

In questo sottocapitolo si prende in esame Unity, il motore sviluppato da Unity Technologies, scelto per lo sviluppo dei progetti.



Figura 3.2: Unity

Dopo un breve accenno alla storia, ai concetti base per l'utilizzo ed all'interfaccia di questo ambiente vengono analizzati il sistema di scripting interno per programmare, il funzionamento e le caratteristiche principali messe a disposizione. Unity è un game engine multipiattaforma sviluppato da Unity Technologies, che viene principalmente utilizzato per sviluppare videogiochi e simulazioni tridimensionali e bidimensionali per computer, console e dispositivi mobili. Prima annunciato solo per OS X alla Worldwide Developers Conference di Apple nel 2005, da allora è stato esteso e ora supporta 27 piattaforme. Unity è un motore di gioco multiuso che supporta la grafica 2D e 3D, funzionalità di trascinamento della selezione e scripting utilizzando C#. Sono stati supportati altri due linguaggi di programmazione: Boo, che è stato dichiarato obsoleto con il rilascio di Unity 5 e JavaScript che ha iniziato il suo processo di decadimento nell'agosto 2017 dopo il rilascio di Unity 2017.1. Il motore si rivolge alle seguenti API grafiche: Direct3D su Windows e Xbox One; OpenGL su Linux, macOS e Windows; OpenGL ES su Android e iOS; WebGL sul web; e API proprietarie sulle console per videogiochi. Inoltre, Unity supporta le API di basso livello Metal su iOS e macOS e Vulkan su Android, Linux e Windows, oltre a Direct3D 12 su Windows e Xbox One. All'interno dei giochi 2D, Unity consente l'importazione di sprite e un renderer avanzato per il mondo 2D. Per i giochi 3D, Unity consente di specificare la compressione delle texture, le mipmap e le impostazioni di risoluzione per ogni piattaforma supportata dal motore di gioco e fornisce supporto per la mappatura degli urti, la mappatura della riflessione, la mappatura del parallax, l'occlusione ambientale dello spazio dello schermo (SSAO), ombre dinamiche che utilizzano mappe ombra, render-to-texture e effetti di post-elaborazione a schermo intero. Unity offre anche servizi agli sviluppatori: Unity Ads, Unity Analytics, Unity Certification, Unity Cloud Build, Unity Everyplay, Unity IAP, Unity Multiplayer, Unity Performance Reporting e Unity Collaborate che facili-

tano il lavoro di inserimento di Ads (pubblicità) o di meccanismi di matchmaking o multiplayer. Unity supporta la creazione di vertici personalizzati, frammenti (o pixel), tassellazione, shader di elaborazione e shader di superficie di Unity utilizzando Cg, una versione modificata del linguaggio di shading di alto livello di Microsoft. Unity supporta la build per 27 piattaforme diverse. Le piattaforme sono elencate di seguito: iOS, Android, Tizen, Windows, Universal Windows Platform, Mac, Linux, WebGL, PlayStation 4, PlayStation Vita, Xbox One, Wii U, 3DS, Oculus Rift, Google Cardboard, Steam VR, Playstation VR, Gear VR, Windows Mixed Reality, Daydream, Android TV, Samsung Smart TV, tvOS, Nintendo Switch, Fire OS, Facebook Gameroom, Apple ARKit, Google ARCore e Vuforia. Unity ha precedentemente supportato altre 7 piattaforme tra cui il suo Unity Web Player, rimosse in quanto ritenute obsolette ed andavano ad appesantire il processo di build del prodotto. Unity Web Player era un plugin per browser supportato solo in Windows e OS X, che è stato deprecato in favore di WebGL. Unity è il kit di sviluppo software (SDK) predefinito per videogiochi Wii-U di Nintendo, con una copia gratuita inclusa da Nintendo con ciascuna licenza per sviluppatore Wii-U. Unity Technologies definisce questo “bundle” di un SDK di terze parti un “primo del settore” in cui investire. Unity viene fornito con quattro opzioni di licenza. Ecco l’elenco di tutte le licenze disponibili e le loro differenze:

License Name	All Engine Features and Platforms	Splash Screen	Cloud Build Queue	Multiplayer	Revenue Capacity	Performance Reporting	Premium Support	Access to Source Code	Price
Personal	Yes	Made With Unity and optional Custom Animation	Standard	20 CCUs	\$100,000	No	No	No	Free
Plus	Yes	Custom Animation and/or Made with Unity	Priority	50 CCUs	\$200,000	Yes	No	No	\$35 Monthly
Pro	Yes	Custom Animation and/or Made with Unity	Concurrent Builds	200 CCUs	Unlimited	Yes	Yes	No	\$125 Monthly
Enterprise	Yes	Custom Animation and/or Made with Unity	Dedicated Build Agents	Custom Multiplayer	Unlimited	Yes	Yes	Yes	Negotiated Pricing

Figura 3.3: Licenze Unity

Nel 2012, VentureBeat ha dichiarato: “Poche aziende hanno contribuito tanto allo sviluppo di giochi prodotti in modo indipendente come Unity Technologies. Oltre 1,3 milioni di sviluppatori utilizzano i propri strumenti per creare grafica gee-whiz nella loro console iOS, Android, PC e giochi basati sul Web. ... Unity vuole essere il motore per i giochi multipiattaforma, punto”. Per gli Apple Design Awards alla fiera del WWDC del 2006, Apple, Inc. ha nominato Unity come secondo classificato per la categoria Best Use di Mac OS X Graphics, un anno dopo il lancio di Unity alla stessa fiera. Unity Technologies afferma che questa è la prima volta che uno strumento di progettazione di giochi è stato nominato per questo premio. Un sondaggio del maggio 2012 della rivista

Game Developer ha indicato Unity come il suo motore di gioco principale per piattaforme mobili. A luglio 2014, Unity ha vinto il premio “Best Engine” agli Develop Industry Excellence Awards del Regno Unito. Unity 5 è stato accolto con elogi analoghi, con The Verge che afferma che “Unity ha iniziato con l’obiettivo di rendere lo sviluppo del gioco universalmente accessibile .... Unity 5 è un passo tanto atteso verso quel futuro”. Dopo la pubblicazione di Unity 5, Unity Technologies ha sollevato alcune critiche per l’elevato volume di giochi prodotti rapidamente pubblicati sulla piattaforma di distribuzione di Steam da parte di sviluppatori inesperti. L’amministratore delegato John Riccitiello ha detto in un’intervista che ritiene che questo sia un effetto collaterale del successo di Unity nel democratizzare lo sviluppo del gioco: “Se potessi scegliere, mi piacerebbe vedere 50 milioni di persone che usano Unity, anche se non penso che possa succedere presto, compresi ragazzi delle scuole superiori e del college. Persone al di fuori del settore principale, penso che sia triste che molte persone siano consumatori di tecnologia e non creatori. Un luogo in cui le persone sanno come creare, non solo consumare, e questo è ciò che stiamo cercando di promuovere”. Nel dicembre 2016, Unity Technologies ha annunciato che cambierà il sistema di numerazione delle versioni per Unity da identificatori basati su sequenze fino all’anno di rilascio per allineare il controllo delle versioni con la loro frequenza di rilascio più frequente.

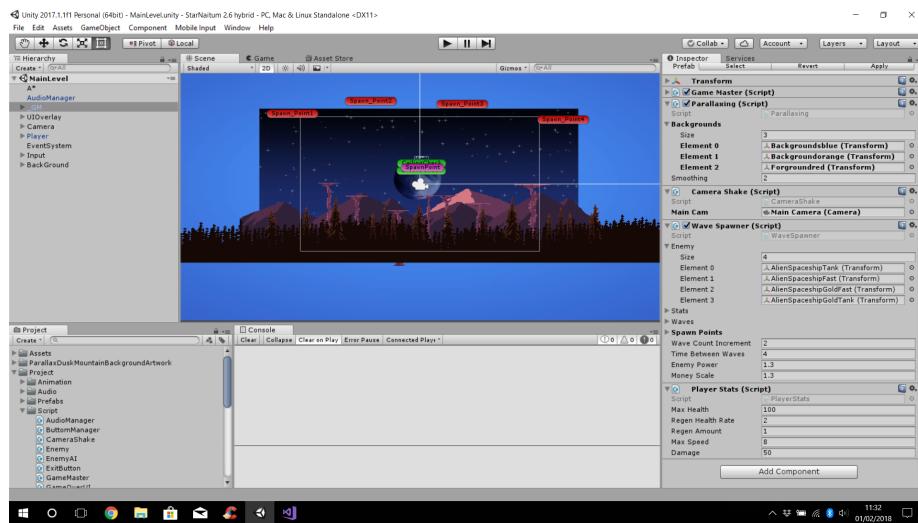


Figura 3.4: Unity Interface

**Interfaccia:** La finestra principale di Unity è composta da varie schede, chiamate Viste. Ogni vista ha il suo scopo specifico che sarà descritto nelle

prossime sezioni.

*Project View:* Ogni progetto di Unity contiene una cartella Assets (attività). Il contenuto di questa cartella è rappresentato nella Project View (finestra di progetto). È qui che si memorizzano tutti gli Assets che compongono il gioco, come scene, script, modelli 3D, texture, file audio, e Prefabs (prefabbricati). Se si preme su uno qualsiasi degli Assets nella finestra di progetto, è possibile scegliere Reveal in Explorer / Reveal in Finder on Mac (Mostra in Esplora risorse / Mostra nel Finder su Mac) per vedere effettivamente dove è posizionato l'Asset nel file system.

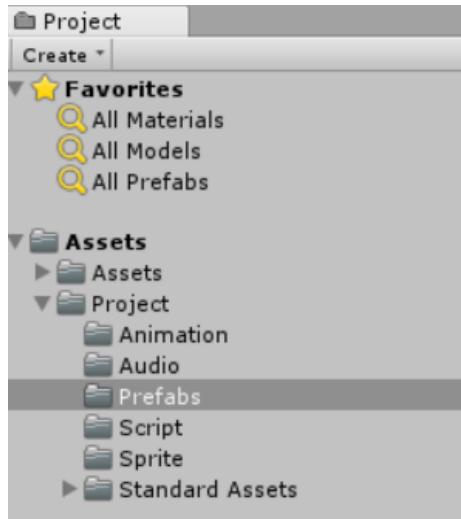


Figura 3.5: Project View

*Hierarchy View:* La Hierarchy View (finestra gerarchia) contiene ogni GameObject presente nella scena. Alcuni di questi sono istanze di Assets come i modelli 3D, e altri sono esempi di prefabbricati, oggetti personalizzati che compongono gran parte del gioco. È possibile selezionare gli oggetti della gerarchia e trascinare un oggetto su un altro e fare uso di Parenting per inglobare un oggetto dentro l'altro. Quando gli oggetti sono aggiunti e rimossi nella scena, essi appaiono e scompaiono pure dalla Gerarchia.



Figura 3.6: Hierarchy View

**Toolbar:** La barra degli strumenti è composta da cinque controlli di base, ognuno dei quali si riferisce a diverse parti dell'editor.



Figura 3.7: Toolbar

La **Transform Tools** viene usata nella finestra Scene per traslare, ruotare e scalare.



Figura 3.8: Transform Tools

La **Transform Gizmo Toggles** serve per spostare il pivot dell'oggetto e per passare dalle coordinate locali a quelle di mondo. Viene sempre usata nella finestra Scene.



Figura 3.9: Transform Gizmo Toggles

I pulsanti **Play/Pause/Step** vengono usati nella Game View.



Figura 3.10: Play/Pause/Step

Il **Layers Drop-down** controlla quali oggetti vengono visualizzati nella finestra Scene.



Figura 3.11: Layers

Il **Layout Drop-down** controlla la disposizione di tutte le View.



Figura 3.12: Layout

*La Scene View:* Finestra della scena, rappresenta l'accesso alla scena interattiva. Si utilizzerà la vista della scena per selezionare e posizionare gli ambienti, il giocatore, la camera, i nemici, e tutti gli altri GameObjects. Manovrare e manipolare gli oggetti all'interno della Scena sono alcune delle funzioni più importanti in Unity e a tal fine Unity fornisce una sequenza di tasti per le operazioni più comuni, ad esempio cliccando un GameObject e premendo il tasto F da tastiera la Scena e il Pivot verranno centrati sull'oggetto.

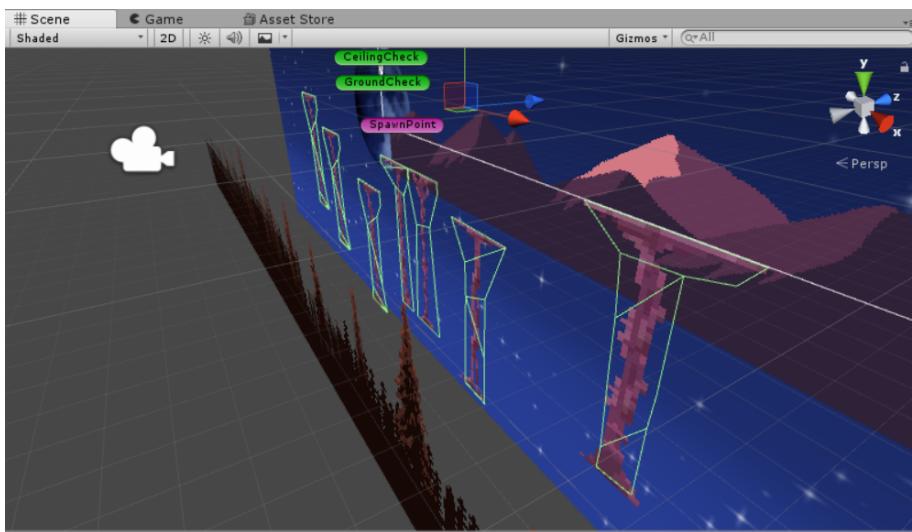


Figura 3.13: Scene View

Nell'angolo in alto a destra della vista scena si trova il Gizmo della scena. Questo mostra l'orientamento attuale della Camera nella scena, e consente di modificare rapidamente l'angolo di visualizzazione. Ciascuno dei colori del gizmo rappresenta un asse geometrico. È possibile fare clic su uno degli assi per impostare la Camera ad una vista ortogonale (cioè, prospettiva-free) allineata lungo l'asse corrispondente. È possibile fare clic sul testo sotto il gizmo per passare dalla vista in prospettiva normale ad una vista isometrica.

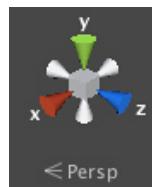


Figura 3.14: Gizmo

Per traslare, ruotare e ridimensionare i singoli GameObject della scena si utilizzano gli strumenti di trasformazione nella barra degli strumenti. Ciascuno ha un Gizmo corrispondente che appare attorno al GameObject selezionato. È possibile utilizzare il mouse e manipolare ogni asse del Gizmo per modificare la Trasformazione del GameObject, oppure è possibile digitare i valori direttamente nei campi numero del componente di trasformazione nella vista Inspector.

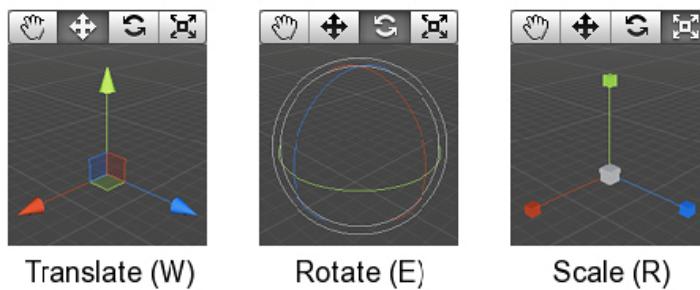


Figura 3.15: Trasformazioni del GameObject

La **Scene View control bar** consente di vedere la scena in varie modalità di visualizzazione: Textured, Wireframe, RGB, Overdraw e molti altri. Essa consentirà inoltre di vedere (e udire) l'illuminazione, gli elementi di gioco, e i suoni nella scena.



Figura 3.16: Scene View control bar

*Inspector:* I giochi in Unity sono costituiti da diversi tipi di GameObject che contengono mesh, script, suoni o altri elementi grafici come le luci. La vista Inspector visualizza informazioni dettagliate sul GameObject attualmente selezionato, inclusi tutti i componenti collegati e le loro proprietà. In questa vista è possibile modificare la funzionalità del GameObject nella scena. Qualsiasi proprietà che viene visualizzata nella finestra di ispezione può essere modificata direttamente. Anche le variabili degli script possono essere modificate senza modificare lo script stesso. È possibile utilizzare l'Inspector per cambiare le variabili in fase di esecuzione e per sperimentare diversi comportamenti. In uno script, se si definisce una variabile pubblica di un tipo di oggetto (come ad GameObject o Transform), è possibile poi trascinare e rilasciare un GameObject o un Prefab nella finestra per assegnare l'oggetto a quella variabile. Infine è possibile utilizzare il menù a discesa Layer per assegnare un livello di rendering diverso ad un GameObject. Usare il menu a discesa Tag per assegnare un tag diverso ad un GameObject.



Figura 3.17: Inspector

*Game View:* La Game View (visuale di gioco) è mostrata dalla Camera presente nel gioco ed è fondamentale per produrre la sua rappresentazione finale e pubblicarlo. Sarà necessario quindi utilizzare una o più camere per controllare ciò che il giocatore vede effettivamente quando sta giocando.



Figura 3.18: Game View

Utilizzare i pulsanti nella barra degli strumenti per il controllo della modalità Play e per vedere come funzionerà il gioco una volta pubblicato. In modalità Play, tutte le modifiche apportate sono temporanee, e saranno resettate una volta rimesso in stop.



Figura 3.19: Pulsanti Play Mode

Il primo elenco a discesa sulla barra di controllo del gioco è Aspect. Qui, è possibile forzare le proporzioni della finestra di visualizzazione del gioco su valori diversi. Può essere usato per testare come sarà su monitor con proporzioni diverse. Più a destra vi è il pulsante Maximize on Play toggle. Una volta attivato questo strumento la visuale di gioco sarà impostata al 100% rispetto alla finestra dell'editor per un'anteprima a schermo intero quando si è nella modalità di riproduzione. Proseguendo a destra vi è il Gizmos toggle (selettore del Gizmo). Con lo strumento attivato, tutti i Gizmo che appaiono nella vista scena saranno visualizzati nella scena del gioco. Infine abbiamo il pulsante Stats (statistiche). Questo pulsante, se attivato, mostra le statistiche del Rendering utili per l'ottimizzazione delle prestazioni grafiche.



Figura 3.20: GameView Control Bar

*Scripting:* Linguaggi Il sistema di scripting è basato su Mono, una versione open-source di .NET. Come .NET, Mono supporta molti linguaggi di programmazione, ma Unity supporta solo C#, Boo, e Javascript. Ogni linguaggio ha i suoi vantaggi e svantaggi. C# ha il vantaggio di avere una definizione ufficiale e molti libri didattici e tutorial online, oltre che ad essere simile ad un linguaggio principale come .NET e Mono, e quindi viene utilizzato in un gran numero di software commerciali ed opensource. Di contro C# è la scelta più ridondante. Boo sembra essere ben voluto dai programmatori Python, ma è probabilmente utilizzato dal minor numero di utenti Unity e quindi è il linguaggio con il minimo supporto. La versione di Javascript in Unity è più ristretta ma a quanto pare deriva dal Boo e non è proprio come il Javascript standard (per cui è stato suggerito che possa essere chiamato UnityScript).

*Gli script sono Assets:* proprio come texture e modelli e tanti altri tipi di oggetti. Diversi script sono forniti di base nella cartella Standard Assets. Gli script possono essere importati come tutti gli altri Assets dal menu Asset/Import.

*Gli script sono Component:* Unity è programmato tramite script - non vi è alcuna programmazione C++ in questione (a meno che non si scrivano plug-in o si compri una licenza dei file sorgente). Lo Scripting in Unity è un pò diverso dalla programmazione tradizionale, ma simile all'utilizzo del linguaggio di scripting Linden in Second Life e Lua in CryEngine ove singoli script sono attaccati agli oggetti nel gioco. Ciò si presta ad una progettazione orientata agli oggetti in cui la scrittura degli script serve per controllare particolari entità. In Unity, gli script sono componenti e fanno parte degli oggetti del gioco, proprio come gli altri componenti.

```
Object
  AnimationClip
  AssetBundle
  AudioClip
  Component
    Behaviour
      Animation
      AudioListener
      AudioSource
      Camera
      ConstantForce
      GUIElement
        GUIText
        GUITexture
      GUILayer
      LensFlare
      Light
    MonoBehaviour
```

Figura 3.21: La classe Object

In particolare, gli script fanno parte della classe MonoBehaviour, una sottoclasse diretta di Behaviour, il quale è un componente che può essere abilitato / disabilitato.

*Gli script sono classi:* Ogni script definisce in realtà una nuova sottoclasse di MonoBehaviour. Con uno script Javascript, Unity considera questa definizione come implicita - la nuova classe è denominata in base al nome del file, e il contenuto dello script è trattato come se fosse all'interno di una definizione di classe.

*Struttura Script:* Un “pezzo” di codice associato a un oggetto non è molto interessante, se poi non viene eseguito. Gli script aggiungono comportamenti agli oggetti mediante l’attuazione di funzioni di callback che vengono richiamati dal motore Unity durante la vita di un gioco. Qui di seguito vengono elencate le più importanti.

- Awake: viene chiamata quando l’oggetto viene caricato.
- Start: viene chiamata subito dopo Awake, ma solo quando o se l’oggetto è attivo (e solo dopo la prima attivazione, non se lo stato dell’oggetto attivo viene ripetutamente acceso o spento).
- Update: viene chiamata fotogramma per fotogramma mentre l’oggetto è attivo, dopo lo Start.
- FixedUpdate: viene chiamata dopo un intervallo di aggiornamento che viene fissato a priori (lo stesso intervallo usato per gli aggiornamenti della fisica).

A volte si desidera eseguire azioni quando un oggetto di gioco è attivato o disattivato. Ad esempio, si potrebbe volere che la tabella dei punteggi di un gioco venga visualizzata ogni volta che debba essere attivata. Se si implementa quel codice nella funzione di Start sarebbe stato eseguito solo una volta al massimo. Proprio per questo è necessario usare la funzione OnEnable e se vi è necessità esiste anche la funzione inversa OnDisable. Altre funzioni vengono richiamate solo per certi tipi di eventi attivati dalla fisica:

- OnCollisionEnter
- OnCollisionStay
- OnCollisionExit

*Rendering:* Unity supporta diversi tipi di Rendering. La scelta di uno o dell’altro dipende dal tipo di videogioco che si sta creando e dal tipo di piattaforma e di hardware che si intende usare. Ogni Rendering Path ha differenti

caratteristiche e prestazioni diverse che possono interagire con le luci e le ombre. Il Rendering Path da usare dovrà essere selezionato in Player Settings. Se la scheda grafica non può supportare il rendering path selezionato Unity ne userà automaticamente uno con una qualità minore. In questo modo in una GPU che non può supportare il Deferred Lighting verrà usato il Forward Rendering. Se il Forward Rendering non è supportato verrà usato Vertex Lit. Il Deferred Lighting è il rendering path con la miglior qualità di illuminazione e di creazione delle ombre. È la scelta migliore se abbiamo un gran numero di luci in realtime, ma richiede un certo livello di hardware infatti non è supportato su dispositivi mobile. Forward è un rendering path basato sugli shader. Supporta l'illuminazione pixel per pixel (includendo le normal map) e la creazione di ombre in realtime da una luce direzionale. Nelle impostazioni di default vengono calcolate un numero limitato di luci pixel per pixel, mentre il resto delle luci sono calcolate sui vertici dell'oggetto. Nel Forward Rendering, le luci più vicine all'oggetto sono completamente renderizzate attraverso la modalità di illuminazione per-pixel (tecnica in cui l'immagine della scena calcola l'illuminazione per ogni pixel che viene renderizzato). Dal quarto punto di luce in poi vengono calcolate per vertex (tecnica che calcola l'illuminazione di un modello 3D per ogni vertice e poi interpola i valori risultanti sopra le facce del modello per calcolare i valori finali del colore per ogni pixel). Le altre luci sono calcolate attraverso la tecnica Spherical Harmonics che è più veloce ma è solo un'approssimazione. Vertex Lit è il rendering path con il livello di qualità di illuminazione più basso e non vi è supporto per le ombre in realtime. È la miglior scelta su hardware limitato o su piattaforme mobili che non supportano gli altri tipi di rendering.

*Shader:* Unity viene fornito con più di 100 shader che vanno dai più semplici (Diffuse, Glossy, ecc) a quelli molto più avanzati (Self-Illuminated, Bumped specular ecc). Gli shader in Unity sono creati attraverso il pannello Materials, che essenzialmente combina il codice degli shader con altri parametri come texture. Le proprietà del Materiale appariranno nel pannello Inspector quando un Material o quando un GameObject che contiene un materiale viene selezionato. La parola inglese shader indica uno strumento della computer grafica 3D che generalmente è utilizzato per determinare l'aspetto finale della superficie di un oggetto. Consiste essenzialmente in un insieme di istruzioni. Gli shader devono riprodurre il comportamento fisico del materiale che compone l'oggetto cui sono applicati. Si può quindi creare uno shader per i metalli, uno per la plastica, uno per il vetro e così via, e riutilizzarli più volte all'interno di una scena. Una volta modellato un oggetto complesso, come può essere ad esempio una finestra, si associa al modello della cornice uno shader per il legno, uno per la maniglia, e

uno per il vetro. La caratteristica riutilizzabilità di questo strumento è preziosa nel lavoro con la computer grafica 3D, sia in termini di tempo che di risultato finale. Per determinare l'apparenza della superficie, essi utilizzano tecniche già consolidate come l'applicazione di texture e la gestione delle ombre. Gli shader possono anche essere usati per applicare effetti di postprocessing. Essendo programmi a tutti gli effetti, è possibile utilizzarli anche per la replicazione di eventi fisici molto complessi quali collisioni e simulazioni fluidodinamiche. Il pannello Material appare così:

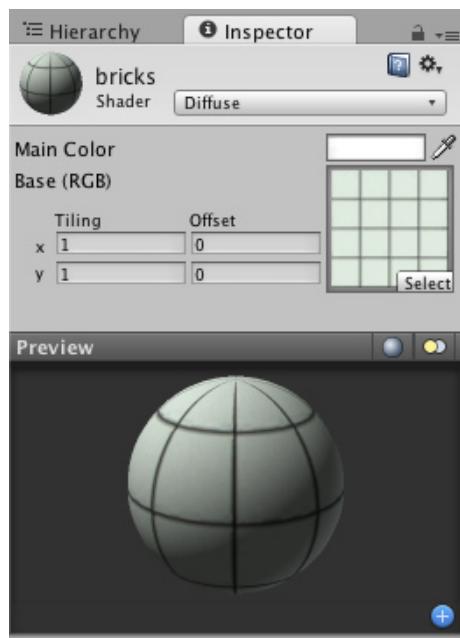


Figura 3.22: Pannello Material

Ogni materiale apparirà in modo diverso nell'Inspector dipendentemente dallo specifico shader che si sta usando. Lo shader determina quale tipo di proprietà sono disponibili per le modifiche nel pannello Inspector per quel tipo di materiale. Bisogna ricordare che uno shader viene implementato attraverso l'uso di un materiale. Così mentre gli shader definiscono proprietà che vengono mostrate nell'Inspector, ogni materiale contiene i dati riguardanti slider, colori e texture. La cosa più importante da ricordare è che un singolo shader può essere utilizzato in più Material, ma un singolo Material non può utilizzare più di uno shader.

*Scalabilità:* Quando si utilizzano effetti di shader avanzati si vuole assicurare che il gioco funzioni bene su qualsiasi hardware di destinazione. Il sistema di shader di Unity fa ampio uso di un sistema di fallback in modo che ogni

utente possa avere la migliore esperienza possibile e per questo supporta anche l’emulazione della scheda grafica per semplificare i test.

*Post-processing:* Unity ha un gran numero di effetti di post-elaborazione delle immagini a tutto schermo tra i quali troviamo: riflessi di luce attraverso gli alberi, profondità di campo ad alta qualità, effetti di lente, aberrazione cromatica, curve di correzione del colore e molto altro. Con Unity 3, sono stati introdotti i Surface Shaders, un nuovo metodo semplificato per costruire shader per più tipi di dispositivi e rendering paths. Abbiamo la possibilità di scrivere un semplice programma e Unity compilerà sia per Forward & Deferred Rendering rendendolo funzionante con le lightmaps e convertendolo automaticamente in GLSL per i dispositivi mobili. Questo significa che è possibile concentrarsi su come ottenere il look giusto e fare in modo che possa adattarsi fino ad usare le Spherical Harmonics o funzioni fisse di illuminazione su hardware di bassa fascia. Se si desiderano modelli di illuminazione personalizzati basta applicare l’equazione di illuminazione all’interno di una funzione e Unity fa tutto il resto per qualsiasi metodo di rendering.

*Prestazioni:* Batching Per ridurre al minimo le chiamate di rendering, Unity combinerà automaticamente la geometria delle scene in batches. Questo minimizzerà in modo significativo i sovraccarichi mantenendo comunque la massima flessibilità. Unity, inoltre, combina insieme oggetti statici in fase di compilazione per assicurare la massima capacità di elaborazione della geometria. Occlusion Culling Insieme a Umbra Software Unity ha sviluppato una tecnica completamente nuova per quanto riguarda soluzioni di visibilità precalcolate. L’Occlusion Culling funziona su cellulari, web e console con sovraccarico minimo a livello di runtime, riducendo il numero di oggetti renderizzati a quelli necessari. Viene usata per determinare quale superficie e quali parti delle superfici non dovrebbero essere visibili all’utente da un certo punto di vista e che non verranno renderizzate. Considerando i vari passi riguardanti la rendering pipeline, la proiezione, il piano di clipping e la rasterizzazione viene implementata attraverso l’uso dei seguenti algoritmi:

- Z-buffering - Scan Line Edge List
- Depth Sort (Algoritmo del Pittore)
- Binary space partitioning (BSP)
- Ray tracing

*GLSL Optimizer:* Le OpenGL ES consentono di usare gli shader su dispositivi mobili. Purtroppo molti driver grafici lasciano ancora a desiderare, così è

stato sviluppato un ottimizzatore per le GLSL shader con un miglioramento di 2-3 volte del fillrate 2 della scheda video.

*LOD Support:* Siccome le scene diventano più complesse le prestazioni sono da tenere sempre più in considerazione, un modo per gestire questa situazione è quello di avere mesh con differenti livelli di dettaglio a seconda di quanto la camera è lontana dall'oggetto. Unity può gestire la situazione usando LOD (level of detail) Groups grazie al supporto interno sul livello di dettaglio da applicare all'oggetto. Il Fillrate è un parametro per valutare le prestazioni di una scheda video. Esistono due tipologie di fillrate che vengono prese in considerazione: Il pixel fillrate indica la quantità di pixel che la GPU (Graphics Processing Unit) è in grado di scrivere nella memoria video in un secondo e viene misurato in Megapixel/s. Esso corrisponde al prodotto tra la frequenza di clock del processore grafico e il suo numero di pipeline; il texel fillrate indica la quantità di texel (elemento fondamentale di una texture) visualizzabili in un secondo e viene misurato in Megatexel/s. Attraverso questo tipo di tecnica avremo la possibilità di usare due tipi di mesh: una ad alta risoluzione da utilizzare quando la camera è più vicina e l'altra a bassa risoluzione da utilizzare quando siamo lontani. In questo modo le prestazioni saranno sicuramente migliori.

*Illuminazione:* Il motore di rendering interno di Unity supporta il Linear Space Lighting (correzione di gamma) e il rendering HDR. La resa dell'illuminazione è ancora più veloce in Unity grazie al nuovo motore di render multi-thread. Il **Linear Lighting** si riferisce al processo di illuminazione di una scena che abbia tutti gli ingressi linearizzati. Normalmente le textures sono create con una gamma pre-applicata e ciò significa che quando vengono utilizzate nei materiali non saranno lineari. Se queste texture verranno utilizzate nell'equazione di illuminazione globale porteranno a risultati non corretti del calcolo, poiché ci si aspetta di avere ingressi linearizzati prima dell'uso. Quindi grazie al processo di linearizzazione possiamo essere sicuri del fatto che sia gli ingressi che le uscite di uno shader siano nella corretta gamma di colori e questo è il risultato di un'illuminazione corretta che dovrebbe rispecchiare meglio la realtà. Linear Intensity Response quando si sta usando il **Gamma Space Lighting** i colori e le texture interne ad uno shader hanno una correzione di gamma. Infatti i colori usati in uno shader con un'alta illuminazione sono più brillanti e luminosi di quello che dovrebbero essere nell'illuminazione lineare. Questo significa che più la luce cresce più la superficie diventerà brillante in maniera non lineare. In questo modo l'illuminazione potrebbe essere troppo forte in alcuni punti e potrebbe anche restituire modelli e scene non corrispondi al reale. Quando si usa il Linear Lighting succede che la risposta della superficie rimarrà lineare an-

che se cresce l'intensità della luce. Questo tecnica porta ad avere superfici più realistiche e una miglior resa dei colori nella scena di gioco dando la possibilità di regolare l'illuminazione in maniera più realistica.

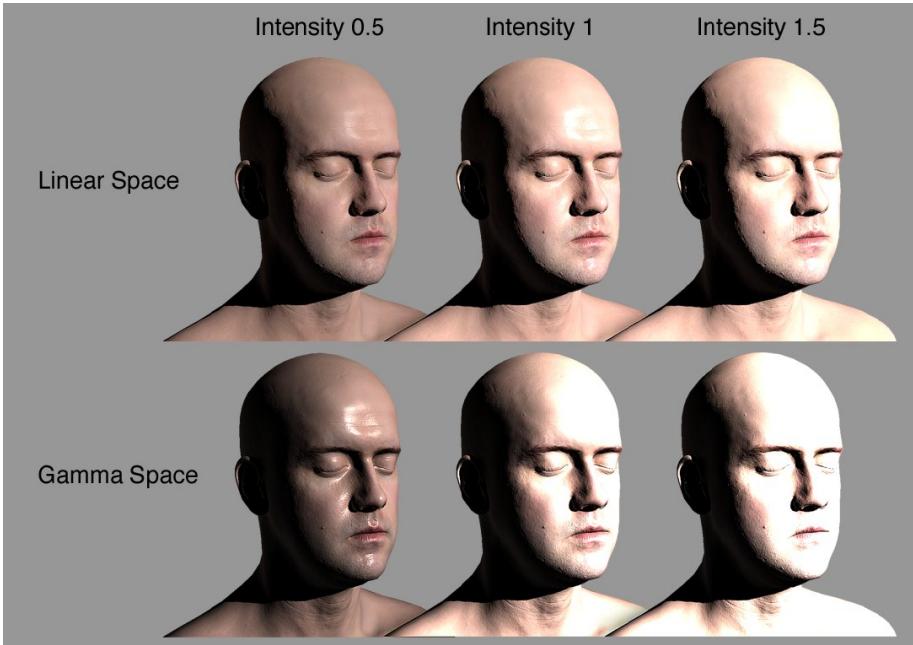


Figura 3.23: Infinite, 3D Head Scan created da Lee Perry-Smith

Attivare il linear lighting è molto semplice. Questa caratteristica può essere implementata in ogni progetto: Edit/Project/Settings/Player/Other Settings.

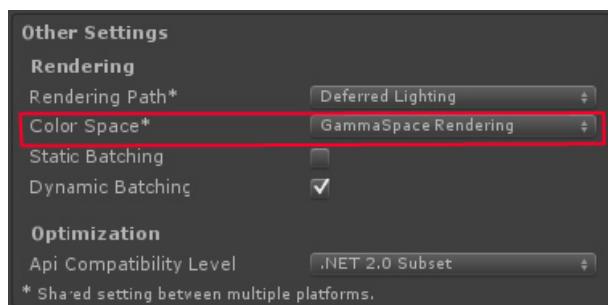


Figura 3.24: Other Settings

*Lightmapping:* Quando si utilizza il linear lighting tutte le luci e le texture vengono linearizzate, questo significa che i valori che verranno passati al lightmapper hanno bisogno di essere modificati. Cambiando il tipo di illuminazione

è necessario rifare il bake delle lightmap. L'interfaccia di Unity mostra un avviso se le lightmap non si trovano nello spazio di colori corretto.

**HDR (High Dynamic Range):** Nel rendering standard i valori del rosso, del verde e del blu per ogni pixel sono ognuno rappresentato in un range compreso tra 0 e 1 dove 0 rappresenta l'intensità minima, mentre 1 rappresenta l'intensità massima per lo schermo del dispositivo. Questo concetto è semplice e facile da applicare, ma non riflette con precisione il modo in cui funziona l'illuminazione in una scena di vita reale. L'occhio umano tende ad adattarsi alle condizioni di illuminazione locali, poiché un oggetto che appare bianco in ambiente poco illuminato può apparire meno luminoso di un oggetto che sembra grigio in pieno giorno. Inoltre, l'occhio è più sensibile alle differenze di luminosità nella parte bassa del range rispetto alla fascia alta. Gli effetti visivi più convincenti possono essere ottenuti se la resa è atta a permettere gli intervalli di valori di pixel che riflettono più accuratamente i livelli di luce che sarebbero presenti in una scena reale. Permettere alla rappresentazione interna della grafica di utilizzare i valori al di fuori del range 0..1 è l'essenza del rendering in High Dynamic Range (HDR). HDR può essere attivato separatamente per ogni camera presente nella scena usando le impostazioni nel pannello del componente Camera:

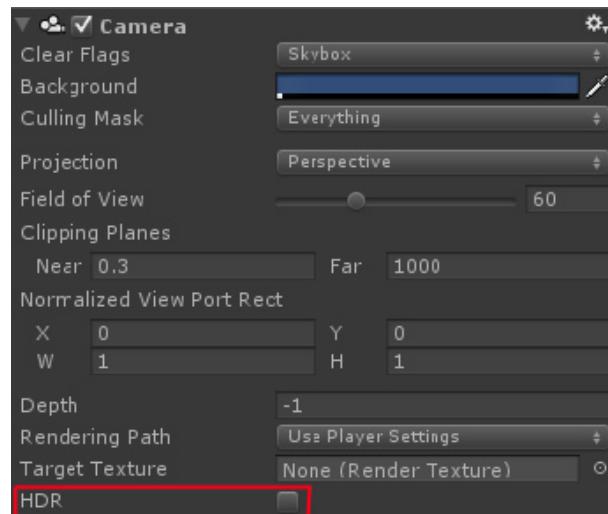


Figura 3.25: Componente Camera

**Realtime Shadows:** Le ombre in tempo reale possono essere create da qualsiasi luce nella scena. Una varietà di strategie vengono utilizzate per renderle davvero performanti anche su computer datati. Per quanto riguarda la pubblicazione di un gioco per piattaforme desktop, Unity ci dà la possibilità di usare

ombre real-time per ogni sorgente di luce. Gli oggetti possono creare ombre su altri oggetti e anche su parti di loro stessi (“self shadowing”). Tutti i tipi di luci (directional, spot e point) supportano le ombre real-time che possono essere di due tipi: **Hard Shadows** oppure **Soft Shadows**. Curiosamente le migliori ombre sono quelle non real-time. Quindi nel caso che la geometria del livello di gioco e l’illuminazione siano statici è consigliato precalcolare le lightmap nell’applicazione 3D. Le ombre calcolate in questo modo saranno di qualità migliore e più performanti piuttosto che quelle prodotte attraverso la tecnica in real-time. *Qualità delle ombre:* Unity usa quelle che vengono chiamate Shadows Maps per creare le ombre. La tecnica dello Shadow Mapping è un approccio basato sulle texture infatti possiamo pensare ad esse come “texture di ombre” proiettate dal punto di luce direttamente sulla scena. La qualità della mappatura dipende da due fattori:

- La risoluzione (dimensione) delle shadow map: più larga sarà la shadow map più alta sarà la qualità dell’ombra.
- Il filtering delle ombre: le hard shadow usano i pixel più vicini alla shadow map, mentre le soft shadow calcolano la media basata sui pixel di più shadow map ottenendo così un effetto risultante più morbido (sono più pesanti nel calcolo del render).

**Shadow mapping** è una tecnica attraverso la quale le ombre sono state create nella grafica computerizzata in 3D. Questo concetto è stato introdotto da Lance Williams nel 1978 in un documento intitolato “Casting curved shadow on curved surfaces”. Da questo momento in poi sono state usate sia in scene pre-renderizzate che in scene real-time in diversi giochi per console e PC. In questa tecnica le ombre sono create testando se un pixel è visibile da una certa sorgente di luce ricavando il risultato da un’immagine zbuffer creata dal punto di vista di una certa sorgente di luce e che infine verranno memorizzate in forma di texture.

*Screen Space Ambient Occlusion (SSAO):* Unity presenta SSAO tra l’elenco di image effects in post-rendering che è possibile trovare nell’elenco dei vari componenti standard. Si può aggiungere questo componente a ogni camera per ottenere un’estetica migliore nel gioco ed è molto facile da usare. Si integra completamente con qualsiasi pipeline di rendering scelta. La tecnica dello Screen Space Ambient Occlusion approssima l’Ambient Occlusion nel real-time come effetto d’immagine in post-processing. L’effetto che crea è di rendere più scuri pieghe, buchi e superfici che sono vicini tra di loro ed è simile a come si preserva la luce nella vita reale dove certe zone in cui la luce ambientale viene blocca-

ta appaiono più scure alla vista. L'Ambient Occlusion tende ad approssimare il modo in cui la luce è radiata nella realtà ed è una tecnica di illuminazione globale. Ciò significa che l'illuminazione in ogni punto è in funzione della geometria totale nella scena. Tuttavia è una semplice approssimazione a quella è che l'illuminazione globale. L'effetto ottenuto usando questa tecnica è simile al modo in cui un oggetto potrebbe apparire in una giornata nuvolosa.

*Sun Shafts & Lens Effects:* Unity aggiunge anche Sun Shafts / Godrays (luce volumetrica) come effetto interno al motore. Basta aggiungere il componente ad una Camera e ottenere un effetto immediato di luce volumetrica nell'ambiente circostante. L'effetto di immagine sun shafts simula la dispersione della luce radiale (anche conosciuto come l'effetto del "raggio di Dio") che si verifica quando una fonte di luce molto luminosa è parzialmente oscurata. Inoltre Unity simula anche altri effetti come Lens Flare (bagliori di luce). L'effetto lens flares simula le luci rifrangenti all'interno dell'obiettivo della fotocamera. Viene utilizzato per rappresentare le luci molto luminose o, più in dettaglio, solo per aggiungere un pò più di atmosfera in alcune zone dell'ambiente circostante .



Figura 3.26: Esempio dell'effetto Sun Shafts

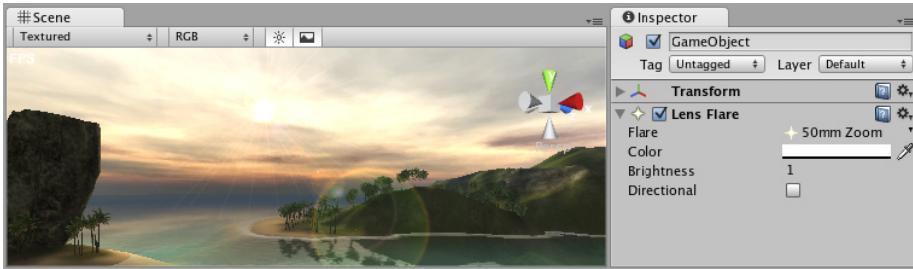


Figura 3.27: Il pannello Lens Flare Inspector

*Lightmapping:* Per avere un controllo preciso dell’illuminazione nell’ambiente del gioco l’utilizzo del lightmapping è l’unico modo di procedere. Con Unity 3 è stato integrato uno dei migliori lightmapper esistenti, ovvero Beast. Grazie a questo strumento è possibile creare le scene di gioco direttamente all’interno di Unity aumentandone la qualità visiva. Una lightmap è una struttura dati che contiene la luminosità delle superfici in applicazioni di grafica 3D come i videogiochi. Le lightmaps sono pre-calcolate, e utilizzate normalmente per gli oggetti statici. Sono particolarmente adatte per ambienti urbani e interni con grandi superfici planari.

*Light Probes (sonde di luce):* Le Light Probes aggiungono realismo alle scene costruite con lightmap, senza aggiungere gli elevati costi di calcolo di luci dinamiche. L’aggiunta di light probes al sistema di illuminazione di Unity è consentita su luci pre-calcolate di personaggi e altri oggetti dinamici. Sebbene il lightmapping aggiunga molto al realismo di una scena, ha lo svantaggio che gli oggetti dinamici della scena sono resi meno realisticamente e il risultato potrebbe non essere corretto. Non è possibile calcolare le lightmap per oggetti in movimento in tempo reale, ma è possibile ottenere un effetto simile utilizzando le light probes. L’idea è che l’illuminazione viene campionata in punti strategici della scena, indicati con le posizioni delle sonde. L’illuminazione in qualsiasi posizione può essere approssimata con un’interpolazione tra i campioni prelevati dalle sonde più vicine. L’interpolazione è abbastanza veloce per essere utilizzata durante il gioco e consente di evitare la separazione tra l’illuminazione di oggetti in movimento e oggetti statici che fanno uso di lightmap nella scena. È necessario piazzare le sonde di luce dove si vogliono creare zone di luce o zone di ombra e formare un volume d’azione tra le varie sonde.

*Dual Lightmapping:* Avere degli ambienti perfettamente illuminati non è tutto. Infatti se si vuole che i personaggi principali si integrino perfettamente con le nostre lightmaps, è possibile usare il dual lightmapping che è supportato

da Unity. In questa tecnica viene usata una lightmap per gli oggetti lontani mentre l'altra lightmap contiene solo la luce che viene riflessa e che rimbalza da ogni direzione. Non è necessario fare l'unwrap dei modelli a mano a meno che non si voglia utilizzarlo per forza, infatti Unity si occuperà di farlo per noi. Per poter creare le lightmap si utilizza il comando di Bake aprendo il pannello delle lightmap e configurando le varie impostazioni. Unity quindi avvierà il processo in background consentendo di continuare a lavorare mentre è in esecuzione. Per un controllo completo è possibile specificare una delle configurazioni messe a disposizione da Beast, in questo modo possiamo sfruttare l'intera gamma di opzioni disponibili dal lightmapper.

*Fisica:* Unity fa utilizzo del potente motore fisico NVIDIA PhysX il quale permette di creare diversi tipi di comportamenti per gli oggetti grazie alle sue innumerevoli funzioni interne. Nel videogioco creato in questa tesi sono state adottate due delle caratteristiche principali offerte dal motore fisico: i Collider e il Raycasting. I Collider sono primitive che definiscono i contorni degli oggetti come potrebbero essere i Rigidbody o i Character Controller e che permettono di creare le collisioni. Infatti per simulare quest'ultime tra gli oggetti nel gioco sono stati usati i Collider combinati all'utilizzo della funzione interna di Unity OnTriggerEnter(). In questo modo ogni volta che il personaggio entra in contatto col Collider di un ostacolo o di un oggetto viene chiamata la funzione OnTriggerEnter() che si occuperà di gestire poi il rallentamento della scena e tutti i comportamenti successivi. Attraverso l'uso del Raycasting si crea un raggio che parte dall'oggetto e va a scontrarsi contro i Collider della scena. Il Raycasting, quindi, è stato usato per la creazione dello script inerente alle meccaniche di shooter. Il Raycasting ha permesso di creare un fascio di luce che identifica il collider intersecato e grazie a tale informazione è stato possibile gestire la logica del danno inflitto e subito per ogni oggetto presente sulla scena. Per ulteriori informazioni riguardanti Unity si demanda alla documentazione ufficiale consultabile da <https://docs.unity3d.com/Manual/index.html>



*Animation Tools:* Questo tool permette di creare animazioni da assegnare ai vari GameObject. Con la parola animazione non si intende solamente la traslazione e la rotazione ma anche qualsiasi cambio di proprietà dell'oggetto: è possibile far cambiare il colore di un oggetto, la variabile pubblica di uno script e tante altre cose. L'animation tool si presenta principalmente con due fogli di lavoro: il Dope Sheet e Curves. Il Dope sheet è semplicemente un foglio dove creando delle Key è possibile memorizzare dei cambi di proprietà nel tempo. Una volta create delle Key, e quindi dei cambi di proprietà, su Curves possiamo

osservare la curva che è stata generata. In questo foglio di lavoro è possibile modificare l'andamento dell'animazione nel tempo tramite l'utilizzo di strumenti come: smooth, flat e broken.

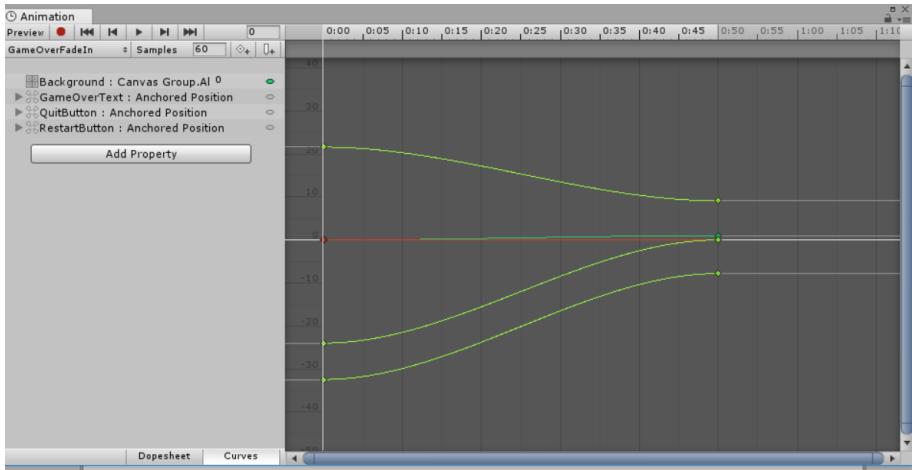


Figura 3.28: Animation Tools

Le varie animazioni possono essere concatenate tra di loro grazie all'utilizzo dell'Animator Tools di supporto per la gestione delle animazioni e che fornisce la base per creare StateChart comportamentali. Grazie all'Animator Tool è possibile creare degli stati in cui l'oggetto può ritrovarsi ed impostare diversi meccanismi come l'attivazione di animazioni specifiche, traslazioni, rotazioni ecc. È possibile inoltre concatenare diversi stati tra loro per creare un automa a stati finiti e specificare i trigger/condizioni di transizione per ogni variabile dell'oggetto o per un sottoinsieme delle variabili. All'interno dell'Animator si possono specificare vari modalità di transizione tra stati che permettono uno switch repentino, smoothed o concatenato che rispettivamente: bloccano l'animazione appena avviene la transizione, bloccano la transizione in maniera "morbida" rendendo il passaggio più fluido, attende il terminare dell'animazione anche se è già partito il trigger di transizione.

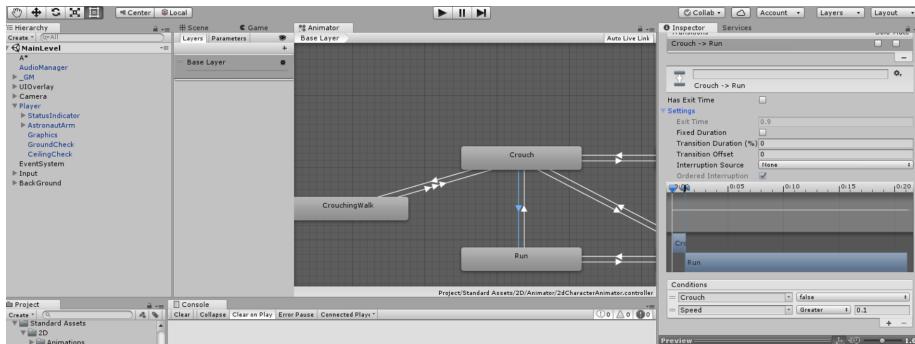


Figura 3.29: Animator Tools

*Prefabs:* Unity permette il salvataggio di oggetti creati nella scena tramite l'utilizzo dei prefab, elementi creabili dalla finestra project che formano un collegamento tra tutti i GameObject dello stesso tipo. Il vantaggio di utilizzare un prefab risiede non solo nel poter salvare all'interno del progetto i GameObject più comuni, ma anche nel permettere che una modifica effettuata su di esso si ripeta su tutte le copie di quell'oggetto, rendendo molto più facile la gestione delle copie.

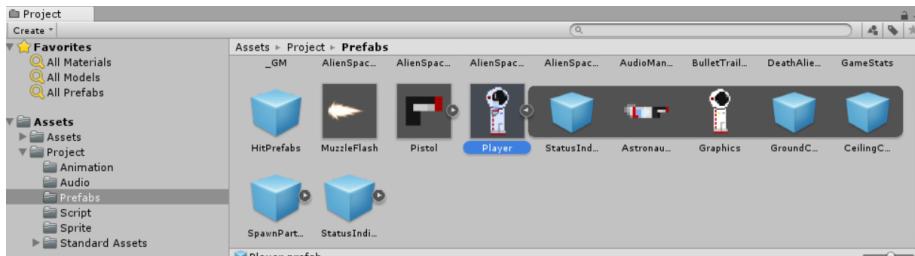


Figura 3.30: Star Naitum Prefabs

### 3.2.2 Unity Assets Store

*Unity Assets Store* ospita una crescente libreria di risorse gratuite e commerciali create sia da Unity Technologies che dai membri della comunità. È disponibile un'ampia varietà di risorse, che coprono tutti gli elementi: da texture, modelli e animazioni a interi esempi di progetti, script e tutorial ed estensioni di editor. Le risorse sono accessibili da una semplice interfaccia integrata nell'editor Unity e vengono scaricate ed importate direttamente nel progetto, il tutto gestito direttamente da Unity. Un Asset (“attività”) è la rappresentazione di qualsiasi oggetto che può essere utilizzato nel progetto. Un Asset può provenire

da un file creato al di fuori di Unity, come un modello 3D, un file audio, un'immagine o uno degli altri tipi di file supportati da Unity. Esistono anche alcuni tipi di risorse che possono essere creati all'interno di Unity, come un controller di animazione, un mixer audio o una texture.

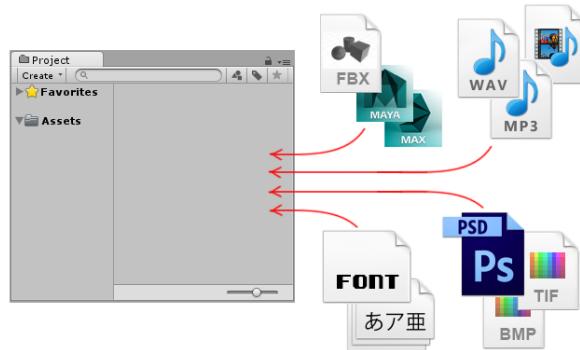


Figura 3.31: Assets unity

Gli *Assets* creati al di fuori di Unity devono essere portati in Unity facendo in modo che il file venga salvato direttamente nella cartella “Assets” del progetto, o copiato in quella cartella. Per molti formati comuni, è possibile salvare il file sorgente direttamente nella cartella Assets del progetto e Unity sarà in grado di leggerlo. Unity noterà quando si salvano le nuove modifiche al file e le reimporterà se necessario. Quando si crea un progetto Unity, si crea una cartella, dal nome del progetto, che contiene le seguenti sottocartelle:

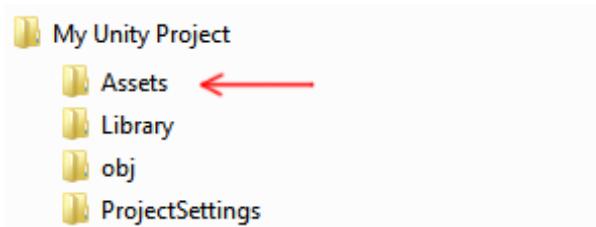


Figura 3.32: Cartelle base di un progetto Unity

*La struttura di base dei file di Unity Project:* Nella cartella Assets è necessario salvare o copiare i file che si desidera utilizzare nel progetto. Il contenuto della finestra Project in Unity mostra gli elementi nella cartella Assets. Quindi, se si salva o copia un file nella cartella Assets, verrà importato e reso visibile nella finestra del progetto. Unity rileverà automaticamente i file man mano

che vengono aggiunti alla cartella Assets o se vengono modificati. Quando si inserisce una risorsa nella cartella Assets, si vedrà la risorsa apparire nella vista Project.

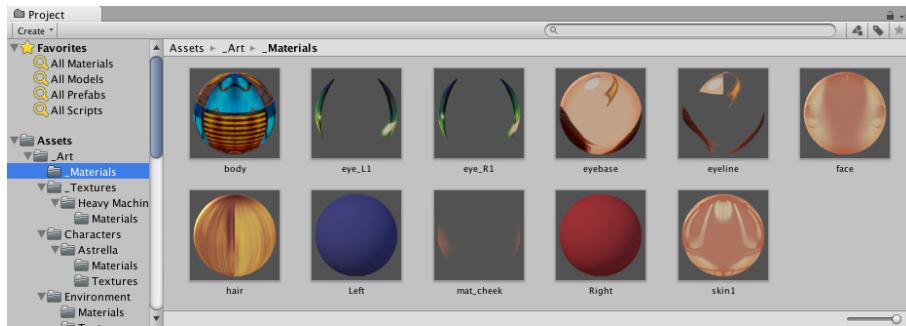


Figura 3.33: Unity project browser

La finestra del progetto mostra le risorse che sono state importate nel progetto. Se si trascina un file nella finestra del progetto di Unity dal computer (ad esempio, dal Finder su Mac o da Explorer su Windows), verrà copiato nella cartella Assets e verrà visualizzato nella finestra Progetto. Gli elementi visualizzati da quest'ultima, rappresentano (nella maggior parte dei casi) file effettivi sul computer e, se si eliminano all'interno di Unity, si elimineranno anche dal computer.

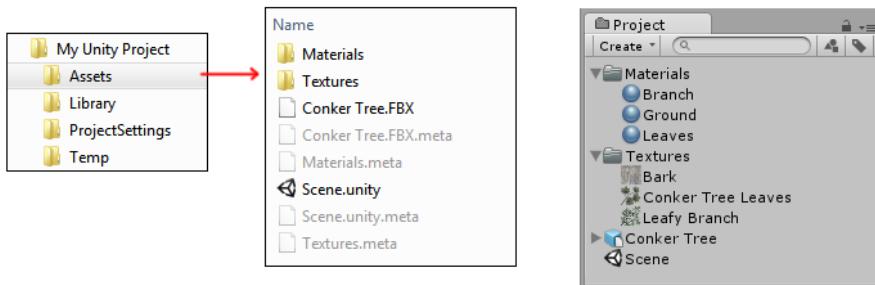


Figura 3.34: Assets workflow

L'immagine sopra mostra un esempio di alcuni file e cartelle all'interno della cartella Risorse di un progetto Unity. Si possono creare tutte le cartelle che si vuole e usarle per organizzare i propri Assets. Nell'immagine sopra si nota che ci sono file .meta elencati nel file system, ma non visibili nella finestra del progetto di Unity. Unity crea questi file .meta per ogni risorsa e cartella, ma sono nascosti di default, quindi rimarranno non visibili anche in Explorer /

Finder. Contengono informazioni importanti su come l'Assets è utilizzato nel progetto e devono rimanere con il file di Assets a cui si riferiscono, quindi se si sposta o si rinomina un file di Assets in Explorer / Finder, è necessario spostare / rinominare il meta file per farlo corrispondere. Il modo più semplice per spostare o rinominare le risorse in modo sicuro è sempre farlo dalla cartella del progetto di Unity. In questo modo, Unity sposta o rinomina automaticamente il meta file corrispondente. Se si desidera portare collezioni di Assets nel progetto, è possibile utilizzare gli Asset Packages. Alcuni tipi comuni di Assets fondamentali sono:

- File di immagine: Sono supportati i tipi di file di immagine più comuni, come BMP, TIF, TGA, JPG e PSD. Se si salva i file Layer di Photoshop (.psd) nella cartella Risorse, verranno importati come immagini a Layer unico.

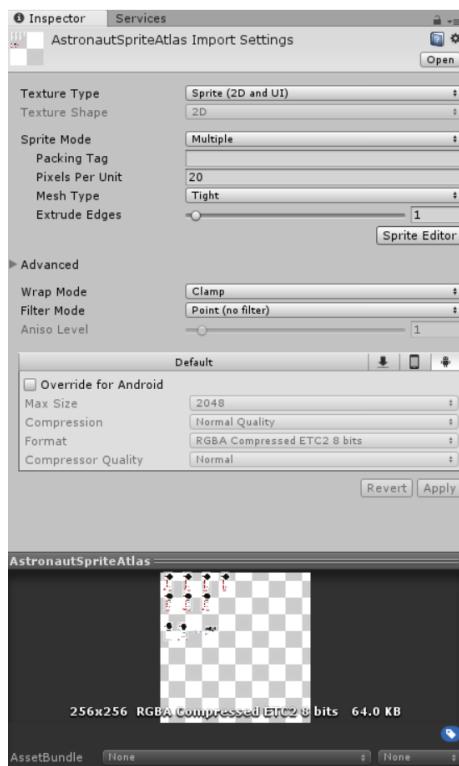


Figura 3.35: Proprietà Immagine

- Modelli 3D: Se si salvano i file 3D dai pacchetti software 3D più comuni nel loro formato nativo (ad esempio .max, .blend, .mb, .ma) nella cartella Risorse, verranno importati richiamando il plug-in di esportazione FBX

del pacchetto 3D. In alternativa si possono esportare come FBX dall'app 3D nel tuo progetto Unity.

- Mesh e animazioni: Indipendentemente dal pacchetto 3D che si sta utilizzando, Unity importerà le mesh e le animazioni da ciascun file. I file mesh non hanno bisogno di avere un'animazione da importare. Se si utilizzano le animazioni, è possibile scegliere di importarle tutte da un singolo file o importare file separati, ciascuno con un'animazione.
- File audio: Se si salvano file audio non compressi nella cartella Risorse, verranno importati in base alle impostazioni di compressione specificate.

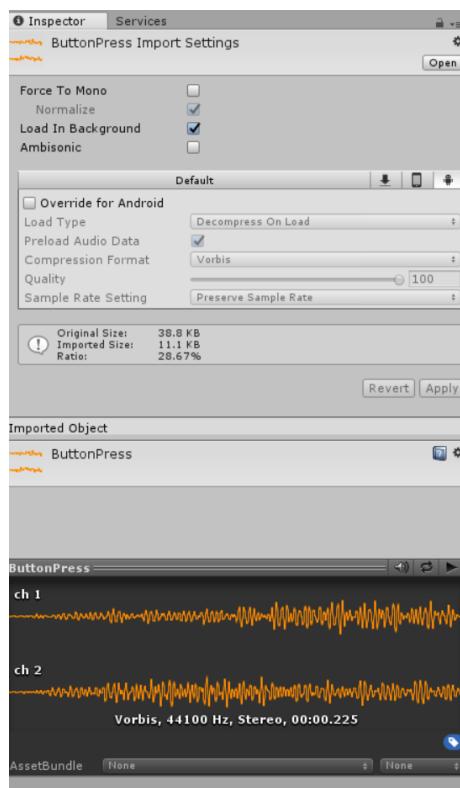


Figura 3.36: Proprietà Audio

- Altri tipi di Assets: In tutti i casi, il file sorgente originale non viene mai modificato da Unity, anche se all'interno di Unity si può spesso scegliere tra vari modi per comprimere, modificare o altrimenti elaborare l'Assets. Il processo di importazione legge il file sorgente e crea internamente una rappresentazione pronta per il gioco della risorsa, facendo corrispondere le impostazioni di importazione scelte. Se si modificano le impostazioni di

importazione per una risorsa o si apporta una modifica al file di origine nella cartella Risorse, l'unità verrà importata nuovamente per riflettere le nuove modifiche.



Figura 3.37: Esempio di pagina web Unity Assets store

L'importazione di formati 3D nativi richiede l'installazione dell'applicazione 3D sullo stesso computer di Unity. Questo perché Unity usa il plug-in FBX di esportazione dell'applicazione 3D per leggere il file. In alternativa, puoi esportare direttamente come FBX dalla tua applicazione e salvarlo nella cartella Progetti. Gli utenti di Unity possono diventare editori su Assets Store e vendere i contenuti che hanno creato. È possibile aprire la finestra Assets Store selezionando Window/Assets Store dal menu principale. Durante la tua prima visita, ti verrà richiesto di creare un account utente gratuito che utilizzerai per accedere successivamente allo Store.

*Packages di Unity:* I packages Unity sono un modo pratico per condividere e riutilizzare i progetti Unity e le raccolte di Assets; I pacchetti sono raccolte di file e dati da progetti Unity o elementi di progetti, che vengono compressi e archiviati in un unico file, in modo simile ai file Zip. Come i file Zip, un pacchetto mantiene la sua struttura di directory originale quando viene decompressa, così come i meta-dati sulle risorse (come le impostazioni di importazione e i collegamenti ad altre risorse). In Unity, l'opzione di menu Esporta pacchet-

to comprime e archivia la raccolta, mentre Importa pacchetto decomprime la raccolta nel progetto Unity attualmente aperto.

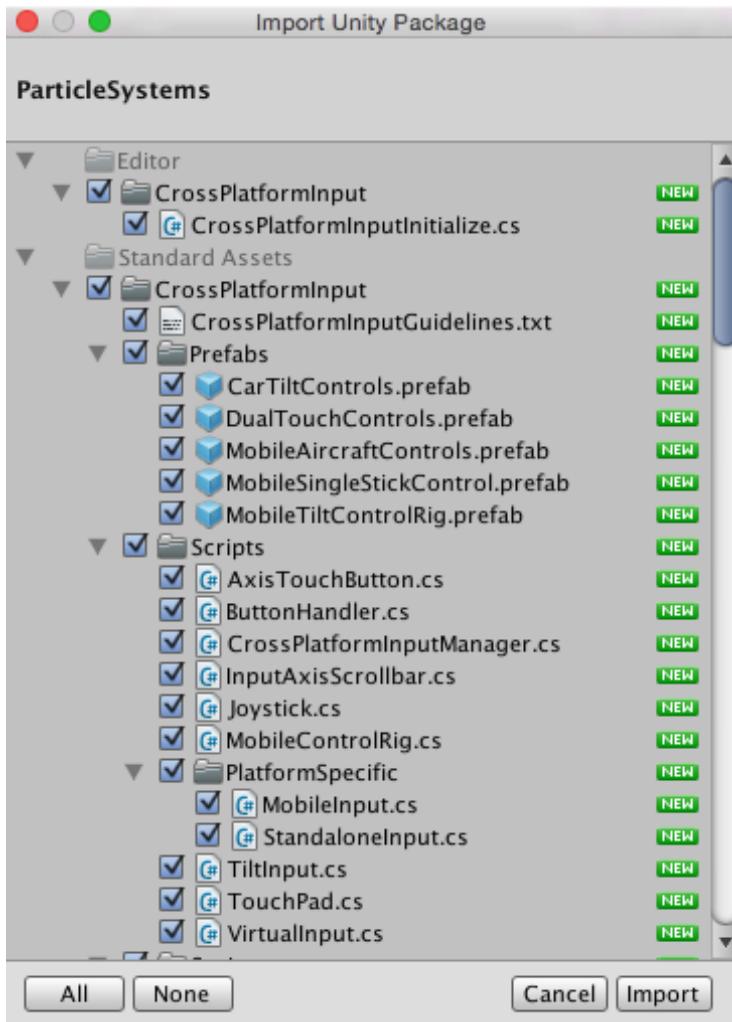


Figura 3.38: Pannello di importazione Assets

Unity viene fornito con più Standard Assets (raccolta di Assets base per la creazione di videogiochi). Queste sono raccolte di risorse ampiamente utilizzate dalla maggior parte dei clienti di Unity. Questi comprendono package: 2D, Telecamere, Caratteri, CrossPlatformInput, Effetti, Ambiente, ParticleSystems, Prototipazione, Utility, Veicoli.

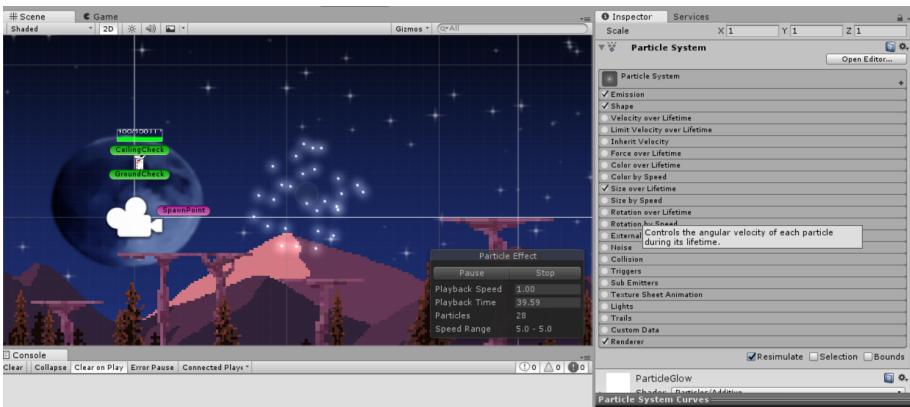


Figura 3.39: StandardAssets ParticleSystem

### 3.2.3 Platform Dependent Compilation

Unity include una funzionalità chiamata **Platform Dependent Compilation**. Questo consiste in alcune direttive del preprocessore che permettono di partizionare gli script per compilare ed eseguire una sezione di codice esclusivamente per una delle piattaforme supportate da Unity. Questa è una caratteristica molto importante in quanto permette di inserire dei costrutti, all'interno degli script che governano il gioco, in modo da compilare solo la parte desiderata.

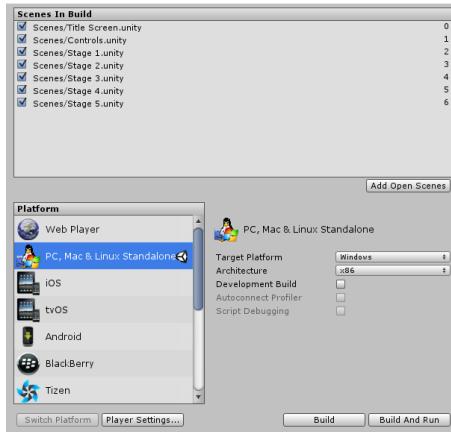


Figura 3.40: Pannello di Build Unity

Questa caratteristica è stata ampiamente utilizzata nello sviluppo di “Star Naitum” in quanto il videogioco è stato sviluppato sia per sistemi Windows e Mac Based che per sistemi Android e quindi risultava molto faticoso e contropro-

ducente dedicarsi ad una sola piattaforma e poi effettuare il “Porting” sull’altra. Questa funzionalità di Unity ha permesso al progetto di evolvere senza appoggiarsi in modo costante ad una piattaforma specifica ma le nuove caratteristiche venivano inserite nello stesso progetto e successivamente confinate all’interno di appositi costrutti `#if` che governano la compilazione del codice appropriato per ogni piattaforma disponibile.

```
#if UNITY_EDITOR
    Debug.Log("Unity Editor");

#elif UNITY_IOS
    Debug.Log("Unity iPhone");

#else
    Debug.Log("Any other platform");

#endif
```

Per ulteriori informazioni:

<https://docs.unity3d.com/Manual/PlatformDependentCompilation.html>

### **3.3 Modellazione e Design**

La modellazione tridimensionale o modellazione 3D è il processo atto a definire una forma tridimensionale in uno spazio virtuale generata su computer; questi oggetti, chiamati modelli 3D vengono realizzati utilizzando particolari programmi software, chiamati modellatori 3D, o più in generale software 3D. La storia della Computer grafica 3D è naturalmente molto recente, lo stesso termine di grafica computerizzata nasce solo nel 1960. Una delle prime rappresentazioni tridimensionali su calcolatore è stata quella del famoso “primo uomo” o “Boeing Man” realizzata da William Fetter; un insieme di linee che descrivevano la sagoma virtuale di un pilota di aereo. A partire dal 1959, la General Motors, in collaborazione con la IBM, sviluppa il sistema “DAC”, uno dei primi sistemi CAD; attraverso una penna ottica e uno schermo sensibile, gli operatori disegnavano delle curve matematiche in uno spazio virtuale, con le quali delimitavano i profili, le sezioni e le superfici delle automobili. Della prima metà degli anni sessanta è anche il sistema chiamato “Adage”, considerata da molti la prima workstation CAD indipendente. La nascita della modellazione 3D è dunque avvenuta in ambito industriale, primariamente come supporto alla progettazione. Da allora i campi di utilizzo della modellazione 3D e della

grafica tridimensionale si sono enormemente ampliati, uscendo in buona parte dall'ambito tecnico. Da un punto di vista tipologico, tutta la modellazione 3D, rientra in due grandi famiglie, ognuna riguardante un ben determinato genere di modelli:

- La Modellazione organica: è la tipica modellazione utilizzata per realizzare gli esseri umani o le creature, animali o umanoidi. Viene usata per tutti i soggetti “naturali”, come rocce, piante, alberi e per il territorio in generale, in questi casi i modelli sono tanto più riusciti quanto più sono ricchi di particolari. Anche molti oggetti di disegno industriale, che abbiano forme morbide e arrotondate, possono servirsi di una modellazione organica.
- La Modellazione geometrica: è il tipo di modellazione meno recente, viene utilizzata per realizzare oggetti tecnici o meccanici, o comunque per qualsiasi cosa che abbia una natura artificiale, e che non rientri nella categoria precedente. Generalmente la complessità dei modelli realizzati con questo genere di modellazione è molto inferiore, se si guarda all'aspetto esteriore delle singole forme, ma non se si considerano aspetti legati alla precisione e alla corrispondenza delle parti.

Naturalmente uno stesso oggetto può contenere sia modellazione organica che geometrica, oppure può essere formato da un insieme di parti contenenti sia modelli organici che geometrici.

Modello 3D da realizzare: (Volto Umano)		
▼		
Tipologia di Modellazione: (Modellazione Organica)		
▼		
Utilizzo del Modello:		
Videogame Realtime	Animazione facciale Lipsinc	stampa in Stereolitografia
SISTEMA DI MODELLAZIONE "A" Modellazione poligonale Low Poly	SISTEMA DI MODELLAZIONE "B" Modellazione poligonale + Superficie di Suddivisione	SISTEMA DI MODELLAZIONE "C" Modellazione poligonale + Scultura 3D + Conversione STL
▼	▼	▼
CARATTERISTICHE DEL MODELLO numero minimo di poligoni mesh molto leggera	CARATTERISTICHE DEL MODELLO superficie ottimizzata per l'animazione	CARATTERISTICHE DEL MODELLO altissimo numero di poligoni mesh molto pesante

Figura 3.41: Tabella di classificazione e tipologie di modellazione

*Tecniche di modellazione 3D:* Si possono dividere in tre categorie principali:

1. Modellazione Procedurale (automatica e semi-automatica).

2. Modellazione Manuale.
3. Da dati provenienti da modelli reali (scansione tridimensionale).

Che a loro volta possono venire suddivise in tre distinti generi di modellazione:

1. Modellazione Solida - dove l'oggetto risultante è considerato come formato da un volume pieno.
2. Modellazione Volumetrica - determina delle entità generanti una superficie implicita.
3. Modellazione di superfici - l'oggetto in questo caso è determinato dalle sue superfici esterne.

In alcuni modellatori un oggetto è considerato formato da superfici finché queste sono aperte, mentre viene riconosciuto come solido una volta che tutte le superfici siano saldate fra di loro e formino un corpo chiuso. Il seguente elenco esamina le diverse tecniche di Modellazione Manuale. Alcune delle tecniche descritte (come ad es. le superfici patch), essendo abbastanza dattate, risultano essere superate e obsolete rispetto a tecniche più recenti e avanzate. Malgrado questo alcuni Modellatori 3D, mantengono al loro interno alcuni di questi strumenti come accessori o utilità. Si tratta di tecniche basilari nell'ambito della grafica 3D. La modellazione poligonale opera su superfici organizzate in maglie più o meno dettagliate di facce poligonali. Queste superfici possono solo approssimare l'oggetto finale se siamo in presenza di un basso livello di poligoni (in questo caso l'oggetto viene detto Low Poly). In altri casi un modello poligonale, a modellazione ultimata, può essere formato anche da un numero molto elevato di facce. I seguenti sistemi procedono dai più elementari ai più evoluti:

*Per spostamento di elementi:* un modello poligonale è formato da 3 elementi essenziali: facce, lati e vertici; lo spostamento arbitrario di un singolo elemento o di gruppi di essi, determina una modifica della mesh di partenza. La selezione di un componente della mesh e il suo spostamento (trascinamento, rotazione, ridimensionamento ecc), nello spazio è la tecnica più elementare di modellazione poligonale.

*Da primitive di base:* Uno dei sistemi più semplici e diretti per iniziare a modellare un oggetto poligonale, è quello di partire da una primitiva poligonale di base, e iniziare a modificarla spostando, ruotando, scalando i suoi componenti, fino a ottenere la forma voluta. Questa tecnica è molto semplice, ma consente in genere di ottenere modelli poco complessi, vincolati cioè alla complessità (anche

in termini di densità poligonale della mesh) della primitiva di partenza.

*Metodo della mesh piana:* oltre a modificare i poligoni di mesh esistenti (ad es. delle primitive), esiste la possibilità di creare singolarmente ogni poligono dell'oggetto e di costruire i poligoni nella posizione più comoda per realizzare il modello finale. Uno dei sistemi di disegno diretto dei poligoni viene detto Metodo della mesh piana. Si tratta in sostanza di creare una griglia di poligoni posizionati in piano e aventi la struttura il profilo e la conformazione generale dell'oggetto finale. Posizionati i poligoni sul piano, si passa a determinarne la tridimensionalità: o spostando i punti della griglia lungo la profondità del modello, o attraverso dei sistemi di estrusione.

*Metodo a tela di ragno:* Si tratta di una variante della precedente tecnica. In questo caso non si costruiscono e posizionano tutti i poligoni di base del modello, ma si parte da una sua zona (centrale), e si iniziano a creare e modellare i singoli poligoni con un sistema appunto a “tela di ragno”, cioè dall'interno e procedendo man mano verso le zone esterne del modello. È un sistema complesso e dispendioso in termini di tempo, utilizzato soprattutto per il suo alto grado di precisione.

*Per Rifinitura Progressiva:* è il sistema più evoluto, può considerarsi uno dei paradigmi della Modellazione 3D. Adottando un qualsiasi metodo analizzato precedentemente si inizia a definire la forma in una maniera molto schematica, perlopiù approssimandone la morfologia e facendo attenzione a tenere estremamente basso il numero iniziale di poligoni. Dovendo gestire pochi poligoni è possibile modificare molto agevolmente le proporzioni e il volume generale della forma. Solo quando si è soddisfatti dell'aspetto grezzo del modello si può iniziare, adottando gli specifici strumenti di ogni pacchetto software, a definire maggiormente la forma. È importante che a ogni passaggio di rifinitura si passi a definire prima i volumi maggiori del modello, per andare poi a definire le zone sempre più piccole, la definizione e il numero di dettagli apportabili è a discrezione del grafico 3D. Il principio fondamentale da tenere a mente è che: tanto minore è il numero di poligoni presenti nel modello, tanto maggiore è la possibilità di modificarne la morfologia generale - tanto maggiore è il numero di poligoni tanto meno si potrà modificare la forma già impostata in precedenza. In pratica ogni passaggio è irreversibile, tanto più si definiscono i particolari dell'oggetto, tanto meno si potrà modificare (o correggere) il suo aspetto generale. A questo problema si può porre rimedio salvando il modello in maniera progres-

siva, in modo da avere a disposizione tutti i passaggi intermedi di modellazione, in caso di errore si può ripartire dal modello precedente a minore dettaglio, se il software utilizzato fa uso dei layer, è possibile conservare le varie versioni in layer separati.

*Principi di corretta modellazione:* Per comprendere quale debba essere il giusto utilizzo dei vari sistemi di modellazione bisognerebbe introdurre il concetto di Modello 3D corretto e Modello 3D scorretto. Si deve cioè spostare l'attenzione dall'aspetto tecnico della modellazione a un'analisi attenta del modello da realizzare. Il processo di modellazione deriva primariamente dalla tipologia del modello da realizzare. La tipologia del modello comporterà una prima scelta tra tecniche di modellazione organica e tecniche di modellazione geometrica (non avrebbe senso approcciare la modellazione di un componente meccanico con delle tecniche organiche; come sarebbe un nonsenso voler realizzare una mano umana con un sistema CAD), questo perché ogni tipologia di oggetto è associabile in maniera naturale a determinate tecniche e non a altre. Ciò che condizionerà la scelta specifica del sistema di modellazione, saranno invece le caratteristiche richieste al modello dalla sua destinazione d'uso. Un modello 3D molto bello da vedersi non è necessariamente eseguito correttamente: perché potrebbe essere inadatto all'utilizzo cui è destinato (ad es. il modello 3D di un'automobile da usarsi in un videogioco, sarà necessariamente diverso dal modello CAD della stessa automobile da utilizzarsi per la produzione di serie). Si adotterà una tecnica di modellazione corretta se sarà adeguata primariamente alla tipologia del modello e secondariamente al suo utilizzo finale.

### **3.3.1 Blender**

Blender è un software libero e multipiattaforma 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. È disponibile per vari sistemi operativi: Microsoft Windows, macOS, GNU/Linux, FreeBSD, assieme a porting non ufficiali per BeOS, SkyOS, AmigaOS, MorphOS e Pocket PC. Blender è dotato di un robusto insieme di funzionalità paragonabili, per caratteristiche e complessità, ad altri noti programmi per la modellazione 3D come Softimage XSI, Cinema 4D, 3D Studio Max, LightWave 3D e Maya. Tra le funzionalità di Blender vi è anche l'utilizzo di raytracing e di script (in Python).



Figura 3.42: Blender

In origine, il programma è stato sviluppato come applicazione interna dallo studio di animazione olandese NeoGeo. L'autore principale, Ton Roosendaal, fondò la società Not a Number Technologies (NaN) nel 1998 per continuare lo sviluppo e distribuire il programma che inizialmente fu distribuito come software proprietario a costo zero (freeware) fino alla bancarotta di NaN nel 2002. I creditori acconsentirono a pubblicare Blender come software libero, sotto i termini della licenza GNU General Public License, per il pagamento una-tantum di 100.000,00 euro. Il 18 giugno 2002 fu iniziata da Roosendaal una campagna di raccolta fondi e il 7 settembre 2002 fu annunciato che l'obiettivo era stato raggiunto e il codice sorgente di Blender fu pubblicato in ottobre. Ora Blender è un progetto open source molto attivo ed è guidato dalla Blender Foundation. Blender richiede poco spazio per essere installato e può essere eseguito su molte piattaforme. Sebbene sia spesso distribuito senza documentazione o esempi il software è ricco di caratteristiche tipiche di sistemi avanzati di modellazione. Tra le sue potenzialità, possiamo ricordare: Supporto per una grande varietà di primitive geometriche, incluse le mesh poligonali, le curve di Bézier, le NURBS, le metaball e i font vettoriali. 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 attraverso il Blender Game Engine, 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 di piccole dimensioni. Blender fornisce una serie di feature quali:

- Motore di rendering interno versatile ed integrazione nativa col motore esterno YafRay (un raytracer open source)

- Motore di rendering unbiased Cycles disponibile internamente a partire da Blender 2.61
- Scripting in python per automatizzare e/o controllare numerosi aspetti del programma e della scena
- Effetti particellari gestiti dal programma

Blender ha fama di essere un programma difficile da imparare. Quasi tutte le funzioni possono essere richiamate con scorciatoie e per questo motivo quasi tutti i tasti sono collegati a numerose funzioni. Da quando è stato pubblicato come opensource, la GUI è stata notevolmente modificata, introducendo la possibilità di modificare il colore, l'uso di widget trasparenti, una nuova e potenziata visualizzazione e gestione dell'albero degli oggetti e altre piccole migliorie (scelta diretta dei colori ecc). L'interfaccia di Blender si basa sui seguenti principi:

- Modalità di modifica (edit): le due modalità principali sono la Modalità oggetto (object mode) e la Modalità modifica (edit mode), ed è possibile passare dall'una all'altra per mezzo del tasto tab. La modalità oggetto può essere usata per manipolare oggetti singoli, mentre la modalità modifica è usata per modificare i dati di un oggetto. Per esempio, in una mesh poligonale, la modalità oggetto può essere usata per muovere, scalare e ruotare l'intera mesh, mentre la modalità modifica è usata per modificare i vertici individuali della mesh. Ci sono anche altre modalità, come la Pittura Vertici, la Modalità Scultura o la Pittura Pesi.
- Scorciatoie da tastiera: la maggior parte dei comandi è impartibile attraverso la tastiera. Fino alla versione 2.x e specialmente nella versione 2.3x, questo era il solo modo per impartire comandi, e questo è stato il principale motivo che ha dato a Blender la reputazione di essere un programma difficile da imparare e capire. Le nuove versioni hanno menù molto più completi, che permettono di usare in larga misura il mouse per impartire i comandi.
- Spazio di lavoro completamente ad oggetti: l'interfaccia di Blender è formata da una o più scene, ognuna delle quali può essere divisa in sezioni e sottosezioni che possono essere formate da una qualunque immagine o vista di Blender. Ogni elemento grafico delle viste di Blender può essere controllato nello stesso modo in cui si controlla la finestra 3D. Si possono ad esempio ingrandire i pulsanti della barra dei menù nello stesso modo in cui si ingrandisce un'immagine nella finestra di anteprima. La disposizione delle componenti dell'interfaccia di Blender è modificabile dall'utente,

che può così lavorare a compiti specifici su un'interfaccia personalizzata e nascondere le caratteristiche non necessarie.

A partire dello sviluppo delle versioni 2.5x, è stata introdotta una nuova interfaccia, oltre al cambiamento di alcune combinazioni di tasti, rendendo il tutto più intuitivo per chi è alle prime armi con il programma.



## **Capitolo 4**

# **Progetti “Star Naitum” e “HologramRun!”**

### **4.1 Pre-Produzione**

In questo capitolo si andranno a definire tutti gli aspetti affrontati durante lo sviluppo di due progetti videoludici, uno più standard denominato “Star Naitum” e un altro più innovativo denominato “HologramRun!”. In questa sezione del capitolo saranno descritte le scelte e la fase di Pre-Produzione che sono condivise tra i due progetti per poi andare ad approfondire le tematiche sullo sviluppo dei due videogiochi in maniera divisa. Lo sviluppo dei progetti è stato delineato dall'applicazione della metodologia Game-Scrum laddove fosse stato possibile, si è cercato di rimanere fedeli alle fasi e agli step imposti da tale metodologia per testarla ed accumulare informazioni utili per la maturità della metodologia stessa. La prima scelta è stata la tipologia di gioco, inizialmente è stata effettuata un'analisi sulla quale si sarebbe potuta sviluppare la giusta tipologia di gioco, mantenendo l'attenzione sul tempo a disposizione (Circa due/tre mesi). Proprio per questo motivo si sono accantonati tipologie di videogiochi dove la qualità della storia risultava una caratteristica predominante, in quanto i tempi si sarebbero allungati di molto, o dove la parte artistica risultava predominante, creazioni di musiche o modelli 3D avrebbero comportato lo stesso problema, quindi si è optato per un gioco Platform 2D ad orde proprio perché non necessità di storia, ma solo di un pretesto per essere giocato, e dove la parte artistica non è predominante in quanto, in questa tipologia di gioco, conta più il gameplay puro per arrivare ad un punteggio più alto e trasmettere al giocatore un senso di soddisfazione e crescita delle proprie skills

che si sviluppa e si apprende solo giocando, proprio questo aspetto consiste nel fattore divertimento di Star Naitum che verrà continuamente monitorato lungo tutto il processo di sviluppo. Quindi ci si è dedicati più ad ottenere il giusto grado di sfida, di feedback con i comandi e di apprendimento delle meccaniche rispetto alla storia e alla grafica. Il secondo lavoro sviluppato invece ha come obiettivo quello di portare un prodotto diverso dallo standard videoludico, cercando di unire il videogioco con una tecnica visiva datata, ma estremamente soddisfacente in ambito videoludico, quale l'olografia. Il fattore divertimento, nel secondo prodotto consiste, sia nell'applicazione dell'olografia che stimola la curiosità degli utenti, e sia nella sfida che si vuole proporre grazie alla tipologia, anch'essa Platform, del videogioco proposto. Una volta individuata la tipologia di videogiochi da sviluppare la scelta cardine è stata quella del Game Engine ovvero il motore grafico e del linguaggio di programmazione. Per questo si è optato per Unity 2017.1, ultima versione ad oggi rilasciata. La scelta è ricaduta su Unity per molteplici motivi: il software dalla versione 2017 in poi è rilasciato completamente gratuito dal sito [www.unity3D.com](http://www.unity3D.com) compreso di tutte le features; Le versioni a pagamento forniscono solo dei vantaggi a livello di pubblicazione su store, diritti e add-on che, per giochi di questa dimensione, possono benissimo essere accantonati. Unity ha un'interfaccia grafica facile da comprendere e intuitiva, approfondita nei capitoli precedenti, e permette di avere una finestra dedicata al videogioco dove è possibile vedere a run-time le modifiche senza dover prima costruire la build definitiva. Unity fa una netta distinzione tra prodotti 2D e 3D e questo favorisce lo sviluppo in quanto ci sono oggetti (sprite, layer ecc) dedicati al 2D mentre altri (texture, modelli 3D ecc) dedicate al 3D, questo evita di fare confusione utilizzando oggetti dedicati al 3D piuttosto che al 2D. La documentazione, che si trova sul sito ufficiale, spiega in maniera dettagliata ogni aspetto del software con esempi e video appositi per il learning del software e di alcune tecniche particolari usate dagli sviluppatori in tale ambito. La caratteristica che forse più di tutte ha fatto pendere l'ago della bilancia verso Unity è stato l'Assets Store. L'Assets Store di Unity è una repository di qualsiasi tipo di file importabile su Unity e permette agli sviluppatori meno esperti in grafica 3D e 2D di trovarsi un numero immenso di modelli, personaggi, ambientazioni e addirittura script pre-fabbricati da altri utenti o dalla community di Unity. Per ovvie ragioni quelli più elaborati vengono erogati a pagamento ma il resto è completamente gratuito e più volte capiterà di stare ore a scegliere il giusto Assets per la propria creazione. Grazie all'applicazione di Game-Scrum, che ha portato un approccio sistematico allo sviluppo, c'è stata la necessità di avere solo una minima quantità di elaborati (documento

di Design minimale e traduzione in Backlog) in quanto l'approccio è orientato allo sviluppo incrementale del prodotto e alla prototipazione piuttosto che alla creazione di documenti. Il tutto viene scandito da cicli in cui il software viene di fatto testato dagli stackholder. Nel nostro caso gli stackholder (le persone che potenzialmente potrebbero interagire con i prodotti sviluppati) sono formati da persone, appassionate e non di videogames, di età compresa tra i 9 e i 60 anni, che si sono prestate al testing dei prodotti e successivamente alla compilazione di un questionario (composto da 8 domande aperte e 8 chiuse) per catturare il feedback, sui vari aspetti del gameplay, importantissimo soprattutto nella prima fase del gioco in cui si ricerca il fattore divertimento dei prodotti.

Nota: Le domande sono strutturate come segue:  
 - Domanda a risposta chiusa. Risposta in scala decimale [1-10]  
 - Domanda aperta.

La domanda aperta può contenere suggerimenti soggettivi e preferenza di qualsiasi genere. La domanda aperta, nel caso in cui la persona sia pienamente soddisfatta della meccanica proposta dal gioco, può anche essere lasciata vuota.

1. Come valuti il divertimento del gioco? voto: 1  
Spiega: In poche righe, quale meccanica vorresti inserita all'interno del gioco per aumentare il voto dato:  
 \_\_\_\_\_  
 \_\_\_\_\_  
 \_\_\_\_\_

2. Come valuti i comandi e la facilità di interagire con il gioco? voto: 1  
Spiega: In poche righe, cosa miglioreresti nei comandi per aumentare il voto dato:  
 \_\_\_\_\_  
 \_\_\_\_\_  
 \_\_\_\_\_

3. Come valuti lo stile e la qualità grafica del gioco? voto: 1  
Spiega: In poche righe, cosa miglioreresti nella grafica del gioco per aumentare il voto dato:  
 \_\_\_\_\_  
 \_\_\_\_\_  
 \_\_\_\_\_

4. Come valuti la sfida proposta dal gioco? voto: 1  
Spiega: In poche righe, che tipo di videogiocatore sei [casual, ~~popolare~~, hardcore] e cosa cambieresti nella difficoltà del gioco per aumentare il voto dato:  
 \_\_\_\_\_  
 \_\_\_\_\_

5. Quanto tempo trascorri mediamente, in un giorno, a giocare al videogioco proposto? minuti: 1 | ore [ ]  
Spiega: In poche righe, cosa miglioreresti/cambieresti per aumentare il tempo di gioco:  
 \_\_\_\_\_

6. Come valuti gli effetti speciali, effetti audio e la musica? voto: 1  
Spiega: In poche righe, cosa miglioreresti per aumentare il voto dato:  
 \_\_\_\_\_  
 \_\_\_\_\_  
 \_\_\_\_\_

7. Come valuti l'effetto olografico? voto: 1  
Spiega: In poche righe, che tipologia di gioco vorresti vedere per aumentare il voto dato:  
 \_\_\_\_\_  
 \_\_\_\_\_  
 \_\_\_\_\_

8. Come valuti l'interazione tramite gli auricolari? voto: 1  
Spiega: In poche righe, come preferiresti interagire con il gioco o quale periferica useresti per aumentare il voto dato:  
 \_\_\_\_\_  
 \_\_\_\_\_  
 \_\_\_\_\_

9. Che domanda aggiungeresti al questionario per rendere più accurata la valutazione?  
 Domanda:  
 \_\_\_\_\_  
 Che risposta daresti alla domanda aggiuntiva?  
 \_\_\_\_\_  
 \_\_\_\_\_

Figura 4.1: Questionario

Una volta analizzate le problematiche e i punti di forza scaturiti dal feedback degli stackholder vengono generati i Backlog per l'iterazione successiva. La scelta di questa metodologia è stata fatta proprio per sperimentare la suddetta metodologia e cercare di ottenere un risultato qualitativamente buono in un tempo relativamente breve facendosi guidare da una procedura ben precisa.

## 4.2 Star Naitum

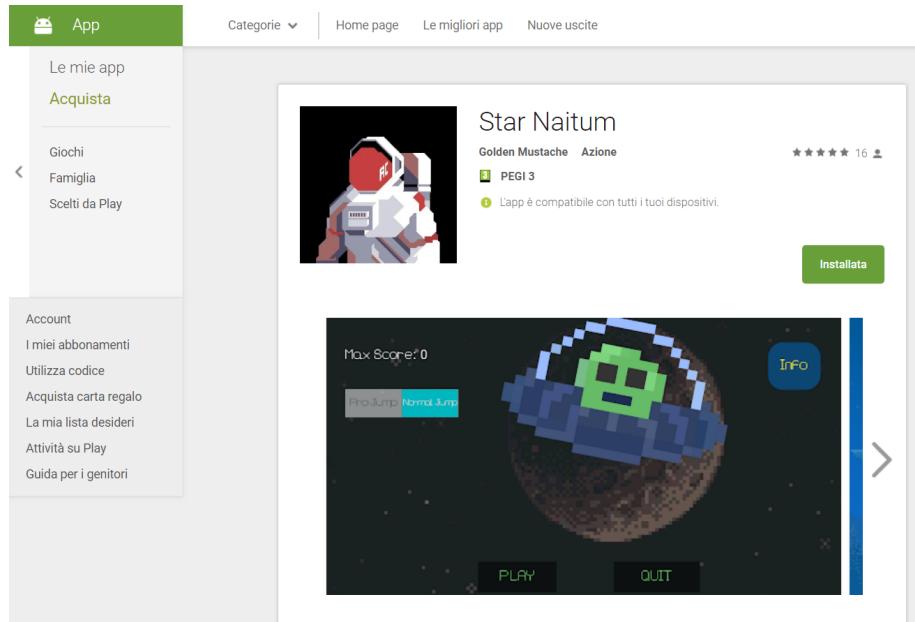


Figura 4.2: Pagina Play Store Star Naitum

### 4.2.1 Progettazione

In questa fase è fondamentale impostare il format del gioco attraverso il documento di Design, si vanno a descrivere le meccaniche principali che si vogliono ottenere andando sempre più nello specifico. “Star Naitum” è nato come un videogioco su piattaforma Windows e l’idea del gioco era quella di creare un ambiente in cui l’utente comandasse un astronauta sbarcato in una stella, denominata Naitum, per liberarla da nemici ostili. La storia ed il pretesto per giocare al titolo è molto semplice, in questa tipologia di giochi l’enfasi viene data al gameplay che deve essere proporzionato e bilanciato in modo da spronare il giocatore ad “imparare” al meglio i comandi, aumentare le sue abilità e quindi totalizzare un punteggio sempre maggiore. La base del progetto è stata la location, si è scelto uno scenario spaziale e quindi, grazie all’Assets Store, si sono prelevate delle immagini concordi all’idea del gioco.



Figura 4.3: Sprite Astronauta

L'immagine in figura contiene più strutture utilizzate all'interno del videogioco che vengono opportunamente sezionate a run-time per permettere il caricamento di 1 solo file in memoria, questo viene fatto perché spesso il collo di bottiglia in applicazioni videoludiche è proprio il caricamento in memoria e quindi in definitiva è molto più efficiente caricare un solo file in Ram e sezionarlo/modificarlo per ottenere gli sprite desiderati invece di caricare 1 file per ogni componente del gioco. Il passo successivo è stato il gameplay, la soluzione trovata è stata quella di inserire dei nemici che si materializzassero in modo casuale ed in punti strategici dello schermo incominciando a inseguire il giocatore e causandogli del danno una volta rilevata la collisione con esso. Dall'altro lato il giocatore può muoversi lungo l'ambiente rimanendo al di sopra di apposite piattaforme, inserite per rendere il grado di sfida più elevato e favorire il salto piuttosto che muoversi solo sull'asse delle ascisse. Il giocatore possiede un'arma con cui può affrontare i nemici infliggendogli danno a distanza. Una volta

sconfitto un nemico esso rilascerà una valuta (Gold) che permette al giocatore di aumentare determinate statistiche quali la velocità di movimento, il danno inflitto ai nemici e la vita massima dell’astronauta. Il giocatore deve scegliere che tipo di statistica aumentare in quanto non sarà possibile aumentare all’infinito tutte le statistiche ma, ad ogni miglioramento, il costo in Gold per un miglioramento successivo incrementerà in modo da porre l’utente a fare delle scelte e quindi ad adottare il suo stile di gioco (agile, difensivo, offensivo o semplicemente un mix dei precedenti). L’ultimo aspetto trattato nella fase di progettazione è stato il grado di crescita dei nemici, per questo la scelta è stata quella di aumentare le statistiche di danno e vita, alla fine di ogni Wave (orda), di una percentuale testata per ottenere il giusto bilanciamento. Successivamente è stato aumentato anche il numero di nemici che il GameManager sceglie di istanziare, aumentando anch’esso dopo ogni Wave. I nemici sono di due tipologie, quelli più veloci e agili ma meno dannosi, e quelli più lenti e prevedibili ma con un danno elevato. Questi sono generati in modo casuale con la stessa probabilità di Spawn ma si è deciso di inserire anche delle versioni potenziate (denominati GoldenVersion in quanto rilasceranno il doppio dei Gold in caso di abbattimento) per ogni tipologia di nemico, che hanno una probabilità di essere generati pari al 20%. Il gioco successivamente è stato esportato anche per sistemi Android con opportune modifiche alle meccaniche di input, per adattarle a dispositivi mobili, e anche alle percentuali di difficoltà, in quanto dai test effettuati e dal feedback raccolto, l’astronauta risultava molto più difficile da governare rispetto alla comodità dei tasti fisici del computer, per questo si è deciso di progettare al meglio i tasti virtuali presenti nella versione Android e si è diminuita la difficoltà (diminuendo la percentuale di potenziamento dei nemici dal 30% al 20%) risultando così paragonabile alla versione computer. Di seguito sono riportate le varie tecniche utilizzate per lo sviluppo, come il Parallaxing e il Tiling, e alcuni degli script più importanti per la gestione delle meccaniche del gioco.

### **4.2.2 Produzione e Script**

Gli script, e quindi tutto ciò che riguarda la logica del prodotto sviluppato, sono stati scritti in C# in quanto Unity utilizza tale linguaggio come linguaggio di riferimento e presente anche nella maggior parte della documentazione presente sul sito ufficiale. Gli script utilizzati in StarNaitum sono:

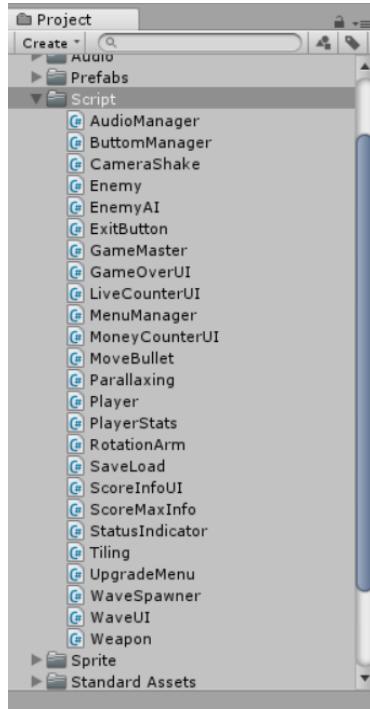


Figura 4.4: StarNaitum Script

tali script sono porzioni di codice assestanti che vanno a gestire un'entità ben precisa andandosi a “collegare” all’oggetto presente nella Scena View del videogioco, governano tutto ciò che può succedere e ciò che si deve calcolare all’interno del videogioco stesso, in seguito andremo ad analizzare solo alcuni di questi script andando a revisionare quelli più interessanti e complessi tra cui:

- PathFinding e EnemyAI
- AudioManager
- Camera2DFollow
- Parallaxing
- Tiling
- Platform2DController
- Weapon
- \_GM

tali script rappresentano la maggior parte della logica che è stata implementata nel videogioco e molti di questi si basano su tecniche di programmazione che spesso si utilizzano nel mondo videoludico quali PathFinding, Parallaxing, Tiling ecc.

*PathFinding:* L'intelligenza artificiale dei nemici è implementata da una serie di script. Il tutto serve a far muovere i nemici, secondo una certa logica ed in maniera realistica, da un punto A ad un punto B cercando di trovare il percorso ideale quando l'ambiente presenta ostacoli.

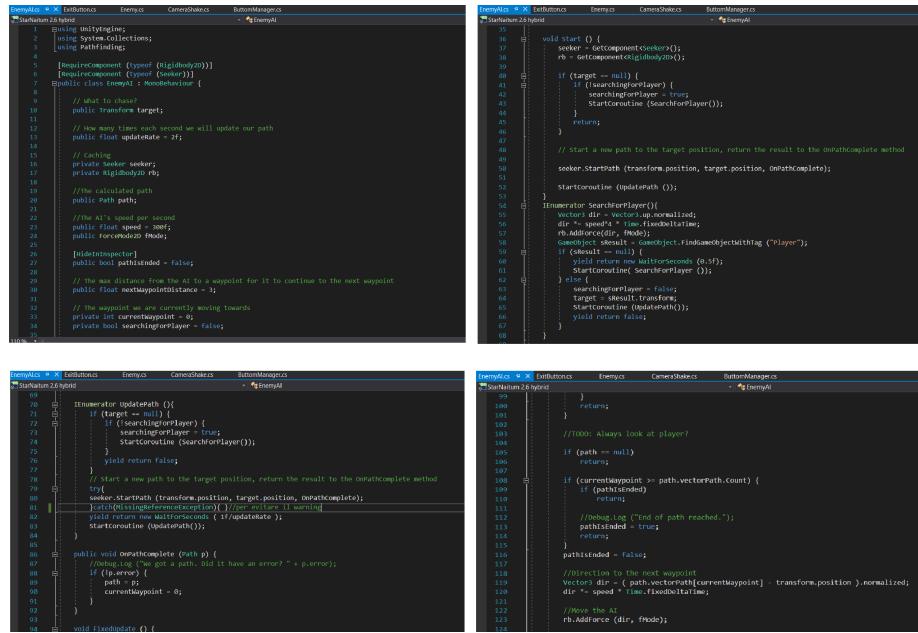


Figura 4.5: Script EnemyAI

Per ottenere questo risultato le tecniche di Pathfinding si basano su algoritmi di ricerca su strutture complesse quali grafi. In particolare si sceglie come rappresentare il mondo circostante nel discreto (insieme finito di punti/nodi/pixel) e si applicano degli algoritmi di ricerca tra cui A\*.

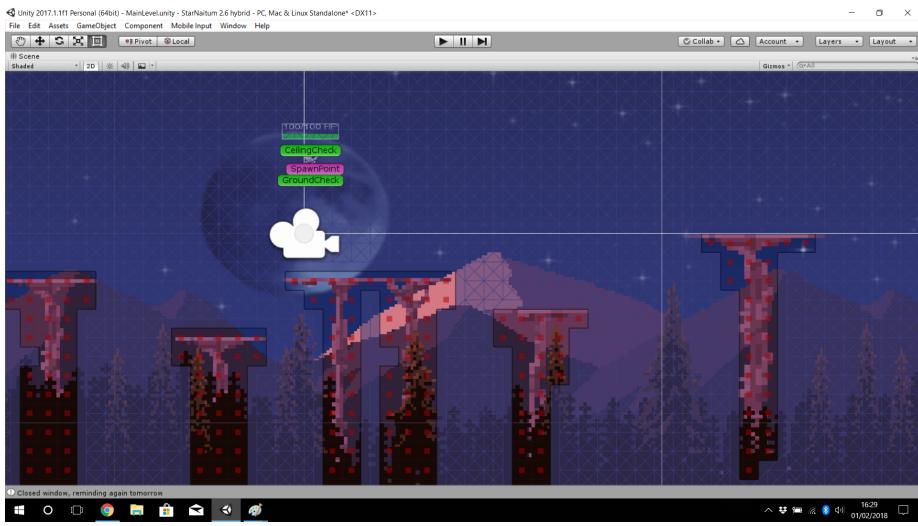


Figura 4.6: Griglia PathFinding

Nell’informatica, A\* (descritto nel 1968 da Peter Hart, Nils Nilsson e Bertram Raphael) è un algoritmo che è ampiamente utilizzato nel pathfinding e nelle tecniche di attraversamento dei grafi, il processo per tracciare un percorso diretto in modo efficiente tra più punti chiamati “nodi”. Gode di un uso diffuso grazie alle sue prestazioni e accuratezza. Tuttavia, nei sistemi di routing ed instradamento, è generalmente superato da algoritmi che possono pre-elaborare il grafico ottenendo così prestazioni migliori. A ogni iterazione del suo ciclo principale, A\* deve determinare quale dei suoi percorsi parziali espandere in uno o più percorsi più lunghi. Lo fa basandosi su una stima del costo (peso totale) per passare al nodo obiettivo. Nello specifico, A\* seleziona il percorso che minimizza la funzione:

$$f(n) = g(n) + h(n) \quad (4.1)$$

dove  $n$  è l’ultimo nodo del percorso,  $g(n)$  è il costo del percorso dal nodo iniziale a  $n$ , e  $h(n)$  è una stima euristica che stima il costo del percorso più “economico” dal nodo  $n$  al nodo obiettivo se non ci fossero ostacoli. Affinché l’algoritmo trovi il percorso effettivo più breve, la funzione euristica deve essere ammissibile, ovvero non sovrastima mai il costo effettivo per raggiungere il nodo obiettivo più vicino. Se l’euristica soddisfa la condizione:

$$h(x) = d(x, y) + h(y) \quad (4.2)$$

$\forall$  arco  $(x, y)$  del grafo (dove  $d$  denota la lunghezza dell’arco), allora  $h$  è chiamata monotona o consistente. In tal caso, A\* può essere implementato in modo più

efficiente: in parole povere, nessun nodo deve essere elaborato più di una volta e A\* equivale all'esecuzione dell'algoritmo di Dijkstra con il costo ridotto:

$$d'(x, y) = d(x, y) + h(y) - h(x) \quad (4.3)$$

Di seguito si descrive lo pseudocodice dell'algoritmo:

```

function A*(start,goal)
    closedset := the empty set           % The set of nodes already evaluated.
    openset := set containing the initial node % The set of tentative nodes to be evaluated.
    g_score[start] := 0                  % Distance from start along optimal path.
    came_from := the empty map          % The map of navigated nodes.
    h_score[start] := heuristic_estimate_of_distance(start, goal)
    f_score[start] := h_score[start]      % Estimated total distance from start to goal through y.
    while openset is not empty
        x := the node in openset having the lowest f_score[] value
        if x = goal
            return reconstruct_path(came_from,goal)
        remove x from openset
        add x to closedset
        foreach y in neighbor_nodes(x)
            if y in closedset
                continue
            tentative_g_score := g_score[x] + dist_between(x,y)

            if y not in openset
                add y to openset

                tentative_is_better := true
            elseif tentative_g_score < g_score[y]
                tentative_is_better := true
            else
                tentative_is_better := false
            if tentative_is_better = true
                came_from[y] := x
                g_score[y] := tentative_g_score
                h_score[y] := heuristic_estimate_of_distance(y, goal)
                f_score[y] := g_score[y] + h_score[y]
        return failure

function reconstruct_path(came_from,current_node)
    if came_from[current_node] is set
        p = reconstruct_path(came_from,came_from[current_node])
        return (p + current_node)
    else
        return the empty path

```

Figura 4.7: A\* PseudoCode

Successivamente è riportata un'illustrazione di A\* in cerca del percorso da un nodo iniziale a un nodo obiettivo in un problema di pianificazione del movimento di un robot. I nodi vuoti rappresentano i nodi nel set aperto, cioè quelli che rimangono da esplorare, e quelli pieni sono nel set chiuso, nodi già visitati. Il colore su ciascun nodo chiuso indica la distanza dall'obiettivo: verde vicino all'obiettivo, rosso più lontano. Si può prima vedere l'A\* muoversi in linea retta nella direzione dell'obiettivo, quindi quando colpisce l'ostacolo, esplorare percorsi alternativi attraverso i nodi dal set aperto.

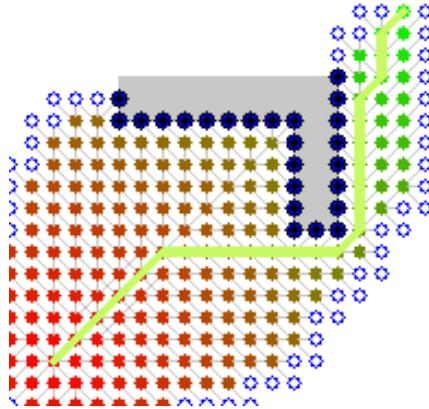


Figura 4.8: Esempio A\* Ottimale

Finché l'ammissibilità è garantita l'algoritmo restituisce un percorso ottimale, questo significa anche che A\* deve esaminare tutti i percorsi ugualmente importanti per trovare il percorso ottimale. Per calcolare percorsi approssimati più brevi, è possibile accelerare la ricerca a discapito dell'ottimalità attenuando il criterio di ammissibilità. Spesso vogliamo limitare questa relazione, in modo da poter garantire che il percorso della soluzione non sia peggiore di  $(1 + \varepsilon)$  volte il percorso della soluzione ottimale. Questa nuova garanzia è indicata come  $\varepsilon$ -ammissibile. Esistono numerosi algoritmi  $\varepsilon$ -ammissibili:

- Weighted A\*/Static Weighting
- Dynamic Weighting
- Sampled Dynamic Weighting
- AlphA\*
- $A_\varepsilon^*$
- $A_\varepsilon$

Di seguito troviamo una ricerca A\* che utilizza un'euristica di  $5.0(=\varepsilon)$  per ottenere un percorso sub-ottimale:

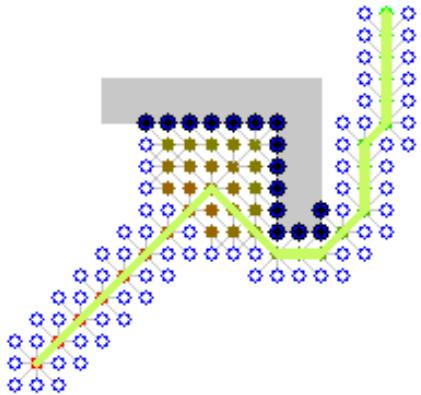


Figura 4.9: Esempio A\* sub-ottimale

La complessità temporale di A\* dipende dall'euristica. Nel caso peggiore di una ricerca nello spazio non limitato, il numero di nodi esplorati è esponenziale rispetto alla profondità della soluzione (il percorso più breve)  $d : O(b^d)$ , dove  $b$  è il fattore di branching. La complessità temporale è polinomiale quando lo spazio di ricerca è un albero, esiste un unico obiettivo e la funzione euristica  $h$  soddisfa la seguente condizione:

$$|h(x) - h^*(x)| = O(\log h^*(x)) \quad (4.4)$$

dove  $h^*$  è l'euristica ottimale, il costo esatto per passare da  $x$  all'obiettivo. In altre parole, l'errore di  $h$  non crescerà più velocemente del logaritmo della “perfetta euristica”  $h^*$  che restituisce la vera distanza da  $x$  all'obiettivo. Gli script utilizzati in Unity fanno uso dell'algoritmo A\*, per trovare il percorso “ottimale” (può non essere il percorso ottimale in base alle varianti utilizzate) ed è stato usato per permettere alle navicelle di inseguire l'utente e arrecargli danno. Il tutto funziona grazie al fatto che, dopo un numero prefissato di frame, l'obiettivo delle navicelle viene aggiornato e quindi lo spostamento dell'utente è calcolato e di conseguenza anche il nuovo percorso “ottimale”.

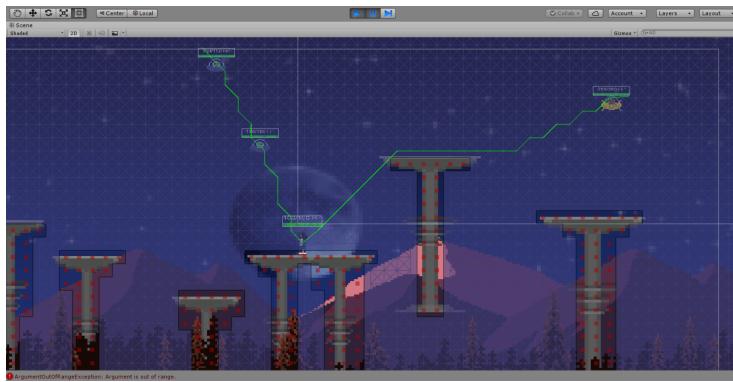


Figura 4.10: PathFinding a RunTime

*AudioManager:* Tale script è stato progettato in modo da persistere alle transizioni delle scene (tale processo distrugge tutti gli oggetti per favorire le performance e il caricamento della scena successiva) in quanto si vuole garantire la possibilità di istanziare un solo oggetto che carichi tutte le fonti sonore e che permetta di manipolare a piacimento e in maniera modulare tutto quello che riguarda l'audio, gli effetti sonori e la musica. In tale script è stato utilizzato il pattern singleton proprio per permettere la creazione di una sola istanza e, grazie ai metodi forniti dalla libreria UnityEngine, l'oggetto che rappresenta l'entità dell'AudioManager riesce a rimanere intatto al passaggio da una scena all'altra. La modularità dello script è ottenuta grazie alla creazione della classe Audio:

```
 AudioManager.cs  X
 StarNautum 2.6 hybrid
 5
 6
 7
 8
 9
 10
 11
 12
 13
 14
 15
 16
 17
 18
 19
 20
 21
 22
 23
 24
 25
 26
 27
 28
 29
 30
 31
 32
 33
 34
 35
 36
 37
 38
 39
 40
 41
 42
 43
 44
 45
 46
 47
 48
 49
 50
 51
 52
 53
 54
 55
 56
 57
 58
 59
 60
 61
 62
 63
 64
 65
 66
 67
 68
 69
 70
 71
 72
 73
 74
 75
 76
 77
 78
 79
 80
 81
 82
 83
 84
 85
 86
 87
 88
 89
 90
 91
 92
 93
 94
 95
 96
 97
 98
 99
 100
 101
 102
 103
 104
 105
 106
 107
 108
 109
 110
 111
 112
 113
 114
 115
 116
 117
 118
 119
 120
 121
 122
 123
 124
 125
 126
 127
 128
 129
 130
 131
 132
 133
 134
 135
 136
 137
 138
 139
 140
 141
 142
 143
 144
 145
 146
 147
 148
 149
 150
 151
 152
 153
 154
 155
 156
 157
 158
 159
 160
 161
 162
 163
 164
 165
 166
 167
 168
 169
 170
 171
 172
 173
 174
 175
 176
 177
 178
 179
 180
 181
 182
 183
 184
 185
 186
 187
 188
 189
 190
 191
 192
 193
 194
 195
 196
 197
 198
 199
 200
 201
 202
 203
 204
 205
 206
 207
 208
 209
 210
 211
 212
 213
 214
 215
 216
 217
 218
 219
 220
 221
 222
 223
 224
 225
 226
 227
 228
 229
 230
 231
 232
 233
 234
 235
 236
 237
 238
 239
 240
 241
 242
 243
 244
 245
 246
 247
 248
 249
 250
 251
 252
 253
 254
 255
 256
 257
 258
 259
 260
 261
 262
 263
 264
 265
 266
 267
 268
 269
 270
 271
 272
 273
 274
 275
 276
 277
 278
 279
 280
 281
 282
 283
 284
 285
 286
 287
 288
 289
 290
 291
 292
 293
 294
 295
 296
 297
 298
 299
 300
 301
 302
 303
 304
 305
 306
 307
 308
 309
 310
 311
 312
 313
 314
 315
 316
 317
 318
 319
 320
 321
 322
 323
 324
 325
 326
 327
 328
 329
 330
 331
 332
 333
 334
 335
 336
 337
 338
 339
 340
 341
 342
 343
 344
 345
 346
 347
 348
 349
 350
 351
 352
 353
 354
 355
 356
 357
 358
 359
 360
 361
 362
 363
 364
 365
 366
 367
 368
 369
 370
 371
 372
 373
 374
 375
 376
 377
 378
 379
 380
 381
 382
 383
 384
 385
 386
 387
 388
 389
 390
 391
 392
 393
 394
 395
 396
 397
 398
 399
 400
 401
 402
 403
 404
 405
 406
 407
 408
 409
 410
 411
 412
 413
 414
 415
 416
 417
 418
 419
 420
 421
 422
 423
 424
 425
 426
 427
 428
 429
 430
 431
 432
 433
 434
 435
 436
 437
 438
 439
 440
 441
 442
 443
 444
 445
 446
 447
 448
 449
 450
 451
 452
 453
 454
 455
 456
 457
 458
 459
 460
 461
 462
 463
 464
 465
 466
 467
 468
 469
 470
 471
 472
 473
 474
 475
 476
 477
 478
 479
 480
 481
 482
 483
 484
 485
 486
 487
 488
 489
 490
 491
 492
 493
 494
 495
 496
 497
 498
 499
 500
 501
 502
 503
 504
 505
 506
 507
 508
 509
 510
 511
 512
 513
 514
 515
 516
 517
 518
 519
 520
 521
 522
 523
 524
 525
 526
 527
 528
 529
 530
 531
 532
 533
 534
 535
 536
 537
 538
 539
 540
 541
 542
 543
 544
 545
 546
 547
 548
 549
 550
 551
 552
 553
 554
 555
 556
 557
 558
 559
 560
 561
 562
 563
 564
 565
 566
 567
 568
 569
 570
 571
 572
 573
 574
 575
 576
 577
 578
 579
 580
 581
 582
 583
 584
 585
 586
 587
 588
 589
 590
 591
 592
 593
 594
 595
 596
 597
 598
 599
 600
 601
 602
 603
 604
 605
 606
 607
 608
 609
 610
 611
 612
 613
 614
 615
 616
 617
 618
 619
 620
 621
 622
 623
 624
 625
 626
 627
 628
 629
 630
 631
 632
 633
 634
 635
 636
 637
 638
 639
 640
 641
 642
 643
 644
 645
 646
 647
 648
 649
 650
 651
 652
 653
 654
 655
 656
 657
 658
 659
 660
 661
 662
 663
 664
 665
 666
 667
 668
 669
 670
 671
 672
 673
 674
 675
 676
 677
 678
 679
 680
 681
 682
 683
 684
 685
 686
 687
 688
 689
 690
 691
 692
 693
 694
 695
 696
 697
 698
 699
 700
 701
 702
 703
 704
 705
 706
 707
 708
 709
 710
 711
 712
 713
 714
 715
 716
 717
 718
 719
 720
 721
 722
 723
 724
 725
 726
 727
 728
 729
 730
 731
 732
 733
 734
 735
 736
 737
 738
 739
 740
 741
 742
 743
 744
 745
 746
 747
 748
 749
 750
 751
 752
 753
 754
 755
 756
 757
 758
 759
 760
 761
 762
 763
 764
 765
 766
 767
 768
 769
 770
 771
 772
 773
 774
 775
 776
 777
 778
 779
 780
 781
 782
 783
 784
 785
 786
 787
 788
 789
 790
 791
 792
 793
 794
 795
 796
 797
 798
 799
 800
 801
 802
 803
 804
 805
 806
 807
 808
 809
 810
 811
 812
 813
 814
 815
 816
 817
 818
 819
 820
 821
 822
 823
 824
 825
 826
 827
 828
 829
 830
 831
 832
 833
 834
 835
 836
 837
 838
 839
 840
 841
 842
 843
 844
 845
 846
 847
 848
 849
 850
 851
 852
 853
 854
 855
 856
 857
 858
 859
 860
 861
 862
 863
 864
 865
 866
 867
 868
 869
 870
 871
 872
 873
 874
 875
 876
 877
 878
 879
 880
 881
 882
 883
 884
 885
 886
 887
 888
 889
 890
 891
 892
 893
 894
 895
 896
 897
 898
 899
 900
 901
 902
 903
 904
 905
 906
 907
 908
 909
 910
 911
 912
 913
 914
 915
 916
 917
 918
 919
 920
 921
 922
 923
 924
 925
 926
 927
 928
 929
 930
 931
 932
 933
 934
 935
 936
 937
 938
 939
 940
 941
 942
 943
 944
 945
 946
 947
 948
 949
 950
 951
 952
 953
 954
 955
 956
 957
 958
 959
 960
 961
 962
 963
 964
 965
 966
 967
 968
 969
 970
 971
 972
 973
 974
 975
 976
 977
 978
 979
 980
 981
 982
 983
 984
 985
 986
 987
 988
 989
 990
 991
 992
 993
 994
 995
 996
 997
 998
 999
 1000
 1001
 1002
 1003
 1004
 1005
 1006
 1007
 1008
 1009
 1010
 1011
 1012
 1013
 1014
 1015
 1016
 1017
 1018
 1019
 1020
 1021
 1022
 1023
 1024
 1025
 1026
 1027
 1028
 1029
 1030
 1031
 1032
 1033
 1034
 1035
 1036
 1037
 1038
 1039
 1040
 1041
 1042
 1043
 1044
 1045
 1046
 1047
 1048
 1049
 1050
 1051
 1052
 1053
 1054
 1055
 1056
 1057
 1058
 1059
 1060
 1061
 1062
 1063
 1064
 1065
 1066
 1067
 1068
 1069
 1070
 1071
 1072
 1073
 1074
 1075
 1076
 1077
 1078
 1079
 1080
 1081
 1082
 1083
 1084
 1085
 1086
 1087
 1088
 1089
 1090
 1091
 1092
 1093
 1094
 1095
 1096
 1097
 1098
 1099
 1099
 1100
 1101
 1102
 1103
 1104
 1105
 1106
 1107
 1108
 1109
 1110
 1111
 1112
 1113
 1114
 1115
 1116
 1117
 1118
 1119
 1119
 1120
 1121
 1122
 1123
 1124
 1125
 1126
 1127
 1128
 1129
 1129
 1130
 1131
 1132
 1133
 1134
 1135
 1136
 1137
 1138
 1139
 1139
 1140
 1141
 1142
 1143
 1144
 1145
 1146
 1147
 1148
 1149
 1149
 1150
 1151
 1152
 1153
 1154
 1155
 1156
 1157
 1158
 1159
 1159
 1160
 1161
 1162
 1163
 1164
 1165
 1166
 1167
 1168
 1169
 1169
 1170
 1171
 1172
 1173
 1174
 1175
 1176
 1177
 1178
 1179
 1179
 1180
 1181
 1182
 1183
 1184
 1185
 1186
 1187
 1188
 1189
 1189
 1190
 1191
 1192
 1193
 1194
 1195
 1196
 1197
 1198
 1199
 1199
 1200
 1201
 1202
 1203
 1204
 1205
 1206
 1207
 1208
 1209
 1209
 1210
 1211
 1212
 1213
 1214
 1215
 1216
 1217
 1218
 1219
 1219
 1220
 1221
 1222
 1223
 1224
 1225
 1226
 1227
 1228
 1229
 1229
 1230
 1231
 1232
 1233
 1234
 1235
 1236
 1237
 1238
 1238
 1239
 1240
 1241
 1242
 1243
 1244
 1245
 1246
 1247
 1248
 1248
 1249
 1250
 1251
 1252
 1253
 1254
 1255
 1256
 1257
 1258
 1258
 1259
 1260
 1261
 1262
 1263
 1264
 1265
 1266
 1267
 1268
 1268
 1269
 1270
 1271
 1272
 1273
 1274
 1275
 1276
 1277
 1278
 1278
 1279
 1280
 1281
 1282
 1283
 1284
 1285
 1286
 1287
 1287
 1288
 1289
 1290
 1291
 1292
 1293
 1294
 1295
 1296
 1297
 1298
 1299
 1299
 1300
 1301
 1302
 1303
 1304
 1305
 1306
 1307
 1308
 1309
 1309
 1310
 1311
 1312
 1313
 1314
 1315
 1316
 1317
 1318
 1318
 1319
 1320
 1321
 1322
 1323
 1324
 1325
 1326
 1327
 1328
 1328
 1329
 1330
 1331
 1332
 1333
 1334
 1335
 1336
 1337
 1338
 1338
 1339
 1340
 1341
 1342
 1343
 1344
 1345
 1346
 1347
 1348
 1348
 1349
 1350
 1351
 1352
 1353
 1354
 1355
 1356
 1357
 1358
 1358
 1359
 1360
 1361
 1362
 1363
 1364
 1365
 1366
 1367
 1368
 1368
 1369
 1370
 1371
 1372
 1373
 1374
 1375
 1376
 1377
 1378
 1378
 1379
 1380
 1381
 1382
 1383
 1384
 1385
 1386
 1387
 1388
 1388
 1389
 1390
 1391
 1392
 1393
 1394
 1395
 1396
 1397
 1398
 1398
 1399
 1400
 1401
 1402
 1403
 1404
 1405
 1406
 1407
 1408
 1408
 1409
 1410
 1411
 1412
 1413
 1414
 1415
 1416
 1417
 1418
 1418
 1419
 1420
 1421
 1422
 1423
 1424
 1425
 1426
 1427
 1428
 1428
 1429
 1430
 1431
 1432
 1433
 1434
 1435
 1436
 1437
 1438
 1438
 1439
 1440
 1441
 1442
 1443
 1444
 1445
 1446
 1447
 1448
 1448
 1449
 1450
 1451
 1452
 1453
 1454
 1455
 1456
 1457
 1458
 1458
 1459
 1460
 1461
 1462
 1463
 1464
 1465
 1466
 1467
 1468
 1468
 1469
 1470
 1471
 1472
 1473
 1474
 1475
 1476
 1477
 1478
 1478
 1479
 1480
 1481
 1482
 1483
 1484
 1485
 1486
 1487
 1488
 1488
 1489
 1490
 1491
 1492
 1493
 1494
 1495
 1496
 1497
 1498
 1498
 1499
 1500
 1501
 1502
 1503
 1504
 1505
 1506
 1507
 1508
 1508
 1509
 1510
 1511
 1512
 1513
 1514
 1515
 1516
 1517
 1518
 1518
 1519
 1520
 1521
 1522
 1523
 1524
 1525
 1526
 1527
 1528
 1528
 1529
 1530
 1531
 1532
 1533
 1534
 1535
 1536
 1537
 1538
 1538
 1539
 1540
 1541
 1542
 1543
 1544
 1545
 1546
 1547
 1548
 1548
 1549
 1550
 1551
 1552
 1553
 1554
 1555
 1556
 1557
 1558
 1558
 1559
 1560
 1561
 1562
 1563
 1564
 1565
 1566
 1567
 1568
 1568
 1569
 1570
 1571
 1572
 1573
 1574
 1575
 1576
 1577
 1578
 1578
 1579
 1580
 1581
 1582
 1583
 1584
 1585
 1586
 1587
 1588
 1588
 1589
 1590
 1591
 1592
 1593
 1594
 1595
 1596
 1597
 1598
 1598
 1599
 1600
 1601
 1602
 1603
 1604
 1605
 1606
 1607
 1608
 1608
 1609
 1610
 1611
 1612
 1613
 1614
 1615
 1616
 1617
 1618
 1618
 1619
 1620
 1621
 1622
 1623
 1624
 1625
 1626
 1627
 1628
 1628
 1629
 1630
 1631
 1632
 1633
 1634
 1635
 1636
 1637
 1638
 1638
 1639
 1640
 1641
 1642
 1643
 1644
 1645
 1646
 1647
 1648
 1648
 1649
 1650
 1651
 1652
 1653
 1654
 1655
 1656
 1657
 1658
 1658
 1659
 1660
 1661
 1662
 1663
 1664
 1665
 1666
 1667
 1668
 1668
 1669
 1670
 1671
 1672
 1673
 1674
 1675
 1676
 1677
 1678
 1678
 1679
 1680
 1681
 1682
 1683
 1684
 1685
 1686
 1687
 1688
 1688
 1689
 1690
 1691
 1692
 1693
 1694
 1695
 1696
 1697
 1698
 1698
 1699
 1700
 1701
 1702
 1703
 1704
 1705
 1706
 1707
 1708
 1708
 1709
 1710
 1711
 1712
 1713
 1714
 1715
 1716
 1717
 1718
 1718
 1719
 1720
 1721
 1722
 1723
 1724
 1725
 1726
 1727
 1728
 1728
 1729
 1730
 1731
 1732
 1733
 1734
 1735
 1736
 1737
 1738
 1738
 1739
 1740
 1741
 1742
 1743
 1744
 1745
 1746
 1747
 1748
 1748
 1749
 1750
 1751
 1752
 1753
 1754
 1755
 1756
 1757
 1758
 1759
 1759
 1760
 1761
 1762
 1763
 1764
 1765
 1766
 1767
 1768
 1768
 1769
 1770
 1771
 1772
 1773
 1774
 1775
 1776
 1777
 1778
 1779
 1779
 1780
 1781
 1782
 1783
 1784
 1785
 1786
 1787
 1788
 1788
 1789
 1790
 1791
 1792
 1793
 1794
 1795
 1796
 1797
 1798
 1798
 1799
 1800
 1801
 1802
 1803
 1804
 1805
 1806
 1807
 1808
 1808
 1809
 1810
 1811
 1812
 1813
 1814
 1815
 1816
 1817
 1818
 1818
 1819
 1820
 1821
 1822
 1823
 1824
 1825
 1826
 1827
 1828
 1828
 1829
 1830
 1831
 1832
 1833
 1834
 1835
 1836
 1837
 1838
 1838
 1839
 1840
 1841
 1842
 1843
 1844
 1845
 1846
 1847
 1848
 1848
 1849
 1850
 1851
 1852
 1853
 1854
 1855
 1856
 1857
 1858
 1859
 1859
 1860
 1861
 1862
 1863
 1864
 1865
 1866
 1867
 1868
 1868
 1869
 1870
 1871
 1872
 1873
 1874
 1875
 1876
 1877
 1878
 1878
 1879
 1880
 1881
 1882
 1883
 1884
 1885
 1886
 1887
 1888
 1888
 1889
 1890
 1891
 1892
 
```

che contiene una variabile AudioSource ed esporta i metodi Play() e Stop() per interagire con tale risorsa, lo script contiene al suo interno un vettore di oggetti Sound e definisce metodi per il caricamento e l'esecuzione dei metodi della classe Audio demandando di fatto l'esecuzione alla classe stessa. Ciò permette di caricare un qualsiasi Assets audio e inserirlo in un vettore, direttamente dall'Inspector di Unity, in modo rapido ed efficiente e dall'Inspector stesso si possono regolare tutte le proprietà che fornisce la classe AudioSource come la possibilità di effettuare il loop, il volume ed alcuni effetti sonori quali il riverbero e il fattore “pitch”.

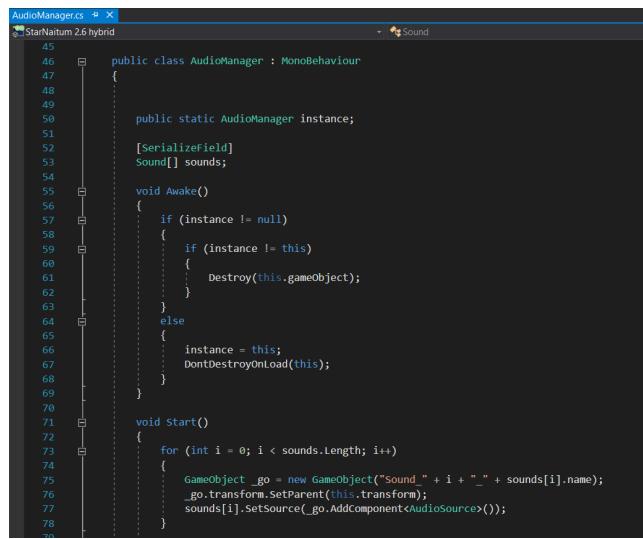
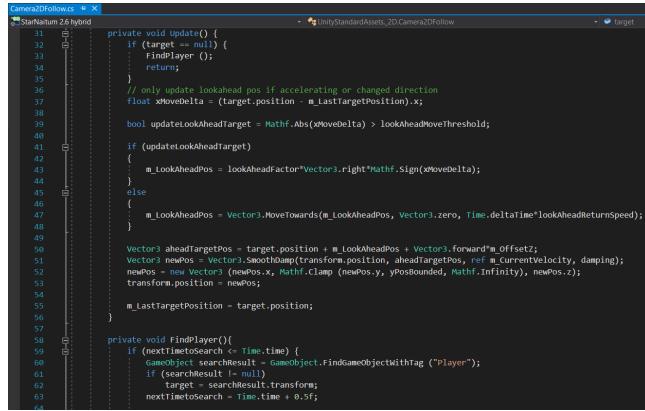


Figura 4.12: Script AudioManager

*Camera2DFollow:* È uno script che fornisce direttamente la libreria di Unity grazie allo UnityStandardAssets e permette il movimento dell'oggetto mainCamera in modo da seguire, in modo più o meno forzato, un target che nel caso del progetto sarà l'astronauta. Questa è una tecnica molto utilizzata nei giochi 2D ed è la base del movimento e della visuale di un gioco platform. L'utilità degli script forniti dallo StandardAssets è che sono pronti all'uso ma soprattutto sono completamente modificabili e quindi, come nel nostro caso, si possono andare a perfezionare alcuni aspetti dello script rendendolo più utile alla causa di quanto non sia già. Di seguito si riporta il semplice script che calcola la posizione del target e di conseguenza applica un movimento alla mainCamera che inquadrerà il target secondo un parametro di “smooth” che rende il movimento più o meno repentino.



```

    Camera2DFollow.cs
    ...
    private void update() {
        if (target == null) {
            FindPlayer();
            return;
        }
        // only update lookahead pos if accelerating or changed direction
        float xMoveDelta = (target.position - m_lastTargetPosition).x;
        bool updateLookAheadTarget = Mathf.Abs(xMoveDelta) > lookAheadMoveThreshold;
        if (updateLookAheadTarget) {
            if (m_lookAheadPos = lookAheadFactor*Vector3.right*Mathf.Sign(xMoveDelta));
            else {
                m_lookAheadPos = Vector3.MoveTowards(m_lookAheadPos, Vector3.zero, Time.deltaTime*lookAheadReturnSpeed);
            }
            Vector3 aheadTargetPos = target.position + m_lookAheadPos - Vector3.forward*m_offsetZ;
            Vector3 newPos = Vector3.SmoothDamp(transform.position, aheadTargetPos, ref m_CurrentVelocity, damping);
            newPos = new Vector3(newPos.x, Mathf.Clamp(newPos.y, yBounded, yUnbounded), newPos.z);
            transform.position = newPos;
            m_lastTargetPosition = target.position;
        }
        private void FindPlayer(){
            if (nextTimeToSearch < Time.time) {
                GameObject searchResult = GameObject.FindGameObjectWithTag ("Player");
                if (searchResult != null)
                    target = searchResult.transform;
                nextTimeToSearch = Time.time + 0.5f;
            }
        }
    }
}

```

Figura 4.13: Script Camera2Dfollow

*Parallaxing:* Anche questo script rientra tra gli algoritmi di base per lo sviluppo di un videogioco in 2D. Tale script permette di muovere lo sfondo in maniera coerente con la posizione del target in modo da ottenere una sensazione di profondità più marcata. Nei videogiochi 2D un grosso problema è la percezione della profondità.

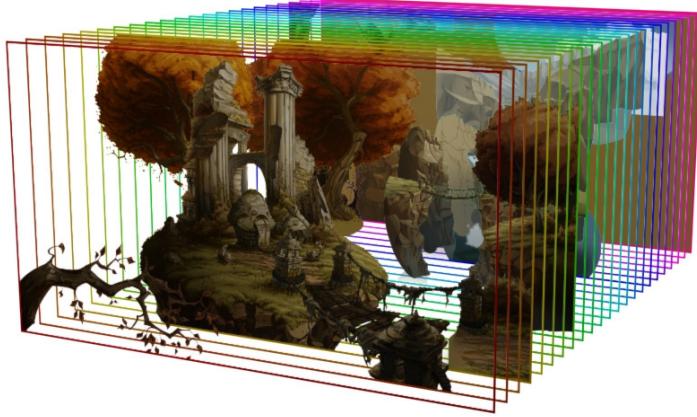
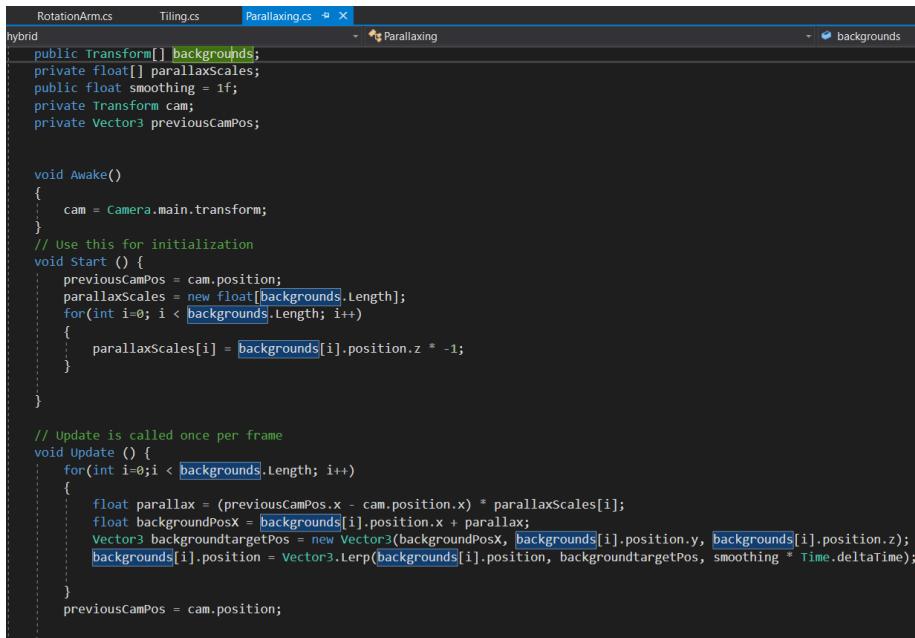


Figura 4.14: Layer Parallaxing

Questa sensazione viene restituita dalla tecnica Parallaxing che consiste nel movimentare lo sfondo in maniera più o meno veloce consentendo di creare dei layer (livelli) che si muoveranno in modo da restituire tale sensazione. Lo script calcola la distanza che è presente tra l'oggetto camera e gli oggetti che caratterizzano lo sfondo, in maniera direttamente proporzionale alla distanza tra di essi e applica una forza (in Unity i movimenti dinamici si ottengono aggiungendo

dei vettori forza alla componente Physics2D dell’oggetto) che muove le strutture tipicamente utilizzate come background. Il risultato della logica di tale script è quello di muovere più velocemente gli oggetti lontani e meno velocemente quelli vicini sempre in maniera coerente al movimento della camera. Questo risultato non è molto intuitivo e sembra in contrapposizione con il risultato che vogliamo ottenere cioè quello che, se un oggetto (come una montagna) è lontana rispetto ad un oggetto più vicino (come una piattaforma), allora la montagna deve muoversi di meno in prospettiva al personaggio proprio in quanto la distanza tra i due risulta maggiore. Questo è giusto ma per far in modo che la montagna sembri quasi ferma rispetto al movimento dell’astronauta, l’oggetto background deve traslare quasi alla stessa velocità dell’astronauta e quindi dell’inquadratura. Risulta logico quindi che il movimento degli oggetti più lontani deve essere maggiore proprio per rimanere in posizione rispetto al resto della scena. Di seguito è riportato il codice dello script:



```

RotationArm.cs   Tiling.cs   Parallaxing.cs  ✘
hybrid
public Transform[] backgrounds;
private float[] parallaxscales;
public float smoothing = 1f;
private Transform cam;
private Vector3 previousCamPos;

void Awake()
{
    cam = Camera.main.transform;
}
// Use this for initialization
void Start () {
    previousCamPos = cam.position;
    parallaxscales = new float[backgrounds.Length];
    for(int i=0; i < backgrounds.Length; i++)
    {
        parallaxscales[i] = backgrounds[i].position.z * -1;
    }
}

// Update is called once per frame
void Update () {
    for(int i=0;i < backgrounds.Length; i++)
    {
        float parallax = (previousCamPos.x - cam.position.x) * parallaxscales[i];
        float backgroundPosX = backgrounds[i].position.x + parallax;
        Vector3 backgroundtargetPos = new Vector3(backgroundPosX, backgrounds[i].position.y, backgrounds[i].position.z);
        backgrounds[i].position = Vector3.Lerp(backgrounds[i].position, backgroundtargetPos, smoothing * Time.deltaTime);
    }
    previousCamPos = cam.position;
}

```

Figura 4.15: Script Parallaxing

*Tiling:* anch’esso fa parte delle tecniche di base 2D, consiste nel ripetere l’immagine desiderata in maniera intelligente in modo da ottimizzare i caricamenti e salvare risorse.

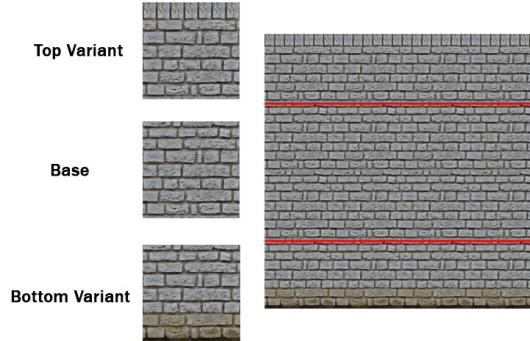


Figura 4.16: Tiling

Questo script calcola la posizione della mainCamera rispetto alla lunghezza dell'immagine stessa, una volta che la visuale sulla scena si trova ad una certa distanza dalla fine dello Sprite allora lo script si incarica di clonare e posizionare in maniera appropriata l'immagine in modo da carregarla solo quando necessario. Questa tecnica, se usata bene, può garantire un'ampiezza della scena quasi infinita a patto di utilizzare in maniera modulare porzioni di sprite in modo da creare ambientazioni credibili e non monotone.

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Tiling : MonoBehaviour
{
    // Update is called once per frame
    void Update()
    {
        // does it still need buddy? If not do nothing
        if (!hasLeftBuddy || !hasRightBuddy) {
            // calculate the position where the camera can see the edge of the sprite (element)
            float edgeVisiblePositionRight = (myTransform.position.x + spriteWidth / 2) - camHorizontalExtend;
            float edgeVisiblePositionLeft = (myTransform.position.x - spriteWidth / 2) + camHorizontalExtend;

            // checking if we can see the edge of the element and then calling MakeBuddy. If we can
            if (cam.transform.position.x >= edgeVisiblePositionRight - offsetX && hasRightBuddy == false)
            {
                MakeBuddy();
                hasRightBuddy = true;
            }
            else if (cam.transform.position.x <= edgeVisiblePositionLeft + offsetx && hasLeftBuddy == false)
            {
                MakeBuddy();
                hasLeftBuddy = true;
            }
        }
    }

    // a function that creates a buddy on the side required
    void MakeBuddy (int rightOrLeft)
    {
        // calculating the new position for our new buddy
        Vector3 newPosition = new Vector3 (myTransform.position.x + spriteWidth * rightOrLeft, myTransform.position.y, myTransform.position.z);
        // instantiating our new body and storing him in a variable
        Transform newBody = Instantiate (myTransform, newPosition, myTransform.rotation) as Transform;

        // if not tilable let's reverse the x size of our object to get rid of ugly seams
        if (!reversibleScale) {
            newBody.localScale = new Vector3 (newBuddy.localScale.x*-1, newBuddy.localScale.y, newBuddy.localScale.z);
        }
    }
}

public class Tiling2 : MonoBehaviour
{
    // Update is called once per frame
    void Update()
    {
        // does it still need buddy? If not do nothing
        if (!hasLeftBuddy || !hasRightBuddy) {
            // calculate the position where the camera can see the edge of the sprite (element)
            float edgeVisiblePositionRight = (myTransform.position.x + spriteWidth / 2) - camHorizontalExtend;
            float edgeVisiblePositionLeft = (myTransform.position.x - spriteWidth / 2) + camHorizontalExtend;

            // checking if we can see the edge of the element and then calling MakeBuddy. If we can
            if (cam.transform.position.x >= edgeVisiblePositionRight - offsetX && hasRightBuddy == false)
            {
                MakeBuddy();
                hasRightBuddy = true;
            }
            else if (cam.transform.position.x <= edgeVisiblePositionLeft + offsetx && hasLeftBuddy == false)
            {
                MakeBuddy();
                hasLeftBuddy = true;
            }
        }
    }

    // a function that creates a buddy on the side required
    void MakeBuddy (int rightOrLeft)
    {
        // calculating the new position for our new buddy
        Vector3 newPosition = new Vector3 (myTransform.position.x + spriteWidth * rightOrLeft, myTransform.position.y, myTransform.position.z);
        // instantiating our new body and storing him in a variable
        Transform newBody = Instantiate (myTransform, newPosition, myTransform.rotation) as Transform;

        // if not tilable let's reverse the x size of our object to get rid of ugly seams
        if (!reversibleScale) {
            newBody.localScale = new Vector3 (newBuddy.localScale.x*-1, newBuddy.localScale.y, newBuddy.localScale.z);
        }

        newBody.parent = myTransform.parent;
        if (rightOrLeft == 0) {
            newBody.GetComponent<Tiling>().hasLeftBuddy = true;
        }
        else {
            newBody.GetComponent<Tiling>().hasRightBuddy = true;
        }
    }
}

```

Figura 4.17: Script Tiling

*Platform2DController:* Insieme di script forniti da UnityStandardAssets che permettono la cattura di comandi da input e l'esecuzione delle movenze basilari di un qualsiasi personaggio 2D cioè il movimento lungo l'asse *XY* (compreso la meccanica di salto), l'orientamento dello sprite in maniera coerente al movimento e la gestione delle animazioni relative a tali movimenti: Move, Run, Jump, Crunch. Questo script è stato opportunamente modificato per abilitare i suoni durante le animazioni e per avviare le animazioni stesse che permettono agli sprite del personaggio di interscambiarsi, in modo corretto, per ottenere l'effetto della camminata, del salto e della mira. Tale script è stato ulteriormente manipolato per ricevere i comandi non solo da tastiera, come nelle prime versioni del gioco, ma anche da pad virtuali che sono presenti nella versione Android e permettono l'interazione con il software. Unity fa uso di input virtuali che vengono pre-configurati in modo da fornire allo sviluppatore comandi precisi come Jump, Fire1, Fire2 e Run che vengono mappati, in fase di configurazione di Unity, su dispositivi di input (generalmente tastiere). Questo garantisce un'astrazione che aiuta molto lo sviluppatore in quanto il codice del gioco può basarsi su comandi generali per tutte le piattaforme come quelli precedentemente elencati mentre l'implementazione e il mapping viene fatto in un secondo momento e non intacca le funzionalità del prodotto.

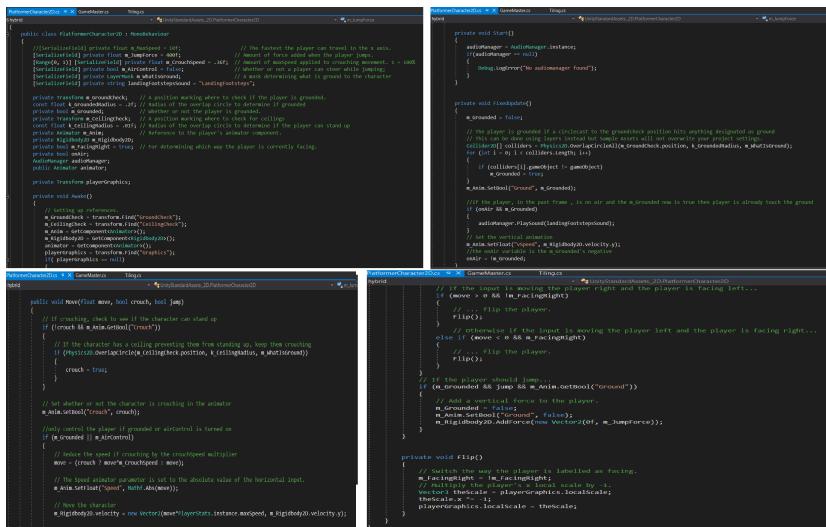


Figura 4.18: Script Platform2DController

*Weapon:* Lo script più complesso, nonché quello che ha impiegato il maggior tempo di sviluppo, è stato sicuramente Weapon. Questo script fornisce le

funzionalità delle meccaniche di “shooting” e governa il sistema di puntamento dell’utente, il sistema di sparo e di fire-rate, la creazioni di un oggetto RayCaster che permette la simulazione del proiettile e il calcolo delle collisioni che avvengono tra l’oggetto RayCaster e il mondo di gioco. In particolare ogni volta che il “raggio” collide con un oggetto viene verificata la variabile collider di RaycastHit2D che verifica quale sia l’oggetto colliso e gestisce il calcolo o meno dei danni. Nelle ultime versioni del gioco si è usato il Platform Dependent Compilation fornito da Unity che permette di scrivere codice compilabile secondo la piattaforma scelta. Questo ha permesso un notevole speedUp nello sviluppo del progetto in quanto il tutto veniva sviluppato per la versione Android e poi le modifiche (quelle definitive e testate) venivano trasferite anche sulla versione PC. Questa funzionalità permette di avere una sola build del progetto su cui lavorare e quindi un risparmio notevole di tempo, che è stato impiegato nella ricerca di ulteriori funzionalità e nel play-testing. Di seguito è riportato lo script che evidenzia l’utilizzo del PlatformDependentCompilation:

```

Weapon.cs - Win32
StarName:3.0
1 //using UnityEngine;
2 //using System.Collections;
3
4 [public class Weapon : MonoBehaviour {
5
6     public float fireRate = 0;
7     public LayerMask raycastLayer;
8     public GameObject bulletFlashPrefab;
9     public Transform muzzleFlashPrefab;
10    public Transform hitPrefab;
11    public float camshakeIntensity = 10;
12    public float camshakeDuration = 0.05f;
13    public float camshakeLength = 0.1f;
14
15    float timeToSpawnEffect = 0f;
16    float timeOfFire = 0;
17    Transform firePoint;
18    Camershake camshake;
19    RightJoyStick rj;
20
21    void Awake () {
22        firePoint = transform.Find("FirePoint");
23        if (firePoint == null) {
24            Debug.LogError("No FirePoint? WHAT!!!");
25        }
26    }
27
28    AudioManager audioManager;
29
30    private void Start()
31    {
32        camshake = GameMaster.gm.GetComponent<CamerShake>();
33    }
34
35    #if UNITY_ANDROID
36
37    void update()
38    {
39        if (fireRate == 0)
40        {
41            if (Input.GetButtonDown("Fire1"))
42            {
43                Shoot();
44            }
45            else
46            {
47
48                if (rj.getTouched() && Time.time > timeOfFire)
49                {
50                    timeOfFire = Time.time + 1 / fireRate;
51                    Shoot();
52                }
53            }
54        }
55
56        if (Time.time > timeToSpawnEffect) // mobile version
57        {
58            (timeOfFire = Time.time + 1 / fireRate);
59            Shoot();
60        }
61    }
62
63    #endif
64
65 }

```

```

Weapon.cs - Win
StarName:3.0
1 private void Start()
2 {
3     camshake = GameMaster.gm.GetComponent<CamerShake>();
4
5     #if UNITY_ANDROID
6
7         fireRate = 5;
8         GameObject searchResult = GameObject.FindGameObjectWithTag("Input");
9         rj = searchResult.GetComponent<RightJoyStick>();
10        if (rj == null)
11            Debug.LogError("No RightJoyStick found");
12
13        #elif UNITY_STANDALONE_WIN
14            fireRate = 0;
15
16        #endif
17
18        if (camshake == null)
19        {
20            Debug.LogError("No Camershake script found on GM object.");
21
22            audioManager = AudioManager.instance;
23            if (audioManager == null)
24            {
25                Debug.LogError("No AudioManager found!!!");
26            }
27        }
28
29        void update()
30        {
31            if (fireRate == 0)
32            {
33                if (Input.GetButtonDown("Fire1"))
34                {
35
36                    if ((Time.time > timeToSpawnEffect)
37
38                        {
39                            Vector2 mousePosition = new Vector2(R.input.mousePosition.x, R.input.mousePosition.y);
40                            Vector2 firePointPosition = new Vector2 (firePoint.position.x, firePoint.position.y);
41                            RaycastHit2D hit = Physics2D.Raycast (firePointPosition, mousePosition, 10000f, whatHit);
42                            Debug.Log(hit);
43                            if (hit.collider != null)
44                            {
45                                if (enemy != null)
46                                {
47                                    enemy.Damage ((int)playerStats.instance.damage);
48                                }
49                            }
50                        }
51
52                        if ((Time.time > timeToSpawnEffect)
53
54                            {
55                                Vector3 hitPos;
56                                Vector3 hitNormal;
57                                if (hit.collider != null)
58                                {
59                                    hitPos = (mousePosition * 10000);
60                                    hitNormal = new Vector3(0,0,0);
61                                }
62                                else
63                                {
64                                    hitPos = hit.point;
65                                    hitNormal = hit.normal;
66                                }
67
68                                Effect(hitPos, hitNormal);
69                                timeToSpawnEffect = Time.time + 1 / effectsSpawnRate;
70                            }
71
72                        #endif
73                    }
74                }
75            }
76        }
77
78        #endif
79    }
80
81 }

```

```

125     }
126 
127     #elif UNITY_STANDALONE_WIN
128     {
129         Vector3 mousePosition = new Vector3 (Camera.main.ScreenToWorldPoint (input.mousePosition).x, Camera.main.ScreenToWorldPoint (input.mousePosition).y);
130         Vector2 firePointPosition = new Vector2 (firePoint.position.x, firePoint.position.y);
131         RaycastHit2D hit = Physics2D.Raycast (firePointPosition, mousePosition - firePointPosition, 10000f, whatToHit);
132         Debug.DrawRay(firePointPosition, (mousePosition - firePointPosition)*100, color.green, 5f);
133         if (hit.collider != null)
134             {
135                 Enemy enemy = hit.collider.GetComponent<Enemy> ();
136                 if (enemy != null)
137                     {
138                         enemy.damage ((int)playerstats.instance.damage);
139                     }
140             }
141         if (Time.time > timeToSpawnEffect)
142         {
143             Vector3 hitPos;
144             Vector3 hitNormal;
145             if (hit.collider == null)
146                 hitPos = (mousePosition - firePointPosition) * 10000f;
147                 hitNormal = new Vector3(9999, 9999, 9999);
148             else
149                 {
150                     hitPos = hit.point;
151                     hitNormal = hit.normal;
152                 }
153             Effect(hitPos, hitNormal);
154             timeToSpawnEffect = Time.time + 1 / effectSpawRate;
155         }
156     }
157 
158 #endif
159 }
160 
161 void Effect(Vector3 hitPos ,Vector3 hitNormal){
162     Transform trail = Instantiate(BulletTrailPrefab, firePoint.position, firePoint.rotation);
163     LineRenderer lr = trail.GetComponent<LineRenderer>();
164     if(lr != null)
165     {
166         lr.SetPosition(0, firePoint.position);
167         lr.SetPosition(1,hitPos);
168     }
169     Destroy(trail.gameObject, 0.04f);
170     if(hitNormal != new Vector3(9999, 9999, 9999))
171     {
172         Transform hitcone = (Transform)Instantiate(HitPrefabs, hitPos, Quaternion.FromToRotation(Vector3.right, hitNormal));
173         Destroy(hitcone.gameObject, 1f);
174     }
175     Transform clone = (Transform)Instantiate (MuzzleFlashPrefab, firePoint.position, firePoint.rotation);
176     if (clone == null)
177     {
178         Debug.LogError ("failed to instantiste MuzzleflashPrefab");
179     }
180     clone.parent = firePoint;
181     float size = Random.Range (0.6f, 0.9f);
182     clone.localScale = new Vector3 (size, size, 0);
183     Destroy (clone.gameObject, 0.62f);
184     CamShake.Shake(CamShakeAmt, camshakeLength);
185     AudioManager.PlaySound("defaultShoot");
186 }
187 }
188 
```

Figura 4.19: Script Weapon

*\_GM:* Nel mondo videoludico è buona norma avere un oggetto Game Master a cui si delega la responsabilità di gestire l'intero lifecycle del gioco, proprio per questo è stato sviluppato l'oggetto *\_GM*. Esso contiene lo script nonché cuore del gioco, che controlla in generale tutte le meccaniche soprattutto quelle che riguardano la gestione della vita, lo spawn dei nemici, il menu Upgrade e il GameOver. Lo script in se è molto facile ma proprio per questo molto efficace in quanto si evitano calcoli che non sono necessari in ogni frame, in modo da garantire un frameRate costante e quindi un gioco fluido e piacevole. Questa semplicità è ottenuta grazie ai metodi forniti dalla libreria di UnityEngine e dal componente Physics2D che esporta metodi per la gestione di collisioni ed eventi che vengono invocati da Unity sotto determinate condizioni che evitano di fatto il BusyWaiting, cosa molto rischiosa in questo tipo di applicazioni software. Lo script esporta metodi che si differenziano dalla signature e vengono richiamati dalle entità in gioco, quali il player e i nemici, e provvedono ad eliminare tali

oggetti in modo da liberare risorse di calcolo.

Figura 4.20: Script Game Manager

### 4.2.3 Play-Testing

In questa fase, a prodotto ultimato, c'è stata una serie intensa di test che hanno evidenziato alcuni bug e problematiche: i layer del background in determinate occasioni non si collocavano bene nella scena e si muovevano in modo molto evidente; questo problema era creato dallo script Parallaxing che non riusciva a prendere in maniera corretta le proporzioni dello sprite e aggiungeva una forza troppo elevata andando a causare il bug. Il tutto è stato risolto andando a modificare lo script e aggiungendo il calcolo del fattore di "scale" che non era previsto nella versione precedente. Il secondo bug si verificava quando due

nemici collidevano con l’astronauta andando a richiamare due volte il metodo per il calcolo dei danni nello stesso frame, nel caso in cui il danno fosse stato superiore alla vita il \_GM avrebbe dovuto richiamare il metodo Respawn che veniva richiamato due volte facendo apparire due entità astronauta; questo è stato risolto aggiungendo un controllo allo script che gestisce il respawn così da permettere una sola invocazione del metodo per frame. L’ultimo bug è quello che causava un salto troppo elevato rispetto a quello desiderato ed è stato risolto andando a modificare lo script di movimento fornito da unity Platform-Controller2D dove è stato inserito un controllo nel metodo Jump() che permette l’invocazione atomica del metodo. Una buona parte dello sviluppo è stata dedicata all’adeguamento del gioco per sistemi mobili, il problema più grande è stato quello di riprodurre i controlli in maniera virtuale sullo schermo degli smartphone in modo da rendere l’esperienza videoludica paragonabile a quella su PC. Il tutto è stato fatto attraversando una fase di porting del gioco, da piattaforma PC a smartphone, e quindi si è preso il progetto StandAlone\_Pc e si sono andati a modificare gli script che governavano i comandi andando ad inserire oggetti quali pad virtuali: uno per consentire il movimento del personaggio e uno per consentire la mira e lo sparo. Una meccanica che ha semplificato molto questa fase è stata la caratteristica presente in Unity denominata “Platform Dependent Compilation”, descritta nei capitoli precedenti, che ha permesso di lavorare su una sola Build a patto di inserire degli statement decisionali, formati da #if #elif #endif, che permettono di compilare porzioni di codice in base alla piattaforma scelta dell’apposito pannello di sviluppo.

<b>UNITY_EDITOR</b>	#define directive for calling Unity Editor scripts from your game code.
<b>UNITY_EDITOR_WIN</b>	#define directive for Editor code on Windows.
<b>UNITY_EDITOR OSX</b>	#define directive for Editor code on Mac OS X.
<b>UNITY_STANDALONE OSX</b>	#define directive for compiling/executing code specifically for Mac OS X (including Universal, PPC and Intel architectures).
<b>UNITY_STANDALONE WIN</b>	#define directive for compiling/executing code specifically for Windows standalone applications.
<b>UNITY_STANDALONE LINUX</b>	#define directive for compiling/executing code specifically for Linux standalone applications.
<b>UNITY_STANDALONE</b>	#define directive for compiling/executing code for any standalone platform (Mac OS X, Windows or Linux).
<b>UNITY_WII</b>	#define directive for compiling/executing code for the Wii console.
<b>UNITY_IOS</b>	#define directive for compiling/executing code for the iOS platform.
<b>UNITY_IPHONE</b>	Deprecated. Use <b>UNITY_IOS</b> instead.
<b>UNITY_ANDROID</b>	#define directive for the Android platform.
<b>UNITY_PS4</b>	#define directive for running PlayStation 4 code.
<b>UNITY_XBOXONE</b>	#define directive for executing Xbox One code.
<b>UNITY_TIZEN</b>	#define directive for the Tizen platform.
<b>UNITY_TVOS</b>	#define directive for the Apple TV platform.
<b>UNITY_WSA</b>	#define directive for Universal Windows Platform. Additionally, <b>NETFX_CORE</b> is defined when compiling C# files against .NET Core and using .NET scripting backend.
<b>UNITY_WSA_10_0</b>	#define directive for Universal Windows Platform. Additionally <b>WINDOWS_UWP</b> is defined when compiling C# files against .NET Core.

Figura 4.21: PlatformDependent Architetture Compatibili

## 4.3 Hologram Run

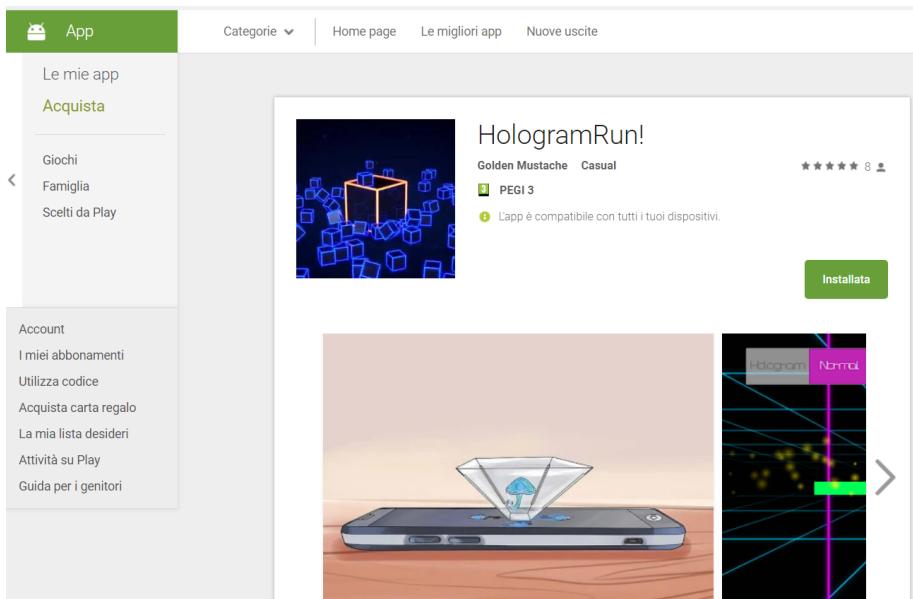


Figura 4.22: Pagina Play Store HologramRun

### 4.3.1 Progettazione

Il secondo progetto è stato sviluppato partendo da un'idea ben precisa cioè quella di riuscire a creare un videogioco olografico. L'olografia è il processo che permette di registrare e visualizzare immagini tridimensionali: tali immagini sono chiamate ologrammi. L'olografia fu scoperta nel 1947 dal fisico ungherese Dennis Gabor. Gli ologrammi originali di Gabor erano registrati e ricostruiti con la luce emessa da lampade a vapori di mercurio. Questi ologrammi oggi non sono un granché, considerati i progressi fatti in questo campo, ma allora costituivano una dimostrazione di un nuovo principio dell'ottica e, per tale merito, fu insignito nel 1971 del premio Nobel per la fisica. Nel 1962 il ricercatore russo Yu. N. Denysiuk, combinando il processo olografico con un metodo della fotografia a colori inventata dal fisico francese Gabriel Lippmann, ottenne ologrammi in luce bianca che sono una pietra miliare nel campo della tecnica olografica. Dopo varie ricerche sulla possibilità o meno di creare ologrammi si è riusciti a trovare e progettare una struttura a prisma in grado di generare, grazie alla forma e al materiale della struttura stessa, un ologramma andando a riprodurre sui 4 lati la stessa immagine e quindi vedendola riflessa all'interno della struttura in maniera omogenea creando l'illusione di avere un oggetto 3D.

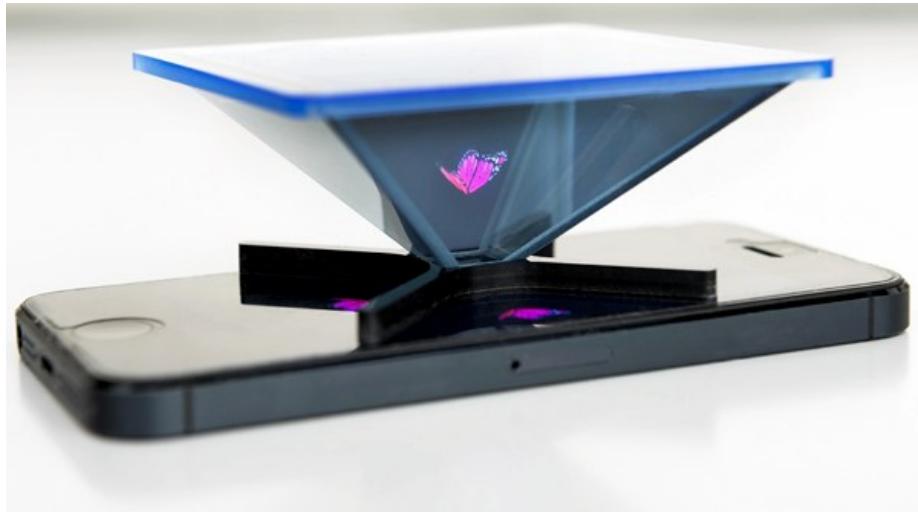


Figura 4.23: Prisma olografico

La struttura per generare gli ologrammi è molto semplice e consiste in un prisma composto da 4 lati costruito con un materiale adatto alla riflessione della luce quale vetro o plastica trasparente. La struttura, per adattarsi meglio al videogioco, è stata realizzata andando a sezionare la cima per ottenere una base minore in modo da poterla poggiare direttamente sul display senza alcun supporto. Il materiale scelto per la realizzazione è stato il plexiglass, in quanto materiale dal costo contenuto e dalla facile reperibilità, che ha garantito una discreta qualità e una notevole leggerezza. Per prima cosa quindi è stato testato l'angolo per una giusta riflessione che, dopo vari test, è stato stimato essere tra i 50 e i 60 gradi prendendo, come asse delle ascisse, lo schermo del telefono e come asse delle ordinate la normale uscente dallo schermo. Per rendere il prisma compatto, si è deciso di creare una struttura formata da 4 facce di forma trapezoidale di dimensioni: base maggiore 12cm, altezza 7cm, base minore 2cm così da formare un angolo di circa 55°.

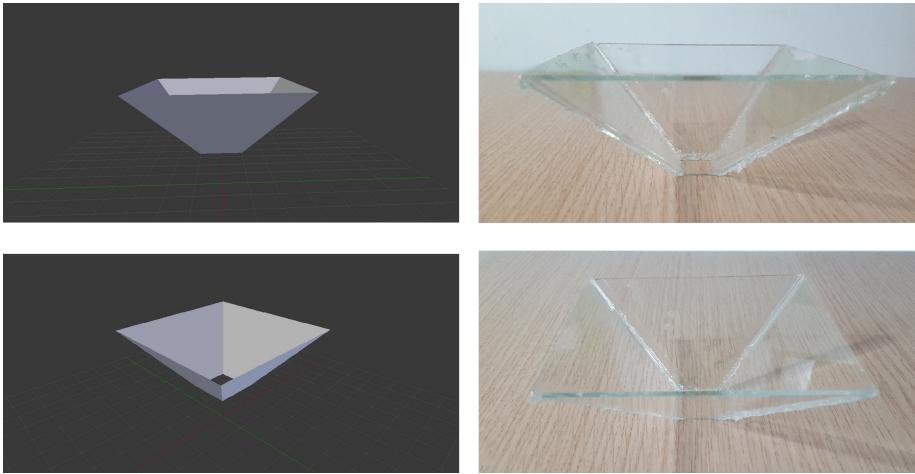


Figura 4.24: Modello 3D Prisma e Riproduzione in Plexiglass

Il videogioco progettato è stato pensato proprio per essere visualizzato come ologramma e questo ha causato non poche limitazioni alla tipologia e alla grandezza del gioco. Esso doveva essere giocato ad una distanza ragionevole (30cm circa) per visualizzare al meglio l'effetto olografico, e questo ha limitato l'interazione che l'utente ha con il gioco. Non potendo usufruire del semplice touch-screen la soluzione adottata è stata quella di usare un controller esterno al dispositivo che fosse economico e di comune utilizzo, per questo sono state accantonate periferiche di input quali mouse, joypad e tastiere Bluetooth. Per queste considerazioni, si è scelto di sviluppare un gioco funzionante a single-input e si è deciso di usare, come periferica predefinita, gli auricolari che possiedono i tasti per la regolazione del volume e delle tracce audio. Questa soluzione permette a quasi tutti gli utenti di interagire con il gioco utilizzando un qualsiasi tipo di auricolari e, grazie a questa periferica, si è riusciti a risolvere il problema dell'interazione a distanza. La scelta del gioco è stata anch'essa limitata dalla tecnologia olografica in quanto il gioco doveva essere riprodotto da quattro angolazioni diverse e tutte e quattro dovevano essere visualizzate simultaneamente sullo schermo di uno smartphone. Questo ha limitato sia la complessità del prodotto che, per essere fluido e performante, doveva quindi possedere un numero limitato di oggetti a schermo ed è stato limitato l'angolo di visuale per lo stesso motivo, in quanto il gioco deve essere visualizzabile in 1/4 dello spazio disponibile sullo schermo. Per i precedenti motivi quindi si è optato per un prodotto molto semplice ma dall'impatto visivo gratificante che, grazie allo stile scintillante dei neon ed ai alcuni effetti particellari modellabili direttamente da Unity, una volta trasmesso sull'ologramma risulta molto interessante sia per via

dell’interazione old-style, tramite gli auricolari, che per la visione dovuta alla grafica minimale e scintillante che garantisce un effetto olografico nitido soprattutto in ambienti poco luminosi. La struttura è facilmente realizzabile partendo da una lastra di plexiglass intagliato in modo da formare 4 trapezi con le misure fornite precedentemente e collegando le forme tra di loro con un collante per plastica.

#### 4.3.2 Produzione e Script

Gli script presenti nel progetto HologramRun sono meno numerosi rispetto a quelli del prodotto precedentemente descritto in quanto il gioco in questione è più semplice e presenta una logica nettamente inferiore.

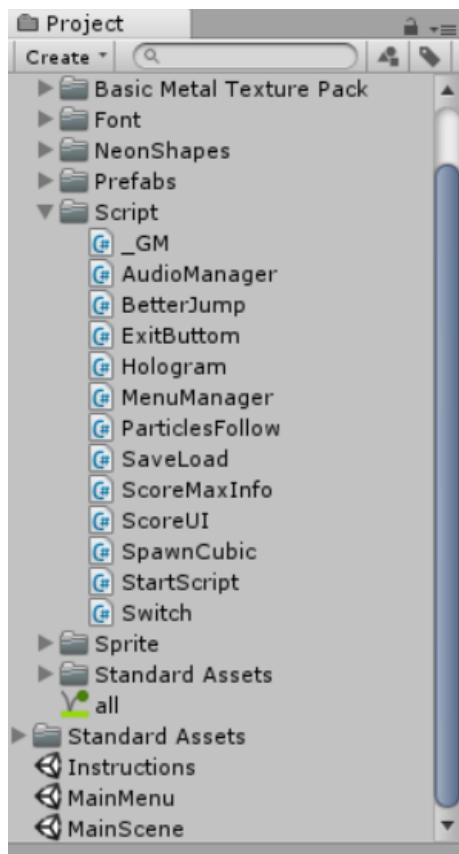


Figura 4.25: Script HologramRun

L’impegno maggiore, nello sviluppo di tale prodotto, è stato dedicato alla fase di progettazione e di setting delle inquadrature, degli oggetti e della resa grafica per ottenere un buon effetto olografico mentre il resto dello sviluppo è

stato dedicato alla programmazione degli script di base e alla compatibilità e mapping delle periferiche di gioco. Di seguito viene riportato lo SpawnCubic script:

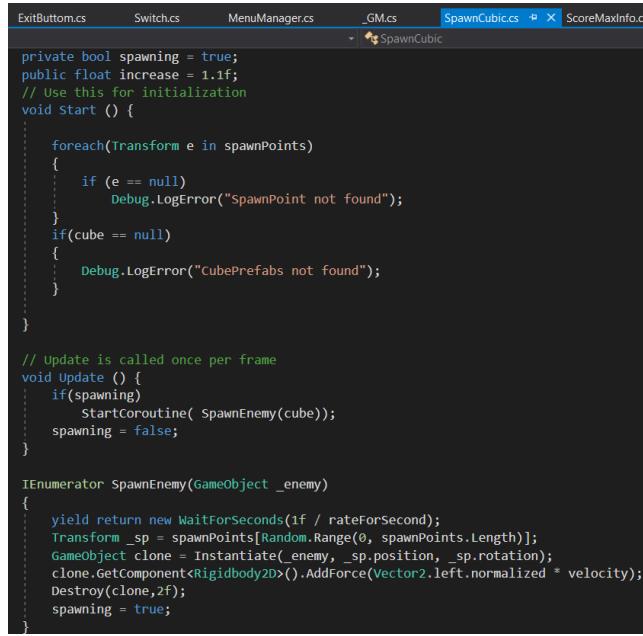


Figura 4.26: Script SpawnCubic

Lo script in questione istanzia i prefabs (oggetti pre-configurati e istanziabili con l'apposito metodo di GameObject) nella posizione desiderata grazie agli oggetti spawnposition che permettono di avere un punto di riferimento all'interno dello script e del mondo di gioco; successivamente aggiunge una forza che li rende dinamici e questa forza è resa direttamente proporzionale al tempo che passa così da rendere il gioco man mano più impegnativo e aumentare il grado di sfida. Per un corretto utilizzo delle risorse di calcolo, dopo un tempo prefissato, il prefab viene distrutto. Per ottenere l'effetto olografico la visuale del gioco doveva essere riprodotta in maniera speculare, sia orizzontalmente che verticalmente, sullo schermo del dispositivo. Tale riproduzione è stata implementata andando a creare 4 oggetti camera che, opportunamente posizionati e ruotati, vanno ad inquadrare la scena di gioco in modo da ottenere l'effetto desiderato. Si è deciso anche di inserire uno script per governare la modalità di visuale in modo da poter scegliere sia la visuale olografica, attivando simultaneamente le 4 camere, e sia la visuale normale per permettere l'interazione anche senza l'utilizzo del prisma.

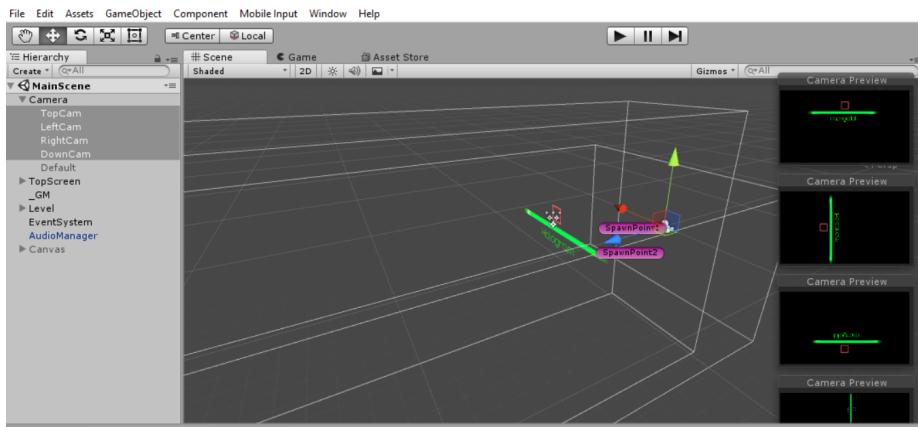


Figura 4.27: Posizionamento multicamera HologramRun

### 4.3.3 Play-Testing

Durante la fase di play-testing si è evidenziato un bug che non permetteva il salto del cubo una volta arrivato quasi alla fine della piattaforma, l’errore era causato da un oggetto che serve allo script di Platform2DCharacter, il ground-Checked, che rileva la collisione con il terreno e serve per gestire i movimenti, la transizione degli stati e le animazioni del cubo. Il groundChecked era situato al centro del cubo e questo impediva di rilevare la collisione una volta che la metà del cubo sporgeva rispetto alla piattaforma quindi è bastato collocare il ground-Checked all’estrema destra del cubo e ridimensionare il parametro m\_grounded, che viene utilizzato come raggio per creare un oggetto Collider di forma sferica, per far in modo che il tutto funzioni.

## 4.4 Post-Produzione

In quest’ultima fase c’è stata l’analisi (postmortem) di tutto lo sviluppo e del processo impiegato. L’idea è stata quella di rimanere il più possibile fedeli ai concetti suggeriti da GameScrum almeno negli aspetti fondamentali. I punti di forza che si sono evidenziati dall’utilizzo di tale metodologia sono diversi:

- **Approccio Sistematico:** Il processo di sviluppo ben suddiviso in fasi ha apportato un notevole beneficio in termini di organizzazione riuscendo a scandire bene il lavoro effettuato e da effettuare.
- **Backlog e Time-Boxing:** La creazione dei Backlog è stata di notevole aiuto in quanto rendeva chiaro ed esplicito il lavoro da effettuare sul prodotto

e l'utilizzo insieme al Time-Boxing ha aumentato l'efficienza del processo perché si aveva una visione perfetta del lavoro da compiere in quel determinato giorno o in quel range di tempo delimitato dalle scadenze.

- **Prototipazione e Feedback:** La prototipazione ha apportato notevoli vantaggi in termini di tempo in quanto le meccaniche prima di essere inserite in maniera definitiva venivano di fatto testate grazie alla creazione di un'Alpha del prodotto che veniva rilasciata solo a persone che collaboravano con il testing per accertarsi della qualità e dell'inserimento o meno della nuova feature. Il feedback, oltre ad essere raccolto in maniera diretta durante questa fase, proviene anche dalla compilazione del questionario che è stato molto utile, soprattutto per quelle persone che non potevano dedicare molto tempo al testing del prodotto, per raccogliere anche le opinioni di persone meno esperte nel settore ma che comunque costituiscono una potenziale utenza attiva.
- **Software:** L'utilizzo del software Unity è stato un vantaggio non indifferente in quanto, grazie soprattutto alle tecniche fornite dall'engine, si è potuto lavorare ad un livello di programmazione molto alto e quindi ignorare (per buona parte del processo) tutto quello che riguarda librerie grafiche, piattaforme, compatibilità e Hardware focalizzandosi solo sul prodotto videoludico e sulle sue meccaniche. L'Assets Store ed in generale la community di Unity è stato un altro aspetto molto utile perché ha garantito una qualità gradevole degli Assets grafici e sonori.

Gli aspetti negativi evidenziati dall'utilizzo del processo sono riconducibili all'assenza di un team di sviluppo in quanto tutto il lavoro è stato effettuato solo dallo scrittore della tesi con qualche collaborazione indispensabile per la creazione di contenuti multimediali quali Sprite, Effetti sonori, Loghi ecc. Tale processo dovrebbe essere utilizzato da team multidisciplinari sia per documentare i benefici ma soprattutto per documentare le difficoltà che possono scaturire dal processo di sviluppo. Per concludere il lavoro di tesi i due prodotti sono stati pubblicati sullo Store Andorid per renderli fruibili gratuitamente su tutti i dispositivi con S.O. Android 4+ nella loro ultima versione (3.0 StarNaitum, 2.5 HologramRun!). I due prodotti hanno attraversato molteplici versioni, che ne hanno aggiunto e raffinato le funzionalità, documentate in un file apposito per mantenere traccia delle modifiche apportate e degli errori o migliorie rilevate.

Nome	Ultima modifica	Tipo	Dimensione
hologram.apk	15/01/2018 12:59	Nox.apk	22.207 KB
hologramColorful.apk	16/01/2018 10:42	Nox.apk	22.363 KB
hologramRun.apk	23/01/2018 17:53	Nox.apk	24.919 KB
hologramRun2.0.apk	24/01/2018 12:00	Nox.apk	24.923 KB
hologramRun2.1.apk	26/01/2018 16:14	Nox.apk	24.907 KB
hologramRun2.2.apk	31/01/2018 18:55	Nox.apk	24.907 KB
hologramRun2.3.apk	01/02/2018 14:31	Nox.apk	24.960 KB
hologramRun2.4.1.apk	03/02/2018 17:21	Nox.apk	24.960 KB
hologramRun2.4.2.apk	05/02/2018 17:23	Nox.apk	25.180 KB
hologramRun2.4.apk	02/02/2018 11:53	Nox.apk	24.960 KB
hologramRun2.5.apk	14/02/2018 09:35	Nox.apk	25.785 KB
starNaitum.apk	12/01/2018 15:14	Nox.apk	27.694 KB
starNaitum2.4.apk	13/01/2018 12:21	Nox.apk	27.694 KB
starNaitum2.5.apk	13/01/2018 17:52	Nox.apk	27.717 KB
starNaitum2.6.1Final.apk	24/01/2018 11:59	Nox.apk	27.882 KB
starNaitum2.6.3.apk	26/01/2018 16:20	Nox.apk	27.882 KB
starNaitum2.6android.apk	23/01/2018 10:57	Nox.apk	27.796 KB
starNaitum2.6Final.apk	23/01/2018 17:15	Nox.apk	27.794 KB
starNaitum2.7.1android.apk	03/02/2018 17:23	Nox.apk	27.678 KB
starNaitum2.7android.apk	31/01/2018 22:41	Nox.apk	27.677 KB
starNaitum2.8.1android.apk	04/02/2018 18:00	Nox.apk	27.035 KB
starNaitum2.8.2.apk	05/02/2018 17:12	Nox.apk	27.039 KB
starNaitum2.8android.apk	04/02/2018 17:02	Nox.apk	27.608 KB
starNaitum2.9.1.apk	11/02/2018 16:55	Nox.apk	26.967 KB
starNaitum2.9.apk	07/02/2018 10:46	Nox.apk	27.039 KB
starNaitum3.0.apk	14/02/2018 17:35	Nox.apk	27.351 KB

Figura 4.28: Versioni Build Android

In conclusione StarNaitum è un giusto compromesso tra divertimento e sfida, mira a coinvolgere il giocatore per la sua difficoltà e sprona a testare nuove build del personaggio per trovare quello più adatto al proprio stile di gioco. Il progetto può ancora svilupparsi molto in quanto può essere inserita una modalità multiplayer locale (utilizzando un canale di comunicazione come il Bluetooth/-Wireless) o possono essere inserite nuove mappe di gioco con una collocazione più o meno difficile delle piattaforme; possono essere inseriti nuovi nemici con altrettante versioni potenziale o può anche essere inserito uno Shop per sbloccare ulteriori sprite per il personaggio, per l’ambiente o per comprare armi con caratteristiche diverse.

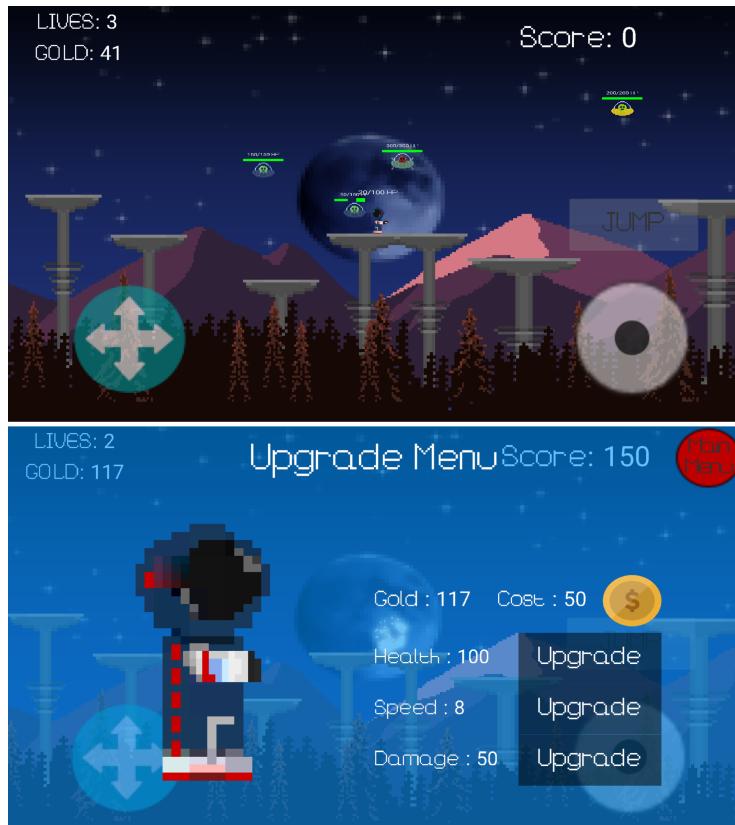


Figura 4.29: Progetto Star Naitum

Analizzando invece HologramRun, uno sviluppo a breve termine potrebbe essere quello dell'inserimento del multiplayer locale in modo da permettere a due utenti di sfidarsi in un 1 vs 1 classico dove la struttura a prisma, sfruttando l'angolo di visuale utilizzato, fa da multischermo proiettando le due partite simultaneamente. Un altro sviluppo potrebbe consistere nel creare una sfida in cui uno dei due utenti governa il cubo rosso e il secondo può decidere dove e come generare i cubi blu per mettere in difficoltà il primo giocatore. Lo sviluppo di applicazioni utilizzando l'ologramma in contesti videoludici ha limiti solo di immaginazione in quanto nessun altro gioco presente tutt'ora sfrutta questa tecnologia a pieno e il suo utilizzo può sicuramente svilupparsi sia per quanto riguarda la qualità dei giochi e sia per quanto riguarda la qualità delle strutture dedicate. Essendo un progetto a lungo termine potrebbe essere interessante sviluppare un simulatore di guida spaziale in 3D per dispositivi di una certa dimensione in modo da avere una visuale abbastanza ampia. Questa tecnologia potrebbe diventare l'alternativa low-cost all'esperienza VR che tanto sta spopo-

lando in questi ultimi anni ma che ancora, sia per colpa dei requisiti hardware richiesti che per il loro costo effettivo, non è una tecnologia accessibile e fruibile da tutti. I progetti olografici, pur non avendo una base solida, possono essere fruiti da molte più persone e garantire comunque un’esperienza visiva immersiva e diversa dal solito schermo.

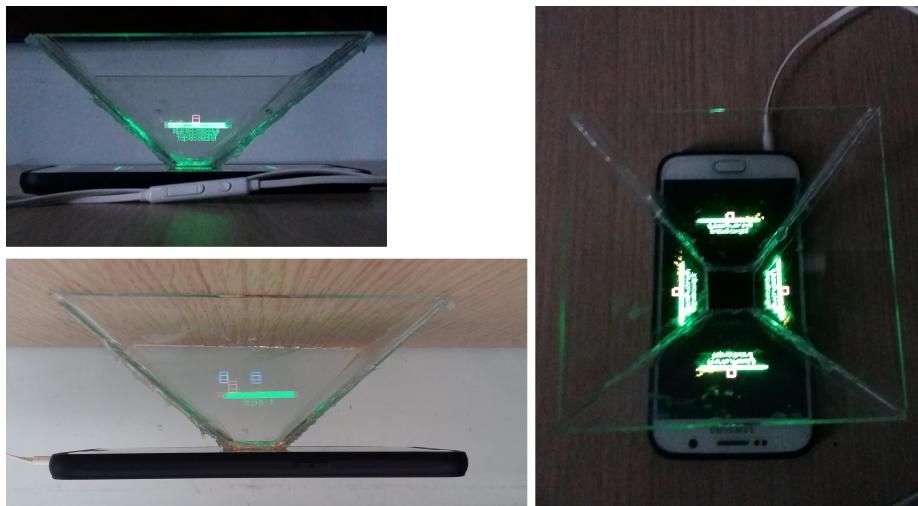


Figura 4.30: Progetto HologramRun!

# Conclusioni

Il lavoro di tesi è stato uno stimolo in più per avvicinarsi al mondo dello sviluppo di videogames. I due progetti sviluppati mi hanno permesso di imparare un software tra i più famosi nello sviluppo di videogames quale Unity e un linguaggio di programmazione tra i più utilizzati in questo ambito cioè C#. Il tutto ha approfondito la mia cultura su tali prodotti grazie all'analisi e alla descrizione della storia, delle tipologie e di tutto quello che tratta la prima parte della tesi. La seconda parte mi ha permesso di approfondire gli aspetti più tecnici del videogioco quali le fasi di sviluppo, le metodologie e il Design di un prodotto videoludico. Durante lo sviluppo dei progetti si sono affrontate le problematiche generali che nascono nello sviluppo di tali prodotti e si sono apprezzati tutti i meccanismi e le caratteristiche di Unity che, soprattutto per la fase di porting e Design dell'ambiente, è stato un software molto utile e facilmente apprendibile grazie soprattutto alla documentazione presente sul sito ufficiale. L'idea e la realizzazione dell'ogramma e del gioco annesso sono state un ottimo pretesto per creare qualcosa di nuovo e di "fresco" in un mercato, quello dei videogiochi, dove è davvero difficile distinguersi per innovazioni vista la vastissima presenza di titoli che vanno a ricoprire più o meno qualsiasi tipologia di gioco esistente. La realizzazione dei due progetti per la tesi mi ha permesso di mettere in pratica le teorie e le nozioni apprese durante la stesura della tesi stessa in modo da toccare con mano le problematiche e gli studi effettuati su tali prodotti.



# Bibliografia

- [1] Rouse, Richard.(2005) “*Game Design: theory & practice by Richard Rouse III*”; illustrations by Steve Ogden.-2nd ed. ISBN 1-55622-912-7 (pbk.)
- [2] Sanjay Madhav. (2014) “*Game Programming Algorithms and Techniques: A Platform-Agnostic Approach*”; ISBN-13: 978-0-321-94015-5; ISBN-10: 0-321-94015-6.
- [3] Chris Totten. (2012) “*Game Character Creation with Blender and Unity*”; ISBN-13: 978-1118172728; ISBN-10: 1118172728.
- [4] Andr e Godoy Ellen F. Barbosa. (2010) “*Game-Scrum: An Approach to Agile Game Development*”, Articolo, Facolt  di Scienze Matematiche e Informatiche di S o Carlos (SP), Brasile.
- [5] Davide Aversa. (2012) “*Primi Passi nel Game Development*”. Articolo, <http://davideaversa.it/slashcode/>.
- [6] StarBytes. (2015) “*Creazione di videogame: le fasi della progettazione*”. Articolo, <https://www.starbytes.it>.
- [7] Lorenzini Tomas Alberto. (2015) “*The videogames's world: market and events*”, Tesi di Laurea, Universit  degli studi di Padova.
- [8] Santina Nibali. (2014) “*Storia ed Evoluzione dei Videogiochi: Dai Coin-Op ai Social-Games*” Tesi di Laurea, Universit  degli Studi di Catania.
- [9] Lorenzo De Donato. (2012) “*Progettazione, Modellazione 3D e Sviluppo di un videogioco multipiattaforma*”. Tesi di Laurea, Universit  degli studi di Cesena.
- [10] NEWZOO. (2015) “*Newzoo summer series #19: Italian-German-US-China-France Games Market*”
- [11] AESVI, Associazione Editori Sviluppatori Videogiochi Italiani. (2016) “*Il mercato dei videogiochi in Italia 2015-2016*”.

- [12] <https://www.gamasutra.com/>, The Art & Business of Making Games.
- [13] <https://en.wikipedia.org/wiki/Game>, Chris Crawford Interview.

# **Ringraziamenti**

Giunto alla fine di questo lavoro sento il desiderio di ringraziare le persone che mi sono state vicine in questo percorso.

Ringrazio il mio relatore, il professore Sergio Flesca che mi ha permesso di lavorare sulla mia passione e mi ha seguito nell'elaborazione della tesi.

Ringrazio la mia famiglia che ha creduto in me prima ancora di me stesso.

Ringrazio la mia ragazza che mi ha sempre supportato e sopportato in questo percorso standomi vicino nei momenti difficili.

Ringrazio il mio gruppo di studio: Totò, Luigge, Angielo, Tony, Nichio, Mukiele e Palmiero, perché senza il loro supporto e il nostro lavoro di squadra non sarei mai arrivato a questo traguardo.

Ringrazio i miei colleghi nonché collaboratori del progetto: Andrea, Tommaso, Mimmo, Cuco e Ciccio, per il loro aiuto e per i giorni di spensieratezza e divertimento vissuti insieme.

Ringrazio i fattarielli per l'amore immenso che provano nei miei confronti e che ricambio sempre.

Ringrazio tutti i miei amici di Amantea e in particolare Andrea e Felice che mi hanno sopportato nelle “depressioni di fine settimana”.

Ringrazio tutti i parenti, i colleghi e in generale le persone che hanno creduto in me in questi anni.

Grazie a tutto il vostro supporto sono diventato la persona che sono adesso: un Peter Pan che crede ancora di poter volare!

