



UNIVERSITÀ DEGLI STUDI ROMA TRE

Dipartimento di Ingegneria
Corso di Laurea in Ingegneria Informatica

Tesi Di Laurea

Analisi, progettazione e implementazione
di Metarace: un gioco 3D nel Metaverso

Laureando
Emanuele Pietropaolo
Matricola 513489

Relatore
Prof. Franco Milicchio

Anno Accademico 2021/2022

Ringraziamenti

Grazie a tutti

Introduzione

Questa tesi tratta l'analisi, la progettazione e l'implementazione di Metarace, un'applicazione in Unreal Engine 5 sviluppata per il Metaverso. Poiché il Metaverso necessita ancora di nuove tecnologie e nuovistrumenti per essere realizzato, verrà discusso cosa si intende quando si parla di Metaverso oggi e cosa significa sviluppare in quest'ottica. Metarace si avvicina a questo concetto costruendo un mondo interattivo al quale gli utenti possono accedere attraverso un avatar digitale. Metarace, infatti, è un gioco in cui gli utenti hanno la possibilità di possedere dei cavalli, di farli partecipare a gare di velocità e di assistervi attraverso un visore per la realtà virtuale. L'esito della gara verrà stabilito da un server sulla base di caratteristiche intrinseche del cavallo, come la velocità, la resistenza e la potenza insieme ad una componente casuale che garantisce la non prevedibilità del risultato. Questo lavoro tratta la realizzazione dell'intera architettura di Metarace, tra cui la logica implementativa di base, lo strato del network, l'interfaccia utente, la creazione del mondo virtuale e degli oggetti 3D. Il mondo di gioco è stato realizzato con il motore grafico Unreal Engine 5 utilizzato in combinazione con la suite 3D Blender. Il software è stato progettato seguendo l'approccio della programmazione ad eventi e sono state messe in atto tecniche per sostenere un basso livello di accoppiamento e un alto livello di coesione nel codice. La comunicazione tra gli utenti e il server è stata implementata attraverso un'architettura client-server, utilizzando il protocollo WebSocket per poter tenere attive delle connessioni di tipo full-duplex bidirezionali con più giocatori. Durante la trattazione verrà posta particolare attenzione a tutti gli strumenti utilizzati durante l'implementazione ed alla traduzione del modello di dominio in codice. Metarace si andrà ad inserire in un contesto più ampio di applicazioni con cui comporrà l'universo virtuale dell'azienda Ringmaster.

Indice

Introduzione	iii
Indice	iv
Elenco delle figure	vi
Elenco degli algoritmi	viii
1 Background tecnologico	1
1.1 Il Metaverso	1
1.1.1 I visori per la realtà aumentata e virtuale	2
1.1.2 Storia del metaverso	6
1.1.3 Stato dell'arte	8
1.2 Il motore grafico Unreal Engine 5	11
1.3 Il software di modellazione Blender	16
2 Il core di Metarace	19
2.1 Metarace	19
2.2 La meccanica di base	20
2.2.1 L'algoritmo per il movimento dei cavalli	20
2.3 Animazione dei cavalli	27
2.3.1 Animazione del galoppo creata con Blender	29
2.4 Progettazione del mondo 3D	31
2.4.1 I modelli 3D	32
2.4.2 Strumenti forniti da Unreal Engine 5	35

3 Architettura del software	38
3.1 L'architettura di Metarace	38
3.1.1 Progettazione guidata dalle responsabilità	39
3.1.2 L' <i>Application Manager</i>	41
3.1.3 Lo <i>SceneController</i>	44
3.1.4 Il <i>WidgetController</i>	46
3.1.5 Il CameraController	51
3.1.6 Il <i>VRController</i>	53
4 Networking	55
4.1 Architettura Client - Server e WebSocket	55
4.1.1 Implementazione MetaraceNetworkActor	60
4.2 Programmazione ad Eventi (EDP)	62
Conclusioni e sviluppi futuri	72
Bibliografia	74

Elenco delle figure

1.1	Stereoscopio statunitense regolabile.	3
1.2	Il sistema di realtà virtuale progettato da Ivan Sutherland e soprannominato La Spada Di Damocle.	3
1.3	Il Nintendo Virtual Boy	4
1.4	La serie 1000 dei visori Virtuality	5
1.5	Il prototipo per il visore Sega VR	5
1.6	Una tipica scena in Habitat.	7
1.7	Set di The Mandalorian con scenografia renderizzata con Unreal Engine	10
1.8	Screenshot della prima versione di UnrealEd.	12
1.9	Screenshot della versione 5 dell'editor di Unreal Engine.	13
1.10	Interfaccia di default di Unreal Engine 5	14
1.11	Esempio di classe Blueprint per l'apertura e la chiusura di una porta in gioco	16
1.12	Una Mesh poligonale in Blender	18
2.1	CurveFloat generata con tutti i nodi interpolati (a) CurveFloat generata con tutti i nodi interpolati eccetto l'ultimo. (b)	25
2.2	<i>UCurveFloat</i> generata dall'algoritmo per il movimento dei cavalli	26
2.3	<i>AnimGraph</i> che gestisce l'animazione del cavallo.	27
2.4	<i>BlandSpace2D</i> per l'interpolazione delle animazioni sulla base della velocità.	28
2.5	Vista in Blender della Mesh del cavallo, con il suo rig associato, in una pose tra le 12 dell'animazione del galoppo	29
2.6	12 frame disegnati di un cavallo al galoppo	30

2.7	Schermata in cui è possibile vedere la Pose Mode di Blender, l'Action Editor con le keyframe, la mesh del cavallo in una pose con il suo scheletro e l'immagine utilizzata come reference.	31
2.8	Sequenza di 16 scatti con cui Muybridge dimostrò che i cavalli non toccano terra durante il galoppo.	32
2.9	Terreno di gioco in Blender (a) Altri modelli in Blender. (b)	33
2.10	Esempio di associazione di UV islands a singoli colori di una sola texture . .	34
2.11	Schema di funzionamento dell'unwrapping	35
2.12	Utilizzo del Foliage Tools in Unreal Engine 5	36
2.13	Schermata che riprende la vista dagli spalti del percorso	36
3.1	Modello di Dominio parziale per Metarace	40
3.2	Vista dall'inizio del percorso, dopo che lo SceneController ha fatto comparire i cavalli	44
3.3	Widget renderizzato in 3D per la modalità VR	49
3.4	Foto con visore VR durante una partita di Metarace	50
3.5	CineCameraActor che riprendere la griglia di partenza	51
3.6	Visuale della gara in VR da una posizione appositamente creata	53
4.1	Diagramma di sequenza per gli eventi GetAvailableRacesEvent e OnAvailableRacesFormatsEvent	63
4.2	Diagramma di sequenza per gli eventi JoinEvent e PlayerJoinedEvent . .	64
4.3	Diagramma di sequenza per gli eventi StartingGridEvent e AnotherPlayerJoinedEvent	65
4.4	Diagramma di sequenza per gli eventi CountdownEvent e RaceEvent . . .	68
4.5	Diagramma di sequenza per gli eventi RaceFinishedEvent e LeaderboardEvent	71

Elenco degli algoritmi

1.1	File header generato alla creazione di un Actor	12
1.2	Specificatore UPROPERTY per esporre una variabile all'editor	13
2.1	Sezione del file header (movimento cavallo)	21
2.2	Inizializzazione della Timeline nel file source (movimento cavallo)	21
2.3	Funzione che sfrutta il valore restituito dalla timeline nel tempo (movimento cavallo)	22
2.4	Funzione Tick (movimento cavallo)	22
2.5	Definizione variabili temporanee (movimento cavallo)	23
3.1	Sezione dell'header file della classe ApplicationManager dove vengono referenziate altri Blueprint basati su classi C++	41
3.2	Sezione del file source dell'Application Manager dove vengono creati gli oggetti SceneController e NetworkActor	41
3.3	Sezione del file source dell'Application Manager dove viene formato il Bind dei delegates	42
3.4	Sezione del source della classe SceneController dove viene fatto il controllo per sapere se il giocatore indossa un visore	44
3.5	File header della classe StartMenuItem	46
3.6	File source classe StartMenuItem	47
3.7	File header della classe astratta BaseWidgetController	48
3.8	Crezione di istanza di Widget con StartMenuItem come esempio	49
3.9	Aggiunta di Widget al Viewport	49
3.10	Creazione Widget VR per la classe <i>StartMenuItem</i>	50

caption = Funzione per aggiornare la posizione della CineCamera lungo il RigRail	52
3.11 Chiamata per l'aggiunta di una funzione all'evento overlap di un <i>TriggerBox</i>	52
3.12 Funzione per impostare la telecamera per la vista dell'utente	52
3.13 Funzioni per teleportare il giocatore lungo le posizioni per assistere alla gara	53
4.1 Metarace.build.cs file	56
4.2 Sezione del file header di BaseNetworkActor dove vengono importati i moduli necessari alla WebSocket e viene definita la mappa degli eventi	56
4.3 Sezione del file header di BaseNetworkActor dove viene definita la funzione che esegue il put nella mappa degli eventi	57
4.4 Sezione del file source di ABaseNetworkActor dove viene creata la WebSocket	57
4.5 La funzione che gestisce i messaggi in entrata alla WebSocket	58
4.6 Lambda function per la gestione della chiusura della connessione e dell'errore durante la connessione	58
4.7 Connessione della WebSocket	59
4.8 funzione che gestisce la ricezione del messaggio in ABaseNetworkActor	59
4.9 Funzione per inviare dati al server attraverso la WebSocket	59
4.10 Dichiarazione delegate nel file header di MetaraceNetworkActor	60
4.11 Namespace in cui vengono definite le keys	61
4.12 Funzione Initialize nel NetworkActor	62
4.13 Funzione OnAvailableRaceFormatsEvent	63
4.14 Funzione che spacchetta il file Json per la griglia iniziale	66
4.15 Funzione in <i>SceneController</i> che fa comparire i cavalli	67
4.16 Funzione nello SceneController che inizializza un <i>FTimerHandle</i>	69
4.17 Funzione che spacchetta il file Json relativo ai dati di gara	69
4.18 Funzione dello SceneController che inizializza i cavalli	70

Capitolo 1

Background tecnologico

1.1 Il Metaverso

Il concetto di Metaverso nasce nella letteratura di fantascienza, infatti il termine fu coniato da Neal Stephenson nel suo libro *Snow Crash* del 1992. Il termine descriveva uno spazio tridimensionale dove la realtà si univa con un mondo virtuale costantemente attivo. Nonostante questo concetto non sia recente, saper dare una definizione di cosa è il Metaverso genera molte difficoltà, infatti è una tecnologia che per la maggior parte ancora non esiste e sebbene riusciamo a ragionare su esperienze tecnologiche future siamo ancora lontani dal renderla possibile. Ancora non possiamo sapere quali caratteristiche saranno più importanti e che tipo di dinamiche guideranno la sua formazione, come negli anni '80 era difficile descrivere cosa sarebbe stato internet nel 2022, allo stesso modo è difficile saper descrivere il Metaverso. Ad oggi si può dire che il Metaverso è il nuovo principale obiettivo delle grandi compagnie di tecnologia mondiali, come Facebook - non a caso rinominata *Meta* - e Epic Games - azienda proprietaria del motore grafico Unreal Engine e di Fortnite, il videogioco che ad oggi viene considerato la piattaforma più vicina al Metaverso che sia stata fatta.

Matthew Ball, CEO di Epyllion e scrittore di *The Metaverse and How it Will Revolutionize Everything*, lo definisce in questo modo [1]:

Io definisco il metaverso come un network ampiamente scalabile e interoperabile di mondi virtuali 3D renderizzati in tempo reale che possono essere

vissuti, in modo sincrono e persistente, da un numero infinito di user effettivi, ciascuno con un senso di presenza individuale, supportando al contemporaneo continuo di dati quali cronologia, identità, comunicazione, pagamenti, diritti e oggetti.

Secondo Matthew Ball il Metaverso è quindi una combinazione di molte tecnologie diverse che collaborano per costruire un'esperienza continua e persistente. È l'unione di mondi virtuali, tecnologie - quali visori per la realtà virtuale, dispositivi indossabili e camere a proiezione 3D - e internet. È una nuova era della tecnologia che verrà costruita iteramente e lentamente al di sopra delle infrastrutture e dei protocolli esistenti che verranno migliorati o sostituiti in base alle esigenze. Un ruolo fondamentale lo avranno le piattaforme virtuali, esse infatti daranno effettivamente vita ai mondi virtuali in cui le persone potranno entrare. Alcune di queste hanno scopi puramente di intrattenimento - come Roblox, The Legend of Zelda o Minecraft - altri hanno intenti accademici e professionali - come Osso VR o come i simulatori di volo per l'addestramento di piloti.

1.1.1 I visori per la realtà aumentata e virtuale

Un importante contributo alla costruzione del Metaverso lo dobbiamo all'avanzamento tecnologico dei dispositivi per la realtà aumentata (AR) e per la realtà virtuale (VR). Questa tecnologia ha accompagnato il concetto del Metaverso sin dai suoi albori.

I visori per la realtà aumentata e per la realtà virtuale sono un avanzamento di un dispositivo ottico più antico, lo stereoscopio. Inventato nella prima metà dell'800, lo stereoscopio simula la tridimensionalità del mondo reale per come viene percepita dagli occhi umani. Infatti gli occhi umani trasmettono al cervello due immagini della stessa scena con due punti di osservazione leggermente diversi. Il cervello sovrapponendo le due immagini riesce a valutare la distanza degli oggetti: più un oggetto è scostato nelle due immagini più esso viene percepito come vicino, al contrario minore lo scostamento, maggiore è la distanza percepita. Lo stereoscopio permette di far vedere a ciascun occhio una tra due immagini molto simili tra loro, realizzate appositamente per essere percepite dal cervello umano come se fosse un singolo punto di vista reale. Questo conferisce al soggetto la tridimensionalità desiderata.

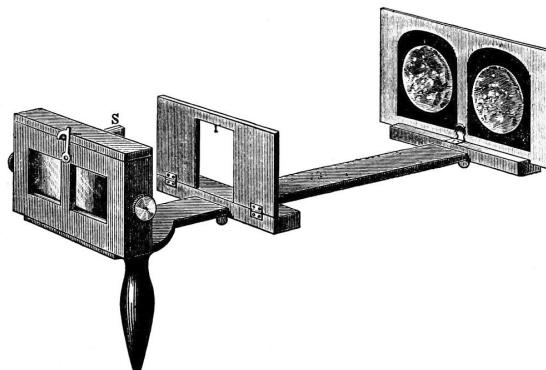


Figura 1.1: Stereoscopio statunitense regolabile.

Il primo a sfruttare questo principio per immergersi in una realtà simulata fu Ivan Sutherland nel 1968. Insieme ai suoi studenti, egli creò il primo sistema di realtà virtuale con visore che permetteva di osservare un ambiente virtuale renderizzato in wire-frame model (metodo che disegna solo gli spigoli di un oggetto 3d). Il visore era così pesante che doveva essere appeso al soffitto, per questo fu battezzato *La spada di Damocle*.

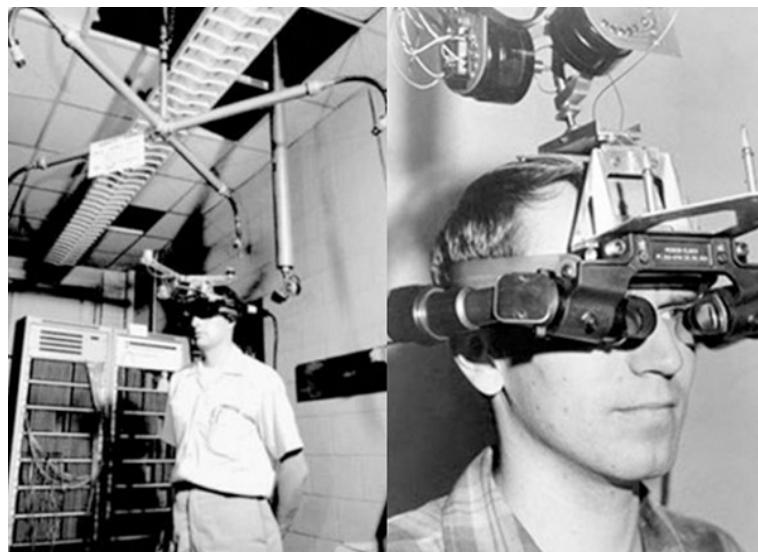


Figura 1.2: Il sistema di realtà virtuale progettato da Ivan Sutherland e soprannominato La Spada Di Damocle.

Dagli anni '70 fino agli anni '90 i dispositivi AR/VR furono designati quasi esclusivamente per scopi medici, simulazioni di volo, progettazione nell'industria automo-

bilistica e addestramento militare [15]. Bisogna aspettare fino al 1991 per assistere al primo visore VR lanciato sul mercato, ossia quando l'azienda Virtuality Group lanciò la serie 1000 dei prodotti Virtuality che girava sul Commodore Amiga 3000. Oltre ai prodotti Virtuality, gli anni '90 videro altre aziende puntare su questa tecnologia, come Sega con l'annuncio del Sega VR e la Nintendo con il lancio del Virtual Boy.



Figura 1.3: Il Nintendo Virtual Boy

Dopo allora l'interesse verso i visori da parte di grandi aziende è stato modesto fino al 2010, anno in cui venne prodotto il primo prototipo dell'Oculus Rift. Questo visore aveva il tracciamento della posizione, godeva di un angolo di visione di 90° e risolveva i problemi di distorsione che avevano i precedenti visori pre-distorcendo l'immagine renderizzata in tempo reale. Inoltre, nel 2013 l'azienda Valve inventò un display a bassa latenza che permise di creare schermi privi di lag e di motion blur indesiderato (il cosiddetto "*smear-effect*"). Questo in aggiunta all'avanzamento tecnologico di vari componenti sviluppati per smartphone - giroscopi, sensori di movimento, piccoli schermi HD e processori potenti miniaturizzati - diedero nuova linfa a questa tecnologia e vennero



Figura 1.4: La serie 1000 dei visori Virtuality



Figura 1.5: Il prototipo per il visore Sega VR

lanciati sul mercato molti modelli di visori.

Nel 2014 Google annunciò Cardboard, una piattaforma per la realtà virtuale fruibile con lo smartphone inserito all'interno di un visore. Un simile progetto lo distribuì Samsung nel 2015 con Samsung Gear VR. Entrambi i progetti non offrivano un'esperienza

coinvolgente e, nonostante fossero economici, non riscossero il successo desiderato e vennero dismessi. Nel 2016 Valve e l'azienda HTC lanciarono il visore HTC Vive. Questo apparecchio sfruttava una nuova tecnologia chiamata "*room scale*" che permetteva all'utente di muoversi liberamente in una zona di gioco invece di essere vincolato a stare fermo, inoltre i controller erano tracciati grazie a telecamere montate sul visore stesso e grazie ad una serie di sensori che andavano montati nella stanza. Sempre nel 2016 l'azienda Sony lanciò il Playstation VR, visore che montava un pannello OLED a 1080p e aveva un angolo di visuale di 100°. Questo visore non godeva di componenti di rilievo in confronto ai competitor ma si affacciava ad un mercato diverso, quello delle console.

1.1.2 Storia del metaverso

Sebbene il termine fu coniato nel 1992, il concetto di Metaverso affonda le radici nella letteratura Cyberpunk. Tale letteratura comprende romanzi come *True Names di Vernor Vinge* del 1981, che descrive quello che può essere considerato il primo esempio di cyber-spazio, e *Neuromancer di William Gibson* nel 1985, che invece descrive un cyber-spazio dalle caratteristiche molto simili al Metaverso che intendiamo oggi.

Il primo dei due libri è particolarmente importante perché è citato come fonte di ispirazione per il primo gioco nel Metaverso: Habitat [14].

1.1.2.1 Habitat - la prima implementazione di Cyber-spazio

Habitat viene definito dai creatori un *ambiente virtuale online multigiocatore* [14], ogni utente utilizzava il proprio Personal Computer come frontend comunicando su un network a commutazione di pacchetto con un sistema back-end centralizzato. La prima versione di Habitat era stata sviluppata per il Commodore 64 nel 1985 e non era possibile avere stanze virtuali popolose né grafica 3D a causa delle limitate risorse che il modello offriva. Nonostante questo, Habitat rese possibile per la prima volta a persone da tutto il mondo di incontrarsi in uno spazio virtuale. Gli utenti avevano una rappresentazione virtuale di se stessi in terza persona, questa rappresentazione venne chiamata avatar, come nel romanzo *True Names*. L'utente, attraverso il proprio avatar, aveva la possibilità di interagire con oggetti, parlare con altri avatar attraverso una chat che

appariva a schermo in stile "word balloon" e muoversi nel mondo di Habitat composto da un grande numero di posizioni che venivano chiamate *Regioni*.



Figura 1.6: Una tipica scena in Habitat.

Habitat fu un importante progetto che fece capire agli sviluppatori le difficoltà nell'approciarsi alla creazione di un ambiente virtuale multigiocatore e lasciarono in eredità un testo con le lezioni principali che impararono. [14]

1.1.2.2 Second Life

Un altro importante esempio di cyber-spazio è Second Life, una piattaforma online lanciata nel 2003 dalla società Linden Lab. Second Life è un ambiente virtuale online multigiocatore 3D e gli utenti sono rappresentati digitalmente attraverso un avatar tridimensionale. Attraverso gli avatar gli utenti hanno la possibilità di socializzare con altri utenti, sia attraverso chat testuale che vocale, esplorare il mondo virtuale, composto da migliaia di regioni chiamate *Sim* e partecipare ad attività di vario genere come concerti, raduni e lezioni e molto altro. La particolarità di questa piattaforma,

che la distingue da altri videogiochi 3D, è che il contenuto del mondo di gioco viene interamente creato dai giocatori e non c'è un obiettivo da perseguire né una storia. Inoltre Second Life possiede una sua economia interna e un token virtuale a circuito chiuso chiamato *Linden Dollar L\$*. Questa valuta non ha valore monetario ma può essere scambiata con Linden Lab per un corrispettivo in dollari. Essa può essere usata per comprare, vendere, affittare o commerciare beni e servizi con altri giocatori all'interno del mondo di gioco.

In Second Life per la prima volta brand e organizzazioni parteciparono alla realizzazione di oggetti ed eventi nel mondo virtuale portando il gioco ad evolversi e distaccarsi dall'essere una pura esperienza d'intrattenimento. Brand come Adidas, Calvin Klein e Lacoste lanciarono linee di vestiti indossabili dagli avatar dei giocatori [11] mentre alcune università hanno usato Second Life con obiettivi educativi e formativi, incluse l'Università di Harvard e di Oxford [16] ma anche alcune italiane come le università di Milano, di Torino, di Salerno e di altre città. [4] Un altro evento importante fu lo sciopero dei lavoratori IBM organizzato su Second Life nel 2007, durante questo sciopero migliaia di avatar contemporaneamente si radunarono per protesta sulla Sim dell'azienda.

1.1.3 Stato dell'arte

Oggi parlando di Metaverso si fa riferimento a piattaforme che riescono ad avvicinarsi ma che ancora non sono il Metaverso come definito in precedenza. Piattaforme come Roblox, Fortnite e Horizon Worlds presentano molti elementi del Metaverso, come una consistente rappresentazione 3D degli utenti, come la possibilità per gli utenti di creare i contenuti e come la possibilità di portare queste in una moltitudine di esperienze diverse. Questi "*giochi*" combinano insieme tante tecnologie e provano a produrre un'esperienza che si discosta da tutto ciò che è venuto prima. Ma per avere il Metaverso descritto da Matthew Ball ci vorranno ancora anni di ricerca e sviluppo. Infatti mancano ancora i protocolli, le innovazioni che possono permettere quella visione e soprattutto le tecnologie necessarie.

A livello infrastrutturare le tecnologie per accomodare migliaia o addirittura milioni di persone allo stesso momento in un'esperienza sincrona e persistente semplicemente

non esistono. E non solo, internet non è stato progettato per permettere esperienze simili. La maggior parte delle connessioni che avvengono sfruttano il protocollo TCP che garantisce la qualità dei dati ricevuti ma permette solo connessioni singole tra due utenti. In effetti è quello che sperimentiamo quando inviamo una mail o ci connettiamo su Facebook. Avvengono milioni di connessioni singole simultaneamente nel mondo ma è tutt'oggi molto difficile permettere a più di un centinaio di utenti di essere connessi tra di loro con connessioni full-duplex. Per garantire un'esperienza sincrona, il Metaverso avrebbe invece bisogno proprio di questo tipo di infrastruttura, un sistema simile alle attuali videochiamate o alle connessioni che avvengono all'interno dei videogiochi online ma in scala molto più grande. Le attuali piattaforme virtuali nascondono questa mancanza dividendo il loro mondo virtuale in diverse zone o partite, ognuna connessa con un server e con un numero finito di utenti, e mascherano il passaggio dell'utente da un server ad un altro come un passaggio tra queste divisioni.

Per garantire l'interoperabilità e la sincronia inoltre, il Metaverso avrà bisogno di un largo e complesso insieme di nuovi protocolli e standards. Infatti i sistemi attuali operano su formati simili ma non compatibili tra loro (basti pensare ai numerosi standard per la compressione d'immagini che sono presenti in internet). Questo è mitigato dal fatto che l'attuale esperienza di internet è asincrona, perciò quando un contenuto viene migrato da un sistema ad un altro viene spesso convertito nel formato di destinazione durante la fase non connessa. La difficoltà di questo passaggio è causata dalla naturale resistenza che hanno aziende e organizzazioni diverse a collaborare per creare uno standard condiviso, un esempio di ciò è come Apple sia resistente ad adottare lo standard USB-C sui suoi dispositivi o anche come siano presenti un gran numero di standard per le prese di corrente nel mondo.

Perciò Matthew Ball immagina che il Metaverso arriverà, similmente per come è stato con internet, come un disordinato insieme di processi e sviluppi diversi che gradualmente si uniformerà. La differenza sostanziale è che internet si sviluppò in ambito universitario e aveva come scopo quello di mettere in comunicazione le facoltà sparse nel mondo. Al contrario in questo momento storico è l'industria privata che detiene la consapevolezza del potenziale del Metaverso, le risorse economiche ed ingegneristiche per svilupparlo e tutto l'interesse a far sì che questo divenga realtà.

1.1.3.1 Epic Games

Una delle caratteristiche più importanti che deve avere il Metaverso secondo Ball è quella di essere un ambiente attraente per gli utenti e le aziende. Quello che ad oggi si è avvicinato di più a questo scopo è il gioco Fortnite. Sviluppato e distribuito da Epic Games, questo gioco nel tempo, e grazie alla sua fama, è diventato un'occasione di incontro e di socialità, ma soprattutto si è avvicinato al concetto di Metaverso in quanto il mondo virtuale di gioco è diventato luogo di eventi sociali indipendenti dal gioco stesso quali concerti e luogo di visione di contenuti multimediali a cui gli utenti possono assistere con il proprio avatar. Nella piattaforma, infatti, si sono svolti concerti di artisti famosi come Marshmellow, Ariana Grande e Trevis Scott e sono stati distribuiti i trailer di Star Wars e di Tenet in anteprima.

Fortnite inoltre è una delle poche piattaforme che presenta livelli di interoperabilità così elevati. Infatti, Fortnite si interseca con varie proprietà intellettuali di aziende quali la Marvel, la DC, la Sony, è accessibile attraverso dispositivi correnti (iPhone, Android, Playstation, Xbox e computer) e include anche diversi sistemi di identità e



Figura 1.7: Set di The Mandalorian con scenografia renderizzata con Unreal Engine

pagamento.

Inoltre Epic Games è proprietaria del secondo game engine più utilizzato del settore: Unreal Engine. Questo engine nel tempo è diventato così completo e realistico che ha iniziato a venir usato anche per grosse produzioni hollywoodiane, come *The Mandalorian*, ma anche per progetti di architettura urbana, per progettazione di automobili e per esercitazioni militari simulate dell'esercito degli Stati Uniti e del Regno Unito [1, 2, 13]. Tutto questo rende il loro sistema potenzialmente già compatibile con centinaia di contenuti velocizzando il processo di migrazione di applicativi all'interno del Metaverso.

1.2 Il motore grafico Unreal Engine 5

Come anticipato nel capitolo 1.1.3.1, Unreal Engine è un motore grafico 3D sviluppato da Epic Games. La prima versione del software fu sviluppata interamente da Tim Sweeney, il fondatore della società, dal suo garage. Tim Sweeney iniziò lo sviluppo nel 1995 e il motore grafico debuttò con un videogioco chiamato *Unreal*, sviluppato da Cliff Bleszinski e James Schmalz, nel 1998. Nonostante né il motore grafico né il gioco fossero completati del tutto, Unreal ottenne il successo desiderato anche grazie a una fedele community che utilizzando l'UnrealScript (il codice proprietario sviluppato da Sweeney per il game engine) espansero il gioco e rese Epic MegaGames (poi rinominata Epic Games) un nome importante nell'industria videoludica. Inoltre il gioco fu anche la vetrina desiderata per l'engine che da quel momento fu utilizzato sotto licenza da altri sviluppatori che portarono Unreal Engine a diventare un motore grafico apprezzato ed accreditato nel settore [17, 18].

Unreal Engine fu lanciato poco prima il passaggio alle schede grafiche dedicate. Questo ha fatto sì che la prima versione si basava totalmente sul software rendering, ossia sfruttava unicamente la CPU. Sweeney sviluppò l'editor di Unreal engine in Visual Basic e prese fortemente ispirazione dall'ambiente di lavoro di quest'ultimo per l'interfaccia utente, soprannominò l'editor Unrealed. Infatti già dalle prime versioni di Unreal Engine c'erano dei moduli che si potevano trascinare nella scena direttamente, poi una volta nella scena bastava fare un doppio click su di essi per far comparire l'editor di testo e poter scrivere il codice desiderato [12].

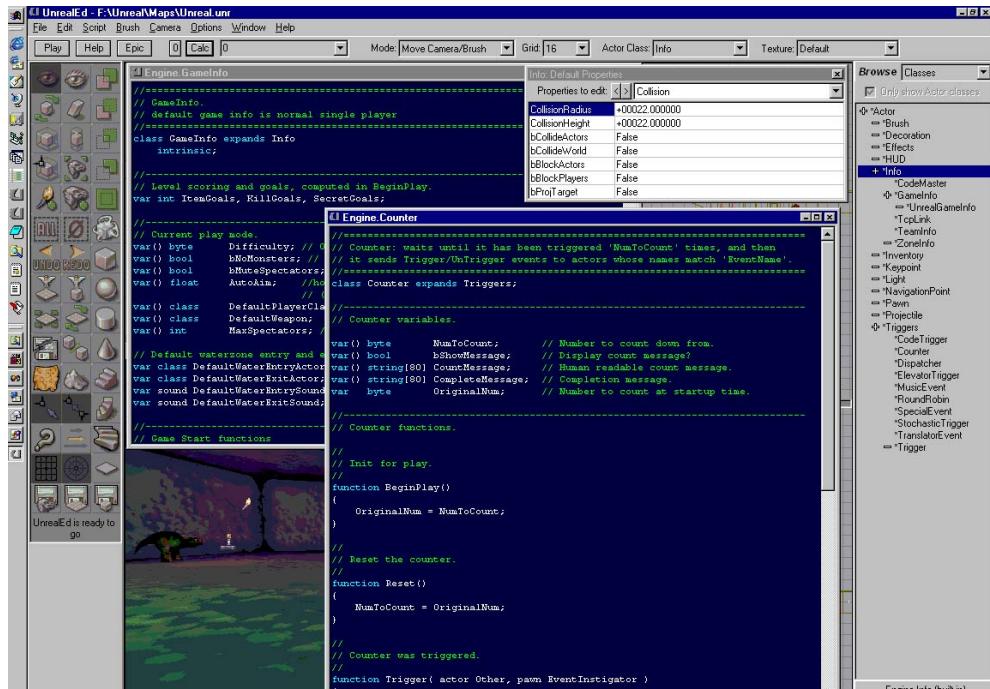


Figura 1.8: Screenshot della prima versione di UnrealEd.

Ad oggi il motore grafico ha abbandonato sia UnrealScript che Visual Basic ma le scelte iniziali sono rimaste. Infatti le classi native che troviamo oggi nella libreria Unreal C++ sono le stesse che c'erano in UnrealScript. Sono ancora presenti le classi native principali Actor, Pawn e Character: la classe Actor è la classe base per tutti gli oggetti che vengono disposti o vengono generati nel livello; la classe Pawn è un'espansione della classe Actor per tutti gli oggetti che posso essere governati da giocatori o dall'intelligenza artificiale; la classe Character è un'espansione della classe Pawn che possiede una Mesh, delle Collisions e una logica di movimento incorporata. Alla creazione di una classe l'editor fa scegliere la parent class e in base alla scelta viene generato automaticamente il rispettivo script di base [6].

Algoritmo 1.1: File header generato alla creazione di un Actor

```

1 #include "GameFramework/Actor.h"
2 #include "MyActor.generated.h"
3
4 UCLASS()
5 class AMyActor : public AActor

```

```

6 {
7     GENERATED_BODY()
8
9 public:
10
11     // Sets default values for this actor's properties
12     AMyActor();
13
14     // Called when the game starts or when spawned
15     virtual void BeginPlay() override;
16
17     // Called every frame
18     virtual void Tick(float DeltaSeconds) override;
19
20 };

```

È possibile inoltre esporre una variabile dello script all'editor tramite appositi specificatori, espandibili con una lista di proprietà tra cui scegliere:

Algoritmo 1.2: Specificatore UPROPERTY per esporre una variabile all'editor

```

1 UPROPERTY(EditAnywhere, category="VariableCategory")
2 int32 MyVariable;

```

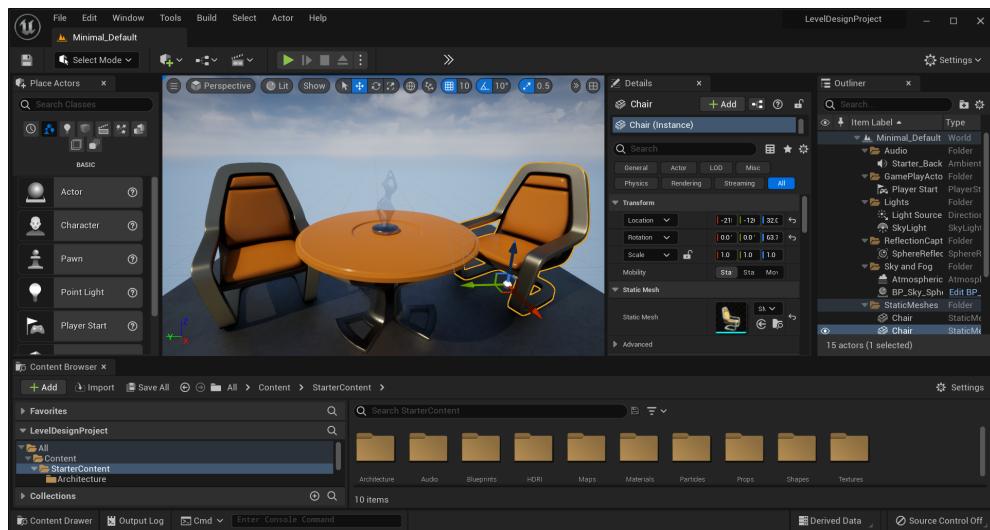


Figura 1.9: Screenshot della versione 5 dell'editor di Unreal Engine.

Anche l'impostazione iniziale organizzata in moduli è stata mantenuta. I moduli sono gli oggetti che possono essere trascinati in scena e che hanno funzionalità definite. È possibile espandere la lista di oggetti trascinabili in scena con quelli personalizzati dallo sviluppatore, si troveranno nel Content Browser e non nel pannello di Place Actor.

In Unreal Engine la scena nel quale viene creata l'esperienza di gioco è chiamata Level. Un Level è un ambiente 3D che può essere popolato da oggetti e geometrie. Ogni oggetto che viene posizionato nel Level è un Actor. Più tecnicamente infatti un Actor è la classe che definisce un qualsiasi oggetto a cui è associato un Transform, ossia l'informazione sulla posizione, sulla rotazione e sulla scala [8].

L'interfaccia utente è composta da diverse finestre ed è altamente personalizzabile. Alla prima accensione dell'engine viene proposto il layout di default.



Figura 1.10: Interfaccia di default di Unreal Engine 5

1. **Tab Bar and Menu Bar:** Il Lever Editor ha delle schede simili a quelle che hanno i browser in alto. Infatti oltre al livello si possono navigare tutti i tipi di oggetti singolarmente e qui possono essere raggruppate tutte le schede.
La Menu Bar offre accesso agli strumenti generali dell'editor quali le impostazioni del progetto e le preferenze.
2. **Toolbar:** Questo pannello mostra un gruppo di comandi, offrendo così rapido accesso agli strumenti e alle funzionalità più utilizzate.

3. **Bottom Toolbar:** Contiene scorciatoie per la Console di comando, per l'Output Log e per la funzionalità di Delivered Data
4. **Place Actor / Modes:** Il Lever Editor può essere messo in diverse modalità attivando specifiche interfacce di editing per particolari tipi di azioni che si vogliono compiere nel livello. Le possibili modalità sono:
 - Select: per selezionare, spostare e aggiungere attori nella scena.
 - Landscape: per la creazione e la modifica di vaste aree di terreno.
 - Foliage: per la creazione e la modifica di un alto numero di istanze di Static Mesh per la popolazione della scena con elementi naturali
 - Mesh Paint: per la pittura di vertici e texture su Static Mesh direttamente nella Viewport.
 - Modeling: per la creazione di mesh poligonali semplici
 - Fracture: per la creazione di oggetti e ambienti distruttibili
 - Brush Editing: per la modifica della geometria delle mesh.
 - Animation: per creare e modificare animazioni.
5. **Viewports:** questo pannello è la finestra verso il mondo che l'utente sta creando. Permette di selezionare, spostare, ruotare e scalare gli oggetti direttamente con il mouse e di muovere il punto di vista all'interno della scena.
6. **Content Browser / Content Drawer:** permette di visualizzare tutti gli asset di gioco e di organizzarli in cartelle
7. **Outliner:** permette di visualizzare e selezionare tutti gli attori presenti nella scena in una vista gerarchica ad albero.
8. **Details:** questo pannello contiene le informazioni, le funzionalità e le utilità degli oggetti che vengono selezionati. Contiene i box per la modifica dei dati del Transform per muovere, ruotare o scalare gli oggetti e mostra tutte le proprietà editabili in base al tipo di attore che si seleziona. È qui che vengono visualizzate le variabili esposte all'editor dallo script in C++.

Unreal Engine permette di aggiungere funzionalità a classi già presenti del motore tramite le classi Blueprints. Esse permettono di raggruppare componenti di tipologie diverse, chiamati appunto Components, sotto un'unica classe e di espandere le loro funzionalità attraverso la programmazione. In Unreal Engine la programmazione può essere implementata sia attraverso C++ che attraverso la programmazione visiva organizzata a nodi dei Blueprints, chiamata Blueprints Visual Scripting. Il vantaggio della Blueprints Scripting è che è di veloce implementazione soprattutto per i principianti della programmazione. D'altro canto però questa programmazione visiva non da libero accesso al codice di gioco, è meno flessibile e meno efficiente della programmazione C++. Invece, lo svantaggio principale di C++ è che è molto più facile commettere errori che compromettono l'intero engine. Infatti bisogna fare particolare attenzione all'utilizzo dei puntatori, e delle funzioni che li sfruttano, perché l'utilizzo di uno di questi ancora non inizializzato porterà ad una *null pointer exception* e al conseguente crash dell'engine. Inoltre, questo tipo di errori sono spesso di difficile interpretazione.



Figura 1.11: Esempio di classe Blueprint per l'apertura e la chiusura di una porta in gioco

1.3 Il software di modellazione Blender

Unreal Engine è un motore di gioco che sebbene presenta una grande varietà di strumenti, non dà la possibilità di creare da zero oggetti con geometrie complesse. Per questo Unreal Engine, come altri motori di gioco, viene utilizzato in combinazione con altri software per la creazione di oggetti 3D, come Maya3D, Blender e Rhinoceros. Per

la creazione degli oggetti 3D di Metarace è stato scelto il software Blender per la sua natura multipiattaforma e per la sua leggerezza di utilizzo. Infatti occupa poco spazio sulla memoria del dispositivo e permette di essere utilizzato anche su macchine poco potenti.

Blender è un software per la creazione di contenuti 3D gratuito e *open source*. Supporta l'intera pipeline di strumenti per il 3D - modellazione, rigging, animazioni, creazione e pittura di texture, simulazioni, rendering, compositing e motion tracking. È un software cross-platform, è disponibile per Linux, Windows e iOS e l'interfaccia utilizza OpenGL. La natura open source del progetto permette al pubblico di fare modifiche al codice sorgente e di creare plug-in per aggiungere features [3].

Blender fu creato da Ton Roosendaal, direttore artistico della casa di animazione olandese NeoGeo e sviluppatore software autodidatta. Il primo file sorgente fu creato nel 1994 e il software doveva essere uno strumento proprietario della casa di animazione. Nel 1998, dopo la chiusura di NeoGeo, Ton Roosendaal fondò l'azienda *Not a Number Technologies (NaN)* per continuare lo sviluppo e Blender fu distribuito come freeware. Nel 2002 la NaN andò in bancarotta e lo sviluppo di Blender fu interrotto. Gli investitori chiedevano un pagamento per la licenza di Blender perciò Roosendaal nello stesso anno creò una fondazione senza scopo di lucro, chiamata Blender Fondation, per raccogliere i fondi necessari. La fondazione lanciò una campagna di crowdfunding che raccolse centodiecimila euro in 7 settimane e nel 2002 Blender fu distribuito come software *open source*. Oggi lo sviluppo di Blender continua grazie alla numerosa community e a 24 dipendenti della Blender Institute.

Dal lancio del software come open source l'interfaccia utente è cambiata e il motore di render si è evoluto ma le caratteristiche principali sono rimaste invariate. Una delle caratteristiche principali di Blender è che è possibile chiamare quasi tutte le funzioni tramite scorciatoie da tastiera. Per questo motivo quasi tutti i tasti della tastiera sono associati ad una o più funzioni. Le funzioni più utilizzate sono associate a tasti che richiamano il nome della funzione che utilizzano, ad esempio il tasto "*G*" chiama la funzione *Grab* che permette di spostare l'elemento selezionato, il tasto "*R*" la funzione *Rotate*, il tasto "*E*" la funzione *Extrude* e così via.

Un'altra caratteristica di Blender è che lo spazio di lavoro è totalmente ad oggetti:

l'interfaccia è divisa in finestre che è possibile dividere a loro volta in altre finestre e sottofinestre ognuna delle quali può diventare qualsiasi tipo di vista o immagine che il programma supporta. L'utente può personalizzare la combinazione di finestre e viste per ognuna delle operazioni che preferisce e può posizionare questi layout in schede similmente a come avviene in un browser web. Il programma offre già layout di default per lavorare a compiti diversi.

Blender permette di creare delle Mesh poligonali e di esportarle in formati compatibili con altri programmi 3D, compreso Unreal Engine. Una Mesh poligonale è una struttura dati che rappresenta oggetti nello spazio 3D. In inglese significa maglia perché è strutturata come un reticolo.



Figura 1.12: Una Mesh poligonale in Blender

Capitolo 2

Il core di Metarace

2.1 Metarace

Metarace è un progetto in Unreal Engine 5 che punta a sfruttare i dispositivi AR/VR per far immergere l'utente in un mondo virtuale di competizioni equestri. Si definisce un progetto nel Metaverso perché punta ad avere le caratteristiche che definiscono tutte le piattaforme attualmente vicine a questo concetto: una rappresentazione digitale della persona sotto forma di avatar, un ambiente virtuale in cui è possibile interagire e socializzare con altri utenti, degli oggetti di gioco personalizzabili e portabili al di fuori di Metarace, un'economia interna e una progettazione focalizzata sulla scalabilità per permettere in futuro di aggiornare l'applicativo all'arrivo di nuove tecnologie. In Metarace il giocatore può possedere dei cavalli virtuali e li può iscrivere a delle competizioni in cui competono con i cavalli di altri giocatori. Il giocatore può assistere alle competizioni con un avatar virtuale entrando nel mondo attraverso un visore per la realtà virtuale. In mancanza di questo può comunque assistere alla gara tramite desktop. Il vincitore di ogni gara sarà determinato da una componente intrinseca al cavallo in termini di forza innata e una componente randomica relativa alla singola gara.

L'idea dietro al gioco è quella di far immergere l'utente in un ambiente simile a quello delle competizioni equestri che si svolgono nel mondo reale, dove quindi gli spettatori e i proprietari dei cavalli non prendono parte attiva alla competizione ma vi assistono dagli spalti e dove la vittoria non è mai certa. L'idea di Metarace è di riproporre questa

formula in un ambiente virtuale immersivo, nel Metaverso.

2.2 La meccanica di base

Una partita di Metarace consiste in una gara di cavalli in cui c'è sempre un vincitore, con una piccola probabilità di pareggio. Il risvolto della gara è dato da caratteristiche intrinseche dei cavalli gareggianti e da fattori casuali.

Siccome il giocatore non prende parte attiva alla gara ma vi assiste attraverso il proprio avatar, l'esito della gara è deciso da un server che invia il risultato a ciascun client dei partecipanti. Il client calcola, attraverso un algoritmo, il comportamento dei cavalli in modo che sia coerente con il risvolto della gara inviato dal server. Il server invia al client un insieme di array di tempi. Il client divide il percorso in un numero di step uguale alla cardinalità di questi array e poi fa percorrere ai cavalli questi step sulla base dei tempi stessi. Il client perciò si occupa di renderizzare il mondo di gioco, di far iscrivere l'utente alle competizioni che desidera e a consentire la fruizione della gara. Durante lo sviluppo, è stata trovata subito la necessità di testare come poter tradurre questi dati inviati dal server in movimenti dei gareggianti all'interno del mondo di gioco.

2.2.1 L'algoritmo per il movimento dei cavalli

Lo strumento più adatto tra quelli messi a disposizione da Unreal Engine 5 per implementare questa meccanica è la Spline Component. Una Spline Component è un percorso che lo sviluppatore può definire per utilizzare dati posizionali. Può essere usata per far muovere oggetti o per posizionare una serie di Actor lungo di essa. Questo componente è composto da una serie di nodi ordinati, posizionabili all'interno del mondo di gioco, e da una linea che segue questi nodi in maniera interpolata andando a creare così il percorso. È inoltre possibile far seguire una Spline ad una Mesh (o ad un Actor) impostando un'animazione basata sul tempo. Questo in Unreal è possibile farlo attraverso una Timeline. Una Timeline è uno strumento offerto da Unreal Engine 5 che permette di ottenere dei valori di output nel tempo basati su una Curve impostata inizialmente.

Quello che ho implementato è stato di creare una classe C++ che rappresenta il cavallo e un Blueprint che la estendeva. Inoltre ho creato un altro Blueprint che contiene

tutte le Spline del percorso. Ogni cavallo sarà associato ad una delle Spline e la seguirà secongo l'algoritmo seguente:

Algoritmo 2.1: Sezione del file header (movimento cavallo)

```

1 FTimeline RaceTimeline;
2 UPROPERTY()
3 USplineComponent* SplineComponent = nullptr;
4 UPROPERTY()
5 UCurveFloat* CurveFloat;
6

```

La Timeline si utilizza attraverso una funzione che ne cattura il progresso (legata con una *FOnTimelineFloat*) e grazie ad una *CurveFloat* che ne definisce il comportamento. Per legare un oggetto *FOnTimelineFloat* alla funzione per catturare l'output della Timeline si utilizza la funzione *BindUFunction*. Per associare la Timeline alla *FOnTimelineFloat* e alla *CurveFloat* si utilizza la funzione *AddInterpFloat* chiamata dalla Timeline stessa.

Algoritmo 2.2: Inizializzazione della Timeline nel file source (movimento cavallo)

```

1
2 void AHorse::Inizialize()
3 {
4     [...] //creazione della CurveFloat
5
6     FOnTimelineFloat RaceTimelineProgress;
7     RaceTimelineProgress.BindUFunction(
8         this, FName("RaceTimelineProgress"));
9     if(RaceCurveFloat){ //per evitare nullptr exception
10         RaceTimeline.AddInterpFloat(
11             RaceCurveFloat, RaceTimelineProgress);
12     }
13     [...] // Settaggio del RatePlay della Timeline
14 }
15
16

```

Per far partire la Timeline bisogna chiamare la funzione *Play()*.

```

1 void AHorse::StartTimeline()

```

```

2 {
3     RaceTimeline.Play();
4 }
```

Il valore di output della Timeline (che ho chiamato *Value*) è standardizzato tra 0 e 1. È possibile eseguire un interpolazione tra 0 e la lunghezza della Spline basata su questo valore e ottenere così una posizione lungo la Spline. La posizione dell'Actor viene quindi impostata uguale alla posizione trovata.

Algoritmo 2.3: Funzione che sfrutta il valore restituito dalla timeline nel tempo (movimento cavallo)

```

1 void AHorse::RaceTimelineProgress(float Value)
2 {
3     if(MeshComponent && SplineComponent)
4     {
5         float DistanceInSpline = FMath::Lerp(0,
6             SplineComponent->GetSplineLength(),
7             Value);
8         FTransform TransformAtDistance =
9             SplineComponent->GetTransformAtDistanceAlongSpline(
10                 DistanceInSpline,
11                 ESplineCoordinateSpace::World);
12         SetActorLocationAndRotation(
13             TransformAtDistance.GetLocation(),
14             TransformAtDistance.GetRotation());
15     }
16 }
```

Questa funzione viene chiamata automaticamente dalla Timeline se all'interno della funzione *Tick* le viene passato il valore *DeltaTime*. La funzione *Tick* è chiamata automaticamente da Unreal Engine ad ogni frame e il valore *DeltaTime* è il tempo che è intercorso dall'ultima chiamata della funzione *Tick* (equivalente al tempo intercorso dall'ultimo frame).

Algoritmo 2.4: Funzione Tick (movimento cavallo)

```

1 void AHorse::Tick(float DeltaTime)
2 {
```

```

3     Super::Tick(DeltaTime);
4
5     RaceTimeline.TickTimeline(DeltaTime);
6 }
7

```

Per far muovere la Mesh Component coerentemente all'array di tempi inviato dal server, bisogna creare una *FloatCurve* che si basa su di essi. Questo viene fatto nella funzione *Inizialize*. La classe *UCurveFloat* non offre funzioni che permettono di instanziarne una a tempo di esecuzione, per questo viene utilizzata una *FRichCurve* per la sua creazione runtime. Sia la *FRichCurve* che la *CurveFloat* sono costituite da una serie di punti su un piano cartesiano e hanno entrambe delle regole di interpolazione da un punto ad un altro. Le regole per la definizione dell'interpolazione si possono definire grazie alla struct *ERichCurveInterpMode* che può essere: *RCIM_Linear* (interpolazione lineare), *RCIM_Cubic* (interpolazione cubica) oppure *RCIM_None* (nessuna interpolazione). Si inizia con la definizione di variabili temporanee per lo spazio e il tempo, si definisce inoltre la lunghezza standardizzata di ogni step sulla curva e si calcola il tempo totale che il cavallo impiegherà a percorrere tutto il percorso:

Algoritmo 2.5: Definizione variabili temporanee (movimento cavallo)

```

1 void AHorse::Inizialize()
2 {
3     float TempS = 0; //Variabile temporanea per lo Spazio
4     float TempT = 0; //Variabile temporanea per il Tempo
5
6     //Lunghezza di ogni step standardizzata
7     float DeltaS = 1 / float(TimeArray.Num());
8
9     for(float Time : TimeArray)
10    {
11        //Tempo complessivo per percorrere tutto il percorso
12        TotalTime += Time;
13    }
14

```

Una volta definite queste variabili si inizia la creazione della curva. Il numero dei punti sul piano cartesiano equivale al numero degli elementi dell'array dei tempi più

un punto iniziale all'origine del piano cartesiano. Un'istanza della classe *FRichCurve* è costituita da una serie di *FRichCurveKey*, perciò popolo un *TArray* con delle keys di questo tipo.

```

15     RichCurve = new FRichCurve();
16
17     TArray<FRichCurveKey> Keys;
18     FRichCurveKey FirstKey = FRichCurveKey(0, 0, 0, 1,
19                                         ERichCurveInterpMode::RCIM_Cubic);
20     Keys.Add(FirstKey);
21

```

Per la creazione delle keys itero su tutti gli elementi dell'array di tempi e ad ogni iterazione sommo il tempo i con la somma di tutti i tempi precedenti diviso la somma di tutti i tempi in modo da ottenere un valore standardizzato. Ossia ogni key ha come variabile tempo:

$$T_{i=0..n} = \sum_{j=0}^i \frac{t_j}{t_{tot}} \quad (2.1)$$

Ogni key ha come variabile spazio la lunghezza standardizzata del singolo step moltiplicato il numero di iterazione corrente:

$$S_{i=0..n} = \Delta S(i + 1) \quad (2.2)$$

Infine è stato scelto di non interpolare l'ultima key generata perché questo comporterebbe la generazione di una CurveFloat che riporterebbe a zero la velocità all'avvicinarsi del traguardo. Si può notare la differenza nella curva generata nella Figura: 2.1.

```

22     for (int i = 0; i < TimeArray.Num(); i++)
23     {
24         TempS = DeltaS * (i + 1);
25         TempT += TimeArray[i] / TotalTime;
26         FRichCurveKey Key;
27         if ((i + 1) < TimeArray.Num())
28         {
29             Key = FRichCurveKey(TempT, TempS, 0, 0,
30                                 ERichCurveInterpMode::RCIM_Cubic);
31         }

```

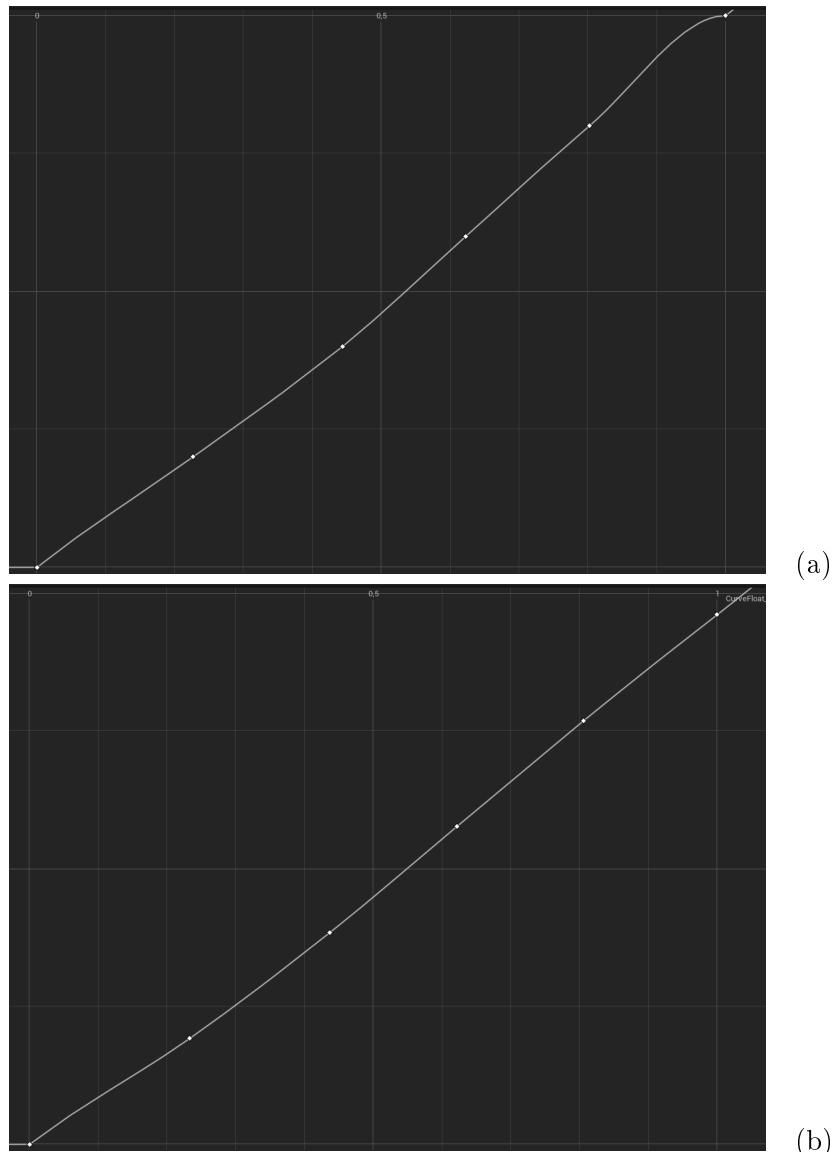


Figura 2.1: CurveFloat generata con tutti i nodi interpolati (a) CurveFloat generata con tutti i nodi interpolati eccetto l'ultimo. (b)

```
32     else
33     {
34         Key = FRichCurveKey(TempT, TempS, 1, 0,
35                             ERichCurveInterpMode::RCIM_None);
36     }
37     Keys.Add(Key);
```

```
38     }
39 }
```

Una volta generate le keys le aggiungo alla *FRichCurve*, poi istanzio un nuovo oggetto *UCurveFloat* e setto le sue keys essere quelle della *FRichCurve*.

```
40     RichCurve ->SetKeys(Keys);
41     UCurveFloat* RaceCurveFloat = NewObject<UCurveFloat>();
42     RaceCurveFloat->FloatCurve = *RichCurve;
43
44     [...] // Inizializzazione della Timeline
45 }
```

Infine imposto il RatePlay della Timeline uguale alla frequenza con cui il cavallo si muove lungo il percorso.

```
46     RaceTimeline.SetPlayRate(1 / TotalTime);
47 }
48 }
```

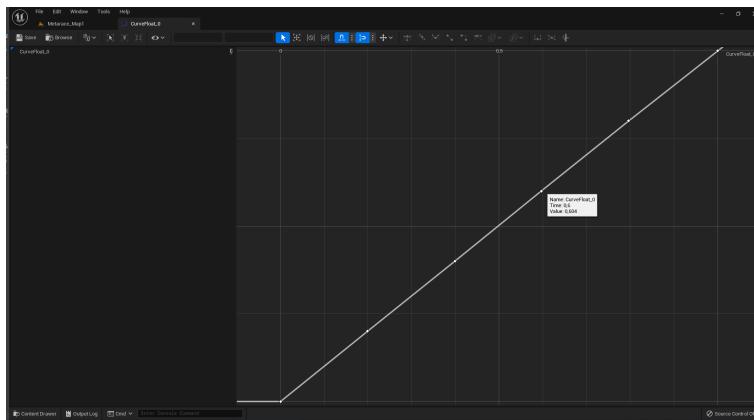


Figura 2.2: *UCurveFloat* generata dall'algoritmo per il movimento dei cavalli

Questo processo genera una curva come quella mostrada in Figura 2.2. Questa curva garantisce un cambiamento di velocità fluido del cavallo durante tutta la gara nonostante, dall'immagine, i cambiamenti di coefficiente angolare non siano apprezzabili.

Infine è stata utilizzata una seconda *Timeline* per la decellerazione del cavallo che avviene dopo il traguardo. In questo contesto la Timeline è stata utilizzata per creare una decellerazione coerente con la velocità posseduta da ogni cavallo al momento del

passaggio al traguardo: più un cavallo arriva veloce al traguardo più spazio percorrerà per fermarsi. In questo caso però la curva di input per la Timeline è fissa.

2.3 Animazione dei cavalli

I cavalli non soltanto si muovono lungo il percorso ma possiedono una serie di animazioni che vengono riprodotte in base allo stato in cui si trovano e alla velocità che possiedono. I cavalli infatti sono delle Mesh associate ad uno scheletro gerarchico di ossa che può essere animato per muovere la Mesh. Questo tipo di Mesh vengono gestite da Unreal Engine come delle Skeletal Mesh. Per la gestione delle animazioni in Unreal Engine si può utilizzare uno strumento chiamato *AnimGraph*. Questo strumento valuta la pose da mostrare per una Skeletal Mesh in ogni frame in base alle animazioni passate come input [10].

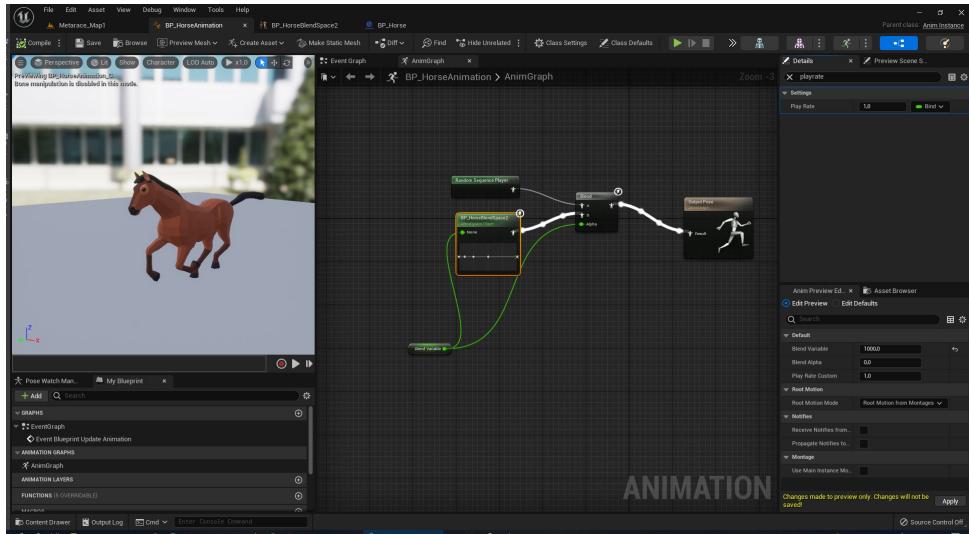


Figura 2.3: *AnimGraph* che gestisce l'animazione del cavallo.

L' *AnimGraph* mostrato in Figura 2.3 utilizza due lettori di animazione: un *BlendSpace2D* e un *Random Sequence Player*.

Il *Random Sequence Player* è un lettore che permette di salvare una lista di animazioni da riprodurre. Queste animazioni vengono riprodotte in maniera casuale una

dopo l'altra. Questo nodo è perfetto per riprodurre le animazioni *Idle* del cavallo (le animazioni per quando il cavallo è fermo).

Il *BlendSpace2D* invece è un asset speciale che permette di fare il blend tra più animazioni sulla base di un valore in input. Per sfruttarlo bisogna inserire le animazioni lungo l'asse delle ascisse che il nodo mostra. Posizionando le animazioni in maniera adeguata, ossia per far combaciare il valore di input con la frequenza di riproduzione delle animazioni, si ottiene un blend automatico tra le animazioni legato al valore di input. L'editor aiuta in questo processo segnalando un errore quando l'animazione non combacia con il valore sull'asse. In questo contesto è stato utilizzato come valore di input la velocità del cavallo e questo fa sì che ognuno di questi possiede, in ogni momento, un'animazione coerente con l'andatura che mantiene. Questa tecnica risolve il problema del "pattinamento" - ossia quando un'animazione è più lenta o più veloce di quanto un Actor si muova in scena avendo come effetto quello di sembrare che scivoli (che appunto pattini) sul terreno - nonostante le animazioni date in input non sarebbero adeguate per la velocità sostenuta. Questo perché l'animazione risultante è in realtà un'interpolazione delle animazioni di input in base, appunto, alla velocità. Le animazioni inserite in questo BlendSpace sono 4: una di *Idle*, una di camminata, una per il trotto, una per la corsa leggera e una per il galoppo.

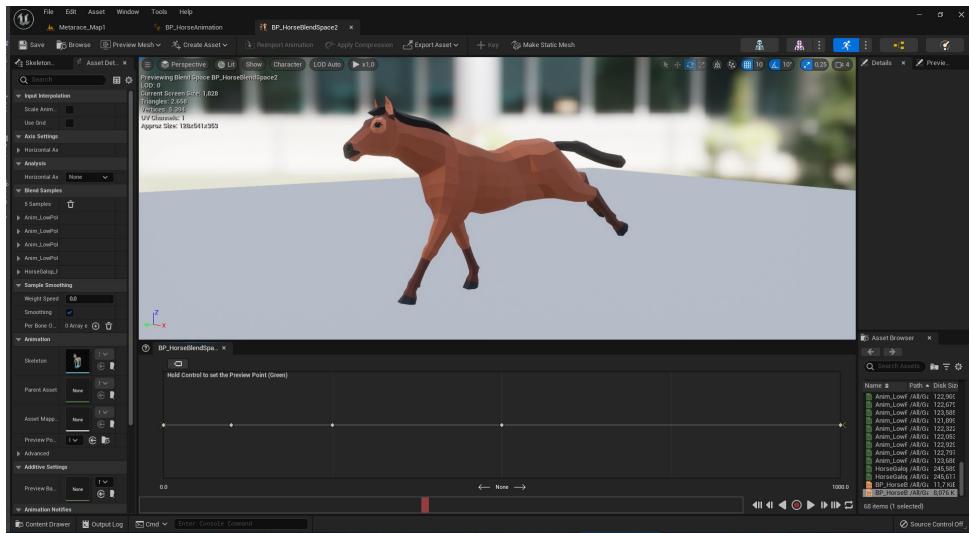


Figura 2.4: *BlendSpace2D* per l'interpolazione delle animazioni sulla base della velocità.

Come si può vedere dalla Figura 2.3, il valore di input per il *BlandSpace* è utilizzato anche come valore di blend tra il *BlendSpace2D* e il *Random Sequence Player*. Questo perché quando il valore di input è nullo vuol dire che si è raggiunta l'andatura in cui è possibile riprodurre una tra le animazioni di *Idle* e perciò ne viene riprodotta una casualmente.

2.3.1 Animazione del galoppo creata con Blender

Un cavallo non può considerarsi un cavallo da corsa senza un'animazione del galoppo. Per questo motivo ho ideato e creato l'animazione in questione nella suite Blender.

La suite Blender permette di animare qualsiasi tipo di oggetto e qualsiasi tipo di deformazione o traslazione tramite la tecnica di inserimento di *keyframe poses* all'interno di una finestra apposita: l'Action Editor. Letteralmente keyframe pose vuol dire "posa del fotogramma chiave", questo perché questa tecnica prevede di salvare tutti i dati sul Transform (quindi su posizione, rotazione e scala) di uno o più oggetti all'interno di vari frame che riprodotti in sequenza andranno a creare l'effetto di movimento che compone l'animazione. Questa tecnica è particolarmente efficace per le mesh associate ad uno scheletro gerarchico di ossa (chiamato rig). Infatti muovendo una o più ossa si andrà

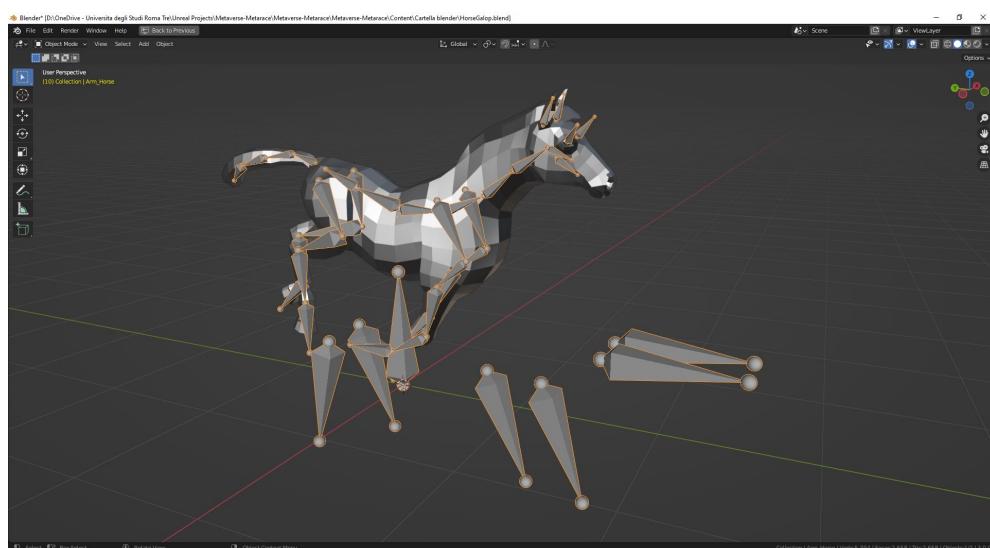


Figura 2.5: Vista in Blender della Mesh del cavallo, con il suo rig associato, in una pose tra le 12 dell'animazione del galoppo

a deformare la mesh senza cambiarne la geometria ma solo cambiandole la posizione, creando una pose, e salvando queste pose nelle keyframe si potrà salvare ed esportare la riproduzione delle keyframe come un'animazione.

Creare animazioni convincenti è un'arte molto complessa che va studiata a fondo. Spesso il modo migliore per crearne una è partire da una reference del mondo reale e così ho fatto. Sono partito da un'immagine che rappresenta lo studio alla base del movimento del galoppo diviso in 12 frame e ho cercato di far combaciare la posizione della mesh con quella che si vede nell'immagine per ogni singolo frame.

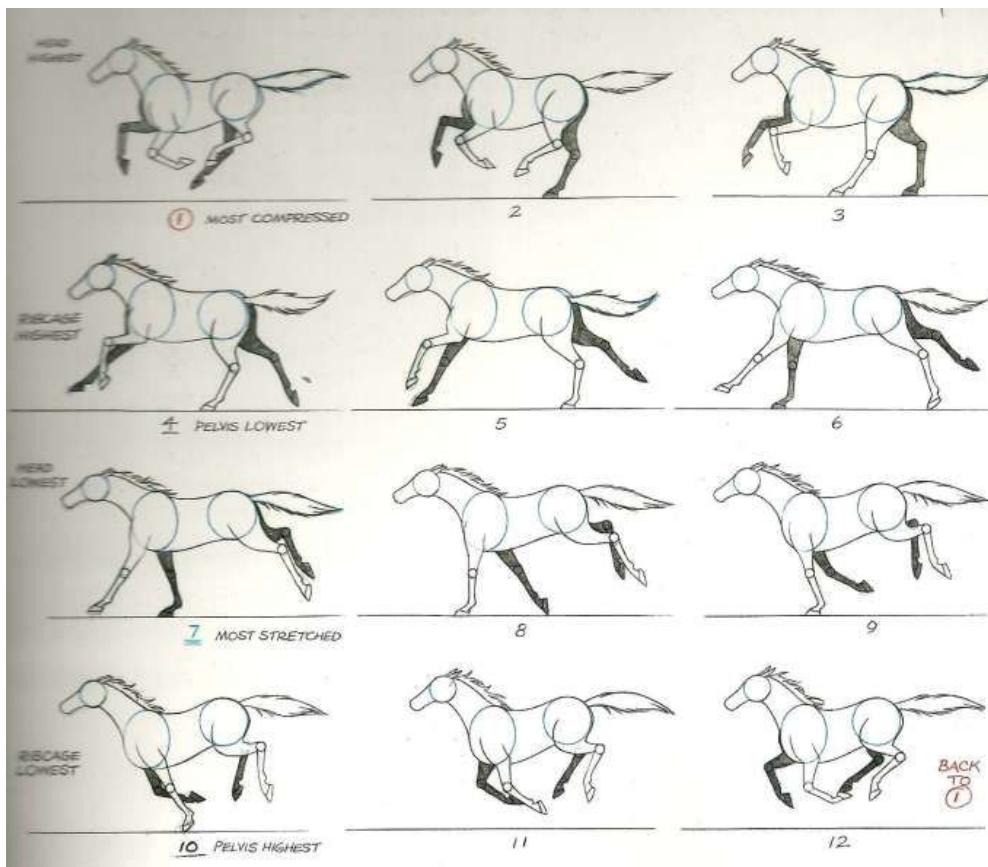


Figura 2.6: 12 frame disegnati di un cavallo al galoppo

L'analisi dei frame è stata usata la prima volta nel 1878 da Eadweard Muybridge proprio per studiare il movimento del cavallo al galoppo. Muybridge fu incaricato da Leland Stanford, ex-governatore della California, di trovare una prova che certificasse che i cavalli non toccano terra durante il galoppo. Per analizzare il movimento del

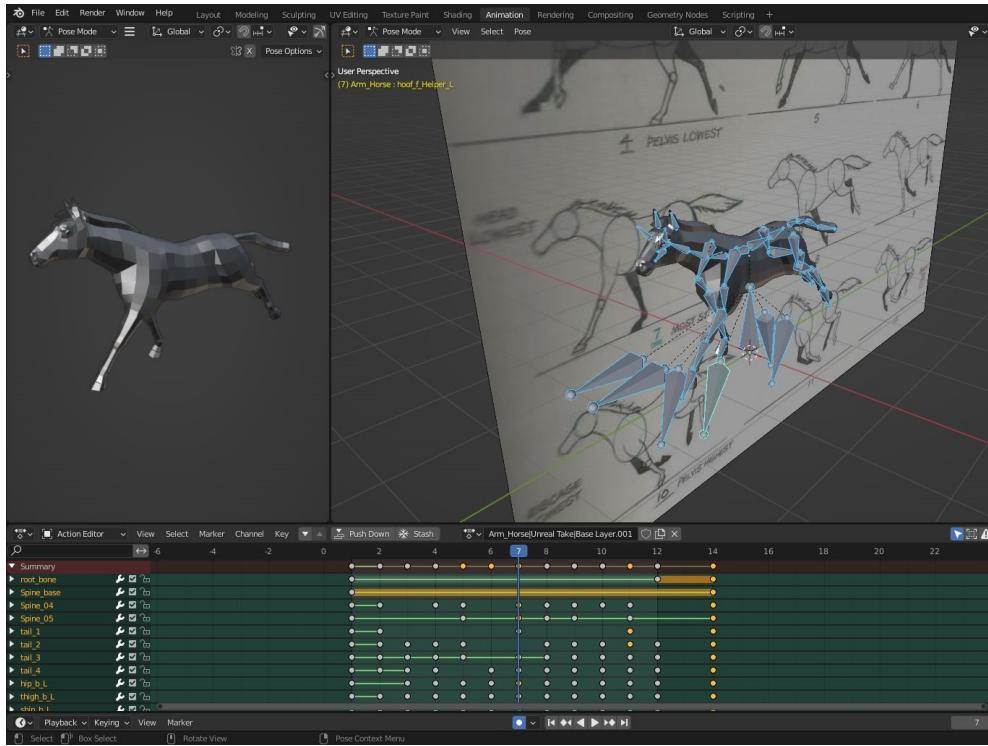


Figura 2.7: Schermata in cui è possibile vedere la Pose Mode di Blender, l’Action Editor con le keyframe, la mesh del cavallo in una pose con il suo scheletro e l’immagine utilizzata come reference.

cavalo in galoppo pensò allora di catturare in una sequenza di 16 foto un ciclo intero del movimento. Grazie a questa tecnica Muybridge dimostrò che il cavallo non tocca terra.

2.4 Progettazione del mondo 3D

Uno degli elementi più importanti quando si parla della costruzione di un mondo immersivo è la creazione del livello 3D. Per costruire il livello di gioco sono state utilizzate diverse tecniche offerte dal motore grafico Unreal Engine 5 ma sono stati utilizzati anche degli strumenti offerti dalla suite 3D Blender. Unreal Engine infatti permette di riempire il livello di oggetti con varie tecniche diverse ma questi oggetti devono inizialmente essere creati da un software di modellazione come Blender oppure importati da un asset del marketplace.

2.4.1 I modelli 3D

La suite Blender, oltre che per la creazione dell'animazione del galoppo, è stata usata per creare le mesh per questo progetto. È stata scelta questa suite per via della sua natura multipiattaforma e per la leggerezza di utilizzo. Questo software è stato utilizzato per creare delle mesh interamente e per modificare asset di gioco già esistenti.

Infatti, Blender permette di creare Mesh poligonali a partire da figure elementari, queste figure sono: il piano, il cubo, il cerchio, la UV sfera, la ICO sfera, il cilindro, il cono e l'anello. Inoltre permette di partire anche da due modelli più complessi: la griglia e la scimmietta Suzanne caratteristica di Blender. Questa scimmietta ha fatto la sua comparsa nel 2002, quando l'azienda fondata da Ton Roosendaal, la *Nan*, era ormai destinata alla bancarotta. Gli sviluppatori, in vista dell'inevitabile discontinuità del software, fecero uscire un aggiornamento poco prima della definitiva chiusura che, come piccolo personale tocco, includeva questo Easter Egg. Venne chiamata Suzanne come la scimmia del film *Jay and Silent Bob... Fermate Hollywood!* Da quel momento

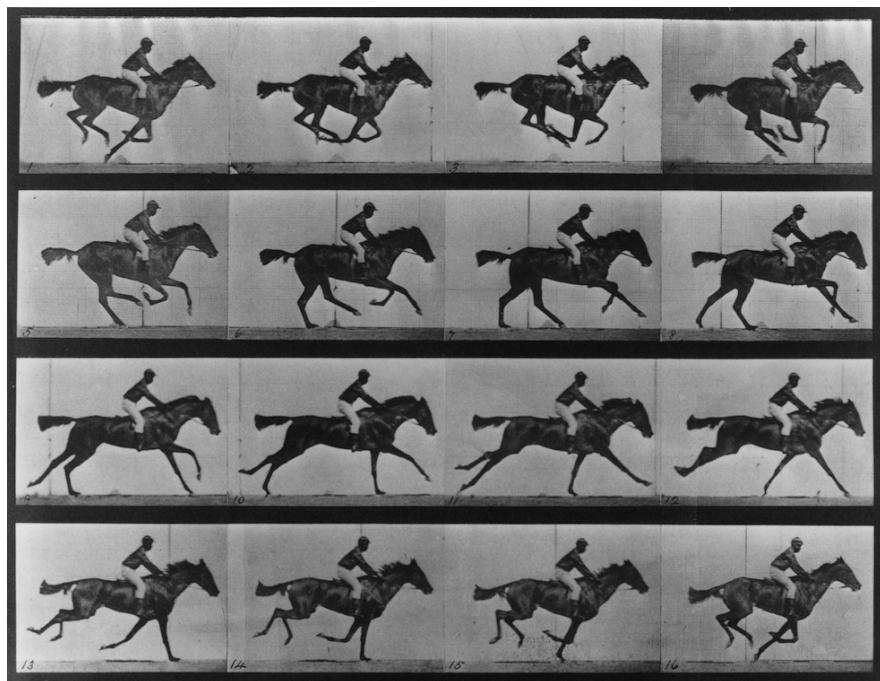


Figura 2.8: Sequenza di 16 scatti con cui Muybridge dimostrò che i cavalli non toccano terra durante il galoppo.

Suzanne divenne l'alternativa utilizzabile in Blender come modello di test, oltre ai più comuni Utah Teapot e Stanford Bunny, per provare materiali, texture ed altro.

Generalmente, le mesh sono costituite da tre elementi di base: vertici, facce e spigoli. Blender permette di modificare la mesh agendo direttamente su questi elementi utilizzando la Edit Mode. Le mesh create per Metarace con il software Blender sono state: il terreno di gioco, la griglia di start, gli spalti e altri oggetti di gioco come la stalla per i cavalli, una fontana e un campo ippico per la disciplina degli attacchi.

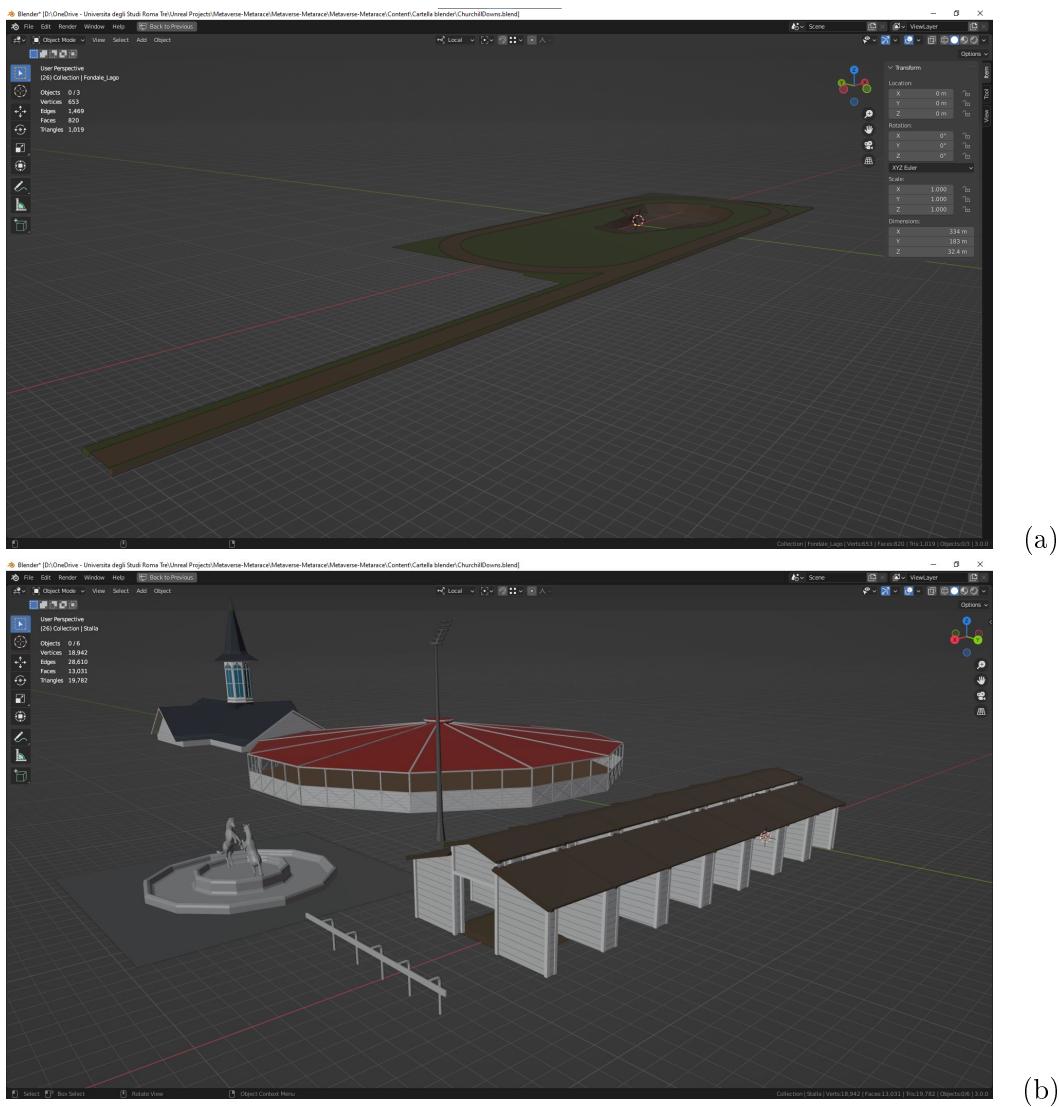


Figura 2.9: Terreno di gioco in Blender (a) Altri modelli in Blender. (b)

Queste mesh sono state create in modo da risultare composte da sole facce triangolari. Infatti un game engine può usare solo delle mesh con questo tipo di facce. È possibile comunque importare su Unreal Engine delle mesh con facce con più lati ma queste verranno automaticamente ricorrette in triangoli. Perciò per evitare distorsioni indesiderate ed errori nella conversione bisogna utilizzare sole facce triangolari oppure facce con al massimo 4 lati che non presentino angoli maggiori di 180°. Inoltre maggiore sarà il numero di vertici che una mesh possiede maggiore sarà il carico che il motore di gioco dovrà sopportare per la renderizzazione. Per questo motivo le mesh sono state create facendo in modo che il numero dei vertici fosse il più basso possibile.

Sono state inoltre utilizzate due tecniche di UV mapping (mappatura UV) per la creazione di texture del gioco. La mappatura UV è una tecnica che permette di proiettare una o più texture 2D su di un oggetto 3D. Le lettere U e V fanno riferimento ai due assi dell'immagine 2D e vengono usate al posto di X e Y in quanto queste ultime vengono già utilizzate con riferimento agli assi dell'ambiente 3D. Vista la natura low poly del gioco, la tecnica principale per la creazione di texture è stata quella di utilizzare una singola immagine per il maggior numero di elementi. È possibile infatti creare una texture estremamente leggera costituita da una palette di colori scelti, fare lo unwrap della texture e ridimensionare le UV islands delle facce condensandole in un unico punto

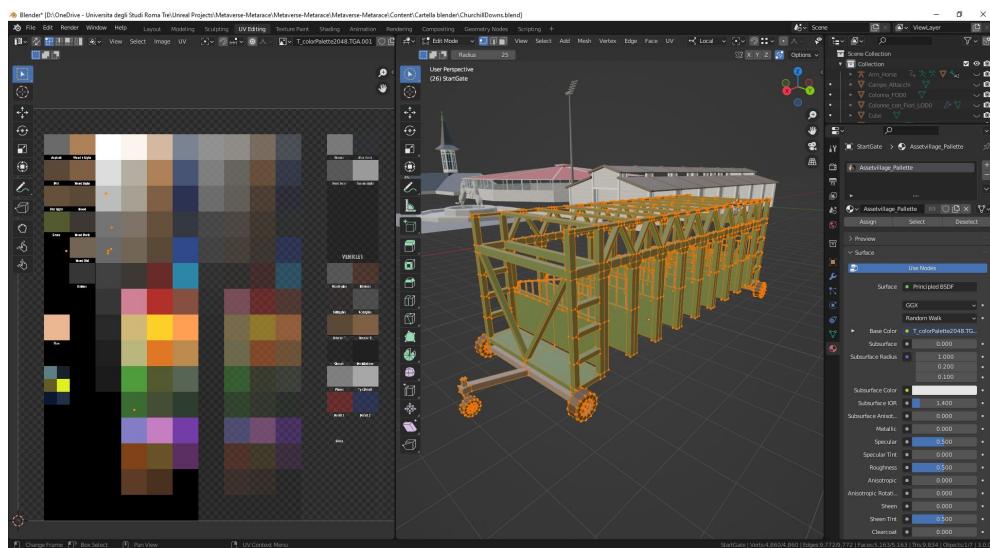


Figura 2.10: Esempio di associazione di UV islands a singoli colori di una sola texture

della texture in cui è presente il colore che andrà a riempire le facce corrispondenti.

Questo riduce il numero di istanze di materiali diversi che dovranno esserci nel gioco sia la complessità nel creare le texture stesse e il relativo UVMapping.

Questa tecnica è una semplificazione della più classica tecnica di mappatura UV dove le UV islands vengono direttamente posizionate sopra la texture in modo che ad ogni faccia corrisponda una particolare zona della texture. Il vantaggio di questa tecnica è che è possibile aggiungere della complessità visiva ad una faccia senza aggiungere complessità geometrica. Ad esempio per alcuni oggetti di gioco in cui le facce hanno dei pannelli di legno o dei mattoni quelle sono texture assegnate con la tecnica della mappatura UV classica.

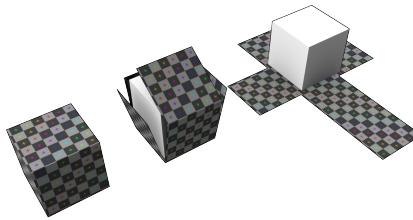


Figura 2.11: Schema di funzionamento dell'unwrapping

2.4.2 Strumenti forniti da Unreal Engine 5

Una volta create le mesh, queste sono state importate all'interno di Unreal Engine 5.

Una delle tecniche utilizzate per comporre il livello di gioco è stata quella di utilizzare oggetti modulari. Ossia oggetti composti da un insieme di mesh statiche che unite insieme possono creare combinazioni anche molto diverse tra loro. Questa tecnica ha il vantaggio di ridurre il numero di asset di gioco utilizzati senza perdere l'unicità degli ambienti che è possibile creare. Tutti gli edifici in scena sono stati creati utilizzando questa tecnica.

Inoltre Unreal Engine permette di aggiungere mesh di gioco in maniera rapida grazie ai Foliage Tools. Come detto in 1.2, è possibile accedere alla Foliage Mode tramite il menu delle modalità. Questo strumento permette di renderizzare Static Mesh o Actor

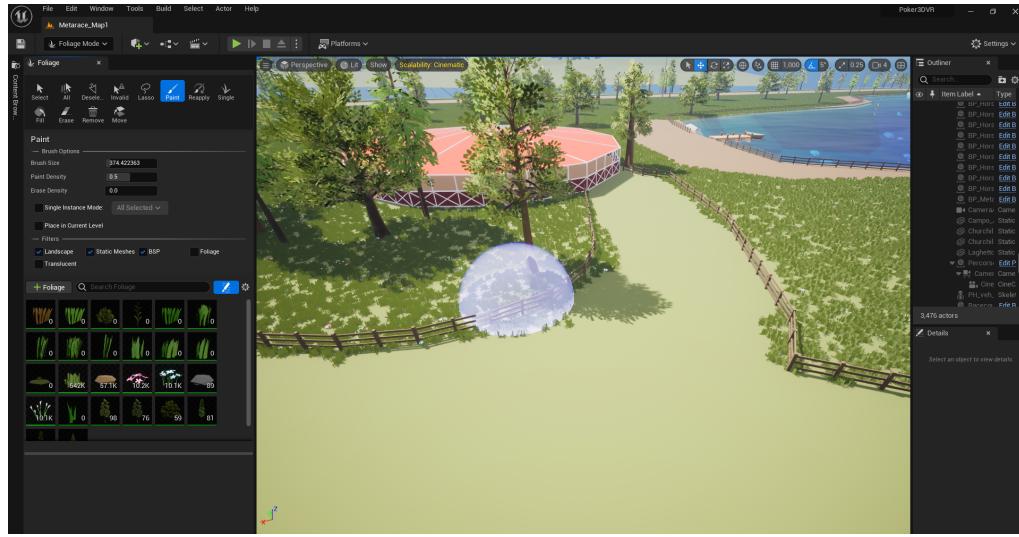


Figura 2.12: Utilizzo del Foliage Tools in Unreal Engine 5

Foliage sulla superficie di altre geometrie per usarle come effetto di copertura del terreno. Tutta la natura presente in Metarace è stata distribuita nel mondo di gioco con questo strumento.

Un’altro strumento utilizzato è stato il Landscape Tools per la generazione di terreno. Per creare scenari molto grandi è meglio utilizzare questa tecnica piuttosto che le



Figura 2.13: Schermata che riprende la vista dagli spalti del percorso

Static Mesh perché utilizza solo 4 bytes per i dati di ciascun vertice. Le Static Meshes utilizzano invece 12 byte per vertice, più i dati per le tangenti X e Z per ogni faccia racchiuse in 4 byte ciascuno più da 16 a 32 bit per i dati UV per un totale di 24 o 28 byte a vertice [7]. Inoltre questo strumento fornisce una serie di implementazioni atte ad aumentare l'ottimizzazione del rendering. Infatti il sistema di Landscape salva i dati di render del terreno nella memoria dedicata della scheda grafica sotto forma di texture, oltre che supportare il caricamento delle zone distanti del landscape in LOD (Level Of Details) via via meno complessi geometricamente.

Capitolo 3

Architettura del software

Gli elementi grafici e funzionali sono solo una parte dell’architettura del software di Metarace. Un’altra componente importante dell’architettura di Metarace è che sfrutta il paradigma della programmazione guidata dagli eventi. Nella programmazione guidata dagli eventi il flusso del programma è determinato da eventi esterni ad esso, come l’input di un utente o come un messaggio arrivato da un server. Infatti Metarace ha una struttura capace di rispondere agli eventi avviati dal server ma anche agli eventi innescati dall’utente. Il software presenta un’interfaccia utente che permette al giocatore di avviare una partita e, dato che esistono diverse tipologie di partite, di scegliere la tipologia di gara a cui partecipare. L’insieme degli eventi che possono verificarsi è stato scelto in fase di elaborazione ed è costituito da 13 eventi. Parte del mio lavoro è stato la progettazione e l’implementazione di 9 di questi, che sono: PlayEvent, AvailableRaceEvent, JoinEvent, PlayerJoinedEvent, StartingGridEvent, AnotherPlayerJoinedEvent, CountdownEvent, RaceEvent, LeaderboardEvent.

3.1 L’architettura di Metarace

Un videogioco è un applicativo che è particolarmente adatto alla divisione a strati. Infatti l’architettura di Metarace può essere divisa in vari strati che si occupano del suo funzionamento. È possibile individuare lo strato della Logica Implementativa, lo strato del Network, lo strato dell’Interfaccia Grafica e lo strato dell’Interfaccia Utente (UI).

Lo strato della Logica Implementativa è composto da tutti gli script che permettono il funzionamento dell'applicazione.

Lo strato del Network permette di comunicare da e verso il server e di tradurre queste comunicazioni in istruzioni per lo strato della Logica Implementativa.

Lo strato dell'Interfaccia Grafica è composto dall'engine di gioco e da tutti gli elementi che vengono renderizzati a schermo.

Lo strato della UI permette di catturare l'input dato dall'utente e di tradurlo in rispettivi eventi di gioco.

3.1.1 Progettazione guidata dalle responsabilità

Nella progettazione guidata dalle responsabilità gli oggetti software sono considerati come dotati di responsabilità, di ruoli e di collaborazioni. Sulla base del modello di dominio parziale mostrato in Figura 3.1 vengono assegnate le responsabilità di ciascun oggetto.

Le responsabilità sono di due tipi: *Responsabilità di fare* e *Responsabilità di conoscere*. Un importante tipo di responsabilità di fare è quella del pattern *Creator*. La maggior parte delle responsabilità di questo tipo sono state assegnate all'oggetto *Application Manager*. Questo è anche dovuto dal fatto che nel momento in cui il gioco viene fatto partire, un attore deve essere già in scena per poter instanziare tutti gli altri Actor. Questo attore è proprio l'*Application Manager*.

Un'altro importante pattern da seguire quando si assegnano le responsabilità è il pattern *Low Coupling*. Questo pattern punta a cercare il progetto che abbia il minor livello di accoppiamento. Per tenere l'accoppiamento basso in Metarace è stato sfruttato uno strumento offerto da Unreal Engine: i delegates.

I delegates permettono di chiamare funzioni di oggetti C++ in modo generico e type-safe [5]. Un delegate può essere legato dinamicamente ad una funzione di un oggetto scelto e permette di eseguire questa funzione in un momento futuro senza che il chiamante debba conoscere il tipo dell'oggetto chiamato. Questo strumento riduce di molto l'accoppiamento in quanto in questo modo due oggetti possono collaborare senza conoscersi.

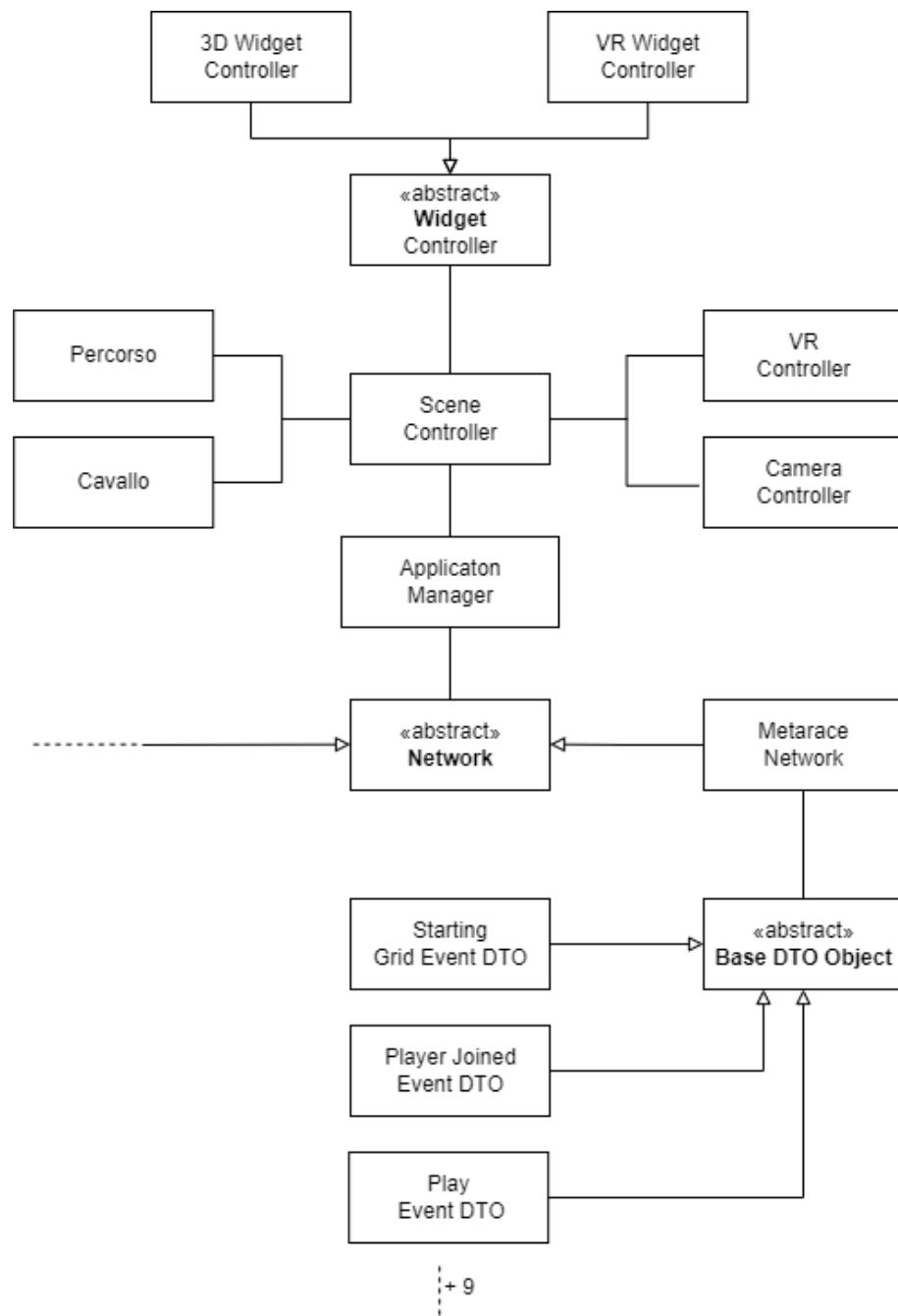


Figura 3.1: Modello di Dominio parziale per Metarace

3.1.2 L'Application Manager

L'Application Manager è l'unica classe già presente in scena durante la prima chiamata alla funzione *Begin Play*. In Unreal Engine è possibile posizionare una classe C++ nel mondo di gioco attraverso la creazione di un Blueprint sulla base di essa. Allo stesso modo all'interno di una classe C++ è possibile avere dei riferimenti ad altri Blueprint basati sempre su classi C++. Infatti, Unreal Engine permette di creare questo tipo di riferimenti attraverso il template di classe *TSubclassOf<T>*.

Algoritmo 3.1: Sezione dell'header file della classe ApplicationManager dove vengono referenziate altri Blueprint basati su classi C++

```

1 #include ...
2
3 UCLASS()
4 class Metarace AMetaraceApplicationManager : public AActor
5 {
6     [...] //Costruttore di classe, funzione BeginPlay e Tick
7
8 protected:
9     UPROPERTY(Category=TableController, EditAnywhere,
10     BlueprintReadWrite)
11     TSubclassOf SceneControllerBP;
12     UPROPERTY(Category=TableController, EditAnywhere,
13     BlueprintReadWrite)
14     TSubclassOf NetworkActorBP;
15
16 private:
17     UPROPERTY()
18     AMetaraceSceneController* SceneController = nullptr;
19     UPROPERTY()
20     AMetaraceNetworkActor* NetworkActor = nullptr;
21 };

```

Nella funzione *Begin Play* viene messo in pratica il pattern *Create* per gli oggetti SceneController e NetworkActor:

Algoritmo 3.2: Sezione del file source dell'Application Manager dove vengono creati gli oggetti SceneController e NetworkActor

```

1 void AMetaraceApplicationManager::BeginPlay()
2 {
3     Super::BeginPlay();
4
5     UWorld* World = GetWorld();
6     if (!World) { return; }
7
8     FActorSpawnParameters SpawnParams;
9     if (SceneControllerBP)
10    {
11         SceneController = World->SpawnActor

```

L'ApplicationManager si occupa anche di creare i Bind per tutti i delegate visto che già ha i riferimenti a tutti gli oggetti che li utilizzano.

Algoritmo 3.3: Sezione del file source dell'Application Manager dove viene formato il Bind dei delegates

```

22
23     if (SceneController && NetworkActor)
24    {
25         SceneController->OnWantToSendMessage .
26             BindUObject(NetworkActor,
27                         &MetaraceNetworkActor::SendMessage);
28         SceneController->WidgetController->OnWantToSendMessage .
29             BindUObject(NetworkActor, 
```

```

30         &AMetaraceNetworkActor::SendMessage);
31
32     NetworkActor->OnAvailableRacesFormatsEventDelegate.
33         BindUObject(SceneController,
34             &AMetaraceSceneController::ShowAvailableRaceFormats);
35     NetworkActor->OnPlayerJoinedEventDelegate.
36         BindUObject(SceneController,
37             &AMetaraceSceneController::PlayerJoined);
38     NetworkActor->OnStartingGridEventDelegate.
39         BindUObject(SceneController,
40             &AMetaraceSceneController::StartingGrid);
41     NetworkActor->OnRaceEventDelegate.
42         BindUObject(SceneController,
43             &AMetaraceSceneController::InitRace);
44     NetworkActor->OnCountdownEventDelegate.
45         BindUObject(SceneController,
46             &AMetaraceSceneController::ShowCountDown);
47     NetworkActor->OnAnotherPlayerJoinedEventDelegate.
48         BindUObject(SceneController,
49             &AMetaraceSceneController::AnotherPlayerJoined);
50     NetworkActor->OnLeaderboardEventDelegate.
51         BindUObject(SceneController,
52             &AMetaraceSceneController::OnLeaderboardEvent);
53     }
54 }
55

```

Lo SceneController possiede un delegate perché a gara finita deve poter inviare il file *Json* per l'evento *RaceFinischedEvent* verso il server. Il WidgetController anche possiede un delegate perché deve poter inviare messaggi verso il server dato che è lui che si occupa degli eventi *PlayEvent* e *JoinEvent*. NetworkActor possiede inoltre un delegate per ogni evento che deve gestire e viene creato il Bind con lo SceneController per delegare quest'ultimo a eseguire la logica implementativa.

Dopo l'esecuzione della funzione *Begin Play* l'ApplicationManager non viene più chiamato.

3.1.3 Lo *SceneController*

Lo *SceneController* è la classe implementata per permettere di interfacciarsi con lo strato di Logica Implementativa e con lo strato dell’Interfaccia Utente. A questa classe sono state assegnate le responsabilità di conoscere gli oggetti più importanti del livello di gioco: il percorso di gara e i cavalli. Infatti, si occupa dello *spawn* dei cavalli dei giocatori e del settaggio delle loro variabili. Inoltre si occupa della creazione delle classi WidgetController, CameraController e VRController. Lo SceneController crea l’istanza del WidgetController in base alla presenza o meno del VR connesso al computer. Questo può essere controllato grazie alla libreria offerta da Unreal Engine *UHeadMountedDisplayFunctionLibrary*.



Figura 3.2: Vista dall’inizio del percorso, dopo che lo SceneController ha fatto comparire i cavalli

Algoritmo 3.4: Sezione del source della classe SceneController dove viene fatto il controllo per sapere se il giocatore indossa un visore

```

1 void AMetaraceSceneController::BeginPlay()
2 {
3     Super::BeginPlay();
4     UWORLD* World = GetWorld();
5     if (!World) { return; }
6

```

```
7     IsInVR = UHeadMountedDisplayFunctionLibrary::  
8     IsHeadMountedDisplayConnected();  
9  
10    if (IsInVR)  
11    {  
12        if (BP_VRController)  
13        {  
14            VRController = World->SpawnActor<AVRMetaraceController>(BP_VRController, GetTransform(), SpawnParams);  
15        }  
16        if (BP_VRWidgetController)  
17        {  
18            WidgetController = World->SpawnActor<AMetaraceVRWidgetController>(BP_VRWidgetController, GetTransform(), SpawnParams);  
19        }  
20    }  
21    }  
22    if (BP_CameraController)  
23    {  
24        CameraController = World->SpawnActor<AMetaraceCameraController>(BP_CameraController, GetTransform(), SpawnParams);  
25    }  
26    if (BP_3DWidgetController)  
27    {  
28        WidgetController = World->SpawnActor<AMetaraceWidgetController>(BP_3DWidgetController, GetTransform(), SpawnParams);  
29    }  
30    [...]  
31    }  
32    }  
33    }  
34 }
```

Lo SceneController implementa tutte le funzioni che sono collegate con i delegate del NetworkActor.

3.1.4 Il *WidgetController*

Il *WidgetController* è la classe implementata per permettere di gestire la creazione, l'interazione e la visualizzazione dei Widget. I Widget sono delle strutture offerte da Unreal Engine con cui è possibile creare e far comparire l'Interfaccia Utente. Questi oggetti sono il punto di accesso al sistema offerto al giocatore. Infatti, permettono di mostrare testi e pulsanti (più in generale interfacce UI) con cui il giocatore può interagire.

La classe *WidgetController* è astratta perché esistono due implementazioni in base a se il giocatore è in VR oppure no. Infatti i Widget hanno un comportamento molto diverso nei due casi. Se il giocatore non indossa un visore VR i widget saranno soltanto degli oggetti 2D che verranno mostrati a schermo. Se invece il giocatore indossa un visore VR i widget dovranno essere istanziati come oggetti 3D nel mondo di gioco. In entrambi i casi le funzionalità dei singoli widget dovranno essere mantenute.

Ho creato i Widget partendo dalla classe C++. Infatti può essere istanziato sulla base di una classe che estende la classe di Unreal Engine *UUserWidget*. Sulla base di questa classe va creato un Blueprint di tipo Widget (un particolare tipo di Blueprint) in cui inserire la classe come istanza di Widget. È possibile definire degli elementi del Widget (testuali o interagibili come dei bottoni) all'interno del codice e collegarli con le variabili nell'istanza Blueprint con lo specificatore **UPROPERTY** con l'opzione *meta = (BindWidget)*. Questa specificazione crea un bind automatico con la variabile all'interno del Blueprint ma quest'ultima deve avere necessariamente lo stesso nome della corrispettiva nello script (in questo caso *StartButton*). È mostrato il codice relativo alla classe che implementa il Widget per il pulsante *Start*, chiamata *StartMenuWidget*:

Algoritmo 3.5: File header della classe *StartMenuWidget*

```

1 #include ...
2
3 DECLARE_DYNAMIC_MULTICAST_DELEGATE_OneParam(
4     FDelegateOnClickStartRaceEventUI , bool , toShow);
5
6 UCLASS()
7 class Metarace UStartMenuWidget : public UUserWidget
8 {
9 }
```

```

8  GENERATED_BODY()
9
10 public:
11     UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (BindWidget))
12     class UButton* StartButton;
13
14     UPROPERTY()
15     class AMetaraceBaseWidgetController* WidgetController = nullptr;
16
17     virtual void NativeConstruct() override;
18
19     UFUNCTION()
20     void OnStartClicked();
21
22     FDelegateOnClickStartRaceEventUI Delegate_OnClickStartRaceEventUI;
23
24 };
25

```

Per definire la funzione da chiamare quando un bottone viene premuto dall'utente bisogna fare creare un legame con la funzione scelta (che ho chiamato in questa classe *onStartClicked*). Questo si può chiamare con la funzione *AddUniqueDynamic* come mostrato di seguito.

Nella funzione *OnStartClicked* viene mostrato l'esecuzione del delegate che si occupa di inviare il file Json al server.

Algoritmo 3.6: File source classe StartMenuWidget

```

1 #include ...
2
3 void UStartMenuWidget::NativeConstruct()
4 {
5     Super::NativeConstruct();
6
7     StartButton->OnClicked.AddUniqueDynamic(this,
8                                         &UStartMenuWidget::onStartClicked);
9 }
10
11 void UStartMenuWidget::onStartClicked()

```

```

12 {
13     if(WidgetController &&
14         WidgetController->OnWantToSendMessage.IsBound())
15     {
16         FJsonObject o;
17         o.SetStringField("event", "Play");
18         WidgetController->OnWantToSendMessage.
19             Execute(Utility::JsonToString(o));
20     }
21 }
22

```

La classe *BaseWidgetController* definisce le funzioni seguenti:

Algoritmo 3.7: File header della classe astratta BaseWidgetController

```

1 #include ...
2 DECLARE_DELEGATE_OneParam(FOnWantToSendMessage, const FString&);
3
4 UCLASS()
5 class Metarace AMetaraceBaseWidgetController : public AActor
6 {
7     GENERATED_BODY()
8     [...] //Constructor, BeginPlay e Tick Function
9 public:
10     virtual void ShowStartUI() {};
11     virtual void ShowRaceFormats(TArray<URaceFormatObject*> Formats)
12     {};
13     virtual void RemoveRaceFormats() {};
14     virtual void UpdateTimer() {};
15     virtual void ShowCountdown(int Seconds) {};
16     virtual void ShowRaceWidget() {};
17     virtual void ShowLeaderboard(TArray<ULeaderboardObjectDTO*>
18         LeaderboardLines) {};
19
20     FOnWantToSendMessage OnWantToSendMessage;
21 };
22

```



Figura 3.3: Widget renderizzato in 3D per la modalità VR

Nel caso in cui il giocatore non stia collegato con un visore si possono far comparire i Widget come delle immagini 2D a schermo. Ho perciò utilizzato le funzioni seguenti per questo scopo:

Algoritmo 3.8: Crezione di istanza di Widget con StartMenuWidget come esempio

```

1 StartMenuWidget = CreateWidget<UStartMenuWidget>(GetWorld(),
    BP_StartWidget);
2 StartMenuWidget->WidgetController = this;
3

```

Algoritmo 3.9: Aggiunta di Widget al Viewport

```

4 StartMenuWidget->AddToViewport(0);
5

```

Mentre nel caso in cui il giocatore stia collegato con un visore per la realtà virtuale i Widget sono stati inseriti come Widget Component all'interno di un Blueprint comune. In questo modo è possibile renderizzarli come delle immagini all'interno dell'ambiente 3D di gioco, come mostrato in figura 3.3.

Per questo scopo si utilizzano le funzioni per lo spawn degli Actor e si cerca successivamente il Widget Component al loro interno in modo da ottenere la classe Widget con le funzionalità cercate. È possibile interagire con il Widget mentre si è in modalità VR puntando il joystick verso il pulsante e premendo il tasto corrispondente

all’interazione. Il codice seguente mostra l’utilizzo di queste funzioni per il VRWidget *StartMenuWidget*:

Algoritmo 3.10: Creazione Widget VR per la classe *StartMenuWidget*

```

1 VRStartMenu = World->SpawnActor<AActor>(BP_VRStartMenu,
    SpawnTransform, SpawnParams);
2 UWidgetComponent* WidgetComponent = VRStartMenu->FindComponentByClass
    <UWidgetComponent>();
3 StartMenuWidget = Cast<UStartMenuWidget>(WidgetComponent->GetWidget()
    );
4 StartMenuWidget->WidgetController = this;
5

```

È possibile infine mostrare o nascondere il Blueprint desiderato attraverso la libreria *ESplate Visibility* che permette di scegliere tra *Visible*, *Hidden* e altre opzioni.

Degno di nota è infine lo strumento *ULListView* che permette di inserire a tempo di esecuzione un numero variabile di Widget all’interno di un altro Widget, potendo perciò creare dei Widget a cascata anche a tempo di esecuzione. Equivalente è lo strumento *UTile View*. I due strumenti sono stati utilizzati rispettivamente per renderizzare il Widget per la classifica finale e per renderizzare la griglia di pulsanti con cui è possibile



Figura 3.4: Foto con visore VR durante una partita di Metarace

scegliere il formato di gara a cui partecipare.

3.1.5 Il CameraController

Il *CameraController* è la classe implementata per permettere di gestire il movimento delle camere e di cambiare visione da una camera ad un'altra in base alle esigenze.

Sebbene potrebbe presentare funzioni simili al VRController, queste due classi non sono state pensate per avere le stesse funzioni perché il CameraController non è un sostituto dell'implementazione VR. Infatti le telecamere non servono unicamente per renderizzare il mondo di gioco al vista del giocatore, ma possono renderizzare il mondo di gioco anche come texture di oggetti all'interno del mondo stesso. L'implementazione di questa classe quindi può essere scalata per utilizzare l'output delle telecamere per simulare una televisione o un altro dispositivo digitale per far vedere al giocatore immerso nel mondo di gioco un canale televisivo con in onda la gara stessa.



Figura 3.5: CineCameraActor che riprendere la griglia di partenza

Per le camere di gioco sono state scelte le *CineCameraActor* perché permettono di tenere al centro dell'inquadratura un attore nel mondo di gioco. Questa caratteristica è particolarmente utile per ricreare l'effetto di una camera ferma in un punto nello spazio che gira su se stessa per inquadrare il soggetto, come accade nelle trasmissioni televisive che riprendono le competizioni. È stato usato anche un oggetto *CameraRigRail* per ottenere un binario cinematografico per ricreare l'effetto di una telecamera che segue

i cavalli. Questo strumento è composto da una Spline e da un modello 3D per far visualizzare allo sviluppatore la posizione della camera lungo di essa. La CameraRigRail si è dimostrata particolarmente comoda in questo contesto perché visto che i cavalli si muovono lungo una Spline si è potuto impostare il carrello della stessa lunghezza del percorso. A quel punto è stato possibile passare il valore della distanza a cui si trova il cavallo lungo la spline al carrello per posizionare la camera alla stessa altezza dei cavalli. La position lungo il rail è standardizzata tra 0 e 1 perciò basta passare il valore di output della Timeline, per questo scopo è stata usata la funzione seguente:

```

1 void AMetaraceCameraController::UpdateCameraPositionOnSpline (float
    TimelineValue)
2 {
3     CameraRail->CurrentpPositionOnRail = TimelineValue;
4 }
```

All'interno del percorso sono stati inseriti degli oggetti *ATriggerBox* per poter cambiare telecamere al raggiungimento di questi trigger, nella Figura 3.5 ne è mostrato uno. Gli oggetti *ATriggerBox* permettono di impostare degli eventi *Overlap* che permettono di cambiare camera quando i cavalli raggiungono un punto lungo il percorso.

Algoritmo 3.11: Chiamata per l'aggiunta di una funzione all'evento overlap di un *TriggerBox*

```

1 StartRaceTriggerBox->OnActorEndOverlap.AddDynamic (this ,
2     &AMetaraceCameraController::OnStartTriggerExit);
3 }
```

È possibile passare da una camera ad un'altra chiamando la funzione *SetViewTargetWithBlend* che offre la possibilità di avere un movimento fluido per il cambio da un'inquadratura ad un'altra su una base di un valore temporale passato in input.

Algoritmo 3.12: Funzione per impostare la telecamera per la vista dell'utente

```

1 void AMetaraceCameraController::OnStartTriggerExit (AActor* This ,
    AActor* Other)
2 {
3     if (CineCamera)
4     {
```

```

5     PlayerController->SetViewTargetWithBlend(
6         CineCamera, 3);
7 }
8 }
9

```

3.1.6 Il *VRController*

La classe *VRController* si occupa di implementare le chiamate dello *SceneController* quando il giocatore ha eseguito l'accesso al gioco tramite un visore per la realtà virtuale.

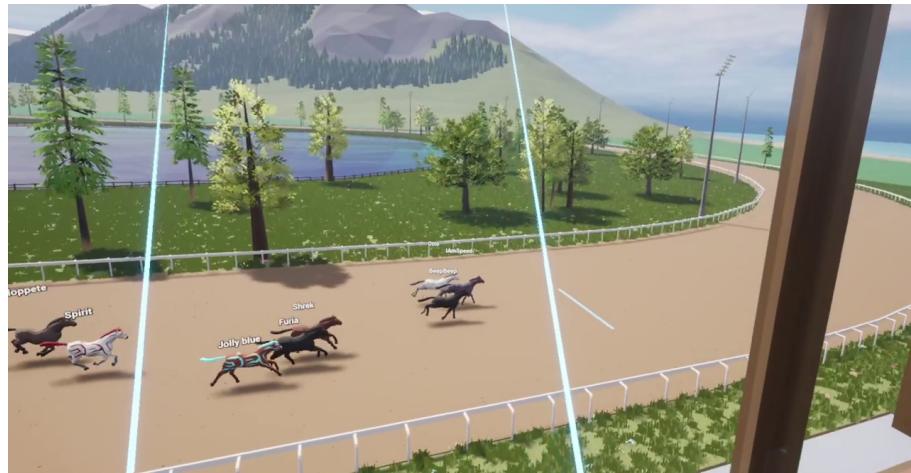


Figura 3.6: Visuale della gara in VR da una posizione appositamente creata

Il giocatore in questa modalità oltre che poter interagire con i widget può assistere alla gara dagli spalti e da posizioni lungo il percorso appositamente create per lo scopo. Ad esempio ho creato una sala interna di un edificio che affaccia sul percorso da dove è possibile avere una visuale della gara.

Questa classe possiede un *TArray* di posizioni che implementano una logica di ordinazione. Le posizioni sono ordinate e posizionate lungo il percorso che fanno i cavalli in modo che il giocatore può scorrerle avanti e indietro. Le funzioni che implementano questa logica sono:

Algoritmo 3.13: Funzioni per teleportare il giocatore lungo le posizioni per assistere alla gara

```

1 void AVRMetaraceController::NextPosition()
2 {
3     int32 temp = CurrentPosition+1;
4     if(!PlayerPositions.IsEmpty() && temp<PlayerPositions.Num() &&
5         temp>=0)
6     {
7         if(Player)
8             Player->TeleportTo(
9                 PlayerPositions[temp]->GetActorLocation(),
10                PlayerPositions[temp]->GetActorRotation());
11         CurrentPosition = temp;
12     }
13 }
14 }
15
16 void AVRMetaraceController::PrevPosition()
17 {
18     int32 temp = CurrentPosition-1;
19     if(!PlayerPositions.IsEmpty() && temp<PlayerPositions.Num() &&
20         temp>=0)
21     {
22         if(Player)
23             Player->TeleportTo(PlayerPositions[temp]->
24             GetActorLocation(),
25             PlayerPositions[temp]->GetActorRotation());
26         CurrentPosition = temp;
27     }
28 }
29

```

Capitolo 4

Networking

Una parte importante dell'architettura di Metarace è la sua architettura client-server. Infatti è questa che gli permette di interfacciarsi con il server e di scambiare tutti i messaggi che guideranno il flusso del programma.

4.1 Architettura Client - Server e WebSocket

Metarace si collega al server con un canale di comunicazione bidirezionale basato sul protocollo WebSocket. Questo protocollo è stato scelto per due motivi: il primo è che permette di tenere aperta una connessione tra client e server fino a che non viene terminata da uno dei due partecipanti; il secondo è che non è necessario definire il protocollo di comunicazione dato che si basa sul diffuso protocollo TCP.

Questo protocollo trasferisce i dati su una connessione full-duplex su singolo socket e permette l'invio e la ricezione di messaggi da parte di ambedue gli endpoint. Una connessione WebSocket è instaurata con un handshake specifico basato sul protocollo HTTP. A seguito di questo handshake il protocollo di comunicazione cambia, e passa da HTTP a WebSocket tramite la connessione TCP utilizzata inizialmente.

Unreal Engine 5 fornisce supporto per l'utilizzo dei WebSocket tramite il modulo *IWebSocket* [19]. Per poter utilizzare questo modulo bisogna aggiungere la dependency all'interno del file ".build.cs" nel quale vengono specificate le dipendenze a componenti e plugin esterni all'engine di gioco permettendo la compilazione del progetto. Qui è mostrato il codice, in Algoritmo 4.1:

Algoritmo 4.1: Metarace.build.cs file

```

1 using UnrealBuildTool;
2
3 public class Metarace : ModuleRules
4 {
5     public Metarace(ReadOnlyTargetRules Target) : base(Target)
6     {
7         PublicDependencyModuleNames.AddRange(new []
8             { "Core", "CoreUObject", "Engine",
9                 "InputCore", "WebSockets", "Json",
10                "UMG", "HeadMountedDisplay", "CinematicCamera",
11                "Niagara" });
12    }
13 }

```

La WebSocket viene creata all'interno di una classe astratta, chiamata *BaseNetworkActor*. Il motivo di questa scelta deriva dal fatto che Metarace non è l'unico applicativo dell'universo dell'azienda Ringmaster. Con questa struttura è infatti possibile creare implementazioni diverse per ogni applicativo senza dover cambiare ogni volta il tipo di classe utilizzato. Una classe astratta non viene instanziata ma viene ereditata da sottoclassi che ne implementano o ne sovrascrivono i metodi. La scelta di una classe astratta invece di una interfaccia è dovuta dal fatto che ci sono implementazioni che devono essere uguali tra tutte le sottoclassi (l'intaurazione e il rilascio della connessione ad esempio).

Qui viene mostrato il codice di questa classe:

Algoritmo 4.2: Sezione del file header di BaseNetworkActor dove vengono importati i moduli necessari alla WebSocket e viene definita la mappa degli eventi

```

1 #include "WebSocketsModule.h" // Module definition
2 #include "IWebSocket.h" // Socket definition
3
4 UCLASS()
5 class Metarace : ABaseNetworkActor : public AActor
6 {
7     GENERATED_BODY()
8 protected:
9     TMap<int32, std::function<void(ABaseNetworkActor*, FJsonObject)>>

```

```
10     mEventMap;
```

Viene definita una *TMap* per mappare tutte le funzioni che gestiscono gli eventi ricevuti dal server. Questa mappa ha come key degli interi e come value delle funzioni di sola lettura, definite *const*, che prendono come parametro un riferimento ad un'istanza della classe *BaseNetworkActor* o di una sua sottoclasse e un *JsonObject*. La comunicazione si baserà su messaggi di tipo *Json*, in Unreal Engine esistono gli oggetti di tipo *JsonObject* e definiti come *FJsonObject*. La mappa è un campo protetto perciò è stata implementata una funzione che esegue il *put*, definita come segue:

Algoritmo 4.3: Sezione del file header di *BaseNetworkActor* dove viene definita la funzione che esegue il put nella mappa degli eventi

```
11 public:
12     virtual void observeEvent(const int32 EventType,
13         const std::function<void(ABaseNetworkActor*, 
14             const FJsonObject&)>& iCallback);
15     [...]
16 }
```

Per creare la *WebSocket* chiamo la funzione *CreateWebSocket* dal modulo:

Algoritmo 4.4: Sezione del file source di *ABaseNetworkActor* dove viene creata la *WebSocket*

```
1 void ABaseNetworkActor::Connect(const FString& Endpoint)
2 {
3     const FString& WebsocketEndpoint = Endpoint;
4     const FString& Protocol = TEXT("json");
5
6     MSocket = FWebSocketsModule::Get().CreateWebSocket(
7         WebsocketEndpoint, Protocol);
```

Definisco quindi tramite delle *Lambda* gli eventi: *OnConnected*, *OnMessage*, *OnClosed* e *OnConnectionError*.

```
7     MSocket->OnConnected().AddLambda(
8         [=]() -> void
9     {
10         OnSocketConnected();
11     }
```

```
12     );
```

Il file *Json* arriva sotto forma di stringa, perciò va inizializzato un *TJsonReader* che la legge come file *Json* e permette quindi di trasformarlo in un *FJsonObject*. Si noti che in queste implementazioni viene fatto uso di un oggetto della libreria *Unreal Smart Pointer Library*, il puntatore smart *TSharedPtr*. Questa libreria è un'implementazione di Unreal in C++11 degli *Smart Pointer* ideata per alleggerire il peso dell'allocazione e del tracciamento della memoria [9]. Uno *Shared Pointer* possiede l'oggetto a cui fa riferimento e gestisce la cancellazione dello stesso impedendola indefinitivamente oppure cancellandolo quando nessun puntatore vi fa più riferimento.

Algoritmo 4.5: La funzione che gestisce i messaggi in entrata alla WebSocket

```
13     MSocket->OnMessage().AddLambda(
14         [=](const FString& iMessage) -> void
15     {
16         const TSharedRef<TJsonReader<>> JSONReader = TJsonReaderFactory
17             <>::Create(iMessage);
18
19         if (TSharedPtr<FJsonObject> JSONObject; FJsonSerializer::
20             Deserialize(JSONReader, JSONObject))
21         {
22             OnMessageReceived(JSONObject);
23         }
24     });
25 }
```

Seguono quindi le *Lambda Function* per la gestione degli eventi rimanenti:

Algoritmo 4.6: Lambda function per la gestione della chiusura della connessione e dell'errore durante la connessione

```
24     MSocket->OnClosed().AddLambda(
25         [=](int32 iStatusCode, const FString& iReason, bool iWasClean) ->
26             void
27         {
28             UE_LOG(LogTemp, Display, TEXT("AHackathonBaseNetworkActor::
29                 BeginPlay::OnClosed message = %d %s %d"),
30                 iStatusCode, *iReason, iWasClean);
31         }
32     );
33 }
```

```

30 );
31 MSocket->OnConnectionError().AddLambda(
32 [=](const FString& iError) -> void
33 {
34     UE_LOG(LogTemp, Display, TEXT("AHackathonBaseNetworkActor::"
35     "BeginPlay::OnConnectionError error = %s"),
36     *iError);
37 }
38 );

```

E solo a questo punto si può connettere la WebSocket al server tramite la funzione *Connect*.

Algoritmo 4.7: Connessione della WebSocket

```

38 MSocket->Connect();
39 }

```

La funzione che legge il file *Json* e che cerca nella *TMap* la funzione corrispondente cerca all'interno del file un campo "event" dove viene specificato l'intero corrispondente al value della funzione del client che il server vuole raggiungere:

Algoritmo 4.8: funzione che gestisce la ricezione del messaggio in ABaseNetworkActor

```

1 void ABaseNetworkActor::OnMessageReceived(const TSharedPtr<
2     FJsonObject>& Message)
3 {
4     if (Message.IsValid())
5     {
6         const FJsonObject& JSONObject = *(Message.Get());
7         if (const int32 EventType =
8             JSONObject.GetIntegerField("event");
9             mEventMap.Contains(EventType))
10        {
11            mEventMap[EventType](this, JSONObject);
12        }
13    }

```

Infine, la funzione per inviare dati al server è definita come segue:

Algoritmo 4.9: Funzione per inviare dati al server attraverso la WebSocket

```

1 void ABaseNetworkActor::SendMessage(const FString& Message) const
2 {
3     if (MSocket->IsConnected())
4     {
5         MSocket->Send(Message);
6     }
7 }
```

4.1.1 Implementazione MetaraceNetworkActor

Dopo aver definito la classe astratta per la creazione e gestione della *WebSocket* si passa a definire la sottoclasse che ne eredita le funzionalità: la classe *MetaraceNetworkActor*.

Questa classe si occupa di mappare i messaggi che arrivano dal server.

I delegates visti in precedenza (in 3.3) corrispondono a tutti gli eventi che *MetaraceNetworkActor* deve gestire, vengono definiti nel file header nel modo che segue (in 4.1.1). Per ogni messaggio in ingresso è stato inoltre creato un oggetto DTO per ospitare i dati contenuti nel Json dell'evento in questione, ed è proprio di questo tipo l'oggetto che il delegate passerà quando verrà eseguito:

Algoritmo 4.10: Dichiarazione delegate nel file header di MetaraceNetworkActor

```

1 #include ...
2
3 DECLARE_DELEGATE_OneParam(FOnAvailableRacesFormatsEvent, const
4                             UAvailableRacesFormatsEventDTO* );
5 DECLARE_DELEGATE_OneParam(FOnPlayerJoinedEvent, const
6                           UPlayerJoinedEventDTO* );
7 DECLARE_DELEGATE_OneParam(FOnStartingGridEvent, const
8                           UStartingGridEventDTO* );
9 DECLARE_DELEGATE_OneParam(FOnAnotherPlayerJoinedEvent, const
10                            UAnotherPlayerJoinedEventDTO* );
11 DECLARE_DELEGATE_OneParam(FOnCountdownEvent, const UCountdownEventDTO
12                           * );
13 DECLARE_DELEGATE_OneParam(FOnRaceEvent, const URaceEventDTO* );
14 DECLARE_DELEGATE_OneParam(FOnLeaderboardEvent, const
15                            ULleaderboardEventDTO* );
```

```

11 UCLASS()
12 class Metarace AMetaraceNetworkActor : public ABaseNetworkActor
13 {
14     GENERATED_BODY()
15
16 public:
17     FOnLoginRequiredEvent OnLoginRequiredEventDelegate;
18     FOnMetaraceConnectedEvent OnMetaraceConnectedEventDelegate;
19     FOnAvailableRacesFormatsEvent
20         OnAvailableRacesFormatsEventDelegate;
21     FOnPlayerJoinedEvent OnPlayerJoinedEventDelegate;
22     FOnStartingGridEvent OnStartingGridEventDelegate;
23     FOnAnotherPlayerJoinedEvent OnAnotherPlayerJoinedEventDelegate;
24     FOnCountdownEvent OnCountdownEventDelegate;
25     FOnRaceEvent OnRaceEventDelegate;
26     FOnLeaderboardEvent OnLeaderboardEventDelegate;
27
28     [...]
29 }

```

Inoltre tutto lo spazio di eventi viene inserito in un apposito namespace in modo da non dover ricordare il valore numerico associato all'evento:

Algoritmo 4.11: Namespace in cui vengono definite le keys

```

1 namespace Events
2 {
3     static constexpr int32 GMetaraceConnected_Event = 0;
4     static constexpr int32 GMetaraceLogin_Event = 1;
5     static constexpr int32 GPlay_Event = 2;
6     static constexpr int32 GAvailableRaceFormats_Event = 3;
7     static constexpr int32 GJoin_Event = 4;
8     static constexpr int32 GPlayerJoined_Event = 5;
9     static constexpr int32 GStartingGrid_Event = 6;
10    static constexpr int32 GAnotherPlayerJoined_Event = 7;
11    static constexpr int32 GCountDown_Event = 8;
12    static constexpr int32 GRace_Event = 9;
13    static constexpr int32 GRaceFinishedEvent = 10;
14    static constexpr int32 GLeaderboard_Event = 11;
15 }

```

16

Infine la funzione per popolare la mappa degli eventi:

Algoritmo 4.12: Funzione Initialize nel NetworkActor

```

1 void AMetaraceNetworkActor::Initialize()
2 {
3     observeEvent(Events::GAvailableRaceFormats_Event, CALLBACK(&
4         AMetaraceNetworkActor::OnAvailableRacesFormatsEvent));
5     observeEvent(Events::GPlayerJoined_Event, CALLBACK(&
6         AMetaraceNetworkActor::OnPlayerJoinedEvent));
7     observeEvent(Events::GAnotherPlayerJoined_Event, CALLBACK(&
8         AMetaraceNetworkActor::OnAnotherPlayerJoinedEvent));
9     observeEvent(Events::GStartingGrid_Event, CALLBACK(&
10        AMetaraceNetworkActor::OnStartingGridEvent));
11    observeEvent(Events::GCountDown_Event, CALLBACK(&
12        AMetaraceNetworkActor::OnCountdownEvent));
13    observeEvent(Events::GRace_Event, CALLBACK(&AMetaraceNetworkActor::
14        OnRaceEvent));
15    observeEvent(Events::GLeaderboard_Event, CALLBACK(&
16        AMetaraceNetworkActor::OnLeaderboardEvent));
17 }
```

4.2 Programmazione ad Eventi (EDP)

Verranno discussi nel dettaglio tutti gli eventi che sono stati implementati.

4.2.0.1 PlayEvent e AvailableRacesFormatsEvent

L'evento *PlayEvent* viene innescato quando il giocatore preme il pulsante *Start*, come visto in Algoritmo: 3.6. Il delegate in questione è legato alla funzione vista in Algoritmo: 4.9. Tramite l'esecuzione del delegate, il client invia al server il messaggio relativo all'intenzione del giocatore di voler iniziare una partita. Il server invia in risposta un messaggio verso il client con tutte le tipologie di gare a cui è possibile accedere in quel momento.

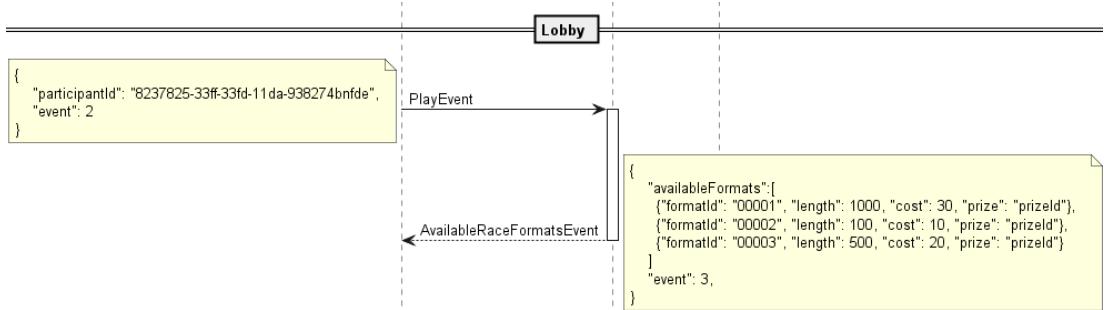


Figura 4.1: Diagramma di sequenza per gli eventi GetAvailableRacesEvent e OnAvailableRacesFormatsEvent

Il Json in arrivo dal server, come si vede nel diagramma di sequenza nella Figura: 4.1, avrà come NumberField("event") il valore 3. Questo valore permette di trovare nella mappa degli eventi con la funzione vista in Algoritmo:4.8 la funzione *OnAvailableRaceFormatsEvent*.

Algoritmo 4.13: Funzione OnAvailableRaceFormatsEvent

```

1 void AMetaraceNetworkActor::OnAvailableRaceFormatsEvent(
2     AMetaraceNetworkActor* NetworkActor, const FJsonObject& JsonObject
3 )
4 {
5     UAvailableRaceFormatsEventDTO* Dto =
6         NewObject<UAvailableRaceFormatsEventDTO>();
7     Dto->AddToRoot();
8
9     const TArray<TSharedPtr<JsonValue>> AvailableFormatsJson =
10        JsonObject.GetArrayField("availableFormats");
11
12     for(auto Format : AvailableFormatsJson)
13     {
14         URaceFormatObject* RaceFormat =
15             NewObject<URaceFormatObject>();
16
17         RaceFormat->Id = Format.Get()->AsObject()
18             .Get()->GetStringField("formatId");
19
20         RaceFormat->Length = Format.Get()->AsObject()
21             .Get()->GetStringField("length");
22
23         RaceFormat->Cost = Format.Get()->AsObject()
24             .Get()->GetStringField("cost");
25
26         RaceFormat->Prize = Format.Get()->AsObject()
27     }
28
29 }
```

```

20         .Get() -> GetStringField("prize");
21     Dto->Formats.Add(RaceFormat);
22 }
23
24 if (NetworkActor && NetworkActor->
25     OnAvailableRacesFormatsEventDelegate.IsBound())
26 {
27     NetworkActor->OnAvailableRacesFormatsEventDelegate.Execute(
28         Dto);
29 }
```

Come si vede in Algoritmo: 4.13 il file Json viene spacciato e salvato dentro un oggetto DTO. Il delegate relativo a questo evento viene quindi eseguito passando l'oggetto DTO appena creato. Il delegate, come visto nella funzione *Begin Play* dell'*ApplicationManager* (Algoritmo: 3.3), è legato alla funzione *ShowAvailableRacesFormats* implementata nello *SceneController*. Questa funzione a sua volta passa gli oggetti rappresentati i formati della gara al *WidgetController* che si occuperà di farli visualizzare al giocatore sotto forma di bottoni per poter scegliere a quale gara partecipare.

4.2.0.2 JoinEvent e PlayerJoinedEvent

L'evento *JoinEvent* viene innescato quando un giocatore preme su un bottone corrispettivo ad una specifica tipologia di gara. Similmente a come accade con il bottone *Start* (Algoritmo: 3.6), l'esecuzione del delegate permette di inviare al server un messaggio

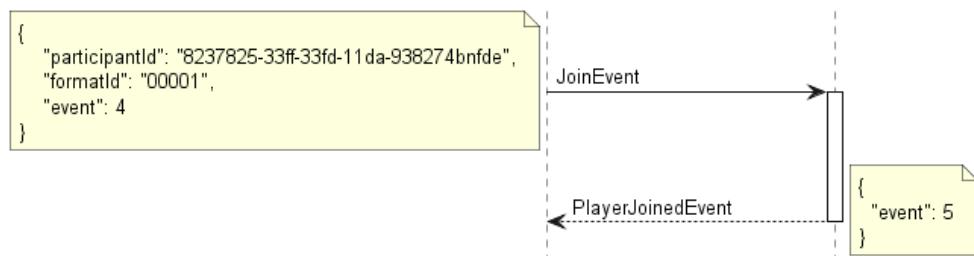


Figura 4.2: Diagramma di sequenza per gli eventi JoinEvent e PlayerJoinedEvent

rappresentante la gara scelta. Il formato del file Json in questione è descritto nella Figura: 4.2.

Il server alla ricezione del messaggio inserisce il giocatore in questione nella gara scelta e invia un messaggio di conferma avvenuta connessione alla gara: questo è l'evento *PlayerJoinedEvent*. Anche questo evento è mappato dentro lo *SceneController* e permette di passare alla vista di gara.

4.2.0.3 StartingGridEvent e AnotherPlayerJoinedEvent



Figura 4.3: Diagramma di sequenza per gli eventi StartingGridEvent e AnotherPlayerJoinedEvent

Questi due eventi permettono al client di ricevere le informazioni riguardanti la gara alla quale si è connesso. L'evento *StartingGridEvent* permette di ricevere le

informazioni della griglia di partenza presente al momento della connessione alla gara, quindi arriveranno tutte le informazioni dei cavalli in attesa dell'avvio della gara già presenti sulla griglia. Il file Json in Figura 4.3 è un esempio di griglia di partenza. L'evento *AnotherPlayerJoinedEvent* permette di ricevere le informazioni dei giocatori che si connetteranno alla gara successivamente. Ne arriverà uno per ogni giocatore aggiuntivo che si connette.

Algoritmo 4.14: Funzione che spacchetta il file Json per la griglia iniziale

```

1 void AMetaraceNetworkActor::OnStartingGridEvent(
2     AMetaraceNetworkActor* NetworkActor, const FJsonObject&
3     JsonObject)
4 {
5     UStartingGridEventDTO* Dto = NewObject<UStartingGridEventDTO>();
6     Dto->AddToRoot();
7
8     const TArray<TSharedPtr<JsonValue>> PlayersArrayJson =
9         JsonObject.GetArrayField("horses");
10
11    for(TSharedPtr<JsonValue> horse : PlayersArrayJson)
12    {
13        UHorseObject* H = NewObject<UHorseObject>();
14        H->Name = horse.Get()->AsObject().Get()->GetStringField("horseName");
15        H->Id = horse.Get()->AsObject().Get()->GetStringField("horseId");
16        H->MeshReference = horse.Get()->AsObject().Get()->GetNumberField("mesh");
17        H->Position = horse.Get()->AsObject().Get()->GetNumberField("position");
18
19        Dto->Players.Add(H);
20    }
21
22    if(NetworkActor &&
23        NetworkActor->OnStartingGridEventDelegate.IsBound())
24    {
25        NetworkActor->OnStartingGridEventDelegate.Execute(Dto);
26    }

```

26 }

27

L'esecuzione di questo delegate chiama la funzione dello *SceneController* che si occupa dello spawn dei cavalli (Algoritmo 4.15):

Algoritmo 4.15: Funzione in *SceneController* che fa comparire i cavalli

```

1 void AMetaraceSceneController::StartingGrid(
2     const UStartingGridEventDTO* StartingGridEventDto)
3 {
4     UWORLD* World = GetWorld();
5     if (!World) { return; }
6
7     FActorSpawnParameters SpawnParams;
8     if (BP_Horse)
9     {
10        for (UHorseObject* HorseObject :
11            StartingGridEventDto->Players)
12        {
13            int32 Position = HorseObject->Position;
14            FTransform SpawnTransform =
15                Splines[Position]->GetTransformAtSplinePoint(
16                    0, ESplineCoordinateSpace::World);
17            AHorse* Horse = World->SpawnActor<AHorse>(
18                BP_Horse, SpawnTransform, SpawnParams);
19            Horse->SetSkeletalMesh(
20                Int2Mesh[HorseObject->MeshReference]);
21            Horse->SetSplineComponent(Splines[Position]);
22            Horse->Name = HorseObject->Name;
23            Horse->Id = HorseObject->Id;
24
25            // I riferimenti ai cavalli vengono tenuti in una TMap
26            Horses.Add(Horse->Id, Horse);
27        }
28    }
29 }
```

4.2.0.4 CountdownEvent e RaceEvent

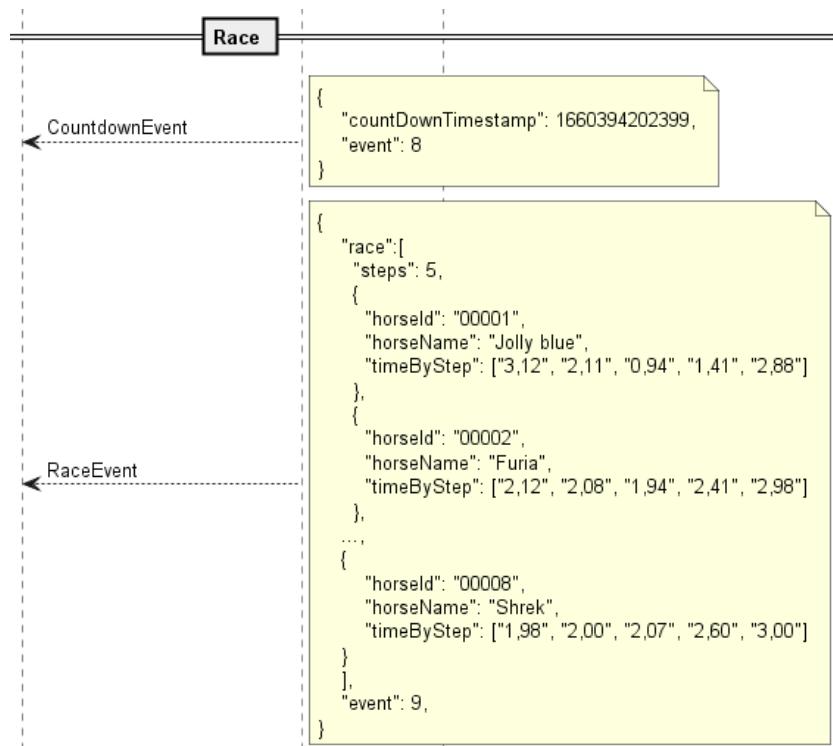


Figura 4.4: Diagramma di sequenza per gli eventi CountdownEvent e RaceEvent

Il *CountdownEvent* è un evento che parte dal server per sincronizzare tutti i client partecipanti alla gara. Esso viene lanciato dal server nel momento in cui le condizioni per poter iniziare la partita sono state raggiunte. Le condizioni sono: il raggiungimento del numero massimo dei partecipanti oppure la scadenza di un conto alla rovescia partito al raggiungimento del numero minimo dei partecipanti. Il file Json associato a questo evento, come si vede nella Figura 4.4, porta con sé solo la durata del timer rimanente per l'inizio della gara. In Unreal Engine ci sono degli oggetti appositamente creati per la gestione del conto alla rovescia, i *FTimerHandle*. I timer in Unreal Engine permettono di definire una funzione da chiamare allo scadere del tempo impostato. La funzione dello SceneController chiamata dal delegate di questo evento comunica la durata del countdown al WidgetController - che si occupa di fare il display del conto alla rovescia - e sfrutta un *FTimerHandle* per far partire la gara - con la funzione *StartRace* - allo scadere del tempo comunicato dal server.

Algoritmo 4.16: Funzione nello SceneController che inizializza un *FTimerHandle*

```

1 void AMetaraceSceneController::ShowCountDown(
2     const UCountdownEventDTO* CountdownEventDto)
3 {
4     WidgetController->ShowCountdown(
5         CountdownEventDto->CountdownTimestamp);
6
7     FTimerHandle TimerHandle2;
8     GetWorldTimerManager().SetTimer(TimerHandle2, this,
9         &AMetaraceSceneController::StartRace, 1.f, false,
10        CountdownEventDto->CountdownTimestamp);
11 }
12

```

Il *RaceEvent* è un altro evento che parte dal server e viene lanciato consecutivamente al *CountdownEvent*. Il file Json associato a questo evento porta tutte le informazioni per lo svolgersi della gara. Questo è il file che contiene l'insieme degli array dei tempi di cui si è discusso nel Paragrafo 2.2, ossia l'insieme dei tempi con cui il cavallo percorrerà ciascuno step del percorso.

Algoritmo 4.17: Funzione che spacchetta il file Json relativo ai dati di gara

```

1 void AMetaraceNetworkActor::OnRaceEvent(AMetaraceNetworkActor*
2     NetworkActor,
3     const FJsonObject& JsonObject)
4 {
5     URaceEventDTO* Dto = NewObject<URaceEventDTO>();
6     Dto->AddToRoot();
7
8     Dto->NumStep = JsonObject.GetNumberField("step");
9
10    const TArray<TSharedPtr<JsonValue>> HorsesJson =
11        JsonObject.GetArrayField("horses");
12    for(TSharedPtr<JsonValue> Horse : HorsesJson)
13    {
14        UHorseObject* H = NewObject<UHorseObject>();
15        H->Id = Horse.Get()->AsObject()
16            .Get()->GetStringField("horseId");
17        const TArray<TSharedPtr<JsonValue>> TimeArrayJson =

```

```

17         Horse.Get()->AsObject().Get()->GetArrayField("timeByStep"
18     );
19
20     for(auto Time : TimeArrayJson)
21     {
22         H->TimeArray.Add(Time.Get()->AsNumber());
23     }
24
25
26     if(NetworkActor && NetworkActor->OnRaceEventDelegate.IsBound())
27     {
28         NetworkActor->OnRaceEventDelegate.Execute(Dto);
29     }
30 }
31

```

La funzione dello *SceneController* chiamata dal delegate per il RaceEvent in Algoritmo 4.17, si occupa di passare l'array di tempi ai cavalli e di chiamare la funzione *Inizialize* vista in Algoritmo: 2.2.

Algoritmo 4.18: Funzione dello SceneController che inizializza i cavalli

```

1 void AMetaraceSceneController::InitRace(
2     const URaceEventDTO* StartRaceEventDto)
3 {
4     for(auto HorseObject : StartRaceEventDto->Horses)
5     {
6         AHorse* Horse = *Horses.Find(HorseObject->Id);
7         Horse->TimeArray = HorseObject->TimeArray;
8         Horse->Inizialize();
9     }
10 }
11

```

4.2.0.5 RaceFinishedEvent e Leaderboardevent

L'evento *RaceFinishedEvent* viene lanciato dallo SceneController stesso. Infatti, in scena è presente un *ATriggerBox* in corrispondenza del traguardo che al verificarsi

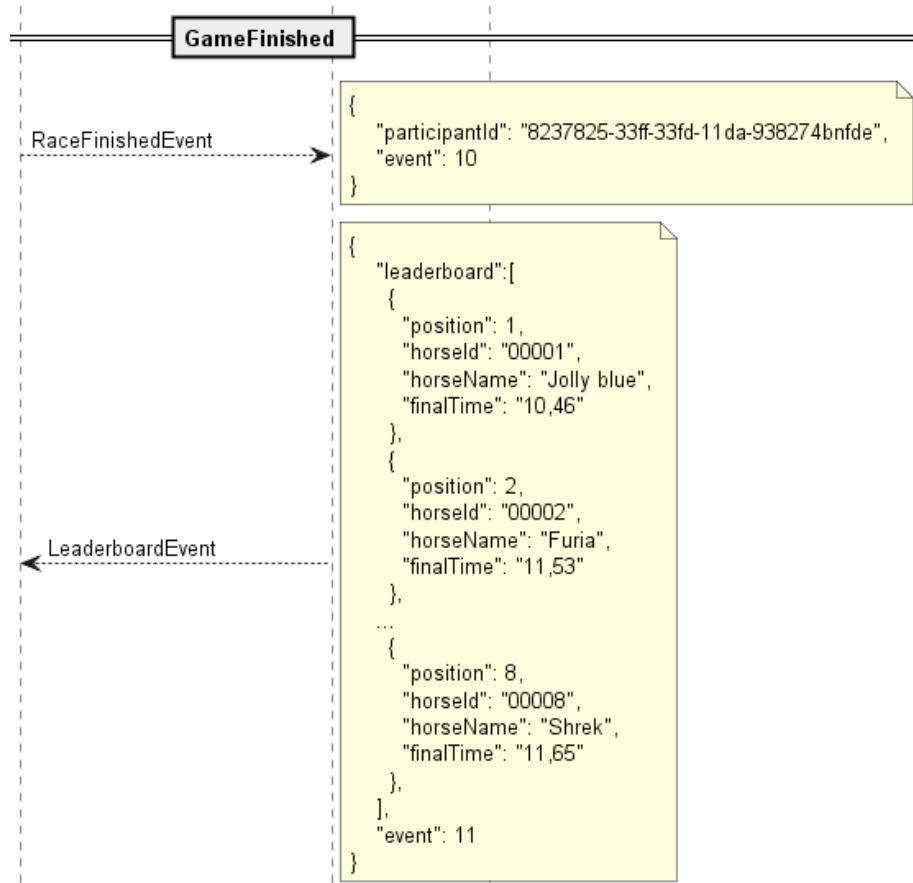


Figura 4.5: Diagramma di sequenza per gli eventi `RaceFinishedEvent` e `LeaderboardEvent`

dell'*OverlapEvent* (ossia quando i cavalli raggiungono il traguardo) chiama la funzione che genera e invia il file Json relativo a questo evento. L'obiettivo di questo evento è quella di far sapere al server che il client ha fatto visualizzare l'intera gara e che aspetta i dati per fare il display della *Leaderboard*. Questo è il motivo per cui anche lo SceneController ha un delegate legato alla funzione *SendMessage* del NetworkActor, Algoritmo 4.9.

Conclusioni e sviluppi futuri

Metarace riesce a creare un ambiente virtuale immersivo grazie all'utilizzo di Unreal Engine e di un visore per la realtà virtuale. Il giocatore può muoversi all'interno di questo mondo per assistere alla gara da diversi punti di osservazione, che coincidono con gli spalti all'interno del gioco. All'interno del mondo di gioco sono state create delle scenografie digitali che hanno come obiettivo quello di rendere il gioco simile ad un'esperienza reale. Ho voluto però mantenere uno stile grafico low poly per dargli una caratterizzazione particolare. In questa tesi mi sono poi soffermato sulle tecnologie e gli strumenti utilizzati per la sua realizzazione, a partire dalle tecniche di modellazione 3D, passando per la programmazione di codice coeso e poco accoppiato, fino ad arrivare all'implementazione del paradigma di programmazione ad eventi. Lo strumento principale per la realizzazione di Metarace, Unreal Engine 5, si è dimostrato un software affidabile e completo per la realizzazione degli obiettivi posti inizialmente. L'utilizzo del linguaggio di programmazione C++ unito all'elevata modularità del codice sorgente di Unreal Engine offre la possibilità di scegliere quali parti di architettura dell'engine utilizzare e quali invece scartare perché superflue. Grazie inoltre al software di modellazione Blender è stato possibile creare e modificare oggetti e animazioni digitali che hanno contribuito alla costruzione del mondo 3D.

Il mondo 3D di Metarace è uno dei mondi all'interno dell'universo virtuale dell'azienda Ringmaster. Questa caratteristica, unita al fatto che può essere sperimentato attraverso un avatar digitale e un visore per la realtà 3D, e che i contenuti del mondo di gioco dovranno essere altamente personalizzabili, gli conferisce i presupposti per essere incluso in una futura implementazione del Metaverso per come è stato definito. Il Metaverso, per come è stato immaginato e concettualizzato nella letteratura Cyber-

punk prima e da Matthew Ball successivamente, comporterebbe l'arrivo di una nuova era tecnologica che andrebbe a rivoluzionare il modo con cui l'uomo si approccia alla tecnologia. La realizzazione del Metaverso necessiterebbe di una nuova iterazione di Internet, di nuovi protocolli ma soprattutto di nuove tecnologie che ancora non sono state concepite. Per questo motivo non possiamo sapere, ad oggi, se il Metaverso verrà mai realizzato nella sua concezione originaria. Nel frattempo quello che si può fare è progettare applicativi altamente scalabili in modo che siano adattabili alla prossima innovazione tecnologica.

Bibliografia

- [1] M. L. Ball. The metaverse explained in 14 minutes, 2022.
- [2] M. L. Ball and J. Navok. Epic's flywheel and unreal engine, 2020.
- [3] Blender. The freedom to create.
- [4] U. di Torino. Second life e unito.
- [5] U. E. Documentation. Delegates.
- [6] U. E. Documentation. Introduction to c++ programming in ue4.
- [7] U. E. Documentation. Landscape outdoor terrain.
- [8] U. E. Documentation. Level editor.
- [9] U. E. Documentation. Unreal smart pointer library.
- [10] U. Engine. Animgraph documentation.
- [11] W. S. Life. Fashion in second life.
- [12] D. Lightbown. Classic tools retrospective: Tim sweeney on the first version of the unreal editor, 2018.
- [13] S. Lozé. Simcentric scales up tactical military simulation training with unreal engine, 2020.
- [14] C. Morningstar and F. R. Farmer. The lessons of lucasfilm's habitat, 1990.
- [15] U. of Illinois. National center for supercomputing applications: History, 2009.

-
- [16] Q. Parker. A second look at school life. *The Guardian*, 2007.
 - [17] C. Plante. Better with age: A history of epic games, 2012.
 - [18] M. Thomsen. History of the unreal engine, 2010.
 - [19] U. C. Wiki. Websocket client - c++.