Project Final Report Template.

Leo Marzoli

Abstract

Il progetto "Team Deathmatch in UE5" si propone di sviluppare un videogioco che ricrea l'esperienza di un classico Team Deathmatch. Il giocatore avrà la possibilità di assumere il ruolo di server (G.S.) o client (G.C.) al momento dell'avvio, consentendo così di avviare una sessione di gioco come host o di unirsi ad essa come cliente. La battaglia si svolge in una mappa il più minimale possibile ai fini del progetto. Il gioco termina quando il team che elimina per primo tutti i membri della squadra avversaria vince. L'aspetto grafico e di gameplay è volutamente non curato, in quanto al di fuori dello scopo del progetto.

Il videogioco è sviluppato utilizzando Unreal Engine 5 (UE5), una delle piattaforme di sviluppo videogiochi più avanzate e potenti disponibili. L'obiettivo principale del progetto è quello di fornire una sessione multiplayer in tempo reale e affrontare argomenti trattati nel corso relativi ai sistemi distribuiti.

Goal/Requirements

Obiettivi del Progetto:

Il progetto "Team Deathmatch in UE5" ha l'obiettivo di sviluppare un videogioco che riproduca l'esperienza classica del Team Deathmatch, con un'attenzione particolare alla componente multiplayer e alla gestione dei sistemi distribuiti. Gli obiettivi principali includono:

1. Implementazione di un'esperienza di gioco coinvolgente e reattiva, che catturi l'essenza del Team Deathmatch.
2. Creazione di un'infrastruttura multiplayer stabile e affidabile utilizzando Unreal Engine 5.
3. Sviluppo di meccanismi di gestione delle sessioni di gioco, inclusi la join e il leave dei giocatori, la sincronizzazione dello stato di gioco tramite replicazione e la gestione degli eventi multiplayer.
4. Rilascio di uno standalone del gioco che ne dimostri il funzionamento.

Requisiti di Sistema in Q/A.

Q: Qual è l'obiettivo del progetto?

A: L'obiettivo del progetto è realizzare un gioco di Team Deathmatch utilizzando Unreal Engine 5, consentendo ai giocatori di partecipare a sessioni di gioco sia come server che come client. Si mira a fornire flessibilità nell'aggiungere o rimuovere giocatori durante le partite. **Q: Quali sono le principali funzionalità che il sistema deve implementare?**

A: È importante che il sistema implementi in modo efficace le meccaniche di base del gioco, come il movimento, lo sparo e la gestione della vita dei personaggi. Inoltre, è fondamentale fornire una mappa di gioco funzionale per le battaglie di Team Deathmatch.

Q: Qual è l'attenzione principale del progetto in termini di gameplay e tecnologia?

A: Il focus principale del progetto è sulla realizzazione di un gioco multiplayer che metta in evidenza la replicazione dello stato e delle componenti dell'engine.

Q: Quali sono le aspettative per la stabilità e l'affidabilità del sistema?

A: Si prevede che il sistema sia stabile, sincronizzato e privo di crash durante le sessioni di gioco. È essenziale che la replicazione delle componenti dell'engine funzioni in modo affidabile, garantendo una consistenza ottimale dell'esperienza di gioco per tutti i giocatori.

Esiti Attesi:

Gli esiti attesi del progetto includono:

1. Un videogioco funzionante del Team Deathmatch per i giocatori.
2. Un codice ben strutturato e documentato, che facilita la manutenzione e l'ulteriore sviluppo del progetto in futuro.

Scenarios

Come un utente interagisce col sistema:

Gli utenti interagiscono con il progetto attraverso le seguenti modalità:

1. Apertura del gioco: Avviato lo standalone i giocatori interagisco con il main menù di gioco per assumere il ruolo di server (G.S.) o client (G.C.).
2. Accesso come Host: Diventando un server un giocatore mette a disposizione la propria macchina per creare una rete multiplayer a cui potrà partecipare.
3. Accesso come Client: Gli utenti possono partecipare alle sessioni di gioco come giocatori, impegnandosi in battaglie di Team Deathmatch con altri giocatori online.

Perché un utente interagisce col sistema:

Gli utenti interagiscono con il sistema per le seguenti ragioni:

1. Intrattenimento: Gli utenti giocano al gioco per divertirsi e sfidare se stessi e altri giocatori in battaglie multiplayer.
2. Socializzazione: Gli utenti partecipano alle sessioni di gioco multiplayer per interagire e socializzare con altri giocatori online, collaborando o competendo con loro nelle battaglie di Team Deathmatch.
3. Esplorazione: Gli utenti esplorano la meccanica di gioco per migliorare le loro prestazioni in gioco.

Self-assessment policy

Valutazione della Qualità del Software:

La qualità del software prodotto ha una natura conflittuale con quello che è l'approccio test-driven development, in quanto essendo Unreal Engine un game engine il risultato di tutto ciò che viene prodotto in esso è puramente visuale/grafico. Pertanto, la correttezza del codice è sicuramente verificata tramite il sistema di build integrato nell'editor. Per quanto riguarda il testing delle funzionalità questo può sicuramente avvenire tramite l’utilizzo dell’engine in quanto i cambiamenti come così tutto il codice sono direttamente verificabili nel simulatore che in caso di build corretta accerta visivamente se ciò che si è sviluppato funziona o meno. Ma in questo caso sono stati sviluppati per fine didattici dei Functional Test. Unreal engine stesso mette a disposizione un sistema di testing facilitato, non a caso di seguito riporto un esempio delle impostazioni di unreal engine che evidenziano quanto sia avanzata la simulazione:

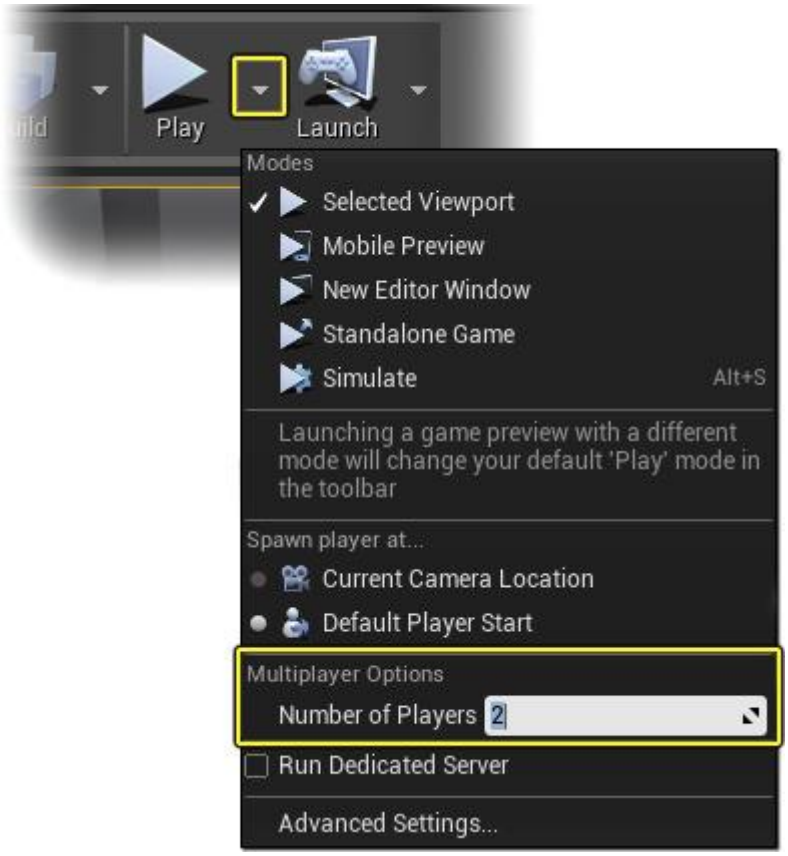

Fig. 1 – Settings per impostare la simulazione di gioco.

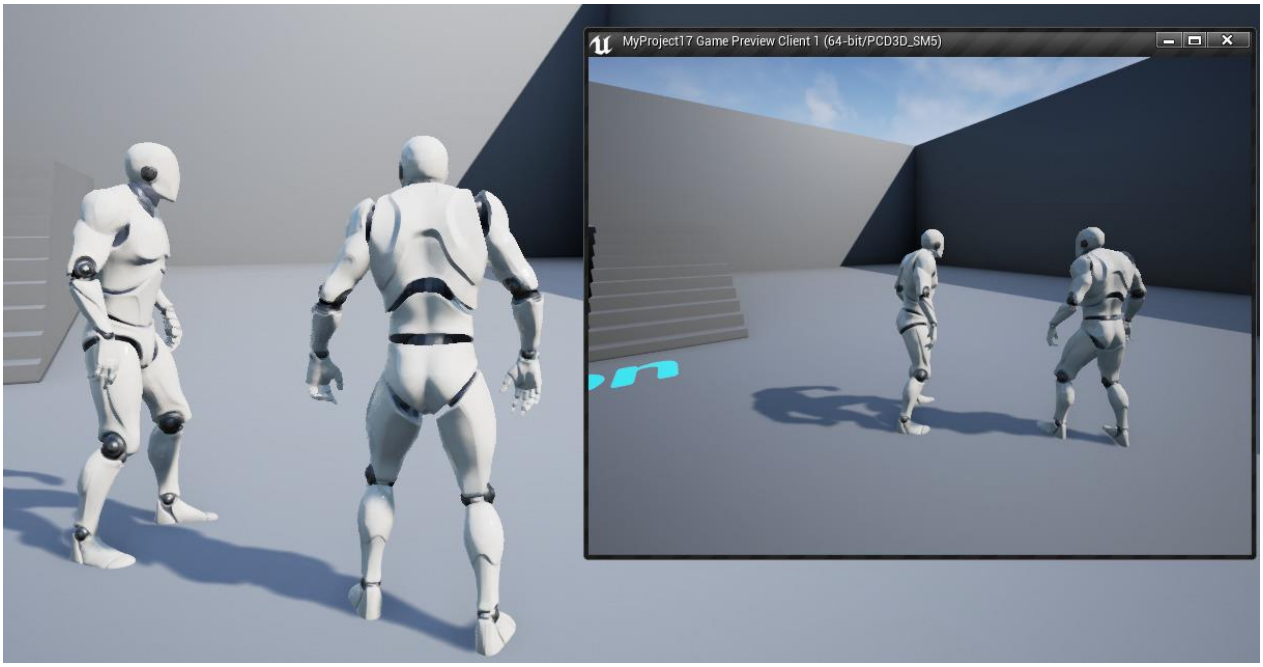

Fig. 2 – Finestre di gioco al momento dell’avvio della simulazione.

Valutazione dell'Effettività dei Risultati del Progetto:

L'efficacia dei risultati del progetto sarà valutata attraverso i seguenti indicatori:

1. Il primo impatto e l'usabilità del gioco da parte degli utenti è una misura di correttezza.
2. Si accertano le prestazioni del gioco durante le sessioni multiplayer, il tempo di risposta e la stabilità della rete, per valutare l'efficacia del sistema nel gestire carichi di lavoro variabili e garantire un'esperienza di gioco fluida e senza intoppi evitando conseguenti crash.

Requirements Analysis

Requisiti Funzionali

1. Il sistema deve consentire agli utenti di mantenere una sessione di gioco come server, permettendo loro di controllare e gestire la partita.
2. I giocatori devono avere la possibilità di unirsi a una sessione di gioco esistente come client, partecipando alle partite in corso.
3. I giocatori devono poter utilizzare componenti di movimento in modo efficace all'interno di una rete di gioco, garantendo un movimento fluido e reattivo per i personaggi.
4. Il sistema deve permettere agli sviluppatori di aggiungere la replicazione a variabili o attori del sistema, garantendo la loro sincronizzazione tra server e client.
5. Gli sviluppatori devono essere in grado di utilizzare RepNotifies per modificare una variabile in modo sincronizzato tra server e client.
6. Gli sviluppatori devono poter utilizzare chiamate di procedura remote in C++ per gestire comunicazioni complesse tra server e client.
7. Gli sviluppatori devono poter controllare il ruolo di rete di un attore per filtrare le chiamate a funzioni specifiche, garantendo un corretto funzionamento della logica di gioco.
8. Il sistema deve gestire in modo efficace la dimensione delle partite per i giocatori.
9. Il sistema deve implementare le meccaniche di gioco quali: sparare proiettili e perdere vita, con conseguente morte del giocatore.
10. Creazione di una mappa di gioco funzionale, anche se minimale, che fornisca un ambiente adeguato al Team Deathmatch.
11. Il sistema deve consentire a un numero variabile di giocatori di partecipare alle sessioni di gioco, con la possibilità di aggiungere o rimuovere giocatori in modo dinamico.
12. Realizzazione di un'interfaccia utente intuitiva per consentire ai giocatori di avviare o unirsi a sessioni di gioco.

Requisiti Non Funzionali:

1. Affidabilità: Il sistema deve essere stabile e privo di crash durante le sessioni di gioco, garantendo un'esperienza fluida per i giocatori.

2. Scalabilità: Il sistema deve essere in grado di gestire un numero crescente di giocatori senza compromettere le prestazioni o la stabilità del gioco.
3. Usabilità: L'interfaccia utente deve essere intuitiva e ben progettata, facilitando l'accesso e la partecipazione dei giocatori alle sessioni di gioco.

Requisiti Tecnici

1. Game Engine: utilizzo di un game engine in grado realizzare il gioco, in questo caso si è volutamente scelto Unreal Engine 5.
2. Scelta del linguaggio di programmazione: si è scelto appositamente di sviluppare il progetto in C++ in quanto nativamente compatibile con l'engine e con lo sviluppo di videogiochi per la sua velocità computazionale.

Design

In un gioco multiplayer, Unreal Engine utilizza un modello client-server. Un computer nella rete agisce come server e ospita una sessione di gioco multiplayer, mentre tutti gli altri computer dei giocatori si connettono al server come client. Il server, quindi, condivide le informazioni sullo stato di gioco con ciascun client connesso e fornisce un mezzo per comunicare tra loro. Il server, in quanto host del gioco, detiene l'unico vero e autoritario stato di gioco. In altre parole, il server è dove il gioco multiplayer sta effettivamente avvenendo. I clienti controllano in remoto i Pawn (personaggi) di cui sono proprietari sul server, inviando procedure per farli eseguire azioni di gioco. Tuttavia, il server non trasmette direttamente le immagini visive ai monitor dei clienti. Invece, il server replica le informazioni sullo stato di gioco a ciascun client, dicendo loro quali Attori dovrebbero esistere, come quegli Attori dovrebbero comportarsi e quali valori dovrebbero avere diverse variabili. Ogni client utilizza quindi queste informazioni per simulare un'approssimazione molto vicina di ciò che sta accadendo sul server.

Di seguito si riportano i passaggi principali che avvengono durante una connessione client-server:

1. Il client invia una richiesta di connessione.
2. Se il server accetta, invierà la mappa/livello attuale.
3. Il server attende che il client carichi questa mappa/livello.
4. Una volta caricata, il server chiamerà localmente *AGameModeBase::PreLogin*.
    - Questo darà al *GameMode* (che risiede solo sul server e non è replicato) la possibilità di rifiutare la connessione.
5. Se accettato, il server chiamerà quindi *AGameModeBase::Login*.
    - Il ruolo di questa funzione è creare un PlayerController, che verrà quindi replicato al client appena connesso. Una volta ricevuto, questo PlayerController sostituirà il PlayerController temporaneo dei client utilizzato come segnaposto durante il processo di connessione. ◦ Si noti che *APlayerController::BeginPlay* verrà chiamato qui. Si fa notare che NON è ancora sicuro chiamare le funzioni RPC su questo attore. È necessario attendere che venga chiamato *AGameModeBase::PostLogin*.

6. Supponendo che tutto sia andato bene, verrà chiamato *AGameModeBase::PostLogin*.
    - A questo punto, è sicuro per il server iniziare a chiamare le funzioni RPC su questo PlayerController.

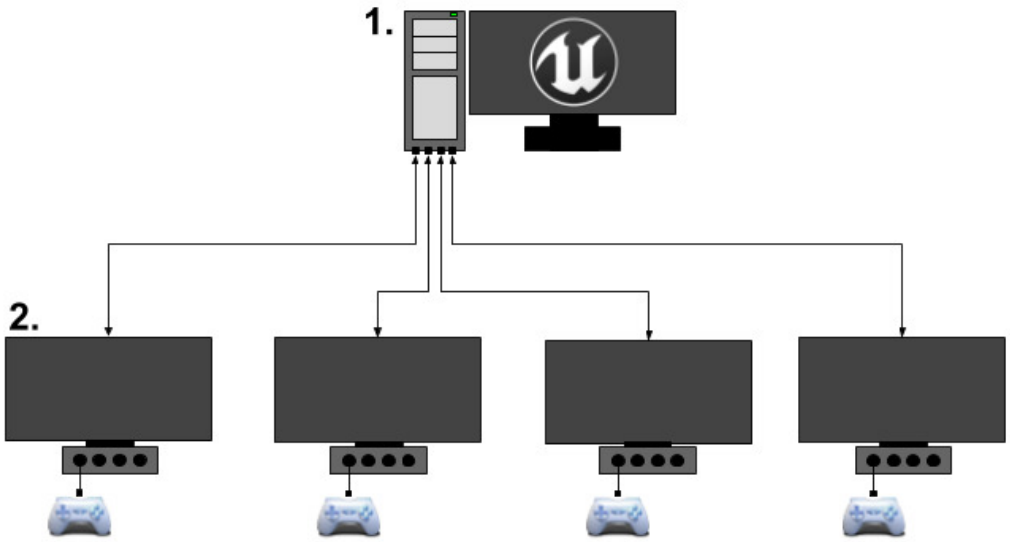

Fig. 3 – Architettura di Unreal Engine

Quando viene fatto partire l'hosting di gioco Unreal Engine inizializza una socket sulla porta 17777. Ciascun client al momento del suo avvio si connetterà a questa porta. Nel multiplayer, le interazioni avvengono in diversi "mondi": uno che esiste sul server, uno che esiste sul client del Giocatore 1, uno che esiste sul client del Giocatore 2 e un mondo aggiuntivo per ogni altro client che partecipa alla sessione. Ogni mondo su ogni macchina diversa ha la propria copia dei Pawn, delle loro armi e dei proiettili che sparano. Il server è dove il gioco viene effettivamente giocato, ma dobbiamo fare in modo che i mondi dei client sembrino che gli stessi eventi stiano accadendo in essi. È quindi necessario inviare selettivamente informazioni a ciascun client per creare una rappresentazione visiva del mondo sul server.

Questo processo introduce una divisione tra interazioni di gioco essenziali (collisioni, movimento, danni), effetti cosmetici (effetti visivi e audio che in questo caso non sono stati implementati) e informazioni private del giocatore (aggiornamenti dell'HUD). Ognuno di questi è rilevante per una macchina specifica o un insieme di macchine nella rete. Tuttavia, il processo di replicare queste informazioni non è automatico, e bisogna specificare quali informazioni vengono replicate a quali macchine mentre si programma il gameplay. Questa è la principale sfida, ovvero scegliere quali informazioni bisogna replicare e come farlo per fornire un'esperienza coerente a tutti i giocatori, ma anche minimizzare la quantità di informazioni che si stanno replicando per utilizzare la minor larghezza di banda possibile.

Structure

In questa sottosezione si riportano le principali classi dell'engine che sono state coinvolte nella realizzazione del progetto assieme a quelle create appositamente a tale

scopo. Si evita di riportare per ogni classe l'intero elenco di attributi e membri poiché questo faciliterà molto la dimensione e la leggibilità degli UML, ma cercherò di focalizzarmi su ciò che è importante per ciascuna classe rappresentata. Di seguito viene riportato il Class Diagram:

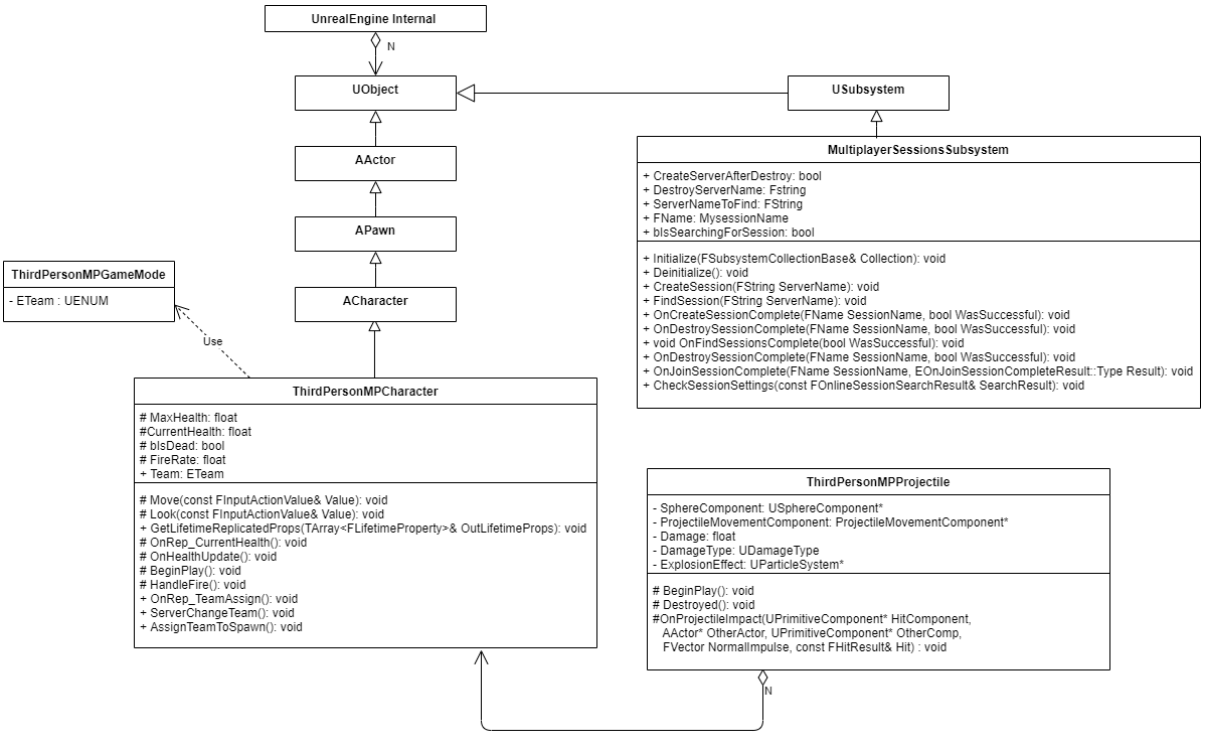

Fig. 4 – Diagramma delle classi della struttura del progetto.

Dovendo descrivere brevemente ciò che viene rappresentato si possono identificare tre macro-classi su cui la maggior parte del lavoro è stato svolto:

- ThirdPersonMPCharacter: questa è la classe che opera sia da host/server che da client e raggruppa tutta la logica del giocatore.
- ThirdPersonMPProjectile: questa rappresenta il proiettile emesso dal giocatore, il cui scopo è quello di "ferire".
- MultiplayerSessionsSubsystem: è la classe dedicata alla creazione e al ritrovamento di sessioni di gioco.

Come si evidenzia dal diagramma delle classi, le classi sviluppate sono accumunate da un superclasse UObject, questa è la classe padre in Unreal Engine da cui poi tutte le altre classi ereditano.

Inoltre, si fa notare che la classe ThirdPersonMPGameMode è volutamente scoleggata dal contesto in quanto essa è utilizzata all'interno dell'engine in maniera default, riconosciuta tramite naming. Si fa notare la presenza anche di:

- AActor
- APawn
- ACharacter

Queste classi specifiche sono una la specializzazione dell'altra secondo l'ordine riportato. Rappresentano rispettivamente: un attore qualsiasi all'interno del livello/scena di gioco, un attore qualsiasi in grado di essere "controllato" dal

giocatore nel livello/scena di gioco, un attore qualsiasi in grado di rappresentare il main player e le sue funzionalità.

Tramite UnrealEngine Internal si vuole evidenziare come la classe ThirdPersonMPCharacter e MultiplayerSessionsSubsystem utilizzino al loro caricamento meccanismi per comunicare come server o come client grazie alla rete che Unreal Engine crea al momento dell'avvio del gioco tramite connessione socket sulla porta 17777. Questa porta è il default, ma andando nelle impostazioni è possibile cambiarla a proprio piacimento.

Per quanto riguarda invece la modularizzazione del progetto si riporta di seguito un diagramma delle componenti:

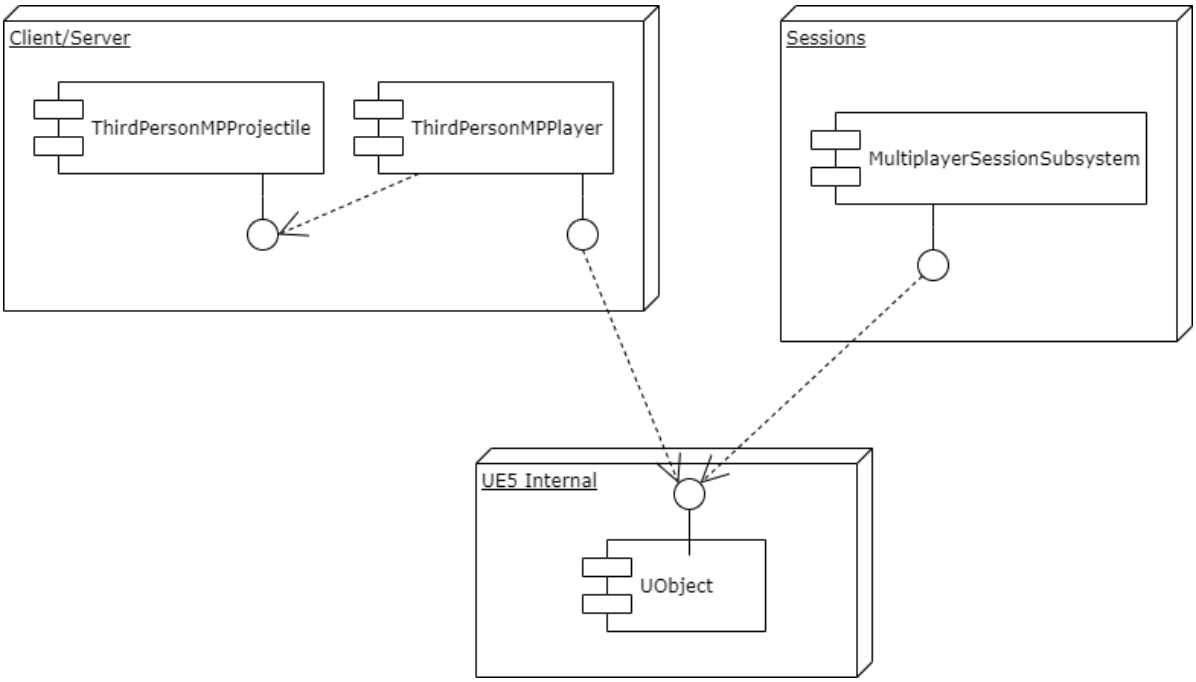

Fig. 5 – Diagramma delle componenti del progetto

Al suo interno si vuole evidenziare come le varie sezioni del progetto possono essere raggruppate. Come si evince dal diagramma si ha una sezione per rappresentare:

- La logica del giocatore che opera sia come client che come server a seconda del ruolo che ad esso viene assegnato da unreal engine nel momento in cui decide se "hostare" o "trovare" una sessione di gioco.
- Vi è la componente di gestione delle sessioni. Entrambe le precedenti comunicano con una componente interna ad unreal engine per la gestione dei ruoli di server o client o per la gestione delle sessioni.

Behaviour

Innanzitutto, bisogna evidenziare che Unreal Engine come qualsiasi altro game engine al momento del suo avvio sia che avvenga nell'editor o tramite standalone dispone di una serie di funzioni che ne determina il flusso di esecuzione (queste prendono nomi specifici, non alterabili) che poi vengono utilizzate all'interno dello sviluppo del gioco stesso, per determinare un sott'ordine (scelto dallo sviluppatore) di eventi fra i quali alcuni esempi: far comparire il menù, spawnare il personaggio, iniziare a giocare, ecc. Banalmente tuttò ciò avviene tramite un overriding dei metodi in questione.

Un esempio che dovrebbe chiarire le idee è proprio l'implementazione di questi metodi dai nomi predefiniti, in modo tale da orchestrare al meglio la logica di gioco.

```
void AThirdPersonMPCharacter::BeginPlay()
{
  // Call the base class
  Super::BeginPlay();

  // Here goes everything that the player must do at beginning.
}
```

Di seguito viene riportato un diagramma delle attività che vuole rappresentare gli stati assunti da ciascuna entità sviluppata all'interno del progetto, durante l'avvio del gioco:

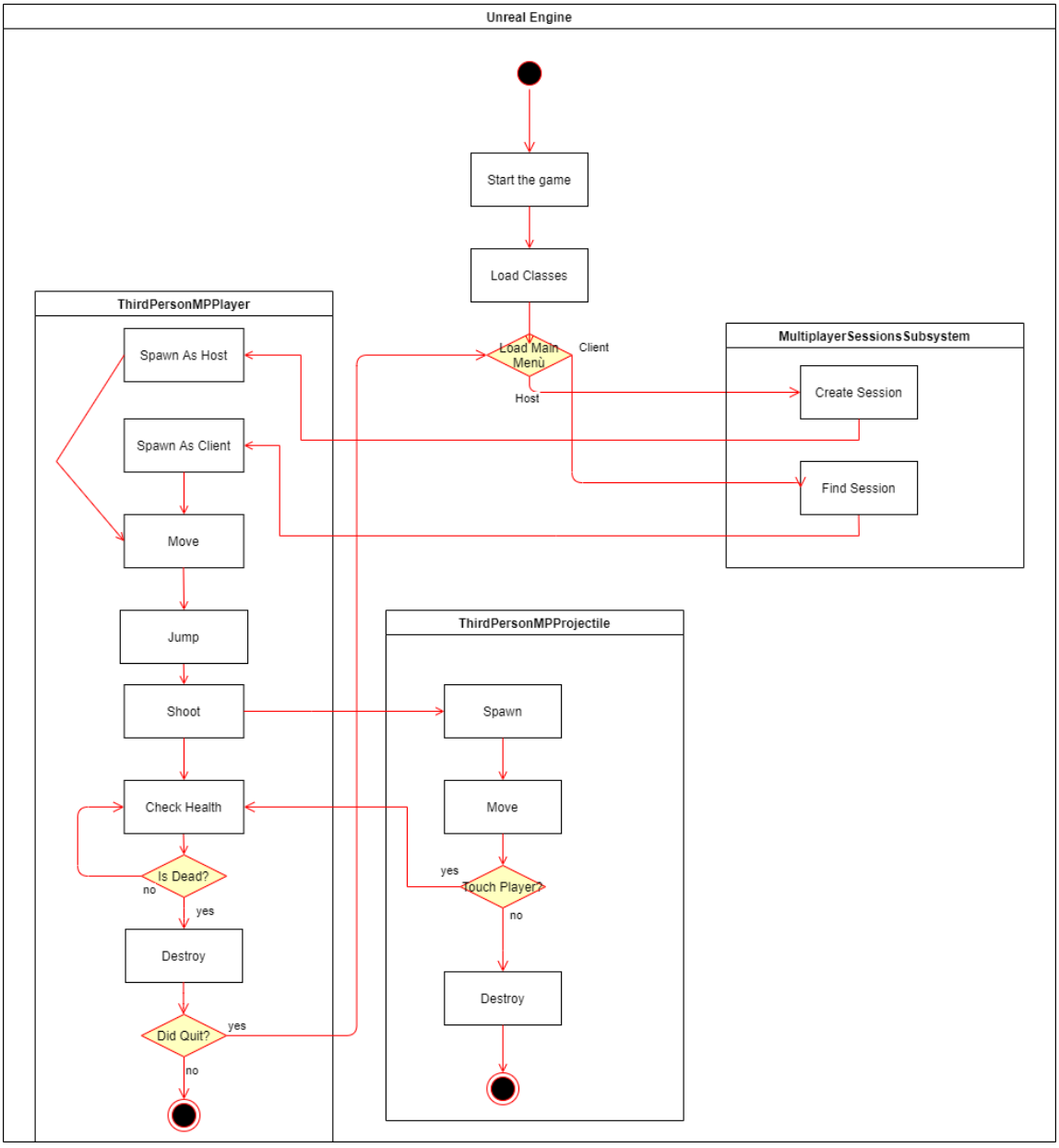

Fig.6 – Diagramma delle attività del progetto.

Descrivendo in breve ciò che viene raffigurato, si ha che Unreal Engine è colui che inizia il gioco (dando per scontato che il gioco viene fatto avviare da una persona fisica), per poi caricare tutte le classi di gioco (visibile tramite il diagramma delle attività di UE5 raffigurato prima) e i vari assets. Caricato il main menù si hanno a disposizione due pulsanti: Host a Session e Join a Session, questi attiveranno MultiplayerSessionsSubsystem che rimane in ascolto degli eventi generati dal "click" di questi due pulsanti e si preoccuperà di "spawnare" un ThirdPersonMPPlayer all'interno del livello di gioco, esso potrà essere sia Server che Client ed entrambi potranno fare azioni di base come muoversi, saltare, tuttavia sparare creerà una nuova entità di gioco ThirdPersonMPProjectile che "spawna" nella posizione del giocatore e si muove nella direzione puntata dalla telecamera associata al giocatore (simulando una mira), questa continuerà a vagare finché non colliderà con un'altra entità di gioco e se questa è il giocatore allora li toglie della vita. Il giocatore dalla supa parte controlla periodicamente sulla base di una RepNotifier sulla vita se è morto, in caso debba essere distrutto. Premendo il tasto Q della tastiera si accede ad un pulsante di Quit che permette a client o server di uscire dalla sessione di gioco, ritornando al main menù e potendo rigiocare.

Interaction

Di seguito si riporta il sequence diagram che rappresenta come le varie entità interagiscono tra di loro:

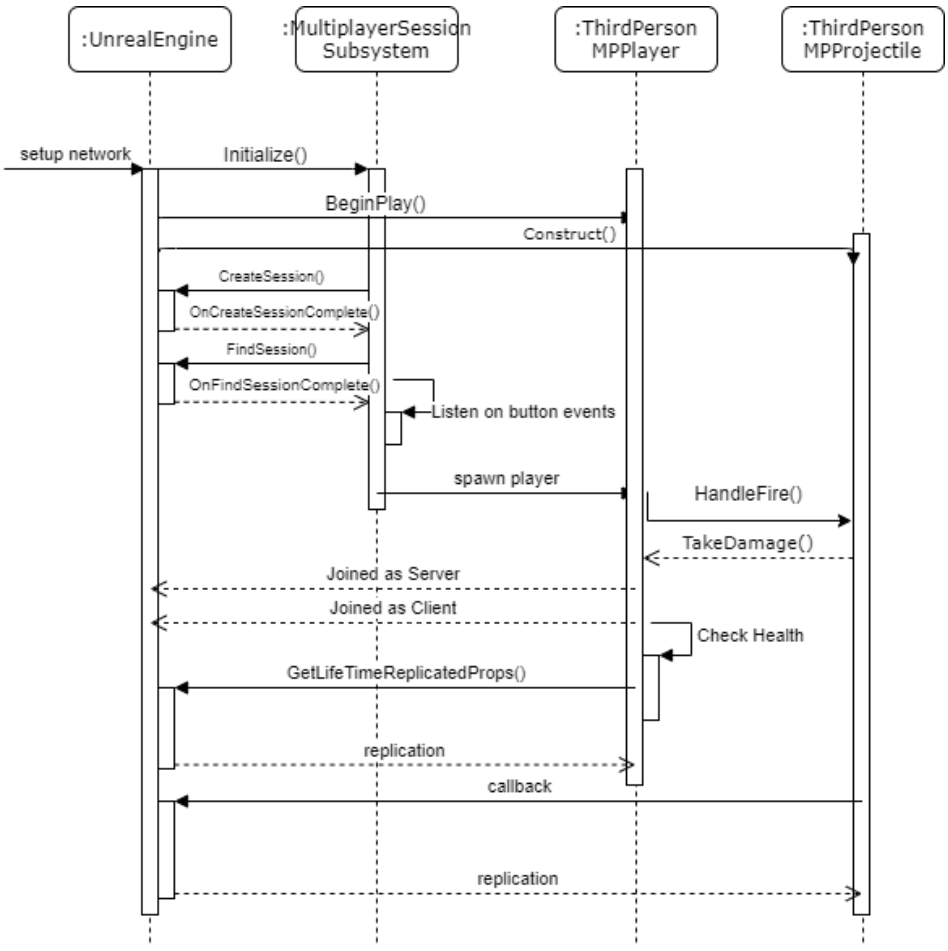

Fig. 7 – Diagramma di sequenza che mostra le interazioni del progetto.

Descrivendo brevemente ciò che viene raffigurato si ha che inizialmente Unreal Engine si occupa di richiamare tutti quei metodi (come esposto nel diagramma delle attività precedente) utili al caricamento delle entità al suo interno; dunque, avrà una chiamata per ogni classe. Da qui, ci si può focalizzare su:

- MultiplayerSessionsSubsystem: effettua delle callback, utilizzando una comunicazione asincrona sia per la creazione della sessione che per cercare una sessione esistente. Ciò che viene ritornato quando ciascuno di questi metodi viene invocato è un evento che "triggera" l'esecuzione di OnCreateSessionComplete() e di OnFindSessionComplete(). Nel mentre continua a rimanere in ascolto per ulteriori eventi generati dal click dei bottoni del main menù. E infine "spawna" il player, facendo in modo che venga "caricato" nel livello dedicato al gameplay.

- ThirdPersonMPPlayer: Esso nel momento in cui viene spawnato ha un ruolo assegnato dal server che lo autorizza come client o come server/host. Inoltre a fare ciò che è stato detto in precedenza si dedica anche tramite la funzione "HandleFire()" allo "spawn" un proiettile il cui contatto ritornerebbe un valore, che viene sottratto alla vita del giocatore, controllando periodicamente se la sua vita ha raggiunto lo zero e in caso muore scomparendo dal livello del gioco. Infine, cerca sempre di replicare comunicando con Unreal Engine le variabili.

- ThirdPersonMPProjectile: il suo ruolo oltre è quello di ridurre la vita del giocatore e di replicarsi tramite Unreal Engine.

Implementation Details

In questa sezione verranno riportati soltanto dettagli non banali del codice sviluppato a cui comunque è assistita una notevole presenza di commenti proprio per facilitarne la comprensione. Di seguito si riporta una descrizione di alcune componenti del sistema che permettono di capire la logica di base che poi è stata utilizzata per creare il progetto. **Replicare la vita del giocatore tramite RepNotifiers**

I giocatori hanno bisogno di un valore di salute in modo che possiamo infliggere loro danni durante il gioco. Quel valore deve replicarsi in modo che tutti i clienti abbiano informazioni sincronizzate sulla salute di ciascun giocatore, e abbiamo bisogno di fornire un feedback a un giocatore quando subisce danni. Questa sezione dimostrerà come è possibile utilizzare un RepNotify per sincronizzare tutti gli aggiornamenti essenziali a una variabile senza fare affidamento su RPC (Remote Procedure Calls).

```
//ThirdPersonMPCharacter.h

/** The player's maximum health. This is the highest that their health can be, and the value that their health starts at when spawned.*/
UPROPERTY(EditDefaultsOnly, Category="Health")
float MaxHealth;

/** The player's current health. When reduced to 0, they are considered dead.*/
UPROPERTY(ReplicatedUsing=OnRep_CurrentHealth)
float CurrentHealth;

/** RepNotify for changes made to current health.*/
UFUNCTION()
void OnRep_CurrentHealth();
```

Vogliamo controllare attentamente come viene modificata la salute del giocatore; pertanto, questi valori di salute hanno i seguenti vincoli:

- *MaxHealth* non si replica ed è modificabile solo nei valori predefiniti (in termini di Blueprints) . Questo valore è precomputato per tutti i giocatori e non cambierà mai.
- *CurrentHealth* si replica, ma non è modificabile o accessibile da nessuna parte nei Blueprints.
- Sia *MaxHealth* che *CurrentHealth* sono protected, il che impedisce loro di essere accessibili da classi C++ esterne. Possono essere modificati solo all'interno di AThirdPersonMPCharacter o altre classi da essa derivate.

Ciò riduce al minimo il rischio di causare modifiche indesiderate alla salute corrente o massima di un giocatore durante il gioco in tempo reale. A queste verranno affiancate delle funzioni pubbliche per ottenere e modificare questi valori in un secondo momento.

Il modificatore "Replicated" consente la copia di un attore sul server per replicare il valore di una variabile a tutti i client connessi ogni volta che cambia.
"ReplicatedUsing" fa la stessa cosa, ma ci consente di impostare una funzione RepNotify che verrà attivata quando un client riceve con successo i dati replicati. Utilizzeremo "OnRep_CurrentHealth" per eseguire gli aggiornamenti su ciascun client basati sulle modifiche a questa variabile.

```
//ThirdPersonMPCharacter.h

/** Property replication */
void GetLifetimeReplicatedProps(TArray<FLifetimeProperty>& OutLifetimeProps) const
override;
```

```
//ThirdPersonMPCharacter.cpp
```

```
void AThirdPersonMPCharacter::GetLifetimeReplicatedProps(TArray <FLifetimeProperty> &
OutLifetimeProps) const
{
    Super::GetLifetimeReplicatedProps(OutLifetimeProps);

    //Replicate current health.
    DOREPLIFETIME(AThirdPersonMPCharacter, CurrentHealth);
}
```

La funzione *GetLifetimeReplicatedProps* è responsabile della replica di tutte le proprietà a cui attribuiato l'identificatore *Replicated*, e ci consente di configurare come una proprietà verrà replicata. Qui stiamo utilizzando l'implementazione più basilare per CurrentHealth. Tutte le variabili che vogliamo essere replicate andranno inserite in questa funzione.

Bisogna invocare la Super di GetLifetimeReplicatedProps, altrimenti le proprietà ereditate dalla classe genitore dell’attore personale non verranno replicate, anche se la classe genitore le segna come replicate.

```
//ThirdPersonMPCharacter.cpp

void AThirdPersonMPCharacter::OnRep_CurrentHealth()
{
    OnHealthUpdate();
}

void AThirdPersonMPCharacter::OnHealthUpdate()
{

    //Functions that occur on all machines.

}
```

Useremo *OnHealthUpdate()* per eseguire gli aggiornamenti in risposta ai cambiamenti della *CurrentHealth* del giocatore. In questo modo le variabili vengono replicate ogni volta che il loro valore cambia anziché replicarsi costantemente, e i RepNotifies vengono eseguiti ogni volta che il client riceve con successo un valore replicato per una variabile. Pertanto, ogni volta che cambiamo la *CurrentHealth* del giocatore sul server, ci aspettiamo che OnRep_CurrentHealth venga eseguita su ogni client connesso. Questo rende OnRep_CurrentHealth il luogo ideale per chiamare OnHealthUpdate sulle macchine dei clienti. **Sparare il proiettile**

In questa sezione si vuole evidenziare l'utilizzo di attività che soltanto il server deve gestire e per fare ciò utilizziamo delle RPC.

```
//ThirdPersonMPCharacter.h

/** Delay between shots in seconds. Used to control fire rate for our test projectile, but also to prevent an overflow of server functions from binding SpawnProjectile directly to input.*/
UPROPERTY(EditDefaultsOnly, Category="Gameplay")
float FireRate;

/** If true, we are in the process of firing projectiles. */
bool bIsFiringWeapon;

/** Function for beginning weapon fire.*/
UFUNCTION(BlueprintCallable, Category="Gameplay")
void StartFire();

/** Function for ending weapon fire. Once this is called, the player can use StartFire again.*/
UFUNCTION(BlueprintCallable, Category = "Gameplay")
void StopFire();

/** Server function for spawning projectiles.*/
UFUNCTION(Server, Reliable)
void HandleFire();

/** A timer handle used for providing the fire rate delay in-between spawns.*/
FTimerHandle FiringTimer;
```

Queste sono le variabili e le funzioni che utilizzeremo per sparare i nostri proiettili. *HandleFire* è l'unico RPC che implementeremo e sarà responsabile della generazione dei proiettili sul server. Poiché ha il specificatore Server, qualsiasi tentativo di chiamarlo su un client comporterà che la chiamata venga instradata sulla rete al Character "authoritive" sul server.

Poiché *HandleFire* ha anche il specificatore Reliable, viene inserito in una coda per gli RPC Reliable ogni volta che viene chiamato, e viene rimosso dalla coda quando il server lo riceve correttamente. Ciò garantisce che il server riceverà definitivamente questa chiamata di funzione. Tuttavia, la coda per gli RPC Reliable può traboccare se troppi RPC vengono inseriti contemporaneamente senza rimuoverli, e se lo fa allora costringerà l'utente a disconnettersi. Pertanto, bisogna essere cauti su quanto spesso si concede ai giocatori di chiamare questa funzione.

```
//AThirdPersonMPCharacter.cpp

void AThirdPersonMPCharacter::StartFire()
{
    if (!bIsFiringWeapon)
    {
        bIsFiringWeapon = true;
        UWorld* World = GetWorld();
        World->GetTimerManager().SetTimer(FiringTimer, this,
&AThirdPersonMPCharacter::StopFire, FireRate, false);
        HandleFire();
    }
}

void AThirdPersonMPCharacter::StopFire()
{
    bIsFiringWeapon = false;
}
```

```
void AThirdPersonMPCharacter::HandleFire_Implementation()
{
    FVector spawnLocation = GetActorLocation() + ( GetControlRotation().Vector()  *
100.0f ) + (GetActorUpVector() * 50.0f);
    FRotator spawnRotation = GetControlRotation();

    FActorSpawnParameters spawnParameters;
    spawnParameters.Instigator = GetInstigator();
    spawnParameters.Owner = this;

    AThirdPersonMPProjectile* spawnedProjectile = GetWorld()-
>SpawnActor<AThirdPersonMPProjectile>(spawnLocation, spawnRotation, spawnParameters);
}
```

StartFire è la funzione che i giocatori chiamano sul proprio computer locale per avviare il processo di "fire" (questa viene chiamata tramite input controller premendo il tasto sinistro del mouse), e limita quanto spesso all'utente è consentito chiamare *HandleFire* in base ai seguenti criteri:

- L'utente non può sparare un proiettile se è già nell'azione di "fire". Questo è indicato con *bFiringWeapon*, che viene impostato su true quando viene chiamato *StartFire*.

- *bFiringWeapon* viene impostato su false solo quando viene chiamato StopFire.
  StopFire viene chiamato quando un timer con una durata pari a *FireRate* finisce.

Questo significa che, quando l'utente spara un proiettile, deve aspettare un numero di secondi pari a *FireRate* prima di poter sparare di nuovo. Questo funzionerà in modo coerente indipendentemente da quale tipo di input controller *StartFire* è vincolato (in questo caso il click del mouse). Ad esempio, se l'utente associa il comando "Fire" a una rotella di scorrimento o a un input simile inappropriato, o se preme ripetutamente il pulsante, questa funzione verrà comunque eseguita a un intervallo di tempo accettabile e non riempirà la coda dell'utente per le funzioni affidabili con chiamate a *HandleFire*.

Poiché *HandleFire* è un RPC Server, la sua implementazione nel file CPP deve avere il suffisso "_Implementation" aggiunto al nome della funzione. L' implementazione qui utilizza la rotazione del Personaggio per ottenere la direzione in cui la fotocamera è rivolta, quindi spawnare il proiettile in quella direzione, consentendo al giocatore di mirare. Il Componente di Movimento del Proiettile poi si occupa di muoverlo nella direzione mirata.

Infine, si vuole evidenziare che la replicazione dello stato del sistema avviene attraverso la replicazione immediata delle funzioni chiamate. Quando si invocano funzioni, queste vengono immediatamente inserite nella coda di rete in uscita. Tuttavia, le variabili non vengono replicate fino alla fine del tick corrente, il che potrebbe causare una ricezione delle funzioni prima delle variabili corrispondenti da parte del client. Per evitare problemi, è consigliabile passare le variabili necessarie direttamente alla funzione. Le funzioni possono essere affidabili o non affidabili: le prime garantiscono l'ordine di esecuzione, mentre le seconde no. Esiste anche un tipo intermedio per funzioni che devono essere eseguite in ordine ma possono tollerare la perdita di pacchetti. Sul server, tutte le funzioni vengono eseguite, a meno che non siano specificamente replicate al client. Sul client, le funzioni devono avere un'origine, come una funzione replicata dal server o un evento. Gli eventi sono semplici e si verificano in base allo stato del gioco. Le funzioni che non soddisfano le condizioni per l'esecuzione non vengono eseguite e ritornano valori predefiniti. La replicazione delle funzioni può avvenire solo se l'attore è posseduto da un PlayerController. La replicazione multicast non esiste, ma si possono usare variabili replicate per trasmettere dati a tutti i giocatori. Per quanto riguarda la gestione dei fault, l'implementazione può prevedere server secondari di backup per garantire la continuità del servizio in caso di guasti che però per motivi di tempo e complessità non sono stati implementati nel progetto.

Utilizzo dei Blueprints in simbiosi con il codice sviluppato.

Dovendo descrivere brevemente i Blueprints direi che essi sono è un completo sistema di scripting del gameplay basato sul concetto di utilizzare un'interfaccia basata su nodi per creare elementi di gameplay all'interno dell'Unreal Editor. Come molti linguaggi di scripting comuni, è utilizzato per definire classi o oggetti orientati agli oggetti (OO) nel motore.

Questo sistema è estremamente flessibile e potente in quanto fornisce la possibilità ai designer di utilizzare praticamente l'intera gamma di concetti e strumenti generalmente disponibili solo ai programmatori. Inoltre, il markup specifico di Blueprint disponibile nell'implementazione C++ di Unreal Engine consente ai programmatori di creare sistemi di base che possono essere estesi dai designer.

Ciò che ci si aspetta dal loro utilizzo è una sinergia con il codice sviluppato in C++ in quanto a livello prestazionale i due operando insieme migliorano notevolmente le performance del progetto.

La tabella qui sotto fornisce un confronto su come vari aspetti sarebbero gestiti in UnrealScript (da Unreal Engine 3), C++ e Blueprints per aiutare coloro che fanno la transizione dalle versioni precedenti del motore e mostrare come il codice nativo e i Blueprints si confrontano.

Blueprints (UE4)	C++ (UE4)
Blueprint Asset	.h/.cpp files
UBlueprintGeneratedClass	UClass
ParentClass	: [ClassName]
Variable	UProperty()
Graphs/Events	UFunction()
Class Defaults	native constructor
Components List	native constructor

Fig.8 – Tabella di confronto tra implementazione Blueprints e Codice C++

Fatte queste premesse descrivo brevemente il collegamento del codice C++ ai Blueprints nel progetto in modo che questi possano essere utilizzati.

```
//UMultiplayerSessionsSubsystem.h

UFUNCTION(BlueprintCallable)
void CreateSession(FString ServerName);

UFUNCTION(BlueprintCallable)
void FindSession(FString ServerName);
```

Queste sono le due funzioni esposte tramite specificatore *UFUNCTION(BlueprintCallable)* che permette di richiamare queste funzioni dall'editor blueprints. Come si può vedere nella seguente figure:

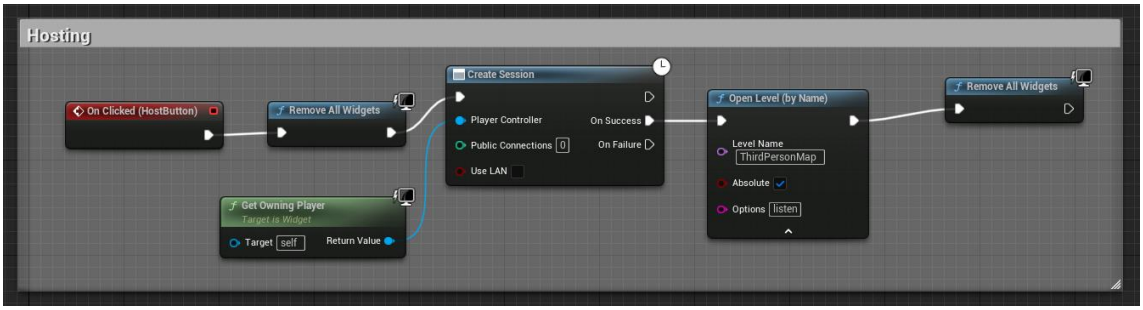

Fig. 9 – Blueprints per la sessione di hosting

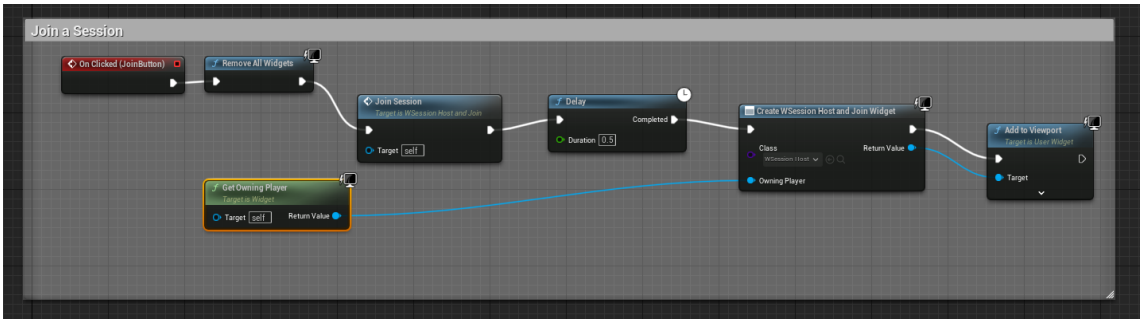

Fig. 10 – Blueprints per il join della sessione di gioco.

Queste rappresentano rispettivamente la creazione di una sessione di gioco da parte di un giocatore che risulterà come server e la partecipazione ad una sessione già esistente (in questo caso si utilizza la 127.0.0.1, dunque in locale).

Infine, il main menù che corrisponde a due semplici pulsanti (uno di host e uno di join) viene aggiunto nel level editor che corrisponde ad un nuovo ambiente blueprint dove si rappresenta solamente logiche da attuare in quel determinato livello. Apparendo come segue:

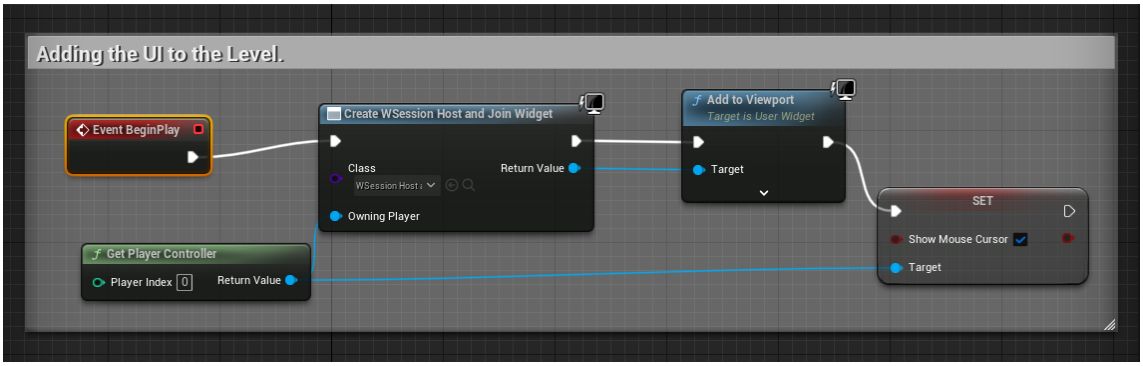

Fig. 11 – Blueprints per aggingere lo UI alla scena di gioco.

Tramite questo quando si avvia l'applicazione si aggiunge il widget del main menù e si aspetta che il giocatore compia una scelta. Hostare una sessione o partecipare ad una già esistente.

Integrazione dei Test in Unreal Engine 5.

Integrare i test in Unreal engine 5 non è una operazione banale. In esso, infatti, si possono implementare soltanto Functional Test o eventualmente Unit Test. Per fare ciò vi sono due metodi: l'utilizzo di nodi Blueprints o lo sviluppo di codice. Il primo è applicabile soltanto in un ambiente dello stesso tipo, cioè avendo sviluppato la funzionalità tramite dei Blueprints allora si può testare con altrettanti Blueprints, detto ciò il mio sistema ricade nel secondo caso. Di seguito si riporta un esempio di codice per la creazione di un Test (ulteriori test sono visibili nel sorgente MyFunctionalTest.cpp):

```
//MyFunctionalTest.cpp

void AMyFunctionalTest::TestMultiplayerConnection()
{
    if (TestCharacter->GetLocalRole() == ROLE_Authority
       || TestCharacter->GetLocalRole() == ROLE_AutonomousProxy)
    {
        UE_LOG(LogTemp, Warning,
      TEXT("Multiplayer connection successful."));
    }
    else
    {
        FinishTest(EFunctionalTestResult::Failed,
      TEXT("Multiplayer connection failed."));
    }
}
```

Descrivendolo brevemente ciò che esso fa è dato un character all'interno della scena di gioco, verificare il ruolo che Unreal Engine gli ha assegnato al momento della sua creazione, questo verificherà se effettivamente sia client o server.

Per andare a verificare che questo test funzioni, bisogna creare un blueprint che abbia come "parent class" il file contenente quest'ultimo e tutti i futuri test e inserirlo in

una mappa di test quanto più minimale possibile, risultando in questo modo:

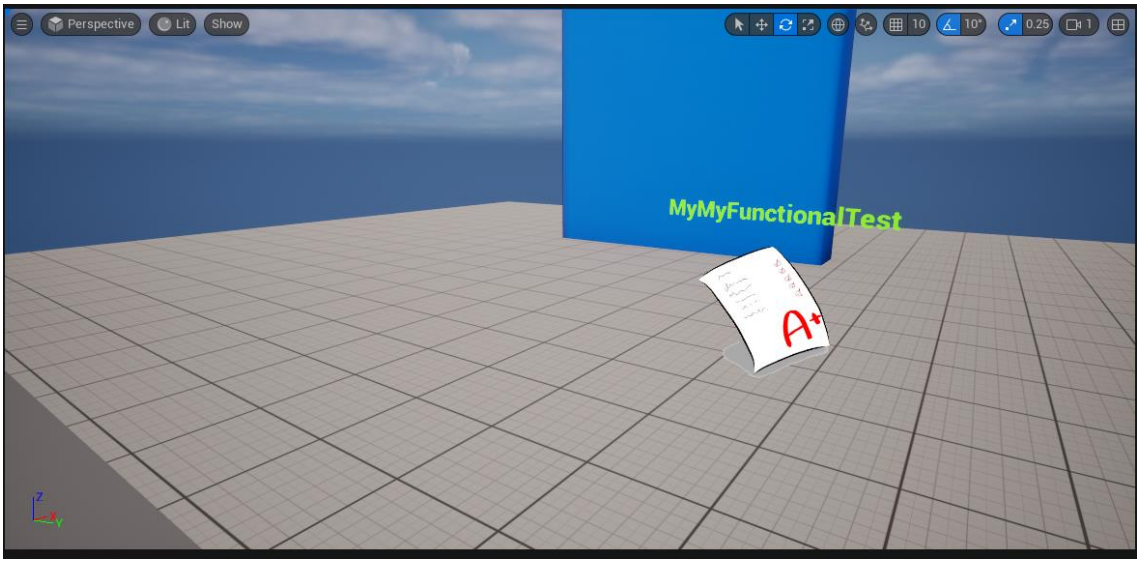

Fig. 12 – Scena di gioco in cui il Functional di Test è attivo.

A questo punto andando in Tools>Automation>Project> è possibile vedere le mappe di testing a disposizione e la classe di functional testing sviluppata. Eseguendo il run di questa classe Unreal Engine 5, creerà una sessione minimalistica tramite la mappa di testing precedente sviluppata e verifichera il comportamento dei test.

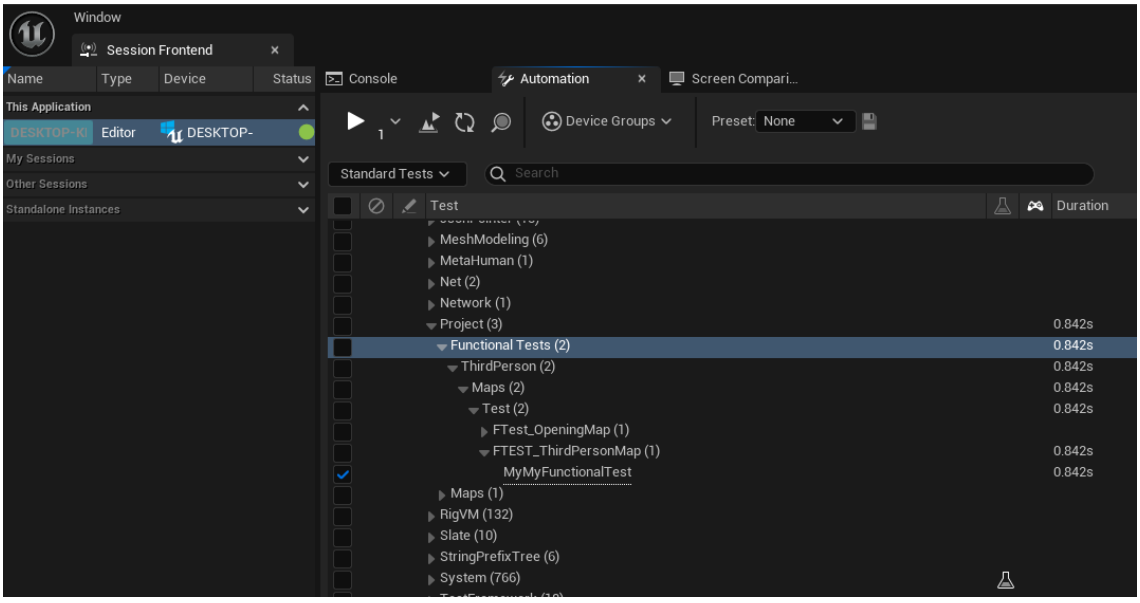

Fig. 13 – Finestra di sistema per l’esecuzione dei test.

Tramite la selezione della classe di Testing è possibile premendo il pulsante play in alto ricevere i log che accertano o meno se un test sia passato o meno.

Self-assessment

Per l’accertamento delle funzionalità principali del gioco si sono sviluppati dei Functional Test per rappresentare per l’appunto le funzionalità principali che il gioco deve assumere, di seguito viene riportato il risultato finale dei test:

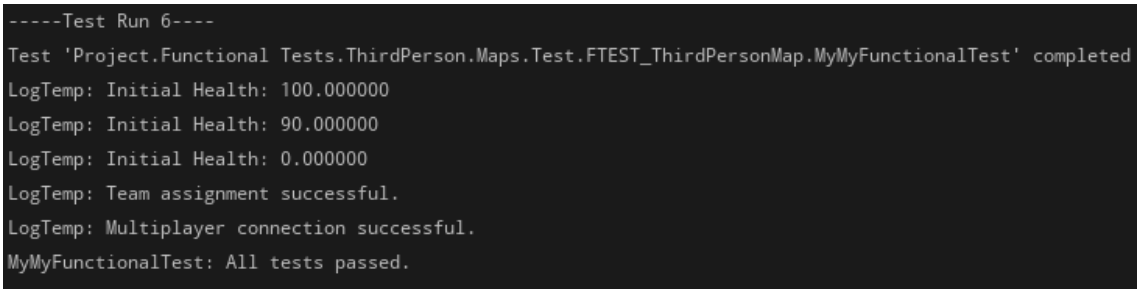

Fig. 13 – Schermata integrata in UE che mostra il risultato dei test effettuati

Tramite essi è sicuramente accertata l’usabilità del gioco e in parte la sua affidabilità. In quanto si verifica che il player inizi la partita con una vita, che possa prendere danni, che possa effettivamente morire, che gli venga assegnata una squadra e che venga stabilità una connessione al server e che gli venga assegnato un ruolo specifico.

Il criterio di valutazione del sistema conforme al requirements di affidabilità è identificabile in: completezza e coerenza dei dati replicati tra server e client. Si valuta se tutti i dati rilevanti per la giocabilità vengono replicati correttamente e se i cambiamenti apportati da un client o un server vengono riflesso accuratamente su tutti gli altri client. **Metriche:**

1. **Completezza dei dati replicati:** Valutare se tutte le informazioni critiche per il gameplay, come la salute del giocatore, vengono replicate correttamente tra server e client.
2. **Coerenza dei dati replicati:** Verificare se i cambiamenti apportati da un client vengono riflessi correttamente su tutti gli altri client senza perdita di dati o discrepanze.

Metodo di Valutazione:

- Esaminare i meccanismi di replica implementati nel software e confrontarli con i requisiti di replica specificati. Condurre test di gioco su una rete simulata per
- verificare la coerenza dei dati tra server e client durante l'interazione del giocatore con l'ambiente di gioco.

Il criterio risulta altamente conforme se tutti i dati rilevanti vengono replicati accuratamente e in modo coerente tra server e client. Altrimenti risulta bassa conformità se vi sono discrepanze o perdita di dati durante la replica dei dati critici per il gameplay. Questo è tutto per accertarsi l’affidabilità del sistema.

Mentre, per quanto riguarda la scalabilità questa viene accertata tramite l’avvio di molteplici standalone, che scelgono di essere client o server. In essi comunque ricade il peso di una connessione ad Unreal Engine 5, che ha comunque a livello di performance un contributo non indifferente. Questo per dire che l’accertamento di questo requisito è variabile a seconda dell’hardware di cui si dispone, dato che ogni nuovo client o server che viene istanziato avrà un peso che ricade sulla scheda grafica in quanto deve essere

fatto il loading di tutti gli assets di gioco e delle meccaniche che li coinvolgono, e successivamente sulla connessione stabilita. Dovendo citare il mio hardware attuale (che ritengo abbastanza vecchio) sono state fatte prove su una scheda video Nvidia GTX 2070 e il numero di standalone che essi siano indifferentemente client o server che si riesce a sopportare senza incombere in crash o rallentamenti è 10. Il sistema può essere ulteriormente espanso per sostenere molte più sessioni, ma quest'ultimo è il numero massimo raggiungibile dalla mia macchina.

Deployment Instructions

In questa sezione bisogna fare una premessa in quanto, ciò che viene rilasciato è uno standalone del gioco. Questo viene interamente prodotto da Unreal Engine 5 sulla base di ciò che viene inserito all'interno dell'engine dagli assets, alle classi ai vari file di configurazione e build interni. Al termine della procedura di deployment viene rilasciato una cartella contenente l'eseguibile del gioco (.exe) e altri file autogenerati. Dunque, si vuole evidenziare come lo standalone non presenti il codice sviluppato, ma per poter rieseguire uno standalone (intendo proprio una nuova cartella con un nuovo eseguibile al suo interno) è necessario riavere lo stesso stato dell'engine con cui ho dovuto lavorare io, per fare ciò bisogna:

1. Installare Epic Games Launcher
2. Una volta aperto creare un account epic games.
3. Andare nella sezione Unreal Engine del launcher (nella categoria Libreria) e installare la versione 5.3 di Unreal Engine.
4. Aspettare che il download finisca. (Peserà molti Giga, non è una applicazione indifferente)
5. A questo punto si ha UE5 installato sul sistema e aprendo il file che si trova sulla repo: "ThirdPersonMP.uproject" si aprirà da solo l'UE installato replicando interamente l'enviroment con cui ho lavorato io per sviluppare il gioco.
6. Avere installato sul sistema un qualsiasi IDE che supporti il linguaggio C++, io personalmente utilizzo Visual Studio Code, ma consiglio anche Visual Studio o JetBrains.
7. Inserire l'ide che si preferisce (va fatto dall'editor) Edit > Edit Preferences > Source code :

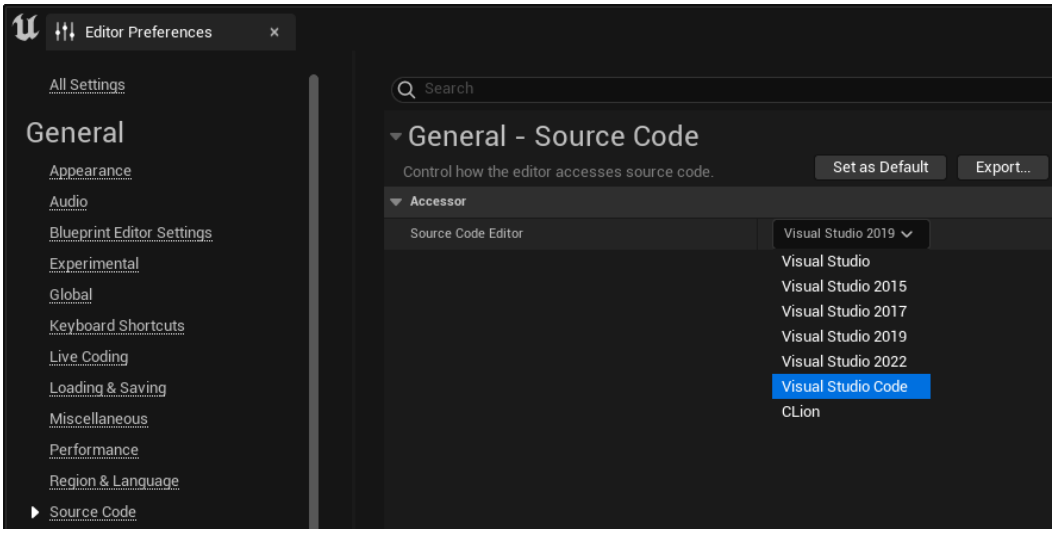

Fig. 14 – Vista delle impostazioni con modifica della preferenza dell'editor.

8. Per vedere le classi C++ sviluppate premere in basso a sinistra sulla cartella "C++ Classes" e al suo interno si troveranno degli oggetti che se premuti apriranno automaticamente l'ide (di default dovrebbe essere visual studio, ma è modificabile dalle impostazioni nell'editor) e visualizzeranno il codice relativo. Nota: lo stesso codice degli oggetti è quello che si può trovare nella cartella Source.

In caso si voglia solamente analizzare il codice sviluppato si può trovare tutto nel repo GitLab sotto la cartella Source. E' possibile aprire il repo in un IDE qualsiasi e navigare le classi, ma in questo caso non sarà possibile testare il diretta le sue funzionalità. Si può verificare il risultato del codice prodotto dallo standalone (.exe) rilasciato nello zip **Windows**, vedere la sezione seguente per capire come fare.

Usage Examples

Per eseguire lo standalone basta anche soltanto scaricare lo zip file Windows e non l'intera repo (e volendo nemmeno estrarlo, basta solamente eseguire il .exe) esso è eseguibile solamente su windows, in quanto con unreal engine si deve fare uno standalone diverso per Sistema Operativo o piattaforma/console cambiando le impostazioni direttamente nell'editor.

Cliccando due volte sull'eseguibile del gioco (.exe) si aprirà a schermo intero il gioco che dovrebbe apparire nel seguente modo:

Fig. 15 – Schermata iniziale di gioco.

Premendo il tasto Host Session si verrà teletrasportati nel livello di gioco come Server/host (in seguito spiegherò come si fa a capirlo) altrimenti premendo col mouse sul textbox e digitando 127.0.0.1 e solo dopo aver riempito il campo premendo il bottone Join Session si verrà teletrasportati nel livello come client. **Consiglio**: premere sempre Host Session per primo, perché se si fa Join Session senza un Host attivo lo standalone rimane in attesa finché non ne arriva uno (tutto ciò risulta in "non fa nulla").

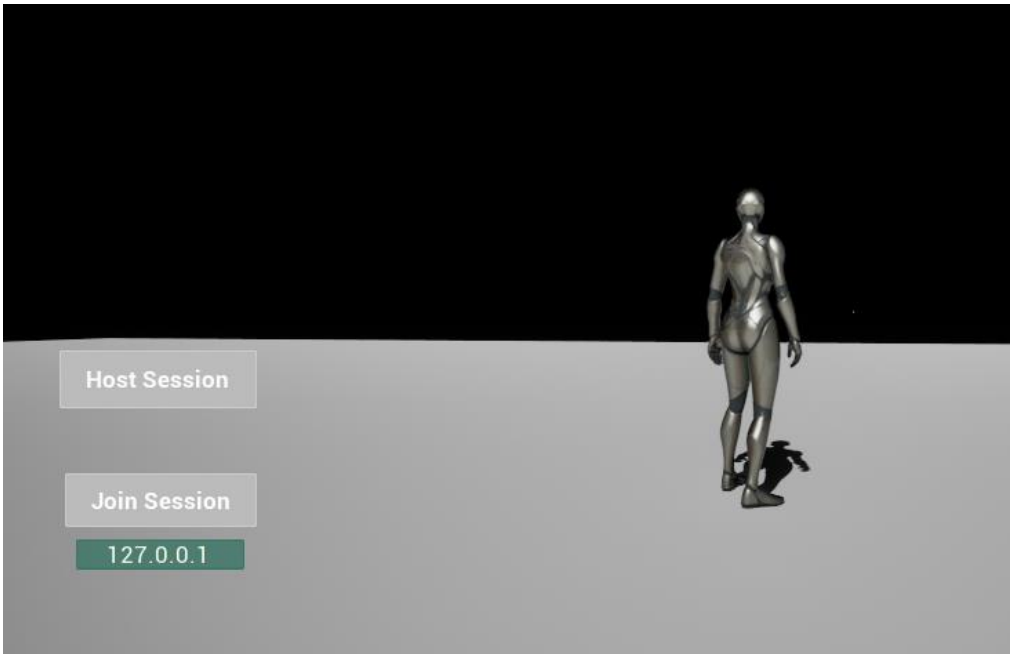

Fig. 16 – Riempimento dei campi per iniziare una sessione di gioco.

Al momento dello spawn del giocatore si assegna un colore randomico al mesh in modo da distringuere i Team (potrebbe capitare lo scenario dove tutti i giocatori hanno lo stesso colore, ma basterà uscire e rientrare nella sessione di gioco tramite il pulsante "quit" che spiegherò a breve)

Fig. 17 – Vista della mappa dove si svolge il gameplay.

A questo punto, **riaprire** (senza chiudere quella precedente) una nuova istanza di gioco ripremendo di nuovo sull'eseguibile (.exe) così si avranno due istanze dello stesso gioco, e questa volta entrare come client, dunque riempendo il campo con 127.0.0.1 e

premendo Join Session, si verrà teletrasportati nel livello di gioco e si avrà una nuova squadra assegnata sempre rappresentata dal colore.

Fig. 18 – Vista con due client della mappa di gioco.

Per far vedere che il gioco esegue con molteplici istanze in questo scenario ne ho 3 attive: 1 Server (Host) e 2 Client.

Fig.19 – Vista con tre client della mappa di gioco.

Aprendo una qualsiasi delle istanze di gioco (ovviamente siccome sono tutte aperte dalla stessa persona se ne controllerà una alla volta), è possibile: muoversi nella mappa, saltare con la barra spaziatrice e sparare proiettili con il left click del mouse. Sparando ad un giocatore (anche se stessi o gli alleati) con un proiettile si toglierà

10 di vita al giocatore colpito, ad ognuno di essi viene assegnato un valore iniziale di 100, quindi ci vorranno circa 10 colpi per eliminare un giocatore.

Fig. 20 – “Uccisione” di uno dei client da parte di un altro client.

Il gioco finisce quando tutti i giocatori della squadra nemica sono stati eliminati. (Termina semplicemente perché i giocatori spariscono dal livello, quando spariscono tutti "si ha vinto"). Quando un giocatore muore o in qualsiasi momento da vivo, è possibile premere il pulsante *"Q"* della testiera e comparirà un pulsante di Exit per ritornare al menu principale, facendo sì che si possa ripetere lo scenario di gioco appena descritto.

Fig. 21 – Pulsante di exit per uscire da una sessione.

Note importanti:

- Se un host muore non si chiude la sessione, i client continuano a giocare. Ma se un host effettua l'exit mentre dei client giocano, tutti vengono disconessi.
- Se un client si disconnette, solamente quel client sparirà dalla sessione e può rientrare nuovamente nella sessione già esistente rifacendo lo step di Join, oppure hostarne una nuova lui stesso.
- La morte dei giocatori non affligge le sessioni, soltanto il tasto quit.
- Una volta fatto partire il .exe non è stata implementata una chiusura del gioco (essendo a schermo intero), bisogna terminare il processo con ctrl+alt+del > gestione attività > selezionare i processi e fare termina attività.
- Nel gioco distribuito non ci saranno le scritte blu, poiché sono di debug e funzionano soltanto nell'engine. Ma il resto della logica rimane invariato.

Conclusion

In conclusione, ricapitolando ciò che è stato fatto:

- Installato Unreal Engine 5.3.
- Collegato il progetto al version controlling in gitLab.
- Creato un setup iniziale in C++ per il mio gioco basato su una visione in terza persona.
- Configurato il progetto per supportare il networking e la replicazione.
- Sviluppato nuove logiche di gioco.
- Adattato queste logiche ad un sistema multiplayer con replicazione.
- Assegnato un ruolo (autorizzazione) di esecuzione di certe funzionalità solo al server, e altre solo ai client tramite uso di RepNotifiers e RPC.
- Sviluppato un codice per le sessioni di gioco tramite programmazione asincrona.
- Collegato il codice alle funzionalità native (blueprints) di Unreal Engine.
- Realizzato dei Functional Test per veririficare le meccaniche principali.
- Simulato il gioco nell'editor per provare le funzionalità.
- Creato uno standalone del gioco utilizzabile da chiunque su windows.

Future works

Ovviamente il mio progetto risulta in un gioco vero e proprio, ma ha volutamente tralasciato ogni aspetto di design del gameplay. Sicuramente tra i future works ci sono l'estetica, l'aumento dell'usabilità e del senso di appagamento che il gioco in sè può offrire. Per quanto riguarda lato codice e distribuito, tuttò ciò che riguarda il multiplayer è stato inserito in questo progetto, essendo unreal engine un dominio

limitato di funzionalità, però sicuramente si può provare a sviluppare il gioco tramite un Server dedicato invece di un "listening server" come quello che unreal engine utilizza di default, inoltre per questioni di complessità ma soprattutto di tempo, sarebbe stato interessante utilizzare un sistema diverso di networking, come quello basato su Steam, ma questo vorrebbe dire cambiare notevolmente il codice sviluppato e alcuni file di configurazione di Unreal Engine.

What did we learned

Questo progetto è stato molto divertente da realizzare. Tutto ciò è stato possibile grazie al fatto che sono partito da una buona base di Unreal engine avendolo studiato prima di proporre questa specifica per 1 anno circa, poiché senza quella i tempi di realizzazione sarebbero stati biblici. Ciò che ho imparato in particolare da questo progetto è proprio l'integrazione del multiplayer in Unreal Engine. Gli esperti sviluppatori di videogiochi in UE consigliano sempre di partire da una soluzione multiplayer in quanto questa funziona anche se il gioco dovesse essere singleplayer, dicono ciò per via dell'anticipation of change dovuta al fatto che magari un giorno il proprio gioco dovrà diventare multiplayer per soddisfare la clientela, ma personalmente non mi ero mai cimentato. Da questo progetto ho imparato concetti importantissimi come: la replicazione, il concetto di server e client in unreal engine, la gestione dei ruoli, le RPC, l'utilizzo di programmazione asincrona in unreal engine. Ovviamente il progetto sviluppato è minimale, ma nel complesso mi reputo soddisfatto del lavoro prodotto.