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 il 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.

0.1 Unity

Unity è un motore grafico sviluppato da *Unity Technologies* che consente di sviluppare giochi multipiattaforma su tutte le piattaforme *mobile*, *console* e *desktop*[**?**]. 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 architetturale* 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[**?**].* 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.

0.1.1 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 funzionalità. Per lo sviluppo di OrbTail è stato deciso di utilizzarne uno solo dal nome *iTween[**?**].* 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.

[1] Il *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.

[2] Le *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.

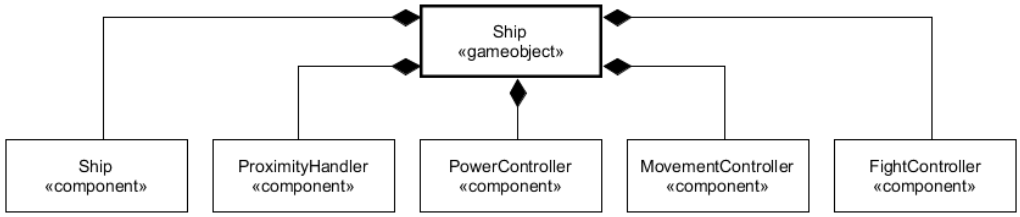

Figura 1: Struttura dei veicoli

0.2 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. 1) .

0.2.1 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 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 *campo gravitazionale* 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.

[3] Il *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.

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.2). Il primo di questi viene

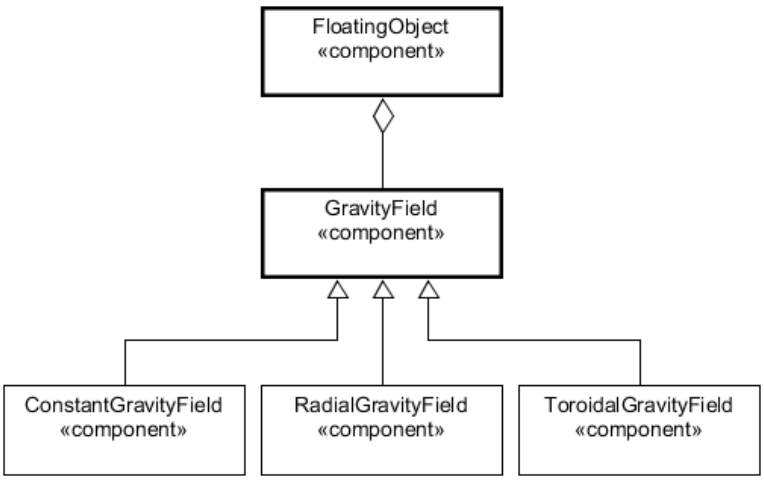

Figura 2: Gestione della gravità.

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

0.2.2 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 **??**):

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

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 passandovici attraverso.

0.2.3 Gestione dei collezionabili

La gestione della collezione degli elementi di gioco è affidata a tre componenti fondamentali (Fig.3). Il primo di questi, il *Proximity Handler*, rappresenta

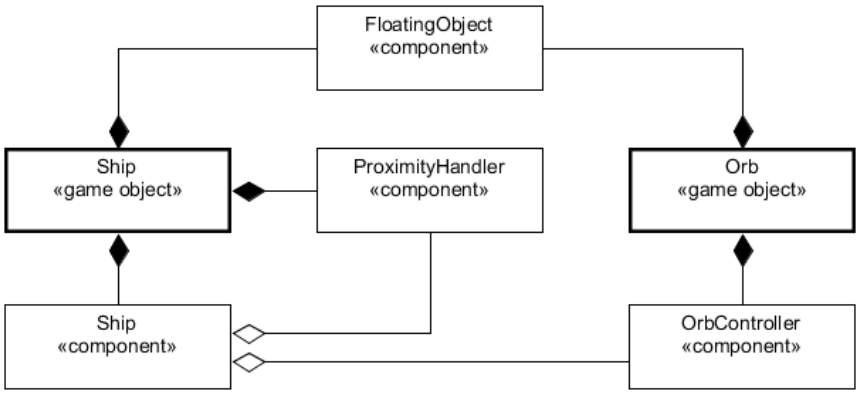

Figura 3: Gestione dei collezionabili.

un *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.

0.2.4 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 $\overrightarrow{p}$ la posizione del veicolo sui cui viene generato l'evento di collisione, $\overrightarrow{f}$ la direzione di quest'ultimo, $\overrightarrow{c}$ il punto di impatto trai veicoli coinvolti e $\overrightarrow{v_{rel}}$ la velocità relativa del veicolo rispetto all'avversario, la formula usata per determinare il numero di *danni* inflitti d è la seguente:

$$\overrightarrow{i_d} = \frac{\overrightarrow{c} - \overrightarrow{p}}{|\overrightarrow{c} - \overrightarrow{p}|}$$

$$i_s = \max(0, \overrightarrow{i_d} \cdot \overrightarrow{f})$$

$$d = |\overrightarrow{v_{rel}}| \cdot i_s \cdot off$$

Il termine $\overrightarrow{i_d}$ 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:

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

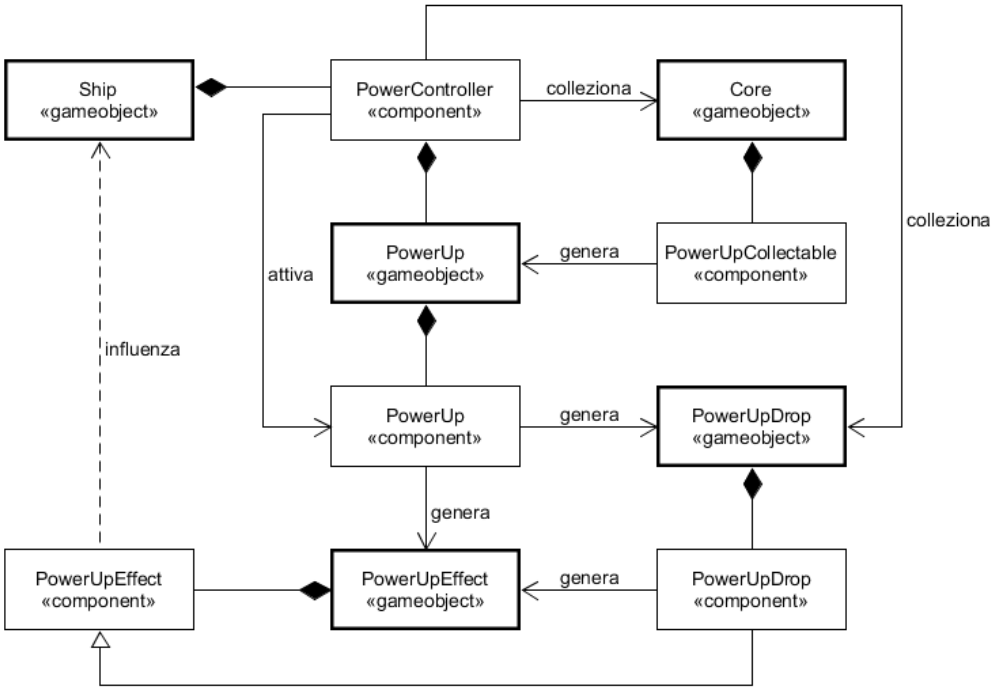

Figura 4: Gestione dei potenziamenti

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.

0.2.5 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. 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 estetica 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 *cool-*

down) 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.

0.2.6 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) . 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 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

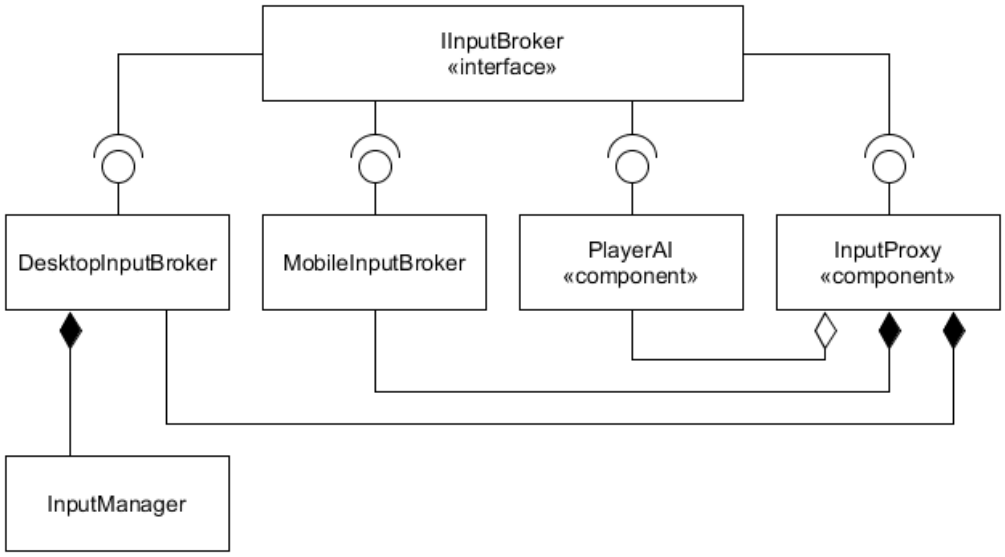

Figura 5: Gestione degli input

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.

0.3 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 soluzione è ottimale perché

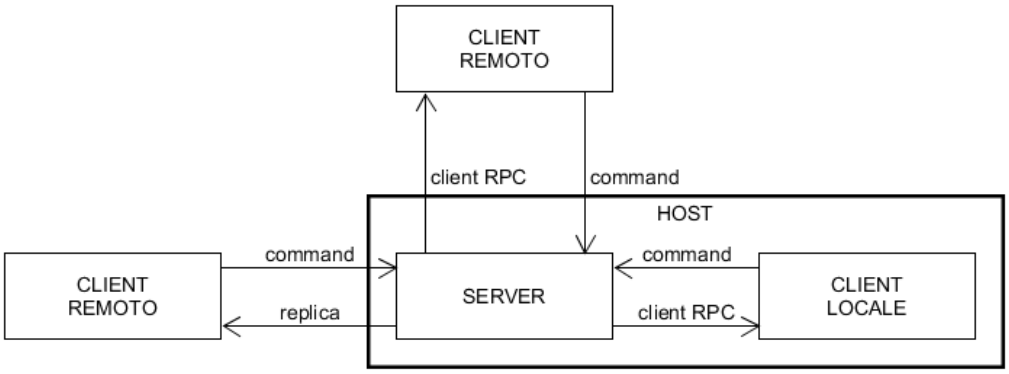

Figura 6: Architettura di rete

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. 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 *co-*

mandi 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.

0.3.1 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 CSV[4]. 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

[4]Il 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.

a giocatore singolo le fasi di ricerca delle *lobby* e di registrazione al servizio di *matchmaking* vengono ignorate ma la *lobby* viene comunque creata.

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.

0.3.2 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. 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 com-

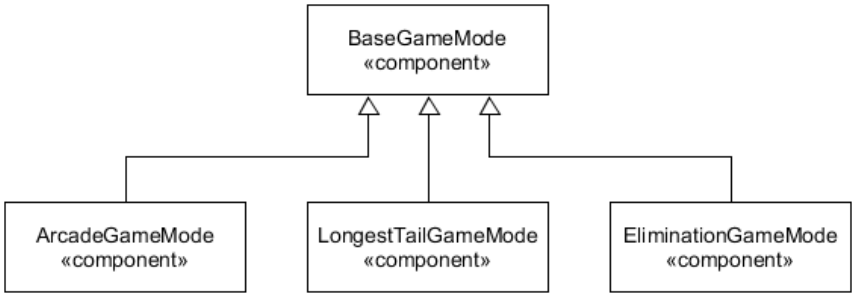

Figura 7: Modalità di gioco

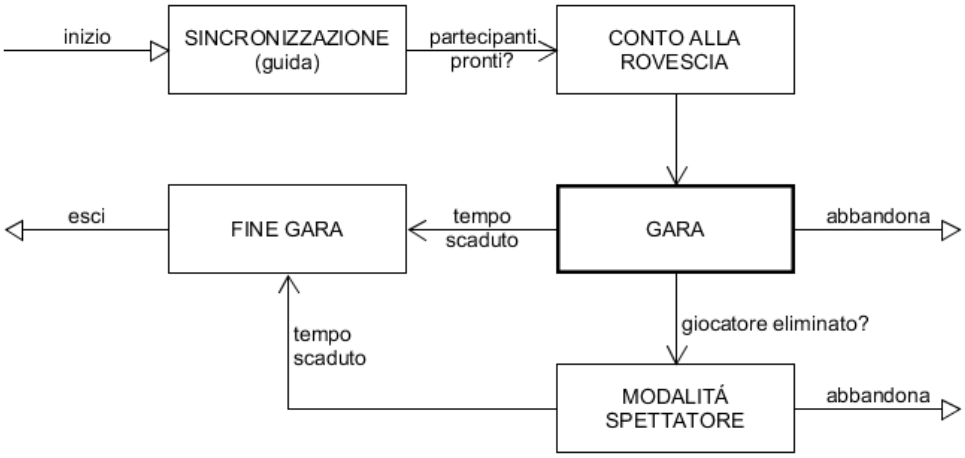

Figura 8: Flusso di gara

portamento 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 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. 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 trai 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 al menù 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

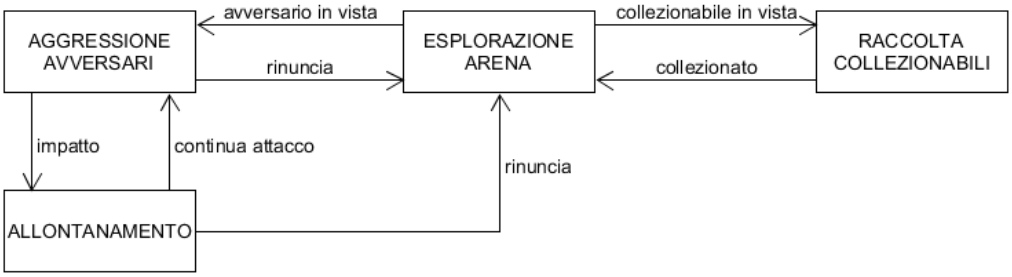

Figura 9: Strategie adottate dall'intelligenza artificiale

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.

0.4 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 0.2.6). 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. 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 $\overrightarrow{d_{rel}}$ di quest'ultimo rispetto al veicolo per calcolare i valori di *accelerazione* a_i e di *azione di sterzo* s_i:

$$\overrightarrow{d_{rel}} = \overrightarrow{p_b} - \overrightarrow{p_v}$$

$$s_i = \overrightarrow{u} \cdot (\overrightarrow{f} \times \overrightarrow{d_{rel}}) \cdot K_s$$

$$a_i = 1 - |s_i|$$

dove $\overrightarrow{u}$ e $\overrightarrow{f}$ sono i versori che rappresentano rispettivamente la direzione longitudinale e quella sagittale del veicolo, $\overrightarrow{p_b}$ e $\overrightarrow{p_v}$ 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 maggiore 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*.

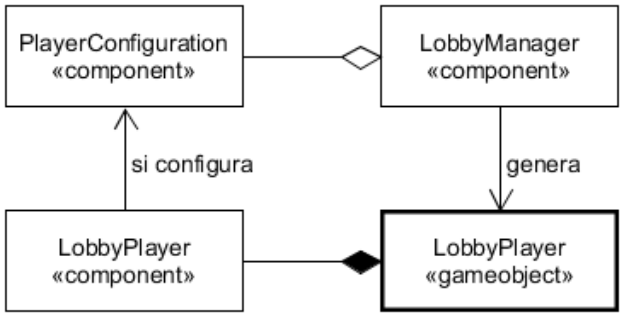

Figura 10: Gestione multigiocatore locale

0.5 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 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 0.3.1). Ciascun *Lobby Player* legge i dati presenti nella configurazione del giocatore e ne propaga lo stato sulla rete (Fig. 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

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

[6]In 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

Camera di conseguenza (Fig. 11) . Nel caso in cui vi sia un solo giocato-

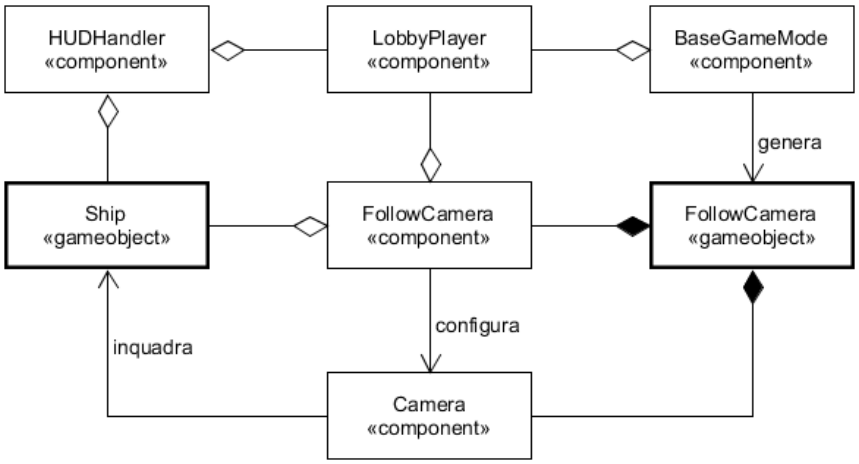

Figura 11: Gestione telecamera

re, 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. 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.

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.

renderizzare i punti di vista di più camere.

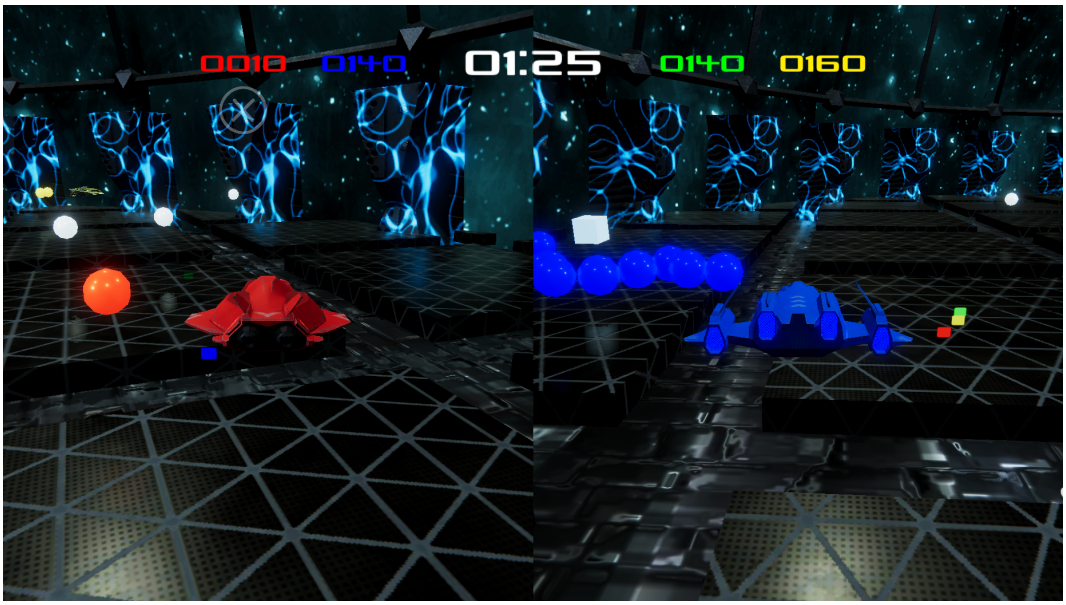

Figura 12: Splitscreen

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.