POLITECNICO DI TORINO

**Laurea Magistrale in Ingegneria Informatica,
orientamento Grafica e Multimedia**

Politecnico
di Torino

Tesi di Laurea Magistrale

Creazione di un'architettura a Server Autoritativo all'interno di un prodotto videoludico multigiocatore

Relatore

Prof. MARCO MAZZAGLIA

Candidato

SIMONE GAINO

Luglio 2022

"Un ingrediente vitale del successo è non sapere che ciò che stai tentando non può essere fatto"
Sir Terry Pratchett

Abstract

Oggetto della tesi è la creazione di un'architettura a Server autoritativo per la gestione di un gioco multigiocatore. Questo lavoro è parte di un progetto più grande di sviluppo, portato avanti da un team di 3 studenti, di un videogioco multiplayer usando Unreal Engine.
Il prototipo sviluppato prende la forma di uno sport di fantasia, in cui un gruppo di insetti umanoidi si confronta usando le proprie caratteristiche e abilità. Nello specifico la tesi si concentra sullo studio delle tecnologie e architetture legate al gioco online e alle sue problematiche, approfondendo soprattutto le tematiche dell'architettura Authoritative Server e di come questa sia implementata all'interno di Unreal Engine.
Oltre a questo nucleo centrale saranno anche approfonditi i ruoli di Game Designer e di Gameplay Programmer, analizzando il processo creativo che ha portato alla nascita dell'idea dietro al prototipo e alla sua implementazione attraverso le tecnologie dell'Engine.
Nello specifico la tesi si divide in 3 sezioni:

1. Analisi del regolamento del gioco e del processo creativo che ha portato alla sua creazione.

2. Analisi delle caratteristiche dei giochi online, seguita da una analisi della libreria di networking di Unreal Engine e dalla spiegazione delle personalizzazioni più importanti effettuate per adattarlo alle necessità del progetto.

3. Analisi del Gameplay Ability System, il plugin di Unreal che ha permesso di creare il set di abilità e di caratteristiche dei personaggi e delle sue caratteristiche legate al gioco online.

In ogni sezione, oltre a discutere delle potenzialità e dei limiti delle tecnologie, si approfondisce anche di come queste caratteristiche vengano espresse nel progetto e come siano state personalizzate per le necessità del prodotto.

Indice

Elenco delle figure

Capitolo 1

Introduzione

Internet nel corso degli ultimi anni ha ottenuto un ruolo importantissimo nelle nostre vite.
Dagli smartphone alle automobili, dagli elettrodomestici alle lampadine di casa, ormai la rete è onnipresente e gestisce praticamente qualsiasi attrezzatura che usiamo nella vita di tutti i giorni. In un mondo con queste caratteristiche i videogiochi non potevano rimanere immuni al fascino delle enormi opportunità di gameplay offerte dalle diffuse connessioni al web.
Nascono così molti dei giochi più popolari degli ultimi anni, in grado di raccogliere anche più di cento milioni di utenti attivi mensili e che per le nuove generazioni hanno assunto un ruolo, a livello di intrattenimento, molto simile se non identico a quello che ha sempre avuto lo sport all'interno della nostra società.
In questa tesi si analizza la creazione di un prodotto videoludico che fa di questa tecnologia il proprio nucleo, studiando il processo creativo che ne ha guidato la nascita e che ne ha plasmato le meccaniche per adattarlo a una sfida online.
Partendo dal risultato del processo creativo si vanno successivamente ad analizzare, in maniera ancor più approfondita, le tecnologie e le scelte architetturali operate dal team di sviluppo, confrontandole brevemente anche con le principali alternative e specificando quali ragioni abbiano guidato tali scelte.
In particolare si approfondisce il concetto di server autoritativo, un tipo di architettura usata da quasi ogni gioco online che punti a un'esperienza seriamente competitiva. In questo contesto un unico server funge da campo da gioco e arbitro per l'intera partita e i singoli client si riducono a semplici schermi sul campo e ricevitori per gli input dei giocatori.
Si scende inoltre nel dettaglio di come questa architettura sia stata implementata all'interno del game engine scelto per il progetto, nello specifico Unreal Engine 4, e di come quest'ultimo implementi i concetti chiave dell'architettura precedentemente presentata e di come siano stati utilizzati per esprimere al meglio le funzioni necessarie per il progetto in sviluppo. Ciò riguarda in particolare la libreria di networking

interna all'engine stesso e due plugin molto utilizzati, lo "Steam subsystem" e il "Gameplay ability system".

1.1 Motivazione

I videogiochi sono sempre stati, fin dalla mia infanzia, una grande passione. Ai miei occhi quelli che prendevano vita sullo schermo non erano solo buffe immagini in movimento, o degli sciocchi modi per sprecare tempo, come li sentivo spesso descrivere da più parti, ma erano invece delle grandi avventure, dei film o dei libri di cui mi veniva messa in mano la trama, per deciderne con i miei sforzi e le mie battaglie il futuro. Un mondo in cui immergermi per poter vivere le avventure di cui la vita di tutti i giorni, soprattutto un quella di un bambino, è tristemente priva.
Con il passare degli anni e con la passione per la tecnologia che cresceva, nella mia mente iniziarono a nascere nuove domande. Con quali mezzi un computer o una console riuscivano a creare animazioni realistiche o a gestire gli eventi di un mondo così grande?
Come potevano così tanti giocatori interagire in contemporanea da posti lontanissimi nel mondo senza avvertire nessun ritardo tra i propri comandi e le reazione del proprio avatar? Come facevano i programmatori a difendere i giocatori dalle azioni fraudolente degli altri?
Questa curiosità e questa passione mi hanno quindi spinto a scegliere, insieme a un piccolo gruppo di colleghi, con cui condivido questa dedizione, una tesi che approfondisse questi argomenti, nella speranza non solo di creare una tesi adatta alla conclusione del mio percorso di studio, ma di andare a mettere un primo fondamentale tassello di quella che spero essere la mia carriera.

1.2 Outline

I capitoli sono organizzati nel modo seguente:

- **Capitolo 2 - Stato dell'arte:** Viene analizzata la storia dei giochi multiplayer, partendo dai primi giochi visualizzati su oscillatori analogici ai moderni giochi online, con migliaia di persone connesse nella stessa partita, descrivendo brevemente le tecnologie che hanno guidato questo sviluppo.

- **Capitolo 3 - Prodotto videoludico:** Descrizione del videogioco e dei suoi concetti chiave, seguita dalla descrizione dei ruoli ricoperti all'interno del team di sviluppo.

- **Capitolo 4 - Game Design:** Analisi del percorso che ha portato dall'ideazione al raffinamento del concetto iniziale del gioco, approfondendo le modifiche che il playtesting ha imposto all'idea iniziale.
- **Capitolo 5 - Studio correlato:** Studio dei problemi principali nella creazione di un gioco online, delle architetture e delle soluzioni tecnologiche più utilizzate e spiegazione delle scelte prese per lo sviluppo del progetto.
- **Capitolo 6 - Networking:** Approfondimento sul codice di rete di Unreal Engine, i suoi concetti chiave, la sua architettura e le personalizzazioni effettuate per il prodotto creato nella tesi.
- **Capitolo 7 - Gameplay Ability system:** Analisi del plugin Gameplay Ability system, dei suoi concetti chiave e funzionamento, con un approfondimento su una abilità specifica sviluppata per il gioco.
- **Capitolo 8 - Conclusioni:** Considerazioni finali sullo stato raggiunto dal progetto e sul suo possibile futuro.

Capitolo 2

Stato dell'arte

2.1 Breve storia dei videogiochi multigiocatore

2.1.1 Gli inizi

La possibilità di far confrontare più giocatori è sempre stata una delle caratteristiche principali dei videogiochi, si pensi a "Tennis for two" del 1958 o al ben più famoso "Pong" degli anni '70, che facevano sfidare due giocatore in una simulazione di una partita di Tennis. Questo tipo di giochi era però limitato dalla necessità di far svolgere l'intera partita sulla stessa macchina, circoscrivendo quindi il numero di sfidanti a una cerchia ristretta di persone.
La programmazione di questo tipo di giochi è del tutto analoga a quella di un gioco singleplayer, con la semplice aggiunta degli input di un giocatore extra e, se necessario, di un nuovo punto di vista.

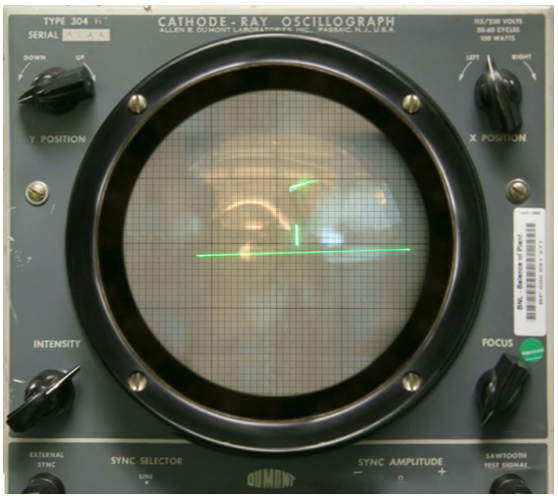


Figura 2.1: Tennis for two

Un primo piccolo passo verso il moderno concetto di gioco online avvenne sempre negli anni '70 all'interno delle università americane, in cui erano messi a disposizione di professori e studenti dei terminali che permettevano di accedere al

cluster universitario con logiche di timesharing.
Proprio sfruttando questa struttura, più terminali riuscivano a condividere le informazioni necessarie ai giochi per creare partite condivise. Il fenomeno si diffuse a tal punto da portare al divieto di questo tipo di programmi in molte università, per via dell'alto consumo di risorse, ma ciò non impedì a questo nuovo modo di vivere il videogioco di diffondersi nelle menti di quelli che sarebbero diventati i programmatori e designer dell'industria videoludica negli anni '80.

2.1.2 Primi sistemi host-based

Con lo sviluppo dei primi protocolli di comunicazione tra dispositivi iniziarono a nascere anche i primi videogiochi ideati per sfruttarne le caratteristiche. Vengono così lanciati sul mercato giochi come MIDI Maze, uno sparatutto che permetteva di creare una partita condivisa tra più giocatori attraverso una connessione creata utilizzando le porte MIDI dell'Atari ST. Ancora molto lontano da quello che è oggigiorno il concetto di gioco online ma per la prima volta veniva introdotto il concetto di due macchine totalmente separate che condividevano i dati necessari al gioco non attraverso la condivisione di un cluster centrale ma grazie a un protocollo di comunicazione.
Le due limitazioni più grandi di questo tipo di tecnologia erano il numero limitato di porte a disposizione di ogni computer (generalmente due porte seriali o MIDI), che costringeva alla creazioni di reti ad anello, e all'obbligo di giocare con tutti i giocatori in un unico luogo, non apportando grandi cambiamenti rispetto ai giochi multigiocatore giocati sulla stessa macchina, tipici delle console casalinghe.

Figura 2.2: Due Game Boy connessi via cavo

Un'applicazione di queste tecnologie fu anche lo sviluppo di porte e protocolli dedicati al solo gioco che si espressero soprattutto nel campo delle console portatili, come il Gamegear di Sega o il Gameboy di Nintendo, che mettevano a disposizione dei giocatori un cavo per connettere più console.

2.1.3 Ethernet e l'avvento di internet

Un primo passo verso la moderna tecnologia di rete avvenne con la diffusione dello standard Ethernet, che permetteva di creare reti locali e, a partire da DOOM (1993), molti giochi iniziarono a supportare questo tipo di connessioni per creare partite competitive per un piccolo numero di giocatori. Negli anni '90, con il

diffondersi delle prime connessioni a internet casalinghe e basandosi sugli stessi concetti sviluppati per i giochi su Ethernet, iniziarono a nascere i primi giochi che prevedevano l'interconnessione tra giocatori come fulcro del proprio gameplay e che permettevano di connettere nella stessa partita persone molto lontane nel mondo reale.
Dalle idee dei game designer videro la luce molti stili di gioco diversi, dai giochi strategici in tempo reale, in cui un limitato numero di giocatori muove una grande quantità di unità contro i propri avversari, ai Massive Multiplayer Online, con centinaia o migliaia di giocatori connessi in contemporanea allo stesso server per giocare partite condivise.

2.2 Multiplayer online odierno

Dopo 30 anni dalla nascita di internet la tecnologia ha fatto grossi passi avanti e il numero di giochi con almeno una componente online è ormai altissimo. Partendo dalla semplice condivisione dei risultati raggiunti fino ai veri e propri confronti diretti nei più disparati tipi di scontri. Ogni gioco ha dovuto ideare una propria infrastruttura di rete per meglio affrontare le proprie difficoltà tecniche. Senza scendere eccessivamente nei dettagli, le tipologie di infrastruttura si dividono principalmente in 3 categorie:

- **Peer to peer:** Non esiste nessuna distinzione tra i vari client e non esiste nessun server. Ogni client comunica a tutti gli altri il proprio stato e tutte le informazioni necessarie a sincronizzare la partita per tutti i giocatori.
- **Client hosted:** Un client ricopre anche il ruolo di server nei confronti degli altri client. Ognuno di quest'ultimi comunica le proprie informazioni all'host della partita e riceve indietro i dati per l'update.
- **Authoritative server:** Esiste una chiara distinzione tra i molti client e il singolo server. Quest'ultimo è l'unico a gestire la partita, ricevendo tutte le comunicazioni di ogni client e comunicando in direzione opposta le informazioni necessarie al singolo client per sincronizzare la propria partita con le altre.

Capitolo 3

Prodotto videoludico

L'obbiettivo del team è quello di sviluppare un videogioco multiplayer online in tempo reale di cui la presente tesi presenta una parte del lavoro.
Il team è composto da 3 studenti magistrali del Politecnico di Torino, datisi il nome di 3Bugstards.
Il lavoro del team si può dividere in 3 rami principali, di cui questa tesi tratterà nei particolari soprattutto elementi dei primi 2:

- **Engineering:** Fondamento tecnico del videogioco.
  - Multiplayer online e networking
  - Gameplay programming
  - Collezione, pulizia e analisi dei dati generati dalle partite
- **Design:** Ideazione e miglioramento delle regole ed elementi di base del gioco
  - Scelta del set iniziale di regole e l'ambientazione
  - Miglioramento progressivo del gioco, adattandosi anche ai limiti tecnici posti dal team di Engineering
  - Confronto con i pareri dei tester del gioco, per migliorare l'esperienza degli utenti
  - Valutazione dei dati raccolti dalle partite
- **Artist/Tech artist:** Sviluppo del lato estetico del gioco
  - Modellazione dei modelli 3D
  - Animazioni dei sopracitati modelli
  - Audio e sound design, scelta e inserimento di musiche e effetti sonori adeguati

- Programmazione degli shader utilizzati
- Texturing dei modelli
- UI design cioè l'ideazione delle interfacce grafiche per l'utente
- VFX, quindi creazione e inserimento degli effetti grafici nel gioco

3.1 Descrizione

Il prodotto ideato dal team è "Bugball", un videogioco sportivo multiplayer, ispirato al gioco reale del Dodgeball, seppur con molte regole modificate. La caratterizzazione del gioco è basata sul mondo degli insetti, in cui ogni personaggio è un insetto (o animali simili, ad esempio ragni) reinterpretato in una forma umanoide, le cui caratteristiche e abilità derivano da quelle dell'animale preso ad ispirazione. L'idea di fondo era di generare un gameplay veloce e frenetico, che costringesse i giocatori a un rapido pensiero strategico e a degli ancora più rapidi riflessi e reazioni.

Figura 3.1: Arena di gioco e personaggi disponibili

3.2 Ruolo primario

3.2.1 Multiplayer/Network programmer

Il ruolo principale svolto nel team è stato quello di Multiplayer e Network programmer.
Questa figura deve ideare e implementare la struttura di rete all'interno di gioco e gli algoritmi di comunicazione, o adattare quelli già disponibili all'interno dell'engine al gioco in sviluppo. È un ruolo estremamente importante nel mercato moderno, in quanto dalla qualità del suo lavoro dipende sia la reattività del gioco agli input, rendendolo quindi piacevole da giocare, sia la sicurezza che nessuno dei giocatori possa compiere azioni scorrette, garantendo quindi una sfida equa tra tutti i partecipanti.

3.3 Ruoli secondari

In aggiunta al ruolo primario, soprattutto a causa delle piccole dimensioni del team, ogni membro ha assunto più ruoli. Lo svolgimento della tesi ha quindi richiesto anche la copertura dei ruoli elencati e descritti qui di seguito.

3.3.1 Game designer

Probabilmente il ruolo più importante all'interno di un team di sviluppo, il game designer è colui che si occupa di ideare il gioco in se, con le proprie regole e processi, e di migliorarlo durante lo sviluppo, confrontandosi con ogni altro membro del team. Nel caso del prodotto discusso in questa tesi il processo di design è stato condiviso tra quasi tutti i membri del team ed è stato diviso in più fasi, una principale e più importante all'inizio per decidere lo schema generale del gioco, con le regole fondamentali, e una serie di fasi di modifica più piccole avvenute durante tutto lo sviluppo, in cui venivano definiti e ridiscussi tutti gli elementi fondamentali del gioco, in modo da renderlo sempre più accattivante da giocare.

3.3.2 Gameplay programmer

Un gameplay programmer è colui che, partendo dalle regole e elementi ideati dal game designer, crea la struttura interna del gioco. Si differenzia dal Network Programmer perchè ricopre una posizione di più ampio respiro, non per forza legata alla comunicazione tra i vari client e il server.
Validi esempi di questa tipologia di lavoro sono la creazione delle abilità del singolo personaggio, l'integrazione dell'interfaccia grafica con i componenti del gameplay o le sezioni di codice che determinano le animazioni dei singoli personaggi.

Capitolo 4

Game Design

In questo capitolo verranno trattate tutte le regole e caratteristiche fondamentali del prodotto ideato, seguite da una breve descrizione del processo di sviluppo che ha portato all'ideazione dei concetti fondamentali e della loro raffinazione durante lo svolgimento della componente tecnica del progetto.

4.1 Regole

Due squadre composte da 1, 2 o 3 giocatori l'una, a seconda della modalità, si sfidano in un campo diviso in due metà, similmente alla pallavolo, seguendo le seguenti regole:

- Ogni giocatore che si scontra con la parete di fondo del campo viene momentaneamente espulso, per un tempo sempre più lungo con il prosieguo della partita, e assegna un punto alla squadra avversaria
- Allo scadere dei 10 minuti di partita la squadra con più punti vince
- In caso di parità vi sono dei tempi supplementari nei quali i giocatori non possono rientrare in partita dopo essere stati eliminati. La vittoria viene assegnata alla prima squadra che riesce ad eliminare totalmente la squadra avversaria
- Se, in un qualsiasi momento, una squadra viene eliminata in tutti i suoi membri la vittoria viene assegnata alla squadra avversaria (unica condizione di vittoria nelle partite 1 contro 1)

Soltanto nelle partite a 2 giocatori è prevista una modifica alle regole dei tempi supplementari, cioè che per tutta la loro durata le barriere di fondo campo compiono un avvicinamento graduale alla linea di centro campo, rendendo la partita sempre più intensa.

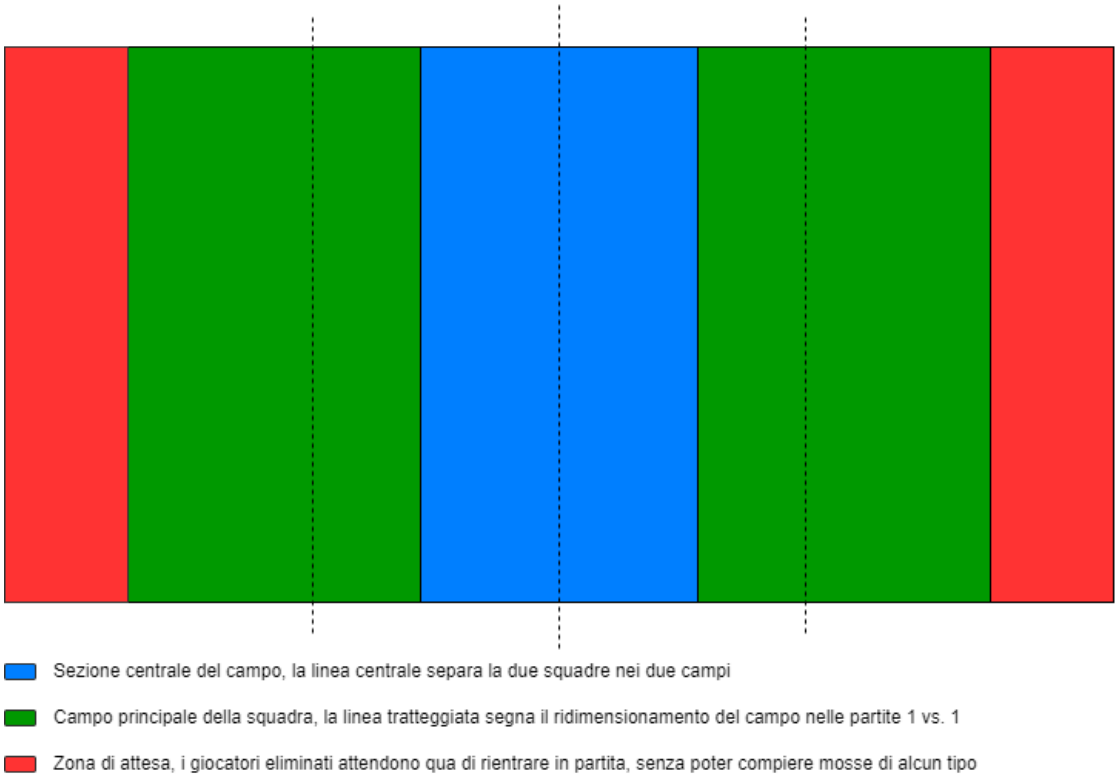


Figura 4.1: Schematica dell'arena di gioco

4.2 Abilità

Per permette una personalizzazione maggiore di ogni personaggio, il team ha pensato di dotare ognuno di essi di 3 slot. Ognuno di questi slot permette di scegliere un'abilità da un piccolo insieme, caratteristico per ogni slot e personaggio, all'inizio di ogni partita.
Ogni abilità ha, oltre alla propria meccanica caratteristica, un tempo di cooldown, che impedisce di usarla in rapida successione, obbligando i giocatori a ragionare con attenzione su che abilità usare in ogni momento.

4.2.1 Tipologie di abilità

Le abilità ideate si possono dividere in differenti categorie, a seconda delle loro caratteristiche:

- **Skillshot:** Abilità in cui viene generato e lanciato un proiettile, che segue un percorso prefissato, senza puntare a nessun bersaglio specifico. Come

suggerisce il nome, il successo di queste abilità dipende molto dalla capacità del giocatore di saperle usare nel modo corretto al momento giusto.

- **Buff/Debuff:** Abilità che permettono rispettivamente di aumentare o diminuire il valore di un attributo (che non sia la resistance) di uno o più giocatori
- **AoE:** Nome breve per l'inglese "Area Of Effect". Queste abilità influenzano tutti i giocatori della tipologia corretta che si trovano all'interno di un'area selezionata al momento di attivazione.
- **Movement:** Abilità che permettono al personaggio di muoversi in maniera particolare, in genere più velocemente della velocità base.
- **Untargetable ability:** Abilità che rendono un personaggio immune ad alcune o a tutte le conseguenze dei colpi in arrivo.

In aggiunta alle categorie precedenti va anche specificata la presenza di un attacco base, comune a tutti, che permette di raccogliere una delle palle presenti nel campo e di lanciarla all'avversario, nel tentativo di infliggergli danni.
Alcune abilità potrebbero rientrare in più di una categoria, come il colpo con il pungiglione ideato per l'ape, uno skillshot che all'impatto applica un debuff che annulla la rigenerazione della Resistance avversaria.

4.2.2 Attributi

Un altro parametro molto importante per creare una miglior distinzione tra i ruoli di ogni personaggio è l'ideazione di un set di attributi che vada a definire le caratteristiche del personaggio. Dopo diverse fasi di prototipizzazione, che hanno visto l'aggiunta e la rimozione di diversi attributi all'elenco, il numero di caratteristiche fondamentali per ogni personaggio si è ridotto a 6:

- **Resistance:** Questo parametro funge da vita del giocatore. Viene ridotto per ogni colpo subito e, una volta raggiunto lo 0, il personaggio non potrà muoversi o usare abilità per qualche secondo.
- **Resistance regeneration:** Quantità di resistance recuperata per ogni secondo di partita passato senza subire colpi
- **Armour:** Influenza, secondo un meccanismo che verrà spiegato in un capitolo successivo, il quantitativo di danno subito dai colpi ricevuti.
- **Speed:** Velocità con cui si sposta il personaggio
- **Throw Speed:** Velocità con cui si muovono le palle lanciate dal personaggio

- **Throw Power:** La forza con cui viene spinto all'indietro un avversario che subisce un colpo da una palla lanciata dal giocatore

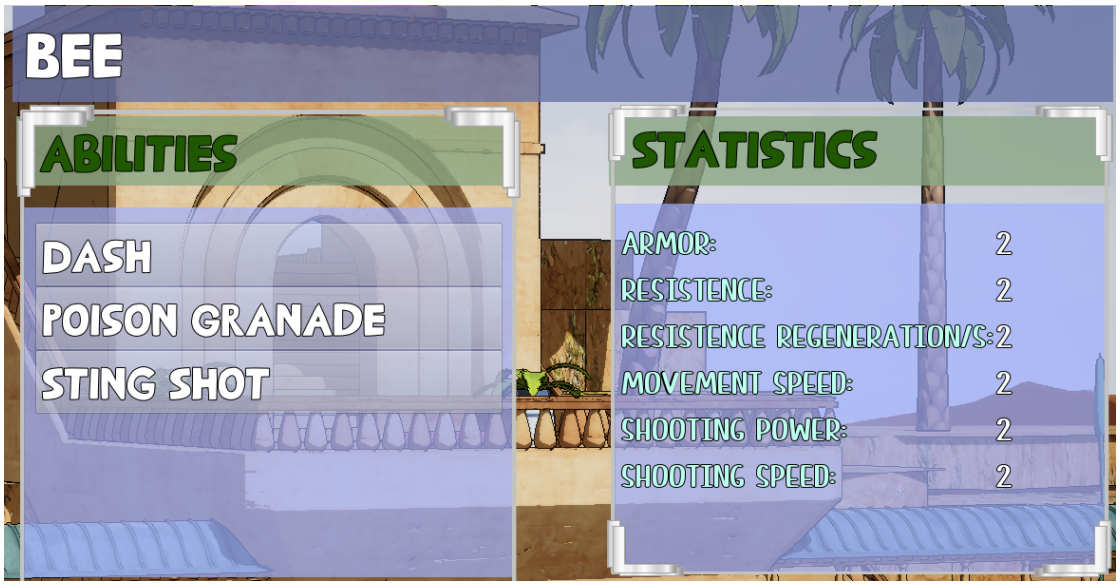


Figura 4.2: Statistiche e abilità del personaggio ispirato a un'ape

4.2.3 Collisione tra proiettili

Per spingere i giocatori a sviluppare al massimo i propri riflessi il team ha deciso di includere una meccanica che permetta di intercettare i proiettili della quadra avversaria con i propri. Ogni proiettile in gioco avrà associato un valore di danno e, a seconda della differenza tra gli oggetti che si scontrano, si avranno due scenari possibili:

- Il danno dei due colpi si equivale. Entrambi saranno distrutti dalla collisione senza altre conseguenze
- Il danno dei due colpi è differente. In questo caso viene distrutto solo il proiettile con il valore di danno più basso, mentre l'altro continuerà la propria traiettoria, con il proprio valore di danno decurtato di un valore equivalente a quello del proiettile distrutto

4.2.4 Le palle neutre

Durante ogni partita nel campo compariranno a intervalli irregolari delle palle, fino a un massimo di 6 in contemporanea, che serviranno ai giocatori per attaccare

la squadra avversaria anche senza la necessità di utilizzare le abilità dei propri personaggi.In questo modo vengono eliminati i tempi morti che potrebbero crearsi nel caso in cui tutti i giocatori usino le proprie abilità in un lasso troppo breve di tempo.
Per evitare che una squadra possa monopolizzare la proprietà delle palle evitando di lanciarle, ad ognuna di esse è legato un timer. Se una palla non viene raccolta e lanciata entro un tempo limite essa scomparirà, per riapparire immediatamente nella sezione avversaria del campo.

Figura 4.3: Una palla neutra non raccolta con relativo countdown

4.2.5 Stato del giocatore

Con lo svolgersi della partita, il subire o utilizzare le abilità può spingere il personaggio in una serie di stati che andranno a influenzarne la partita:

- **Stun:** Stato causato da alcune mosse. Impedisce al giocatore che incorre in questo effetto di muoversi e di usare abilità per una durata che dipende dalla fonte dell'effetto.

- **Iper stun:** Simile al normale stun, questo stato è provocato dal raggiungimento dall'azzeramento della resistenza. Le uniche differenze rispetto al normale stun sono la durata, sempre uguale a prescindere dalla mossa che lo provoca, e il fatto che tutte le forze subite dal personaggio avranno il valore raddoppiato.

- **Avvelenamento:** L'avvelenamento, per ora caratteristica peculiare delle mosse dell'ape, blocca completamente la rigenerazione della resistenza avversaria e ne provoca un calo periodico per un breve periodo di tempo.

4.3 Processo creativo

Il percorso che porta alla nascita di un videogioco è intrinsecamente diverso rispetto a quello degli altri software. Lo scopo non è risolvere un problema con l'ausilio della tecnologia o sfruttare nuovi settori dell'economia, ma è invece quello di creare un'esperienza unica ed emozionante per il giocatore. Nonostante questo obbiettivo abbia trovato molte risposte e declinazioni nel corso degli anni, il processo che porta all'ideazione di un videogioco segue sempre, in linea di massima, le stesse fasi.

4.3.1 Brainstorming

La creazione di ogni gioco parte da una fase di brainstorming. Tutti i membri del team creativo iniziano a raccogliere ogni tipo di idea che viene in mente a ognuno e iniziano a cercare un filo conduttore che possa raccogliere un sottoinsieme di queste idee in un unico progetto. Spesso capita, anche se non è obbligatorio, che questo processo parta già con alcuni limiti o necessità, siano essi tecnologici o creativi. Nel caso di Bugball esistevano già alcuni vincoli fin dall'inizio, legati per lo più alla necessità di sviluppare tre diverse tesi sugli ambiti di interesse di ciascun membro del team:

- Networking e multiplayer
- Data Collection
- Animazione e modellazione 3D

I primi due punti hanno velocemente orientato il team verso l'idea di un gioco multiplayer, possibilmente molto competitivo, per motivare la raccolta di dati al fine di equilibrare le sfide. Le necessità in ambito modellazione e animazione e il desiderio di creare una tesi che esplorasse l'ambito della motion capture hanno invece portato alla scelta di realizzare un gioco in 3D.
Da un'idea iniziale che vedeva gruppi di piccoli gnomi combattere a dorso di insetti come dei fantasiosi cavalieri, l'interesse del team si è pian piano avvicinato al concetto di insetti umanoidi che si scontrano in uno sport usando le proprie capacità. Da questa idea prende pian piano forma Bugball.

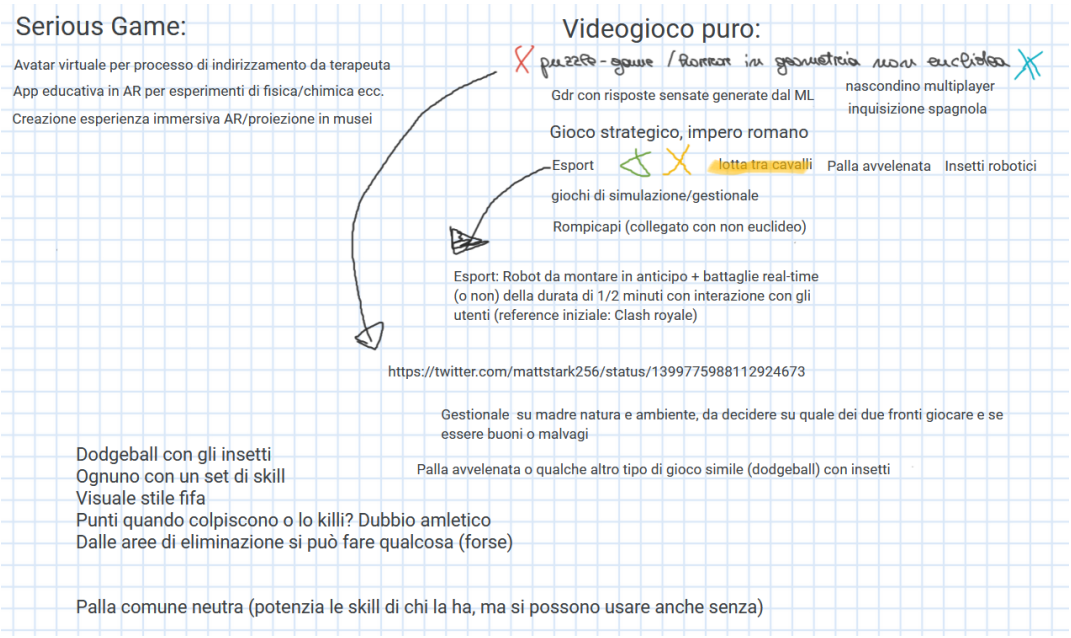


Figura 4.4: Lavagna virtuale usata nella fase di brainstorming

4.3.2 Raggiungimento del risultato per approssimazioni successive

Il processo di design non viene mai interrotto durante tutto lo sviluppo del gioco. Partendo dal prototipo cartaceo del gioco e arrivando alle prime versioni funzionanti della demo, il team di Game design non ha mai smesso di raffinare le meccaniche, apportando continue correzioni di rotta allo sviluppo. Molte idee sono state concepite e aggiunte al progetto e altrettante sono state eliminate per scremare il tutto dalle parti meno divertenti.
Un esempio di una funzione rimossa è l'attributo dell'energia. Prendendo ispirazione da molti giochi online, come League of Legends, si pensava di inserire questo valore come risorsa da spendere per l'utilizzo delle abilità, oltre al cooldown. Ci si è però resi conto, durante le prime fasi di prototipizzazione su carta, che rischiava di generare momenti morti, in cui nessuno sarebbe stato in grado di agire, al di fuori del lancio delle palle neutre. Per questo è stata eliminata, lasciando soltanto la caratteristica del cooldown a determinare il "costo" delle abilità.
Un esempio di funzione aggiunta è invece la possibilità di giocare in formati diversi dal 3 contro 3, che era l'unica modalità creata inizialmente. Durante i primi test completi sul prototipo digitale ci si è resi conto che per un giocatore avrebbe potuto essere complesso riuscire a trovare altri 5 utenti con cui giocare. Si è quindi pensato

che diminuire il numero di giocatori richiesti poteva essere un'ottima soluzione per aumentare la giocabilità.

Capitolo 5

Studio correlato

Una prima importante fase del processo di realizzazione è stata l'analisi delle proposte creative ideate dai designer. Questo passaggio ha permesso di individuarne le caratteristiche principali, e quindi analizzare le tecnologie disponibili per capire quali potessero essere le scelte tecniche migliori.
Nei punti seguenti verranno quindi analizzati i concetti più importanti nell'ambito del networking nei videogiochi moderni e, dove necessario, le scelte adottate all'interno del prototipo, in considerazione delle necessità di un gameplay molto veloce.

5.1 TCP vs. UDP

Uno dei primi problemi nel creare la libreria di networking per un videogioco è la scelta del protocollo di comunicazione per il livello di trasporto del modello OSI.
I due principali protocolli di trasporto, TCP e UDP, hanno caratteristiche molto diverse:

- TCP è un protocollo orientato alla connessione, che garantisce la ricezione affidabile e ordinata di un flusso di pacchetti, che al livello superiore dello stack di rete viene interpretata come un flusso continuo di byte. Queste azioni ovviamente possono provocare dei ritardi, in quanto in caso di rete inaffidabile la trasmissione deve essere ripetuta fino a che la ricezione non viene confermata

- UDP è un protocollo che aggiunge al sottostante livello IP soltanto le caratteristiche minime necessarie al livello trasporto ovvero la multiplazione attraverso l'aggiunta di una porta destinazione all'indirizzo IP e l'integrità dei dati nel pacchetto, attraverso un checksum (in realtà non molto affidabile, essendo di soli 16 bit). Viene quindi a mancare il concetto di connessione e ogni forma di

garanzia sulla ricezione dei dati inviati, ottenendo però una maggior velocità di trasmissione.

Ogni gioco ha delle caratteristiche diverse, quindi entrambi i protocolli sono utilizzati, ma per i giochi che prevedono update di stato molto frequenti e interazioni continue da parte dell'utente, UDP si è affermato come il miglior protocollo tra i due per una serie di motivi.
Innanzitutto molto spesso l'affidabilità della ricezione non è necessaria, in quanto molti tipi di aggiornamento sono ripetuti frequentemente (dalle 20 alle 60 volte al secondo) e i client sono in grado di approssimare il comportamento di molte componenti dei giochi per compensare le eventuali mancanze di piccole parti dell'informazione.
Anche l'ordinamento forzato della ricezione risulta essere un grande problema per molti giochi. Ritardare l'aggiornamento di qualche parametro all'ultimo valore ricevuto in attesa di informazioni su un suo eventuale stato precedente non ha nessuna utilità e rischia di provocare ritardi inaccettabili in caso di rete non affidabile.
Per questi motivi la libreria di networking di Unreal Engine utilizza UDP, al di sopra del quale è stata creato un nuovo protocollo che permetta la creazione di una vera e propria connessione e in cui affidabilità e ordinamento siano attivati solo su richiesta esplicita del programmatore, evitando quindi quell'overhead obbligatorio di TCP, che rischierebbe di rovinare l'esperienza del giocatore.

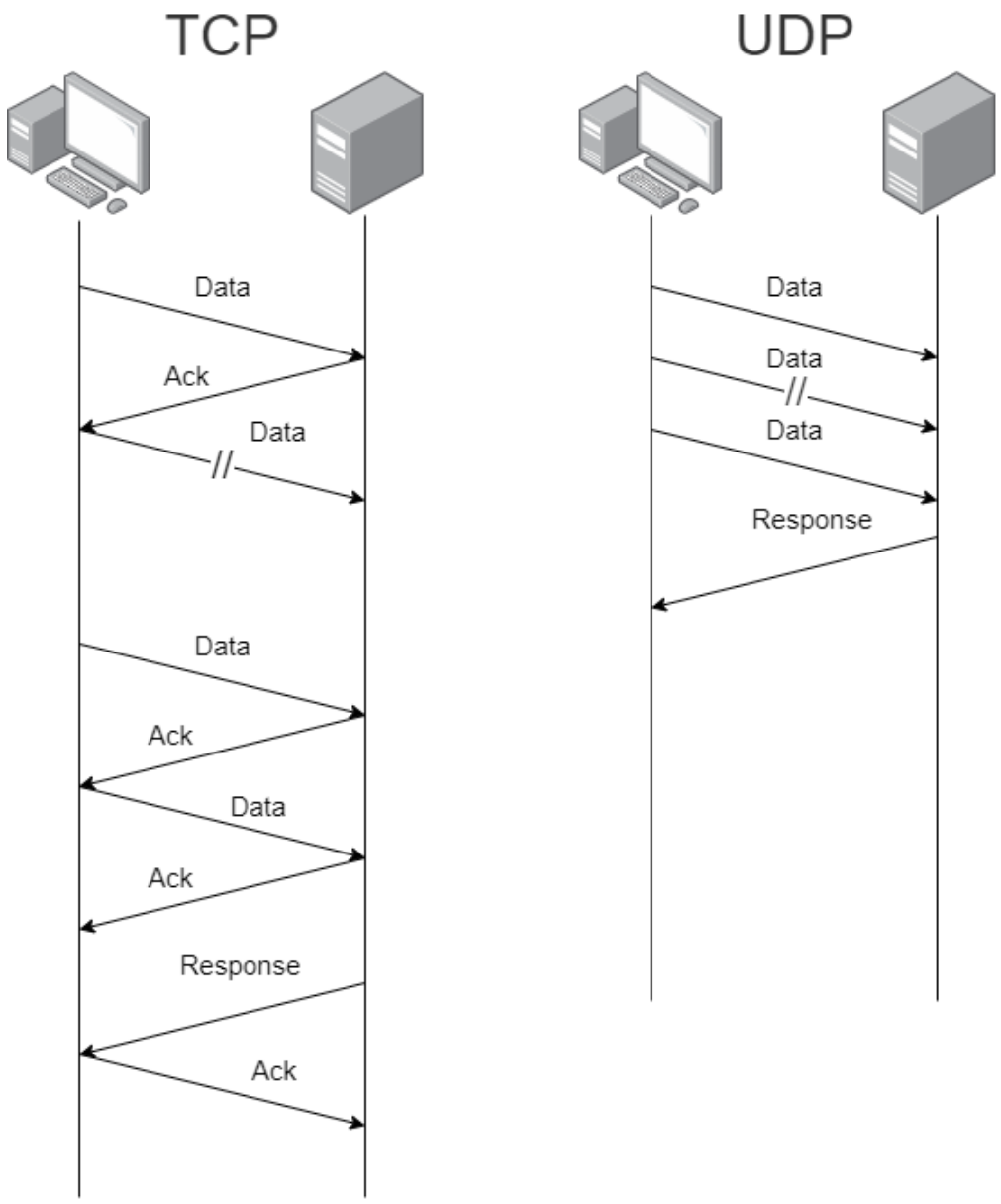


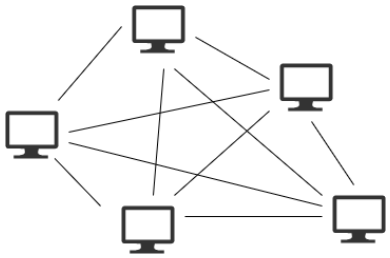
Figura 5.1: Confronto semplificato dei tempi di trasmissione tra UDP e TCP in caso di packet loss

5.2 Possibili architetture

Una volta definito e implementato il protocollo di comunicazione il passo successivo è la decisione dell'architettura dell'infrastruttura di rete.
Come visto nel capitolo sullo stato dell'arte queste possono in genere essere raccolte in 3 principali categorie, ognuna con i propri vantaggi e svantaggi.

5.2.1 Peer-to-peer

La prima tipologia di architettura a vedere la luce nel mondo videoludico. In questo tipo di connessione ogni utente ha esattamente lo stesso ruolo, comunicando a tutti gli altri utenti le informazioni che ritiene necessarie al funzionamento del gioco e ricevendo da tutti gli altri le informazioni per aggiornare la propria partita locale.



Pro:

- Molto semplice da creare. Non esistendo nessuna differenza tra il comportamento dei vari utenti, il programmatore si dovrà preoccupare della creazione di una sola versione del codice.

- Molto resistente agli errori e ai problemi di rete. In caso uno degli utenti interrompa il gioco all'improvviso o risulti irraggiungibile per problemi di rete, la partita può proseguire perchè ognuno degli altri partecipanti possiede tutte le informazioni necessarie.

- È molto economica per l'azienda che sviluppa il gioco, che non si deve preoccupare della creazione di un'infrastruttura di rete per la gestione delle partite, che viene totalmente lasciata in mano agli utenti.

- Permette di creare partite su reti locali, non connesse quindi alla rete internet, ovviamente a patto che tutti gli utenti abbiano la possibilità di accedervi fisicamente.

Contro:

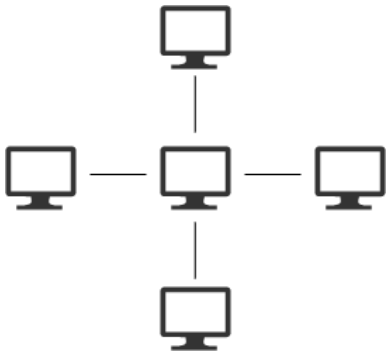
- La mancanza di un'autorità che gestisca la convalidazione dei dati inviati rende molto difficile evitare che qualche giocatore bari, inviando dati fraudolenti o modificando in locale i dati del gioco con il supporto di software esterni. Per questo motivo è quasi assente dall'ambito del gaming competitivo moderno.

- Il fatto che ogni utente debba comunicare con tutti gli altri singolarmente fa ìi che il numero di comunicazioni cresca esponenzialmente con l'aumento dei giocatori, arrivando a sovraccaricare le reti meno performanti. Inoltre, il fatto che non esista un'autorità centrale che smisti i dati solo agli utenti strettamente interessati, porta a trasmettere grandi quantità di informazioni superflue.

- La partita esiste solo nel momento in cui un giocatore decide di crearla e di iniziare una ricerca di eventuali compagni. Questa limitazione non pesa molto dal lato tecnico, ma non permette alcune scelte di game design, impedendo al mondo di gioco di esistere e modificarsi anche in assenza di giocatori, come succede in molti giochi di ruolo moderni online.

5.2.2 Client hosted

In questo tipo di architettura il client di un utente, generalmente quello che ha creato la partita, assurge a ruolo di controllore e gestore delle connessioni. Ogni client comunicherà soltanto con il client con autorità e riceverà soltanto da lui le informazioni per aggiornare la propria simulazione locale della partita.



Pro:

- Con l'aggiunta di un utente con un'autorità rispetto agli altri viene inserita una forma di controllo sulla plausibilità e validità delle comunicazioni dei client. Ovviamente il client che si occupa di verificare la validità dei dati non è sottoposto a nessuna forma di controllo, quindi questa struttura rimane comunque poco adatta al gioco competitivo.

- La difficoltà concettuale nella creazione del codice non aumenta in maniera sostanziale. Il progetto generalmente rimane lo stesso, con semplici controlli di autorità sull'esecuzione delle azioni, che definiranno un comportamento diverso tra client autoritativo e quelli privi di questa prerogativa.

- L'esistenza di un nodo centrale che funge da intermediario tra tutti gli altri limita il traffico di rete, che si riduce a una sola connessione in entrata e una in uscita per ogni client. Oltretutto il client autoritativo può anche effettuare operazioni di limitazione sulle informazioni inviate al singolo giocatore, evitando di informarlo sui cambiamenti degli elementi di gioco ininfluenti per la

sua simulazione locale della partita, diminuendo ancora di più le informazioni comunicate.

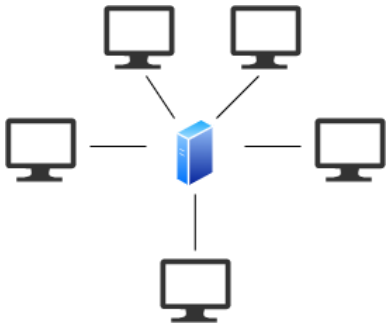
- Come l'architettura precedente, anche questa è sostanzialmente priva di spese per l'azienda produttrice.

Contro:

- Il client che ottiene l'autorità diventa un Single Point of Failure per l'intera architettura. Un malfunzionamento sulla sua macchina o nella sua connessione blocca l'intera partita a tutti gli altri giocatori.
- Il client adibito a host deve eseguire l'intera partita, senza poter filtrare le informazioni necessarie o da scartare per il proprio giocatore. Oltre a questo si aggiunge l'overhead di lavoro per gestire le connessioni con tutti gli utenti e per il controllo di validità sulle informazioni ricevute. Questo rende il gioco estremamente più pesante per un singolo utente, limitando quindi i tipi di gameplay a qualcosa che, dal punto di vista computazionale, sia gestibile da un singolo computer di uso comune.
- Tutto il traffico di rete si concentra su un unico giocatore, che solitamente non ha a disposizione una connessione di tipologia aziendale.
  Questo limita il numero di giocatori a valori piuttosto bassi, rimanendo nell'ordine delle centinaia.
- Permane il problema per cui l'esistenza del mondo di gioco è legata alla creazione della partita da parte di un utente.

5.2.3 Authoritative server

Questa architettura prevede lo stesso concetto di autorità di quella descritta in precedenza ma, a differenza di essa, il ruolo non viene conferito a uno dei client ma a una macchina totalmente separata, un server che si occupa di gestire l'intera partita.



Pro:

- Il controllo di validità e plausibilità sui dati ricevuti diventa una garanzia di legittimità delle mosse di tutti i giocatori (nei limiti di una buona implementazione da parte del programmatore dei necessari controlli).

Proprio per questo motivo questa è la soluzione usata da ogni forma di competizione nell'ambito dei videogiochi

- Mantenendo il concetto di un nodo centrale permane la possibilità di avere una gestione più precisa del traffico, limitando le comunicazione a quelle strettamente necessarie.

- Per quanto permanga il problema legato al Single Point of Failure, le infrastrutture a disposizione di un'azienda informatica sono molto più affidabili di quelle a disposizione di un'utenza privata, riducendo i problemi legati a queste fragilità.

- I server hanno a disposizione specifiche tecniche estremamente più elevate dei computer commerciali, oltre alla possibilità di avere più server adibiti alla gestione di più partite. Questo aumenta di molto sia il numero di possibili giocatori presenti in contemporanea nella stessa partita (toccando anche cifre di qualche migliaio) sia la complessità raggiungibile dal mondo di gioco.

- Il server può essere sempre attivo e disponibile, aprendo quindi la possibilità ai designer di creare stili di gioco in cui il mondo può continuare a evolversi anche in assenza di giocatori e in cui un giocatore può connettersi anche in assenza di altri utenti.

Contro:

- Aumenta di molto la complessità di sviluppo, con la definizione di due comportamenti profondamente diversi. Da un lato ci saranno i client, con ruoli molto ridotti, come ricevere e inviare gli input dei giocatori, aggiornare la simulazione locale con i dati in arrivo ed effettuarne il rendering e gestire con tecniche di approssimazione gli eventuali update mancanti.
  Dall'altro ci sarà il server, che sostanzialmente avrà in esecuzione una versione headless della partita (senza rendering grafico) che andrà ad aggiornare con gli input ricevuti dai client, una volta che questi siano stati verificati, per poi fornire aggiornamenti sullo stato a tutti i client connessi.
  A seconda dell'implementazione del networking nell'engine utilizzato potrebbe diventare necessaria la creazione di due progetti totalmente separati per i due ruoli di rete.

- Costi estremamente elevati e continuativi nel tempo per l'azienda produttrice, che dovrà creare e mantenere una struttura di server adeguata per tutto il tempo in cui prevede di rendere disponibile il proprio prodotto.

5.2.4 Conclusioni

Alla luce delle caratteristiche sopra elencate il team ha deciso di optare per una struttura a *server autoritativo*, quella che meglio si adatta all'idea di un gioco competitivo e che quindi necessita di forti controlli di validità sulle mosse di tutti i giocatori.

5.3 Sincronizzazione client server

Per quanto nascano dalle stesse tecnologie, la programmazione web e quella di un videogioco online hanno problematiche radicalmente diverse.
Mentre per l'applicazione di una banca o di un sito internet tempi di attesa nell'ordine dei decimi di secondo sono accettabili, in un gioco online questo tipo di ritardo tra input e reazione del software non è concepibile, in quanto renderebbe il gioco terribilmente poco reattivo. Per questo sono state sviluppate una serie di tecniche atte a mascherare gli inevitabili ritardi di comunicazione delle reti agli utenti e creare un'illusione di gioco in perfetta sincronia tra tutti i partecipanti.

5.3.1 Client prediction e server Reconciliation

La tecnica da cui prende nome questa sezione si basa sul concetto per cui, nonostante sia ovviamente possibile che l'utente stia cercando di barare, nella maggior parte dei casi gli input dell'utente saranno validi. Essendo di fatto un gioco un sistema deterministico, si concede al client la capacità di simulare in locale il movimento, senza dover prima farlo confermare dal server, mantenendo una buona sicurezza che, in mancanza di errori significativi o di azioni fraudolente, questa azione porterà allo stesso risultato ottenuto dal server. Il server si limiterà quindi a elaborare l'input ricevuto e a confrontare la posizione da lui calcolata con quella raggiunta dal client nella propria simulazione locale.
In caso le due posizioni combacino (con una piccola tolleranza dovuta a problemi che verranno analizzati più avanti) viene inviata una semplice notifica di conferma della simulazione.
Nel caso opposto invece, il server si occuperà di inviare una notifica di errore al client, corredata di timestamp e posizione ufficiale per il giocatore, che il client dovrà usare per aggiornare la propria simulazione alla posizione definita come corretta dall'autorità. Siccome a causa del RTT dei pacchetti è molto probabile che la correzione ricevuta non sia relativa all'ultimo movimento simulato, è necessario che il client mantenga una coda di input memorizzati e che, dopo aver effettuato le correzioni ricevute, le mosse successive siano rieseguite partendo dal nuovo punto di partenza. Questo è necessario per cercare di far combaciare le posizioni di client e server alla fine dell'aggiustamento ed evitare che questo procedimento venga

ripetuto molto spesso e da posizioni estremamente lontane, perchè al giocatore sembrerebbe di non avere il controllo del personaggio.
Il metodo con cui questo rewind delle azioni viene implementato sarà argomento di un approfondimento in un capitolo successivo, in quanto fortemente dipendente dalla tipologia di gioco realizzata.

5.3.2 Interpolazione

Fino a questo punto si è discusso solamente di come rendere fluida l'esperienza del giocatore con i controlli del proprio personaggio, ma un gioco online non si limita solo a questo. Esiste infatti anche la necessità di mostrare al giocatore il comportamento dei suoi avversari o compagni, rendendo quindi visibili le azioni degli altri client, cercando di mascherare il più possibile l'effetto del ritardo introdotto dalla comunicazione di rete.
Per spiegare questo concetto è necessario partire dalla base teorica di ogni game engine, il game loop.

```
1 while (IsRunning())
2 {
3     HandleInput();
4     Update(DeltaTime);
5     Render();
6 }
```

Ogni frame renderizzato all'interno di un gioco è dato dal susseguirsi di queste 3 azioni:

1. Vengono letti gli input da parte del giocatore e vengono quindi memorizzati in una struttura dati apposita

2. Lo stato interno del gioco viene aggiornato sulla base degli input del giocatore e della logica interna, basandosi sul tempo trascorso dall'update precedente

3. La situazione ottenuta dopo l'update viene renderizzata e mostrata a schermo

Sulla base di questa struttura verrebbe semplice pensare di trattare le informazioni del server in maniera simile agli input del giocatore e di usarlo per aggiornare le entità non controllate dal giocatore. Per quanto questa soluzione sarebbe di semplice implementazione si scontra però con una serie di problematiche molto serie che la rendono inutilizzabile. Per prima cosa non è detto che, per motivi già ampiamente trattati, la comunicazione da parte del server sia assicurata a un ritmo costante. A questo già importante problema si deve aggiungere anche il fatto che non è detto che il server esegua i propri update con lo stesso intervallo di tempo dei

client, rischiando di generare tempi di attesa troppo lunghi o brevi tra due update successivi.
Per questi motivi la maggior parte dei giochi opta per delle tecniche di interpolazione. Ogni volta che il server comunica una posizione legata a un determinato momento questa viene replicata dal client, che sposta in maniera concorde le entità nella propria simulazione, mentre nelle pause tra una comunicazione e l'altra simulerà il movimento di queste entità basandosi sulle informazioni relative ai movimenti precedenti. In presenza di perdite di pacchetti contenute, questa tecnica permette di rappresentare le informazioni puntuali ricevute come un movimento continuo e costante, funzionalmente quasi indistinguibile da quello effettuato dagli altri utenti sul proprio client.

5.3.3 Sincronizzazione della fisica

La fisica all'interno di un gioco online è un altro elemento che richiede un attento studio per permettere una sincronizzazione precisa tra i client e il server. La comunicazione dello stato di ogni oggetto soggetto alla fisica più volte al secondo genererebbe un flusso eccessivo di dati che rischierebbe di provocare un sovraccarico delle infrastrutture di rete, considerando che il numero di entità di questo tipo in una scena può facilmente raggiungere il migliaio.
Una tecnica per risolvere questa problematica può essere il Deterministic Lockstep, che consiste nel condividere via rete non le informazioni sulla simulazione della fisica in sè ma soltanto le informazioni relative agli input esterni. Questa tecnica rimuove quindi la necessità di inviare i dati di ogni entità cedendo al client la responsabilità di compiere i calcoli necessari alla propria simulazione. È importante notare che comunque la simulazione calcolata da ogni client è strettamente locale, quindi la modifica artificiosa dei dati in arrivo non andrà a modificare l'esperienza di nessun'altro utente.
Questa tecnica prevede però che il Physics Engine utilizzato sia strettamente deterministico, cioè che dagli stessi input porti sempre agli stessi risultati. Questo non è vero per alcuni motori fisici disponibili, ad esempio Open Dynamics Engine, un diffuso engine open source, che usa un generatore di numeri casuali per definire alcuni risultati delle collissioni, rendendolo ovviamente non deterministico. Altri, come PhysX (usato sia in Unity che in Unreal engine), offrono una forma di determinismo limitata, che prevede che le simulazioni, per poter combaciare, rispettino una serie di parametri molto stringenti come usare lo stesso intervallo di tempo tra gli step della simulazione e usare versioni dell'engine ottenute dallo stesso compilatore.

5.3.4 Problemi a virgola mobile

Purtroppo anche gli Engine deterministici sono basati sulla matematica a virgola mobile, che è suscettibile a piccole variazioni nei risultati a seconda dell'hardware di implementazione e del compilatore. Questo ha spinto team diversi a ricercare soluzioni diverse.
Alcuni hanno deciso di utilizzare soltanto compilatori che assicurassero un completo rispetto dello standard IEEE 754, andando però di fatto a limitarsi parzialmente il mercato delle console. Altre aziende hanno addirittura optato per reimplementare l'intera libreria di matematica a virgola mobile, per assicurare sempre e comunque risultati uguali su ogni architettura.
Per questioni di semplicità il team di Unreal Engine ha deciso di non creare nessuna soluzione ad hoc per questo problema, ripiegando invece su un sistema di sincronizzazione attiva attraverso la rete per tutti gli oggetti soggetti alla fisica che necessitino della corrispondenza perfetta tra client e server.

Capitolo 6

Networking

Nel sesto e nel settimo capitolo sono trattate nello specifico le componenti interne di Unreal usate per la creazione del prototipo. In questo nello specifico viene approfondita l'implementazione delle librerie di networking e il loro utilizzo da parte del team.
Unreal supporta nativamente solo le architetture a *Server autoritativo* o *Client hosted*, entrambe senza la necessità di creare progetti separati per la versione autoritativa e i semplici client. Per riuscire a raggiungere questo scopo il team di sviluppo dell'Engine ha dovuto implementare una serie di concetti, che permettessero di organizzare un eventuale codice di rete senza doverlo differenziare molto da quello di un semplice gioco in locale, mantenendo però una chiara distinzione tra il codice eseguito su client e su server.

6.1 Network lifecycle degli oggetti

Un primo importante concetto della programmazione multiplayer su Unreal è il posizionamento di ogni oggetto all'interno della struttura client-server.

- **Server only:** Gli oggetti di questo tipo esistono solo all'interno dell'istanza del gioco che viene eseguita sul server. L'esponente più importante di questa categoria è la classe GameMode, la classe che si occupa di gestire le regole fondamentali di ogni livello, le cui istanze devono per definizione esistere solo sul server

- **Server e Client:** Fanno parte di questa categoria tutti gli oggetti che esistono sia sul server che su ogni istanza dei client. Tra le classi standard di Unreal le cui istanze appartengono a questa categoria le più importanti sono GameState e PlayerState, che servono alla condivisione delle informazioni riguardanti la

partita, e Pawn, che serve a creare un qualsiasi elemento del gioco che riceva direttamente gli input di un giocatore attraverso un PlayerController.

- **Server e Client proprietario:** Questi oggetti esistono sul server e sul singolo client che ne ha il possesso. Un importante esempio di questa categoria sono i PlayerController, componenti che si occupano di ricevere gli input del giocatore, la cui esistenza è ovviamente necessaria solo sul client proprietario della singola Pawn.

- **Solo client:** Oggetti che esistono solo su un singolo client. Rientrano in questa categoria i componenti dell'interfaccia grafica e qualsiasi oggetto istanziato dal singolo client la cui creazione non sia effettuata attraverso una richiesta al server.

Per ogni sviluppatore è estremamente importante avere ben presente il posizionamento di ogni singolo oggetto all'interno dell'infrastruttura di rete, sia che si parli di classi interne di Unreal, sia che si parli di oggetti creati a runtime, perchè permette di creare una prima chiara divisione tra il codice del server e quello dei client.

6.2 Authority

Per poter definire comportamenti specifici per ognuno degli oggetti presenti su più macchine esiste invece il concetto di Authority.
Questo parametro permette di controllare se la macchina che sta eseguendo il codice ha effettivamente una forma di controllo sull'oggetto, potendone quindi modificare i valori, o se si tratta invece di un oggetto che sta semplicemente riproducendo per l'utente le informazioni ricevute dall'autorità centrale. I valori disponibili per questa variabile sono tre, di cui due esprimono declinazioni leggermente diverse dello stesso concetto:

- **ROLE_Authority:** Se la variabile authority ha questo valore la macchina ha il controllo totale dell'oggetto e qualsiasi azione venga effettuata è considerata valida ai fini del gameplay. Il fatto che la variabile abbia questo valore garantisce quindi che il codice sia eseguito sulla macchina che ha il controllo totale dell'oggetto. Questo ruolo è tipicamente caratteristico del Server, con la sola eccezione dei rari oggetti istanziati solo sul singolo client.

- **ROLE_SimulatedProxy:** Questo valore indica che il codice è in esecuzione su una macchina che non ha nessuna forma di controllo sull'oggetto, quindi di fatto che si ritrova a rappresentare i dati ricevuti dalla rete, implementando dei meccanismi di simulazione per mascherarne il ritardo. In genere questo

stato è visibile solo sui client, in quanto il server ha il controllo totale di ogni oggetto di cui conosca l'esistenza.

- **ROLE_AutonomousProxy:** Ruolo caratteristico degli oggetti posseduti da un PlayerController. Nonostante esprima comunque una mancanza di autorità nell'esecuzione del codice, gli oggetti che hanno questo ruolo sono ricevitori degli input del giocatore e hanno quindi la possibilità di modificare in maniera limitata la propria versione dei dati senza dover ricevere in tempo reale una conferma dal server.
  In pratica possiedono una piccola autonomia decisionale basata sugli input in ingresso che va a sostituire gli algoritmi di simulazione tipici del ruolo precedente.

Effettuando quindi controlli di questo valore è possibile evitare di compiere azioni in assenza di autorità, che porterebbero a modifiche locali non riprodotte per gli altri giocatori, differenziando di fatto il codice dello stesso oggetto senza bisogno di creare versioni diverse.

```
void ANetCharacter::OnHealthUpdate()
{
    //Client
    if (IsLocallyControlled() && GetRemoteRole() != ROLE_AutonomousProxy)
    {
        GEngine->AddOnScreenDebugMessage(Key -1, TimeToDisplay 5.f, FColor::Blue, DebugMessage FString::Printf(
                                         Fmt TEXT("Now you have %f of life"),
                                         CurrentHealth > 0.0f ? CurrentHealth : 0.0f));

        if (CurrentHealth <= 0.01)
        {
            GEngine->AddOnScreenDebugMessage(Key -1, TimeToDisplay 5.f, FColor::Red, DebugMessage FString::Printf(Fmt TEXT("You have been killed.")));
        }
    }
    if (GetLocalRole() == ROLE_Authority)
    {
        UE_LOG(LogTemp, Log, TEXT("%s now has %f health remaining."), *GetFName().ToString(),
               CurrentHealth);
        if (CurrentHealth <= 0)
        {
            Die();
        }
    }
}
```

Figura 6.1: Esempio di un controllo dell'Authority

L'immagine precedente è un chiaro esempio di codice differenziato con l'utilizzo di questa variabile. La funzione OnHealthUpdate() viene richiamata alla modifica del valore di salute di un giocatore sia su un client che sul server, ma nei due casi ha comportamenti profondamente diversi.
Il server, avendo la certezza della legittimità dei propri dati e di avere il diritto di modificarne i valori, si fa carico dei controlli sul risultato della modifica effettuata e degli eventuali update, rappresentati dalla funzione Die().
Il client invece, non avendo di fatto nessuna garanzia sull'accuratezza dei propri dati, si limita semplicemente a usare alcune funzioni di debug per visualizzare le informazioni sul monitor del singolo client.
È molto importante precisare che anche se il codice del client non fosse stato

differenziato, la funzione Die() avrebbe avuto effetti solo locali, senza propagarsi al resto della struttura di rete, creando comunque degli spiacevoli effetti locali in caso di manomissione dei dati ma salvando gli altri giocatori dal subire gli stessi errori.

6.3 RPC

Le RPC (acronimo per Remote Procedure Call) sono funzioni che permetto l'esecuzione di codice su istanze del gioco diverse da quella richiamante. Essendo una RPC il metodo di un Actor, è necessario che l'oggetto in questione sia replicato su ogni istanza in cui si ha intenzione di far eseguire la sezione di codice e la funzione selezionata dovrà essere preceduta, in fase di dichiarazione, dalla macro UFUNCTION() con uno dei seguenti parametri:

- **Client:** La funzione verrà eseguita solo dal client proprietario dell'oggetto da cui viene chiamata, sia che venga richiamata dal client stesso o dal server. Se richiamata da un client diverso dal proprietario l'esecuzione della funzione verrà bloccata.

- **Server:** La funzione verrà eseguita dal server sia che venga richiamata dal server stesso che dal client proprietario. Qualsiasi altro client che provi a richiamare questo tipo di RPC non porterà a nessun effetto sul server.

- **NetMulticast:** Usata dal server per richiamare un metodo su tutte le istanze dell'Actor, sia sui client che sul server stesso. Richiamare questa funzione su un qualsiasi client porterà alla semplice esecuzione in locale della funzione richiamata.

La dichiarazione di una funzione di questo tipo porterà alla creazione nel file .cpp collegato di una funzione con lo stesso nome seguito da "_Implementation", in cui andrà scritto il codice necessario alla RPC.
Esiste anche la possibilità di aggiungere il parametro "WithValidation" all'interno della dichiarazione di UFUNCTION, che porterà alla creazione di una funzione con lo stesso nome seguito da "_Validate" e con un boolean come tipo di ritorno. Questa funzione viene usata per eseguire controlli di validità prima dell'esecuzione, come controllare che il client sia davvero in grado di creare un nuovo proiettile prima di effettuarne lo spawn. Restituire il valore false porterà al blocco della funzione richiamata.
È importante notare che nessuna funzione dichiarata come RPC può avere un qualsiasi valore di ritorno, la comunicazione sarà sempre e solo unidirezionale.

6.3.1 Reliability

Come già esposto nei capitoli precedenti UDP si è imposto come standard per i giochi d'azione e Unreal Engine, vista la propria specializzazione in questo tipo di giochi, non fa eccezione.
Una RPC è quindi normalmente non affidabile, cioè la sua esecuzione non è garantita, ma ne è invece assicurata l'esecuzione nello stesso ordine di chiamata. Non sempre questo comportamento è accettabile, ad esempio lo spawn di un proiettile deve essere garantito per evitare di creare grossi problemi di gameplay. Per questo esiste un parametro specifico, *Reliable*, che impone l'esecuzione garantita della RPC utilizzata.
Durante lo sviluppo è molto importante tener presente che le chiamate a questo tipo di funzione possono intaccare pesantemente le prestazioni del gioco in presenza di una rete non affidabile, in quanto stabiliscono un sistema molto simile a TCP, che impedisce di proseguire la normale esecuzione del gioco finché non viene confermata la ricezione della richiesta. Un'altra questione molto importante è il fatto che per garantirne un'esecuzione ordinata di questo tipo di RPC il server memorizza le chiamate ricevute da ogni client in un'apposita struttura dati di dimensione fissa. Nel caso in cui questa struttura dovesse andare in overflow il client legato alla singola struttura dati verrebbe immediatamente disconnesso dalla partita, motivo per cui le chiamate a RPC Reliable vanno ridotte al minimo necessario per il corretto funzionamento del gioco.

```
UFUNCTION(Server, Reliable, WithValidation)
void ConfirmCharacterSelected(int ClassIndex);

void ADCharacterSelectionController::ConfirmCharacterSelected_Implementation(int ClassIndex)
{
    const ADPlayerState* DPlayerState = GetPlayerState<ADPlayerState>();
    ACharacterSelectionMode* Mode = Cast<ACharacterSelectionMode>(GetWorld()->GetAuthGameMode());
    if (Mode)
        Mode->ConfirmCharacterSelection(DPlayerState->GetUniqueId().ToString(), ClassIndex, DPlayerState->bIsSquadA);
}

bool ADCharacterSelectionController::ConfirmCharacterSelected_Validate(const int ClassIndex)
{
    return ClassIndex >= 0 && ClassIndex < 2;
}
```

Figura 6.2: Esempio di RPC affidabile e con controllo di validità

Nell'immagine precedente viene mostrata un esempio di RPC per cui l'affidabilità e il controllo di validità sono molto importanti. La funzione in oggetto si occupa di comunicare al server la selezione di un personaggio da parte del giocatore e la disponibilità a iniziare la partita. Una mancata ricezione della chiamata porterebbe a una differenza di stato tra client e server, con il secondo ancora in attesa di una conferma che il primo ritiene di aver già fornito. Il controllo di validità è necessario per garantire che il giocatore vada a selezionare uno dei personaggi disponibili e che non stia modificando tale valore in maniera non accettabile.

```
UFUNCTION(Server, Unreliable)
void SetCursorLocation(FVector Location);

void ADProjectPlayerController::SetCursorLocation_Implementation(FVector Location)
{
    CursorPosition = Location;
}
```

Figura 6.3: Esempio di RPC non affidabile

La funzione mostrata nell'immagine precedente è un buon esempio di RPC che non necessita dell'affidabilità della ricezione. Nello specifico la funzione trattata serve a comunicare al server, per ogni update, la posizione del mouse all'interno del mondo di gioco. Questo accade dalle 30 alle 144 volte al secondo, in base alle caratteristiche della macchina su cui avviene l'esecuzione. È facile intuire come una frequenza così alta di chiamate potrebbe facilmente sia bloccare l'esecuzione del gioco in presenza di una rete inaffidabile sia saturare le strutture dati del server.
Va inoltre notato che in così poco tempo è difficile che l'utente riesca a muovere il mouse di grandi distanze, quindi la perdita occasionale di un update della posizione può venir facilmente compensata dal valore della chiamata successiva o precedente.
Per questi motivi è stata scelta una RPC non affidabile, che non provoca nessun blocco nel caso di chiamate perse o non eseguite per l'eccessivo ritardo.

6.4 Replication

La replication è probabilmente la parte centrale e più importante all'interno della libreria di networking di Unreal. Con questo termine si intende l'atto, da parte del server, di condividere informazioni e dati con i client, partendo dalla creazione dei singoli Actor (unica classe la cui replicazione è implementata nativamente nell'engine) fino ai singoli valori delle variabili all'interno di quest'ultimi.
In poche parole l'intero sistema di sincronizzazione delle partite si basa su questa dinamica.

6.4.1 Replication degli Actor

Il programmatore ha la possibilità di personalizzare profondamente la replicazione di ogni singolo Actor usando una serie di parametri all'interno della definzione della classe, che permettono di decidere prima di tutto se esiste la necessità che l'oggetto venga replicato, quali client necessitino di una copia dell'oggetto e la frequenza degli update via rete.
All'interno di questi algoritmi esistono due flow particolarmente importanti, quello che porta alla decisione di quali oggetti abbiano bisogno di venir aggiornati sulla

rete e quello che nello specifico si occupa della replicazione di un singolo oggetto. Quello di decisione effettua, per ogni oggetto per cui sia attiva la replication, la seguente sequenza di azioni:

1. Determina se l'Actor è attivo, in caso contrario interrompe il processo.

2. Determina se in base alla frequenza sia necessario un aggiornamento del valore.

3. Determina per quali connessioni l'Actor sia rilevante e aggiunge un riferimento all'oggetto all'interno di una lista collegata alla connessione.

4. Per ogni oggetto che soddisfi i requisiti viene chiamata la funzione PreReplication, una funzione virtual nel cui override è possibile specificare le condizioni che si desiderano per la replicazione delle variabili interne.

Dopo queste azioni viene eseguito, per ogni connessione aperta, un ciclo sulla lista di oggetti ad essa associata, in cui viene controllato che ogni oggetto sia rilevante per quella connessione con il metodo IsNetRelevantFor() e che il client abbia già concluso di caricare il livello in cui l'oggetto deve essere inserito. Nel caso un oggetto risulti non rilevante per tempi troppo lunghi viene temporaneamente eliminato dagli oggetti legati a quella connessione, fino a che la situazione non richieda di nuovo l'inserimento di quell'oggetto
L'elenco degli Actor viene quindi ordinato in base alla priorità (approfondita nella prossima sottosezione) e, per ognuno di quelli che rientrano nel limite di oggetti aggiornabili in un singolo update, viene chiamato il metodo ReplicateActor. Per tutti gli eventuali oggetti rimasti esclusi da questo procedimento viene forzata una replicazione all'update successivo.
A questo punto è necessario spiegare il comportamento della funzione ReplicateActor() che si compone da 5 fasi principali:

1. Determina se l'oggetto è al primo update dal momento della sua creazione e, nel caso sia effettivamente così, serializza e invia le informazioni necessarie all'istanziamento dell'oggetto sui client. Di default vengono trasmesse solo le informazioni relative a posizione, rotazione e scaling ma possono esserne inserite altre in caso di necessità.

2. Verifica se la connessione possiede l'oggetto e decide in base a questo il corretto ruolo, se ROLE_AutonomousProxy o ROLE_SimulatedProxy.

3. Replica le variabili interne secondo le regole specificate dalla singola classe

4. Replica ogni proprietà dei componenti dell'Actor che richieda di venir replicata (e.g. I tag del Gameplay ability system che verrà approfondito nel prossimo capitolo)

5. Per ogni componente che sia stato cancellato viene inviato un comando di cancellazione

Alla fine di questo procedimento ogni client che necessiti di un Actor e dei suoi valori avrà disponibile in locale una copia dell'oggetto e delle sue variabili considerate rilevanti.

6.4.2 Relevancy e priorità

In un gioco online è importantissimo ottimizzare il flusso di dati in rete perchè comunicare una quantità eccessiva di informazioni può facilmente portare a rallentamenti del gioco o a un sovraccarico delle infrastrutture di rete.
Un elemento importante da notare è che molti giochi hanno livelli di grandi dimensioni nei quali le interazioni del giocatore con oggetti molto lontani potrebbero risultare impossibili. Per questo in Unreal esiste il metodo virtual IsNetRelevantFor(), che permette di controllare se un Actor necessita di venir replicato su una connessione. La definizione standard di questa funzione contiene i seguenti controlli che possono venir incrementati o sostituiti dal programmatore:

1. Se l'Actor è impostato per essere sempre rilevante, è sotto il controllo del proprietario della connessione o ha interagito con la Pawn controllata dal giocatore locale è considerato rilevante

2. Se il boolean bOnlyRelevantToOwner è impostato a true viene automaticamente considerato come non rilevante e scartato

3. Se l'Actor è nascosto, cioè ha il boolean bHidden settato a true, allora l'Actor sarà rilevante soltanto se è in collissione con la Pawn del giocatore

4. Se il gioco è impostato per usare la distanza come parametro per la rilevanza, qualsiasi oggetto oltre la distanza prefissata risulta non rilevante.

Questa serie di regole è piuttosto limitata, quindi potrebbe aver bisogno di estensioni per gameplay molto complessi, ma per un prodotto come quello sviluppato per questa tesi si sono rivelate più che sufficienti a limitare il traffico di rete. Esiste inoltre un sistema di priorità che permette di gestire la porzione di banda utilizzata per ogni singolo oggetto sulla base di un variabile float chiamata NetPriority. Più è alto questo valore, maggiore sarà la banda dedicata alla comunicazione di quel singolo Actor.

6.4.3 Replication delle variabili

All'interno di un Actor replicato molto spesso non è necessario che tutte le variabili vengano replicate sui client. Per questo ogni variabile che si desidera venga replicata

ha bisogno di una dichiarazione esplicita all'interno della macro UPROPERTY() con uno tra due parametri possibili:

- **Replicated:** La variabile verrà istanziata sia sul server che su ogni client che contenga una copia dello stesso oggetto.
- **ReplicatedUsing:** Lo stesso effetto della keyword precedente con l'aggiunta di una callback chiamata alla modifica del valore

Inoltre è possibile anche specificare il comportamento e la frequenza che si desidera per gli update del valore. Ogni Actor ha una funzione virtual, quindi sovrascrivibile, chiamata GetLifetimeReplicatedProps. In questa funzione è possibile specificare se e con quali condizioni si intende replicare una variabile.

```
UPROPERTY(Replicated)
bool bIsSquadA;
void ADPlayerState::GetLifetimeReplicatedProps(TArray<FLifetimeProperty>& OutLifetimeProps) const
{
    Super::GetLifetimeReplicatedProps([&]OutLifetimeProps);
    DOREPLIFETIME(ThisClass, bIsSquadA);
}
```

Figura 6.4: Esempio di variabile replicata senza condizioni

La variabile bIsSquadA all'interno della classe DPlayerState è un esempio di replication che non utilizza nessuna forma particolare di condizione ed esegue la replication continua del valore. Essa è utilizzata per memorizzare in maniera sicura e persistente la squadra di appartenenza di un giocatore.

```
UPROPERTY(Replicated)
FVector CursorPosition;
void ADProjectPlayerController::GetLifetimeReplicatedProps(TArray<FLifetimeProperty>& OutLifetimeProps) const
{
    Super::GetLifetimeReplicatedProps(OutLifetimeProps);
    DISABLE_REPLICATED_PROPERTY(ThisClass, CursorPosition);
}
```

Figura 6.5: Esempio di variabile replicata senza meccanismi di condivisione dei dati

Un caso particolare del progetto è quello della variabile CursorPosition, replicata ma priva di qualsiasi meccanismo di replicazione standard di Unreal. Questo è stato fatto perchè, nonostante la variabile debba esistere sia su client che su server, l'informazione deve seguire un percorso contrario a quello gestito di default dalla replication, cioè dal client al server.

6.5 Personalizzazioni e particolarità del sistema

Non sempre l'architettura strettamente autoritativa dell'infrastruttura è un bene per il gioco. Per quanto garantisca la regolarità di tutte le mosse, aspettare la conferma del server prima di eseguire qualsiasi azione finirebbe per generare una attesa di almeno due volte il RTT tra l'input e la reazione del personaggio. Questo ritardo, accettabile per giochi a svolgimento più lento, come ad esempio gli scacchi, è assolutamente inaccettabile per un gioco che faccia dell'azione il proprio cardine. Oltre a questo primo aspetto esistono anche situazioni in cui, a causa del leggero ritardo sulle informazioni ricevute, un giocatore potrebbe reagire a una situazione che sul server è già considerata conclusa mentre risulta ancora attiva dal suo punto di vista. Per cercare di non punire un giocatore a bassa latenza a causa dei problemi sistematici dell'architettura esistono politiche di contrattazione tra client e server apposite, che permettono di gestire questo tipo di situazioni.

6.5.1 Gestione standard del movimento

Una delle azioni per cui il problema dell'autorizzazione si fa più pressante è il movimento del personaggio sul client. In un gioco d'azione, in cui spesso ci si muove per evitare colpi del nemico o per colpirlo a propria volta, è inconcepibile inserire un ritardo di almeno 30ms tra l'invio dell'input e la sua esecuzione. Proprio per questo all'interno di Unreal è presente un componente chiamato CharacterMovementComponent che si occupa di compensare questo ritardo con una serie di algoritmi studiati appositamente.

Figura 6.6: Situazione iniziale

Per prima cosa al client viene concesso di simulare il movimento, cioè di muovere autonomamente la Pawn controllata dal giocatore. Alla fine di ogni movimento crea una FSavedMove_Character, cioè una struct in cui vengono memorizzate le informazioni del movimento appena compiuto, e questa viene aggiunta in una queue. Dopo un raggruppamento delle struct più simili nella queue queste vengono inviate al server con una RPC.

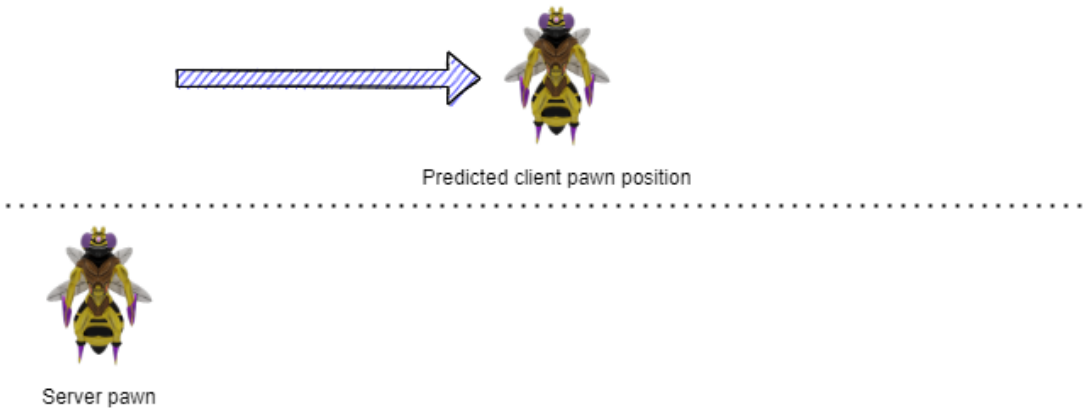


Figura 6.7: Situazione intermedia con la simulazione del client

A questo punto il server si occupa di ricalcolare il movimento del client a partire dalle mosse ricevute. Se la posizione finale combacia, a meno di piccoli errori, con quella comunicata dal client allora il server invia un messaggio di conferma della posizione.

In caso contrario viene utilizzata una RPC per inviare al client posizione, rotazione e stato corretti (walking, falling, etc...) corredate da un timestamp. A questo punto il client userà il prossimo update per resettare la posizione a quella confermata dal server.

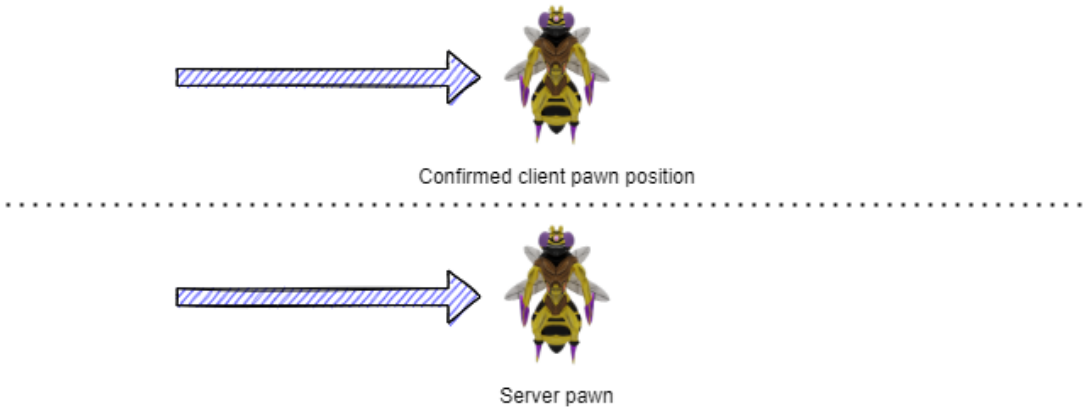


Figura 6.8: Movimento confermato dalla Pawn del server

È importante notare che il server non calcola i movimenti in tempo reale ma raccoglie invece le informazioni sul movimento da diversi client, ognuno con una latenza potenzialmente molto diversa da quella degli altri, per poi processarle tutte in contemporanea. Per questo le correzioni spesso non si riferiscono all'ultima mossa compiuta, ma ad una svolta in precedenza e già conclusa. Quindi, dopo la correzione, il client deve rieffettuare tutti i movimenti successivi a quello di cui non ha ricevuto la conferma, andando a recuperarne le informazioni dalla propria queue SavedMoves. Dopo questa ricostruzione manda una nuova richiesta di conferma al Server per ricevere la conferma della nuova posizione raggiunta.

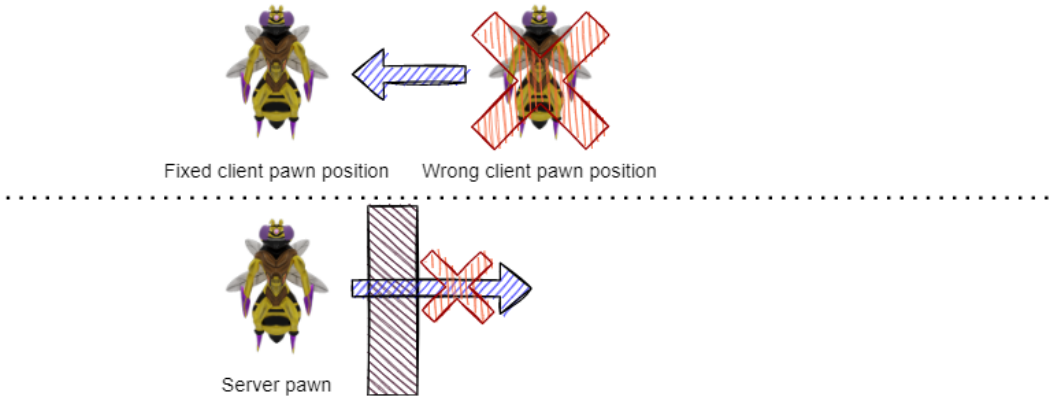


Figura 6.9: Movimento non confermato dalla Pawn del server con un reset alla posizione corretta

I client non proprietari della Pawn ricevono invece solo i dati delle posizioni confermate dal server, che una apposita funzione di smooth che si occupa di interpolare per generare l'impressione di un movimento continuo.

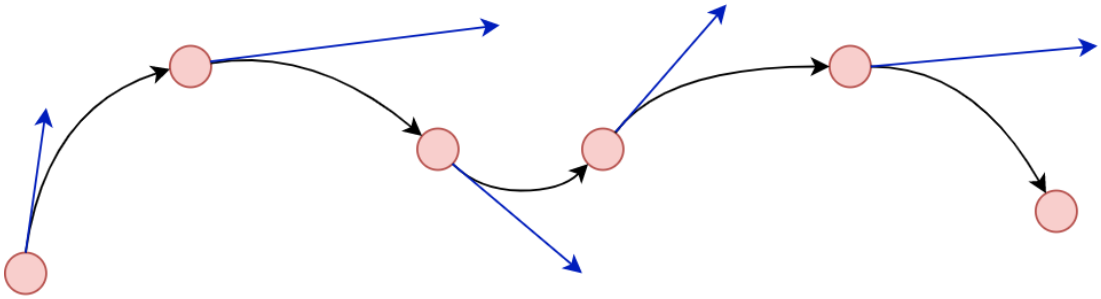


Figura 6.10: Esempio di interpolazione sulla base di una serie di punti (in rosso) e di velocità (in blu)

6.5.2 Personalizzazione dell'algoritmo di movimento

Il sistema di previsione descritto nel paragrafo precedente è ottimo per i giochi in cui l'input di movimento sia fornito dal giocatore sotto forma di direzione in cui ci si intende muovere. Il progetto in analisi ha però delle esigenze molto diverse, in quanto il movimento è gestito attraverso il sistema di navmesh a cui viene dato un punto da raggiungere. Bisogna quindi fare in modo che i percorsi calcolati da client e server combacino e che il sistema di correzione sia calibrato per funzionare ricostruendo tali percorsi.
Per raggiungere questo risultato sono state testate due strade diverse, di cui alla fine è stata scelta la seconda. La prima, per quanto funzionale, aveva grosse problematiche legate al delay di comunicazione, che la rendevano particolarmente spiacevole da usare per l'utente finale.

6.5.3 Prima soluzione: richiesta al server

La prima soluzione prevede l'utilizzo della navmesh di Unreal soltanto sul server, per evitare di dover inserire meccanismi di correzione e riconciliazione, particolarmente complicati. Ci sono quindi, per ogni giocatore, 4 elementi:

- Un PlayerController, che si occupa di ricevere l'input del giocatore riguardo la posizione da raggiungere
- Un AIController, presente solo su server, che riceve l'indicazione della posizione da raggiungere da parte di una RPC nel PlayerController
- Un Character, controllato dall'AIController, che viene replicato su ogni client
- Una Pawn, sotto controllo del PlayerController, che segue i movimenti del Character con una Camera di rendering sempre puntata su di esso

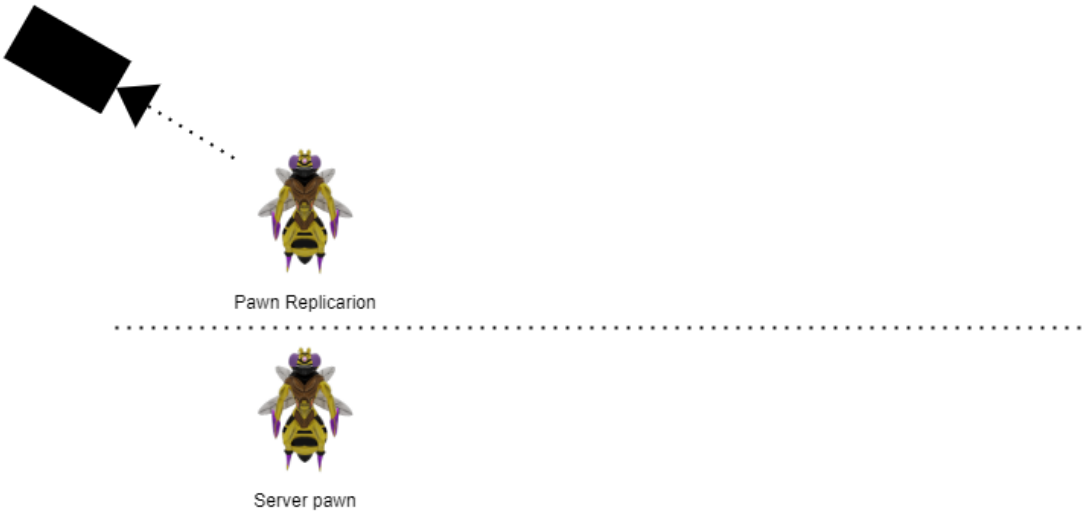


Figura 6.11: Situazione iniziale

Ogni volta che il client comunica la necessità di un movimento, la destinazione viene inviata all'AIController, che calcola i punti da seguire e muove il Character di conseguenza. A questo movimento segue una replicazione della velocità, imitando sul client "proprietario" del personaggio un evento simile a quello che nel movimento standard viene eseguito sugli autonomous proxy.

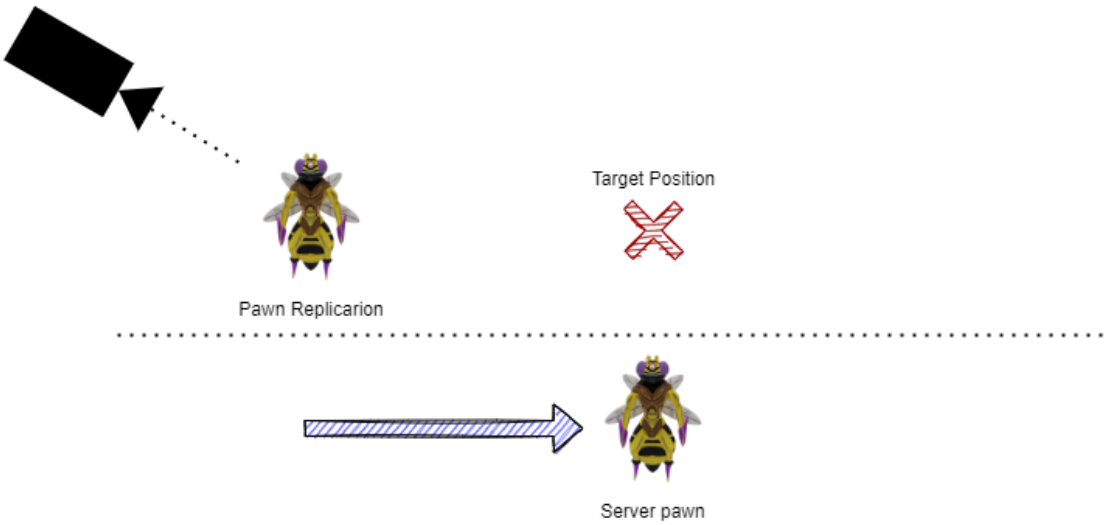


Figura 6.12: Il server esegue il movimento verso la posizione comunicata via RPC

L'oggetto Camera, che esiste solo sul server,inizia a seguire i movimenti replicati della Pawn, permettendo al server di mantenere questa al centro della propria visuale.

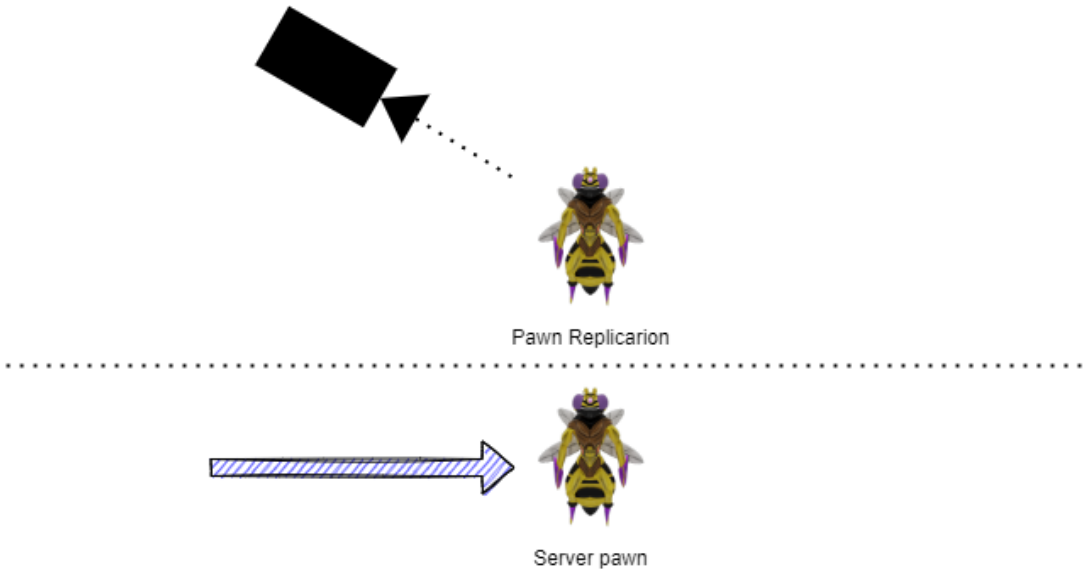


Figura 6.13: La replica della Pawn raggiunge la posizione dettata dal server
Come si può facilmente immaginare, a ogni input corrisponde un ritardo di due volte il RTT più l'overhead di calcolo del server prima dell'esecuzione del comando. Per questo motivo, per quanto semplifichi la gestione del movimento, questa soluzione è stata scartata dopo le prime sessioni di testing del progetto.

6.5.4 Seconda soluzione: previsione del client

Il primo passo di questa soluzione è quello di aggiungere una RPC alla richiesta di movimento che permetta di condividere con il server la destinazione desiderata, in modo da poter effettuare il calcolo del percorso su entrambe le macchine. Questo serve per evitare un delay dell'azione, in quanto il client avrà immediatamente i dati sulla base di cui si muoverà, mantenendo comunque una supervisione da parte dell'autorità centrale.
A questo punto ci sono due scenari possibili, in uno il percorso calcolato dalle due istanze combacia, nell'altro ci sono differenze tali da necessitare di correzioni in corso d'opera. Nel primo caso semplicemente le due unità, avendo una velocità costante, si muovono lungo lo stesso percorso con le stesse tempistiche, non necessitando quindi di nessuna forma di correzione.

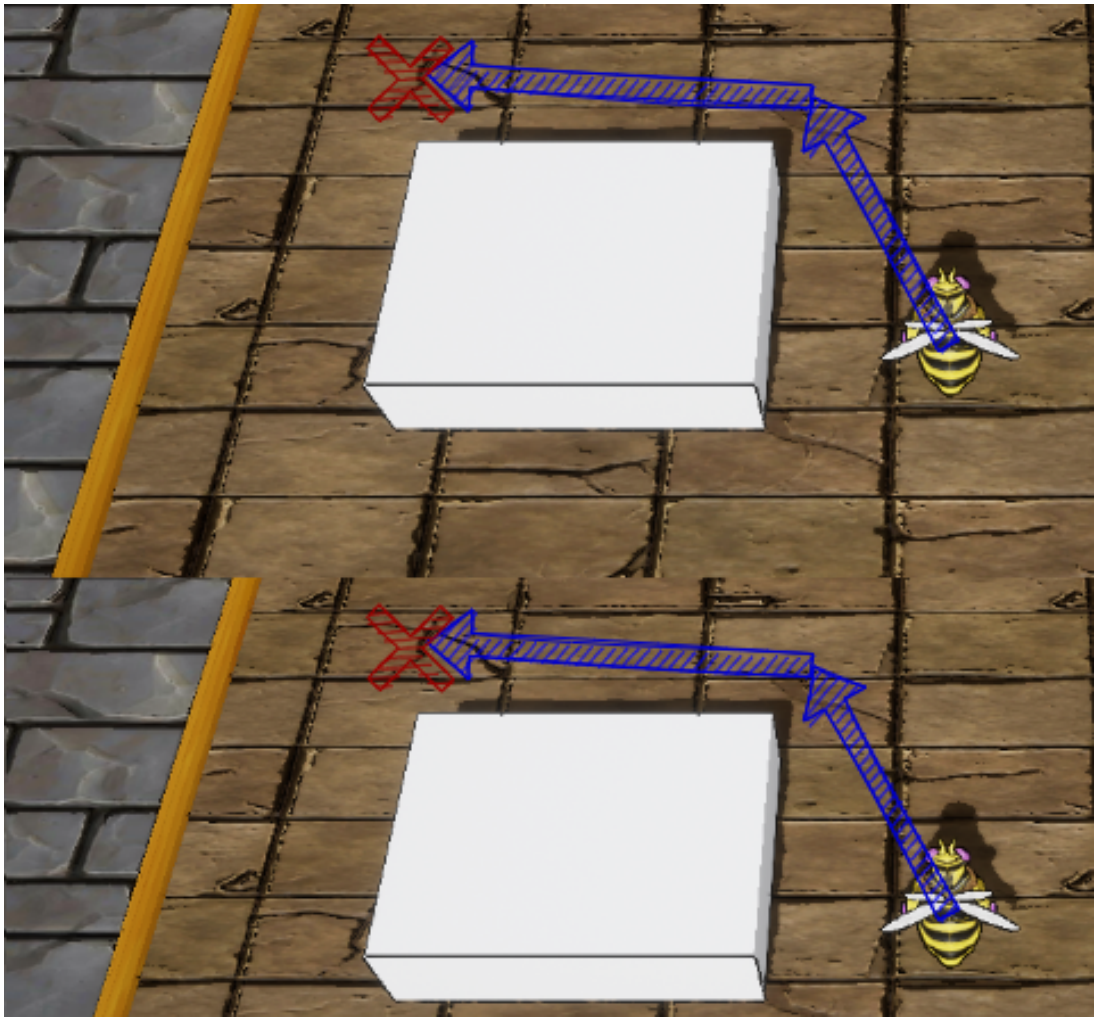


Figura 6.14: Esempio di percorso combaciante tra client e server

Al contrario, nel secondo caso, i due percorsi non sono abbastanza simili da considerare valido quello scelto dal client. A questo punto interviene, come nel caso precedente, una RPC richiamata dal server che comunica al client il movimento errato, con l'aggiunta alle informazioni standard dell'array dei punti del percorso scelto dal server.

Il comportamento del client è invece molto diverso rispetto al movimento standard di Unreal. Dopo aver ricevuto la correzione inizia un ciclo in cui controlla ogni entry nella coda delle mosse effettuate (anche questa struct sarà a sua volta sovrascritta da una nuova versione, con in aggiunta l'informazione del punto finale del percorso durante quella mossa):

1. Se il punto finale del movimento salvato e della correzione combaciano (a meno di minuscoli errori che rientrino nei limiti ritenuti accettabili) si muove di una distanza pari a $deltaTime\_della\_mossa * Speed$ lungo il percorso ricevuto dal server.

2. Se la destinazione della mossa salvata non combacia con quella del percorso ricevuto nella correzione ed è diversa da quella della mossa precedente allora viene generato un nuovo percorso e la Pawn viene mossa seguendolo.

3. Se invece la destinazione della mossa salvata non combacia con quella del percorso ricevuto ma è uguale a quella della mossa precedente viene utilizzato il percorso già creato nella ricostruzione precedente.

A questo punto, come nell'algoritmo standard, la nuova mossa viene comunicata al server per ricevere la conferma definitiva.

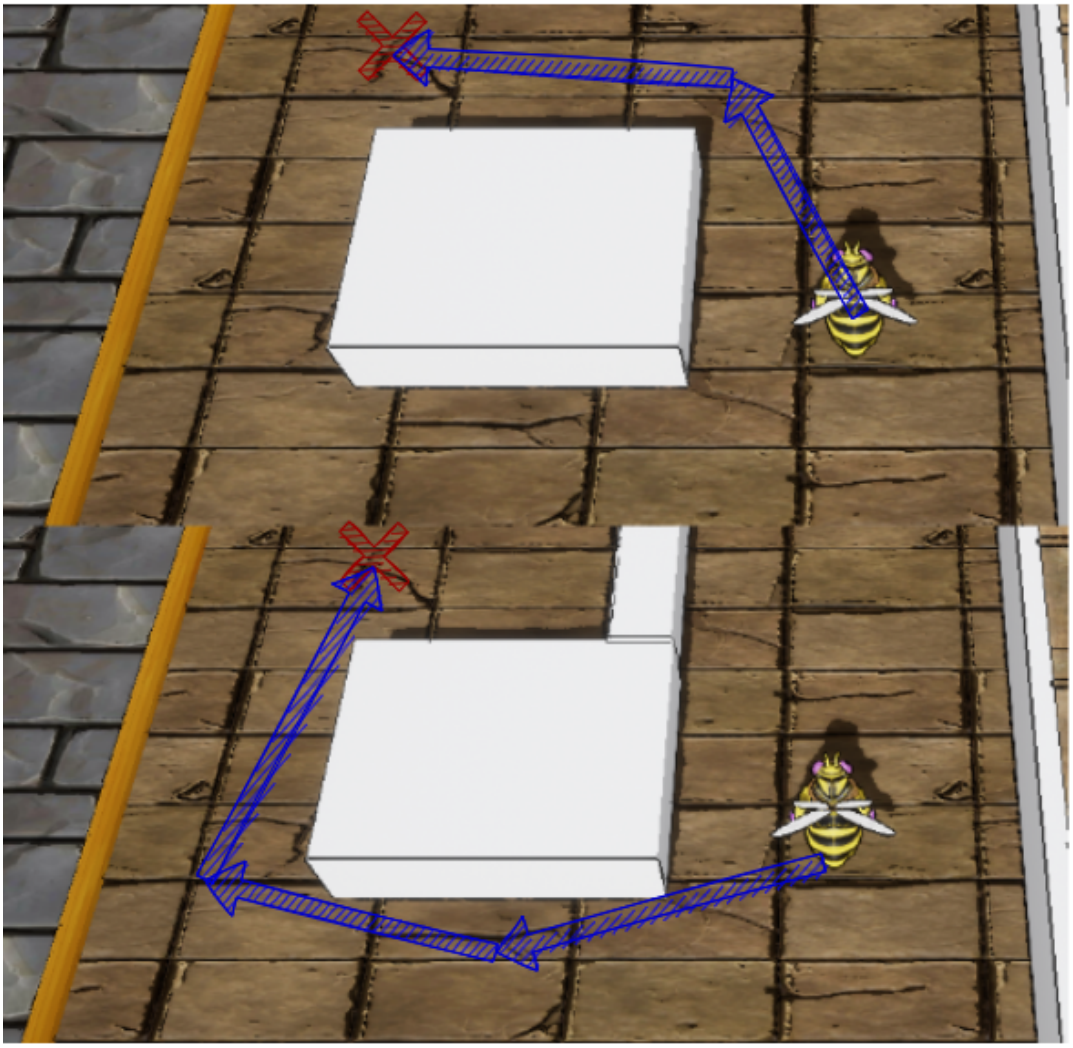


Figura 6.15: Esempio di percorso diverso tra client e server a causa di un Actor non ancora sincronizzato (il muro bianco)

6.5.5 Prediction e traiettoria dei proiettili

Molti giochi che prevedono la creazione di un alto numero di proiettili in breve tempo sfruttano dei sistemi di prediction per quanto riguarda lo spawn e il moto dei proiettili.
In maniera simile alle funzioni del movimento viene concesso al client di effettuare lo spawn e il movimento dei proiettili in locale, richiedendo però conferma di ogni mossa al server. In mancanza di conferma l'evento sarà annullato o corretto, mentre

nel caso avvenga una conferma il client avrà già iniziato a gestire il proiettile, senza attendere un'azione diretta del server.
Il progetto in sviluppo non rientra a pieno nelle caratteristiche appena descritte e si è quindi deciso di optare per un sistema più semplice. Il proiettile viene creato e replicato a partire dal server, a prezzo di un piccolo delay, ma questo garantisce che le informazioni iniziali del moto tra client e server combacino alla perfezione. Si vene così a creare un meccanismo di Deterministic Lockstep in cui le due simulazioni, essendo deterministiche al di fuori dei piccoli errori legati ai calcoli in virgola mobile, combaceranno in tutti i movimenti, essendo generate dallo stesso input. Questo elimina totalmente il bisogno di comunicazioni per la correzione del movimento calcolato dal ProjectileMovementComponent, il componente responsabile del moto dei proiettili all'interno di Unreal Engine.

6.5.6 Collisione dei proiettili

La collisione tra un proiettile e un giocatore è uno dei pochi altri casi oltre al movimento in cui non viene rispettato in maniera assoluta il paradigma del server autoritativo. In particolare per ogni collisione possono esserci tre situazioni diverse:

- La collisione viene rilevata sia dal server che dal client.
- La collisione è rilevata solo dal server.
- La collisione è rilevata solo dal client.

I primi due vengono gestiti allo stesso modo, cioè la collisione viene validata e il server ne valuta le conseguenze all'interno della partita. Il terzo caso invece segue un algoritmo più complesso. Questo viene fatto perchè non è considerato corretto punire un giocatore a bassa latenza (valori prossimi o inferiori ai 25ms) per aver agito in maniera sensata dal proprio punto di vista. Una volta accertata la collisione da parte del client si seguono i seguenti passi:

1. Viene controllata la legittimità dell'esistenza e della posizione del proiettile. Se una delle due risulta non corretta la collisione non viene registrata.
2. Viene verificata la latenza media del giocatore nell'istante della collisione. Se è oltre una soglia limite l'impatto viene invalidato.
3. A questo punto si controlla il comportamento del giocatore colpito. Se era in corso un qualche tentativo di evitare il danno (abilità di movimento, scudi, nascondersi dietro degli oggetti, ecc...) la collisione non viene registrata.
4. Se tutti i passaggi precedenti vengono superati allora si procede a convalidare la collisione anche se l'urto non si è verificato sul server.

6.6 OnlineSubsystem

Oltre alla sincronizzazione della partita un network programmer ha un'altra importante responsabilità, creare il sistema che permetta ai client di trovare delle partite disponibili e di unircisi. Questa funzionalità prende il nome di OnlineSubsystem all'interno di Unreal.

6.6.1 Session Interface

Una sessione rappresenta un'istanza del gioco in esecuzione su un server (o su un client, in caso di una struttura Client Hosted) e che sia disponibile a ricevere connessioni da parte dei giocatori. Ogni sessione ha un ciclo di vita composto dalle seguenti parti:

1. Viene creata la sessione e vengono specificate le sue caratteristiche (se Lan o Online, numero di giocatori ammessi, connessioni solo su invito o aperte a tutti, ecc...). Oltre alle caratteristiche standard è possibile estendere la classe delle impostazioni per includere eventuali informazioni specifiche per il proprio gioco.

2. Il server rimane in attesa dei giocatori. Durante la connessione di un giocatore sono disponibili tre metodi della classe GameMode che permettono di gestire i controlli e le meccaniche legate al login:

   - **Login:** Chiamata dal client per tentare di connettersi a una sessione. Essendo ancora non connesso a un server l'oggetto GameMode è ancora presente sul client, permettendo di richiamare questa funzione. Se la connessione ha successo il metodo restituisce un puntatore al PlayerController del giocatore, altrimenti restituirà un NULL.
   - **PreLogin:** Chiamata sul server, questa funzione deve implementare tutti i controlli necessari a evitare le connessioni fraudolente. Popolare la stringa ErrorMessage, ricevuta come reference, provocherà il rifiuto della connessione in ingresso.
   - **PostLogin:** Prima funzione richiamata dal server alla conclusione di una connessione. A partire da questa funzione è possibile richiamare le funzioni di replicazione sul PlayerController del giocatore.

3. Raggiunta una determinata condizione, che sia il numero di giocatori connessi, la scadenza di un timer o altro ancora, la sessione viene avviata e con lei la partita. A seconda del funzionamento del gioco potrebbe ancora essere possibile unirsi alla partita, o per riempire slot lasciati vuoti dalle disconnessioni di altri giocatori o per diventare spettatori.

4. Raggiunta la fine della partita la sessione viene terminata.
5. Ogni giocatore viene disconnesso, tornando a essere un'istanza isolata del gioco.
6. A questo punto ci sono due opzioni, o la sessione viene aggiornata e rientra in attesa di nuovi giocatori, o viene distrutta, a seconda delle necessità.

La ricerca di una sessione viene effettuata attraverso il metodo FindingSession, che accetta come input i parametri secondo cui deve venir svolta la ricerca. Per rendere la chiamata non bloccante per la durata della ricerca, questa viene effettuata in maniera parallela al normale update di gioco, con un delegate apposito che viene richiamato una volta che la queue di sessioni disponibili viene popolata. La logica che viene utilizzata per la ricerca della nuova sessione dipende dalla tipologia di subsystem utilizzata, con Epic games che ne mette a disposizione alcune già pronte, basate sui più diffusi distributori di servizi per i prodotti videoludici.

6.6.2 Spostarsi tra i livelli

In genere, all'interno di ogni sessione, vengono più volte eseguiti spostamenti tra differenti livelli di gioco. Unreal mette a disposizione tre diverse funzioni per gestire tipologie diverse di movimento:

- **Browse:** Permette al server di cambiare livello, disconnettendo tutti i client connessi in quel momento.
- **ServerTravel:** Utilizzabile solo dal server, permette al server di cambiare livello trasportando con se tutti i client connessi. È il metodo principale con cui si cambia il livello nei giochi online. Questo spostamento può avvenire in maniera seamless e non. Nel primo caso si permette ad alcuni Actor di non venir distrutti tra un livello e l'altro, mentre nel secondo caso tutti gli Actor vengono reinizializzati. Il primo risulta molto più rapido e fluido, ma in genere non può venir utilizzato nel caso sia la prima volta che il livello viene caricato dai client.
- **ClientTravel:** Utilizzabile sia dai client che dal server. Dai primi viene usato per connettersi a un nuovo server a partire dalle sue informazioni di rete, mentre il server può utilizzare il metodo per cambiare livello a uno specifico client, pur mantenendolo connesso a se stesso

6.6.3 OnlineSubsystemSteam

L'OnlineSubsystemSteam è una versione dell'OnlineSubsystem che, pur mantenendo gli stessi metodi, basa il proprio funzionamento sulle API di Steam, il famoso

launcher della Valve Corporation.
Questo permette di utilizzare i servizi interni di Steam per connettere diverse partite anche non presenti sulla stessa rete, e, attraverso apposite impostazioni di debug, queste funzioni sono disponibili gratuitamente per chiunque desideri sviluppare un gioco con queste caratteristiche.

6.6.4 Matchmaking

Molti giochi competitivi moderni non si basano più solo su un sistema di connessione diretta tra i giocatori e le partite. Ogni giocatore che intenda connettersi a una partita si rende disponibile comunicando questo suo stato a un server di matchmaking, in genere diverso da quello su cui viene giocata la partita. Questo server, basandosi sulle statistiche del giocatore e sul tipo di partita richiesta, cercherà un insieme di giocatori in cui inserire il nuovo arrivato, al fine di generare delle squadre in cui il livello di abilità sia equilibrato. Una volta raggiunto il numero di giocatori necessario viene attivata una sessione sul server di gioco e viene comunicato a ogni client l'indirizzo a cui connettersi per avervi accesso.
Mentre i server di gioco sono generalmente gestiti internamente dalle aziende videoludiche, al giorno d'oggi molti servizi di matchmaking vengono gestiti da aziende esterne, che hanno sviluppato ottimi strumenti di analisi dei dati e che gestiscono facilmente l'enorme mole di giocatori che esegue operazioni di questo tipo in contemporanea. Un esempio di questa tipologia di gestione lo si può trovare in Fortnite, gioco della stessa Epic Games, che gestisce il matchmaking attraverso i servizi di AWS.

6.6.5 OnlineSubsystem in Bugball

Essenzialmente all'interno di Bugball l'utilizzo dell'OnlineSubsystem (nella sua declinazione basata su Steam) è molto ridotto. Essendo una demo tecnica senza una grande base di utenza il sistema di matchmaking è completamente assente, com'è assente un sistema di login che permetta al giocatore di avere un'identità univoca e dati persistenti tra diverse partite. L'utilizzo si limita quindi alla ricerca di sessioni disponibili, rese visibili attraverso le interfacce offerte dai servizi di Steam.

Capitolo 7

Gameplay Ability System

Il Gameplay Ability System (GAS in breve) è un plugin per Unreal Engine sviluppato dalla stessa Epic games per un proprio gioco, Paragon, e ampiamente utilizzato in molti altri giochi AAA in commercio.
Questo plugin permette una creazione rapida e veloce di un set di abilità per un personaggio giocante e di gestire un set di attributi e di effetti che li influenzi. Oltre a questo contiene anche una serie di meccanismi di replication, prediction e sincronizzazione che sollevano il programmatore da molti problemi nella creazione di un gioco multiplayer online.

7.1 Struttura interna

Il GAS si compone di un insieme di componenti e oggetti per gestire ogni aspetto delle abilità, ognuna con degli scopi specifici.

7.1.1 AbilitySystemComponent

Si tratta del component che si trova alla base del funzionamento dell'intero plugin. Qualsiasi Actor abbia bisogno interagire con il sistema, quindi che abbia bisogno di possedere delle abilità o di avere degli attributi associati, deve avere un component di questo tipo associato e implementare l'interfaccia IAbilitySystemInterface. Quest'ultima contiene un solo metodo virtual che permette di restituire l'AbilitySystemComponent interno sotto forma di puntatore.
All'interno di ogni AbilitySystemComponent è possibile specificare il comportamento per quanto riguarda la replication degli attributi e degli effetti:

- **Full:** Tutti i dati presenti presenti nell'AbilitySystemComponent su server vengono replicati su tutti i client. Generalmente non molto utile, a meno di

giochi singleplayer, perchè genera volumi di traffico troppo elevati condividendo informazioni che spesso non sono utili alla maggior parte dei client.

- **Mixed:** I Gameplay Effect con le relative modifiche agli attributi vengono replicati soltanto sul client proprietario dell'Actor. Su tutti gli altri client che possiedono una copia dell'oggetto vengono replicati soltanto i GameplayCues e i GamplayTags. È lo standard per qualsiasi Actor che venga controllato direttamente da un giocatore all'interno di una partita multigiocatore.

- **Minimal:** Vengono replicati soltanto i GameplayCues e i GameplayTags mentre i GameplayEffects saranno limitati al solo server. È la modalità consigliata per tutti quegli oggetti il cui comportamento sia legato a una AI, che all'interno di Unreal Engine possono esistere solo lato server, perchè permette di ridurre al minimo le necessità di comunicazione con i client.

7.1.2 GameplayTags

I GameplayTags sono delle stringhe che vengono inserite all'interno dell'AbilitySystemComponent di un Actor per descriverne lo stato. Hanno una struttura gerarchica, in cui ogni tag appartiene a una serie di sottoclassi, i cui nomi vengono separati da punti all'interno della stringa con la seguente struttura "Parent.Child.Grandchild".
Ad esempio, un tag del tipo "State.Debuff.Stun" specifica che il giocatore che lo ha all'interno del proprio AbilitySystemComponent è stordito e che questo effetto rientra nella categoria dei debuff (depotenziamenti), a sua volta parte della classe degli stati applicabili.
Ogni tag all'interno dell'AbilitySystemComponent è memorizzato in una classe apposita chiamata TagMap, che permette di effettuare controlli in maniera ottimizzata, e che affianca a ogni entry il conteggio delle istanze presenti, perchè il tag potrebbe essere stato applicato più di una volta. È possibile legare al cambiamento di ogni tag un delegate, in modo da reagire ai cambiamenti del suo conteggio con una funzione apposita. Ognuna di queste funzioni deve avere come parametri un FGameplayTag e un intero, per sapere quale tag è stato modificato e qual'è il conteggio delle istanze dopo la modifica.

Figura 7.1: Overview parziale dei tag del progetto

7.1.3 Attributes

Un attribute del GAS è essenzialmente una coppia di float salvata all'interno della struct chiamata FGameplayAttributeData. Questi due float servono a memorizzare qualsiasi statistica di un Actor che abbia bisogno di un valore numerico, dalla vita rimanente alla velocità di movimento. Ogni attribute ha due valori perchè viene memorizzato sia il valore corrente che il valore di base, generalmente il valore di partenza.
Una tecnica di programmazione molto utilizzata è anche la creazione di Meta Attributes, usati per memorizzare temporaneamente valori che debbano andare a modificare altri attributi.
Questo permette di separare logicamente l'attribuzione del danno da parte di un Actor esterno e la funzione che si occupa di calcolare come questo valore venga applicato a un attributo del personaggio.

```
// Apply the health change and then clamp it
float NewResistance;
if(GetArmour() >= 0)
    NewResistance= GetResistance() - LocalDamageDone * (100 / (100 + GetArmour()));
else
    NewResistance= GetResistance() - LocalDamageDone *  (2 - 100 / (100 - GetArmour()));
// const float NewResistence = GetResistence() - LocalDamageDone;
SetResistance( NewVal: FMath::Clamp( X: NewResistance, Min: 0.0f, GetMaxResistence()));
```

Figura 7.2: Funzione che applica i danni alla resistenza del giocatore

Come si può vedere nell'esempio, il Meta Attribute LocalDamage viene applicato alla resistenza venendo prima modificato dal valore di Armour del giocatore, permettendo quindi a un secondo attributo di intervenire nel calcolo di questa diminuzione senza dover aggiungere nessuna logica all'interno della fonte del danno.

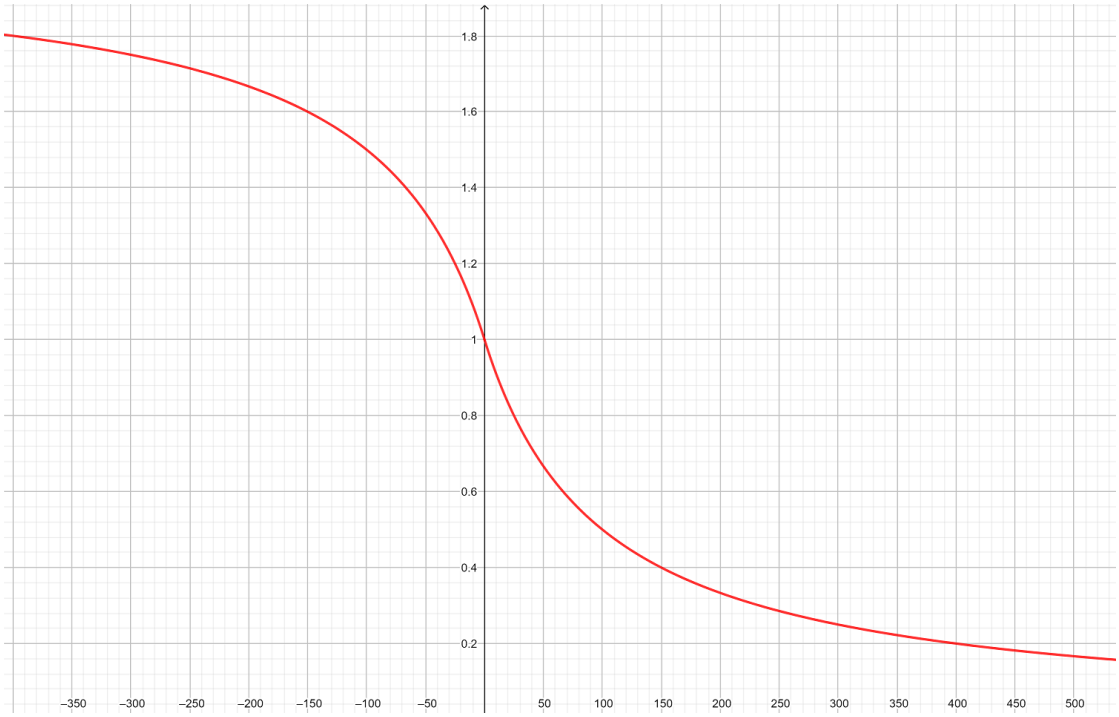


Figura 7.3: Grafico del moltiplicatore per il danno subito sulla base del valore dell'attribute Armour

7.1.4 Attribute Set

L'Attribute Set è la classe all'interno di cui si definisce un insieme di Attribute e che ne gestisce i cambiamenti. Il programmatore dovrà quindi estendere la classe UAttributeSet per crearne una propria versione, definendo gli Attribute necessari. Ogni AbilitySystemComponent permette di avere uno o più Attribute Set, quindi è possibile creare strutture per cui gli attribute vengono assegnati o rimossi su necessità.
Una delle funzioni più importanti è la PreAttributeChange. Questa permette di intervenire prima che un attributo venga modificato, intervendo sull'entità del cambiamento stesso. È ad esempio possibile imporre che un attribute non possa cambiare più di un determinato valore a ogni update.
Esiste anche la funzione PostGameplayEffectExecute che permette di intervenire dopo il cambiamento di un Attribute, ad esempio mostrando a schermo l'entità del cambiamento o eseguendo animazioni del personaggio.

7.1.5 Gameplay Effects

I Gameplay Effect sono il mezzo con cui le abiità interagiscono con gli Attribute e con i Gameplay Tag.
Ogni Gameplay effect può modificare il valore di un Attribute sia in maniera diretta attraverso un Execute, come nel caso del danno subito dal personaggio nella sezione precedente, o attraverso un Modifiers temporaneo, che applica una modifica fintanto che l'effetto risulta attivo all'interno del component.
Ogni Effect può avere una durata diversa in base al parametro duration:

- **Instant:** Applica il cambiamento di valore in maniera istantanea attraverso la execute.
- **Duration:** Permette di specificare il tempo per cui l'effetto è applicato a un AbilitySystemComponent. Allo scadere del timer l'Effect verrà rimosso e con lui tutti i tag e i cambiamenti agli attributi legati alla sua applicazione.
- **Infinite:** Segue lo stesso comportamento della keyword Duration, ma non verrà mai eliminato in modo automatico.

L'applicazione di un Modifier può invece seguire un comportamento legato a un'operazione matematica, andando quindi a sommarsi o a moltiplicare l'attributo modificato, o effettuandone un override, sovrascrivendone il valore.
Per l'applicazione dei tag è invece presente una struttura dati articolata in diversi set, che permette di specificare diversi gruppi di tag con diversi comportamenti:

- **Gameplay Effect Asset Tags:** Tag che caratterizzano l'effect stesso ma che non modificano in alcun modo l'AbilitySystemComponent a cui l'effect è applicato. Vengono usati per descrivere la categoria a cui appartiene l'effect.

- **Granted Tags:** Tag che vengono aggiunti all'AbilitySystemComponent per tutta la durata dell'effect.
- **Ongoing Tag Requirements:** Tag che determinano se l'effetto sta agendo. Un effetto potrebbe essere applicato ma non per questo avere un effetto tangibile, ad esempio una riduzione periodica di un attributo potrebbe non agire se l'attributo ha già raggiunto il valore minimo.
- **Application Tag Requirements:** Tag che devono essere presenti sul bersaglio perchè l'effect possa essere applicato. Se anche uno dei tag specificati non è presente allora l'effetto non verrà applicato.
- **Removal Tag Requirements:** Nel caso l'effect venga applicato qualsiasi altro effect che abbia un tag rientrante in questo elenco viene rimosso.

Per ogni Gameplay Effect è possibile creare un Gameplay Effect Spec, che è definibile come un'istanza di un Gameplay Effect, un po' come un oggetto è l'istanza di una classe. Questo permette di creare effetti i cui valori dei Modifier non sono definiti in anticipo ma vengono decisi a runtime.

7.1.6 Gameplay Abilities

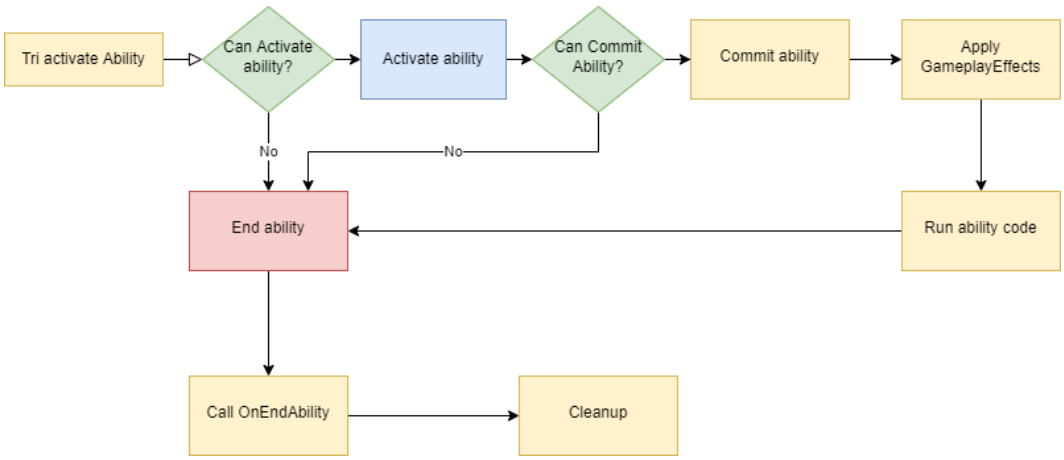


Figura 7.4: Flowchart di un'abilità

Le Gameplay Abilities servono a creare le capacità del singolo Actor, offrendo al programmatore un sistema già integrato nell'engine con cui sia possibile specificare molto facilmente gli eventuali costi di un'abilità, quali situazioni possono interromperne l'utilizzo e il comportamento. In poche parole ogni singola abilità è un Delegate, o un Functor di C++, che viene associato all'AbilitySystemComponent insieme alle proprie clausule di attivazione e può venir richiamato sia in risposta a

un input di un giocatore che come reazione automatica ad alcuni eventi. Come i Gameplay Effect anche le abilità hanno una serie di strutture interne per memorizzare dei tag che consentono di definire in maniera molto semplice alcune regole di funzionameto:

- **Ability Tags:** Elenco dei Tag che definiscono l'abilità. Forniscono un modo rapido per identificare un'abilità senza doverne conoscere la classe.
- **Cancel abilities with Tag:** Qualsiasi abilità in esecuzione che possieda un Tag presente in questo elenco viene interrotta.
- **Block abilities with Tag:** Qualsiasi abilità in esecuzione che possieda un Tag presente in questo elenco viene temporaneamente fermata durante l'esecuzione della nuova abilità.
- **Activation Owned Tags:** Elenco dei Tag che vengono aggiunti all'AbilitySystemComponent dell'Actor attivatore dell'abilità mentre quest'ultima è in esecuzione.
- **Activation Required Tag:** Elenco dei Tag che devono essere presenti nell'AbilitySystemComponent dell'Actor attivatore dell'abilità.
- **Activation Blocked Tags:** Elenco dei Tag che, se presenti nell'AbilitySystemComponent dell'Actor attivatore, bloccano l'attivazione dell'abilità stessa.

Degli ultimi due parametri esistono due versioni aggiuntive che permettono di effettuare lo stesso tipo di controlli anche sull'Actor fonte dell'abilità, che non obbligatoriamente combacia con l'Actor che ha attivato l'abilità, e sull'eventuale bersaglio dei suoi effetti.

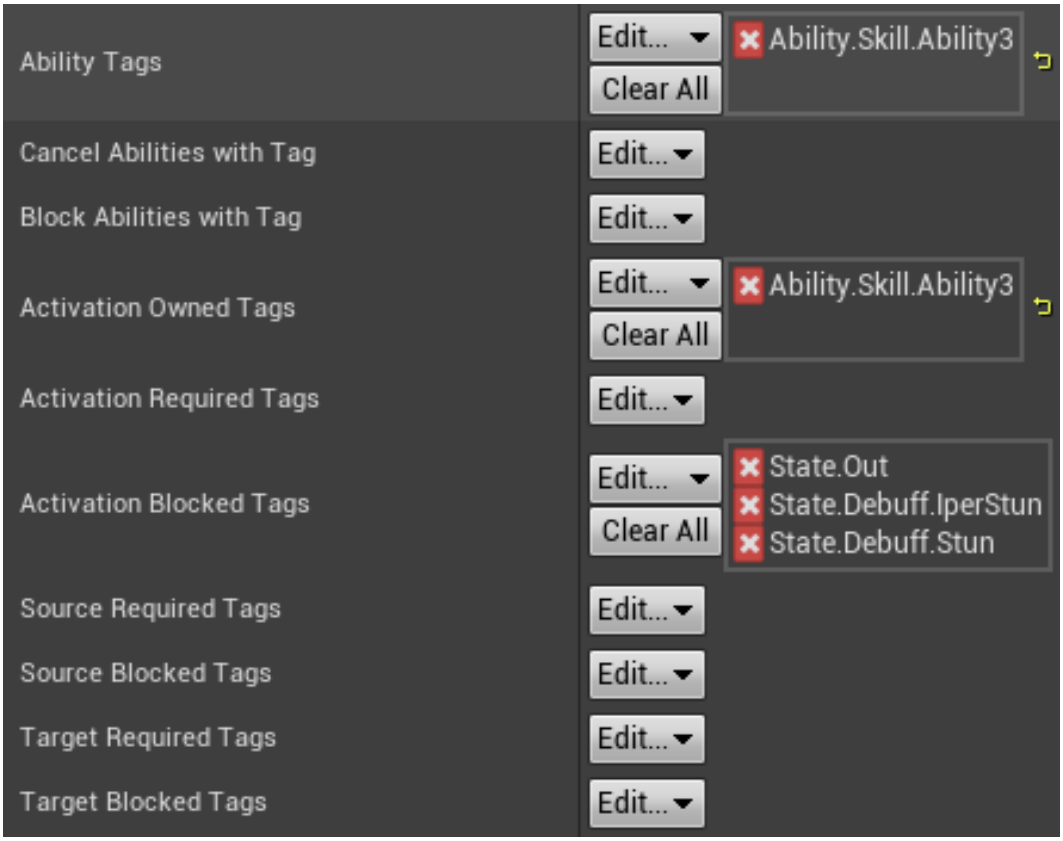


Figura 7.5: Esempio dei Tag di un'abilità

Oltre a questo è anche possibile specificare due serie di effetti, Cooldown e Cost. Il primo fa in modo che un insieme di GameplayEffect con una duration siano applicati al'AbilitySystemComponent e vengano utilizzati per imporre un intervallo di tempo minimo tra due attivazioni successive della stessa abilità. Il secondo invece serve a stabilire una serie di modifiche a un AttributeSet che vanno a comporre il costo in "risorse" dell'abilità. In caso queste modifiche non possano venir effettuate l'abilità viene annullata.
Per quanto riguarda il comportamento all'interno di un gioco online ogni abilità ha due parametri diversi. La Net Execution Policy, che determina la possibilità per il client di effettuare una predizione dell'esito dell'abilità, e la Net Security Policy, che serve a limitare a specifici ruoli chi sia in grado di richiamare le abilità.
La prima delle due policy può avere i seguenti valori:

- **Local Only:** L'abilità viene eseguita solo sul client proprietario. Usata in genere per creare abilità con semplici effetti visuali e nessuna modifica al gameplay. È importante notare che, nonostante il nome sembri suggerirlo, nei giochi single player non vada usata questa policy ma la "Server Only".

- **Server Only:** L'abilità viene eseguita solamente sul server. È il comportamento utilizzato in genere per le abilità la cui attivazione non sia legata a un input dell'utente.
- **Local Predicted:** L'abilità viene attivata sul client e successivamente sul server. La versione dell'abilità eseguita sul server correggerà qualsiasi discrepanza dei dati del client, in maniera simile al sistema di movimento.
- **Server Initiated:** L'abilità segue una sequenza di attivazione inversa a quella appena descritta nella Local Predicted.

Per quanto riguarda la Net Security Policy i quattro valori possibili sono invece:

- **ClientOrServer:** Non esiste nessuna forma di sicurezza, sia client che server possono attivare liberamente l'abilità.
- **ServerOnlyExecution:** I client non potranno richiedere l'attivazione dell'abilità, ma ne potranno chiedere l'interruzione o annullamento al server.
- **ServerOnlyTermination:** Speculare all'impostazione precedente, i client potranno richiedere l'attivazione di una abilità, ma non potranno in nessun modo interromperne l'esecuzione.
- **ServerOnly:** I client non hanno alcuna forma di controllo sull'abilità, qualsiasi loro richiesta verrà scartata e ignorata dal server,

7.1.7 Ability Task

L'esecuzione di un'abilità è sempre contenuta all'interno dell'update di stato del game engine e questo ne limita la durata a un solo frame. Per qualsiasi tipo di effetto che necessiti di un tempo di esecuzione più lungo esistono quindi gli Ability Task.
Questi oggetti permettono di legare delle sezioni di codice alla chiamata di un delegate o allo scadere di un timer, allungando quindi la durata di un'abilità anche oltre il singolo frame in cui viene richiamata. Uno dei più utilizzati è PlayMontageAndWaitForEvent, che permette di far eseguire un'animazione a un Actor e di legare una serie di funzioni a dei delegate richiamati in momenti specifici dell'animazione stessa.

Figura 7.6: Task PlayMontage

7.1.8 Gameplay Cues

I Gameplay Cue sono dei messaggi che vengono usati per richiedere l'attivazione remota di qualsiasi effetto che non faccia strettamente parte del gameplay, dall'emissione di un suono all'attivazione di un particellare. Per fare questo viene comunicato al GameplayCueManager, attraverso una GameplayCueNotify, un tag che identifichi in maniera univoca l'effetto da generare.

Ad ognuna di queste comunicazioni è affiancata una struttura dati chiamata FGameplayCueParameters, che permette di specificare una serie di parametri necessari alla corretta applicazione dell'effetto. Questi vanno dal livello dell'abilità che ha generato la comunicazione alla magnitudo dell'effetto stesso, che può quindi essere usata come parametro per definire la potenza di un suono o il numero di mesh visualizzate in un particellare.
Il sistema di comunicazione dei Gameplay Cue è molto semplice, a ogni nuova istanza di un effetto corrisponde una nuova RPC Multicast per comunicarne l'attivazione a tutti i client connessi. Questa semplicità rischia però di creare intasamenti nel caso di eventi che provocano una grande quantità di effetti in contemporanea, tant'è che Unreal stesso limita il numero di questo tipo di chiamate a solo due per ogni net update. Per aggirare queste limitazioni è bene fare in modo che il maggior numero possibile di Cues sia di tipo locale, quindi generate dai client stessi sulla base dei dati della propria simulazione locale.

Figura 7.7: Effetti visuali eseguiti attraverso un Gameplay Cue

7.2 Prediction

Essendo nato come strumento per gestire le abilità dei personaggi di un gioco online, il Gameplay Ability System offre nativamente un'integrazione alla libreria di networking. Più precisamente esistono dei sistemi di prediction che permettono ai client, in maniera simile alla gestione del movimento trattata in precedenza, di simulare temporaneamente in locale l'attivazione di una abilità e le sue conseguenze, prima che il server confermi o corregga la situazione.
Più precisamente, i client sono in grado di predirre e simulare:

- L'attivazione di un'abilità.
- L'attivazione di un effetto.
- L'applicazione di un effetto (ma non la sua rimozione o attivazione periodica).
- L'attivazione di un Gameplay Cue.

- Le animazioni.
- I movimenti.

7.3 Utilizzo all'interno del progetto

In questa sezione verranno trattati alcuni dei numerosi componenti del gioco il cui sviluppo è basato sulle tecnologie del Gameplay Ability System.

7.3.1 DProjectile

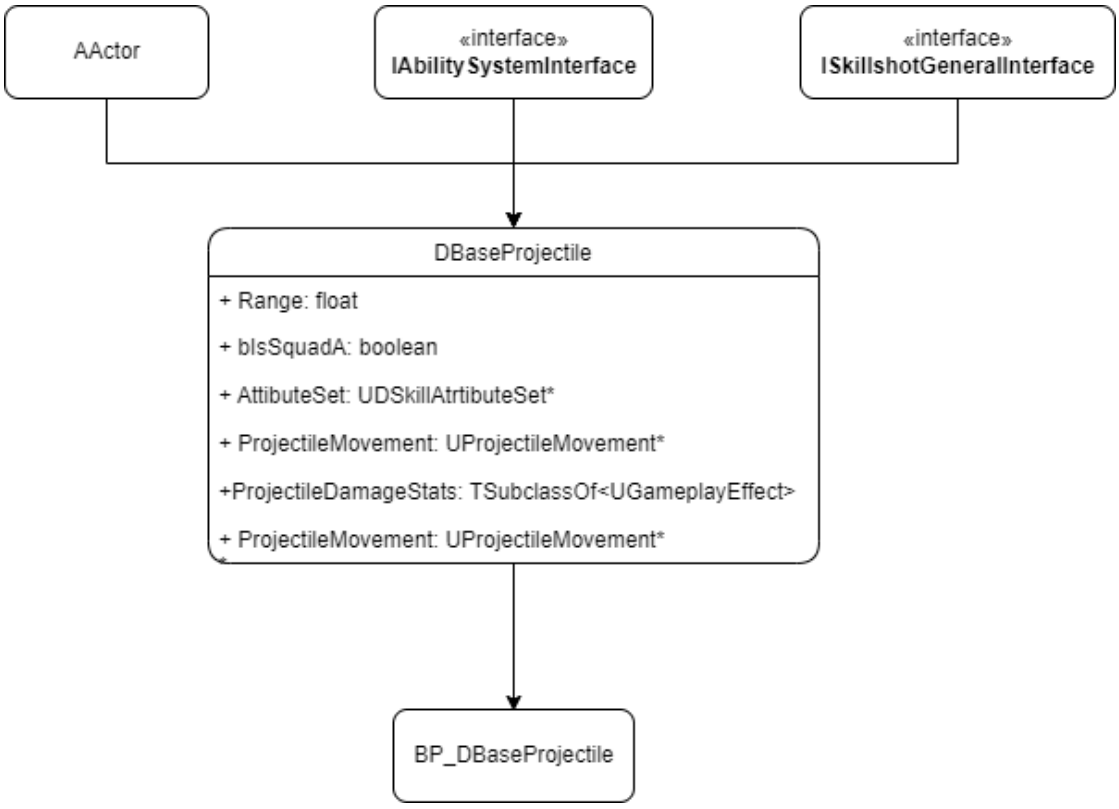


Figura 7.8: UML della classe BP_DBaseProjectile

Questa classe modella il comportamento di tutti i proiettili all'interno del gioco e per realizzarla sono stati utilizzati alcuni componenti legati al Gameplay Ability System. Questo per permettere di definire i diversi parametri che si utilizzano per creare differenti tipologie di proiettili. Più nello specifico i suoi componenti sono:

- **AttributeSet:** Istanza della classe UDSkillAttributeSet, specializzazione di un Attribute Set, contenente 3 attributi. Due attributi servono a definire

velocità e danno del colpo e sono stati ideati per permettere ai designer di modificarli con semplicità senza bisogno di intervenire a livello del codice sorgente, limitandosi a creare una nuovo effetto per inizializzare le statistiche a valori diversi. Il terzo attributo è invece un meta Attribute utilizzato per gestire il danno subito nel caso di collisione con un proiettile della squadra avversaria.

- **ProjectileDamageStat:** Variabile che permette di specificare la classe dell'effetto che inizializza le statistiche del proiettile. Durante la creazione di quest'ultimo viene definita un'istanza di questa classe che viene applicata all'Attibute Set.
- **bIsSquadA:** Boolean che definisce la classe di appartenenza del giocatore che ha creato il proiettile. Serve per definire con quali giocatori e proiettili questo oggetto debba interagire.
- **Range:** Definisce la distanza limite entro cui il proiettile si muove. Alla creazione dell'oggetto verrà istanziato un timer con una durata pari a *Range/Speed* che si occuperà di distruggere l'oggetto una volta raggiunta la destinazione.
- **ProjectileMovement:** Semplice istanza di un UProjectileMovement che muoverà il proiettile. La sua veloctà viene definita in fase di creazione sulla base del valore contenuto nell'Attribute Set, a sua volta definito dalla classe ProjectileDamageStat.
- **IAbilitySystemInterface:** Interfaccia standard del Gameplay Ability System, permette a qualsiasi componente di accedere all'istanza dell'Ability System Component legata al proiettile.
- **ISkillshotGeneralInterface:** Interfaccia custom creata dal team che permette al proiettile di distinguere qualsiasi oggetto sia in grado di interferire con il proiettile stesso, che sia un altro proiettile simile o una palla neutra lanciata da un giocatore, e di leggerne l'output di danno per applicare gli effetti necessari.

La classe BP_DBaseProjectile è invece una semplice estensione in blueprint della classe C++, che permette di associare una mesh al proiettile in maniera più rapida dall'interfaccia grafica dell'engine, senza bisogno di modificare e ricompilare il codice sorgente.

7.3.2 Aumento dell'armour

Quest'abilità è stata completamente creata in blueprint, il linguaggio di scripting visuale presente all'interno di Unreal Engine. Il suo obbiettivo è di applicare un

effetto che incrementi l'attribute Armour di tutti i compagni di squadra all'interno di una piccola area, rendendoli più resistenti ai colpi degli avversari. Per fare ciò è stata creata la classe MouseTargetActor, da utilizzare con il task Wait Target Data, che permette di mantenere l'abilità in attesa di una conferma dell'utente. Mentre l'abilità rimane in attesa viene creata un'istanza del MouseTargetActor, che segue i movimenti del mouse rimanendo entro una distanza limite dal giocatore.

Figura 7.9: MouseTargetActor a fianco al giocatore che lo sta generando

Quando quest'ultimo conferma l'area desiderata per l'applicazione dell'abilità il Target Actor resistuisce un elenco di tutti i giocatori al proprio interno appartenenti a una specifica squadra. A questo punto, se l'array di giocatori non è vuoto viene eseguito il commit dei costi dell'abilità, iniziandone il cooldown, altrimenti l'abilità viene annullata. Prima di applicare il Gameplay effect viene effettuato un branch valutando il valore della local authority, in quanto non ha nessuna utilità aggiungere effetti a un giocatore all'interno del client, visto che non verrebbero replicati sul server. Quindi, dopo il branch, il server si occupa di applicare l'effetto a tutti i giocatori nell'array e poi questa informazione sarà replicata a ogni client, seguita da eventuali Cue necessari all'avvio degli opportuni effetti visuali e sonori.

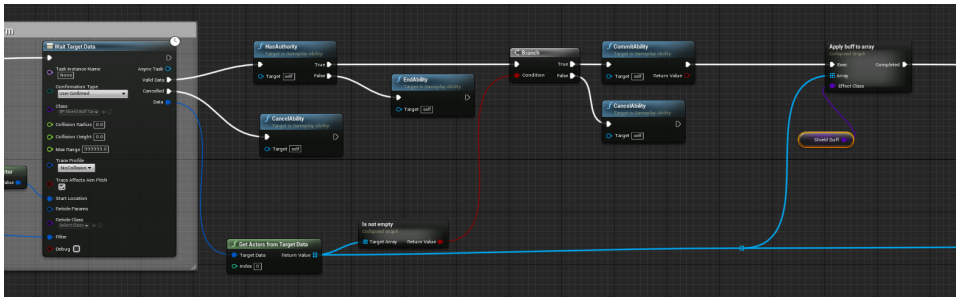


Figura 7.10: Blueprint dell'abilità

L'effetto applicato ha ogni giocatore è di tipo Duration e Add. Avrà quindi una durata limitata, alla fine della quale l'effetto verrà eliminato senza bisogno di nuovi interventi dell'abilità stessa, e andrà a modificare il valore corrente dell'attributo Armour incrementandolo del valore definito all'interno dell'Effect stesso.

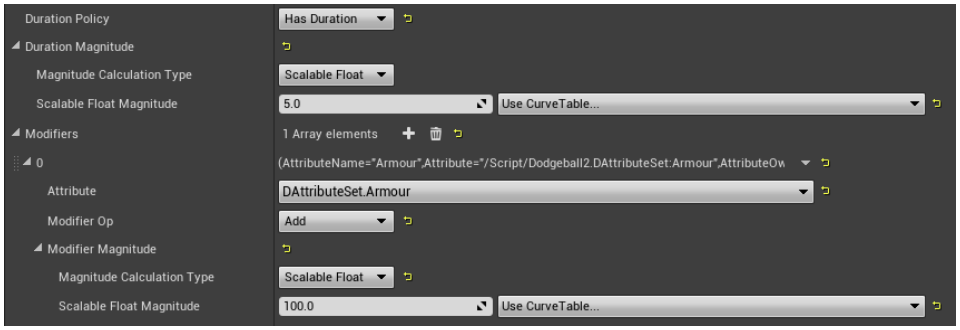


Figura 7.11: Parametri dell'effetto

7.3.3 Granata velenosa

Il personaggio dell'ape presente nel gioco possiede un'abilità interessante dal punto di vista del Gameplay Ability System. Questa si compone di due parti separate, una prima che prende la forma di una Gameplay Ability e una seconda che invece è un'istanza di una specializzazione della classe BP_ExplosiveGrenade.
La Gameplay Ability si compone di due task separati, un primo che, come nell'abilità precedente, permette di generare un target che segue la posizione del mouse, con la differenza che il valore di interesse per l'abilità sarà la posizione dell'Actor e non i giocatori compresi nel suo raggio. Il secondo invece esegue un'animazione e pone tre delegate, definiti in C++, in attesa di cambi di stato nell'animazione stessa:

- **OnCancelled:** Reagisce all'annullamento dell'animazione, annullando l'abilità.

- **OnCompleted:** Reagisce al completamento dell'animazione, mettendo fine all'abilità.

- **OnEventReceived:** All'interno dell'animazione è possibile specificare degli eventi a punti specifici. Nel momento in cui l'animazione li raggiunge viene richiamato questo delegate, passando come parametro l'evento scatenante della chiamata. Nel caso l'evento abbia il Tag "Event.Montage.SpawnProjectile" allora viene richiamata la funzione Throw_Granade().

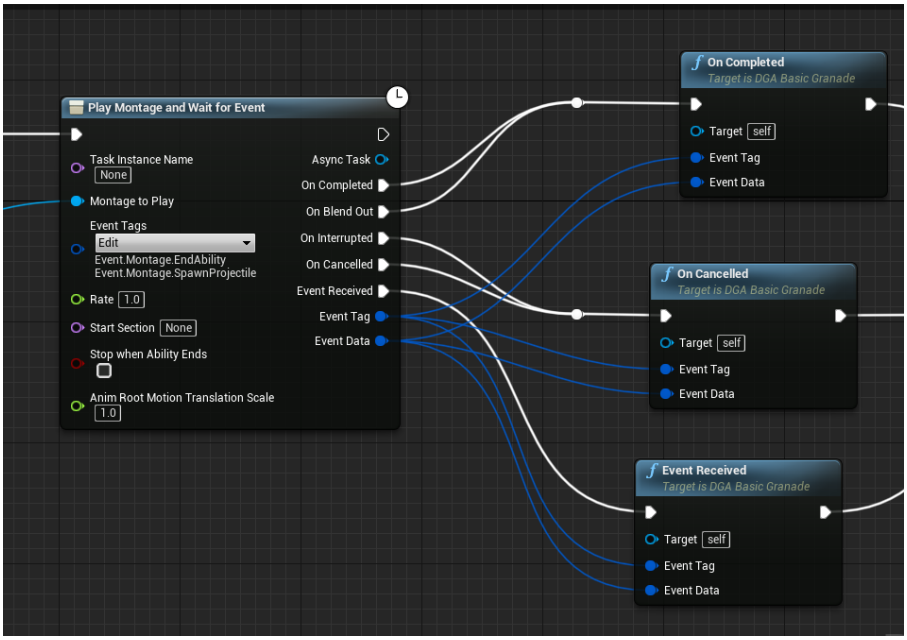


Figura 7.12: Task animation montage e relativi delegate esperri all'interno del blueprint

Quest'ultima funzione richiama quindi un nuovo evento blueprint, che si occupa di creare un'istanza della granata stessa, associarle l'effetto di danno che dovrà infliggere ai bersagli colpiti e quindi deciderne l'angolazione di tiro attraverso il metodo SuggestProjectileVelocity. Questa funzione, partendo da una velocità massima, un punto di arrivo e un'origine, è in grado di calcolare la direzione in cui va lanciato l'oggetto per fargli seguire una traiettoria tra i due punti.
Durante tutta la durata dell'animazione viene impedito al personaggio di muoversi, per via di una scelta di design dell'abilità stessa, e questo viene fatto attraverso le funzioni DisablePath (per disabilitare il movimento) e EnablePath (per riabilitarlo). A questo punto il personaggio risulta libero di muoversi e non più impegnato nell'abilità, permettendo quindi di iniziare il conto alla rovescia per il cooldown. Il resto dell'abilità sarà quindi gestito dal codice all'interno della granata stessa.

Figura 7.13: In giallo l'arco calcolato attraverso la funzione SuggestProjectile-Velocity

Il comportamento di base della granata è descritto dalla classe C++ BaseGrenade. Questa classe è composta da una mesh, che rappresenta il corpo stesso della granata, a cui è legato un volume sferico diverse volte più grande della mesh, impostato per rilevare ogni sovrapposizione con altri oggetti ma per non partecipare alla simulazione fisica. Appena la mesh entra in contatto con una qualsiasi superficie con il tag "Floor" la velocità viene azzerata, rendendo la granata un oggetto completamente statico, e parte un timer. Allo scadere del timer, l'oggetto viene distrutto e a tutti i giocatori all'interno del volume sferico, che appartengano alla squadra avversaria, viene applicato un effetto chiamato BP_GrenadePoison_GE e una leggera spinta radiale rispetto alla posizione della mesh.

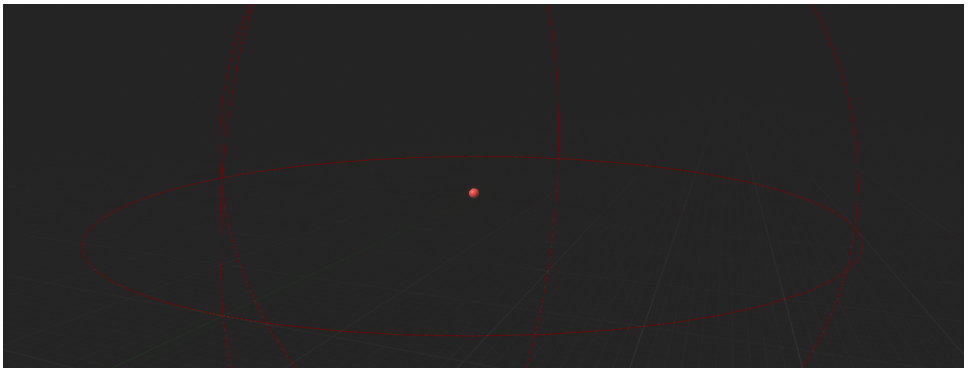


Figura 7.14: Placeholder della granata in fase di sviluppo. In rosso sono visibili le linee che delimitano il volume sferico per rilevare i giocatori avversari
L'effetto appena citato è di tipo Duration, con una durata di 6 secondi e una periodicità di 1.5 secondi, e un modifier di tipo Add con valore -5 per l'attribute Resistance. Questo porta a una applicazione del modifier per quattro volte, ognuna a una distanza di 1.5 secondi da quella precedente. Il fatto che sia modificato direttamente un Attribute invece di passare per il Meta Attribute del danno subito serve a fare in modo che questo effetto non risenta del valore di armatura del giocatore che lo ha subito.

Capitolo 8

Conclusioni

Questa tesi ha tentato di evidenziare quale sia il lavoro, sia creativo sia tecnico, che esiste dietro alla creazione di molti dei giochi online che popolano il mercato videoludico moderno, sottolineando il grande overhead di lavoro che uno sviluppo di questo tipo comporta rispetto a un più semplice gioco single player.

8.1 Considerazioni sul risultato raggiunto

Lo stato raggiunto dallo sviluppo del progetto è sufficientemente avanzato da poter iniziare una prima fase di testing approfondita sulle meccaniche di gioco e una iniziale raccolta dati sulle partite, in quanto i sistemi di rete risultano stabili e in grado di supportare lo svolgimento di una partita dall'inizio alla fine, resistendo anche a errori di comunicazione che rimangano all'interno di valori contenuti.
Il raggiungimento di questa fase di sviluppo è molto importante per un videogioco che intenda presentarsi a un publisher in ricerca di finanziamenti, perchè dimostra, seppur a un livello ancora basilare, la fattibilità tecnica del gioco e le sue potenzialità.
Detto questo, il progetto mostra ancora una serie di mancanze rispetto a quello che ci si aspetta da un gioco pienamente sviluppato. È completamente assente un sistema di rete che permetta di effettuare un matchmaking tra giocatori che non si conoscono, fatto che limita fortemente l'ambito competitivo, e l'appoggiarsi del sistema di connessione ai servizi di una sola azienda impedisce di estendere il funzionamento al di fuori del mercato del singolo fornitore di servizi, ad esempio approdando sul mercato delle console.
In un eventuale futuro commerciale questo problema andrà risolto, possibilmente creando una libreria che consenta un livello di astrazione superiore, rendendo le funzioni di connessione indipendenti dal sistema sottostante. Oltre a questo si potrebbe anche pensare alla creazione di un proprio launcher per la versione PC,

eliminando la necessità di appoggiarsi ai servizi di aziende esterne.
Inoltre anche diversi ambiti del gameplay richiedono ancora raffinamenti prima di potersi dire conclusi, come il sistema di scelta delle abilità a inizio partita, per ora solo in stato embrionale, o il raggiungimento di un numero più elevato di personaggi disponibili.

Bibliografia

Bibliografia Classica

[1] Sanjay Madhav Joshua Glazer. *Multiplayer Game Programming: Architecting Networked Games*. 1ª ed. Addison-Wesley, 2015. ISBN: 978-013-403430-0.

Sitografia

[2] *Unreal Engine Documentation*. URL: https://docs.unrealengine.com/4.27/en-US/.

[3] *Gaffer on Games*. URL: https://gafferongames.com/.

[4] *Fast-Paced Multiplayer*. URL: https://www.gabrielgambetta.com/client-server-game-architecture.html.

[5] *GAS Documentation*. URL: https://github.com/tranek/GASDocumentation#concepts-ge-mods.

[6] *Unreal Engine Network Compendium*. URL: https://cedric-neukirchen.net/Downloads/Compendium/UE4_Network_Compendium_by_Cedric_eXi_Neukirchen.pdf.

[7] *Wikipedia*. URL: https://en.wikipedia.org/wiki/Multiplayer_video_game.

[8] *Wikipedia*. URL: https://en.wikipedia.org/wiki/History_of_online_games.

[9] *AWS Game Tech*. URL: https://aws.amazon.com/it/gametech/game-servers-networking/.