

POLITECNICO DI MILANO
Scuola di Ingegneria Industriale e dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Elettronica, Informazione e Bioingegneria



OrbTail: design e sviluppo di un videogioco multigiocatore
con interoperabilità tra ecosistemi diversi.

Relatore: Prof. Pier Luca LANZI

Tesi di laurea di:
Raffaele Daniele FACENDOLA Matr. 795968

Anno Accademico 2017–2018

Ringraziamenti

Desidero ringraziare i miei genitori, mia sorella Angela e i miei nipoti, per avermi sostenuto in tutti questi anni e per essermi stati vicini nonostante la distanza.

Ringrazio i miei compagni di corso ed amici per avermi accompagnato in questa avventura. I miei colleghi, ormai una seconda famiglia, che mi spronano a dare il meglio di me ogni giorno, ricordandomi che lo sviluppo di videogiochi è più di un semplice lavoro.

Ringrazio Andrea Gandola per la sua immensa disponibilità e professionalità: senza il suo intervento questo prodotto non avrebbe mai raggiunto un simile livello qualitativo. Luca Cafasso per i suoi inestimabili consigli. Giuseppe Spizzico per le interminabili partite online che mi hanno aiutato a superare la stesura della tesi.

Sommario

La creazione di videogiochi multigiocatore in grado di adattarsi a più piattaforme costituisce una sfida complessa, richiedendo opportune considerazioni in termini di design ed architettura del software.

In questo elaborato di tesi si descrive il processo di design e sviluppo di OrbTail, un gioco multipiattaforma a più partecipanti che coniuga elementi tipici di *giochi di guida* e *ad arena* e diverse modalità di gioco in un'esperienza rapida e competitiva. Il prodotto, nato come progetto per il corso di *Videogame Design and Programming*, è stato rielaborato ed esteso, andando ad aggiornare l'intero comparto tecnico, il design e la direzione artistica.

In questo documento sono riportate le fasi di creazione del prodotto e le criticità affrontate per garantire un'elevata consistenza dell'esperienza utente su piattaforme profondamente diverse. Vengono inoltre presentati numerosi paralleli con il *concept* originale del gioco, al fine di individuare carenze e spunti per migliorie.

Indice

Introduzione	1
1 Contesto	3
1.1 Piattaforme	3
1.2 Processo produttivo	4
1.3 Multigiocatore	5
2 Analisi dei titoli esistenti	11
2.1 Crash Team Racing	11
2.2 Rocket League	13
2.3 Geometry Wars 3: Dimensions Evolved	15
2.4 Wipeout Omega Collection	17
2.5 TRON RUN/r	17
3 Design di OrbTail	20
3.1 Meccaniche di base	20
3.2 Modalità di gioco	21
3.3 Veicoli	23
3.4 Livelli	25
3.5 Potenziamenti	26
3.6 Interazione utente	28
3.7 Interfaccia utente	29
3.8 Flusso di gioco	30
4 Direzione artistica	32
4.1 Design dei livelli	32
4.2 Design dei veicoli	34
4.3 Produzione delle risorse grafiche	36
5 Sviluppo	37
5.1 Unity	37
5.1.1 Plug-in	38

5.2 Meccaniche di base	39
5.2.1 Gestione della gravità	39
5.2.2 Sistema di controllo	40
5.2.3 Gestione dei collezionabili	41
5.2.4 Gestione degli scontri	42
5.2.5 Armi e potenziamenti	43
5.2.6 Gestione degli input	44
5.3 Architettura di rete	45
5.3.1 Gestione della lobby	47
5.3.2 Gestione della partita	48
5.4 Intelligenza artificiale	51
5.5 Multigiocatore locale	52
Conclusioni	56
Bibliografia	58

Elenco delle figure

1.1	Overcooked	6
1.2	Heroes of Might & Magic	6
1.3	Mario Karts	7
1.4	Battlerite	8
1.5	EVE Online	8
1.6	PlayerUnknown's Battleground	9
1.7	Battlefield 4	10
2.1	Crash Team Racing	12
2.2	Crash Team Racing - Modalità battaglia	13
2.3	Rocket League - Personalizzazione del veicolo	14
2.4	Rocket League	15
2.5	Geometry Wars - Arena sferica	16
2.6	Geometry Wars - Arena ad imbuto.	17
2.7	Wipeout Omega Collection - Ambientazione	18
2.8	Wipeout Omega Collection - Esempio di nave	18
2.9	TRON RUN/r - Tron City	19
2.10	TRON RUN/r - Whiteout	19
3.1	Elemento dell'HUD con funzione di bussola	29
3.2	Flusso di gioco	30
4.1	OrbTail - Stadium	33
4.2	OrbTail - Planetarium	34
4.3	OrbTail - Torus	34
4.4	OrbTail - Design dei veicoli	35
4.5	OrbTail - Esempio di livrea	35
5.1	Struttura dei veicoli	39
5.2	Gestione della gravità.	40
5.3	Gestione dei collezionabili.	41
5.4	Gestione dei potenziamenti	43

5.5 Gestione degli input	45
5.6 Architettura di rete	46
5.7 Modalità di gioco	49
5.8 Flusso di gara	49
5.9 Intelligenza artificiale - Strategie	51
5.10 Gestione multigiocatore locale	53
5.11 Gestione telecamera	54
5.12 Splitscreen	54

Elenco delle tabelle

3.1 Parametri dei veicoli	24
-------------------------------------	----

Introduzione

I videogiochi rappresentano una delle forme di intrattenimento moderne più affermate e diffuse. A differenza di quanto avviene con i mezzi classici quali *film* e *libri*, l'utente ha pieno controllo dell'esperienza: alcuni giochi richiedono concentrazione, altri una buona dose di coordinazione e reattività, altri ancora pazienza e pianificazione. Sebbene vi siano videogiochi che si concentrano principalmente sulla componente a giocatore singolo, i giochi, intesi anche nel senso classico del termine, nascono come mezzo di *condivisione* dell'esperienza con altri partecipanti.

In questa tesi si descrive il processo di design e sviluppo di OrbTail, un videogioco multigiocatore con elementi tipici dei giochi di guida disponibile per dispositivi *mobile* e *desktop* quali Android, iOS, Windows e OSX. Il pregio principale di OrbTail risiede nella sua capacità di poter essere giocato da più giocatori contemporaneamente, online o meno, secondo una qualsiasi combinazione di piattaforme e\o numero di giocatori locali attraverso un'opportuna modalità *cross-platform play*¹. Vi è inoltre la possibilità di condividere uno stesso dispositivo *desktop* tra due o più giocatori locali mediante un'apposita modalità *splitscreen*². Il gioco offre tre modalità di gioco differenti ed altrettante arene e consente ai giocatori di controllare sei veicoli caratterizzati da aspetto e stili di guida diversi.

La necessità di supportare più piattaforme, impedendo che ciascuna di esse potesse risultare avvantaggiata rispetto alle altre durante le sessioni *cross-platform play*, ha portato allo sviluppo di un *gameplay*³ essenziale ed immediato. La durata limitata delle sessioni, unita ad una curva d'apprendimento semplice, rende il prodotto particolarmente adatto ad un pubblico di *casual gamer*⁴.

Questa tesi è strutturata in capitoli.

¹Il termine *cross-platform play* identifica una modalità di gioco online in cui giocatori possono giocare tra loro indipendentemente dalla piattaforma utilizzata.

²Il termine *splitscreen* identifica una modalità di gioco in cui lo schermo viene suddiviso in più quadranti in modo da consentire a più giocatori di usare uno stesso dispositivo contemporaneamente.

³Il *gameplay* (in italiano traducibile come “*esperienza utente*”) comprende gli elementi di gioco quali storia, regole, obiettivi, progressioni, interazione utente, ecc.

⁴Si definisce *casual gamer* un giocatore saltuario, non particolarmente interessato alla cultura dei videogiochi e, solitamente, alla tecnologia in generale.

Nel primo capitolo viene dato un contesto al progetto, fornendo una panoramica delle piattaforme disponibili ed andando ad analizzare i requisiti dello sviluppo multipiattaforma e delle modalità multigiocatore.

Nel secondo capitolo vengono analizzati alcuni titoli esistenti, al fine di individuare spunti per le *meccaniche* di gioco e la direzione stilistica del progetto.

Nel terzo capitolo vengono delineato il design del gioco, partendo dalle meccaniche di base e descrivendo le varie modalità, i livelli, i veicoli e le loro caratteristiche.

Nel capitolo successivo verrà inquadrata la direzione artistica del progetto, quale l'ambientazione ed il design dei veicoli. Vengono inoltre analizzati i requisiti delle risorse grafiche prodotte considerando le limitazioni tecniche e di design.

Il quinto capitolo verte sulle scelte tecnologiche, a partire dalla scelta del motore grafico, delle piattaforme e dell'architettura del gioco. Ogni scelta verrà contextualizzata rispetto a varie alternative e opportunamente giustificata, considerando necessariamente i requisiti di design.

Il capitolo finale è dedicato alle conclusioni nonché ad eventuali sviluppi futuri.

Capitolo 1

Contesto

1.1 Piattaforme

L'esperienza videoludica ha il grande vantaggio di poter essere studiata per soddisfare utenze dai gusti profondamente diversi su un vasto numero di piattaforme. Ad oggi esistono decine di tipologie di dispositivi di intrattenimento differenziati per costo, performance e modalità d'interazione.

Le piattaforme *mobile*, pur essendo nate solo di recente, si sono subito affermate come una delle principali piattaforme da gioco. Il successo senza precedenti di *smartphone* e *tablet* ha consentito all'industria mobile di generare un volume d'affari pari a quello di *pc* e *console* combinati [2].

Pur non godendo delle stesse performance e manifestando un ciclo di vita più breve rispetto alle controparti classiche, i dispositivi *mobile* godono di una più vasta diffusione e si rivolgono ad un pubblico decisamente più ampio.

L'interazione primaria affidata all'*input touch* è affiancata da un grande numero di sensori accessori quali *GPS*, *accelerometri*, *giroscopi*, *bussole* e *videocamere*. Recenti sviluppi tecnologici quali *realità aumentata*¹ e *realità virtuale*², unito all'uso sapiente di queste nuove modalità d'interazione, ha permesso la nascita di nuovi paradigmi di *gameplay* rimasti finora inesplorati.

I *pc* e le *console* sono le piattaforme storiche su cui sono nati e diffusi i videogame. Accomunate da paradigmi d'interazione e performance paragonabili, queste due famiglie di dispositivi sono nate per scopi diversi. Laddove le *console* sono pensate per essere un dispositivo d'intrattenimento *dedicato*, che fa della *facilità d'uso* il suo cavallo di battaglia, i *pc* hanno una natura più generica e garantiscono performance

¹La realtà aumentata consiste nell'arricchire il mondo circostante attraverso contenuti di tipo visivo, aptico o uditorio generati da un elaboratore.

²La realtà virtuale consiste nel simulare un ambiente tridimensionale tramite un elaboratore e lasciare che l'utente vi interagisca attraverso periferiche di input specializzate quali visori, cuffie e pad.

e flessibilità maggiori al costo di una richiesta più elevata di competenza da parte dell’utenza.

Il ciclo di vita e il costo delle due piattaforme è inoltre molto diverso. Le *console* sono dei *sistemi embedded* precostruiti e limitatamente aggiornabili; il loro ciclo di vita è generalmente superiore al quinquennio e sono caratterizzati da un costo relativamente basso. I *pc*, d’altro canto, possono essere assemblati scegliendo i componenti che meglio si adattano alle esigenze dell’utente ed aggiornati quando ritenuto necessario. Il costo di un *pc* è generalmente molto più alto rispetto a quello di una console, anche a parità di specifiche tecniche.

L’ampio parco di periferiche di input supportate, dalle più comuni quali tastiere, mouse e gamepad a quelle più specializzate quali volanti, pedaliera e *HOTAS*³, rendono queste piattaforme adatte a qualsiasi tipologia di gioco ed interazione utente.

1.2 Processo produttivo

Il processo produttivo di un videogioco cambia radicalmente in funzione della piattaforma e dell’utenza a cui viene destinato il prodotto. Un design di successo deve essere in grado di sfruttare le peculiarità di ciascun dispositivo e considerarne le limitazioni tecniche, senza pregiudicare l’esperienza utente.

Lo sviluppo risulta solitamente tanto più avvantaggiato quanto minori sono le limitazioni delle piattaforma o varietà di specifiche tra dispositivi. La presenza di dispositivi *identici* tra *console* appartenenti ad uno stesso ecosistema garantisce un’elevata consistenza dell’esperienza utente e consente interventi di ottimizzazione ad una granularità molto fine. Laddove questo processo è solitamente molto *efficiente* su console, lo stesso non può essere detto per piattaforme *mobile* e *desktop* per via dell’elevata varietà di specifiche tecniche o combinazioni di componenti. I *pc* risultano avvantaggiati per via delle elevate performance e assenza di grosse limitazioni e ciò consente loro di mantenere un elevata fedeltà e fruibilità del contenuto. Lo sviluppo su dispositivi quali *smartphone* e *tablet*, d’altro canto, deve scontrarsi con la presenza di dispositivi con capacità profondamente diverse e compatti tecnici non sempre in equilibrio tra loro (non è raro assistere a dispositivi che associano elevate risoluzioni a performance mediocri). In questo caso è richiesto uno sforzo maggiore affinché l’applicazione scali in funzione del dispositivo per garantire una buona esperienza utente.

Sebbene esistono titoli sviluppati in *esclusiva* per alcune piattaforme, la necessità di aumentare il bacino d’utenza, e di conseguenza i ricavi, solitamente richiede che un prodotto venga distribuito su più ecosistemi. Un tempo processo lungo ed

³L’*HOTAS* (acronimo di «*hands on throttle and stick*»), usato solitamente per giocare a simulatori di volo, consiste di un joystick a 4 o più assi e una leva d’accelerazione.

oneroso per via delle marcate differenze tra le architetture e carenza di strumenti di sviluppo, ad oggi lo sviluppo multipiattaforma risulta molto avvantaggiato. Il rilascio di motori grafici di terze parti, unito alla convergenza delle piattaforme verso architetture simili, permette agli sviluppatori di lavorare ad un livello di astrazione più elevato evitando di creare supporti di basso livello specifici per dispositivo.

Laddove sviluppare un videogioco per piattaforme simili quali potrebbero essere *pc* e *console* oppure *smartphone* e *tablet*, costituisce più un problema implementativo che non a livello di gameplay, sviluppare per piattaforme profondamente diverse richiede importanti considerazioni a livello di *design* e produzione delle risorse di gioco. In primo luogo il differente grado di apprezzamento dell'utenza delle varie piattaforme può precludere il successo a certe tipologie di gioco di nicchia (quali potrebbero essere simulatori o strategici) e favorire i design che si rivolgono al grande pubblico. Le differenti modalità d'interazione potrebbero inoltre richiedere la rivisitazione dell'interfaccia grafica e l'eventuale eliminazione degli input disponibili solo su certe tipologie di dispositivi. Il comparto tecnico deve inoltre consentire al prodotto di poter scalare in funzione delle performance a disposizione. Possibili interventi consistono nella riduzione del dettaglio delle *texture* o della complessità poligonale, riduzione o rimozione dell'effettistica, limitazione degli oggetti a schermo. Ove ciò non fosse possibile o insufficiente potrebbe essere necessario rivisitare il *gameplay* attraverso la riduzione di giocatori o avversari, semplificazione della *IA* o rimozione di funzionalità particolarmente onerose.

1.3 Multigiocatore

L'interazione sociale come mezzo per aumentare il coinvolgimento videoludico è il motivo principale per cui i videogiochi multigiocatore hanno da sempre riscosso un grande successo. Sebbene esistono infinite variazioni sul tema, le modalità multigiocatore possono essere classificate in due famiglie: *cooperative* e *competitive*. Alla prima categoria appartengono quelle modalità in cui due o più giocatori *collaborano* tra loro al fine di raggiungere un obiettivo comune. In queste modalità l'elemento di sfida è rappresentato dal gioco stesso e governato da *intelligenze artificiali* più o meno sofisticate. Nelle modalità *competitive* l'elemento di sfida è invece rappresentato dai giocatori stessi: gli utenti sono portati a confrontarsi gli uni con gli altri al fine di raggiungere un obiettivo impedendo al contempo che gli altri giocatori possano fare altrettanto. Il livello di sfida offerto da alcuni titoli è tale per cui, pur di eccellere, alcuni giocatori hanno deciso di farne una carriera, sottoponendosi costantemente a veri e propri allenamenti. Questo fenomeno ha di recente portato alla nascita di competizioni a livello agonistico, organizzate e regolamentate da entità terze, in cui i partecipanti non sono semplici giocatori ma veri e propri atleti. Queste competizioni prendono il nome di *e-sport* (sport elettronici).

Le modalità multigiocatore hanno accompagnato lo sviluppo dei videogiochi fin dagli albori, in un'epoca in cui la diffusione di Internet era molto limitata. Le realizzazioni più semplici consistono nello sfruttare uno stesso dispositivo col quale tutti i giocatori possono interagirvi. La presenza fisica di tutti i partecipanti coinvolti favorisce un'interazione sociale più immediata e rende queste modalità di gioco particolarmente adatte ai *party-game*⁴. Le implementazioni più semplici consistono nell'utilizzare una visuale condivisa tra tutti i partecipanti (Fig. 1.1) oppure, laddove l'interazione in contemporanea non fosse necessaria, sfruttare la meccanica dei *turni* affinché ciascun giocatore possa godere di un punto di vista unico sul mondo di gioco (Fig. 1.2).



Figura 1.1: Overcooked - Party game a visuale condivisa



Figura 1.2: Heroes of Might & Magic - Multigiocatore locale a turni

⁴I *party-game* sono l'equivalente videoludico dei giochi di società.

Un’implementazione più sofisticata, nota col termine *splitscreen*, consiste nel suddividere lo schermo in quadranti, solitamente da due a quattro, e mostrare in ciascuno di essi il punto di vista di uno dei giocatori in maniera indipendente (Fig. 1.3). L’elevato impatto sulle performance, tale da rendere necessarie ottimizzazioni



Figura 1.3: Mario Karts - Multigiocatore in splitscreen

particolarmente aggressive, unita al ridotto bacino d’utenza cui queste modalità si rivolgono, ha di recente portato ad un calo di titoli che offrono questo tipo di esperienza.

La rapida diffusione di Internet, unita ai suoi sviluppi degli ultimi decenni, ha consentito la nascita di nuove modalità multigiocatore *online*. Rispetto all’approccio classico, un multigiocatore online permette agli utenti di usare il proprio dispositivo per accedere a sessioni di gioco con un numero di partecipanti che varia tra la decina e il migliaio, indipendentemente dalla distanza fisica che li separa. Il rinnovato successo dei giochi multigiocatore ha consentito la nascita di nuovi generi, dai *MOBA* in cui due squadre dal numero ristretto di giocatori si sfidano all’interno di un’arena (Fig. 1.4), agli *MMO* i cui mondi persistenti vantano migliaia di giocatori attivi contemporaneamente (Fig. 1.5), passando per i più recenti *Battle Royale* caratterizzati da centinaia di giocatori che lottano per la sopravvivenza (Fig. 1.6).

Le implementazioni più comuni sono riconducibili a due diverse *architetture*, in funzione del ruolo che i diversi dispositivi hanno all’interno della sessione di gioco. Nell’architettura *client-server* i dispositivi dei giocatori, detti *client*, sono connessi ad un dispositivo centrale, detto *server*, il cui ruolo consiste nel coordinare tutti i partecipanti, gestire l’evoluzione della partita, propagare lo stato condiviso di gioco ed eventualmente validare le azioni dei singoli giocatori per evitare *cheat*⁵. I *client* comunicano esclusivamente col *server* limitandosi a dichiarare le azioni effettuate

⁵Il *cheat*, in italiano «imbrogliare», rappresenta una qualsiasi tecnica atta a sovvertire le regole di gioco affinché un giocatore ne ottenga un beneficio immeritato.



Figura 1.4: Battlerite - MOBA: Multiplayer online battle arena.



Figura 1.5: EVE Online - MMO: Massive multiplayer online game

dall'utente. Il ruolo di *server* può essere ricoperto sia da una macchina esterna che non prende parte alla partita detta *server dedicato*, oppure direttamente da uno dei dispositivi dei giocatori, detto *host*. L'approccio con *server dedicato* permette di sfruttare macchine remote caratterizzate dalla elevate performance al fine di gestire efficientemente un numero anche elevato di giocatori. Il secondo di questi, invece, consente di evitare i costi associati alla gestione o noleggio del server al costo di una minore scalabilità. L'esperienza multigiocatore con questo tipo di approccio è inoltre soggetta a fattori difficilmente prevedibili quali prestazioni del dispositivo che fa da *host*, qualità della connessione verso i *client* e la possibilità di disconnessione del *server* durante una partita che potrebbe richiedere il *trasferimento* del ruolo di server ad uno degli altri dispositivi (processo noto col termine *host migration*).

Nelle architetture *peer-to-peer*, a differenza di quanto avviene con l'approccio



Figura 1.6: PlayerUnknown’s Battleground - Battle royale

client-server, lo stato di gioco è condiviso su tutti i dispositivi e gestito in maniera distribuita. Sebbene questa architettura impedisce che il carico computazionale si concentri su un unico dispositivo, evitando inoltre i costi associati a macchine server esterne, la condivisione dell’*autorità* sullo stato di gioco favorisce il proliferare del fenomeno del *cheating*, oltre che a complicare la sincronizzazione tra vari dispositivi e la risoluzione di eventuali discrepanze nello stato di gioco.

Per far fronte al numero crescente di giocatori all’interno di ciascuna sessione di gioco, le implementazioni più moderne si basano su un approccio *misto client-server* in cui parte della computazione viene delegata ai vari client che, pertanto, possiedono un certo grado di autorità sulla partita.

Sebbene esistano numerosi paradigmi consolidati per la gestione dei giochi multigiocatore, la quasi totalità delle implementazioni *segmentano* il bacino d’utenza in funzione della piattaforma d’appartenenza. Questo approccio semplifica enormemente la gestione dei servizi online accessori, specifici per piattaforma e molto diversi tra loro, evitando i costi associati alla gestione dell’interoperabilità tra di essi, tuttavia impedisce che giocatori su piattaforme diverse possano giocare gli uni contro gli altri. Esistono alcuni esempi di giochi multiplattforma che consentono a giocatori su *smartphone* e *tablet* di interagire ed influenzare partite in corso su altre piattaforme (quali *pc* e *console*), tuttavia essi rappresentano una nicchia ristretta che non ha mai riscosso un vero e proprio successo per via dei ruoli particolarmente asimmetrici e gameplay profondamente diversi tra vari dispositivi (Fig. 1.7).



(a) PC - First person shooter. Il *soldati* prendono controllo di punti strategici facendo guadagnare risorse al *comandante*.



(b) Mobile - Strategico. Il *comandante* gestisce la conquista dispiegando soldati e risorse, lanciando attacchi missilistici e proteggendo i propri soldati dagli avversari.

Figura 1.7: Battlefield 4 - Multigiocatore online con interoperabilità pc\mobile

Capitolo 2

Analisi dei titoli esistenti

In questo capitolo vengono presentati alcuni titoli esistenti al fine di analizzarne le meccaniche di base e trovare spunti interessanti da integrare all'interno del *design* di OrbTail. Verranno inoltre analizzate le scelte stilistiche in modo da individuare una possibile direzione artistica da seguire durante la produzione del videogioco. I titoli sono stati selezionati tra diverse piattaforme in modo da individuare un sottoinsieme di caratteristiche desiderate e facilmente adattabili alla grande diversità di dispositivi per cui verrà sviluppato il prodotto. La data di pubblicazione dei titoli scelti copre inoltre un orizzonte temporale molto vasto: l'intento è quello di coniugare tra loro caratteristiche dei giochi recenti con vecchi paradigmi di gameplay da rivisitare in chiave moderna.

2.1 Crash Team Racing

Crash Team Racing, sviluppato da *Naughty Dogs* e pubblicato da *Sony Computer Entertainment* nel 1999 per *PlayStation*, è un videogioco di guida tratto dalla fortunata serie platform *Crash Bandicoot*. Il gioco ruota attorno alle vicende di una squadra di piloti di *kart* intenti a difendere la Terra da un alieno, *Nitros Oxide*, il quale vuole trasformare il pianeta in un parcheggio e rendere schiavi i suoi abitanti.

Nel gioco l'utente controlla uno tra quindici personaggi disponibili, contraddistinti ciascuno da un proprio kart dalle caratteristiche uniche. I veicoli sono in grado di accelerare, sterzare, frenare e *saltare*; la meccanica del *power slide* garantisce inoltre un'accelerazione temporanea durante i *drift*. I veicoli sono caratterizzati da tre parametri fondamentali, *velocità*, *accelerazione* e *manovrabilità*, i quali influenzano lo stile di guida e la difficoltà del personaggio.

All'interno di ogni circuito sono distribuite delle *casse* speciali che, raccolte, garantiscono al giocatore un potenziamento casuale, tanto più potente quanto più svantaggiosa la sua posizione nella gara corrente. Questi oggetti collezionabili comprendono un ricco assortimento di armi usate per intralciare gli avversari, barriere difensi-



Figura 2.1: Crash Team Racing - Gara a giocatore singolo.

ve e bonus temporanei alla velocità del veicolo. Speciali frutti «wumpa» consentono ai veicoli di procedere più velocemente e migliorare i suddetti potenziamenti.

Il gioco offre cinque modalità di gioco, di cui tre a giocatore singolo (Fig. 2.1) e due multigiocatore locale in splitscreen. Nella modalità «avventura» il giocatore seleziona uno dei personaggi a disposizione, gareggiando su sedici circuiti diversi, al fine di collezionare *trofei*, *reliquie* ed altri oggetti necessari per poter proseguire con la trama. I circuiti sono suddivisi in diversi *mondi*, ciascuno contenente quattro tracciati, al termine dei quali viene richiesto di fronteggiare un *boss* in una gara testa a testa. All'interno della modalità avventura sono inoltre presenti diverse *sotto-modalità* che cambiano leggermente le regole delle varie sfide: collezionare oggetti nascosti all'interno del circuito entro la fine della gara, vincere più gare in successione o terminare la gara entro un tempo limite usando un singolo potenziamento in grado di congelare il tempo per pochi secondi. Nella modalità «sfida a tempo», il giocatore corre da solo al fine di registrare il miglior tempo su tutti i tracciati disponibili. Questa modalità è caratterizzata dall'assenza di avversari e potenziamenti di alcun genere. Nella modalità «sala giochi» il giocatore può gareggiare su circuito a scelta o organizzare *campionati* formati da quattro tracciati. In questa modalità è prevista la presenza di otto partecipanti e l'intero arsenale di armi e potenziamenti. Quest'ultima modalità è anche fruibile in formato multigiocatore locale con numero di giocatori compreso tra due e quattro. L'ultima modalità «battaglia» stravolge le regole del gioco: i partecipanti si sfidano all'interno di un'arena chiusa collezionando armi e potenziamenti al fine di colpire e danneggiare gli avversari (Fig. 2.2). Esistono diverse condizioni di vittoria configurabili: una richiede il raggiungimento di un certo numero di punti guadagnabili colpendo gli avversari, un'altra ha un tempo limite scaduto il quale il giocatore col punteggio più alto viene proclamato vincito-



Figura 2.2: Crash Team Racing - Modalità «battaglia» tra due giocatori in splitscreen.

re, nell'ultima ogni giocatore inizia la sfida con un numero predefinito di vite che vengono decrementate ogni volta che il kart viene colpito da un'arma. Quando il numero di vite scende a zero il giocatore viene eliminato: l'ultimo rimasto in partita è proclamato vincitore. Questa modalità è caratterizzata dall'assenza di partecipanti governati dall'*intelligenza artificiale* ed è pertanto preclusa dall'esperienza a giocatore singolo.

2.2 Rocket League

Rocket League è un gioco multiplattaforma sviluppato e pubblicato da *Psyonix* nel 2015 per *Windows* e *PlayStation 4* e successivamente rilasciato per *OSX*, *Linux* e *Nintendo Switch*. Il gioco combina due generi, quello *sportivo* e quello *corsa*, in un gameplay inedito in cui due squadre formati da potenti veicoli si sfidano ad una partita di *calcio* all'interno di uno *stadio*.

Ogni giocatore ha a disposizione decine di veicoli differenti, identici nelle prestazioni e modelli di guida, ma riccamente personalizzabili tramite *livree*, *decalcomanie*, *pneumatici*, *razzi* e molto altro ancora (Fig. 2.3). I veicoli sono inoltre in grado di saltare e roteare in volo al fine di colpire la palla, effettuare salvataggi o portarsi in vantaggio rispetto agli avversari. Un potente razzo posteriore consente di aumentare drasticamente la propria velocità, planare o spiccare il volo per brevi periodi. È inoltre possibile usare il proprio veicolo come un ariete contro i veicoli avversari in modo da farli esplodere e rimuoverli dalla partita per pochi secondi.

Ogni *stadio* è caratterizzato da uno stile diverso. La presenza di zone speciali all'interno di essi consente ai veicoli di ottenere un'accelerazione aggiuntiva quan-



Figura 2.3: Rocket League - Personalizzazione del veicolo

do attraversate; particolari oggetti distribuiti nello stadio consentono al veicolo di recuperare parte dell’energia necessaria per azionare i razzi posteriori.

Il gioco offre diverse modalità a *giocatore singolo* e *multigiocatore locale* tramite *splitscreen*, *online* o *misto*. Le modalità *online* sono inoltre fruibili nel formato *cross-platform play* tra *Windows* e *PlayStation 4* oppure tra *Windows*, *XBoxOne* e *Nintendo Switch*. La modalità di gioco principale «Carlio» (gioco di parole originato dalle parole «car» e «calcio») prevede la sfida tra due squadre di dimensione variabile tra uno e quattro giocatori (Fig. 2.4). I partecipanti usano i propri veicoli per colpire una grossa palla al fine di spingerla nella porta avversaria ed ottenere punti. Allo scadere del tempo la squadra col maggior numero di punti è dichiarato vincitore. In caso di parità la prima squadra a segnare oltre lo scadere del tempo vince. Ogni giocatore ha inoltre un proprio punteggio personale che può essere incrementato tirando la palla in porta, segnando, effettuando salvataggi, distruggendo i veicoli avversari e così via. Aggiornamenti successivi del gioco hanno introdotto nuove modalità di gioco quali «Snowday» ispirata ad una partita di *hockey su ghiaccio* in cui la palla è sostituita da un *disco* caratterizzato da una fisica diversa, «Hoops» ispirata alla *pallacanestro* in cui la porta è sostituita con un *canestro* ed infine «Rissa» in cui i giocatori possono usare diversi potenziamenti casuali al fine di congelare la palla, agganciarla con una fune e così via. L’ultima modalità di gioco introdotta nel 2017 «Dropshot» stravolge le regole del gioco, rimuovendo completamente le porte. In questa modalità il pavimento è suddiviso in piastrelle esagonali che, colpiti dalla palla due volte in successione, crollano lasciando una voragine nel pavimento che funge da porta per la squadra a cui appartiene la mezzeria di campo.

Il gioco presenta inoltre una modalità *competitiva online* basata sulle regole principali le cui *stagioni* hanno durata pari ad alcuni mesi. I giocatori sono organizzati



Figura 2.4: Rocket League

in *categorie*, in funzione del proprio livello di abilità, e suddivise a loro volta in quattro *divisioni*; i risultati della squadra consentono allo stesso di cambiare di divisione all'interno della *categoria* d'appartenenza.

2.3 Geometry Wars 3: Dimensions Evolved

Geometry Wars 3: Dimensions Evolved è un gioco *shoot 'em up*¹ sviluppato da *Lucid Games* e pubblicato da *Sierra Entertainment* nel 2014 per tutte le maggiori piattaforme *desktop*, *mobile* e *console*. Terzo capitolo della saga *Geometry Wars* è il primo gioco della serie ad introdurre una modalità «avventura» ed un ambiente di gioco totalmente tridimensionale (Fig. 2.5) .

Nel gioco l'utente controlla una piccola astronave in grado di muoversi rapidamente e sparare in qualsiasi direzione al fine di evitare e distruggere ondate di astronavi nemiche. Il giocatore ha a disposizione un *drone* che lo accompagnerà in battaglia aumentandone la potenza di fuoco ed aiutandolo a collezionare gli oggetti sparsi per la mappa e una *super* abilità dagli effetti devastanti utilizzabile un numero limitato di volte. Esistono diversi *droni* e *super* abilità, ciascuno caratterizzato da diversi stili di gioco ed effetti. Durante le partite è inoltre possibile collezionare *super stati*, potenziamenti generati casualmente nel livello che forniscono *barriere*, *armi potenziate* e *magneti* per brevi periodi di tempo, e *geom*, oggetti che fungono da *moltiplicatore di punteggio* ottenibili abbattendo le navi avversarie. Collezionare *geom* è una delle meccaniche fondamentali del gioco in quanto consente al giocatore di aumentare il proprio punteggio di svariati ordini di grandezza. Al raggiungimento

¹I videogiochi *shoot 'em up*, letteralmente «spara a tutti», sono caratterizzati da meccaniche in cui il giocatore deve sparare continuamente a qualche obiettivo di gioco.



Figura 2.5: Geometry Wars 3: Dimensions Evolved - Arena sferica.

di certe soglie di punti, il giocatore viene inoltre premiato con *vite extra* o potenti *bombe* in grado di distruggere tutte le astronavi nemiche.

I livelli sono caratterizzati da diverse *topologie* e *dimensioni* e si sviluppano formando solidi tridimensionali quali *sfere*, *cubi* e *nastri* percorribili lungo la superficie (Fig. 2.6). I nemici vengono generati ad *ondate* in punti specifici di ciascuna mappa e sono caratterizzati da diversi schemi di movimento e strategie: alcuni tendono ad essere più aggressivi, altri più rapidi, altri ancora prediligono una strategia più elusiva. La successione delle ondate in ciascun livello è prestabilita ed è pertanto possibile ricordarla a memoria al fine di massimizzare il punteggio a fine gara.

Il gioco prevede *dodici* modalità di gioco a giocatore singolo organizzate in una modalità «classica» che comprende *sei* delle modalità storiche della saga e giocabili su qualsiasi livello ed una modalità «avventura» in cui il giocatore affronta *cinquanta* livelli diversi in sequenza, ciascuno dei quali caratterizzato da regole proprie. Le modalità di gioco consistono nel ottenere il maggior numero di punti avendo a disposizione solo un numero limitato di vite ed oggetti speciali oppure entro un tempo limite prestabilito. Alcune modalità inoltre introducono diverse regole, quali numero limitato di munizioni, assenza di armi, presenza di nemici in grado di dividersi, aree di gioco che diventano via via sempre più ristrette, e così via. Durante la modalità «avventura» il giocatore sarà costretto ad affrontare dei *boss*, potenti nemici in grado di sostenere molteplici danni e in grado di usare speciali scudi per rigenerare la propria *salute*. Sono inoltre disponibili due modalità *multigiocatore*: una *locale* cooperativa a visuale condivisa, caratterizzata dalla sola presenza di arene bidimensionali e un numero di giocatori compreso tra due e quattro, ed una *online competitiva* fino ad un massimo di otto giocatori.

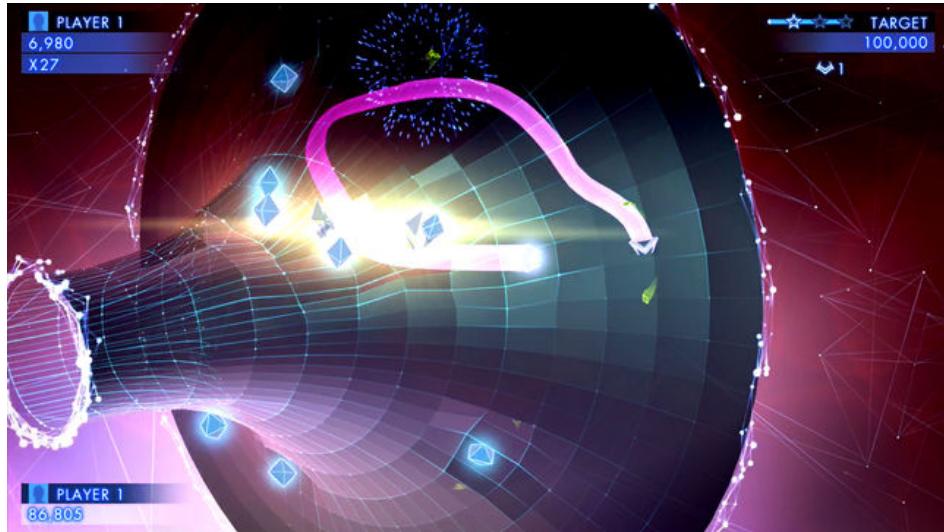


Figura 2.6: Geometry Wars 3: Dimensions Evolved - Arena ad imbuto.

2.4 Wipeout Omega Collection

Wipeout Omega Collection è un videogioco di *corse futuristiche* sviluppato da *XDev*, *Clever Beans* e *EPOS Game Studios* nel 2017 disponibile per *PlayStation4*. Il gioco è ambientato in un futuro verosimile in cui ciascun giocatore guida *navi* ad *anti-gravità* in grado di usare diverse *armi* e *potenziamenti* al fine di ostacolare gli avversari e tagliare il traguardo prima che questi possano fare altrettanto. I tracciati percorribili sono caratterizzati da ampie carreggiate, curve paraboliche, salti e topologie sinuose. L'ambientazione del gioco è varia ma predilige il contesto urbano: i tracciati attraversano città futuristiche caratterizzate da tinte neutre ma arricchite da cartelloni pubblicitari al neon, segnaletica luminosa e dettagli in materiali traslucidi, i quali vengono risaltati ancor di più nelle modalità notturne (Fig. 2.7). Non mancano inoltre ambienti naturali in cui gareggiare quali montagne, canyon e deserti o circuiti volanti immersi nelle nubi. Il gioco offre un gran numero di *navi*, riccamente personalizzabili attraverso *livree* ed altri dettagli meccanici. Le *navi* sono caratterizzate da forme affusolate ed aerodinamiche, materiali metallici in cui risalta l'abitacolo di vetro e dettagli che ricordano vagamente veicoli da guerra (Fig. 2.8).

2.5 TRON RUN/r

TRON RUN/r è un videogioco sviluppato da *Sanzaru Games* e pubblicato da *Disney Interactive* nel 2015 su *PlayStation4*, *XBoxOne* e *Windows*. Il gioco è un'*avventura dinamica* focalizzata sulle *corse* all'interno del mondo virtuale di *TRON*, siano esse *podistiche* oppure tramite l'uso di una motocicletta virtuale detta *Light Cycle*. Le ambientazioni presentate sono tre. La prima è costituita dalla città vir-



Figura 2.7: Wipeout Omega Collection - Ambientazione



Figura 2.8: Wipeout Omega Collection - Esempio di nave

tuale di *Tron City*, caratterizzata da lunghe strade immerse in elementi geometrici che ricordano vagamente edifici, ponti o tunnel (Fig. 2.9) . I percorsi presentano un gran numero di ostacoli e piattaforme sospese che si materializzano mentre il giocatore vi si avvicina. L'ambiente è caratterizzato dalla massiccia presenza di effetti di luce al neon che fanno da contrasto a geometrie riflettenti e cieli tetri. La seconda ambientazione è nota col termine «*Whiteout*» ed è costituita da un lungo tracciato percorribile tramite la propria *Light Cycle* (Fig. 2.10) . Il tracciato si sviluppa in maniera sinuosa in un ambiente asettico caratterizzato da tonalità neutre molto chiare che quasi soffocano gli onnipresenti effetti al neon. La terza ambientazione, nota col nome «*Outlands*», unisce gli elementi delle altre due in una città virtuale in declino, caratterizzata da ampi ambienti immersi in elementi dalle tonalità cupo.

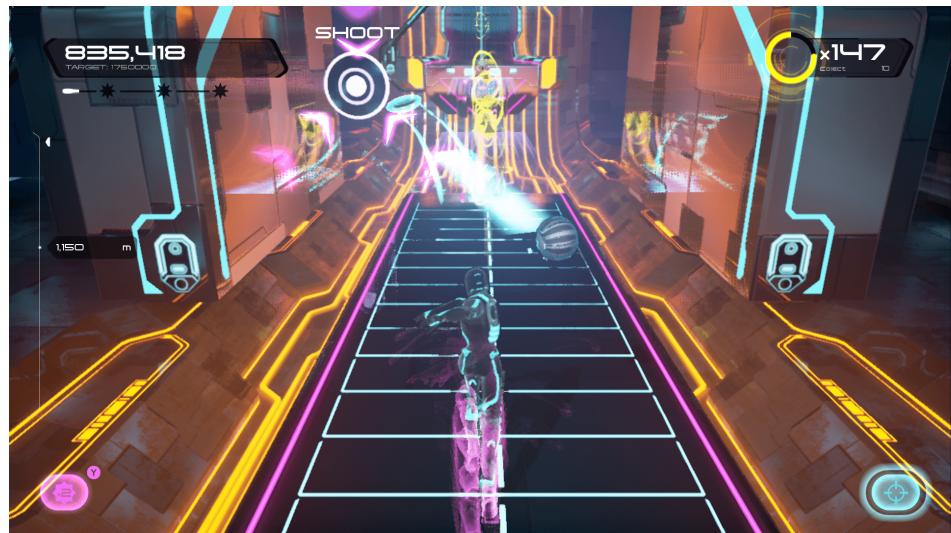


Figura 2.9: TRON RUN/r - Tron City



Figura 2.10: TRON RUN/r - Whiteout

Capitolo 3

Design di OrbTail

Questo capitolo è dedicato al design di OrbTail, partendo dalla definizione delle meccaniche di base e delle modalità di gioco, fino all’analisi dei livelli e la descrizione dettagliata di tutti gli elementi di gioco. I contenuti si concentreranno principalmente sugli aspetti *concettuali* del gioco, in maniera agnostica rispetto alla direzione artistica e dalle soluzioni tecnologiche che verranno adottate (sebbene ne verranno considerate le eventuali limitazioni).

3.1 Meccaniche di base

OrbTail è un gioco competitivo con elementi tipici dei *giochi di corsa* in cui quattro partecipanti si sfidano all’interno di un’arena. Il titolo offre diverse modalità di gioco a tempo caratterizzate da condizioni di vittoria differenti e molteplici arene con un diverso grado di difficoltà. Il titolo è stato sviluppato per le principali piattaforme *desktop* e *mobile* quali Windows, OSX, Android e iOS e consente agli utenti di giocare tra loro indipendentemente dalla piattaforma utilizzata in modalità *cross-platform play*.

In OrbTail ogni giocatore controlla un veicolo in grado di muoversi agilmente in un’arena, collezionando elementi di gioco sferici detti *orb* che si agganciano ad esso formando una lunga coda. È possibile utilizzare il proprio veicolo per scontrarsi contro quelli avversari al fine di sottrarre *orb* in misura proporzionale alla direzione e alla forza d’impatto. Questi elementi sono inizialmente distribuiti lungo la superficie dell’arena e la loro acquisizione costituisce la meccanica principale attorno alla quale ruotano le varie modalità di gioco descritte nei capitoli successivi. Gli *orb* rappresentano la *risorsa limitata* su cui si concentra l’*elemento di sfida* tra i vari giocatori. A seguito di valutazioni empiriche è stato deciso che il numero di *orb* per arena è pari a *ventotto*: tale quantitativo garantisce un buon bilanciamento tra fasi di scontro e fasi di raccolta.

All'interno dell'arena sono inoltre distribuiti degli elementi di gioco rari, detti *core*, che garantiscono al giocatore un potenziamento casuale temporaneo. Lo scopo di questi potenziamenti è quello di aggiungere *profondità* al *gameplay* e scuotere i normali equilibri di gioco.

Per garantire la massima fruizione del videogioco in varie casistiche, tutte le modalità di gioco ed arene sono disponibili in configurazione a *giocatore singolo*, *multigiocatore locale* e, qualora dovesse essere disponibile una connessione ad Internet, *multigiocatore online*. È inoltre possibile giocare in *modalità online mista* in cui più giocatori condividono una stessa postazione e giocano con altri giocatori su altri dispositivi. La configurazione *multigiocatore locale* è gestita tramite *splitscreen* con numero di partecipanti compreso tra due a quattro. Questa configurazione è disponibile per le sole piattaforme *desktop* essendo queste le uniche in grado di supportare più periferiche di input contemporaneamente. I dispositivi *mobile* ne sono esclusi sia per ragioni di performance e sia perché queste piattaforme sono solitamente ottimizzate per l'interazione da parte di un unico utente.

Tutte le modalità di gioco sono state concepite considerando un numero di partecipanti fisso pari a *quattro* e bilanciate di conseguenza: qualora il numero di giocatori dovesse essere inferiore, i veicoli rimanenti verranno scelti e controllati da un'opportuna *intelligenza artificiale*. Questi veicoli sono in grado di eseguire semplici compiti quali collezionare gli elementi di gioco, esplorare l'ambiente e scontrarsi con gli avversari. Per ragioni tecniche il livello di difficoltà dell'*intelligenza artificiale* è unico e non è possibile configurarlo. Nel design originale di gioco l'*intelligenza artificiale* era disponibile nella sola modalità a giocatore singolo: era pertanto possibile giocare sessioni *online* con un numero inferiore di giocatori. Questa limitazione è stata infine *rimossa* al fine di garantire una più elevata consistenza tra le esperienze a giocatore singolo e multigiocatore e, allo stesso tempo, aumentare la difficoltà delle sessioni online.

Le meccaniche di base sono pensate per essere semplici, al fine di adattarsi ad un vasto pubblico di giocatori su un gran numero di piattaforme differenti e bilanciate in modo da non avvantaggiare nessuna di queste durante le sessioni *cross-platform play*. La durata ridotta delle sessioni di gioco, solitamente inferiore ai *tre minuti*, unita al gran numero di configurazioni disponibili, garantisce una *versatilità* del prodotto unica: il gioco è tanto adatto al *casual gamer* in cerca di un'esperienza a giocatore singolo, quanto ad un gruppo di amici in cerca di un *party-game* rapido e competitivo.

3.2 Modalità di gioco

Il titolo offre diverse modalità di gioco accomunate dalle medesime meccaniche di base ma caratterizzate da condizioni di vittoria differenti.

Arcade In questa modalità l’obiettivo è quello di fare più punti possibile entro un tempo limite prestabilito di *centoventi secondi*. Ogni *orb* collezionato aumenta di *dieci* il punteggio del giocatore. Questa modalità di gioco è pensata per favorire scontri trai veicoli dei partecipanti ed è caratterizzata da un *ritmo* frenetico per l’intera durata della sfida. Il numero limitato di *orb* porta i giocatori allo scontro continuo in modo che ve ne sia sempre un discreto numero da collezionare e garantire un flusso costante di punti. In questa modalità gli scontri trai veicoli possono risultare tanto positivi per l’attaccante quanto per il difensore: una volta collezionati tutti gli *orb* a disposizione, l’unico modo per fare più punti è liberarne di nuovi, anche a discapito dei propri. Questa meccanica consente di mantenere un elevato grado di sfida in quanto, al termine di ciascuno scontro, ogni giocatore dovrà fare del suo meglio per raccogliere gli *orb* staccati prima che gli avversari possano fare altrettanto. Nelle fasi finali i giocatori dovranno inoltre essere più cauti, valutando gli esiti di ciascun scontro ed evitandone di inutili in caso di vantaggio.

Coda-più-lunga In questa modalità l’obiettivo è quello di terminare la partita col maggior numero di *orb* attaccati alla propria coda. Qualora non dovesse essere possibile determinare un vincitore unico al termine dei *centoventi secondi*, la partita termina in *parità*. Questa modalità è caratterizzata da un *ritmo* crescente che culmina in una fase finale particolarmente *frenetica* ed *imprevedibile*. L’esito della partita determinato solo dalle condizioni finali di gioco, favorisce la nascita di *alleanze implicite* e *mutevoli* trai partecipanti e finalizzate ad intralciare il giocatore attualmente in vantaggio ed impedire che questi vinca. Secondo questa dinamica di gioco, il giocatore con la coda più lunga sarà portato a *fuggire* dagli avversari, evitando di perdere il vantaggio, laddove tutti gli altri saranno portati a intraprendere azioni più spericolate, sperando di ribaltare l’esito della partita. Questi due *ruoli* cambiano rapidamente nel tempo, specialmente nelle fasi finali in cui sarà richiesta un’elevata *concentrazione* e *reattività*.

Eliminazione L’obiettivo di questa modalità è *eliminare* tutti gli altri partecipanti prima che questi possano fare altrettanto. All’inizio della partita gli *orb* presenti nel livello vengono equamente ripartiti fra tutti i giocatori, formandone la *coda* iniziale. Un giocatore rimane in partita fintanto che la sua coda contiene almeno un elemento, altrimenti viene *eliminato*. Questa modalità è caratterizzata da un elevato grado di *competizione* e un livello di difficoltà che cresce al diminuire del numero di partecipanti in gara. Ogni volta che un giocatore viene eliminato, i veicoli rimanenti potranno spartirsi i suoi *orb*, aumentando le loro possibilità di sopravvivenza. Come per la precedente modalità, i partecipanti saranno portati a concentrare i propri sforzi per eliminare il giocatore

col maggior numero di *orb* e ciò consente di bilanciare il *gameplay* in presenza di giocatori particolarmente forti. I giocatori eliminati possono assistere al resto della sfida attraverso un’opportuna modalità *spettatore* che consente loro di inquadrare i partecipanti ancora in gioco. Questa modalità ha inoltre un tempo limite di *centottanta secondi*, scaduti i quali la partita termina in parità. Questo limite temporale è necessario per incentivare lo scontro tra i partecipanti rimanenti, impedendo che la partita si protragga indefinitamente. La durata è più elevata rispetto alle modalità precedenti in quanto si vuole aumentare la probabilità che la partita finisca a causa dell’eliminazione dei partecipanti invece che per esaurimento del tempo di gara.

3.3 Veicoli

Il giocatore ha a disposizione *sei* veicoli caratterizzati da aspetto e stili di guida diversi. I veicoli possono *accelerare*, *frenare* e *sterzare*; l’uso prolungato del freno consente al veicolo di *accelerare* in *retromarcia*. I diversi stili di guida consentono ai veicoli di adattarsi a diverse tipologie di giocatori o strategie e sono determinati da tre *parametri* fondamentali:

Velocità Rappresenta la massima velocità raggiungibile dal veicolo, influenzandone il *potenziale offensivo*, *difensivo* e *tattico*. Un valore elevato di *velocità* consente di staccare più *orb* dagli avversari a seguito di uno scontro, raggiungere più velocemente obiettivi di gioco e allontanarsi rapidamente dai veicoli avversari vanificando eventuali tentativi offensivi.

Accelerazione Rappresenta la massima accelerazione del veicolo. Un valore più elevato consente al veicolo di raggiungere velocità maggiori sulle brevi distanze, causando danni maggiori in ambienti ristretti e minimizzando le circostanze in cui il veicolo si trova fermo e quindi facilmente bersagliabile. Questo parametro ha valenza principalmente *tattica* e, più limitatamente, *offensiva*.

Manovrabilità Rappresenta la massima velocità a cui il veicolo può sterzare. Veicoli caratterizzati da elevata *manovrabilità* sono facilitati nel cambio di direzione e pertanto sono più efficienti durante le fasi di raccolta di elementi di gioco quali *orb*. La possibilità di cambiare rapidamente traiettoria consente inoltre una maggiore precisione durante gli scontri o le schivate.

Il design originale di OrbTail prevedeva due parametri aggiuntivi, *attacco* e *difesa*, i quali influivano direttamente sull’esito degli scontri e il numero di *orb* staccati. Dopo varie iterazioni è stato deciso di rimuovere questi due parametri in quanto il loro contributo risultava impercettibile durante le sessioni e perché l’interazione con il parametro *velocità* rendeva difficile differenziare e bilanciare i vari veicoli. Questi

due parametri sono utilizzati internamente per determinare l'esito degli scontri tra i veicoli ma sono configurati allo stesso modo su ciascuno di essi.

Di seguito sono riportate le descrizioni dettagliate di tutti i veicoli disponibili. I parametri sono stati configurati in maniera *euristica* al fine di esaltare diversi stili di gioco e bilanciati in modo da non avvantaggiare nessun veicolo (Tab. 3.1) .

Tabella 3.1: Parametri dei veicoli

	Parametro		
Veicolo	Velocità	Accelerazione	Manovrabilità
Flare	• • • • •	• • • • •	• • • • • •
Flash	• • • • • •	• • •	• •
Glow	• • •	• • • •	• • • • • • •
Radiance	• • • •	• • • • • •	• • •
Shine	• • • • •	• • • • •	• • •
Sparkle	• • •	• • • • •	• • • • • •

Flare Caratterizzato da uno stile di guida *bilanciato*, questo veicolo è pensato per adattarsi ad ogni tipo di strategia. Indicato per i nuovi giocatori.

Flash Dotato di un' *elevata velocità*, questo veicolo esprime tutto il suo *potenziale offensivo* sulla lunga distanza sacrificando gran parte della manovrabilità. Particolarmenete adatto per i giocatori più esperti che prediligono strategie ad alto rischio ed alto guadagno.

Glow Questo veicolo è dotato di un'elevata mobilità e ciò lo rende particolarmente adatto per collezionare *orb* e *core*. La bassa velocità è compensata dalla facilità di accesso ai potenziamenti e ciò lo rende adatto ai giocatori che prediligono uno stile di gioco più strategico.

Radiance L'elevata *accelerazione* e *velocità* di questo veicolo lo rendono particolarmente efficace in *mischia*. Adatto per strategie aggressive sulle medie e brevi distanze.

Shine Veicolo bilanciato, sacrifica parte della manovrabilità per un più elevato *potenziale offensivo*. Adatto a giocatori intermedi.

Sparkle Veicolo ottimizzato per la *mobilità* e caratterizzato dall'elevato potenziale *difensivo*. La reattività di questo veicolo lo rendono perfetto per *schivare* tentativi offensivi da parte degli avversari.

Al fine di facilitare il riconoscimento dei partecipanti in gara, ciascun veicolo prevede quattro differenti *livree*, caratterizzate da un aspetto e colore diverso. La livrea associata al veicolo viene automaticamente selezionata dal sistema in funzione dell'indice del giocatore all'interno della partita: il primo giocatore avrà una *livrea rossa*, il secondo una *blu* e gli altri due rispettivamente una *livrea verde* e una *gialla*.

3.4 Livelli

Il titolo offre *tre* livelli totali, caratterizzati da diverso aspetto e *topologia*. Le arene si sviluppano in solidi tridimensionali: i veicoli possono muoversi solamente lungo loro superficie ma non possono mai passarvi attraverso o volare al loro interno. Tutti gli elementi di gioco *fluttuano* al di sopra della superficie dell'arena ad una breve distanza. Sebbene le arene si differenzino per dimensione, l'ambiente di gioco rimane sempre ragionevolmente limitato. Le motivazioni di questa scelta di design sono molteplici. In primo luogo la durata limitata delle partite richiede che i giocatori debbano interagire tra di loro più frequentemente possibile nel breve tempo a disposizione, si vuole evitare quindi il dover attraversare grandi distanze prima di potersi scontrare. In secondo luogo si vogliono evitare strategie particolarmente *elusive* in cui un giocatore continua a fuggire dagli avversari senza che questi possano raggiungerlo o *accerchiarlo*. Questa proprietà è fondamentale per garantire un grado di sfida adeguato nelle modalità *coda-più-lunga* ed *eliminazione*. Originariamente era inoltre prevista la presenza di elementi attivi ed ostacoli all'interno dei livelli: sebbene questa funzionalità avrebbe certamente arricchito l'esperienza di gioco, ciò avrebbe richiesto lo sviluppo di un'*intelligenza artificiale* più sofisticata. Questa motivazione, unita ai tempi di sviluppo ridotti ha portato alla rimozione dei suddetti.

Di seguito sono riportate le descrizioni delle arene disponibili.

Stadium Il livello è costituito da un'arena *emisferica* in cui i veicoli possono correre sulla sua base circolare ma non possono superarne i bordi. Questo livello è caratterizzato da una topologia *piana* e pertanto tutti gli elementi di gioco sono sempre ben visibili. Questa arena è adatta per i nuovi giocatori oppure per gli utenti che preferiscono concentrarsi sull'azione di gioco evitando la difficoltà introdotta da topologie diverse.

Planetarium Questa arena è ricavata all'interno di una *sfera* ed è caratterizzata da una forza di gravità radiale che spinge i veicoli e tutti gli elementi di gioco sulla *superficie interna* della stessa. La topologia di questa arena richiede un livello di abilità maggiore da parte dei partecipanti, in quanto la curvatura limita la visibilità degli elementi di gioco. La grande estensione superficiale

del livello rende inoltre più complicato individuare la posizione degli avversari e capirne le traiettorie.

Torus Arena caratterizzata da una topologia *toroidale* e campo gravitazionale ad anello in cui gli elementi di gioco sono distribuiti lungo la *superficie interna*. Questo livello costituisce il massimo grado di sfida offerto dal gioco. La topologia complicata rende particolarmente arduo valutare correttamente le distanze degli oggetti e prevedere le traiettorie degli avversari. La guida dei veicoli all'interno di quest'arena richiede inoltre un più elevato grado di precisione in quanto non è sempre semplice individuare il percorso migliore per raggiungere un determinato obiettivo.

Durante lo sviluppo sono state inoltre valutate topologie *convesse* (come ad esempio la superficie esterna di una *sfera* o di un *toro*), tuttavia a parità di dimensione con la controparte *concava*, il raggio di curvatura risultava tale da ridurre drasticamente il numero di elementi di gioco a schermo e rendere ancora più complicato individuare avversari ed oggetti. A seguito di queste considerazioni è stato deciso di evitare l'aggiunta di suddette topologie.

3.5 Potenziamenti

All'interno delle arene sono distribuiti degli speciali elementi di gioco detti *core* che, una volta raccolti, garantiscono al giocatore un *potenziamento* o un'arma casuale attivabile a comando. Questi elementi di gioco sono stati introdotti al fine di smuovere l'*equilibrio* della partita, introducendo una maggiore profondità e varietà al *gameplay*. Il *sapiente* uso di questi potenziamenti al momento giusto può ribaltare completamente le sorti di una partita o consolidare la propria posizione di vantaggio. Per evitare eventuali abusi i potenziamenti *non sono cumulabili* e vengono *consumati* una volta utilizzati.

In una prima iterazione, questi potenziamenti erano ottenibili raccogliendo degli *orb* che venivano «imbevuti» casualmente una volta ogni *dieci* secondi e iniziavano a brillare vistosamente. Durante lo sviluppo ci si è resi conto che sfruttare gli *orb* stessi per fornire potenziamenti impediva ai giocatori di opporsi agli avversari in posizione di estremo vantaggio ed in generale rendeva molto difficile ottenerne di nuovi: basta considerare che durante la partita il numero di *orb* non collezionati è sempre molto basso (e quindi nel caso generico non ne venivano «imbevuti» abbastanza). Per questo motivo è stato deciso di introdurre un elemento di gioco collezionabile apposito, il *core*, il cui numero per arena è fissato a *quattro*. A differenza degli *orb*, questi elementi scompaiono una volta collezionati e vengono rigenerati nello stesso punto dopo un breve periodo di *cinque secondi*.

Di seguito sono riportati i potenziamenti ottenibili attraverso i *core*.

Turbo Una volta attivato, questo potenziamento imprime una forza costante che spinge il veicolo in avanti per *tre* secondi a grande velocità. Questo potenziamento è pensato per essere utilizzato in ambito *offensivo* per aumentare il numero di *orb* staccati o *difensivo*, quando è necessario distanziare gli avversari rapidamente.

Gravità Una volta attivato, permette al veicolo di attrarre a se tutti gli *orb* nel raggio di *tre* metri per *sette* secondi. Grazie a questo effetto è possibile raccogliere un gran numero di *orb* in poco tempo e ciò rende questo potenziamento uno dei più apprezzati e versatili.

Dirrottamento Il veicolo lascia cadere un elemento di gioco visivamente identico ad un *orb*. Gli avversari che tentano di raccogliere questo oggetto perdono il controllo del proprio veicolo per *due* secondi. L'uso di questo potere è principalmente difensivo e permette di neutralizzare gli avversari temporaneamente per allontanarsene.

Missile Lancia un missile a ricerca verso l'avversario più vicino. Il missile esplode a contatto oppure automaticamente dopo 5 secondi: i veicoli coinvolti nell'esplosione perdono *due orb*. Quest'arma ha effetti particolarmente devastanti nella modalità *eliminazione* o nelle fasi finali di *coda-più-lunga* in quanto consente di ribaltare le sorti della partita. Il raggio di curvatura limitato di quest'arma consente ai giocatori più abili di schivare il missile, in attesa che questi esploda automaticamente.

Invincibilità Una volta attivato, il veicolo del giocatore non può perdere *orb* per *due* secondi. Sebbene rappresenti il potere col più alto potenziale *difensivo*, in grado di proteggere i veicoli da armi avversarie o scontri, la sua breve durata richiede un'adeguata temporizzazione per ottenerne il massimo effetto.

Sovraccarico Raddoppia la velocità massima del veicolo per *sette* secondi. A differenza del «turbo», garantisce una manovrabilità maggiore al costo di un aumento di velocità inferiore.

Proiettile Il veicolo spara un proiettile in linea retta che esplode a contatto oppure automaticamente dopo *sette* secondi. I veicoli coinvolti nell'esplosione perdono *4 orb*. Quest'arma risulta particolarmente efficace sulle brevi distanze dove vi è una più elevata probabilità di colpire gli avversari. Il gran numero di *orb* staccati consente di ridurre il vantaggio degli avversari oppure di *neutralizzarli* definitivamente qualora dovessero averne in numero esiguo.

Scudo Il veicolo del giocatore è immune da tutte le armi nemiche per *sette* secondi.

A differenza dell'«invincibilità», questo potere garantisce una protezione più estesa da tutte le armi avversarie.

Nel *concept* originale erano previsti alcuni poteri che avevano dagli effetti *negativi* sul veicolo del giocatore. L'intento era quello di introdurre una meccanica simile ad una *scommessa*, secondo la quale il giocatore che collezionava un potenziamento non era sempre sicuro di ottenerne un beneficio. Nella versione finale queste meccaniche sono state rimosse al fine di promuovere ancor di più l'uso di potenziamenti ed in generale perché risultavano ingiustificatamente punitive.

3.6 Interazione utente

OrbTail è caratterizzato da un numero molto limitato di controlli ed azioni che l'utente può effettuare e ciò lo rende facilmente fruibile su tutte le piattaforme supportate.

Per le piattaforme *mobile* l'interazione utente all'interno del menù è affidata al *touch*, a differenza della gara vera e propria in cui, per evitare che l'utente possa coprire gran parte dello schermo con le proprie mani, i controlli sono affidati all'*accelerometro*. L'inclinazione del dispositivo consente al veicolo di *accelerare* oppure di andare in *retromarcia*. Piegando il dispositivo a destra o sinistra è inoltre possibile *sterzare* in misura tanto maggiore quanto più grande l'angolo d'inclinazione. L'attivazione dei potenziamenti e delle armi è invece affidata ad un semplice *tocco* dello schermo in qualsiasi punto.

La versione *desktop*, a differenza di quella *mobile*, deve essere inoltre interagibile da più partecipanti contemporaneamente ed è pertanto previsto il supporto per *tastiere*, *mouse* e fino a *quattro pad*. Tramite *tastiera* è sufficiente usare le *frecce direzionali* per muovere il veicolo e la *barra spaziatrice* per attivare i potenziamenti. Questi tasti sono liberamente configurabili da parte dell'utente. L'interazione tramite *pad* prevede l'uso dei *grilletti* posteriori per accelerare, frenare o attivare la retromarcia, la *leva sinistra* per sterzare e un *tasto* apposito per attivare i potenziamenti. L'assenza di gradi di libertà da parte della tastiera impedisce all'utente di poter guidare con precisione il veicolo e ciò rende il titolo più adatto all'interazione tramite *pad*. È inoltre possibile interagire con tutti gli elementi dell'interfaccia grafica (in menù e in gara) attraverso il *mouse*.



Figura 3.1: Elemento dell'HUD con funzione di bussola

3.7 Interfaccia utente

L'interfaccia utente è pensata per essere poco intrusiva: gli elementi dell'*HUD*¹ sono ridotti al minimo e consentono di avere sempre una buona visione del campo di gioco, ciò è particolarmente importante sui dispositivi *mobile* caratterizzati solitamente da schermi dalle dimensioni ridotte.

I menù all'interno del gioco sono caratterizzati da elementi sempre ben visibili, sono escluse pertanto liste a scorrimento o menù con un gran numero di voci. Questa tipologia di menù è necessaria per facilitare l'interazione tramite *touchscreen*. Laddove ci si aspetta che l'utente faccia più scelte (ad esempio modalità di gioco, arena e veicolo), le singole scelte sono suddivise su più pagine.

La *HUD* all'interno di ogni sessione consiste di un *cronometro* che mostra il tempo rimanente prima della conclusione della partita e altri *quattro* elementi colorati finalizzati a mostrare il punteggio o la lunghezza della coda di ciascun partecipante a seconda della modalità di gioco. Un elemento grafico apposito consente di visualizzare il *potenziamento* o l'*arma* a disposizione del partecipante. Un tasto consentirà al giocatore di abbandonare la partita e ritornare al menù principale. Per evitare pressioni accidentali sarà necessario interagire due volte in rapida successione al fine di confermare l'azione.

Il gioco prevede infine un elemento tridimensionale finalizzato ad aiutare il giocatore ad orientarsi meglio. Questo elemento ha funzione di *bussola* e consente di visualizzare la direzione relativa dei veicoli avversari rispetto a quella del giocatore (Fig. 3.1) .

¹HUD è un acronimo che sta per *head-up display* (letteralmente *display a testa alta*). Questo termine identifica l'interfaccia grafica in sovrapposizione durante le partite.

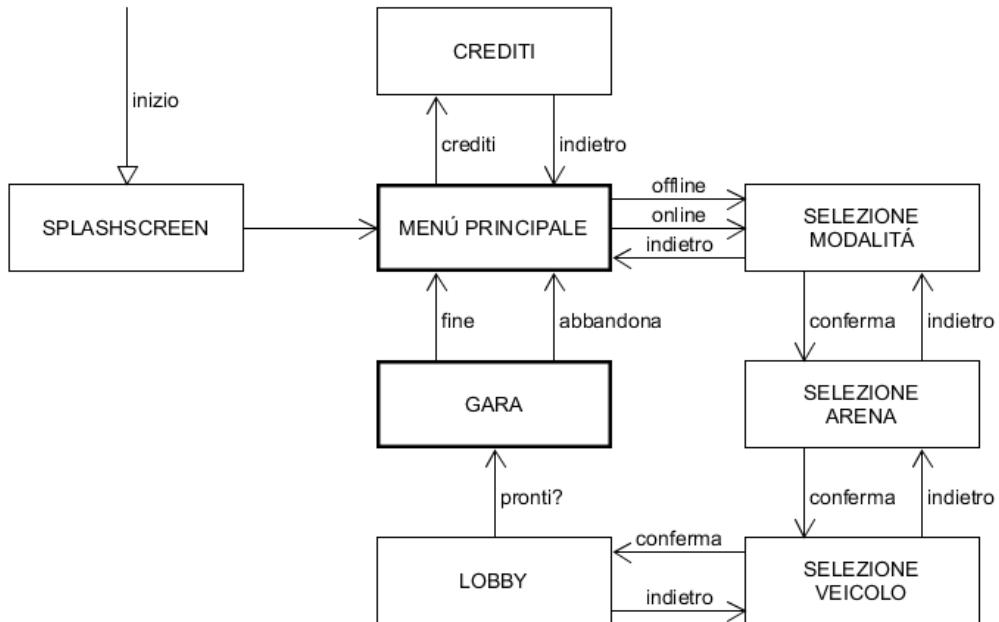


Figura 3.2: Flusso di gioco

3.8 Flusso di gioco

Il flusso di gioco è pensato per portare rapidamente l’utente in partita, riducendo al minimo il numero di interazioni necessarie (Fig. 3.2). Il flusso prevede la selezione della *modalità di gioco*, dell’*arena* e del *veicolo*, dopodiché l’utente sarà portato in un’apposita schermata di *matchmaking*² in cui il sistema provvederà a creare la *lobby*³. La selezione della modalità di gioco e dell’arena è *opzionale*: qualora l’utente non dovesse esprimere alcuna preferenza, il sistema provvederà automaticamente a riempire una delle *lobby* esistenti. Durante la selezione del veicolo sarà possibile aggiungere più giocatori locali alla partita (attraverso un opportuno tasto sul *pad*) ed unirsi ad una *lobby* in gruppo. Qualora non dovesse esserci nessuna *lobby* disponibile, il sistema provvederà a crearne una nuova a cui altri utenti online potranno liberamente accedervi. Questo processo non è necessario per la modalità a *giocatore singolo*.

Affinché la partita possa iniziare, *tutti* i giocatori in *lobby* devono esplicitamente dichiarare di essere *pronti*: il sistema provvederà ad aggiungere *intelligenze artificiali* finché il numero di partecipanti non diventa esattamente *quattro*. Un breve conto alla rovescia porterà i giocatori in gara. All’inizio di ogni partita è prevista una

²Col termine *matchmaking* si intende un particolare servizio offerto dai videogiochi usato per individuare partite online e automaticamente assegnare il giocatore ad una di esse.

³Il termine *lobby* identifica una *stanza virtuale* all’interno di una sessione online. Ogni *lobby* è associata ad una particolare partita in corso o in fase di creazione.

schermata contenente le regole della modalità di gioco selezionata ed i potenziamenti disponibili. Questa schermata funge anche da punto di sincronizzazione, in attesa che tutti gli altri giocatori online abbiano caricato il livello. La partita inizia una volta che tutti i giocatori hanno congedato la schermata.

Gli utenti possono tornare al menù principale una volta terminata la partita oppure *abbandonarla* in qualsiasi momento.

Capitolo 4

Direzione artistica

In questo capitolo si descrive la direzione artistica seguita dal OrbTail, andando a definire l'ambientazione e la resa visiva. Viene inoltre dettagliato il design dei vari elementi di gioco quali arene e veicoli. Le scelte qui presentate considerano i requisiti di design precedentemente illustrati, senza perdere di vista le limitazioni tecniche e temporali a cui sottostare. In generale si è cercato di individuare uno stile semplice e pulito, che potesse risultare estremamente consistente su tutte le piattaforme supportate.

Nel *concept* originale, OrbTail era nato come gioco 2D con camera dall'alto e grafica minimale al neon. Durante il processo di sviluppo questa idea si è evoluta per adeguarsi meglio al design, passando ad una grafica completamente 3D, usando una telecamera ad inseguimento. Questa scelta ha consentito di caratterizzare in modo più profondo le diverse arene e i veicoli contenuti in essa. Lo stile scelto per OrbTail è caratterizzato da elementi fortemente futuristici, in cui elementi *metallici* fanno contrasto ad *intense luci* pulsanti. Nonostante la componente *fantascientifica* del gioco, l'intera ambientazione rimane comunque molto *verosimile*.

Al fine di ottenere una resa visiva di livello più elevato è stato deciso di fare un uso estensivo di tecniche di *post-processing*¹ e *rendering physically-based*. Quest'ultimo consente di ottenere una resa grafica molto fedele alla realtà in quanto si basa sulla descrizione delle proprietà fisiche dei materiali, quali *metallicità* e *ruvidezza*.

4.1 Design dei livelli

I livelli sono caratterizzati da uno stile molto particolare, in cui *effetti di luce* si mescolano a elementi *cyberpunk* quali *gabbie*, *pavimenti in metallo* ed elementi architettonici vagamente *alieni*. I dettagli all'interno di ciascuna arena sono puramente estetici e servono solo per rinforzare la direzione artistica. Gli ambienti sono

¹In *computer grafica*, il processo di *post-processing* consiste nel manipolare un'immagine mediante appositi *filtri*, al fine di migliorarne la resa visiva o il suo valore artistico.

volutamente cupi, in modo da mettere in risalto, tramite effetti di luce, particolari elementi di gioco quali *orb* e *core* oppure dettagli utili per fare orientare il giocatore al loro interno. L'arena «*Stadium*» è caratterizzata da una forma emisferica, in cui le dinamiche di gioco si sviluppano all'interno della base circolare (Fig. 4.1). Il

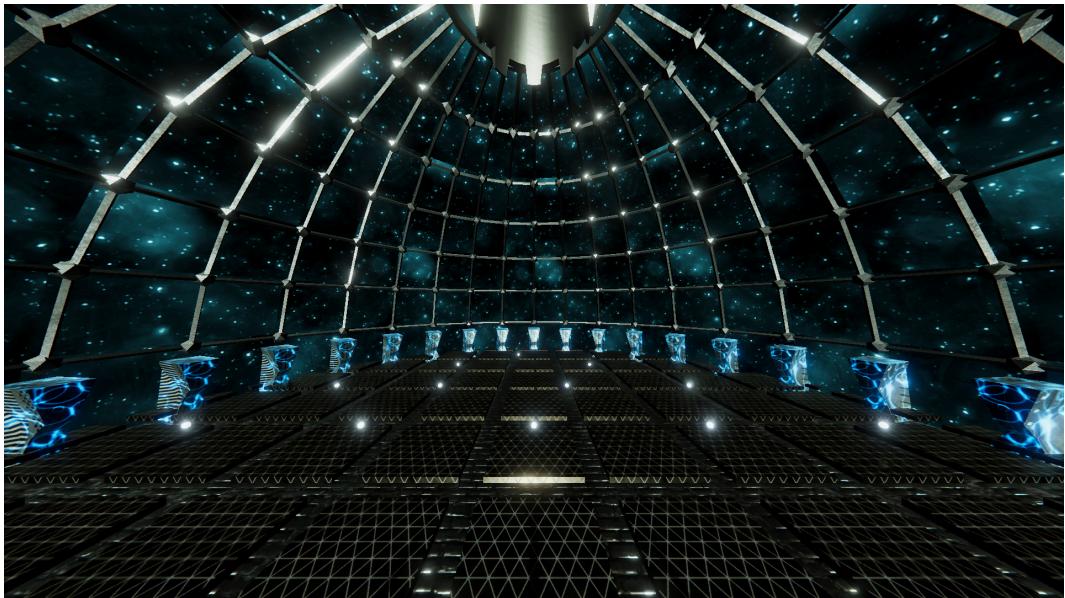


Figura 4.1: OrbTail - Stadium

livello presenta un pavimento costituito da pannelli metallici fluttuanti, circondati da elementi alieni a forma di colonna caratterizzati da striature luminose azzurre. L'arena è racchiusa all'interno di una gabbia metallica oltre la quale si può osservare il cielo stellato. La seconda arena, «*Planetarium*», presenta una geometria sferica ed è caratterizzata da uno stile particolarmente *convoluto*, specie da un punto di vista interno all'arena stessa (Fig. 4.2). L'arena, dotata di notevoli dimensioni, è completamente racchiusa in una sfera costituita da pannelli *tassellati* e rinforzata da una spessa gabbia metallica. Il pavimento è arricchito tramite striature luminose azzurre. L'elemento grafico principale è costituito da una grossa colonna che collega i due poli dell'arena, caratterizzata da uno stile architettonico vagamente antico ma decisamente *extraterrestre*. Questo elemento è l'unico in gioco a costituire un ostacolo per i veicoli dei giocatori ed è pertanto possibile usarlo in maniera strategica per ripararsi dagli avversari. L'ultima arena, «*Torus*», è l'arena geometricamente più semplice (Fig. 4.3). Vista dall'esterno, questo livello ricorda vagamente un acceleratore di particelle in cui le pareti lasciano trasparire un flusso energetico di colore *arancio*. I solchi sulle pareti aiutano il giocatore a determinare la curvatura dell'arena in ogni punto. L'assenza di ulteriori elementi grafici si è rivelata necessaria per ridurre lo sforzo sensoriale richiesto per orientarsi all'interno di questo livello.



Figura 4.2: OrbTail - Planetarium

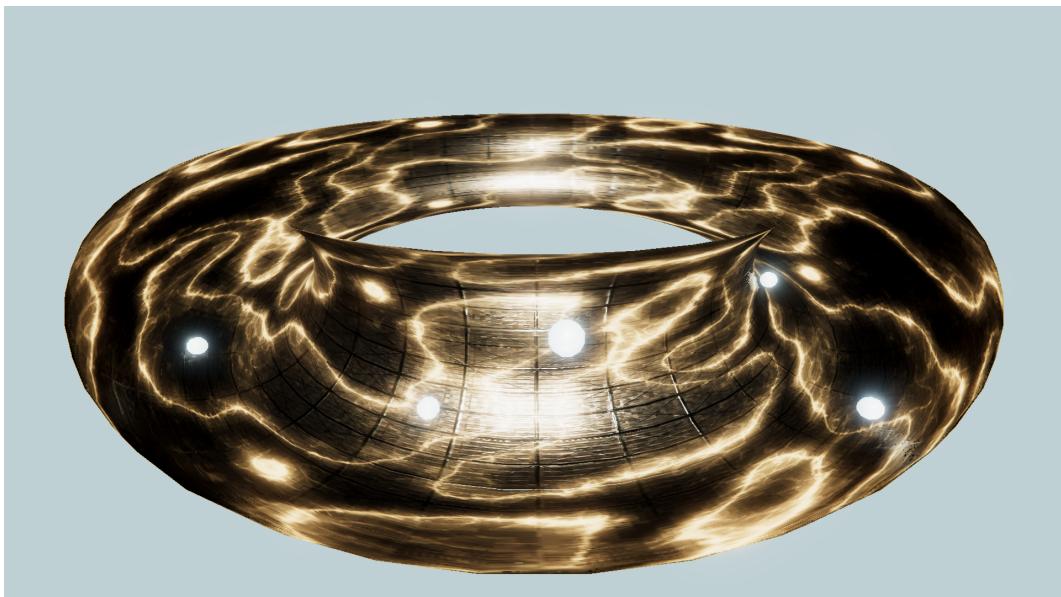


Figura 4.3: OrbTail - Torus

4.2 Design dei veicoli

I veicoli presenti in gioco sono stati reinterpretati in chiave vagamente *sportiva*, assumendo la forma di *navi futuristiche* in grado di fluttuare sulle arene. Le geometrie sono caratterizzate da un *design* molto *verosimile*, con linee morbide e pochi dettagli tecnici (Fig. 4.4). Le livree, sebbene molto minimali, sono arricchite tramite loghi e luci, in modo da far *spiccare notevolmente* i veicoli rispetto all'ambiente circostante (Fig. 4.5). Subdoli dettagli quali leggeri graffi ed ammaccature permet-



Figura 4.4: OrbTail - Design dei veicoli

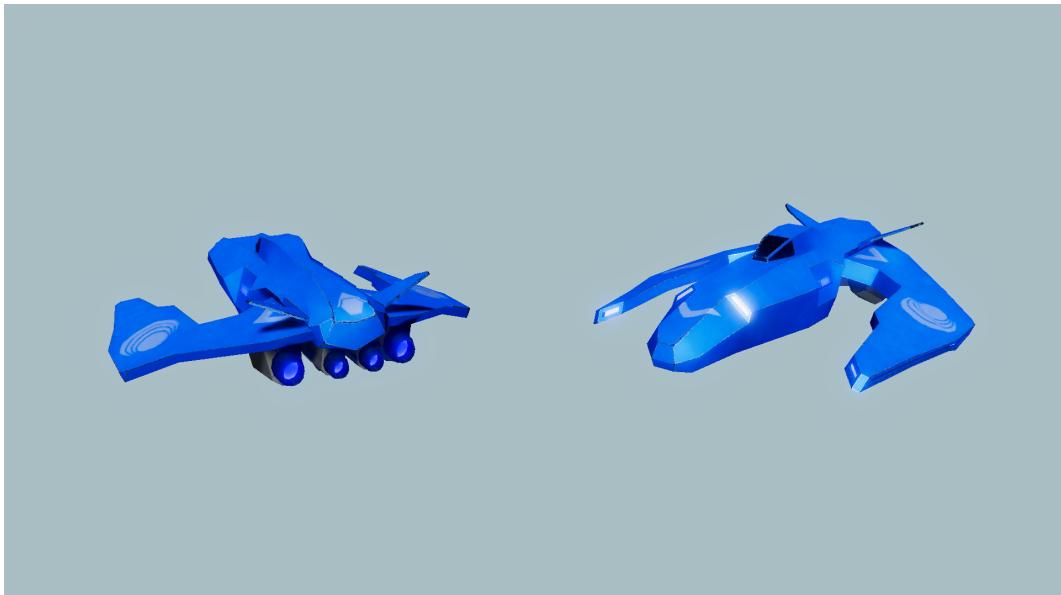


Figura 4.5: OrbTail - Esempio di livrea

tono a ciascuno di essi di mantenere un aspetto pulito, senza tuttavia nascondere la brutalità degli scontri passati. I colori delle livree, brillanti e molto saturi, consentono agli utenti di identificare molto rapidamente gli avversari durante le gare. Il colore di ciascun veicolo è inoltre usato per associare a ciascuno di essi gli elementi della *HUD* quali *punteggi* e *bussole* (vedi 3.7). Il design di ciascun veicolo sottolinea inoltre la sua *personalità* e lo stile di guida: veicoli veloci sono caratterizzati da linee estremamente aerodinamiche, quelli dedicati all'esplorazione e alla raccolta presentano invece *grossi motori* e forme più *massicce*.

4.3 Produzione delle risorse grafiche

Le risorse prodotte per OrbTail usano *due o tre texture* al fine di codificare le informazioni fisiche dei materiali degli oggetti. La «*diffuse-map*» è una texture che rappresenta il contributo principale in termini di colore percepito ed è usata per determinare il colore delle livree dei veicoli o degli elementi dell'arena. Per ridurre il numero di *poligoni* degli elementi di scena ed aumentare di conseguenza le performance, è stato deciso di usare la tecnica del *normal-mapping* al fine di aggiungere dettagli su scala medio-piccola a tutti gli elementi delle arene. I veicoli sono stati giudicati sufficientemente dettagliati e, pertanto, sono stati esclusi da questa gestione. Le informazioni relative alle normali sono codificate tramite un'apposita *texture* nota come «*normal-map*». L'ultima *texture* contiene le informazioni necessarie al *rendering physically-based (PBR)*: il primo canale contiene il valore di *ruvidezza* della superficie, dove *zero* rappresenta una superficie liscia ed *uno* una superficie molto ruvida, il secondo canale codifica la *metallicità* dell'oggetto, ovvero quanto questo si comporta similmente ad un *metallo* (valori vicini ad *uno*) o ad un *materiale dielettrico* (valori prossimi allo *zero*). L'ultimo canale è invece dedicato al valore di *emissività*, ovvero quanta luce viene *emessa* dall'oggetto stesso. Questo valore è monocromatico e viene modulato secondo un colore configurato opportunamente dallo *shader* al fine di ottenere colori emissivi diversi. Il valore di emissività è usato per aggiungere la maggior parte degli effetti luminosi in scena e alcuni dettagli dei veicoli, quali motori e spie.

Capitolo 5

Sviluppo

Per la realizzazione di OrbTail ci si è serviti di un metodo di sviluppo a fasi. La fase preliminare si è concentrata sulla definizione delle tecnologie impiegate: l'attenta scelta di quest'ultime è di cruciale importanza per minimizzare l'impatto sul processo produttivo in caso di criticità o limitazioni, specie nelle fasi più avanzate del progetto. Al fine di mantenere lo storico dei file di progetto e controllare lo sviluppo delle varie funzionalità ci si è affidati a *git*, un software per il *controllo di versione* distribuito.

Un altro aspetto fondamentale riguarda la scelta del *motore grafico*. Fin da subito ci si è resi conto che sviluppare senza usare motori grafici di terze parti avrebbe aumentato esponenzialmente i tempi di sviluppo, rendendo di fatto impossibile supportare tutte le piattaforme desiderate. Sebbene ad oggi esistono diversi motori grafici gratuiti in grado di soddisfare la maggior parte delle esigenze, per lo sviluppo di OrbTail sono state valutate due sole alternative: *Unreal Engine 4* e *Unity 2017*. Sebbene il primo di questi avrebbe garantito una resa visiva eccellente, il suo uso è stato rapidamente accantonato per via delle iterazioni di sviluppo incredibilmente lente, la mancanza di servizi di *matchmaking* e, soprattutto, delle criticità riguardanti lo sviluppo su piattaforme *mobile* (specie in termini di prestazioni). Unity, d'altro canto, è sembrato fin da subito adatto agli scopi del progetto, specie per quanto riguarda il supporto dei dispositivi *mobile*, ed in generale per la sua grande intuitività. La presenza di un servizio di *matchmaking* integrato, in grado di gestire tutte le piattaforme contemporaneamente, ne ha consolidato l'adozione.

5.1 Unity

Unity è un motore grafico sviluppato da *Unity Technologies* che consente di sviluppare giochi multipiattaforma su tutte le piattaforme *mobile*, *console* e *desktop* [3]. La presenza di una versione gratuita, unita ad un elevato grado di intuitività, ne ha garantito la rapida affermazione da parte di sviluppatori *indipendenti* e non. Questo motore è caratterizzato da paradigmi di sviluppo molto chiari, il che lo rende

tanto adatto agli sviluppatori alla prime armi quanto ai più esperti. Il *pattern architettonico* principale, rappresentato dall'*'entity-component'*, consiste nell'implementare funzionalità autoconclusive all'interno di *componenti* indipendenti ed usare i *game object* (termine usato da Unity per descrivere le *entità*) come aggregatori di quest'ultimi al fine di modellare comportamenti più complessi. Questo paradigma è stato migliorato nella versione del motore del 2018, introducendo il concetto di *system*. Secondo questa variante, i componenti espongono solo dei dati e le logiche sono implementate all'interno dei *sistemi*, garantendo un disaccoppiamento ancora più forte tra le varie componenti del gioco. Unity consente di implementare le logiche di gioco tramite script *C#* o *Javascript* e *shader* personalizzati tramite il linguaggio CG. L'assenza completa di codice nativo, ad eccezione del *core*¹, rende le iterazioni di sviluppo incredibilmente veloci in quanto non esiste un processo apposito di compilazione.

La limitazione maggiore di questo motore risiede nel fatto che il suo *core* è completamente *closed-source* (a meno di non pagare per ottenerne l'accesso) e ciò impedisce agli sviluppatori di poterne analizzare il flusso di esecuzione, rendendo particolarmente difficile il processo di *debugging*. Questa limitazione è stata mitigata durante il primo trimestre del 2018, quando Unity Technologies ne ha reso pubblico il codice sorgente C# su *bitbucket* [4]. Nonostante il rilascio del codice sia un notevole passo avanti, permane ancora l'impossibilità di modificare il codice del motore per adattarlo meglio alle proprie esigenze e ciò richiede talvolta l'impiego di soluzioni temporanee o alternative.

Per lo sviluppo di OrbTail sono state usate prevalentemente funzionalità di base legate al 3D, al motore fisico, alla gestione delle scene e delle funzionalità di rete. L'uso di funzionalità generiche ha permesso di evitare tutte le problematiche legate ai sistemi più specifici, quali gestione del 2D e delle *nav mesh*², le quali si sono talvolta dimostrate inaffidabili, limitate o afflitte da bug nascosti. L'intero codice sorgente del gioco è stato sviluppato in C#, utilizzando il paradigma *entity-component*. L'integrazione del pattern *entity-component-system* è stata evitata in quanto questi risultava ancora in fase sperimentale e per via del fatto che lo sviluppo del gioco era già in fase avanzata e si voleva evitare di correre *rischi* inutili.

5.1.1 Plug-in

Unity mette a disposizione un gran quantitativo di *plug-in* esterni al fine di aumentare la resa dei giochi, semplificare il processo di sviluppo o aggiungere nuove

¹Il *core* di un motore grafico è il modulo software di più basso livello. Esso si occupa di astrarre le funzionalità specifiche di ciascuna piattaforma e di dettare il flusso di esecuzione di tutti gli altri sottosistemi, quali il renderer, il motore fisico, la logica di gioco e la gestione della memoria.

²Le *nav-mesh* (in italiano «mesh di navigazione») sono delle superfici poligonali solitamente usate dal sistema di *intelligenza artificiale* per determinare il percorso ottimo (o subottimo) tra due punti nello spazio.

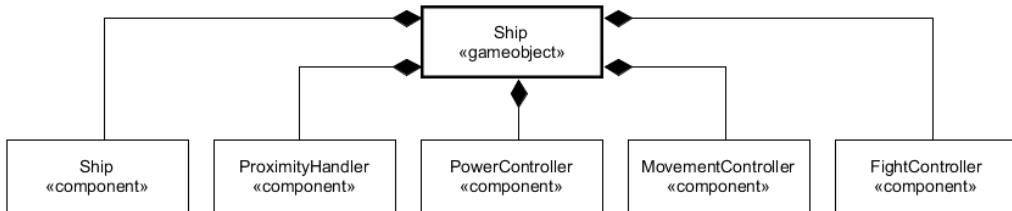


Figura 5.1: Struttura dei veicoli

funzionalità. Per lo sviluppo di OrbTail è stato deciso di utilizzarne uno solo dal nome *iTween* [1]. Questo plug-in gratuito è usato per gestire in maniera semplice ed automatica il processo di *tweening*, ovvero l’interpolazione automatica di valori in un certo periodo di tempo attraverso il sistema di *coroutine* di C#. Il plug-in è stato utilizzato principalmente per aggiungere animazioni agli elementi dell’interfaccia grafica ed in misura molto minore per gli elementi di gioco 3D. L’uso di questo sistema ha permesso di risparmiare un notevole quantitativo di tempo, garantendo una resa visiva molto buona. Quest’ultimo si presenta come un unico file monolitico che può essere integrato nel progetto e non richiede alcuna forma di configurazione.

5.2 Meccaniche di base

La prima fase dello sviluppo di OrbTail si concentra sulla definizione dell’architettura generale del gioco ed in particolar modo delle meccaniche di base. Quest’ultime sono condivise tra tutte le modalità e livelli e pertanto sono state pensate per essere completamente indipendenti da essi. Tra le meccaniche di base troviamo la gestione della gravità, del sistema di controllo dei veicoli, degli oggetti collezionabili, degli scontri e dei potenziamenti (Fig. 5.1) .

5.2.1 Gestione della gravità

I livelli offerti dal gioco sono caratterizzati da topologie profondamente diverse, il che rende impossibile utilizzare la gravità automatica fornita dal motore fisico di Unity per far fluttuare gli oggetti su di essi. Durante una prima iterazione ci si è affidati al sistema di *raycasting*³ al fine di individuare un punto sulla superficie del livello da usare come base per il calcolo della gravità, usando l’asse longitudinale di ciascun oggetto come direzione. Sebbene l’uso di questa tecnica permette di scrivere un codice di gestione unico che si adatta a tutte le topologie, determinare la direzione del raggio usando solo l’orientamento dell’oggetto può generare risultati ambigui o imprevedibili. Queste problematiche sono esacerbate da rapidi cambi

³Il *raycasting* è una tecnica che consente di determinare le intersezioni tra un raggio descritto da un punto iniziale ed una direzione ed una o più superfici poligonali.

di direzione a seguito di esplosioni o durante il normale *rollio* degli oggetti lungo la superficie del livello. Per risolvere questa ambiguità ci si è affidati ad una descrizione *analitica* della forza di gravità per ciascuna delle topologie supportate. Il così descritto consente di ottenere una direzione non ambigua in cui effettuare il *raycasting* per ogni punto dello spazio. Una volta individuato il punto sulla superficie, una semplice simulazione di un *moto oscillatorio smorzato* permette di modellare lo stazionamento degli oggetti sull'arena.

La soluzione adottata prevede l'uso di due elementi principali: un componente base *Gravity Field*, da cui derivano le diverse descrizioni dei campi di gravità supportati, e il *Floating Object* che, assegnato agli oggetti di gioco, consente loro di fluttuare sull'arena (Fig.5.2). Il primo di questi viene assegnato all'arena e consente

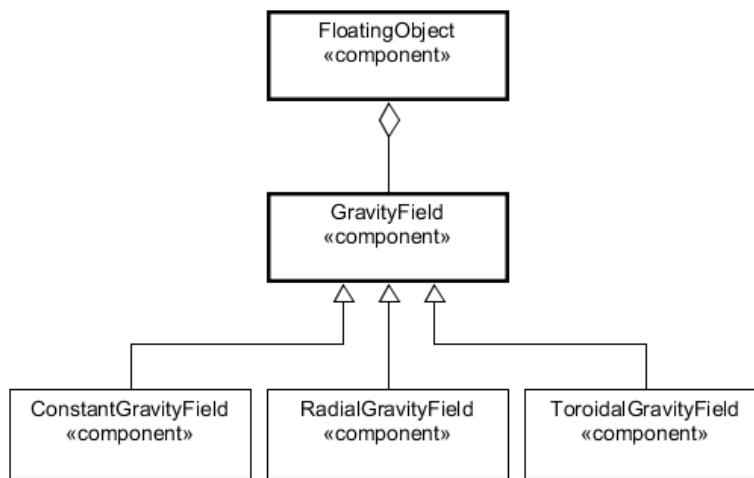


Figura 5.2: Gestione della gravità.

di configurare sia l'intensità della gravità e sia la distanza di stazionamento degli oggetti rispetto alla superficie per impedire che questi possano compenetrafi. Il secondo si occupa di simulare lo stazionamento dell'oggetto a partire dalla direzione della forza di gravità.

5.2.2 Sistema di controllo

Il sistema di controllo di ciascun veicolo si occupa di gestire il movimento di quest'ultimo all'interno dell'arena a partire dalla direzione della forza di gravità e dall'input dell'utente. Il componente *Movement Controller* determina la velocità lineare e quella angolare del veicolo in ogni istante e applica opportune forze al *corpo rigido* di quest'ultimo per causarne il movimento fisico. La gestione del movimento non è simulata in maniera fisicamente accurata, bensì è affidata ad un semplice controllore *PID*. Il sistema adoperato consente di controllare la velocità da applicare $u(t)$ in funzione di quella attuale del veicolo $y(t)$ e di quella desiderata $r(t)$ usando

la sola azione di controllo *proporzionale* con costante K_p . Un controllore analogo è utilizzato per gestire l'azione di sterzo e la velocità angolare risultate. Per questa particolare implementazione le azioni *integrali* e *derivative* non sono state ritenute necessarie:

$$u(t) = K_p(r(t) - y(t))$$

La velocità desiderata $r(t)$ è determinata dal valore massimo di *velocità* del veicolo $r_{max} > 0$ e dall'input dell'utente $r_{in} \in [-1; +1]$. Il termine proporzionale K_e è invece proporzionale al suo parametro di *accelerazione*. L'azione di sterzo segue un principio identico ma ha costanti che dipendono dalla *manovrabilità* del veicolo (vedi 3.3):

$$r(t) = r_{max} \cdot r_{in}$$

La direzione di accelerazione è ricavata tramite il componente *Floating Object* in funzione della tangente del campo di gravità e della rotazione del veicolo, e ciò impedisce che quest'ultimo possa percorrere la superficie del livello passandovi attraverso.

5.2.3 Gestione dei collezionabili

La gestione della collezione degli elementi di gioco è affidata a tre componenti fondamentali (Fig.5.3). Il primo di questi, il *Proximity Handler*, rappresenta un

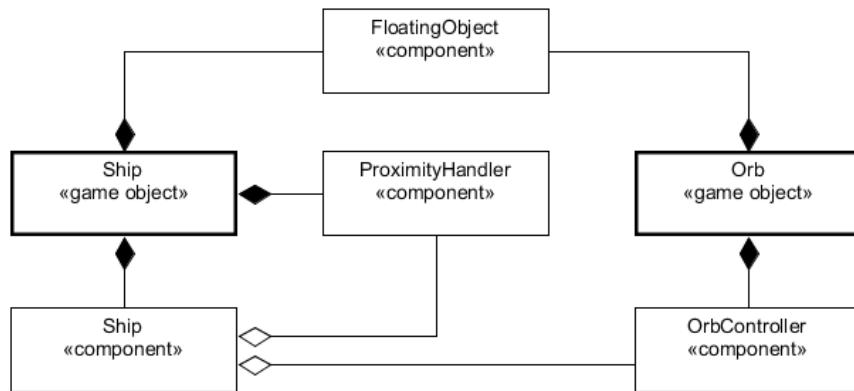


Figura 5.3: Gestione dei collezionabili.

volume di collisione sferico attorno al veicolo il quale lancia un opportuno evento quando un *orb* vi compenetra. Questo componente fa parte del *game object* che rappresenta il veicolo, assieme ad un ulteriore componente *Ship* che si preoccupa di gestire le logiche di base della coda, quali aggiungere o rimuoverne elementi. Il collegamento tra diversi *orb* è gestito attraverso un *vincolo fisico* di tipo molla, con una lunghezza massima tale per cui questi possano muoversi liberamente senza

però allontanarsi da quello che li precede nella coda. Una volta rimossi dalla coda, il vincolo fisico viene rimosso e l'*orb* viene proiettato in una direzione casuale al fine di allontanarlo dal veicolo. Il *game object* che rappresenta ciascun *orb* usa il componente *Floating Object* al fine di fluttuare sull'arena e un altro componente, l'*Orb Controller*, al fine di gestire i vincoli fisici di cui sopra.

5.2.4 Gestione degli scontri

Il sistema di gestione degli scontri tra veicoli è implementato attraverso un componente *Fight Controller*, il cui scopo è quello di rilevare gli impatti e notificare il componente *Ship* del numero di *orb* persi come conseguenza. Le collisioni tra veicoli sono gestite direttamente dal motore fisico di Unity. In una prima versione ciascuno di essi aveva uno o più *collisori fisici* che approssimavano più o meno precisamente la loro topologia, tuttavia questo approccio causava comportamenti inaspettati durante la risoluzione delle collisioni quali veicoli che si incastravano tra loro oppure la generazione di impulsi fisici di entità molto elevata che causavano la perdita di tutti gli *orb* disponibili. Al fine di non avvantaggiare nessun veicolo ed evitare queste problematiche, nell'ultima versione ci si è affidati all'uso di un *collisore sferico* unico che approssima la superficie del veicolo racchiudendone il modello grafico. La forma e la dimensione del collisore è identica per tutti i veicoli e ciò garantisce un'elevata consistenza durante la risoluzione degli scontri. Una volta rilevata una collisione durante quest'ultimi, il motore fisico genera una coppia di eventi, uno per ogni componente coinvolto nell'impatto. Ciascun *Fight Controller* reagisce a questi eventi, determinando il numero di *orb* persi a causa dello scontro e lasciando al *Fight Controller* del veicolo avversario il compito di staccare *orb* dal proprio. Il numero di *orb* persi durante uno scontro dipende dalla direzione d'impatto e dall'orientamento dei veicoli coinvolti: uno scontro frontale deve causare un distacco di *orb* da parte di entrambi i veicoli, laddove colpire un veicolo su una fiancata non deve penalizzare in alcun modo l'aggressore. Sia \vec{p} la posizione del veicolo sui cui viene generato l'evento di collisione, \vec{f} la direzione di quest'ultimo, \vec{c} il punto di impatto tra i veicoli coinvolti e \vec{v}_{rel} la velocità relativa del veicolo rispetto all'avversario, la formula usata per determinare il numero di *danni* inflitti d è la seguente:

$$\vec{i}_d = \frac{\vec{c} - \vec{p}}{|\vec{c} - \vec{p}|}$$

$$i_s = \max(0, \vec{i}_d \cdot \vec{f})$$

$$d = |\vec{v}_{rel}| \cdot i_s \cdot off$$

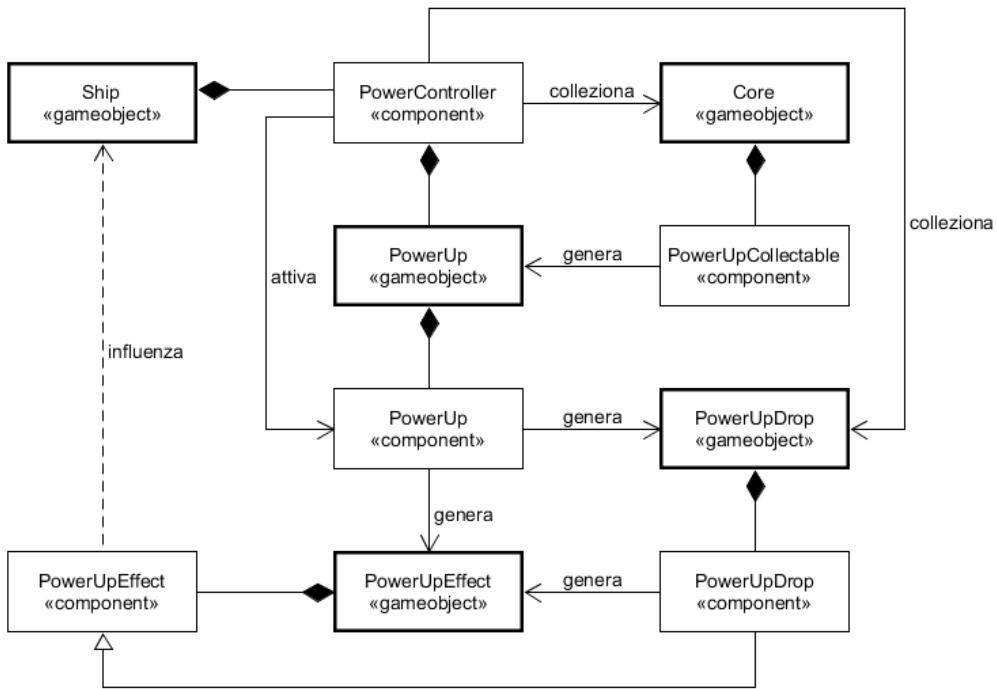


Figura 5.4: Gestione dei potenziamenti

Il termine $\vec{i_d}$ rappresenta la *direzione relativa di impatto*, i_s è invece un fattore di scala che impedisce ai veicoli di causare danni al di fuori di un cono frontale. Il numero di *orb* o_{imp} persi dal veicolo a seguito dell'impatto è calcolato come segue:

$$o_{imp} = \lfloor d \cdot def^{-1} \rfloor$$

I termini *off* e *def* sono due parametri che identificano il potere offensivo e quello difensivo di ciascun veicolo. Un tempo differenziati per veicolo, questi parametri sono oggi usati per rimappare il valore del danno causato e ricavare il numero di *orb* perduti e sono uguali per tutti.

5.2.5 Armi e potenziamenti

La gestione delle armi e dei potenziamenti dei veicoli è affidata a *cinque* componenti diversi, alcuni dei quali sono associati al veicolo come il *Power Controller*, altri ai *core* come il *Power-Up Collectable* ed altri a *game object* che rappresentano gli effetti di tali potenziamenti (Fig. 5.4) .

Il *Power Controller* è un componente che si occupa di *collezionare* i *core* distribuiti sull'arena, in maniera analoga a quanto avviene per gli *orb*. Ogni *core* usa un componente speciale, il *Power-Up Collectable*, al fine di generare un potere casuale da associare al veicolo. Questo componente si occupa inoltre di gestire la parte este-

tica del *core* e delle logiche di attivazione e disattivazione temporizzate. I poteri sono modellati attraverso un componente *Power-Up* che ne contiene varie informazioni di base quali il numero massimo di utilizzi, il tempo minimo tra due utilizzi consecutivi (detto *cooldown*) ed un riferimento ad un componente apposito che ne modella l'effetto una volta attivato: il *Power-Up Effect*. Questo componente è associato ad un oggetto che viene creato in fase di attivazione del *Power-Up* e contiene le informazioni di base dell'effetto, quali la sua durata, il veicolo che lo *possiede* e il veicolo *bersaglio* (se presente). Questa classe è estesa attraverso opportune derivate, una per ogni potenziamento o arma supportata. L'oggetto associato al *Power-Up Effect* contiene tutte le logiche associate al potenziamento e il suo aspetto visivo. Questo oggetto è *gerarchicamente dipendente* dal veicolo per poteri che devono «seguire» graficamente gli spostamenti, quali ad esempio le barriere, oppure può muoversi liberamente come nel caso di missili e proiettili. Una categoria particolare di *Power-Up Effect* è costituita dai *Power-Up Drop*, speciali potenziamenti che, attivati, generano un oggetto di gioco all'interno dell'arena. Questi oggetti assegnano un effetto negativo (derivato dal componente *Power-Up Effect*) ai veicoli che lo collezionano. Questi speciali effetti consentono di modellare potenziamenti quali *trappole* e *mine*. L'uso di questa architettura permette di gestire in maniera molto efficiente un gran numero di potenziamenti. La configurazione di quest'ultimi avviene direttamente in editor e garantisce dei tempi d'iterazione molto rapidi durante il loro bilanciamento.

5.2.6 Gestione degli input

L'architettura alla base del sistema di input consente di gestire l'interazione utente in maniera completamente trasparente rispetto alle periferiche utilizzate e alla piattaforma su cui è eseguito il gioco. Il sistema si basa su un componente *Input Proxy*, di cui ogni veicolo ne possiede un'istanza, il quale astrae tutte le azioni eseguibili dall'utente o dall'*intelligenza artificiale*. Queste azioni comprendono l'*accelerazione*, l'*azione di sterzo*, l'*attivazione dei potenziamenti* e tutti gli input necessari per navigare nel menù. Questo componente è una *façade* usata sia dal *Movement Controller* che dal *Power Controller*, la quale si interfaccia a sua volta con oggetti che descrivono il sistema di input della particolare piattaforma su cui viene eseguita l'applicazione. Questi oggetti implementano un'interfaccia generica *IInput Broker* e comprendono il gestore degli input sulle piattaforme desktop *Desktop Input Broker*, mobile *Mobile Input Broker* e il componente che si occupa di gestire l'intelligenza artificiale *PlayerAI* (Fig. 5.5) . Il *Mobile Input Broker* si occupa di leggere lo stato dell'accelerometro del dispositivo *mobile* al fine di registrare le azioni di *accelerazione* e di *sterzo* e lo stato del touchscreen per l'utilizzo dei potenziamenti. Durante l'inizializzazione di ogni partita, questo componente *calibra* l'input relativo all'accelerometro in modo che si adatti all'inclinazione attuale del dispositivo: questa posizione viene usata come punto di riferimento per calcolare tutte le altre

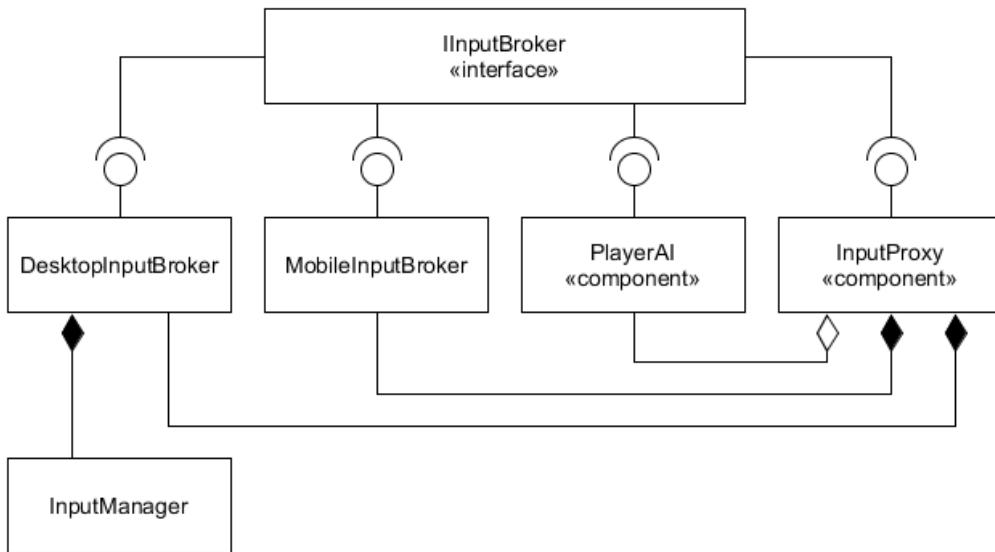


Figura 5.5: Gestione degli input

inclinazioni e di conseguenza le azioni utente. Il *Desktop Input Broker*, invece, si affida a sua volta ad un oggetto *Input Manager* al fine di astrarre le azioni utente dalle periferiche utilizzate, quali *tastiere* e *pad*. L'applicazione prevede un *Input Manager* per ciascun utente locale. Le azioni sono gestite direttamente dal sistema di input di Unity, tuttavia è stato necessario aggiungere un livello di astrazione aggiuntivo in quanto quest'ultimo è stato giudicato inadeguato per le esigenze del progetto. Il sistema di input di Unity consente di registrare più azioni alle quali sono associati in maniera esclusiva fino a due tasti oppure degli assi (come ad esempio gli *stick* dei *pad* o i *grilletti* posteriori), rendendo impossibile associare ad una singola azione più periferiche contemporaneamente. È inoltre impossibile rimappare delle azioni esistenti su *pad* diversi, pertanto è stato necessario duplicare le azioni per ciascuno dei *quattro* pad supportati. L'oggetto *Input Manager* consente di usare l'indice dell'utente al fine di indicizzare le azioni opportune tra quelle registrate al sistema nativo di Unity. Il primo giocatore è inoltre l'unico a poter usare indipendentemente la tastiera o il pad.

5.3 Architettura di rete

L'architettura di gioco di OrbTail è pensata per gestire in maniera uniforme tutte le modalità e configurazioni supportate, evitando di creare classi apposite per gestire le modalità a giocatore singolo, quelle multigiocatore locale e quelle multigiocatore *online*. L'architettura di rete è di tipo *client-server* e uno dei dispositivi dei giocatori è usato come *host* per la partita. Dato il basso numero di giocatori coinvolti, questa

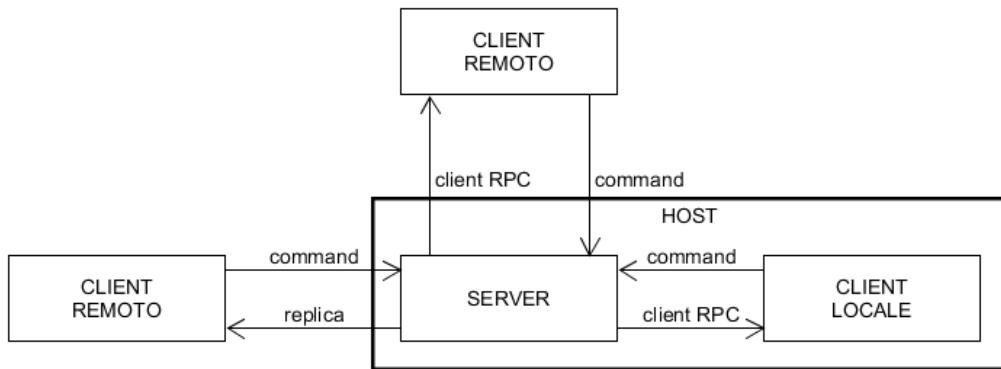


Figura 5.6: Architettura di rete

soluzione è ottimale perché garantisce un buona esperienza *online* evitando i costi associati ad un server dedicato. Il *server* si occupa di gestire tutti gli elementi di gioco quali *orb* e *core* e i veicoli gestiti dalla *IA*, laddove ogni *client* gestirà uno o più veicoli associati ai giocatori locali (in caso di *multigiocatore splitscreen*). Sebbene il dispositivo *host* sia contemporaneamente *server* e *client*, le due parti sono gestite in maniera indipendente. L'*host* in funzione di *client* richiede in ogni caso una connessione al *server* al fine di inviare comandi e riceverne aggiornamenti: il sistema di rete di Unity provvederà a simulare le richieste di rete sulla stessa applicazione in maniera trasparente.

La sincronizzazione degli stati di gioco avviene secondo due possibili meccanismi forniti da Unity: la *replicazione* e le *chiamate remote* (Fig. 5.6). Il primo di questi consente al server di propagare il valore di uno stato di gioco su tutti gli oggetti client costantemente. Questo metodo è particolarmente adatto per valori che cambiano molto rapidamente di cui però non è necessario che vi sia una forte corrispondenza tra la versione sul *server* e quella sul *client*, quali ad esempio la posizione dei veicoli ed il punteggio dei giocatori. Il *server* ha sempre *autorità* sui valori sincronizzati in questo modo e pertanto non è possibile utilizzare questo meccanismo nel senso opposto al fine di inviare dati dal *client* al *server*.

Il secondo meccanismo, noto col termine «remote procedure call» (*RPC*), consente di chiamare un metodo di un oggetto su un'istanza remota di gioco. Unity ne fornisce due varianti, la prima è rappresentata dai *comandi* i quali sono usati per inviare un'azione dal *client* al *server*, la seconda è rappresentata dalle *client RPC* le quali vengono inviate dal *server* e propagate su *tutti* i client. Questo metodo è adatto per eventi di sincronizzazione che avvengono di rado ma per i quali vi deve essere l'assoluta certezza che i *client* ne vengano notificati, come ad esempio gli eventi che determinano l'inizio o la fine di una partita.

Il paradigma fondamentale su cui si basa Unity consiste nell'inviare *comandi* dal *client* al *server*, attendere che questo ne verifichi la validità aggiornando lo stato

di gioco e quindi propagare quest'ultimo tramite *replicazione* oppure tramite *client RPC*. Secondo questo paradigma, ignorando i tempi necessari per inviare i messaggi sulla rete, tutti i *client* osservano sempre lo stesso stato di gioco.

Ad ogni giocatore connesso ad una partita viene associato un particolare *game object* che ne contiene i dati identificativi e ne rappresenta la volontà. Per ragioni di sicurezza è possibile inviare *comandi* al server solo attraverso di esso e solo se il giocatore locale ne ha l'autorità, ovvero se il *game object* in questione rappresenta la sua identità e non quella di un altro giocatore.

5.3.1 Gestione della lobby

Una *lobby* è una *stanza virtuale* a cui altri utenti online possono connettersi. Quest'ultima contiene le informazioni di configurazione della partita prima del suo avvio quali la modalità di gioco e l'arena selezionata. Le *lobby* sono gestite attraverso un *singleton* derivato dalla classe di Unity *Network Lobby Manager*, la quale astrae tutte le logiche di creazione o ricerca di una partita, connessione e disconnessione degli utenti e sincronizzazione della configurazione di gioco. Ogni giocatore connesso ad una *lobby* è associato ad un oggetto di tipo *Lobby Player*, il quale viene usato sia per inviare comandi al *server* e sia per rappresentare i dati del giocatore, quali il suo indice nella partita, il veicolo scelto e le informazioni riguardanti la connessione.

Una volta che l'utente ha deciso la configurazione della partita, il *Lobby Manager* usa i servizi di *matchmaking* di Unity al fine di individuare *lobby* esistenti con configurazione compatibile. Questo servizio indicizza tutte le *lobby* pubbliche, esponendo varie informazioni quali i loro nomi, il numero massimo di giocatori, il numero di utenti attualmente connessi e tutte le informazioni necessarie per connettersi ad essa. Sebbene il sistema consente di associare a ciascuna *lobby* delle *metainformazioni* (come ad esempio la modalità di gioco), un *bug* di Unity impedisce a quest'informazione di essere replicata correttamente, impedendone completamente il suo utilizzo. Per ovviare a questo inconveniente è stato deciso di usare il nome della *lobby* per *codificare* le metainformazioni necessarie in formato CSV⁴. La lista di *lobby* così ottenuta viene filtrata a seconda della configurazione dell'utente ed il sistema provvede a connettersi ad una qualsiasi tra quelle rimaste. Se il tentativo di connessione fallisce (come potrebbe capitare se, nel frattempo, la partita in quella lobby è già iniziata) vengono effettuati tentativi di connessione addizionali sulle altre lobby fino ad un numero massimo configurabile. Se non è stato possibile individuare una *lobby* compatibile, il sistema provvede a crearne una nuova e a registrarla pubblicamente al servizio di *matchmaking* affinché altri giocatori possano connettersi. Nel caso di partite a giocatore singolo le fasi di ricerca delle *lobby* e di registrazione al servizio di *matchmaking* vengono ignorate ma la *lobby* viene comunque creata.

⁴Il formato CSV («comma-separated values») codifica tutti i campi di una struttura in un'unica stringa in cui due campi consecutivi sono separati da un punto-e-virgola.

Ogni volta che un giocatore entra in una *lobby*, il *LobbyManager* crea un oggetto di tipo *Lobby Player* che lo rappresenta, replicando tutte le informazioni salienti agli altri giocatori in partita (quali l'indice del giocatore nella partita ed il veicolo scelto). Un utente può abbandonare una *lobby* in qualsiasi momento.

Per procedere con l'avvio della partita, gli utenti connessi devono esplicitamente inviare un *comando* al server, dichiarando di essere «pronti». Quando l'ultimo di questi *comandi* viene ricevuto, il server *blocca* la partita pubblica (se presente) rimuovendola dal servizio di *matchmaking*, impedendo ad altri giocatori di potersi connettere. A questo punto il *server* genera *Lobby Player* addizionali finché il numero di partecipanti non diventa esattamente *quattro*: questi giocatori sono configurati per essere comandati dal sistema di *intelligenza artificiale*. Dopo un breve conto alla rovescia, un'opportuna *client RPC* notifica i vari client dell'inizio della partita, iniziando il caricamento del livello corretto su tutti i dispositivi.

In una prima versione il sistema di creazione delle *lobby* era esplicito: l'utente poteva decidere se creare un nuovo match oppure se partecipare ad uno esistente. Questo tipo d'interazione è stato rivisto perché giudicato obsoleto: nel caso in cui non vi fosse nessuna *lobby* disponibile, l'utente era costretto a ritornare al menù principale per crearne una e ciò aumentava ingiustificatamente il numero di interazioni necessarie per entrare in partita. Sebbene i flussi di creazione di una partita online ed offline sono identici, la selezione di una o dell'altra modalità deve essere esplicitamente dichiarata dal giocatore all'inizio del flusso di gioco. Per questa iterazione era stato valutato un sistema che avrebbe permesso al giocatore di entrare in partita e lasciare che il sistema decidesse se avviare una partita online o offline (a seconda se vi fosse una connessione Internet o meno), tuttavia, a causa di tempi di sviluppo ridotti, questa funzionalità è stata rimandata ad iterazioni future.

5.3.2 Gestione della partita

La gestione dello stato di gioco è affidata ad un oggetto *Game Mode* il cui tipo, derivato dalla classe *Base Game Mode*, dipende dalla modalità di gioco selezionata (Fig. 5.7) . Questo oggetto descrive le regole di gioco e le condizioni di vittoria, gestendo al contempo il flusso di gara. Tra i suoi parametri di configurazione troviamo la durata massima della partita, il numero di punti guadagnati per *orb* collezionato, il numero di *orb* iniziali e così via. Lo stato di gioco è replicato internamente alla classe e notificato attraverso eventi al quale *osservatori*, quali *HUD* e i veicoli, possono registrarsi e reagire. La classe base *Base Game Mode* contiene tutte le logiche condivise tra le varie modalità, mentre le sue derivate ne possono estendere il comportamento implementando logiche specifiche, come ad esempio il calcolo dei punti e la gestione dell'eliminazione dei partecipanti. Questo oggetto viene creato dal *server* in fase di creazione della *lobby* e replicato su tutti i client e rimane in vita per tutta la durata della partita. La gestione degli eventi di gioco quali l'acquisizione

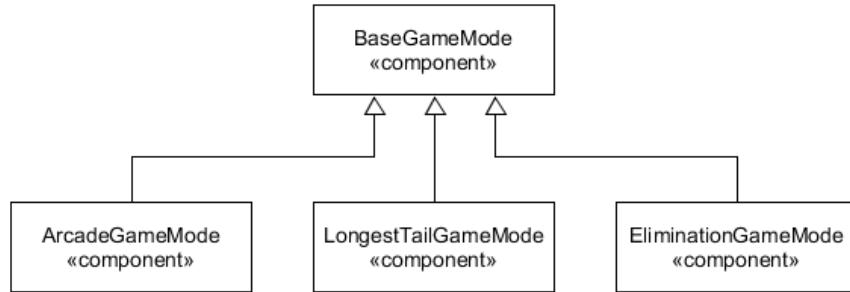


Figura 5.7: Modalità di gioco

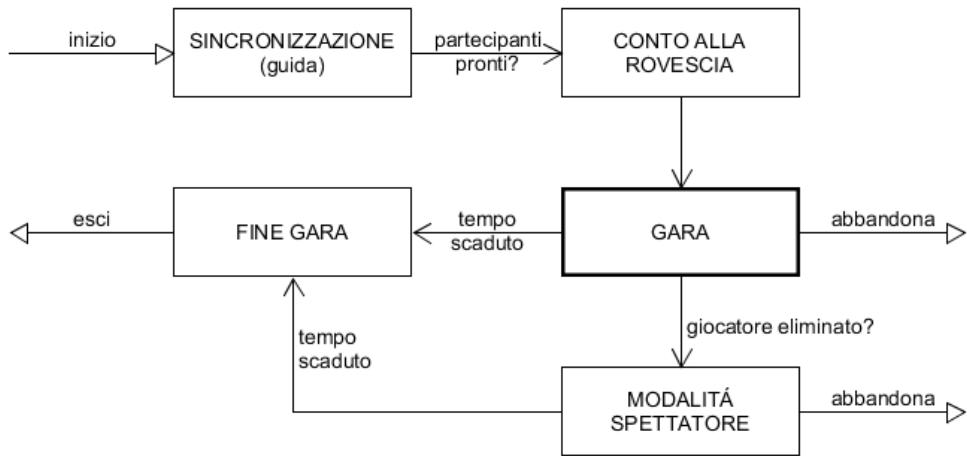


Figura 5.8: Flusso di gara

o la perdita di *orb* e *potenziamenti*, è gestita direttamente dal *server* e i *client* ne subiscono passivamente gli esiti attraverso opportuni eventi propagati tramite *client RPC*. Gli stati di gioco *non critici*, quali tempo rimanente e punteggi, sono invece sia calcolati sul server e *replicati* e sia simulati da tutti i *client* in locale. Questa strategia consente a *client* di osservare sempre una situazione verosimile anche in caso di perdita temporanea della connettività.

Ciascuna partita è strutturata in *quattro* fasi differenti che si susseguono una dopo l'altra fino al termine (Fig. 5.8) . Durante la fase iniziale il *server* mette i giocatori in attesa, fintanto che tutti partecipanti abbiano caricato la mappa. Ogni volta che un dispositivo ha caricato la mappa corretta, il *server* provvede a creare il suo veicolo e a replicarlo: in questo istante il controllo del giocatore viene *trasferito* dal *Lobby Player* al *veicolo*, in modo che quest'ultimo possa essere usato per inviare *comandi* al *server* durante la partita. L'oggetto *Lobby Player* viene preservato in modo da poter contenere informazioni associate al giocatore, quali ad esempio il suo *punteggio*. Sull'istanza locale di gioco, l'oggetto *Base Game Mode* determina la creazione di tutti gli oggetti necessari al giocatore che non devono essere replicati,

quali ad esempio il *controllore della camera* e della *HUD*. Durante questa breve fase di sincronizzazione, il gioco mostra una breve guida che descrive le regole di gioco e tutti i potenziamenti disponibili. Questo stratagemma consente sia di ridurre il tempo di attesa percepito e sia di informare l'utente delle regole specifiche di gioco. Una volta che tutti i partecipanti hanno congedato questa schermata, il *server* procede con la fase successiva di *conto alla rovescia*, lasciando che gli utenti possano prepararsi alla sfida. La fase successiva è quella di gioco e dura fintanto che il cronometro di gara non scende a *zero*. In questa fase il *server* si occupa di gestire gli scontri tra i veicoli e la collezione di oggetti e potenziamenti e l'attivazione di quest'ultimi. Gli eventi associati a queste gestioni sono propagati attraverso delle *client RPC*. La gestione del movimento fa invece eccezione: gli oggetti che rappresentano i veicoli dei giocatori sono creati sui rispettivi *client* i quali ne esercitano l'autorità determinandone la posizione e la gestione degli input. In questo caso il *server* viene usato semplicemente per propagare l'informazione agli altri dispositivi. La modalità di gioco «eliminazione» prevede inoltre uno stato aggiuntivo che viene abilitato quando il giocatore locale viene eliminato dalla partita: in questo stato il giocatore fa da *spettatore*, controllando una camera che può inquadrare i veicoli dei giocatori rimasti in partita e ciclando tra di essi fino a fine partita. In questo stato, l'oggetto usato dal giocatore per inviare comandi al *server* viene rimpiazzato con il *controllore della camera spettatore* e il veicolo viene distrutto. Al termine del tempo massimo di partita, o quando tutti i giocatori sono stati eliminati, il sistema si porta nella fase finale in cui viene dichiarato il vincitore a seconda di condizioni specifiche della modalità di gioco. In questa fase è possibile consultare l'esito della partita e abbandonarla, ritornando al menu principale. L'abbandono di una partita causa la distruzione completa di tutti gli oggetti coinvolti, quali la *lobby*, i *veicoli*, il *Game Mode* e il *Lobby Manager*.

La gestione delle modalità online ed offline è stata uniformata in modo da seguire lo stesso flusso e basarsi sulle stesse classi, evitando gestioni specifiche. Secondo quest'ottica le modalità *offline* sono gestite come fossero delle partite *online* in cui il server è l'applicazione stessa ed in cui vi sono solo giocatori locali e veicoli gestiti dalla *IA*. Una limitazione di questo approccio è data dal fatto che in modalità *online* il motore non supporta la replicazione di oggetti già posizionati in scena e si è pertanto costretti a crearli dinamicamente. Per ovviare a questa limitazione è stato creato un *game object* apposito, detto *Network Spawner*, il quale può essere posizionato direttamente in scena e configurato con un riferimento ad un altro *game object*. Durante la creazione del *server* questo *script* si attiva, causando la generazione del *game object* replicato su tutti i client. Questo oggetto è stato usato principalmente per gestire gli *orb* e i *core* in quanto era necessario che le loro posizioni fossero configurabili manualmente su tutte le arene.

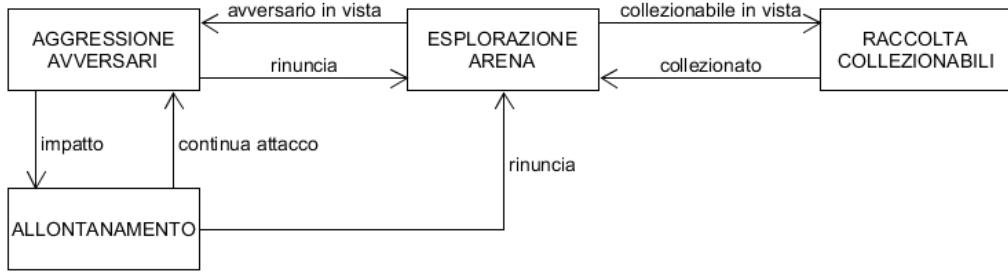


Figura 5.9: Strategie adottate dall'intelligenza artificiale

5.4 Intelligenza artificiale

L'*intelligenza artificiale* di gioco è gestita attraverso un singolo componente *Player AI* che, attivato su un qualsiasi veicolo, ne gestisce direttamente gli input, quali *accelerazione* ed *azione di sterzo* (vedi 5.2.6). Questo componente è generato automaticamente durante l'inizializzazione del componente *Ship* se il giocatore associato ad esso è governato dalla *IA*.

Il sistema di *intelligenza artificiale* è molto semplice e si basa su *tre* strategie diverse che vengono attivate in funzione di eventi di gioco particolari (Fig. 5.9) . Nella forma più semplice, ciascuna strategia si basa sul raggiungimento di un *game object* bersaglio il quale, a seconda della strategia attualmente attiva, corrisponde ad un *orb*, un *core*, un *veicolo avversario* oppure un *punto di controllo*. Una volta determinato l'oggetto da raggiungere, il sistema usa la direzione relativa \vec{d}_{rel} di quest'ultimo rispetto al veicolo per calcolare i valori di *accelerazione* a_i e di *azione di sterzo* s_i :

$$\vec{d}_{rel} = \vec{p}_b - \vec{p}_v$$

$$s_i = \vec{u} \cdot (\vec{f} \times \vec{d}_{rel}) \cdot K_s$$

$$a_i = 1 - |s_i|$$

dove \vec{u} e \vec{f} sono i versori che rappresentano rispettivamente la direzione longitudinale e quella sagittale del veicolo, \vec{p}_b e \vec{p}_v sono invece la posizione dell'oggetto bersaglio e quella del veicolo stesso. Il calcolo dell'*azione di sterzo* s_i permette al veicolo di sterzare in maniera proporzionale alla sua differenza angolare rispetto al suo obiettivo. K_s è un fattore moltiplicativo che permette di accentuare o diminuire questa azione ed è liberamente configurabile. L'azione di *accelerazione* è calcolata sulla base del valore di *sterzo* e, sebbene impedisce che il veicolo comandato dalla *IA* possa accelerare e sterzare al massimo in un dato istante, garantisce un controllo

maggiori e una strategia più consistente, impedendo allo stesso tempo che il veicolo possa collidere con ostacoli durante un cambio di direzione.

La strategia fondamentale è quella dedicata all'*esplorazione* e consente al veicolo di muoversi liberamente nell'arena al fine di individuare oggetti da collezionare o veicoli da aggredire. Questa strategia si basa sul raggiungimento di particolari *punti di controllo* invisibili posizionati nell'arena e selezionati in maniera *casuale*. L'intelligenza artificiale usa un componente aggiuntivo, l'*AI Field-of-View*, che ne modella il suo campo visivo. Ogni volta che un oggetto entra all'interno del campo visivo, il sistema ne determina la sua *utilità* e stabilisce se raccoglierlo (in caso di *orb* o *core*) o aggredirlo (in caso di un *veicolo avversario*), attivando una strategia opportuna. Questi oggetti possono causare un cambio di strategia solo se il veicolo è in fase *esplorativa*. Se l'oggetto bersaglio dovesse finire fuori dal campo visivo dell'intelligenza artificiale, il sistema annulla qualsiasi strategia in corso, riabilitando quella dedicata all'*esplorazione* dell'arena. La strategia di collezione degli oggetti è molto semplice, in quanto consiste nel calcolare un valore di input tale da permettere il raggiungimento dell'oggetto nel minor tempo possibile. La strategia *offensiva* invece si sviluppa in due fasi. Nella prima viene determinata l'utilità del veicolo da aggredire, ignorando tutti quelli con un numero di *orb* esiguo. Una volta che un *veicolo* viene selezionato come bersaglio, l'*AI* cerca di collidervi alla massima velocità (in maniera analoga a quanto avviene durante la collezione degli oggetti) e, ad impatto avvenuto, determina se vi è ancora necessità di proseguire l'attacco (valutando nuovamente il numero di *orb* posseduti). In caso positivo, l'*AI* cerca di allontanarsi dal veicolo avversario per un certo periodo di tempo selezionato in un intervallo casuale di pochi secondi, dopodiché procede nuovamente con la fase di *attacco*. In caso negativo, l'*AI* si porta in fase *esplorativa*. Per impedire che un'*IA* possa accanirsi contro un veicolo avversario, l'intera strategia *aggressiva* è temporizzata e viene disabilitata dopo un tempo casuale, portando l'*AI* nuovamente in fase *esplorativa*.

5.5 Multigiocatore locale

OrbTail offre un sistema in grado di gestire in maniera uniforme partite con un numero di giocatori locali compreso tra *uno* e *quattro*. Le modalità a giocatore singolo o multigiocatore online vengono gestite attraverso lo stesso sistema e non richiedono pertanto supporti specifici. In fase di selezione del veicolo, il gioco consente ai partecipanti di unirsi alla partita mediante la pressione di un tasto sul *pad* e selezionare un proprio veicolo. Ciascuna selezione è contenuta all'interno di un componente di tipo *Player Configuration*, il quale contiene l'indice del *giocatore locale*, il veicolo selezionato e l'identità del giocatore (umano o comandato dall'*intelligenza artificiale*). In fase di creazione della *lobby*, il *Lobby Manager* controlla

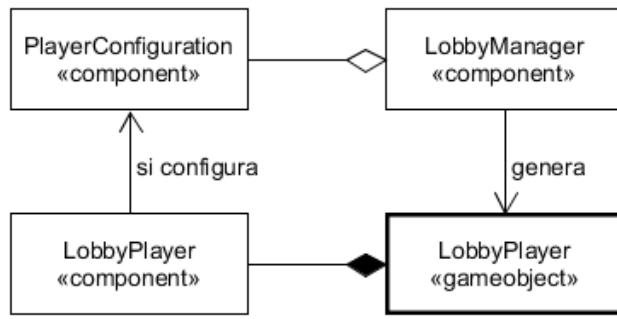


Figura 5.10: Gestione multigiocatore locale

il numero di *Player Configuration* presenti e crea un *Lobby Player* per ciascuno di essi. Questo meccanismo è inoltre utilizzato per l'aggiunta di giocatori comandati dall'*intelligenza artificiale* (vedi 5.3.1). Ciascun *Lobby Player* legge i dati presenti nella configurazione del giocatore e ne propaga lo stato sulla rete (Fig. 5.10) .

Durante l'inizializzazione della partita, il *Base Game Mode* crea i *controllori delle camere* per ciascuno dei giocatori umani, associando ad essi il *Lobby Player* corretto. La gestione della telecamera di ciascun giocatore è affidata al componente *Follow Camera*, il quale si occupa di «inseguire» il veicolo, e al componente *Camera*, messo a disposizione da Unity, il quale si occupa di gestire il *rendering*⁵. Quando il sistema associa il *Lobby Player* alla *Follow Camera*, quest'ultimo confronta l'indice del giocatore locale col numero totale di giocatori locali in partita e configura il *viewport*⁶ della *Camera* di conseguenza (Fig. 5.11) . Nel caso in cui vi sia un solo giocatore, il sistema assegna l'intero schermo; con più giocatori lo schermo viene diviso verticalmente (con *due* giocatori locali) o in quadranti (con *quattro* giocatori). La gestione di un numero di giocatori locali pari a *tre* fa invece eccezione. Originariamente era stato valutato di suddividere lo schermo in quadranti e lasciare uno di questi vuoto, tuttavia l'effetto non risultava piacevole e pertanto è stato deciso di suddividere in maniera asimmetrica lo schermo, assegnando una metà al primo giocatore e lasciando che gli altri due si spartissero l'altra metà (Fig. 5.12) .

L'introduzione della modalità *splitscreen* ha inoltre richiesto la suddivisione dell'interfaccia grafica in due: una dedicata alle informazioni di gioco quali il *cronometro di gara* e i *punteggi*, disegnata a tutto schermo, e una dedicata alla visualizzazione delle informazioni di ciascun giocatore locale, quali ad esempio il *potenziamento* attivo, disegnata nel quadrante opportuno.

⁵In computer grafica, il *rendering* è il processo di generazione un'immagine a partire da una rappresentazione matematica di una scena tridimensionale.

⁶In computer grafica, il *viewport* della telecamera rappresenta la porzione di schermo su cui viene renderizzata la scena inquadrata da quest'ultima. Solitamente il *viewport* copre l'intero schermo, tuttavia è possibile specificare una porzione più piccola al fine di renderizzare i punti di vista di più camere.

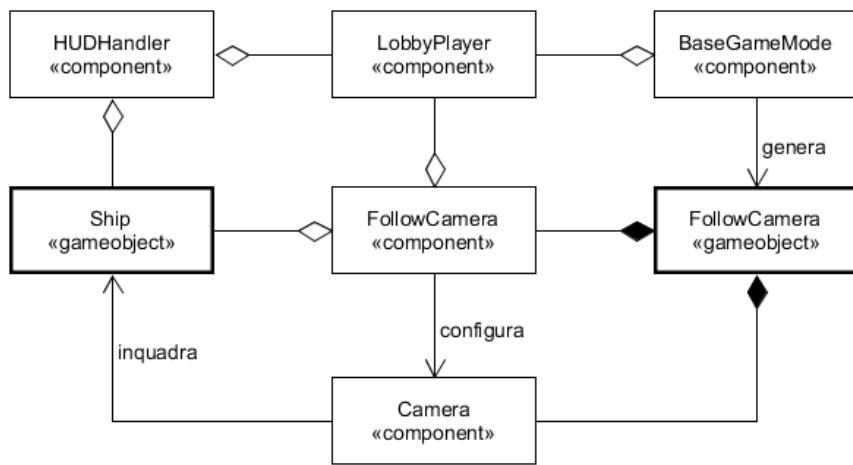


Figura 5.11: Gestione telecamera

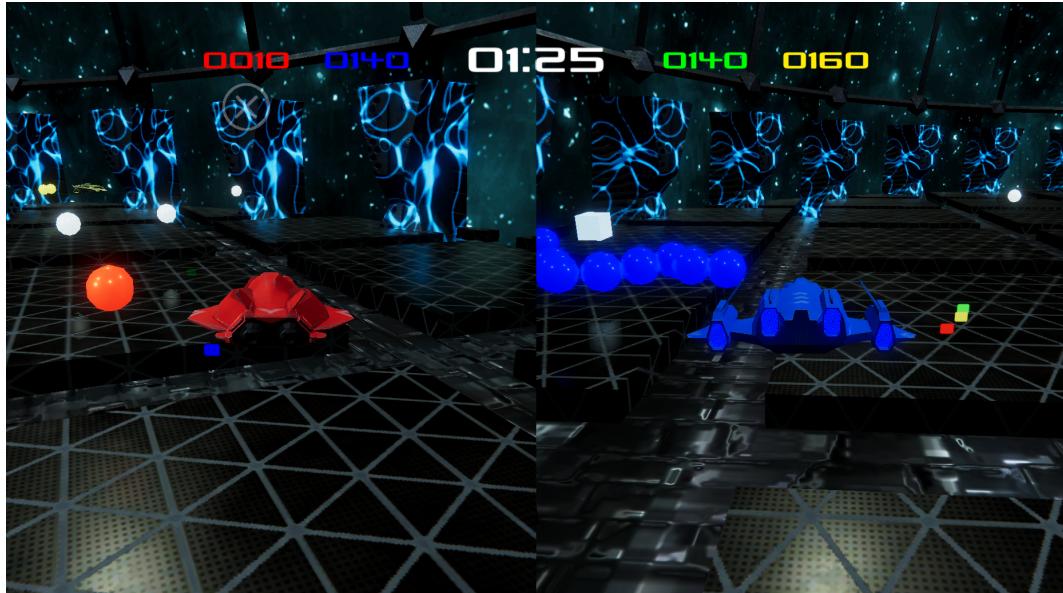


Figura 5.12: Splitscreen

Il gioco non usa il sistema di interfaccia utente messo a disposizione da Unity ma si basa su un sistema di pannelli tridimensionali che seguono il veicolo. Questa gestione è affidata al componente *HUD Handler* e permette di ottenere un’interfaccia mobile dotata di un certo grado di libertà ed inerzia. Questi pannelli utilizzano componenti come il *Text Mesh*, che consente di generare del testo tridimensionale nello spazio, e lo *Sprite Renderer*, il quale permette di mostrare immagini. L’introduzione di più punti di vista costituiti dalle diverse camere, ha inoltre introdotto una problematica altrimenti inesistente: ognuno può vedere la rappresentazione tridimensionale della *HUD* di ogni altro giocatore in prossimità del veicolo di quest’ultimo. Per ovviare a questo inconveniente ci si è affidati al sistema di *layer* di Unity. Un *layer*

non è altro che un nome simbolico che può essere associato ad uno o più oggetti di gioco e consente a quest'ultimi di essere *nascosti* dal rendering di determinate telecamere. L'*HUD* di ogni giocatore è assegnata ad uno di *quattro* layer, uno per ogni giocatore locale. La camera di ciascun giocatore è configurata per renderizzare solo l'*HUD* del giocatore opportuno e nascondere tutte le altre.

Durante lo sviluppo delle modalità *splitscreen*, ci si è imbattuti in una grave limitazione di Unity, ovvero la mancanza di supporto per *Audio Listener* multipli. Questi componenti sono usati per modellare la posizione virtuale dell'ascoltatore nello spazio, al fine di determinare la resa degli effetti sonori spaziali, quali esplosioni e il motore dei veicoli. Sebbene esistano diversi plugin a pagamento che possono essere usati per ovviare a questo inconveniente, è stato deciso di non usarne nessuno, sia per mancanza di tempo e sia perché le meccaniche di gioco non si basano sul corretto uso dell'audio. Come soluzione temporanea sono stati disabilitati gli *Audio Listener* dei giocatori ad eccezione del primo e si è rimandato lo sviluppo di una possibile soluzione ad iterazioni successive del progetto.

Conclusioni

La creazione di un videogioco in grado di essere fruito consistentemente su un gran numero di piattaforme costituisce, ad oggi, un ostacolo con il quale la maggior parte delle produzioni videoludiche si devono scontrare. Sebbene le tecnologie a disposizione consentono di semplificare il compito degli sviluppatori, trovare un’idea di gioco in grado di adattarsi ai vari ecosistemi e soddisfare diverse tipologie di utente, è una sfida che accompagna tutte le fasi produttive, a partire dal concept fino alla sua messa in opera. La creazione di un videogioco che possa essere giocato da più partecipanti, siano essi nello stesso luogo o separati da grandi distanze e che consenta l’interoperabilità fra ecosistemi differenti, è quanto di più complesso possa offrire l’industria videoludica. Ogni meccanica ed interazione utente deve essere attentamente ponderata, al fine di risultare consistente su tutte le piattaforme supportate. Questo processo non è tuttavia lineare: limitazioni tecniche, unite a criticità non preventivate possono richiedere il dover ritornare sui propri passi, anche più volte, al fine di individuare un nuovo insieme di meccaniche robuste ed efficaci.

In questo lavoro di tesi è stato fatto tesoro delle considerazioni di cui sopra, al fine di creare un videogioco dall’elevata qualità che potesse essere giocato da più giocatori, indipendentemente dal dispositivo utilizzato. Lo sviluppo di un’architettura software flessibile, in grado di gestire in maniera uniforme un gran numero di ecosistemi e casistiche di gioco, si è rivelato di cruciale importanza per mantenere sotto controllo il processo produttivo, evitandone l’esplosione combinatoria. Il risultato finale del processo di creazione di OrbTail ha portato alla nascita di un prodotto caratterizzato da meccaniche semplici ma efficaci, in grado di essere fruito tanto dai giocatori più accaniti quanto da quelli più saltuari.

Sebbene il prodotto risulta completo dal punto di vista delle meccaniche di base, esso si presta molto facilmente ad un gran numero di estensioni e migliorie. Per aumentare la longevità e la difficoltà di gioco è stata prevista la possibilità di integrare nuove arene oppure fornire variazioni di quelle esistenti, aggiungendo elementi *attivi* all’interno delle stesse, quali ostacoli e trappole.

Una modalità di gioco molto interessante presente nel concept originale ma rimosso per mancanza di tempo, è nota col nome «*blackout*». Questa modalità prevedeva la presenza di arene completamente al buio in cui le sole fonti di luce sarebbe-

ro state costituite dagli *orb* e dai *core*. Secondo questa meccanica i veicoli senza alcun *orb* sarebbero stati a tutti gli effetti invisibili agli occhi dei giocatori avversari, introducendo un elemento *stealth* inedito all'interno del gioco, aumentandone la profondità.

L'introduzione di nuove arene e modalità di gioco richiede necessariamente l'introduzione di un'*intelligenza artificiale* più sofisticata, in grado di evitare ostacoli e determinare obiettivi di gioco in funzione della modalità di gioco correntemente attiva. Attualmente le strategie sono generiche e, sebbene manifestano un buon comportamento in tutte le circostanze, non sempre garantiscono un livello di sfida all'altezza degli utenti.

Dal punto di vista dell'esperienza di gioco *online* una miglioria più che benvenuta è costituita dall'introduzione di algoritmi più raffinati per risolvere il problema del *dead-reckoning*: simulare la posizione degli oggetti di gioco in presenza di forte *latenza* o mancanza temporanea della *connettività*. Unity mette a disposizione un sistema molto basilare che però risulta in movimenti molto bruschi da parte di tutti gli oggetti di gioco e ciò costituisce il problema più evidente di OrbTail nelle modalità *online*.

Infine, in un'ottica di monetizzazione del prodotto e per aumentarne la longevità, è inoltre possibile prevedere l'introduzione di elementi a pagamento per personalizzare il proprio veicolo attraverso *livree* speciali ed altri oggetti estetici. Questo *modello di business* consentirebbe di mantenere il gioco costantemente aggiornato e supportato tramite nuovi oggetti in grado di attrarre l'attenzione di alcuni utenti, senza per questo incrinare l'equilibrio di gioco nei confronti degli altri.

Bibliografia

- [1] PixelPlacement. itween. <http://www.pixelplacement.com/itween/index.php>. Ultima visita in data: 24-06-2018.
- [2] Entertainment software association. "essential facts about the computer and video game industry". http://www.theesa.com/wp-content/uploads/2018/05/EF2018_FINAL.pdf, 2018. Ultima visita in data: 24-06-2018.
- [3] Unity Technologies. Manuale di unity. <https://docs.unity3d.com/2017.4/Documentation/Manual/>, 2017. Ultima visita in data: 08-07-2018.
- [4] Unity Technologies. Codice sorgente c# di unity. <https://bitbucket.org/account/user/Unity-Technologies/projects/PROJ>, Marzo 2018. Ultima visita in data: 24-06-2018.