UNIVERSITÀ DEGLI STUDI DI SALERNO

Dipartimento di Informatica

Corso di Laurea in Informatica

Tesi di Laurea

Esperienze ludiche collaborative multicanale

Tramite ambientazioni interattive virtuali

Relatore

Prof. Andrea F. Abate

Correlatore

Dott. Ignazio Passero

Laureando

David Grieco

Matricola: 0512105112

Sommario

Il progetto di tesi è stato finalizzato alla sperimentazione e allo sviluppo di un'applicazione multicanale per esperienze ludiche. In particolare, il gioco proposto è stato organizzato in due esperienze cooperative che portano al raggiungimento degli obbiettivi di gioco. A questo scopo, l'applicazione è stata sviluppata sia per Desktop sia per l'ambiente virtuale immersivo VR sorretto dalla tecnologia Oculus Quest, sfruttando le caratteristiche migliori di entrambi gli ambienti per la realizzazione delle metafore di gioco. Nel dettaglio, l'utente che controlla l'applicazione PC comanda un veicolo di tipo Tank gestendone gli spostamenti sul campo di gioco. Il secondo giocatore, proiettato in un contesto immersivo dal visore VR, comanda a sua volta la mira e il fuoco della torretta del Tank. L'obbiettivo del secondo giocatore è direzionare con precisione tramite la rotazione del capo, la torretta verso i bersagli. Durante il gioco un sistema automatico genera degli eventi particolari chiamati "Catastrofi", durante i quali entrambi i giocatori dovranno collaborare per difendersi. Solo il giocatore VR può sparare ad un Target particolare per interrompere le Catastrofi, ma poiché il suo angolo di tiro è limitato, per il successo dello sparo, dovrà coordinarsi con il giocatore PC. L'applicazione proposta si è dimostrata coinvolgente nelle prime prove effettuate e questo risultato incoraggia il proseguimento dello studio di applicazioni multicanale di questo tipo.

Indice

“To create a new standard, you have to be up for that challenge and really enjoy it.”

Traduzione:

“Per creare un nuovo standard, bisogna essere pronti alla sfida e divertirsi molto nell’affrontarla.”

Shigeru Miyamoto – Creatore di Super Mario e Zelda e game director di Nintendo Company

Capitolo 1

Fondamenti sulle esperienze videoludiche

1.1 I videogiochi nella società odierna

Il gioco è uno tra gli elementi sociali che da più tempo accompagna l'uomo nel suo percorso evolutivo. È proprio tramite il gioco, infatti, che, durante la tenera età, i bambini imparano il funzionamento del mondo. Di conseguenza, non è di grande volume la sorpresa nello scoprire che nonostante siano passati migliaia di anni dai grandi giochi del Colosseo, l'uomo non ha mai smesso di divertirsi e di utilizzare il gioco come lente alternativa su di un mondo che, altrimenti, risulterebbe monotono. Con gli anni, ovviamente, anche le tecnologie che supportano il divertimento sono cambiate: da un semplice pallone in pelle di animale lanciato per curiosità si è passati ad insiemi di regole e dinamiche definite nei minimi dettagli chiamati "sport"; alla materializzazione del proprio pensiero sotto forme artistiche come la pittura e la musica, si è aggiunta anche l'arte digitale. E dall'insieme dei sopracitati fattori, nascono infine i videogiochi: la massima espressione della creatività umana, comprendente la maggioranza delle sfere creative in cui l'uomo si sia mai cimentato.

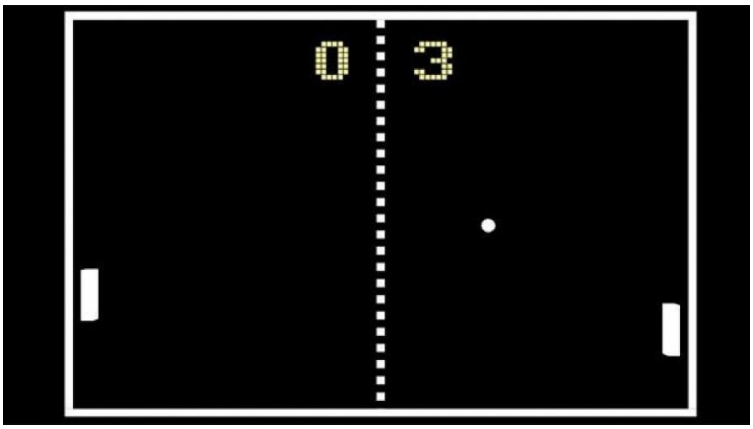

Figura 1.1: Pong (1958) fu il primo videogioco ad entrare nella vita quotidiana delle famiglie.

1.1.1 Pro e contro: un'infinita controversia

Nel corso della loro breve storia i videogiochi hanno subito una trasformazione esponenzialmente veloce data dal continuo perfezionamento dei supporti fisici necessari al loro funzionamento. A trasformarsi è stata anche la visione della società a riguardo. Se per molti i videogiochi sono un ottimo strumento di svago, per altri sono un possibile rischio per l'intelletto, se accolti nel modo sbagliato. Spesso, soprattutto con la crescita di popolarità che il genere FPS (First Person Shooter) o Sparatutto ha acquisito, essi vengono etichettati come la causa di atti di violenza o terrorismo nel mondo, quando, nella maggioranza dei casi, le ragioni affondano le proprie radici in eventi traumatici avvenuti nel trascorso di tali soggetti. Come invece dimostrano scientificamente studi sull'argomento, non vi è in realtà nessuna correlazione tra la violenza nei videogiochi e la violenza nella vita reale espressa dagli adolescenti che ne fanno uso abituale [1]. Altri studi, d'altro canto, sono stati condotti sulle influenze positive che i videogiochi hanno sugli adolescenti in particolare, e più in generale su chiunque ne faccia uso. Ne è scaturito che un sano utilizzo dei mezzi videoludici dona numerosi vantaggi, tra i quali un aumento delle capacità di memorizzazione [2] e cognitive [3], sfatando in tal modo il falso mito condannante tali piattaforme come un danno per la società odierna.

Un altro importante fenomeno in crescita riguardante l'argomento è la crescita in importanza dei cosiddetti "E-Sports" o videogiochi competitivi, in cui videogiocatori professionisti competono a livello agonistico con un seguito ormai paragonabile a quello dei comuni sport. Tali figure, dette Professional Players (spesso abbreviato in Pro Player) dedicano le proprie giornate al perfezionamento dei propri riflessi o delle proprie capacità decisionali, e al miglioramento di molte altre caratteristiche legate al cervello,

a tal punto che nella maggioranza dei casi sono affiancate da figure mediche, quali psicologi, terapeuti e nutrizionisti, in modo da spingere al limite le abilità individuali e di squadra ed avere così la meglio nel momento del bisogno.

Figura 1.2: League of Legends World Championship (2015) fu il torneo di un E-Sport con più spettatori dal vivo di sempre.

1.2 Cenni di esperienze Multiplayer

Di certo uno degli elementi che più attrae del mondo dei videogiochi è la possibilità di giocare con altri giocatori dislocati in tutto il mondo e ciò ha come conseguenza diretta l'aumento di giochi basati sulla componente multigiocatore che si sta avendo negli ultimi anni. Progressivamente l'industria videoludica sta mettendo in secondo piano la componente Single Player (giocatore singolo) dei videogiochi, e si sta quindi notando un graduale

aumento delle uscite di giochi che possono essere giocati esclusivamente connessi ad una rete.

1.2.1 Storia del Multiplayer

Prima di analizzare la storia dei videogiochi Multiplayer è giusto effettuare una banale ma necessaria suddivisione di partenza per evitare ambiguità: I videogiochi Multiplayer si dividono in Networked e Non-Networked, dove i primi sono tali da sfruttare la rete per inviare informazioni tra i singoli utenti, mentre i secondi non necessitano della rete per il proprio funzionamento essendo eseguiti sulla stessa piattaforma. Ovviamente i Non-Networked sono di molto più semplice funzionamento rispetto ai Networked, infatti non devono gestire nessun tipo di sincronizzazione tra gli utenti che ne fanno uso perché essi partecipano alla partita utilizzando la stessa macchina e a quest'ultima arrivano tutti gli input di entrambi. Per tali ragioni la loro storia nasce più indietro nel tempo rispetto alla loro controparte "di rete". Il primo gioco in multigiocatore locale (Non-Networked) risale agli anni '50 ma era molto primitivo e non considerabile un vero e proprio videogioco. Le prime apparizioni di veri e proprio videogiochi Non-Networked si hanno negli anni '70 con una diretta influenza sul nascente mondo degli "Arcade" che fece da padrone negli ultimi anni di tale decennio [4].

I primi giochi Networked nascono invece perlopiù in ambienti universitari di ricerca dove, utilizzando le strutture informatiche all'epoca disponibili in tali ambienti. Gli esperimenti di maggior successo furono i giochi STAR (gioco multiplayer Networked su larga scala) e CAVE (gioco di avventura), tutti sviluppati all'interno delle strutture di ricerca delle rispettive University of California e University of New Hampshire [4]. Tali giochi ovviamente utilizzavano un'architettura di rete molto semplificata rispetto al networking odierno (il primo FPS multiplayer utilizzava un'interfaccia MIDI per la comunicazione tra giocatori [4]), ma con il tempo tali architetture si evolveranno fino a raggiungere la famosa forma Client-Server su cui oggi si appoggia la maggior parte dei videogiochi.

Figura 1.3: MIDI Maze (1987) fu il primo Networked Multiplayer FPS della storia.

1.2.2 Funzionamento di base

Quando si parla di videogiochi Multiplayer Networked, esistono problematiche che vanno assolutamente prese in considerazione e che vanno a plasmare la risultante architettura di rete che si utilizza per il gioco. Tra di esse la più importante in assoluto è il cheating, ovvero la possibilità da parte dei giocatori di "imbrogliare" e cambiare le regole del gioco in proprio favore. Se ciò dovesse avvenire in un gioco a giocatore singolo non sarebbe un grosso problema – le sue azioni non influenzano nessuna se non la sua esperienza di gioco ed ogni giocatore è libero di godersi la propria esperienza come desidera. Il vero problema è quando ad una partita in un gioco competitivo partecipano più utenti connessi ad una rete ed un giocatore imbroglia migliorando la sua esperienza di gioco ma peggiorando quella altrui. Per evitare queste eventualità (o almeno ridurne la frequenza) si è giunti alla seguente scelta di design architetturale: I principali elementi su cui il gioco viene eseguito vengono divisi in Client e Server (a volte singolo); l'intera partita è in esecuzione solo sul server mentre i Client sono semplicemente "degli spettatori privilegiati di quest'ultima" [5]. In tal modo qualsiasi evento riguardante la partita in corso (come il movimento dei giocatori, la collisione con i proiettili, l'acquisizione di punti, ecc.) avviene sul server sotto richiesta dei client; Es.: Se un client vuole muoversi da un punto A ad un punto B in un determinato momento allora invierà una richiesta al Server il quale, dopo aver effettuato il calcolo del movimento e della risultante nuova posizione nella partita in esecuzione su di esso, invia tali nuovi dati al Client richiedente

(e a tutti gli altri in partita per aggiornare la posizione del Client richiedente nella loro partita) il quale aggiornerà la propria posizione nella sua partita locale. Ovviamente esistono casi in cui il Client potrebbe imbrogliare e cambiare dei valori fondamentali nella sua partita locale ma ciò sarebbe inutile non intaccando minimamente l'esperienza condivisa dagli altri giocatori, perché frutto dell'esecuzione di tale partita sul Server.

1.3 L'importanza del cross-platform

Negli ultimi anni un aspetto importante dell'industria videoludica è stato messo sempre di più sotto i riflettori dei media e della critica di settore: il cross-platform. Il cross-platform è una caratteristica di alcuni videogiochi consistente nella possibilità di partecipare a partite in cui sono connessi giocatori di piattaforme di gioco totalmente diverse. Tale scelta comporta una maggiore complessità di sviluppo da parte dello studio rilasciante, ma assicura una base di giocatori decisamente più ampia rispetto ai videogiochi vincolati ad una specifica piattaforma. Inoltre, in tal modo vengono soddisfatti anche quei clienti che comprano un prodotto a patto che possano usufruirne con i propri conoscenti, in media, aventi piattaforme hardware diverse dedicate al gaming. Decisamente una delle combinazioni per la realizzazione di un'esperienza cross-platform meno esplorata attualmente è quella PC-VR, ovvero un'esperienza in cui parte dei giocatori vivono un'immersione offerta dalle capacità della Realtà Virtuale, mentre un'altra parte vive la stessa esperienza ma vista da un'ottica differente tramite applicazione Desktop. La difficoltà principale nella realizzazione di un'applicazione cross-platform PC-VR risiede nella scelta del flusso di gioco, il quale deve sì risultare alternativo in base alla piattaforma sulla quale si partecipa, ma non vi devono essere carenze tali da rovinare l'esperienza vissuta dai possessori di uno dei due hardware. Se da un lato i giocatori VR vivranno un'esperienza più interattiva e immersiva rispetto alla controparte PC, il flusso di quest'ultima dovrà essere compensato aggiungendo elementi innovativi che faranno apprezzare l'esperienza sotto altri punti di vista.

Capitolo 2

Esperienza sviluppata

In questo capitolo verrà illustrato il funzionamento ed il flusso di gioco dell'esperienza proposta. Il tutto verrà descritto ad alto livello, tralasciando dettagli di progettazione a basso livello ritenuti non fondamentali per la comprensione delle scelte di sviluppo e design.

2.1 I dispositivi hardware

I dispositivi hardware utilizzati per il progetto di tesi consistono nel casco di Realtà Virtuale "Oculus Quest" e dei controller "Oculus Quest Touch Controllers". Per quanto riguarda l'esperienza Desktop, è necessario semplicemente un PC dotato di periferiche di input classiche (mouse e tastiera).

2.1.1 Il visore Oculus Quest

Il visore "Oculus Quest" è uno dei migliori caschi per la Realtà Virtuale attualmente sul mercato (superato solo recentemente dalla sua nuova versione "Oculus Quest 2"). La grande specialità di tale casco risiede nella sua portabilità: per il suo utilizzo non è infatti necessario un collegamento ad un PC per l'elaborazione dei dati; l'Oculus Quest, infatti, è un vero elaboratore autonomo, dotato di un suo processore e di una sua scheda grafica, che gli permettono di offrire un'ottima esperienza di gioco anche all'aperto.

2.1.1.1 I benefici della Realtà Virtuale

La Realtà Virtuale (spesso abbreviata in "VR"), dalla sua nascita risalente agli anni '70 con la creazione del software "Aspen Movie Map" [10], è in costante sviluppo e, negli ultimi anni, si iniziano ad osservare risultati decisamente promettenti. Gli utilizzi di tale ramo dell'informatica sono già moltissimi e non accennano a ridursi: componenti di Realtà Virtuale vengono per l'appunto utilizzati in ambito chirurgico, cinematografico e terapeutico, per citarne alcuni. L'utilizzo più comune, al di fuori della sfera videoludica, riguarda le esercitazioni. Esercitazioni che possono essere condotte, a prescindere dall'ambito, in totale sicurezza, esenti da fattori di stress dipendenti da complicanze reali. In ambito videoludico, d'altro canto, i vantaggi della Realtà Virtuale sono sostanziali: la VR permette a chiunque di vivere situazioni del tutto immersive riguardanti eventi che possono essere più o meno vicini a ciò che viene considerato "reale", offrendo, in tal modo, una variegata scelta di esperienze anche a coloro i quali, magari per problemi di salute, sarebbero impossibilitati a prender parte ad avventure simili nella vita reale. Un ulteriore vantaggio della VR consiste nella maggiore attivazione di zone del cervello legate al ramo cognitivo, che permettono una più rapida riabilitazione di soggetti con traumi pregressi [10].

2.1.2 Interfacciamento multicanale

Come citato in precedenza, il sistema proposto unisce le esperienze dell'utente VR e dell'utente Desktop. Tale unione è resa possibile dalla diversificazione delle build finali offerta da Unity. Unity permette, infatti, in

fase di compilazione, di selezionare la piattaforma per la quale si vuole generare l'eseguibile. Tale differenza di piattaforma deve, però essere presa in considerazione anche dal punto di vista software, in particolare al livello di scripting, dove va controllata la piattaforma sulla quale il gioco è in esecuzione per offrire la giusta esperienza dedicata.

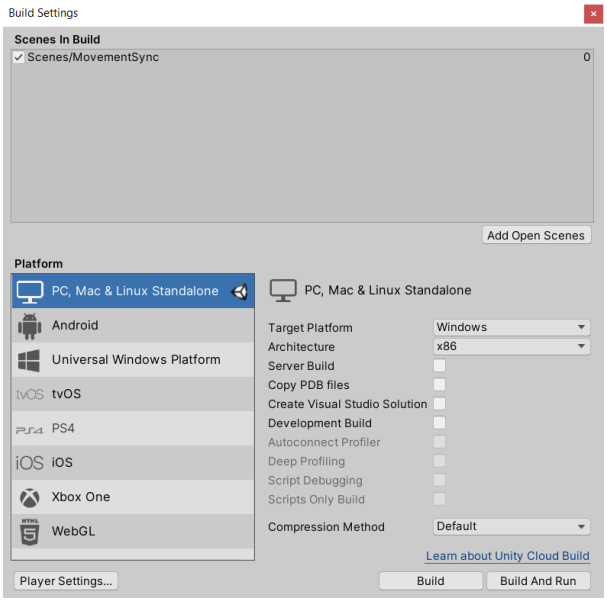

Figura 2.3: Schermata di build di Unity. Sono mostrati alcuni delle più comuni piattaforme target per la compilazione di un progetto.

2.2 Utilizzo del software

Sono stati in precedenza descritti l'ambiente di sviluppo del progetto di tesi e le attrezzature hardware necessarie al suo utilizzo. Nei seguenti paragrafi viene invece descritto il flusso di funzionamento dell'applicativo proposto e le motivazioni delle varie scelte di design e gameplay che lo caratterizzano.

2.2.1 Flusso di funzionamento

Il flusso di funzionamento è diversificato in base alla piattaforma di utilizzo. Tale diversificazione è alla base delle scelte progettuali e costringe ad una maggiore cooperazione tra gli utenti che prendono parte all'esperienza di gioco.

2.2.1.1 Connessione

Per quanto riguarda la connessione, inizialmente entrambi gli utenti (sia VR che Desktop) si troveranno dinanzi ad un menu composto da due opzioni: Host e Client. Un Host (tradotto dall'inglese in "ospite"), in breve, è un nodo di una rete che "ospita" programmi di livello applicativo che sono sia client, sia server [12]. Nel caso del progetto di tesi, il giocatore PC dovrà selezionare l'opzione Host in modo da rendersi disponibile sulla rete come Server a cui accedere, in aggiunta all'essere anche un client partecipante all'esperienza. Il giocatore VR dovrà, invece, selezionare Client, attivando in tal modo la connessione automatica al server appena reso disponibile dal giocatore PC. Conclusa la creazione del server da parte del giocatore PC, quest'ultimo si troverà all'interno dell'area di gioco, in cui, con visuale dall'alto, potrà iniziare a familiarizzare con i comandi del carro armato in attesa della connessione del giocatore VR. Avvenuta tale connessione, il giocatore VR apparirà all'interno dell'area di gioco sotto forma di un soldato nella torretta del carro armato comandato dal giocatore PC.

Figura 2.4: Schermata di connessione dell'applicativo proposto.

2.2.1.2 Gameplay

Lo scopo del gioco è ottenere più punti possibili prima della fine della partita. Per fare ciò bisogna sopravvivere alle continue Catastrofi che si verificano sul campo di gioco. I mezzi a disposizione dei giocatori per fuggire alle catastrofi consistono nell'abilità di guida del giocatore PC ai comandi del

carro armato, il quale dovrà in tutti modi tentare di schivare le Catastrofi e contemporaneamente tentare di raccogliere tutti i punti che appaiono nella zona con il progredire della partita; altro mezzo di difesa dalle Catastrofi consiste nella torretta del carro armato, controllata dal movimento della testa del giocatore VR (entro un raggio di 90 gradi avente centro nel fronte del carro), in grado di sparare proiettili utilizzabili per distruggere dei bersagli al di fuori della zona di gioco esplorabile. Tali bersagli (non visibili al giocatore PC) sono collegati alle Catastrofi attualmente presenti nella zona e, se distrutti, eliminano la Catastrofe ad essi associata. Inoltre, esistono determinate Catastrofi che possono essere direttamente colpite con tali proiettili. In definitiva, lo scopo ultimo del gioco è deciso dalla qualità della collaborazione tra i due giocatori che, sfruttando i diversi punti di vista che le due piattaforme di gioco offrono, devono coordinarsi per il raggiungimento dell'obiettivo comune.

Figura 2.5: Esempio di Catastrofe "Fire" all'interno del gioco.

2.2.1.3 Terminazione

La terminazione di una partita può avvenire in due modi: disconnessione e partita terminata. Nel primo caso uno dei due giocatori semplicemente si disconnette dalla sessione in corso; il giocatore rimanente viene così riportato

al menu principale dove potrà avviare una nuova sessione con altri giocatori. Nel secondo caso, invece, la partita termina a causa dell'esaurimento della risorsa Vita dei giocatori; in questo caso ad entrambi i giocatori verrà mostrata una schermata di riepilogo con i Punti guadagnati in partita e la possibilità di iniziare una nuova sessione di gioco.

2.2.2 Dettagli di design

In questo paragrafo vengono motivate alcune delle scelte caratterizzanti dell'applicativo presentato. Tali scelte sono alla base dell'idea di cooperazione inseguita durante lo sviluppo del progetto e hanno come scopo ultimo la diversificazione dell'esperienza dipendente dalla piattaforma di gioco.

2.2.2.1 Visibilità diversificata

Per visibilità diversificata si intende la capacità da parte del giocatore VR di osservare elementi di gioco non osservabili dal giocatore PC e viceversa. Nel dettaglio, il giocatore VR è in grado di vedere i bersagli al di fuori del campo visivo del giocatore PC e ciò comporta quindi una coordinazione dei due giocatori ai fini di colpire i bersagli e far scomparire le Catastrofi. Ulteriore elemento non visibile al giocatore PC è costituito dal tipo di Catastrofe "Seeker" (tradotto dall'inglese in "inseguitore"), la quale consiste in una singolarità cubiforme che insegue il carro armato ed esplode se lo raggiunge, arrecando danno ad esso. L' unico modo per liberarsi di una Catastrofe "Seeker" è spararla con la torretta del carro ma, non essendo visibile al giocatore PC, è compito del giocatore VR dare continue informazioni a quest'ultimo riguardanti la posizione della Catastrofe. D'altro canto, il giocatore PC gode di una visuale dall'alto, la quale può essere ruotata intorno al campo di gioco mediante il mouse. Questa caratteristica offre al giocatore PC una visione più ampia delle strategie di movimento da intraprendere per la corretta riuscita degli obbiettivi comuni. Insieme, tali diversificazioni sulla visibilità costringono i giocatori ad un continuo scambio di opinioni e tattiche e, quindi, ad una maggiore cooperazione.

2.2.2.1 Coordinazione sul movimento

Come già citato in precedenza, il giocatore VR è in grado di controllare la torretta del carro armato, i cui movimenti sono controllati a loro volta dal giocatore PC. La libertà di rotazione della torretta, però, non è assoluta ma dipende dall'orientamento della punta del carro armato. In particolare, la torretta segue la rotazione del casco di Realtà Virtuale indossato dal giocatore VR se e solo se tale rotazione rientra nei 90 gradi frontali del carro.

Superata tale soglia la torretta resterà bloccata in posizione fino a che la direzione puntata dal casco VR non rientrerà nel range di gradi permesso dai suoi limiti, sia per via di una rotazione del carro da parte del giocatore PC, sia per una rotazione del casco VR da parte del giocatore VR. Tale vincolo sulla rotazione si aggiunge agli elementi di gioco che obbligano ad una coordinazione tra i due giocatori, senza la quale, in questo caso, si ottiene l'impossibilità di difendersi dalle Catastrofi e, di conseguenza, al sicuro fallimento.

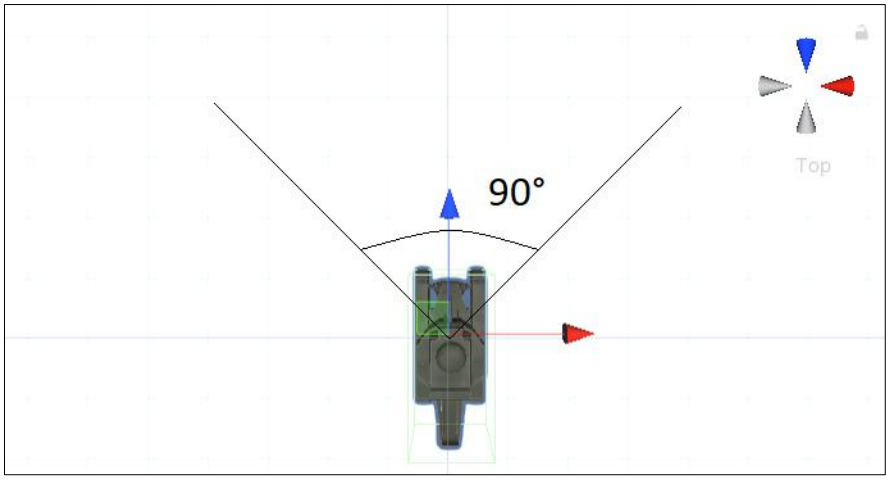

Figura 2.7: Vincoli di rotazione della torretta del carro

```
//Checks if the final position of the turret after the sync with the VR rig rotation would exceed the
boundaries of the 90° cone in front of the tank
    private bool isInBound()
    {
        bool isinbound = true;
        if (leftLimit < 0)
            leftLimit = leftLimit + 360;
        if (rightLimit >= 360)
            rightLimit = rightLimit - 360;
        if ((rightLimit < leftLimit) && (currentZeroOfCamera > rightLimit && currentZeroOfCamera <
leftLimit))
            isinbound = false;
        if ((rightLimit > leftLimit) && !(currentZeroOfCamera < rightLimit && currentZeroOfCamera >
leftLimit))
            isinbound = false;
        return isinbound;
    }
```

Listato 2.1: TurretSync.cs (Riga 33-46).

2.3 Interfaccia utente e punto di vista

In questo breve paragrafo verranno descritti i due tipi di interfacce di gioco e punti di vista, diversificati in aspetto e possibilità di visione in base alla piattaforma utilizzata. L'interfaccia utente ed il punto di vista del giocatore, seppur considerata generalmente di secondaria importanza in applicativi del genere, giocano un ruolo di grande importanza per l'esperienza utente finale.

2.3.1 Interfaccia utente VR

Per quanto riguarda la piattaforma di realtà virtuale Oculus Rift, l'interfaccia utente consiste nella barra della vita e la quantità di punti, aggiornati in tempo reale tramite callback eseguiti al cambiamento dei valori sincronizzati sul Server. Tali informazioni sono poste adiacenti tra loro e vicine al centro del campo visivo dell'utente, in modo da poter essere osservate senza movimenti troppo ampi dell'occhio durante la partita, i quali possono spesso provocare, soprattutto in giochi VR, malori e disagi visivi. Per quanto riguarda il punto di vista, esso è ovviamente in prima persona, posizionato al di sopra del Tank controllato dal giocatore PC; in tal modo il giocatore VR ha una visuale chiara dei dintorni del Tank e dei Target posizionati ai limiti del campo di gioco ma perde in coscienza d'insieme essendo la sua vista focalizzata su un'area ristretta rispetto all'intera zona esplorabile.

2.3.2 Interfaccia utente Desktop

L'interfaccia della versione Desktop possiede le stesse informazioni di quella VR ma pone quest'ultime agli angoli dello schermo, essendo la risoluzione del monitor PC di gran lunga superiore a quella dell'Oculus. Per quanto riguarda il punto di vista dell'utente PC, esso consiste in una vista dall'alto dell'area di gioco, escludendo il cerchio di generazione dei Target non visibile per via dell'apertura del campo visivo della Camera del giocatore PC. Il punto di vista PC, seppur meno effettivo nel notare dettagli come la posizione delle MineCatastrophe (fin troppo piccole per essere facilmente individuate dall'alto), eccelle nella visione di insieme, con la quale è in grado di escogitare strategie vincenti per l'attraversamento del terreno di gioco invaso dalle Catastrofi al fine di raggiungere i Punti ed aumentare il proprio punteggio. Ad avvalorare tale caratterizzante vantaggio, si è scelto di permettere al giocatore PC, tramite il movimento del mouse sull'asse delle ascisse, di ruotare la visuale intorno all'area di gioco (preservando, però, la sua impossibilità di vedere i Target) in modo tale da permettergli una visione chiara del Tank e dei Punti anche in casi in cui le Catastrofi presenti in partita oscurino quest'ultimi al giocatore per mezzo della loro sovrapposizione visiva.

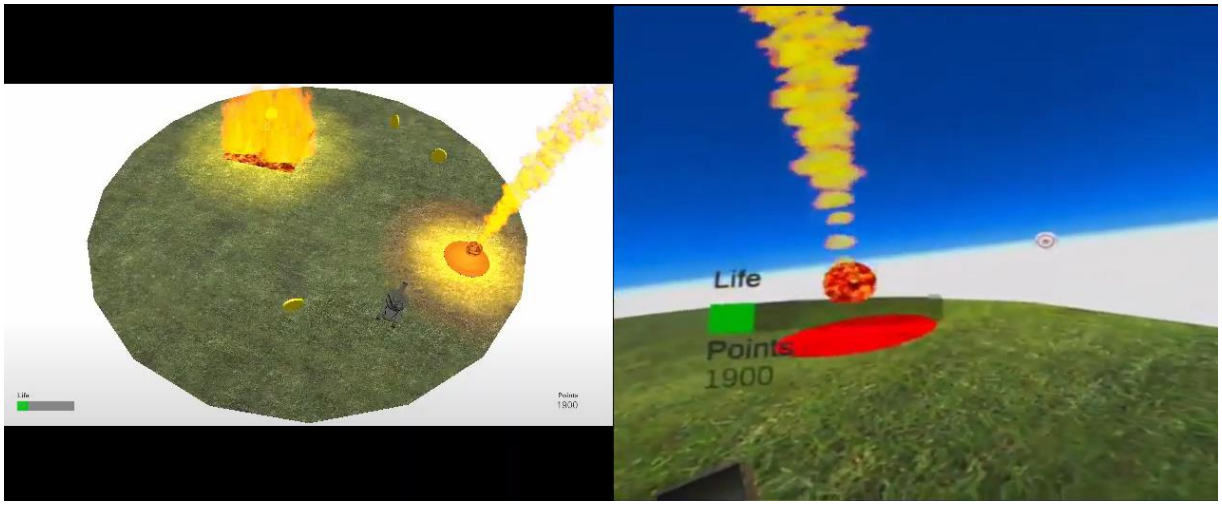

Figura 3.6: I due punti di vista diversificati a confronto

Capitolo 3

Tecnologie utilizzate

In questo capitolo verranno descritti tutti i concetti tecnici di maggiore rilevanza sfruttati per il completamento del progetto proposto e ne verranno spiegati gli utilizzi. Il capitolo inizia con una panoramica degli elementi fondamentali che dominano lo sviluppo nell'ambiente Unity 3D, per poi virare su elementi più specifici necessari per la realizzazione dell'applicativo.

Infine, verrà presa in considerazione l'interfaccia utente disponibile in versione diversificata su entrambe le piattaforme di gioco disponibili e ne verranno evidenziate le differenze.

3.1 Introduzione a Unity3D

Il progetto di tesi proposto è stato interamente sviluppato mediante l'ausilio del motore grafico Unity. «Unity è un motore grafico multipiattaforma sviluppato da Unity Technologies che consente lo sviluppo di videogiochi e altri contenuti interattivi, quali visualizzazioni architettoniche o animazioni 3D in tempo reale». Progettato inizialmente per il settore videoludico, si è con il tempo evoluto in una piattaforma che offre servizi di più ampia estensione, tra i quali cinematografia, edilizia, architettura, e molti altri. Il punto di forza della piattaforma, sul quale Unity Technologies hanno posto maggiore attenzione, è la semplicità di utilizzo; rispetto ai suoi concorrenti, infatti, Unity offre un ambiente di facile comprensione per lo sviluppatore e vanta di un'ottima (ed aggiornata) documentazione online [7] e una vasta gamma di tutorial e guide offerte sia dalla community di sviluppatori, sia dal gruppo di Unity Technologies [8].

3.1.1 Motori grafici

Come riportato nell'introduzione di questo capitolo, Unity3D è un motore grafico, ma cosa si intende esattamente per motore grafico? «Il motore grafico è il nucleo software di un videogioco o di qualsiasi altra applicazione con grafica in tempo reale. Esso fornisce le tecnologie di base, semplifica lo sviluppo, e spesso permette al gioco di funzionare su piattaforme differenti come le console o sistemi operativi per personal computer. La funzionalità di base fornita tipicamente da un motore grafico include un motore di rendering per grafica 2D e 3D un motore fisico o rilevatore di collisioni, suono, scripting, animazioni, intelligenza artificiale, networking e scene-graph» [9]. Ogni motore grafico è composto da due strati. Quello di basso livello, progettato da tecnici del settore, che definisce tutte le funzionalità del motore, e quello di alto livello, che impacchetta tali funzioni, pronte per essere utilizzate semplicemente da chi poi va a sviluppare il videogioco.

Figura 2.1: Interfaccia utente (UI) di Unity3D.

3.1.2 Le potenzialità di Unity3D

Come già citato in precedenza, Unity3D offre numerosi vantaggi agli sviluppatori che lo utilizzano, ciononostante, anche tale ambiente presenta alcuni difetti che devono essere presi in considerazione prima di procedere nella creazione di un'esperienza di gioco. Di seguito vengono descritti i pro e i contro del motore grafico Unity3D.

3.1.2.1 I Pro

Di sicuro il primo vantaggio tra tutti è la natura gratuita ed aperta di Unity: nello store ufficiale della piattaforma sono presenti centinaia di migliaia di componenti gratuiti creati dalla community facilmente importabili nel proprio progetto. Altro vantaggio è la comodità e semplicità di utilizzo del motore che, nella maggioranza dei casi, risulta essere molto intuitivo al programmatore medio; questa parvenza di semplicità nasce anche dal set di API proprietarie di Unity utilizzabili nello scripting che, con poche righe di codice, permettono di svolgere funzioni anche complesse. Inoltre, nei recenti aggiornamenti della piattaforma è stato aggiunto un tool che permette agli utenti che non hanno conoscenze elevate di programmazione di sviluppare un videogioco mediante un linguaggio visuale a blocchi chiamato Bolt. Tramite Bolt il programmatore è in grado di gestire il comportamento dei vari componenti di gioco tramite dei blocchi collegati da "fili logici"; tali fili

"trasportano" informazioni da un blocco ad un altro e determinano le dipendenze tra i vari elementi. Un ulteriore vantaggio di Unity è la semplicità con la quale può essere gestito lo sviluppo di applicazioni cross-platform, data dalla possibilità di compilare il proprio progetto scegliendo da una vasta lista di piattaforme di destinazione; le API di Unity prevedono, inoltre, moltissime funzionalità che gestiscono le diversità delle piattaforme per le quali si sta sviluppando.

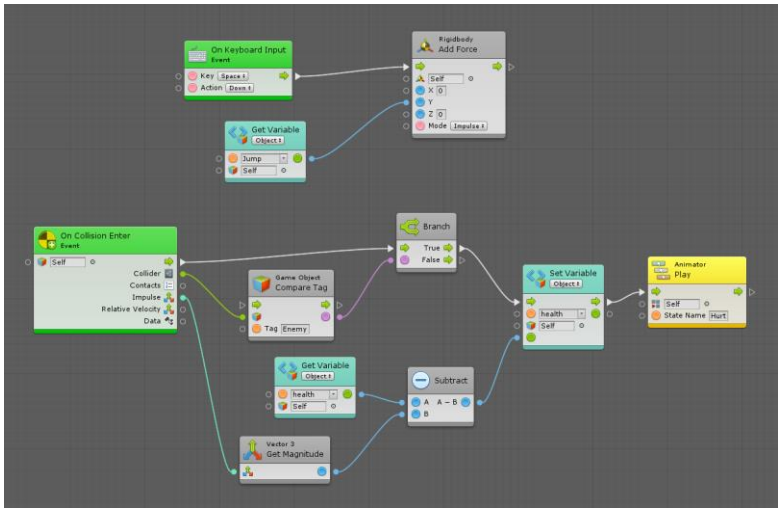

Figura 2.2: Un esempio di comportamenti definiti tramite Bolt.

3.1.2.2 I Contro

Tra gli aspetti negativi più nominati di Unity troviamo una carenza di aggiornamenti della documentazione che, per quanto dettagliata, mostra spesso descrizioni di funzionalità superate o diverse dal presente e ciò porta ovviamente ad una confusione non banale in caso di sviluppo mediante le suddette tecnologie. Altro importante contro da nominare è la mancanza di un editor del terreno di alto livello; l'editor presente in Unity non offre molte possibilità e la maggior parte delle volte ci si vede costretti ad affidarsi a plugin di terze parti per la realizzazione del terreno nel proprio gioco. Sui progetti grandi, infine, Unity non regge il confronto con gli altri contendenti data la sua scarsa ottimizzazione ed essendo che l'engine rimarrà probabilmente a 32 bit, ciò rappresenta un tappo di bottiglia per lo sviluppo di giochi di ampia portata.

3.1.2.3 Il perché della scelta

Nonostante Unity non sia esente da problemi, la scelta di tale engine per lo sviluppo del progetto di tesi è stata guidata dalla semplicità di utilizzo e comprensione, dalla comodità della gestione di componenti cross-platform e dalla velocità di creazione di componenti custom data dalla vasta presenza di microcomponenti gratuiti reperibili sull'Asset Store di Unity. Infine, è stato di grande aiuto il supporto nativo di Unity per lo sviluppo di applicazioni VR, ormai da tempo pietra miliare della piattaforma.

3.1.3 Un'altra alternativa: Unreal Engine

Unreal Engine è un altro motore grafico, sviluppato da Epic Games, mostrato per la prima volta nel 1998. Nonostante sia stato inizialmente pensato per giochi sparatutto in prima persona, è stato usato per giochi di tutti i generi. La prima differenza tra i due motori è la semplicità offerta da Unreal Engine nell'ottenere ottimi risultati in ambito di qualità visiva, punto di forza del motore. Inoltre, Unreal Engine è pensato per grandi progetti in cui il team di sviluppo è composto da molte persone, ognuna focalizzata nella cura di un particolare aspetto del gioco; in Unity invece lo sviluppo è in genere gestito da un minor numero di persone perché le funzionalità sono più semplici e non così personalizzabili come in Unreal Engine. Per quanto riguarda il progetto di tesi, però, Unreal Engine non sarebbe stato sfruttato a pieno e non si avrebbe avuto la semplicità di sviluppo cross-platform offerta da Unity.

3.2 Concetti di Unity3D

Unity 3D, come già detto in precedenza, offre un ambiente di facile comprensione al programmatore medio; caratteristica offerta sia dalla semplicità delle API proprietarie utilizzabili negli script, sia dall'intuitività della GUI di sviluppo. Esistono però alcuni concetti di base su cui l'intera programmazione in Unity affonda le radici, che non sono presenti del tutto all'interno dei paradigmi comuni della programmazione e, per tal motivo, vanno descritti con maggior riguardo.

3.2.1 GameObjects

"Ogni oggetto all'interno del tuo gioco è un GameObject, dai personaggi agli oggetti collezionabili, persino le luci, le camere e gli effetti speciali. Tuttavia, un GameObject non può fare nulla in sé; è necessario che gli si vengano assegnate delle proprietà prima che possa diventare un personaggio, un ambiente o un effetto speciale" [13]. Un GameObject è quindi un oggetto fondamentale in Unity necessario a rappresentare qualsiasi tipo di elemento all'interno di un gioco. Come già citato, però, un GameObject è in sé solo un componente logico e inattivo finché non vengono ad esso assegnate delle proprietà e dei comportamenti che gli permettono di interagire con l'ambiente di gioco e di assumere vita propria. Tali componenti assegnabili sono un ulteriore elemento fondamentale di Unity e vengono chiamati "Components".

3.2.2 Components

I Component in Unity possono essere visti come le diverse caratteristiche di un GameObject. Se considerassimo, a scopo esemplificativo, una persona come GameObject, troveremmo tra i suoi Component la sua posizione, la sua altezza, il colore della sua pelle, la frequenza della sua voce, i suoi tratti caratteriali, e così via. In Unity il Component fondamentale, presente in ogni oggetto, è il Transform Component. Tale componente definisce la posizione, la rotazione e la scala di un GameObject all'interno del mondo di gioco; essendo un componente di importanza assoluta, non può essere rimosso da nessun GameObject, compresi gli Empty GameObject, ovvero i GameObject senza alcun Component. Il Transform Component abilita, inoltre, il concetto di "parenting", ossia la possibilità di assegnare un GameObject come "figlio" di un altro (che diventerà il "padre") [14] in modo tale da far ereditare ad esso alcuni comportamenti del componente padre (seguendo l'esempio precedente, i vestiti sono GameObject figli del GameObject "Persona" perché seguono i suoi movimenti e condividono la medesima posizione nel mondo).

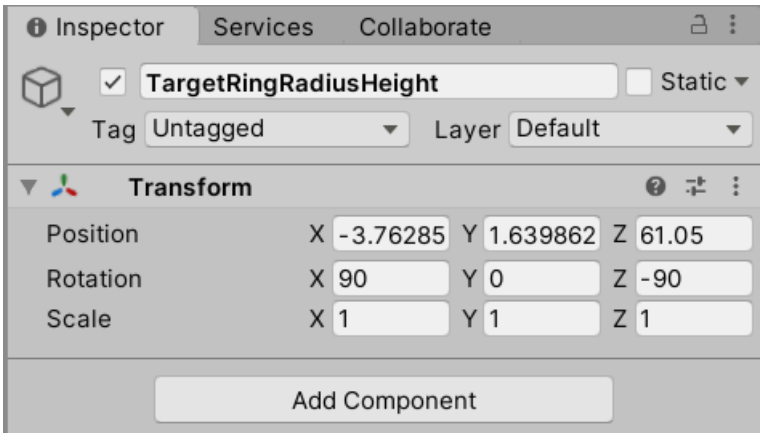

Figura 3.1: Esempio di Transform Component in Unity3D

In Unity esistono centinaia di Component diversi che possono essere assegnati ad un GameObject per fargli assumere un determinato comportamento, ma spesso capita che durante lo sviluppo di un gioco ci si rende conto di aver bisogno di un Component personalizzato ad-hoc per l'applicazione che si sta cercando di creare. In tal caso la soluzione migliore è quella di programmare un Component personale tramite il meccanismo di Scripting offerto da Unity.

3.2.3 Scripts

Come appena descritto, lo Scripting è un meccanismo di Unity3D che permette la creazione da 0 di Component personalizzati da poter assegnare ai vari GameObject all'interno del mondo di gioco. Gli Script permettono di attivare eventi di gioco, modificare proprietà di altri Component nel tempo e reagire agli input dell'utente. Unity supporta il linguaggio di programmazione C# come linguaggio nativo per la programmazione di Script, ma ciò non esclude l'utilizzo di altri linguaggi di programmazione .NET, a patto che possano compilare un DLL compatibile. In genere ogni nuovo Script creato in Unity consiste in una Classe che eredita dalla già esistente Classe "MonoBehaviour", dalla quale si ottengono tutte le API di Unity per il controllo degli eventi di gioco e per la modifica delle caratteristiche dei Component. La Classe MonoBehaviour rappresenta quindi il comportamento generico che tutti i GameObject hanno all'interno del mondo di gioco, e di essa sono utilissimi alcuni metodi, dei quali in genere ne viene utilizzato almeno uno in ogni Script; citando i più importanti, si ha il metodo Start, eseguito all'istanziamento del GameObject nell'ambiente e il metodo Update, eseguito una volta in ogni frame di esecuzione del gioco

(quindi 60 volte al secondo, a patto che non vi siano elevati rallentamenti nell'esecuzione). Utilizzando tali metodi e implementandone di ulteriori, si può velocemente creare un Component personalizzato in base alle necessità di sviluppo da assegnare ai propri GameObject.

```
private void Start()
    {
        manager = GameObject.Find("TanksNetworkManager").GetComponent<TanksNetworkManager>();
        bullet_spawn = GameObject.Find("BulletSpawn");
        devices = new List<InputDevice>();
        InputDevices.GetDevicesWithCharacteristics(InputDeviceCharacteristics.Right |
InputDeviceCharacteristics.HeldInHand | InputDeviceCharacteristics.Controller, devices);
        if (devices.Count > 0)
        {
            device = devices[0];
        }
    }

    void Update()
    {
        if (device != null)
        {
            bool triggerValue;
            if (device.TryGetFeatureValue(UnityEngine.XR.CommonUsages.triggerButton, out triggerValue) &&
triggerValue)
            {
                Fire();
            }
        }
    }
```

Listato 3.1: HandHeldController.cs (Riga 20-42).

3.2.4 Prefabs

I Prefab sono una delle feature più utili in assoluto all'interno dell'ambiente Unity. Essi permettono di creare, configurare e memorizzare un GameObject insieme a tutti i Component ad esso collegati, le sue proprietà e i suoi GameObject figli, come un elemento riutilizzabile [15]. Quando si vuole riutilizzare un GameObject in un modo particolare – come un personaggio non giocabile (NPC) o un elemento dello scenario – in più parti del livello che si sta creando o in più livelli, si può semplicemente convertire tale GameObject in un Prefab. In tal modo è possibile inserire più istanze di tale Prefab dove è necessario e qualsiasi cambiamento effettuato sul Prefab verrà riflesso su tutte le sue istanze, permettendo così uno sviluppo più rapido ed evitando di dover copiare e incollare lo stesso GameObject più volte per poi essere costretti ad aggiornare i cambiamenti effettuati in seguito manualmente su tutte le copie inserite nell'ambiente di gioco. Ciò non vuol dire, però, che tutte le istanze di un Prefab debbano per forza essere uguali tra di loro; Unity permette infatti di sovrascrivere alcune proprietà di un

Prefab solo in alcune istanze di esso, garantendo quindi una personalizzazione assoluta in corso di sviluppo. Un altro utilizzo comune dei Prefab consiste nello spawning di GameObject durante il runtime, ossia la creazione nel mondo di gioco di elementi interattivi durante l'esecuzione del gioco. Anche in tal caso è necessario rendere tali GameObject dei Prefab in modo da potervici accedere con lo Scripting e permettere la loro creazione durante il runtime. Nel caso del progetto di tesi, sono stati creati numerosi Prefab, i più importanti tra i quali sono sicuramente quelli rappresentanti le Catastrofi. Ogni Catastrofe, infatti, non è altro che un Prefab che, durante l'esecuzione, viene istanziato da uno script denominato "CatastropheSpawner", il quale verrà in seguito descritto.

3.2.5 Animators e Animations

Unity, come la maggioranza dei motori grafici, presenta un sistema interno per la gestione delle animazioni. Semplificando a grandi linee il funzionamento di tale sistema: ad ogni GameObject può essere assegnato un Animator Controller, il quale non è altro che un Component che permette di animare il GameObject in base a delle animazioni create in precedenza sotto forma di Animation Clips. L'Animator Controller permette di gestire più stati del GameObject e definisce quale animazione deve essere eseguita in ognuno di essi. Si pensi all'animazione del GameObject Persona presentato in precedenza; il comportamento di tale GameObject può essere racchiuso in quattro stati principali (a scopo di esempio): Fermo, Camminata, Corsa, Salto. Ad ognuno di essi l'Animator Controller permette di associare un Animation Clip da eseguire durante la permanenza in tale stato. Permanenza definita tramite il sistema di Scripting descritto precedentemente. Ogni stato, inoltre, può essere suddiviso in sottostati, garantendo così un realismo maggiore nella gestione delle animazioni. Il Salto, ad esempio, potrebbe essere suddiviso in un sottostato di Preparazione, uno di Slancio, uno di Volo o Fase Aerea, uno di Discesa ed infine uno di Atterraggio, in modo tale da rendere anche possibile l'interruzione di un salto da un evento di gioco come la collisione con un muro o un altro oggetto di gioco.

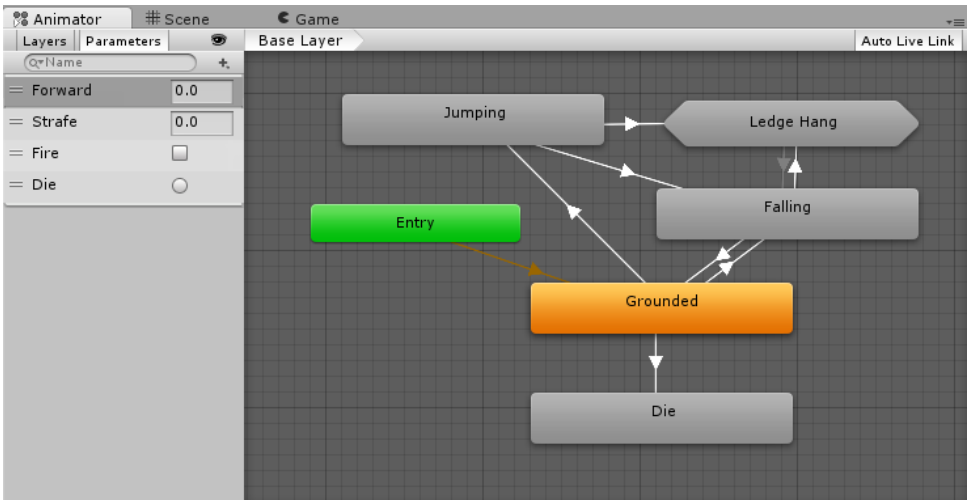

Figura 3.2: L'Animator Controller gestisce gli stati di un GameObject come nodi in un grafo

La creazione delle Animation Clips per le singole animazioni è interamente gestita da Unity e viene realizzata tramite uno specifico editor che permette l'osservazione del cambiamento di una determinata proprietà di un GameObject nel tempo, in modo da registrare tale cambiamento ed eseguirlo come animazione. Ad esempio, per la realizzazione di un'animazione molto semplice come una di fluttuazione, basta osservare la posizione verticale di un GameObject (asse Y), salvare il suo valore nello stato iniziale, spostare il GameObject nel punto di massima elevazione scelto per l'animazione e salvare anche tale valore. Saranno ora presenti nella linea temporale dell'Animation Clip due momenti temporali definiti, il primo avente elevazione minima e il secondo avente elevazione massima. Copiando il primo momento temporale (o Key Frame) e incollandolo alla fine dell'animazione si ottiene così un loop che porta l'elevazione del GameObject dallo stato iniziale a quello di massima elevazione e di nuovo a quello iniziale. Nel progetto presentato, sono state utilizzate diverse animazioni, tra cui la fluttuazione della Catastrofe "Seeker" e la rotazione e la fluttuazione dei Punti. Le animazioni, in genere, hanno come scopi possibili quello di aumentare l'immersione del giocatore all'interno del mondo virtuale presentandolo ad elementi che si comportano in modo simile al mondo reale, o semplicemente quello di rendere più piacevole la presenza scenica di un GameObject animandone alcune parti, e in generale tali animazioni non hanno un impatto sul gameplay. Altre volte, invece, le animazioni hanno un impatto diretto sul gioco; esse, infatti, possono essere applicate anche a proprietà quali la posizione nel mondo di un GameObject ottenendo quindi un risultato non unicamente grafico.

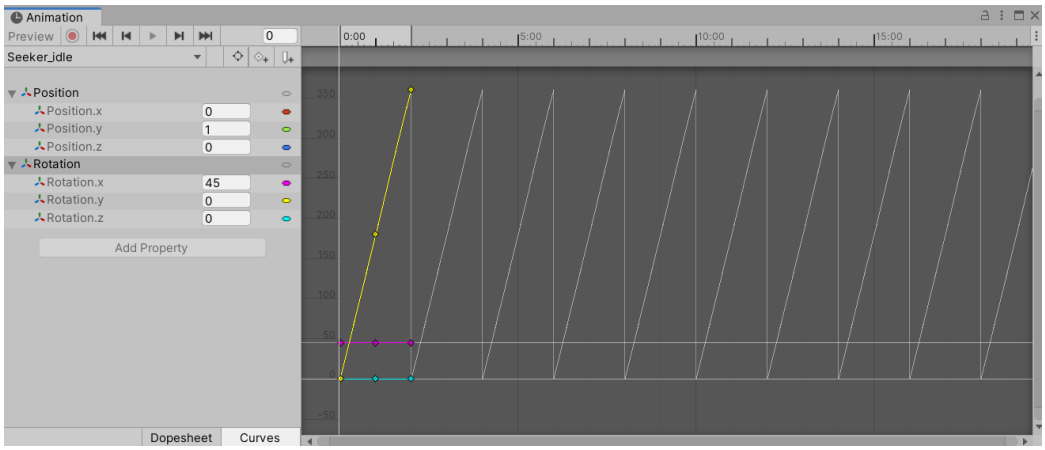

Figura 3.3: L'Animation Clip di rotazione della Catastrofe "Seeker". Sulla destra si può notare la timeline con i cambiamenti della proprietà di rotazione nel tempo.

3.2.6 Scenes

Le Scene contengono gli ambienti e i menu di un gioco, insieme a tutti gli oggetti presenti nel gioco in una determinata fase. Si può pensare ad esse come ai vari livelli unici di un gioco. In ogni Scena vengono posizionati ambienti, ostacoli, decorazioni, luci, progettando, essenzialmente, l'intero gioco in parti di dimensione minore. Il flusso di gioco attraverso le Scene esistenti è gestito tramite Script da un componente interno di Unity denominato "SceneManager" tramite il quale, ad esempio, si può caricare una specifica Scene al verificarsi di un determinato evento in gioco. Nel progetto di tesi è in realtà presente un'unica Scene, ma viene comunque utilizzato lo SceneManager per ricaricare tale Scene in caso di GameOver.

```
public override void OnClientDisconnect(NetworkConnection conn)
    {
        base.OnClientDisconnect(conn);
        SceneManager.LoadScene(0);
    }

    public override void OnStopHost()
    {
        base.OnStopHost();
        SceneManager.LoadScene(0);
    }
```

Listato 3.2: TanksNetworkManager.cs (Riga 88-98)

3.3 Il sistema di Networking

In questo paragrafo verranno descritte le tecniche e le tecnologie utilizzate per la realizzazione dell'architettura di rete dell'applicativo presentato, insieme ad esempi pratici di utilizzo con relativi codici citati. Il principale elemento che ha reso possibile gran parte della sincronizzazione tra i client utenti è la libreria Mirror di Unity.

3.3.1 Introduzione a Mirror

Mirror è la diretta libreria sostitutiva dell'ormai deprecata UNet API con il maggior grado di compatibilità. Essa, infatti presente quasi tutti i componenti e le caratteristiche di UNet, facendo sì che il networking all'interno del progetto sia semplice e di facile manutenzione sia per progetti nuovi sia per la manutenzione di vecchi progetti. Le potenzialità di Mirror sono numerose, ma sicuramente un grande pregio risiede nell'immediata gestione di carichi anche elevati di connessioni diverse e nella sincronizzazione dei componenti dei GameObject in gioco del tutto intuitiva [17]. Per quanto riguarda l'implementazione, Mirror offre diversi Component e Prefab per iniziare rapidamente a rendere il proprio gioco Networked. Il principale tra questi, su cui si basa l'intero sistema, è il NetworkManager.

3.3.2 NetworkManager

Il NetworkManager è il fulcro del networking tramite Mirror. Esso offre le API di gestione della connessione da parte dei client, la creazione di un Server/Host, la chiusura di connessioni e moltissimi tipi diversi di Listener per associare ad ogni evento legato al Networking una specifica serie di azioni da compiere. L'implementazione di un NetworkManager nella propria applicazione consiste, semplicemente, nel creare un nuovo GameObject vuoto ed aggiungere ad esso il NetworkManager component. Tra le varie proprietà modificabili del NetworkManager, quelle che assumono maggior importanza sono le seguenti: Il NetworkAddress, ovvero l'indirizzo IP del Server verso il quale i Client devono connettersi. Il Player Prefab, ossia il Prefab del GameObject che rappresenta il giocatore; tale Prefab verrà in automatico istanziato dal NetworkManager quando un giocatore si connette

al Server. Gli Spawnable Prefabs, ossia una lista di Prefabs che il Manager può "spawnare" (istanziare in modo sincronizzato) su tutti i Client.

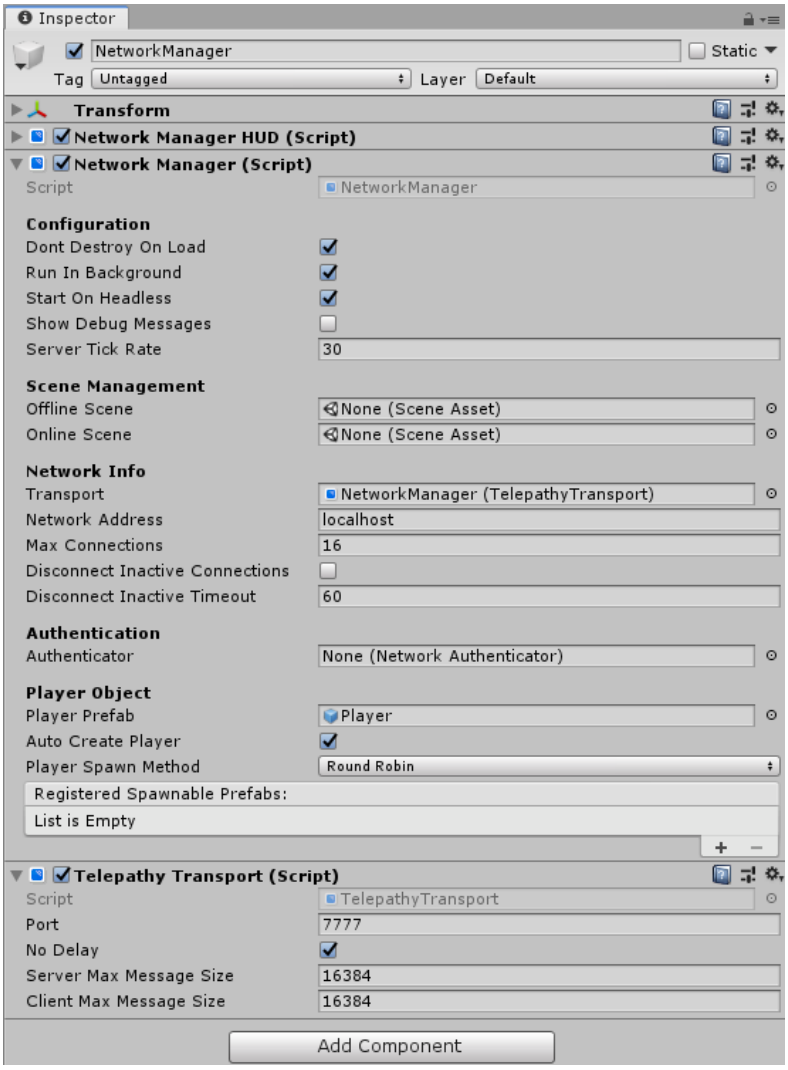

Figura 3.4: Inspector del NetworkManager con le proprietà citate

In aggiunta a tali proprietà, come già nominato in precedenza, il NetworkManager offre la possibilità di sfruttare i suoi Listener per collegare a determinati eventi di rete una reazione definita via codice. Tra di essi troviamo il metodo OnServerAddPlayer, eseguito alla connessione di un nuovo Client al Server attivo; OnClientDisconnect, eseguito alla disconnessione di un Client in precedenza connesso al Server attivo; OnStopHost/OnStopServer, eseguiti quando viene effettuata la chiusura dell'Host/Server attivo. I metodi appena citati vengono utilizzati nell'applicazione presentata per far iniziare la partita alla connessione di entrambi i giocatori, inizializzando tutti i componenti necessari, e per riavviare la Scena nel caso in cui uno dei due giocatori si disconnette.

```
public override void OnServerAddPlayer(NetworkConnection conn)
    {
        switch (numPlayers)
        {
            case 0:
                {
                    GameObject player = Instantiate(playerPrefab, Vector3.zero, Quaternion.identity);
                    controlScript = player.GetComponent<TankControl>();
                    NetworkServer.AddPlayerForConnection(conn, player);
                    tank = player;
                    soldier_spawn = GameObject.Find("Soldier Spawn");
                    TankCameraActivator cam_activator = tank.GetComponent<TankCameraActivator>();
                    cam_activator.my_cam = MainCamera;
                    if (MainCamera != null)
                    {
                        cam_activator.my_audio = MainCamera.GetComponent<AudioListener>();
                    }
                    GameManager gameManager = GameObject.Find("GameManager").GetComponent<GameManager>();
                    gameManager.PCGameOverMenu = PCGameOverMenu;
                    gameManager.deactivatePCGameOverMenu();
                    break;
                }
```

Listato 3.3: TanksNetworkManager.cs (Riga 21-49)

Sia il Player Prefab che i Prefab nella lista di Spawnable Prefabs necessitano di una caratteristica per poter essere "spawnati" dal NetworkManager: devono essere dei Networked Object. Un Networked Object è un GameObject al quale vengono aggiunti dei Networking Component.

3.3.3 Networking Component

I Networking Component sono dei Component che permettono la sincronizzazione dello stato e delle proprietà dei GameObject a cui sono collegati tra i vari Client connessi al Server. Il più importante tra tutti, senza il quale gli altri Networking Component perdono il proprio funzionamento, è la NetworkIdentity.

3.3.3.1 NetworkIdentity

La NetworkIdentity consiste in un'identità univoca sulla rete dell'applicazione che permette di identificare i singoli elementi condivisi tra i

Client e attuare logica su di essi. La NetworkIdentity permette il funzionamento di tutti gli altri Networking Component. Tale componente offre inoltre un attributo, denominato "Server Only" che, se attivato, impedisce che il GameObject a cui la NetworkIdentity è associata, venga "spawnato" sui Client, divenendo quindi un oggetto proprietario del Server. Questa proprietà è molto utile nei casi in cui si vuole avere un oggetto che goda di una maggiore autorità sui flussi logici del gioco e, quindi, si vuole evitare che un Client possa accedervi per questioni di sicurezza.

3.3.3.2 NetworkTransform

All'interno di un'applicazione di rete in cui più Client interagiscono all'interno della stessa area di gioco è di vitale importanza la totale sincronizzazione delle esperienze di gioco. Si pensi ad un gioco competitivo: se un giocatore non dovesse avere una vista sincronizzata sulla partita in corso potrebbe essere un grande svantaggio nei confronti degli altri giocatori. In generale per assicurare un'equa esperienza alla totalità degli utenti la sincronizzazione è un dovere dello sviluppatore. Il NetworkTransform Component aiuta proprio a raggiungere tale scopo; esso, infatti, permette di sincronizzare in automatico, senza dover implementare complicati sistemi di gestione asincrona, il Transform Component del GameObject al quale si collega tale componente su tutti i Client su cui il Server ha "spawnato" quest'ultimo. In altre parole, una volta che un GameObject con collegato il NetworkTransform Component viene "spawnato" da un Server su più Client, qualsiasi movimento o rotazione di esso all'interno della scena sarà replicato su tutti i Client connessi in modo da unificare l'esperienza di gioco. Un esempio di utilizzo nel progetto presentato di tale Component risiede nella sincronizzazione realizzata sul oggetto Tank, la quale posizione e rotazione deve essere sincronizzata sia sul Client Oculus sia su quello PC. Tra le varie proprietà del NetworkTransform, una di particolare importanza è la "Client Authority" che permette una completa personalizzazione dei permessi che i vari elementi di rete hanno sul GameObject in questione e che verrà trattata in seguito.

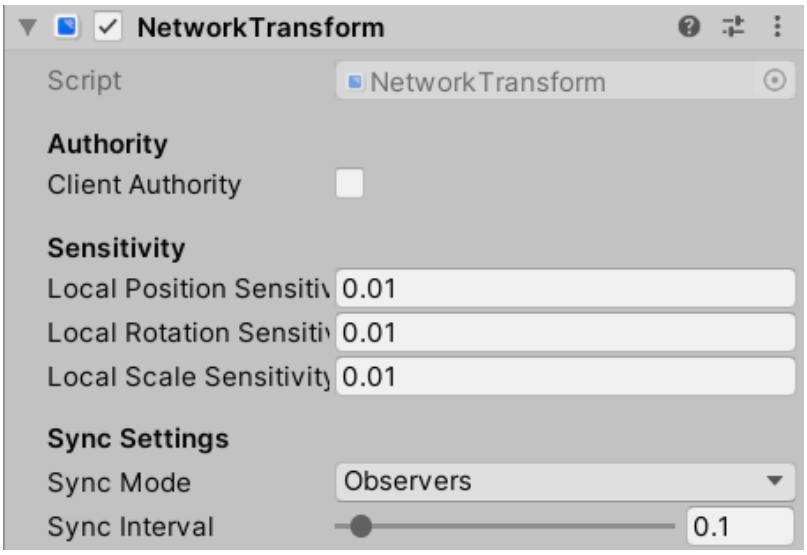

Figura 3.5: Inspector del NetworkTransform

3.3.3.3 NetworkTransformChild

Quando si sincronizza un GameObject tramite NetworkTransform, viene sincronizzata la sua posizione e rotazione ma non quelle dei suoi componenti figli. Questo perché quando si hanno componenti innestati sincronizzati sulla rete non si può assegnare loro una NetworkIdentity se non sul GameObject radice ed essendo che il NetworkTransform può essere associato solo ad un GameObject con una NetworkIdentity, non vi è alcun modo di associare un NetworkTransform ai suoi componenti figli; per risolvere tale problematica, si utilizza il componente apposito NetworkTransformChild, che va aggiunto alla radice di un GameObject di rete innestato; di conseguenza, tramite le proprietà dell'Inspector di tale Component si seleziona il GameObject figlio della radice di cui si vuole sincronizzare posizione e rotazione sulla rete. Il NetworkTransformChild è stato utilizzato nell'applicativo presentato per la sincronizzazione dei figli della maggior parte degli elementi di rete come, ad esempio, per la torretta del Tank, la quale è figlia del GameObject Tank e la cui rotazione deve essere sincronizzata sia sul Client PC che su quello Oculus per permettere la mira ed il corretto fuoco sui bersagli.

3.3.3.4 Authority

Come citato in precedenza, tra le varie proprietà del NetworkTransform (e del NetworkTransfomChild) spicca la "Client Authority". Tale proprietà stabilisce se l'autorità sul movimento e sulla rotazione del GameObject in questione appartiene al Client sul quale il GameObject viene "spawnato" oppure (in caso venga lasciata non selezionata) del Server attivo al quale sono connessi i Client in gioco. Di default la Authority su un Networked GameObject è del Server che lo ha "spawnato" ma tramite tale proprietà, ciò può essere cambiato. In generale la Authority è un modo di decidere chi è il proprietario di un oggetto e chi, di conseguenza, ha il controllo su di esso. Se su di un oggetto vi è una Server Authority, ciò vuol dire che il server ha controllo di quest'ultimo (default) e ne può controllare tutti i movimenti. Se, invece, su di un oggetto si ha una Client Authority, ciò vuol dire che il Client ha controllo sull'oggetto e che può quindi chiamare delle funzioni Command (chiamate eseguite sul Server) per cambiarne la posizione o la rotazione e, inoltre, tale oggetto verrà distrutto in caso di disconnessione del Client. L'Authority generica di un GameObject (da non confondere con quella sul Transform definita tramite il NetworkTransform) può essere assegnata ad un Client tramite la chiamata del metodo AssignClientAuthority(conn), il quale assegnerà la Authority al Client connesso sulla connessione contenuta nel parametro "conn" [18]. Un metodo riguardante l'Authority utile a conoscere il proprietario di un GameObject è "hasAuthority"; tale metodo permette di controllare se il codice attuale sta venendo eseguito dal proprietario dell'oggetto. Inizialmente tale metodo era utilizzato nel progetto per capire se il codice stava venendo eseguito sul Client oppure sul Server, il che è un problema molto importante quando si gestisce un Host che svolge funzioni sia di Client che di Server. Non avendo mai assegnato la Authority di un GameObject ad un Client in particolare, tutti gli oggetti di rete del progetto hanno la Server Authority, eccetto quelli rappresentanti il giocatore (Player Prefab) i quali hanno come proprietari di default i Client connessi. Quindi se il metodo "hasAuthority" dovesse restituire il valore "true" in qualsiasi caso eccetto quello in cui venga eseguito sul PlayerPrefab, ciò vorrebbe dire che il codice attuale sta venendo eseguito dal Server. Con questa tecnica sono state partizionate le sezioni di codice di GameObject presenti sia sul Client che sul Server per evitare che il Client esegua codice dedicato all'esecuzione su Server. In seguito, però, è stata cambiata l'infrastruttura che permetteva la "spawning" delle Catastrofi: in fasi iniziali del progetto, infatti, le Catastrofi venivano istanziate sia sui Client che sul Server in posizioni condivise da tutti, ma non ne veniva sincronizzato il ciclo di vita, affidandosi all'impossibilità di comportamenti diversi di esse su uno dei Client essendo esse di tipo

deterministico. Purtroppo, tale tecnica non prevedeva la differenza di velocità di esecuzione dei vari Client il che portava inevitabilmente ad incongruenze. Per la risoluzione di tale problema si è optato per lo "spawning" delle Catastrofi sul Server e la sincronizzazione sui Client tramite il metodo NetworkServer.Spawn(GameObject obj), il quale permette la sincronizzazione di un oggetto presente sul Server su tutti i Client. Cambiato il sistema di spawning, il metodo "hasAuthority" è stato sostituito con il più consono "isServer", il quale restituisce "true" se e solo se il codice sta venendo eseguito su un GameObject che è presente sul Server attivo. In questo modo si riesce molto facilmente a suddividere il codice tra Client e Server, a patto che esso sia presente in GameObject "spawnati" tramite il metodo descritto in precedenza.

```
public void OnTriggerEnter(Collider other)
    {
        if (other.CompareTag("Player")){
            if (isServer)
            {
                playerController.takeDamage(gameObject.GetComponent<Catastrophe>().getDamage());
            }
        }
    }
```

Listato 3.4: MeteorCatastrophe.cs (Riga 94-102)

3.3.4 Spawning

Più volte, nel corso di questo capitolo, si è parlato di "spawning". Tale termine, se utilizzato in un modello avente autorità lato Server, indica la creazione di un GameObject, presente sul Server, su tutti i client connessi ad esso. Tale GameObject sarà gestito dal sistema di spawning; in particolare, ogni qual volta lo stato di quest'ultimo cambia, tale cambiamento verrà

propagato su tutti i Client in cui il GameObject è stato spawnato. Questo meccanismo permette ad un Client di connettersi anche in seguito ad altri già connessi e sincronizzare la propria vista della scena con quella di tutti gli altri semplicemente spawanando su di esso tutti i Networked GameObject attivi al momento. Nel progetto di tesi il caso maggiore di Spawning avviene per le Catastrofi e per i Punti: sul Server sono presenti due GameObject denominati rispettivamente "CatastropheSpawner" e "PickupSpawner". Essi vengono attivati all'ingresso in partita del secondo giocatore (quindi quando sono presenti tutti i giocatori necessari alla partita) e hanno come scopo lo spawning in posizioni pseudo-casuali (sono controllati da algoritmi che eliminano alcuni casi limite come lo spawn di Catastrofi sulla posizione attualmente occupata dal giocatore) di Catastrofi e Punti. Tali GameObject sono quindi sincronizzati sui Client dal sistema di spawning descritto in precedenza. In tal modo qualsiasi interazione tra le Catastrofi e il giocatore o tra i Punti e il giocatore viene gestita del tutto lato Server (utilizzando il metodo isServer per capire quando l'esecuzione avviene sul Server). Se, ad esempio, il giocatore entra a contatto con una Catastrofe, la collisione viene registrata sul Server il quale cambia il valore della variabile sincronizzata contenente la vita attuale del giocatore e tale cambiamento viene propagato infine ai Client, i quali aggiornano la loro interfaccia per mostrare il nuovo valore della vita.

3.4 Le Catastrofi ed i Networked Prefabs

In questo paragrafo verrà spiegato il significato di Networked Prefab e verranno descritti i Networked Prefab osservabili durante una partita. In particolare, verranno descritte tutte le tipologie di Catastrofi ed i dettagli del loro metodo di spawning, i Target associati ad esse ed il loro funzionamento, ed infine il sistema di Punti presente nel gioco.

3.4.1 Cos'è un Networked Prefab

È stato già citato come, tramite il sistema di spawning, dei GameObject presenti sul server vengono istanziati e sincronizzati sui Client. Il Network Manager, però, può spawnare solo ed unicamente GameObject da Prefab registrati nella lista presente all'interno del suo Inspector (vedi "Network

Manager"). Per poter essere registrati in tale lista, i Prefab in questione devono essere Networked, ovvero devono possedere una NetworkIdentity; senza di essa, infatti, un GameObject non potrebbe essere spawnato su più client e sincronizzato, perché non esisterebbe un modo di identificare tale oggetto univocamente sulla rete. I Networked Prefab spawnabili dal NetworkManager nel progetto presentato sono i Proiettili sparati dal Tank, i quali non necessitano una descrizione approfondita dato il loro funzionamento di banale comprensione, le Catastrofi, i Target ed i Punti.

3.4.2 Catastrofi

Le Catastrofi sono il fulcro del gameplay dell'applicativo in esame; esse applicano un fattore di tensione e pericolo imminente ai giocatori, i quali sono obbligati a adattarsi velocemente ad esse e collaborare per sopravvivergli. In generale una Catastrofe è un evento negativo che si scatena all'interno dell'area di gioco e che, in modo diverso l'una dall'altra, arrecano danno alla vita del giocatore. Alcune di esse sono collegate ad un bersaglio detto "Target" al di fuori dell'area di gioco e visibile solo al giocatore Oculus che, se sparato, farà scomparire la Catastrofe ad esso collegato. Altre, invece, non sono collegate a nessun bersaglio e hanno altri modi di essere superate o evitate. Tutte le catastrofi vengono spawnate dal CatastropheSpawner, il quale è un GameObject attivo sul Server che ad intervalli regolari scelti tramite il suo Inspector, esegue il codice dedicato allo spawning di una Catastrofe. Nell'esecuzione di tale codice viene generata una posizione casuale in un range circolare, evitando di generare la posizione attualmente occupata dal giocatore, per poi spawnare in tale posizione una Catastrofe casuale tra le possibili. Inoltre, in seguito allo spawn della Catastrofe, se tale Catastrofe necessita di un Target, esso viene spawnato in una posizione casuale in un'area circolare al di fuori del campo visivo del PC player.

```
void Update()
    {
        if (canSpawn == true && catastrophes.Count < maxCatastrophes)
        {
            canSpawn = false;
            SpawnCatastrophe(CatastropheRandPosition(out randIndex), TargetRandPosition());
            coroutine = StartCoroutine(delayCoroutine(betweenDelay));
        }
    }
```

Listato 3.5: CatastropheSpawner.cs (Riga 50-58)

Di seguito verranno descritte le singole Catastrofi e i motivi delle loro implementazioni.

3.4.2.1 FlameCatastorphe

La FlameCatastrophe consiste in un'area del campo di gioco che prende fuoco. Essa non ha una durata massima e termina solo quando il Target ad essa collegata viene colpito dai giocatori. Se il giocatore entra a contatto con essa subisce del danno nel tempo che, seppur poco, pone il giocatore in una situazione scomoda dalla quale deve trovare modo di uscire. Altro fattore bilanciante del suo danno minimo è la grande estensione della sua area di attività che costringe il giocatore a cercare percorsi alternativi per navigare l'area di gioco. Dal punto di vista implementativo, il suo tipo di danno è stato implementato tramite un contatore che si azzera ad ogni "tick_rate" (parametro del suo Inspector) secondi e ad ogni azzeramento, se il giocatore è ancora a contatto con la Catastrofe, arreca danno a quest'ultimo.

```
public void OnTriggerStay(Collider other)
    {
        if(!isServer)
        {
            return;
        }
        counter++;
        if (other.CompareTag("Player") && (counter >= (60 / tick_rate)))
        {
            counter = 0;
            playerController.takeDamage(gameObject.GetComponent<Catastrophe>().getDamage());
        }
    }
```

Listato 3.6: FlameCatastrophe.cs (Riga 28-40)

3.4.2.2 MineCatastrophe

La MineCatastrophe consiste in una mina di terra che esplode se entra in contatto con il giocatore. Anche in tal caso l'unico modo per eliminare tale Catastrophe consiste nel distruggere il Target ad essa associato. La particolarità di tale Catastrophe risiede nella visibilità che si ha di essa: la visibilità MineCatastrophe è inversamente proporzionale alla distanza del giocatore da essa; quindi se il giocatore è abbastanza distante non sarà in

grado di vederla mentre se il giocatore è abbastanza vicino sarà in grado di vederla completamente. Questa caratteristica costringe il giocatore PC a non accelerare troppo in una direzione senza aver preso coscienza della mancanza di eventuali mine sul percorso, il che si traduce in ulteriore stress sulle sue spalle e in un'ulteriore coordinazione da parte dell'Oculus player, il quale possiede una visuale più ravvicinata ed è maggiormente in grado di vedere le mine rispetto al PC player.

```
float distance = Vector2.Distance(new Vector2(transform.position.x, transform.position.z),
new Vector2(player.transform.position.x, player.transform.position.z));
float visibility;
float newIntensiy;
if (distance <= vision_range)
{
       //CALCULATES THE VISIBILITY OF THE MINE
       float range_percentage = distance / vision_range;
       // range_percentage :  1f at range distance, 0f at player position
       visibility = Mathf.Lerp(1f, 0f, range_percentage);
       newIntensiy = Mathf.Lerp(lightMaxIntensity, 0f, range_percentage);
}
```

Listato 3.7: MineCatastrophe.cs (Riga 55-65)

3.4.2.3 MeteorCatastorphe

La MeteorCatastrophe consiste in un meteorite che avanza ad un tempo prestabilito verso il suolo, dove un bersaglio lampeggiante indica il luogo di impatto. Al momento dell'impatto il meteorite esplode generando un'onda d'urto che arreca molti danni al giocatore se esso è in contatto con l'area dell'esplosione. Tale Catastrophe è una delle più pericolose perché, nonostante il suo arrivo venga indicato dal bersaglio lampeggiante, se il giocatore dovesse essere impossibilitato dal sostare su tale bersaglio per via di altre Catastrofi in gioco, sarebbe destinato ad un "Game Over" quasi certo, dato dalla grandissima quantità di danni provocata dalla MeteorCatastrophe. Dal punto di vista dell'implementazione, dall'Inspector della Catastrofe è possibile scegliere il tempo in secondi che deve essere impiegato da essa per raggiungere il suolo. In seguito, nello Script, tale tempo viene utilizzato per calcolare la velocità a cui deve viaggiare dal punto di partenza al suolo per raggiungere quest'ultimo nel tempo scelto. Finito questo intervallo di tempo la meteora viene fatta esplodere e viene generata l'area di danno nel punto di impatto.

```
void Start()
    {
        target_renderer = gameObject.transform.GetComponent<Renderer>();
        startingColor = target_renderer.material.GetColor("_Color");
        target_light.SetActive(false);
        gameObject.GetComponent<SphereCollider>().enabled = false;
        launchMeteor();
        playerController = GameObject.Find("PlayerController").GetComponent<PlayerController>();
    }


void launchMeteor()
    {
        float distance = Vector3.Distance(meteor.transform.position, gameObject.transform.position);
        float speed = distance / meteorCountdown;
        meteor.GetComponent<Rigidbody>().velocity = new Vector3(0f, -speed, 0f);
    }
```

Listato 3.8: MeteorCatastrophe.cs (Riga 34-42, Riga 87-92)

3.4.2.4 SeekerCatastorphe

La SeekerCatastrophe consiste in una singolarità di forma cubica che insegue il giocatore e, se vi entra in contatto, esplode arrecando ingenti danni alla vita di quest'ultimo. La particolarità di tale Catastrofe risiede nella sua invisibilità agli occhi del PC player. È solo l'Oculus player, infatti, l'unico in grado di vedere tale Catastrofe e ciò costringe quest'ultimo a coordinarsi con il PC player così da riuscire a muovere il Tank e posizionarlo in modo tale che l'Oculus player possa sparare un proiettile alla Catastrofe. Tale Catastrofe, infatti, non è collegata a nessun Target e, quindi, è eliminabile solo tramite i proiettili della torretta del Tank. Dal punto di vista implementativo la visibilità è gestita tramite uno Script denominato "SeekerActivator" che controlla il tipo di piattaforma del Client su cui viene spawnata la SeekerCatastrophe e, se tale sistema non è un sistema Android (e quindi Oculus), disattiva il componente che permette il rendering grafico del GameObject della Catastrofe.

```
public void Start()
{
	if(Application.platform != RuntimePlatform.Android)
	{
		this.deactivate();
	}
}

public void deactivate()
{
	seeker_block.GetComponent<Renderer>().enabled = false;
	heat.SetActive(false);
}
```

Listato 3.9: SeekerActivator.cs (Riga 11-22)

3.4.3 Target

Molte Catastrofi all'interno del progetto proposto sono collegate ad un bersaglio denominato "Target" consistente in una sfera fluttuante al di fuori del campo visivo del PC player ma visibili all'Oculus player. Tali bersagli hanno raffigurato al loro centro il simbolo della Catastrofe a cui sono associati e, se distrutti mediante i proiettili della torretta del Tank, scompaiono facendo terminare la Catastrofe collegata. Dal punto di vista implementativo lo spawn dei Target per le Catastrofi che li prevedono avviene nello Script del CatastropheSpawner. In particolare, dopo aver selezionato casualmente la Catastrofe da spawnare, il CatastropheSpawner controlla se essa ha bisogno di un Target; in caso affermativo, procede alla generazione di un punto casuale in un'area circolare definita in modo da non essere presente nell'area visibile al PC player. Ricavato il punto casuale, si procede allo spawning del Prefab del Target corretto in base alla Catastrofe e si assegna ad esso la Catastrofe istanziata sul Server; infine si procede anche con lo spawn di essa su tutti i Client. Nello Script del Target, invece, viene gestita la collisione dei proiettili della torretta con la superficie di quest'ultimo. Se in un dato momento essa avviene, lo Script del Target elimina la Catastrophe dal campo di gioco per poi autodistruggersi.

```
if (needTarget)
        {
            GameObject toSpawn = targetPrefab;
            //Instantiating the Target Prefab depending on the Catastrophe Type
            switch (catastrophe.GetComponent<Catastrophe>().type)
            {
                case Catastrophe.TYPE.Mine:
                    {
                        toSpawn = getTargetPrefab("MineTarget", targetPrefab);
                        break;
                    }
                case Catastrophe.TYPE.Flame:
                    {
                        toSpawn = getTargetPrefab("FireTarget", targetPrefab);
                        break;
                    }
                default:
                    {
                        break;
                    }
            }
            GameObject target = Instantiate(toSpawn, new Vector3(tar_pos.x,
outerRingObject.position.y, tar_pos.y), Quaternion.identity);
            TargetController tar_controller = target.GetComponent<TargetController>();
            tar_controller.catastrophe = catastrophe;
            tar_controller.catSpawner = gameObject.GetComponent<CatastropheSpawner>();
            switch (catastrophe.GetComponent<Catastrophe>().type)
            {
                case Catastrophe.TYPE.Mine:
                    {
                        MineCatastrophe mine_cat = catastrophe.GetComponent<MineCatastrophe>();
                        mine_cat.target = target;
                        break;
                    }
                default:
                    {
                        break;
                    }
            }
            NetworkServer.Spawn(target);
        }
```

Listato 3.10: CatastropheSpawner.cs (Riga 199-238)

```
private void OnCollisionEnter(Collision collision)
    {
        if(!isServer)
        {
            return;
        }

        if(collision.gameObject.CompareTag("Bullet"))
        {
            catSpawner.deleteCatastrophe(catastrophe.GetComponent<Catastrophe>().getId());
            NetworkServer.Destroy(gameObject);
        }
    }
```

Listato 3.11: TargetController.cs (Riga 32-44)

3.4.4 Punti

Lo scopo del gioco, come già detto nel capitolo introduttivo, è quello di resistere il più possibile ai pericoli insiti nelle Catastrofi attive nell'area di gioco e cercare di accumulare il maggior numero di punti. I punti si ottengono raccogliendo i PointPickup che appaiono per il campo di gioco sotto forma di monete dorate e vengono generati dal PickupSpawner. Esso, a differenza del CatastropheSpawner, con il quale condivide gran parte del codice, permette lo spawn dei PointPickup anche all'interno dell'area di una Catastrophe, obbligando cosi i giocatori a ragionare su quali punti conviene prendere subito e per quali, invece, conviene attuare una strategia che permetta loro di raccoglierli in totale sicurezza. Il punteggio è gestito dallo Script "PlayerController", il quale offre i suoi metodi allo script del PointPickup per aggiornare il punteggio del giocatore quando quest'ultimo viene raccolto. Tale punteggio, come anche la vita del giocatore, è una SyncVar, ovvero una variabile sincronizzata su tutti i Client il cui valore può essere però cambiato solo dal Server. In questo modo si evita la possibilità che un Client voglia imbrogliare modificando a piacere i propri punti o si renda immortale non permettendo il decremento della propria vita.

```
private void OnTriggerEnter(Collider other)
    {
        if(!isServer)
        {
            return;
        }
        if(other.CompareTag("Player"))
        {

              playerController.gainPoints(gameObject.GetComponent<Pickup>().getBenefitValue());
              pickupSpawner.deletePickup(gameObject.GetComponent<Pickup>().getId());
        }
    }
```

Listato 3.11: CoinPickUp.cs (Riga 22-33)

Conclusioni e sviluppi futuri

In questa sezione conclusiva verranno discussi i punti di forza rivelati dal progetto di tesi ed i possibili miglioramenti attuabili alle sue caratteristiche di base. Il progetto presentato ha come focus primario la valorizzazione delle esperienze multicanale; esse sono già diventate uno standard nell'industria videoludica degli ultimi anni ed il loro progresso non accenna a decelerare. In particolare, esperienze come quella proposta, di cross-platform VR/PC, sono la più recente aggiunta al macro-genere del multicanale e, per tale ragione, sono ancora parzialmente esplorate. Il futuro del VR si prospetta carico di interazioni con altre piattaforme, dato il suo facile adattamento sia sul fronte gameplay, sia su quello software grazie a mezzi di unificazione come quelli offerti dal motore Unity. A tal proposito è giusto citare le enormi possibilità di sviluppo e di miglioramento che si otterranno nei prossimi anni: Unity è il motore che più di tutti offre un ambiente con una componente multicanale intrinseca; la sua gestione di elementi VR e di Networking è, però, ancora in uno stato instabile e per tale ragione è aperta ancora a numerosi miglioramenti, alcuni dei quali già rilasciati nel corso dell'anno corrente [19][20]. A tali future migliorie si aggiungono i possibili cambiamenti attuabili sul design e sul funzionamento del progetto in se; nuove Catastrofi possono essere facilmente implementate ed introdotte all'interno del gioco dato il grande livello di parametrizzazione attuato sul CatastropheSpawner e sulla classe Catastrofe; altri tipi di Pickup possono essere aggiunti per scopi diversi, come, ad esempio un Pickup avente un effetto curativo, oppure un effetto tale da fermare lo spawn di Catastrofi per un tempo limitato. Il sistema di mira della torretta può essere dotato di una seconda dimensione aggiuntiva all'attuale asse orizzontale, per rendere più interessante e impegnativa l'esperienza del giocatore VR. Il sistema di Networking offerto da Mirror, può essere, se pur con un aumento di complessità di sviluppo, da un sistema proprietario costruito utilizzando la comunicazione tramite WebSocket, di sicuro più personalizzabile e performante. Il gameplay del giocatore PC può essere arricchito con delle azioni speciali, come uno scatto rapido in una direzione ricaricabile con particolari Pickup, che permette la schivata istantanea delle Catastrofi ma pone il Tank in uno stato di avanzata incontrollata che potrebbe portare, in alcuni casi, all'involontaria sconfitta data dall'impatto con i confini del campo di gioco. In generale, quando si tratta di giochi aventi uno stile di programmazione e sviluppo "sandbox" (con componenti di facile aggiunta e personalizzazione) come questo, l'unico limite verso il miglioramento risiede solo nelle capacità creative dello

sviluppatore. Per quanto riguarda l'hardware, invece, il miglioramento è in questo caso costantemente attivo: anno dopo anno, nuove versioni di hardware VR vengono rilasciate e, insieme ad esse, nuove feature e potenzialità sono offerte agli sviluppatori che utilizzano tali hardware per la creazione di nuove esperienze. Basti pensare che nel solo corso dell'anno corrente sono stati annunciati tre nuovi visori VR carichi di nuove funzionalità e di hardware aggiornati: il DecaGear [21], l'Oculus Quest 2 [22] (nuova versione del visore utilizzato per il progetto presentato) e il Playstation VR2 (relativo all'uscita della nuova console Sony PS5), di cui non si hanno ancora molte informazioni. In conclusione, il mondo dei videogiochi è in continua espansione verso orizzonti che neanche gli sviluppatori possono prevedere e, con esso, si espandono anche tutte le funzionalità e le possibilità delle esperienze realizzabili; in questo futuro, però, probabile costante resterà il cross-platform e, molto plausibilmente, il VR; per tali ragioni il progetto presentato dispone ferme basi sullo studio delle potenzialità date dall'unione di questi macro-generi verso lo sviluppo di esperienze di interesse sempre maggiore.

Codici citati

In questo capitolo sono presenti tutti i codici citati all'interno della tesi presentata, riportati per intero.

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4
5   public class TurretSync : MonoBehaviour
6   {
7
8       public GameObject turret;
9       public GameObject soldier_cam;
10      public GameObject tank;
11
12      float currentZeroOfTank;
13      float leftLimit;
14      float rightLimit;
15      float currentZeroOfCamera;
16
17      // Update is called once per frame
18      void Update()
19      {
20          if (turret && tank)
21          {
22              currentZeroOfTank = tank.transform.rotation.eulerAngles.y;
23              leftLimit = currentZeroOfTank - 45;
24              rightLimit = currentZeroOfTank + 45;
25              currentZeroOfCamera = soldier_cam.transform.rotation.eulerAngles.y;
26              if (isInBound())
27              {
28                  turret.transform.rotation = Quaternion.Euler(90, 0, -
29  soldier_cam.transform.rotation.eulerAngles.y - 90);
30              }
31          }
32      }
33
34      //Checks if the final position of the turret after the sync with the VR rig rotation
35  would exceed the boundaries of the 90° cone in front of the tank
36      private bool isInBound()
37      {
38          bool isinbound = true;
39          if (leftLimit < 0)
40              leftLimit = leftLimit + 360;
41          if (rightLimit >= 360)
42              rightLimit = rightLimit - 360;
43          if ((rightLimit < leftLimit) && (currentZeroOfCamera > rightLimit &&
44  currentZeroOfCamera < leftLimit))
45              isinbound = false;
46          if ((rightLimit > leftLimit) && !(currentZeroOfCamera < rightLimit &&
47  currentZeroOfCamera > leftLimit))
48              isinbound = false;
49          return isinbound;
50      }
51  }
```

Listato 2.1: TurretSync.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Mirror;
using UnityEngine.XR;
using UnityEngine.UI;
using System;

public class HandHeldController : NetworkBehaviour
{
    public Text logText;
    public GameObject tank;
    public GameObject bullet_spawn;

    List<InputDevice> devices;
    InputDevice device;
    TanksNetworkManager manager;
    bool cannon_loaded = true;

    private void Start()
    {
        manager = GameObject.Find("TanksNetworkManager").GetComponent<TanksNetworkManager>();
        bullet_spawn = GameObject.Find("BulletSpawn");
        devices = new List<InputDevice>();
        InputDevices.GetDevicesWithCharacteristics(InputDeviceCharacteristics.Right |
InputDeviceCharacteristics.HeldInHand | InputDeviceCharacteristics.Controller, devices);
        if (devices.Count > 0)
        {
            device = devices[0];
        }
    }

    void Update()
    {
        if (device != null)
        {
            bool triggerValue;
            if (device.TryGetFeatureValue(UnityEngine.XR.CommonUsages.triggerButton, out
triggerValue) && triggerValue)
            {
                Fire();
            }
        }
    }

    void Fire()
    {
        if (cannon_loaded && bullet_spawn)
        {
            cannon_loaded = false;
            CmdFire();
            StartCoroutine(ReloadCannonCoroutine());
        }
    }

    [Command]
    void CmdFire()
    {
        if (cannon_loaded && bullet_spawn)
        {
            GameObject bullet = Instantiate(manager.spawnPrefabs.Find(prefab => prefab.name
== "Bullet"), bullet_spawn.transform.position, bullet_spawn.transform.rotation);
            NetworkServer.Spawn(bullet);
            StartCoroutine(ReloadCannonCoroutine());
        }
    }

    IEnumerator ReloadCannonCoroutine()
    {
        yield return new WaitForSeconds(2);
        cannon_loaded = true;
    }
```

```
73      }
```

Listato 3.1: HandHeldController.cs

```
1       using System.Collections;
2       using System.Collections.Generic;
3       using UnityEngine;
4       using Mirror;
5       using System;
6       using UnityEngine.UI;
7       using UnityEngine.SceneManagement;
8       public class TanksNetworkManager : NetworkManager
9       {
10          GameObject soldier;
11          GameObject vr_rig;
12          TankControl controlScript;
13          TurretSync turretScript;
14          RigMovController rigMovScript;
15
16          public GameObject soldier_spawn;
17          public GameObject PCGameOverMenu;
18          public GameObject tank;
19          public Camera MainCamera;
20
21          public override void OnServerAddPlayer(NetworkConnection conn)
22          {
23              //ALLO STATO ATTUALE FUNZIONA SOLO SE IL PC PLAYER SI CONNETTE PRIMA DELL'OCULUS
24      PLAYER (DA FIXARE)
25              switch (numPlayers)
26              {
27                  case 0:
28                      {
29                          GameObject player = Instantiate(playerPrefab, Vector3.zero,
30      Quaternion.identity);
31                          controlScript = player.GetComponent<TankControl>();
32                          NetworkServer.AddPlayerForConnection(conn, player);
33                          tank = player;
34                          soldier_spawn = GameObject.Find("Soldier Spawn");
35                          TankCameraActivator cam_activator =
36      tank.GetComponent<TankCameraActivator>();
37                          cam_activator.my_cam = MainCamera;
38                          if (MainCamera != null)
39                          {
40                              cam_activator.my_audio = MainCamera.GetComponent<AudioListener>();
41                          }
42                          GameManager gameManager =
43      GameObject.Find("GameManager").GetComponent<GameManager>();
44                          gameManager.PCGameOverMenu = PCGameOverMenu;
45                          gameManager.deactivatePCGameOverMenu();
46                          break;
47                      }
48                  case 1:
49                      {
50                          if (tank != null)
51                          {
52                              soldier = Instantiate(spawnPrefabs.Find(prefab => prefab.name ==
53      "Soldier"), soldier_spawn.transform.position, tank.transform.rotation);
54                              NetworkServer.Spawn(soldier);
55                              GameObject rig_spawn = GameObject.Find("VR_Spawn");
56                              vr_rig = Instantiate(spawnPrefabs.Find(prefab => prefab.name ==
57      "NetworkedRig"), rig_spawn.transform.position, tank.transform.rotation);
58                              turretScript = vr_rig.GetComponent<TurretSync>();
59                              turretScript.turret = GameObject.Find("Tower_Bone").gameObject;
```

```
                        turretScript.tank = tank;
                        rigMovScript = vr_rig.GetComponent<RigMovController>();
                        rigMovScript.vr_spawn = rig_spawn;
                        vr_rig.GetComponent<HandHeldController>().tank = tank;
                        GameObject catastropheSpawner =
GameObject.Find("CatastropheSpawner");
                        catastropheSpawner.GetComponent<CatastropheSpawner>().tank = tank;

catastropheSpawner.GetComponent<CatastropheSpawner>().startSpawning();
                        GameObject pickupSpawner = GameObject.Find("PickupSpawner");
                        pickupSpawner.GetComponent<PickupSpawner>().tank = tank;
                        pickupSpawner.GetComponent<PickupSpawner>().startSpawning();
                    }
                    else
                    {
                        soldier = Instantiate(spawnPrefabs.Find(prefab => prefab.name ==
"Soldier"), new Vector3(0, 1.4f, 0), Quaternion.identity);
                        NetworkServer.Spawn(soldier);
                        GameObject rig_spawn = GameObject.Find("VR_Spawn");
                        vr_rig = Instantiate(spawnPrefabs.Find(prefab => prefab.name ==
"NetworkedRig"), rig_spawn.transform.position, Quaternion.identity);
                    }
                    GameManager gameManager =
GameObject.Find("GameManager").GetComponent<GameManager>();
                    gameManager.VRGameOverMenu = GameObject.FindWithTag("VRResults");
                    controlScript.soldier = soldier; //aggiunge il soldier al TankControl
script per sincronizzare la position e la rotation dell'XRSoldier con i movimenti del Tank.
                    NetworkServer.AddPlayerForConnection(conn, vr_rig);
                    break;

                }
            }
        }

        public override void OnClientDisconnect(NetworkConnection conn)
        {
            base.OnClientDisconnect(conn);
            SceneManager.LoadScene(0);
        }

        public override void OnStopHost()
        {
            base.OnStopHost();
            SceneManager.LoadScene(0);
        }
}
```

Listato 3.2 – 3.3: TanksNetworkManager.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Mirror;
public class MeteorCatastrophe : NetworkBehaviour
{

    [Tooltip("The number of seconds of delay between two flashes of the countdown light")]
    [Range(0.1f, 5f)]
    public float flashDelay = 1;

    [Tooltip("The number of seconds from the start of the countdown to the explosion of the
Meteor")]
    [Range(1, 5)]
    public int meteorCountdown;

```

```
17              public GameObject meteor;
18
19              public GameObject explosion;
20
21              public GameObject target_light;
22
23              public CatastropheSpawner catSpawner;
24
25              Coroutine flash_coroutine;
26              Coroutine destroy_coroutine;
27              PlayerController playerController;
28              Renderer target_renderer;
29              bool delayElapsed = true;
30              Color startingColor;
31              float fallingSpeed;
32              float count = 0f;
33
34              // Start is called before the first frame update
35              void Start()
36              {
37                  target_renderer = gameObject.transform.GetComponent<Renderer>();
38                  startingColor = target_renderer.material.GetColor("_Color");
39                  target_light.SetActive(false);
40                  gameObject.GetComponent<SphereCollider>().enabled = false;
41                  launchMeteor();
42                  playerController =
43          GameObject.Find("PlayerController").GetComponent<PlayerController>();
44              }
45
46              // Update is called once per frame
47              void Update()
48              {
49                  if (count >= meteorCountdown)
50                  {
51                      Destroy(meteor);
52                      Destroy(target_light);
53                      target_renderer.enabled = false;
54                      explosion.SetActive(true);
55                      gameObject.GetComponent<SphereCollider>().enabled = true;
56                      StopCoroutine(flash_coroutine);
57                      if (isServer)
58                      {
59                          destroy_coroutine = StartCoroutine(TankoopUtils.destroyCoroutine(gameObject,
60          2, catSpawner));
61                      }
62                  }
63                  else
64                  {
65                      count += Time.deltaTime;
66                  }
67                  if (delayElapsed)
68                  {
69                      delayElapsed = false;
70                      flash_coroutine = StartCoroutine(flashCoroutine(flashDelay));
71                  }
72              }
73
74              IEnumerator flashCoroutine(float seconds)
75              {
76                  yield return new WaitForSeconds(seconds);
77                  if (target_renderer.material.color.g == 0f)
78                  {
79                      target_light.SetActive(false);
80                      target_renderer.material.color = startingColor;
81                  }
82                  else
83                  {
84                      target_light.SetActive(true);
85                      target_renderer.material.color = new Color(100f, 0f, 0f, 255f);
86                  }
87                  delayElapsed = true;
88              }
89
90              void launchMeteor()
91              {
```

```
        float distance = Vector3.Distance(meteor.transform.position,
gameObject.transform.position);
        float speed = distance / meteorCountdown;
        meteor.GetComponent<Rigidbody>().velocity = new Vector3(0f, -speed, 0f);
    }

    public void OnTriggerEnter(Collider other)
    {
        if (other.CompareTag("Player")){
            if (isServer)
            {

playerController.takeDamage(gameObject.GetComponent<Catastrophe>().getDamage());
            }
        }
    }
}
```

Listato 3.4: MeteorCatastrophe.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Mirror;

public class FlameCatastrophe : NetworkBehaviour
{

    [Tooltip("The rate at which the Player's health will deplate while remainig
inside the flames (Times per Second)")]
    [Range(1, 30)]
    public int tick_rate = 10;

    private int counter = 0;
    PlayerController playerController;

    // Start is called before the first frame update
    void Start()
    {
        playerController =
GameObject.Find("PlayerController").GetComponent<PlayerController>();
    }

    // Update is called once per frame
    void Update()
    {
        gameObject.transform.GetComponent<Renderer>().material.mainTextureOffset +=
new Vector2(0.05f * Time.deltaTime, 0.10f * Time.deltaTime);
    }

    public void OnTriggerStay(Collider other)
    {
        if(!isServer)
        {
            return;
        }
        counter++;
        if (other.CompareTag("Player") && (counter >= (60 / tick_rate)))
        {
            counter = 0;
```

```
41
42          playerController.takeDamage(gameObject.GetComponent<Catastrophe>().getDamage());
43              }
44          }
45
46          public void OnTriggerExit(Collider other)
47          {
48              if (!isServer)
49              {
50                  return;
51              }
52              if (other.CompareTag("Player"))
53              {
54                  counter = 0;
55              }
56          }
57      }
```

Listato 3.6: FlameCatastrophe.cs

```
1       using System.Collections;
2       using System.Collections.Generic;
3       using UnityEngine;
4       using Mirror;
5
6       public class MineCatastrophe : NetworkBehaviour
7       {
8
9           [Tooltip("The range at which the mine can be seen")]
10          [Range(5f, 200f)]
11          public float vision_range = 50f;
12
13          public GameObject body;
14          public GameObject lightBulb;
15          public Light flashLight;
16          public GameObject explosion;
17          public GameObject target;
18
19          CatastropheSpawner spawner;
20          PlayerController playerController;
21
22          float lightMaxIntensity;
23          Renderer body_renderer;
24          Renderer bulb_renderer;
25          GameObject player;
26
27          // Start is called before the first frame update
28          void Start()
29          {
30              if (body)
31              {
32                  body_renderer = body.transform.GetComponent<Renderer>();
33              }
34              if (lightBulb)
35              {
36                  bulb_renderer = lightBulb.transform.GetComponent<Renderer>();
37              }
38              if (flashLight)
39              {
40                  lightMaxIntensity = flashLight.intensity;
41              }
42              player = GameObject.FindWithTag("Player");
43              if (isServer)
44              {
```

```
            spawner =
GameObject.FindWithTag("CatSpawner").GetComponent<CatastropheSpawner>();
        }
        playerController =
GameObject.Find("PlayerController").GetComponent<PlayerController>();
    }

    // Update is called once per frame
    void Update()
    {
        if (player)
        {
            float distance = Vector2.Distance(new Vector2(transform.position.x,
transform.position.z), new Vector2(player.transform.position.x,
player.transform.position.z));
            float visibility;
            float newIntensiy;
            if (distance <= vision_range)
            {
                //CALCULATES THE VISIBILITY OF THE MINE
                float range_percentage = distance / vision_range;
                // range_percentage :  1f at range distance, 0f at player position
                visibility = Mathf.Lerp(1f, 0f, range_percentage);
                newIntensiy = Mathf.Lerp(lightMaxIntensity, 0f, range_percentage);
            }
            else
            {
                visibility = 0f;
                newIntensiy = 0f;
            }
            if (body_renderer)
            {
                body_renderer.material.color = new
Color(body_renderer.material.color.r, body_renderer.material.color.g,
body_renderer.material.color.b, visibility);
            }
            if (bulb_renderer)
            {
                bulb_renderer.material.color = new
Color(bulb_renderer.material.color.r, bulb_renderer.material.color.g,
bulb_renderer.material.color.b, visibility);
            }
            if (flashLight)
            {
                flashLight.intensity = newIntensiy;
            }
        }
    }

    private void OnTriggerEnter(Collider other)
    {
        if (other.CompareTag("Player"))
        {
            explosion.SetActive(true);
            lightBulb.SetActive(false);
            body.GetComponent<Renderer>().enabled = false;
            gameObject.GetComponent<MineCatastrophe>().enabled = false;
            gameObject.GetComponent<BoxCollider>().enabled = false;
            if (isServer)
            {

playerController.takeDamage(gameObject.GetComponent<Catastrophe>().getDamage());
                NetworkServer.Destroy(target);
                StartCoroutine(TankoopUtils.destroyCoroutine(gameObject, 2, spawner));
            }
        }
    }
}
```

Listato 3.7: MineCatastrophe.cs

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

public class SeekerActivator : MonoBehaviour
{

    public GameObject seeker_block;
    public GameObject heat;

    public void Start()
    {
        if(Application.platform != RuntimePlatform.Android)
        {
            this.deactivate();
        }
    }
    public void deactivate()
    {
        seeker_block.GetComponent<Renderer>().enabled = false;
        heat.SetActive(false);
    }

    public void activate()
    {
        seeker_block.GetComponent<Renderer>().enabled = true;
        heat.SetActive(true);
    }


}
```

Listato 3.9: SeekerActivator.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Mirror;

public class CatastropheSpawner : NetworkBehaviour
{
    [Tooltip("List of Catastrophe Prefabs that can be spawned by the CatastropheSpawner")]
    public List<GameObject> catastrophePrefabs = new List<GameObject>();

    [Tooltip("List of materials that can be set onto the targets depending on which
Catastrophe the targets are connected to")]
    public List<GameObject> targetPrefabs = new List<GameObject>();

    [Tooltip("Seconds after which Catastrophes will begin to spawn")]
    [Range(0, 60)]
    public int initialDelay = 0;

    [Tooltip("Number of seconds between the spawning of two Catastrophes")]
    [Range(1, 5)]
    public int betweenDelay = 1;

    [Tooltip("Maximum amount of Catastrophes active at once")]
```

```
24          [Range(1, 20)]
25          public int maxCatastrophes = 5;
26
27          [Tooltip("The radius of the circular area in which the CatastropheSpawner is active")]
28          public float areaRadius = 20;
29
30          [Tooltip("The objects that indicates the height and the radius of the outer circle where
31      targets are spawned")]
32          public Transform outerRingObject;
33
34          [Tooltip("The target prefab that is going to be spawned with the Catastrophes")]
35          public GameObject targetPrefab;
36
37          public GameObject tank;
38
39          bool canSpawn = false;
40          Coroutine coroutine;
41          int randIndex;
42          int incrementalId;
43
44          //List containing references to the spawned catastrophes
45          List<GameObject> catastrophes = new List<GameObject>();
46
47          private void Start()
48          {
49              incrementalId = 0;
50          }
51
52          void Update()
53          {
54              if (canSpawn == true && catastrophes.Count < maxCatastrophes)
55              {
56                  canSpawn = false;
57                  SpawnCatastrophe(CatastropheRandPosition(out randIndex), TargetRandPosition());
58                  coroutine = StartCoroutine(delayCoroutine(betweenDelay));
59              }
60          }
61
62
63          IEnumerator delayCoroutine(int seconds)
64          {
65              yield return new WaitForSeconds(seconds);
66              canSpawn = true;
67          }
68
69          Vector2 CatastropheRandPosition(out int random_index)
70          {
71              //Creating random point in spawn area circle for the Catastrophe Instantiate
72              random_index = Random.Range(0, catastrophePrefabs.Count);
73              GameObject catastrophe = catastrophePrefabs[random_index];
74              float cat_angle, cat_r, cat_x = 0, cat_z = 0;
75              switch (catastrophe.GetComponent<Catastrophe>().type)
76              {
77                  case Catastrophe.TYPE.Seeker:
78                      {
79                          cat_angle = Random.value * 2 * Mathf.PI;
80                          cat_r = areaRadius;
81                          cat_x = cat_r * Mathf.Cos(cat_angle);
82                          cat_z = cat_r * Mathf.Sin(cat_angle);
83                          break;
84                      }
85                  case Catastrophe.TYPE.Mine:
86                      {
87                          if (tank != null)
88                          {
89                              float x_size = tank.GetComponent<BoxCollider>().size.x + 5f;
90                              float z_size = tank.GetComponent<BoxCollider>().size.z + 5f;
91                              bool acceptable = false;
92                              while (!acceptable)
93                              {
94                                  cat_angle = Random.value * 2 * Mathf.PI;
95                                  cat_r = areaRadius * Mathf.Sqrt(Random.value);
96                                  cat_x = cat_r * Mathf.Cos(cat_angle);
97                                  cat_z = cat_r * Mathf.Sin(cat_angle);
```

```
                            //CHECKS IF THE CHOSEN SPAWN POSITION DOES NOT INTERSECT WITH THE
BOX COLLIDER OF THE TANK
                            if (!(((cat_x >= tank.transform.position.x - (x_size / 2)) &&
(cat_x <= tank.transform.position.x + (x_size / 2))) && ((cat_z >= tank.transform.position.z
- (z_size / 2)) && (cat_z <= tank.transform.position.z + (z_size / 2)))))
                            {
                                acceptable = true;
                            }
                        }
                    }
                    else
                    {
                        cat_angle = Random.value * 2 * Mathf.PI;
                        cat_r = areaRadius * Mathf.Sqrt(Random.value);
                        cat_x = cat_r * Mathf.Cos(cat_angle);
                        cat_z = cat_r * Mathf.Sin(cat_angle);
                    }
                    break;
                }
            default:
                {
                    cat_angle = Random.value * 2 * Mathf.PI;
                    cat_r = areaRadius * Mathf.Sqrt(Random.value);
                    cat_x = cat_r * Mathf.Cos(cat_angle);
                    cat_z = cat_r * Mathf.Sin(cat_angle);
                    break;
                }
        }
        return new Vector2(cat_x, cat_z);
    }

    Vector2 TargetRandPosition()
    {
        //Creating random point in spawn outer circle for the Target Instantiate
        float tar_angle = Random.value * 2 * Mathf.PI;
        float tar_r = Vector2.Distance(new Vector2(outerRingObject.position.x,
outerRingObject.position.z), new Vector2(gameObject.transform.position.x,
gameObject.transform.position.z));
        float tar_x = tar_r * Mathf.Cos(tar_angle);
        float tar_z = tar_r * Mathf.Sin(tar_angle);
        return new Vector2(tar_x, tar_z);
    }

    GameObject getTargetPrefab(string name, GameObject defaultTarget)
    {
        GameObject toReturn = defaultTarget; ;
        targetPrefabs.ForEach((target) => {
            if(target.name.Equals(name))
            {
                toReturn = target;
            }
        });
        return toReturn;
    }


    void SpawnCatastrophe(Vector2 cat_pos, Vector2 tar_pos)
    {
        bool needTarget = true; //Indicates whether the spawned Catastrophe has to be tied to
a Target or not.
        float y_pos;
        float damage = 0f;
        Quaternion rotation;
        switch (catastrophePrefabs[randIndex].GetComponent<Catastrophe>().type)
        {
            case Catastrophe.TYPE.Seeker:
                {
                    needTarget = false;
                    y_pos = 0.1f;
                    damage = 10f;
                    rotation = Quaternion.identity;
                    break;
                }
            case Catastrophe.TYPE.Meteor:
```

```
                    {
                        needTarget = false;
                        y_pos = 0.2f;
                        rotation = Quaternion.identity;
                        damage = 30f;
                        break;
                    }
                case Catastrophe.TYPE.Flame:
                    {
                        needTarget = true;
                        y_pos = 0.1f;
                        rotation = Quaternion.identity;
                        damage = 0.5f;
                        break;
                    }
                case Catastrophe.TYPE.Mine:
                    {
                        needTarget = true;
                        y_pos = 0f;
                        rotation = Quaternion.Euler(0f, 0f, 0f);
                        damage = 10f;
                        break;
                    }
                default:
                    {
                        y_pos = 0.1f;
                        rotation = Quaternion.identity;
                        break;
                    }
            }
            //Instantiating the Catastrophe
            GameObject catastrophe = Instantiate(catastrophePrefabs[randIndex], new
Vector3(cat_pos.x, y_pos, cat_pos.y), rotation);
            catastrophe.GetComponent<Catastrophe>().setId(incrementalId);
            catastrophe.GetComponent<Catastrophe>().setDamage(damage);
            incrementalId++;
            catastrophes.Add(catastrophe);
            if (needTarget)
            {
                GameObject toSpawn = targetPrefab;
                //Instantiating the Target Prefab depending on the Catastrophe Type
                switch (catastrophe.GetComponent<Catastrophe>().type)
                {
                    case Catastrophe.TYPE.Mine:
                        {
                            toSpawn = getTargetPrefab("MineTarget", targetPrefab);
                            break;
                        }
                    case Catastrophe.TYPE.Flame:
                        {
                            toSpawn = getTargetPrefab("FireTarget", targetPrefab);
                            break;
                        }
                    default:
                        {
                            break;
                        }
                }
                GameObject target = Instantiate(toSpawn, new Vector3(tar_pos.x,
outerRingObject.position.y, tar_pos.y), Quaternion.identity);
                TargetController tar_controller = target.GetComponent<TargetController>();
                tar_controller.catastrophe = catastrophe;
                tar_controller.catSpawner = gameObject.GetComponent<CatastropheSpawner>();
                switch (catastrophe.GetComponent<Catastrophe>().type)
                {
                    case Catastrophe.TYPE.Mine:
                        {
                            MineCatastrophe mine_cat =
catastrophe.GetComponent<MineCatastrophe>();
                            mine_cat.target = target;
                            break;
                        }
                    default:
                        {
                            break;
```

```
                        }
                    }
                    NetworkServer.Spawn(target);
                }
                else
                {
                    switch (catastrophe.GetComponent<Catastrophe>().type)
                    {
                        case Catastrophe.TYPE.Meteor:
                        {
                            catastrophe.GetComponent<MeteorCatastrophe>().catSpawner =
gameObject.GetComponent<CatastropheSpawner>();
                            break;
                        }
                        default:
                        {
                            break;
                        }
                    }
                }
                NetworkServer.Spawn(catastrophe);
            }

            public void startSpawning()
            {
                coroutine = StartCoroutine(delayCoroutine(initialDelay));
            }

            public void stopSpawning()
            {
                if (coroutine != null)
                {
                    StopCoroutine(coroutine);
                }
                canSpawn = false;
            }


            public void deleteCatastrophe(int id)
            {
                GameObject toRemove = null;
                catastrophes.ForEach((cat) =>
                {
                    //Debug.Log("cat id : " + cat.GetComponent<Catastrophe>().getId() +
"\ntoDelete.Id : " + toDelete.getId());
                    if (cat.GetComponent<Catastrophe>().getId() == id)
                    {
                        toRemove = cat;
                    }
                });
                if (toRemove != null)
                {
                    catastrophes.Remove(toRemove);
                    NetworkServer.Destroy(toRemove);
                }
            }
        }
```

Listato 3.9: CatastropheSpwner.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Mirror;

public class TargetController : NetworkBehaviour
```

```
{
    [Tooltip("The object towards which this target will be faced (Usually the
player)")]
    public GameObject target;

    [Tooltip("The catastrophe associated with this Target")]
    public GameObject catastrophe;

    public CatastropheSpawner catSpawner;

    void Start()
    {
        if (target == null)
        {
            target = GameObject.FindWithTag("Player");
        }
    }

    void Update()
    {
        if(target != null)
        {
            gameObject.transform.LookAt(target.transform);
        }
    }

    private void OnCollisionEnter(Collision collision)
    {
        if(!isServer)
        {
            return;
        }

        if(collision.gameObject.CompareTag("Bullet"))
        {

catSpawner.deleteCatastrophe(catastrophe.GetComponent<Catastrophe>().getId());
            NetworkServer.Destroy(gameObject);
        }
    }
}
```

Listato 3.9 – 3.11: TargetController.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Mirror;

public class CoinPickUp : NetworkBehaviour
{

    PlayerController playerController;
    PickupSpawner pickupSpawner;

    // Start is called before the first frame update
    void Start()
    {
```

```
15              playerController =
16          GameObject.Find("PlayerController").GetComponent<PlayerController>();
17              if (isServer)
18              {
19                  pickupSpawner =
20          GameObject.Find("PickupSpawner").GetComponent<PickupSpawner>();
21              }
22          }
23
24          private void OnTriggerEnter(Collider other)
25          {
26              if(!isServer)
27              {
28                  return;
29              }
30              if(other.CompareTag("Player"))
31              {
32
33          playerController.gainPoints(gameObject.GetComponent<Pickup>().getBenefitValue(
34          ));
35
36          pickupSpawner.deletePickup(gameObject.GetComponent<Pickup>().getId());
37              }
38          }
39      }
```

Bibliografia

[1] Andrew K. Przybylski e Netta Weinstein «Violent video game engagement is not associated with adolescents' aggressive behavior: evidence from a registered report». In: Oxford Internet Institute, University of Oxford (Feb. 2019). url: https://royalsocietypublishing.org/doi/10.1098/rsos.171474.

[2] Gregory D. Clemenson e Craig E. L. Stark «Virtual Environmental Enrichment through Video Games Improves Hippocampal-Associated Memory». In: University of California (Dic. 2015). url: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4682779/.

[3] Adam C. Oei e Michael D. Patterson «Enchancing Congnition with Video Games: A Multiple Game Training Study». In: Nanyang Technological University (mar. 2013). url: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3596277/

[4] Wikipedia contributors. Multiplayer video game – Wikipedia, The Free Encyclopedia. url: https://en.wikipedia.org/wiki/Multiplayer_video_game (Online, visitato in Novembre 2020)

[5] Gabriel Gambetta «Fast-Paced Multiplayer (Part I): Client-Server Game Architecture» (Online, visitato in Novembre 2020). url: https://www.gabrielgambetta.com/client-server-game-architecture.html

[6] Wikipedia contributors. Unity (game engine) – Wikipedia, The Free Encyclopedia. url: https://en.wikipedia.org/wiki/Unity_(game_engine) (Online, visitato in Novembre 2020).

[7] Unity User Manual. url: https://docs.unity3d.com/Manual/index.html. (Online, visitato in Novembre 2020).

[8] Unity Learn. url: https://learn.unity.com . (Online, visitato in Novembre 2020).

[9] Wikipedia contributors. Motore grafico – Wikipedia, The Free Encyclopedia. url: https://it.wikipedia.org/wiki/Motore_grafico (Online, visitato in Novembre 2020).

[10] Wikipedia contributors. Realtà Virtuale – Wikipedia, The Free Encyclopedia. url: https://it.wikipedia.org/wiki/Realt%C3%A0_virtuale (Online, visitato in Novembre 2020).

[11] Giuseppe Melacca e Sara Invitto «La Realtà Virtuale. Strumento per elicitare processi neurocognitivi per il trattamento in ambito riabilitativo». In: Università del Salento (2016). url: http://siba-ese.unisalento.it/index.php/psychofenia/article/download/16142/13933

[12] Wikipedia contributors. Host – Wikipedia, The Free Encyclopedia. url: https://it.wikipedia.org/wiki/Host (Online, visitato in Novembre 2020).

[13] Unity User Manual. url: https://docs.unity3d.com/Manual/GameObjects.html (Online, visitato in Novembre 2020).

[14] Unity User Manual. url: https://docs.unity3d.com/Manual/Components.html (Online, visitato in Novembre 2020).

[15] Unity User Manual. url: https://docs.unity3d.com/Manual/Prefabs.html (Online, visitato in Novembre 2020).

[16] Unity User Manual. url: https://docs.unity3d.com/Manual/CreatingScenes.html (Online, visitato in Novembre 2020).

[17] Unity Mirror Manual – Index url: https://mirror-networking.com/docs/ (Online, visitato in Novembre 2020).

[18] Unity Mirror Manual – Authority url: https://mirror-networking.com/docs/Articles/Guides/Authority.html (Online, visitato in Novembre 2020)

[19] Unity Asset Store – Mirror Releases url: https://assetstore.unity.com/packages/tools/network/mirror-129321#releases (Online, visitato in Novembre 2020)

[20] Unity Asset Store – XR Interaction Framework Releases url: https://assetstore.unity.com/packages/templates/systems/vr-interaction-framework-161066#releases (Online, visitato in Novembre 2020)

[21] DecaGear – Headset url : https://www.deca.net/decagear/#headset (Online, visitato in Novembre 2020)

[22] Oculus – Quest 2 url: https://www.oculus.com/quest-2/ (Online, visitato in Novembre 2020)