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 criticitá o limitazioni, specie nelle fasi piú avanzate del progetto. Per permettere a piú sviluppatori di lavorare sul progetto in contemporanea, mantenere lo storico dei file di progetto e controllare lo sviluppo delle varie funzionalitá ci si è affidati a *git*, un software per il *controllo di versione* distribuito. Per via degli interventi molto estesi che hanno visto la riscrittura quasi completa dei contributi originali, la *repository* del gioco risalente al 2013 è stata duplicata e il processo di sviluppo descritto in questo lavoro di tesi è proseguito su quest'ultima.

Un altro aspetto fondamentale riguarda la scelta del *motore grafico*. Fin dalla versione originale era chiaro 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 oggi vi sia un gran numero di motori grafici gratuiti in grado di soddisfare la maggior parte delle esigenze, durante lo sviluppo della prima versione di OrbTail le scelte erano ben piú limitate. Le due tecnologie principali erano rappresentate da Unity 4.3 e UDK e, sebbene quest'ultimo garantiva un'ottima resa visiva, lo strumento risultava instabile, poco documentato e particolarmente macchinoso. 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 molto intuitivo. Considerando che per questo lavoro di tesi ci si aspettava la riscrittura della maggior parte del codice, è stata inoltre avanzata l'ipotesi di un cambio motore grafico a favore di Unreal Engine 4. Sebbene quest'ultimo avrebbe garantito un'eccellente resa grafica, l'idea è stata rapidamente accantonata per via delle iterazioni di sviluppo incredibilmente piú lente che caratterizzano questo motore grafico, la mancanza di servizi di *matchmaking* e, soprattutto, delle criticitá riguardanti lo sviluppo su piattaforme *mobile* (specie in termini di prestazioni ed ottimizzazioni). Per questa nuova versione del titolo è stato pertanto deciso di aggiornare il motore grafico, passando da Unity 4.13 alla versione 2017.3, cercando di partire da una base quanto piú stabile possibile e proseguendo per iterazioni successive.

0.1 Unity

Unity è un motore grafico sviluppato da *Unity Technologies* che consente di sviluppare giochi multipiattaforma su tutte le pattaforme *mobile*, *console* e *desktop*. La presenza di una versione gratuita unita ad un elevato grado di intuitivitá 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 piú esperti. Il *pattern architetturale* principale, rappresentato dal*l'entity-component*, consiste nell'implementare funzionalitá autoconclusive all'interno di *componenti* indipendenti ed usare i *game object* (termine usato da Unity per descrivere le *entitá*) come aggregatori di quest'ultimi al fine di modellare comportamenti piú 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 piú 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 engine risiede nel fatto che il suo *core* è completamente *closed-source* (a meno di non pagare per ottenerne l'accesso) e ció 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'impossibilitá di modificare il codice del motore per adattarlo meglio alle proprie esigenze e ció richiede talvolta l'impiego di soluzioni temporanee o alternative.

Per lo sviluppo di OrbTail sono state usate prevalentemente funzionalitá di base legate al 3D, al motore fisico, alla gestione delle scene e delle funzionalitá di rete. L'uso di funzionalitá generiche ha permesso di evitare tutte le problematiche legate ai sistemi piú 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 giá in fase avanzata e si voleva evitare di correre *rischi* inutili. L'architettura di gioco sfrutta inoltre molti oggetti *manager* al fine di scambiare informazioni tra piú livelli e per fornire un unico punto d'accesso ai vari sottosistemi. Un manager è un'*entitá* di gioco che funge da *singleton* accessibile da tutti gli altri sottosistemi al fine di leggerne o modificarne lo stato.

[1] Il *core* di un motore grafico è il modulo software di piú basso livello. Esso si occupa di astrarre le funzionalitá 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.

L'uso dell'apposito strumento di aggiornamento di versione fornito dal motore si è rivelato di notevole importanza, in quanto ha permesso fin da subito di ottenere un prodotto stabile a partire dalla versione originale del 2013. Al termine del processo di *porting* automatico, il gioco ha preservato *tutte* le funzionalitá, comprese quelle legate alla rete, al *matchmaking* e al *cross-platform play* manifestando problematiche minori dovuto all'uso di *API* [3] obsolete (principalmente in ambito *UI* e di gestione dei livelli). Una volta ottenuta una versione stabile, sono state infine aggiornate tutte le funzionalitá superate sostituendole con le nuove varianti messe a disposizione del motore. Al termine di questa fase ci si è concentrati sull'aggiornamento dell'architettura del gioco e sull'implementazione delle nuove funzionalitá.

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 funzionalitá. 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 puó essere integrato nel progetto e non richiede alcuna forma di configurazione. Nella fase di *porting* è stato necessario aggiornare il plug-in alla versione piú recente.

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 modalitá e livelli e pertanto sono state pensate per essere completamente indipendenti da essi. Tra le meccaniche di base troviamo la gestione della gravitá, del sistema di controllo dei veicoli, della gestione degli oggetti collezionabili e degli impatti. Tutte le funzionalitá qui descritte sono state sviluppate in una mappa di test apposita, con risorse grafiche temporanee e gestione degli input molto basilare.

[3] Col termine *API,* acronimo di *application programming interface,* ci si riferisce a tutte quelle funzionalitá di un sistema rese disponibili agli sviluppatori tramite opportune *procedure.*

0.2.1 Gestione della gravitá

I livelli offerti dal gioco sono caratterizzati da topologie profondamente diverse, il che rende impossibile utilizzare la gravitá automatica fornita dal motore fisico di Unity per far fluttuare gli oggetti su di essi. Durante la prima iterazione ci si è affidati al sistema di *raycasting* [4] al fine di individuare un punto sulla superficie del livello da usare come base per il calcolo, tuttavia ci si è presto accorti che il sistema era affetto da gravi difetti. Sebbene l'uso del *raycast* 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 puó 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 ambiguitá ci si è affidati ad una descrizione *analitica* della forza di gravitá per ciascuna delle topologie supportate. Il *campo gravitazionale* cosí 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 *GravityField,* da cui derivano le diverse descrizioni dei campi di gravitá supportati, e il *FloatingObject* che, assegnato agli oggetti di gioco, consente loro di fluttuare sull'arena. (Fig.1). Il primo di questi viene

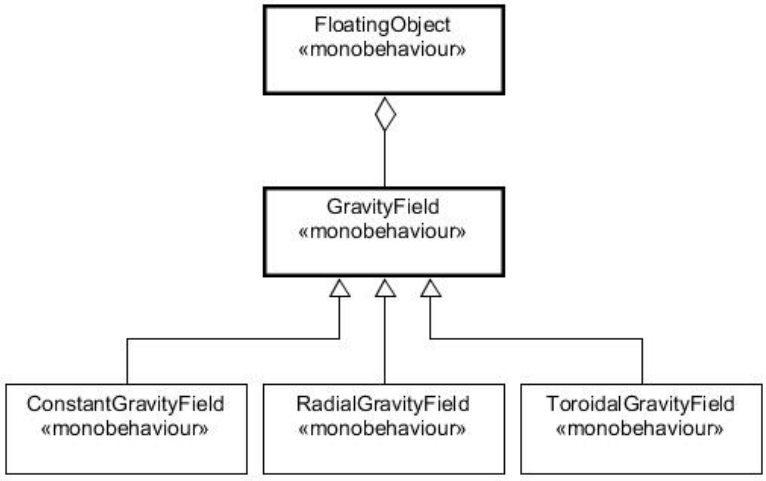

Figura 1: UML Gestione della gravitá.

[4] Il *raycasting* è una tecnica che consente di determinare le intersezioni tra un raggio descritto da un punto inziale ed una direzione ed una o piú superfici poligonali.

assegnato all'arena e consente di configurare sia l'intensitá della gravitá 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 gravitá.

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 considerando la direzione della forza di gravitá e l'input dell'utente.

Il componente *MovementController* determina la velocitá 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, bensí è affidata ad un semplice controllore *PID*. Il sistema adoperato consente di controllare la velocitá 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 velocitá 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 velocitá desiderata $r(t)$ è determinata dal valore massimo di *velocitá* 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 *manovrabilitá* del veicolo (vedi **??**):

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

La direzione di accelerazione è ricavata tramite il componente *FloatingObject* in funzione della tangente del campo di gravitá e della rotazione del veicolo, e ció 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. Il primo di questi, il *ProximityHandler*, rappresenta 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 peró 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 *FloatingObject* al fine di fluttuare sull'arena e un altro componente, l'*OrbController*, al fine di gestire i vincoli fisici di cui sopra.

0.2.4 Gestione degli impatti

Il sistema di gestione degli scontri tra veicoli é implementato attraverso un componente *FightController,* il cui scopo é 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. Nella versione originale ciascuno di essi aveva uno o piú *collisori fisici* che approssimavano piú 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 entitá 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 é 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 é identica per tutti i veicoli e ció 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 *FightController* reagisce a questi eventi, determinando il numero di *orb* persi a causa dello scontro e lasciando al *FightController* del veicolo avversario il compito di staccare *orb* dal proprio.

Il numero di *orb* persi durante uno scontro dipende dalla direzione d'impatto e l'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 p la posizione del veicolo sui cui viene generato l'evento di collisione, f la direzione di quest'ultimo, c il punto di impatto trai veicoli convolti e v_{rel} la velocitá relativa del veicolo rispetto all'avversario, la formula usata per determinare il numero di *danni* inflitti d é la seguente:

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

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

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

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

0.3 Networking

Descrizione dell'architettura di rete, della replicazione dei componenti e dello stato di gara.

0.4 Modalitá di gioco

Descrizione del flusso di gioco, dello stato della gara e delle condizioni di vittoria.

0.5 Potenziamenti

Gestione dei potenziamenti con riguardo per la parte online e di sincronizzazione.

0.6 Intelligenza artificiale

0.7 Splitscreen

0.8 Gestione degli input

0.9 Menú