

UNIVERSITÀ DEGLI STUDI DI SALERNO

Dipartimento di Informatica

Corso di Laurea Triennale in Informatica



Tesi Di Laurea Triennale In Informatica

Utilizzo di Photon e Unity per la realizzazione di un gioco Multiplayer

Relatore

Prof. Andrea F. Abate

Correlatore:

Prof. Ignazio Passero

Laureanda

Giusy Annunziata

Matricola: 0512105197

Anno Accademico 2019/2020

UNIVERSITÀ DEGLI STUDI DI SALERNO

Dipartimento di Informatica

Corso di Laurea Triennale in Informatica

Tesi Di Laurea Triennale In Informatica

Utilizzo di Photon e Unity per la realizzazione di un gioco Multiplayer

Relatore

Prof. Andrea F. Abate

Correlatore:

Prof. Ignazio Passero

Laureando

Giusy Annunziata

Matricola: 0512105197

Anno Accademico 2019/2020

Sommario

Lo scopo della tesi è la realizzazione di un gioco strategico in cui lo scorrere degli eventi e il flusso di gioco sono divisi in parti ben definite chiamate turni. In ogni turno un giocatore può effettuare un certo numero di mosse.

Più precisamente si intende realizzare il gioco da tavolo “Battaglia Navale”, o “BattleShip”, sfruttando le funzionalità messe a disposizione da Photon, una Platform Chat Service che è possibile integrare in Unity, che astrae il concetto di partita con la suddivisione in stanze per i vari Client basandosi sulla logica di comunicazione Client-Server.

La scelta è ricaduta su tale gioco poiché ci permette di sfruttare appieno le potenzialità di Photon, infatti, a differenza degli altri giochi da tavolo in cui si gestisce un'inquadratura a specchio per i due giocatori, nella Battaglia Navale i giocatori hanno due punti di vista simmetrici con dettagli che devono essere visibili ad un player ma non all'altro. Photon permette di utilizzare componenti come PhotonView tramite le quali è possibile gestire la visualizzazione e la sincronizzazione delle istanze dei vari Player all'interno di una scena.

La struttura di base di tale gioco è un oggetto Casella che tiene traccia sia di tutte le informazioni globali relative alla sua posizione nella scena, sia delle interazioni che i vari giocatori faranno su di essa.

Essendo C# un linguaggio Object-oriented ci permette di realizzare una matrice formata dagli oggetti Casella che costituirà la principale struttura dati del gioco.

Implementando IPunObservable tale struttura dati verrà sincronizzata per entrambi i giocatori ad ogni mossa che effettueranno.

Il progetto di tesi è stato un utile esperimento di utilizzo della comunicazione multiplayer e l'applicazione realizzata si è mostrata di facile utilizzo e di piacevole fruizione.

Indice

Capitolo 1	1
BattleShip.....	1
1.1 Cenni Storici	1
1.2 Descrizione del Gioco	1
1.2.1 Svolgimento del gioco	2
1.2.2 Varianti del Gioco.....	2
1.3 Scelta del gioco.....	3
1.3.1 Gioco Proposto.....	4
1.4 Sviluppi già noti	5
Capitolo 2.....	6
Tecnologie Utilizzate.....	6
2.1 Introduzione a Unity.....	6
2.1.1 Perché è stato scelto Unity?.....	6
2.2 Componenti Unity.....	7
2.2.1 GameObject	7
2.2.2 I components e lo scripting	8
2.2.3 Sistema di Illuminazione	9
2.2.4 GUI ed elementi dell'interfaccia.....	9
2.2.5 Prefab	9
2.2.6 Gerarchia	10
2.3 Introduzione a Photon.....	11
2.3.2 Photon Unity Network (PUN).....	12
2.3.2 Perché è stato scelto Photon?	13
2.4 Componenti Photon.....	14
2.4.1 RPC	14
2.4.2 Client	15
2.4.3 PhotonView.....	15

Capitolo 3	17
Sistema Proposto	17
3.1 Struttura Dati	17
3.1.1 Casella	17
3.1.2 Tavola	19
3.1.3 Creazione Tavole	20
3.2 Scene di Gioco	21
3.2.1 HomeMenu	21
3.2.2 Room for 1	23
3.2.3 Room for 2	23
3.3 Avvio della Connessione.....	25
 Capitolo 4.....	 27
Player e Sincronizzazione.....	27
4.1 Gestione dei Player	27
4.2 Gestione della Sincronizzazione	29
4.3 Gestione dei Turni.....	32
4.4. User Interface	33
 Capitolo 5.....	 35
Conclusioni e Sviluppi Futuri	35
5.1 Come giocare a BattleShip	35
5.1.1 Design del Gioco.....	35
5.1.2 Prima fase del gioco	35
5.1.3 Seconda fase del gioco	36
5.2 Conclusioni.....	38
5.3 Sviluppi Futuri.....	38

Codici Citati.....	39
Casella	39
GameController	45
GameManager.....	50
Launcher	52
PlayerManager.....	54
PlayerUI	56
Tavola.....	57
TextUI.....	59

Bibliografia.....	61
--------------------------	-----------

Ringraziamenti	63
-----------------------------	-----------

Capitolo 1

BattleShip

1.1 Cenni Storici

BattleShip è conosciuto in tutto il mondo come un gioco con carta e matita che risale alla Prima guerra mondiale. Si pensa che il gioco di BattleShip abbia le sue origini nel gioco francese “L'Attaque” giocato durante la Prima guerra mondiale ma si dice anche che il gioco sia stato giocato dai russi nel periodo precedente alla Prima guerra mondiale.

La prima versione commerciale del gioco fu The BattleShip Game, pubblicata nel 1931 negli Stati Uniti dalla compagnia Starex.

Nel 1967 fu introdotta una versione del gioco che utilizzava tavole e navi di plastica, dunque iniziava ad essere molto più simile alla versione del gioco che conosciamo noi oggi.

1.2 Descrizione del Gioco

“BattleShip”, conosciuto anche come “Battaglia Navale” o “Sea Battle” è un gioco da tavolo di tipo strategico in cui due giocatori si sfidano.

La piattaforma di gioco è costituita da quattro griglie di carta o cartone, due per ogni giocatore e tipicamente sono quadrate, 10×10 ed i singoli quadrati sono identificati da lettere e numeri.

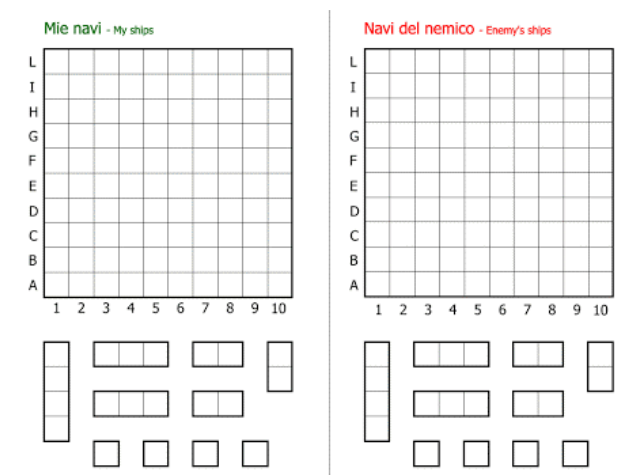


Figura 1.1: Le due griglie di gioco realizzate su carta

1.2.1 Svolgimento del gioco

Prima dell'inizio della partita, ogni giocatore organizza segretamente le navi sulla propria griglia principale. Il numero di quadrati per ogni nave è determinato dal suo tipo.

Dopo che le navi sono state posizionate il gioco procede con una serie di round o turni. In ogni turno un giocatore annuncia una casella nella griglia dell'avversario che deve essere colpita. L'avversario risponde indicando se la casella è occupata o meno da una nave, in caso la casella è occupata dalla nave, tale nave viene colpita ed eventualmente affondata. Sia il giocatore attaccante che il giocatore attaccato segneranno la mossa di attacco effettuata e se ha avuto successo o meno.

L'obiettivo del gioco è affondare tutta la flotta avversaria.

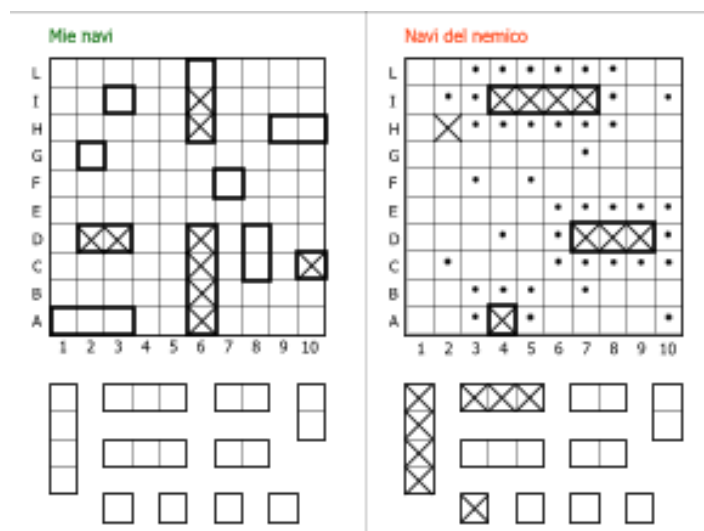


Figura 1.2: Le due griglie di gioco nel mezzo di una partita

1.2.2 Varianti del Gioco

Dalla versione classica sono state realizzate alcune varianti, come ad esempio nell'edizione Salvo del 1931, i giocatori prendono di mira un numero specifico di caselle per turno e tutte le caselle vengono attaccate contemporaneamente. In altre varianti di tale versione il numero di caselle viene fissato a 5 per tutto il gioco e l'avversario può annunciare le navi "colpite" e i colpi "mancati" dandone una definizione numerica, senza specificare in quali caselle tali azioni siano avvenute, lasciando che l'attaccante calcoli le conseguenze del suo attacco.

In un'ulteriore variante di BattleShip i giocatori possono rifiutarsi di annunciare se la nave colpita sia stata affondata, così che l'avversario ignaro di ciò continui a colpire nella medesima area.

1.3 Scelta del gioco

Ho scelto di implementare come gioco Multiplayer digitale BattleShip per via di alcune sue particolarità che lo differenziano dai classici giochi da tavolo come ad esempio la dama e gli scacchi.

In primis possiamo notare che nei giochi tradizionali i “pezzi di gioco” o “pedine” sono ben visibili, garantendo al giocatore tutto il materiale per poter realizzare una strategia per battere l'avversario.

Nella BattleShip, invece, ad ogni giocatore le navi avversarie vengono nascoste, dunque studiare una strategia per affondare tutte le navi nemiche e vincere così la partita risulta più complesso.

Possiamo intuire che non sarà facile realizzare tale meccanica di gioco poiché bisogna fare in modo che ciò che viene visualizzato da un Player sia nascosto all'altro, e viceversa.

Un'altra particolarità sta nel punto di vista dei vari giocatori, a differenze dei classici giochi da tavolo in cui i due avversari hanno un punto di vista a specchio.

BattleShip presenta un punto di vista simmetrico dove entrambi i giocatori effettuano le modifiche alle proprie griglie in merito agli attacchi che effettuano e alle risposte che ricevono dall'avversario.

Non si tratta più di muovere una pedina e far in modo che tale movimento venga visto da entrambi i giocatori, ma bensì di simulare la stessa azione in contesti e su griglie differenti.

Rendiamolo più chiaro con un esempio.

Nella dama il giocatore 1 muove una pedina e il giocatore 2 segue dall'inizio alla fine il movimento fatto dal suo avversario, invece, nella BattleShip la situazione è completamente diversa.

Il giocatore 1 sceglie la casella da attaccare e il giocatore 2 comunica se in tale casella è stata posizionata o meno da lui una nave, in tal caso la nave viene colpita ed eventualmente affondata.

Il giocatore 1 segnerà sulla sua griglia di gioco che ha colpito/affondato una nave nemica mentre il giocatore 2 segnerà sulla sua prima griglia che la sua nave è stata colpita/affondata.

Nel caso in cui non vi era nessuna nave posizionata, entrambi i giocatori segneranno nelle rispettive griglie che il colpo è andato a vuoto.

Non vi è un'interazione diretta, ma ognuno vede solo le proprie due griglie, facendo in modo che vengano modificate simultaneamente e che tutte e quattro le griglie rispecchino lo stato globale della partita senza errori o variazioni.

1.3.1 Gioco Proposto

La versione da me proposta di BattleShip presenta quattro tavole 10x10, due per ogni Player.

Essendo una versione digitale e interattiva ho omesso i numeri e le lettere che identificano ogni singola casella, poiché tali informazioni vengono trattate in background e ai giocatori non serve pronunciare il numero della casella per attaccare poiché basterà un semplice click del mouse.

Il flusso di gioco è diviso in due parti. Si ha una prima parte di “Posizionamento Navi” o “Preparazione del Gioco” in cui ogni giocatore con un click del mouse potrà posizionare un totale di 10 navi sulla tavola che troverà sulla sinistra, una volta completata questa fase comparirà il pulsante “Gioca”.

Dopo aver premuto tale pulsante si dovrà attendere che anche l’altro giocatore sia pronto a giocare e successivamente verrà avviato il gioco. Da questo momento in poi saranno sempre visibili 2 tavole, la tavola sinistra mostrerà al giocatore le navi che egli ha posizionato e le varie azioni dell’avversario, mentre, la destra sarà la tavola su cui dovrà giocare e scegliere le caselle dell’avversario da attaccare.

Per rendere tutto più fluido e intuitivo ho scelto di utilizzare solo navi da una singola casella e sull’interfaccia dell’utente verranno sempre segnalati in alto i nomi dei due Player, sulla sinistra sarà sempre mostrato quello del Player1, mentre sulla destra quello del Player2 e sotto di essi troveremo delle Slider bar che indicheranno il livello di navi ancora in gioco, inizialmente esse saranno piene e diminuiranno gradualmente per ogni nave affondata. Sarà sempre possibile essere a conoscenza del turno poiché verrà segnato in alto il numero del turno attuale e il Player a cui spetta attaccare in esso.

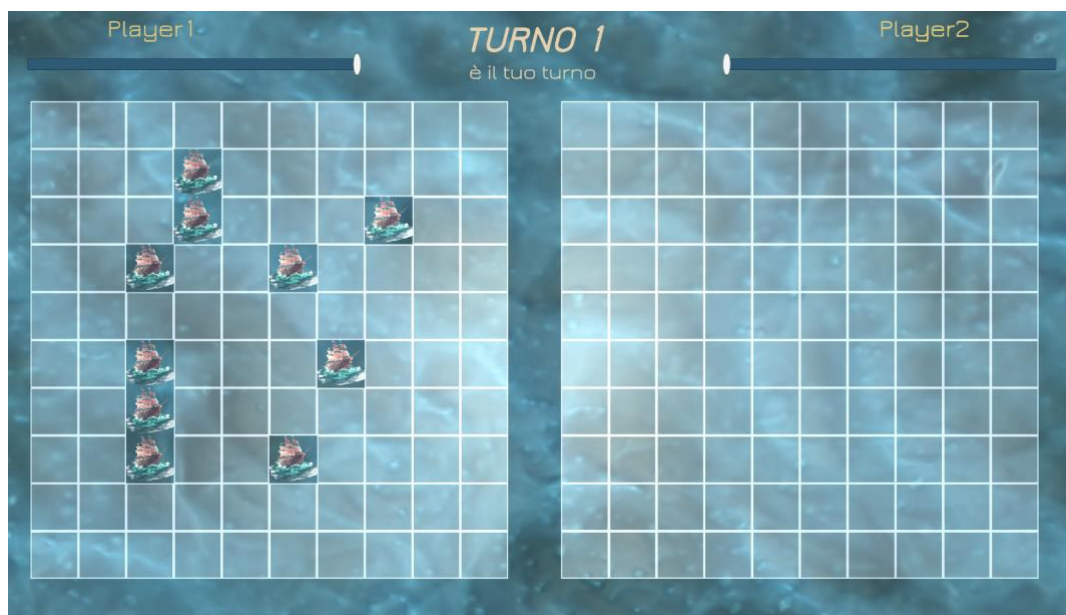


Figura 1.3 : Visuale di Gioco di un Player dell'applicazione BattleShip realizzata

1.4 Sviluppi già noti

BattleShip è stato uno dei primi giochi ad essere prodotto per computer, nel 1979. Oltre a tale versione n'è stata realizzata una digitale anche per altre piattaforme quali Nintendo DS dove BattleShip è noto come Grid Attack. Viene giocato su una griglia 8x8 includendo leggere variazioni come il game play per 4 giocatori e varie dimensioni e forme di navi.

Un ulteriore versione digitale del gioco è stata realizzata anche per piattaforme come PlayStation 2, Wii e Xbox 360 in cui però sono state alterate le regole inclusa la dimensione della griglia, che da una griglia 8x8 passa ad essere una griglia 8x12, e le dimensioni delle navi.

Capitolo 2

Tecnologie Utilizzate

Il capitolo seguente fornisce una descrizione accurata delle tecnologie che sono state utilizzate per lo sviluppo del gioco Multiplayer.

2.1 Introduzione a Unity

Il lavoro di tesi proposto è stato sviluppato in Unity3D (abbreviato in Unity). Unity è un motore grafico multiplatforma, sviluppato da Unity Technologies, e consente lo sviluppo di videogiochi e contenuti interattivi quali visualizzazioni architettoniche o animazioni 3D in tempo reale.

Fu annunciato per la prima volta e rilasciato nel giugno 2005 alla Worldwide Developers Conference di Apple Inc. come motore grafico esclusivo per Mac OS X.

A partire dal 2018 è stato esteso per supportare più di 25 piattaforme. Unity, progettato inizialmente per il settore videoludico, è un ambiente di sviluppo che con il tempo e con costanti aggiornamenti è stato adottato anche in altri ambiti come quello cinematografico, automobilistico e medico.

Il grande punto di forza dell'ambiente è sicuramente il *work flow* molto semplice, infatti, è adatto per tutte le tipologie di team, sia piccoli che grandi. All'interno di Unity possiamo trovare tutti i principali *tool* necessari alla creazione di un videogioco come gestione dei materiali, *animation tool*, gestione di luci ecc...

2.1.1 Perché è stato scelto Unity?

Unity è stato scelto per i seguenti motivi:

- È gratis e non è necessario acquistare alcuna licenza per progetti non commerciali;
- È disponibile una documentazione online ufficiale ben fornita e molti tutorial che trattano le più svariate funzionalità, dalle più basiche a quelle più avanzate.
- Possiede una community molto attiva e presente, infatti, sono facilmente reperibili soluzioni a problemi già sviscerati e risolti dalla community del software.
- In Unity è integrato l'Asset Store, ovvero il negozio ufficiale online del motore grafico, dove è possibile ottenere (gratuitamente o pagando) contenuti digitali di qualsiasi tipo.
- Programmare in Unity3D è molto intuitivo per chi ha un minimo di esperienza con i linguaggi di programmazione Java e C++, infatti, lo Scripting in Unity3D è basato su C#, la cui sintassi e struttura prendono spunto proprio da tali linguaggi.

Unity3D è un motore grafico che, pur offrendo uno svariato numero di funzionalità, rimane estremamente semplice ed intuitivo da utilizzare.

2.2 Componenti Unity

2.2.1 GameObject

I GameObjects sono gli elementi fondamentali in Unity ne consegue che quasi tutto ciò che viene visualizzato in un programma, sviluppato con Unity, è un GameObject.

Quindi ogni oggetto di scena (un personaggio, un'entità, la videocamera che inquadra la scena, la GUI o un semplice testo) che può essere posizionato nello spazio 3D è un GameObject con un nome e alcune proprietà di base e può essere posizionato nello spazio 3D.

Il GameObject può essere inteso come un “contenitore” vuoto all'interno del quale l'utente può inserire dei componenti (in inglese components) che lo caratterizzano e ne implementano le funzionalità reali. Ad esempio, è possibile creare una cubo al quale è possibile associare uno *script* che permette di variarne l'orientamento nel tempo, un colore o *texture*.

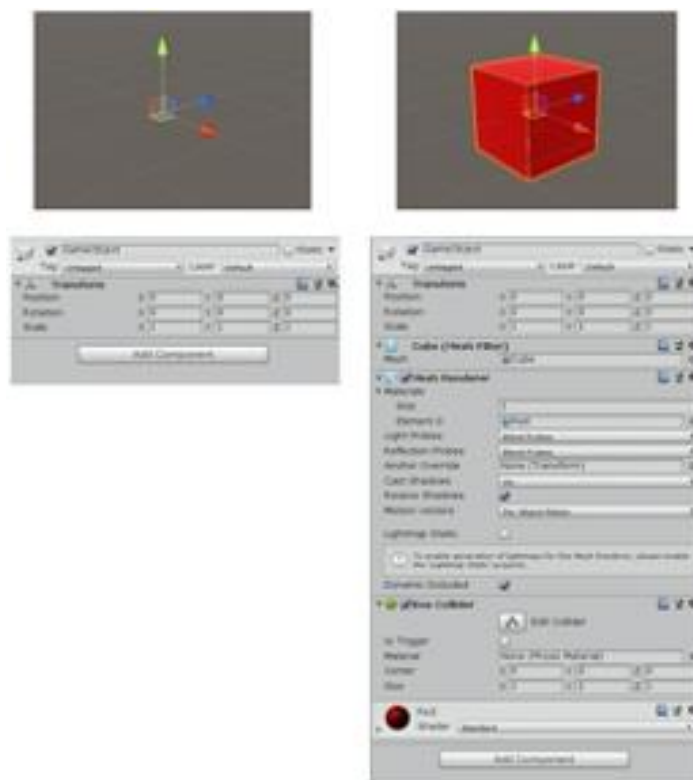


Figura 2.1: A sinistra un GameObject che ha solamente il Transform component, a destra lo stesso GameObject a cui sono stati aggiunti i componenti Cube (Mesh Filter), Mesh Renderer, Box Collider e Material Red.

GameObjects più sofisticati possono essere realizzati tramite un programma di modellazione 3D apposito, esterno a Unity, e successivamente importati.

I componenti fondamentali che ogni GameObject deve possedere sono:

- posizione all'interno del sistema di coordinate;
- orientamento rispetto a tale sistema;
- dimensioni.

È possibile associarvi, inoltre, altri componenti quali script, effetti sonori, proprietà fisiche (*rigidbody*), effetti speciali (*special effects*), animazioni, sistemi di collisione (impongo che il GameObject sia un *collider*) e proprietà di visualizzazione dell'oggetto (*mesh renderer*). I componenti possono essere aggiunti mediante l'interfaccia offerta da Unity o attraverso uno *script* e possono essere eventualmente modificati.

2.2.2 I Components e lo Scripting

Un GameObject è quindi un contenitore per diversi components che ne definiscono il comportamento. Ogni component può essere modificato, disattivato o rimosso tramite l'Editor di sviluppo o tramite script.

Dall'Editor di Unity selezionando un GameObject è possibile visualizzarne i components ed interagire con essi.

Ogni component, infatti, ha una serie di parametri ai quali sono associati dei valori modificabili.

Cambiare tali valori ha degli effetti immediati sulle proprietà del component.

È possibile agire su tali valori anche tramite script con le medesime possibilità offerte dall'Editor.

Allo stesso modo rimuovere o disattivare temporaneamente un oggetto di scena è un'operazione eseguibile tramite l'Editor di sviluppo o tramite script.

Una volta rimosso un GameObject esso non esisterà più nella scena, mentre, se viene disattivato esso sarà invisibile ma sempre presente nella scena e una volta tornato visibile ripristinerà tutte le funzionalità dei propri components.

Un altro strumento per poter fornire proprietà e caratteristiche ai GameObjects, oltre i components già presenti in Unity, sono gli scripts.

Lo Scripting (o la creazione di scripts) consiste nel programmare, tramite l'API di Unity Scripting, le proprie aggiunte alle funzionalità dell'Editor di Unity.

I components degli oggetti non consentono di interagire con gli input dati da un utente o con un sistema di eventi, ecco perché ricorriamo all'uso degli script.

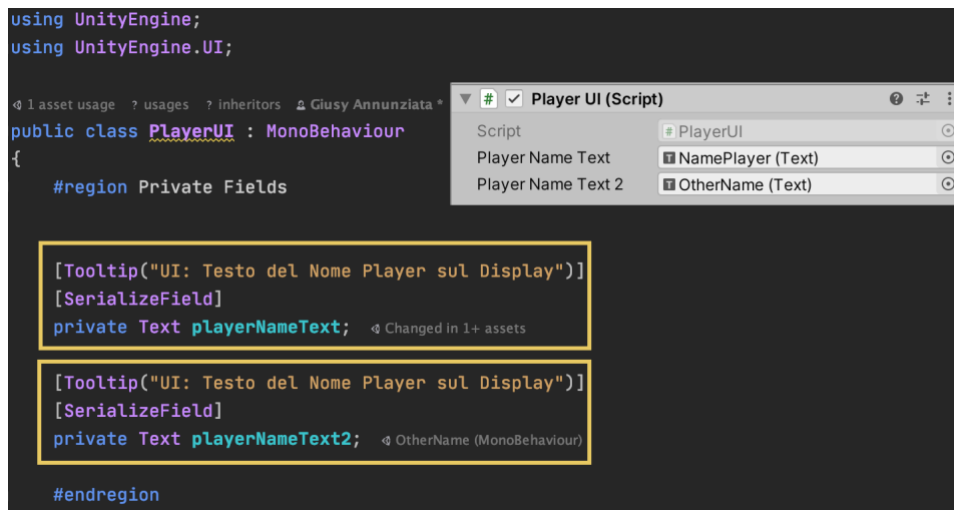


Figura 2.2: Un frammento dello script PlayerUI. Associando tale script ad un GameObject è possibile vederne l'interfaccia da component. Da notare che le variabili dello script compaiono come parametri del component.

2.2.3 Sistema di Illuminazione

La luce, GameObject alla quale è associato il componente *light*, è una parte fondamentale di ogni scena. Il sistema di illuminazione contribuisce a conferire atmosfera all'ambiente creando profondità ed immersione. Unity offre diversi tipi di luci: puntiformi, direzionali, riflettori e luci di ambienti esterni. Sono di fatto gestite come normali GameObjects con proprietà quali raggio, angolo di illuminazione, colore, intensità, tipo di ombra ecc...

2.2.4 GUI ed elementi dell'interfaccia

All'interno della propria applicazione è possibile creare un'interfaccia grafica dell'utente (*Graphic User Interface* - GUI). Essa può essere personalizzata attraverso i GUI Text (per testi in 2D), GUI Texture (icone dove cliccare), 3D Text (per scritte in 3D) o usando la UnityGUI, che consente di creare tramite *scripting* una varietà di GUI (come pulsanti, finestre, barre di scorrimento, inputbox etc.)

2.2.5 Prefab

Il sistema dei Prefab di Unity permette di creare, configurare ed archiviare un GameObject completo, cioè munito di tutti i components, come risorsa riutilizzabile. Una volta creato un Prefab, a partire da un GameObject, funge da modello da cui è possibile creare nuovi oggetti duplicati.

In altri termini, quando si è intenzionati a riutilizzare un GameObject con una configurazione specifica in più punti della scena è buona norma convertirlo in un Prefab.

Ogni modifica apportata al Prefab si riflette su tutte le sue istanze, così è possibile apportare facilmente ampie modifiche in tutto il progetto senza dover ripetere i cambiamenti per ogni copia del Prefab.

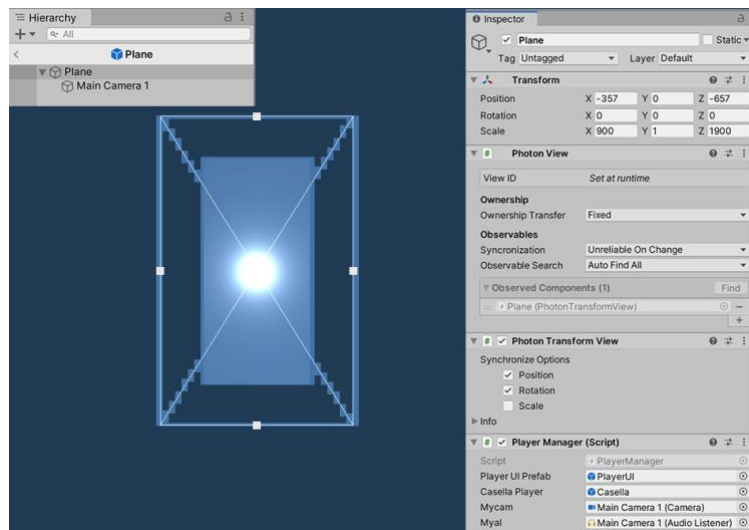


Figura 2.3: Il Prefab di un piano di gioco. A sinistra è mostrata la gerarchia degli oggetti figli, mentre a destra ci sono i components.

2.2.6 Gerarchia

Tra i GameObject di Unity esiste il concetto di gerarchia. Un GameObject può avere vari GameObject al suo interno che verranno definiti “figli”, ma un figlio può appartenere ad un solo GameObject “padre”. Un GameObject figlio può essere padre di un altro GameObject. I figli ereditano le caratteristiche del padre e possono aggiungerne di proprie. È possibile modificare la posizione di un GameObject padre, e tale modifica si rifletterà su tutti gli oggetti facenti parte della sotto gerarchia di tale GameObject, ma ovviamente non vale il viceversa, ovvero le modifiche dei figli non si rifletteranno sul padre.

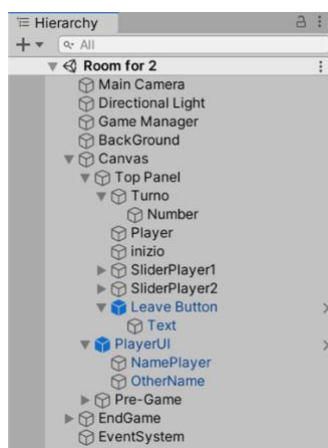


Figura 2.4: Rappresentazione di più gerarchie in Unity

2.3 Introduzione a Photon

Photon Engine, conosciuto come Photon, è una piattaforma multiplayer ampiamente conosciuta che offre agli utenti la possibilità di utilizzare un'area di realtà virtuale. Photon detiene il primo posto nei giochi Unity ed è ampiamente conosciuto come il software multiplayer numero uno.

È composto da 4 prodotti e opzioni per lo sviluppo in Multiplayer:

- Realtime;
- PUN (Photon Unity Network);
- Bolt;
- Quantum.

Contiene anche prodotti per lo sviluppo di chat e mezzi di comunicazione da integrare come:

- Voice;
- Video;
- Chat.



Figura 2.5: Struttura di Photon

Il Realtime offre l'incredibile possibilità di goderti i tuoi giochi preferiti insieme ad altri giocatori in tempo reale. È una piattaforma che offre agli sviluppatori la possibilità di creare e modificare funzionalità all'interno di tutti i tipi di videogiochi.

Questo prodotto è adatto sia per team di sviluppatori di giochi che per un programmatore autonomo.

Quantum è un motore perfetto per creare giochi d'azione. È ottimizzato per le prestazioni, è facile da usare, esporta la navigazione da un prodotto all'altro e possiede un motore fisico 2D.

Photon è dotato di Chat, una funzionalità messaggistica che integra facilmente nelle app un sistema di chat multiplatforma capace di scalare in base alla quantità di chat simultanee ed è dotato di una tecnologia di filtro volgarità di livello mondiale.

Voce è una chat Vocale 3D per Unity Games, VR / AR / MR.

Photon Voice è l'add-on ideale per qualsiasi applicazione e perfettamente adatto per app VR o AR in cui una chat vocale è la scelta migliore per comunicare con altri utenti. Molto importante è SDK Client, un Server multiplatforma per backend personalizzati self-hosted. Utilizza i protocolli UDP / TCP snelli e veloci che utilizzano una larghezza di banda ridotta e consentono una serializzazione ultrarapida. Photon Server risulta così essere il motore multiplayer ideale per qualsiasi tipo di gioco.

2.3.2 Photon Unity Network (PUN)

Photon Unity Networking (PUN) è un pacchetto Unity per giochi multiplayer in cui grazie al matchmaking flessibile è possibile portare i vari giocatori in stanze in cui gli oggetti possono essere sincronizzati sulla rete.

La comunicazione è veloce, affidabile e viene effettuata tramite server Photon dedicati. Il pacchetto PUN racchiude tre livelli di API:

- Il livello più alto è il codice PUN, che implementa funzionalità specifiche di Unity come oggetti in rete e RPC;
- Il secondo livello contiene la logica per lavorare con i server Photon, fare matchmaking e callback;
- Il livello più basso è costituito dai file DLL, che contengono la de / serializzazione e i protocolli.

PUN approfitta di una stretta integrazione con Unity per sviluppare e lanciare facilmente giochi multiplayer esportabili su tutte le piattaforme supportate da Unity, comprese le console.

PUN risulta essere una solida base per qualsiasi tipo di gioco multiplayer basato su una stanza in Unity 3D poiché permette al programmatore di concentrarsi sul gioco mentre PUN si occupa del backend in merito alla comunicazione con il Server online. I giochi realizzati con PUN sono ospitati dal sistema di Photon Cloud distribuito a livello globale che si occupa dell'hosting del servizio, delle operazioni, del ridimensionamento e garantisce una bassa latenza e tempi di risposta brevi.

Photon risulta essere Scalabile, affidabile e sempre connesso; basato su un'architettura client to server, risulta essere la soluzione più stabile per i giochi multiplayer, poiché in altre circostanze i client spesso non possono connettersi a causa di problemi di NAT punch-through.

Questo problema è ancora peggiore nelle reti mobili nelle quali Photon Cloud fa in modo che i giocatori restino sempre connessi grazie ad un continuo monitoraggio del server.

2.3.2 Perché è stato scelto Photon?

Anche Unity Presenta un sistema Client/Server come quello di Photon, infatti, possiamo notare che sia Unity che PUN hanno API di basso livello simili tra loro, ma l'architettura richiesta per utilizzare queste API è il fattore chiave di differenza tra di loro.

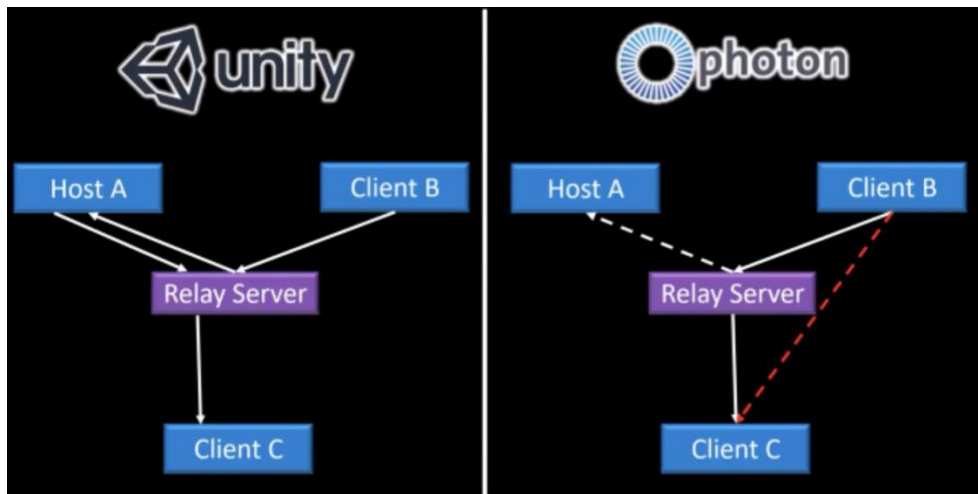


Figura 2.6: L'immagine sulla sinistra rappresenta l'architettura di Unity, mentre quella sulla destra raffigura l'architettura di Photon.

Unity Networking supporta un'architettura Client/Server in cui tutti i messaggi devono passare attraverso il client host e non possono essere inviati direttamente tra i nodi. Ad esempio, in base allo schema sovrastante, i messaggi vengono inviati dal client B al client C utilizzando il seguente percorso:

client B -> Relay Server -> Host A -> Relay Server -> client C

richiedendo così un totale di 4 "salti" per arrivare al destinatario e nel caso l'host viene disconnesso dalla rete il gioco si interrompe.

PUN ha un'architettura Client/Server simile ma supporta anche l'invio di messaggi peer-to-peer. Infatti, notiamo che i messaggi vengono trasmessi dal client B al client C utilizzando il percorso:

client B -> Relay Server -> client C.

Si tratta di un totale di 2 hop rispetto ai 4 di Unity per lo stesso trasferimento di messaggi tra due nodi. Oltre a ciò, PUN potrebbe potenzialmente bypassare completamente il server di inoltro e permettere al client B di comunicare direttamente con il client C, riducendo così i salti a 1.

Un'altra differenza sta nei costi, Unity offre differenti piani per la realizzazione di giochi Multiplayer con differente numero di utenti simultanei (CCU) per ciascuna licenza:

- Personal: 20 utenti simultanei
- Plus: 50 utenti simultanei
- Professional: 200 utenti simultanei

Se si necessita di aumentare il numero di CCU all'interno del gioco bisognerà pagare per la larghezza di banda aggiuntiva.

PUN, invece, fornisce anche fino a 20 CCU, 8000 attività mensili e 500 messaggi per stanza gratuitamente. Oltre al piano gratuito, offre un'opzione a pagamento di una tantum di \$ 95 per 60 mesi, che include 100 CCU, 40.000 attività mensili e 500 messaggi per stanza. Questa opzione è ottima per i piccoli sviluppatori indipendenti che hanno un budget limitato. Inoltre, Photon Unity Network è integrato in Unity, quindi è scaricabile gratuitamente dall'Asset Store. Essendo molto conosciuto PUN è fornito di una Community in cui è possibile confrontarsi con altri utenti o chiedere aiuto per un qualsiasi problema.

2.4 Componenti Photon

2.4.1 RPC

Una caratteristica che distingue PUN dagli altri pacchetti Photon è il supporto delle "Remote Procedure Calls" (RPC).

Tali chiamate di procedura remota sono esattamente ciò che il nome indica, ovvero chiamate di metodo su client remoti nella stessa stanza.

Grazie ad uno specifico `RpcTarget` possiamo indicare se al momento dell'invocazione della chiamata al metodo tale funzione dev'essere eseguita dal Client che la richiama o da alcuni o da tutti i Client presenti nella stanza.

Per abilitare la chiamata remota per qualche metodo è necessario applicare l'attributo `[PunRPC]`, ma per chiamare tale metodo è necessario che ciò venga fatto tramite una componente `PhotonView`.

```
//Il Player 1 Affonda una Nave del Player2
[PunRPC]
public void plaffondanavedip2(int riga, int colonna)
{
    if (PhotonNetwork.IsMasterClient)
    {
        Tavola.table2Player1[riga,colonna].AffondaP2();
        Tavola.table2Player1[riga,colonna].ColpisciP2();
    }
    else
    {
        Tavola.table1Player2[riga,colonna].AffondaP2();
        Tavola.table1Player2[riga,colonna].ColpisciP2();
    }
}

//devo effettuare le modifiche per il numero di navi e il turno
this.pv.RPC("plaffondanavedip2",RpcTarget.All,riga,colonna);
```

Figura 2.7: Un frammento dello script Casella. Viene dichiarato il metodo `p1affondanavep2` come `PunRPC`. Successivamente quel metodo viene invocato, utilizzando come `RpcTarget` `All`, riferito a tutti i Client nella stanza.

2.4.2 Client

In un gioco realizzato con PUN è molto importante il concetto dei Client.

Ogni giocatore, infatti, può essere definito come un Client che viene trasportato in una stanza da Photon (Server).

Solitamente il primo giocatore a connettersi, colui per cui Photon realizzerà una stanza virtuale, sarà etichettato come “MasterClient” e fungerà da host. Qualora tale giocatore, in un momento qualsiasi del gioco, decida di abbandonare la stanza il gioco non verrà chiuso e la stanza non verrà eliminata, bensì il titolo di MasterClient passerà al secondo giocatore entrato nella stanza.

Il programmatore può, inoltre, decidere un limite massimo di persone che potranno accedere ad una stanza ponendo un vincolo di Max Player for Room, ma ciò non impedisce ad altri giocatori di giocare, infatti, sarà compito di Photon creare un’altra stanza virtuale per gli ulteriori giocatori.

2.4.3 PhotonView

Un PhotonView identifica un oggetto attraverso la rete (viewID) e configura il modo in cui il client di controllo aggiorna le istanze remote. Essa può essere considerata come una componente da inserire all’interno di un GameObject di Unity ed è essenziale per la sincronizzazione degli oggetti in un videogioco Multiplayer.

Se un oggetto che contiene una PhotonView viene istanziato da un Player, tale Player risulterà essere il “Proprietario” di quel GameObject, dunque, gli altri Player presenti nella stanza non potranno vedere tale oggetto o le modifiche che gli verranno apportate se il programmatore decide di non renderle visibili.

Ciò è gestibile estendendo l’interfaccia `IPunObservable`.

Il Player che controlla un GameObject dotato di PhotonView resterà sempre in possesso di tale controllo a meno che:

- il proprietario è nullo, in tal caso il GameObject viene istanziato alla creazione della scena e non tramite Player, quindi il MasterClient ne prenderà il controllo
- il proprietario si disconnette dalla stanza, infatti, è possibile che un Player lasci temporaneamente la stanza e vi si unisca di nuovo in seguito. Mentre il proprietario è disconnesso in modo soft, il MasterClient diventa proprietario di quel GameObject e quando il proprietario si riconnetterà, ne riprenderà il controllo.

Capitolo 3

Sistema Proposto

Nel capitolo 1 ho parlato delle regole e della logica generale del gioco BattleShip, in particolare mi sono soffermata a spiegare la mia proposta di gioco nel paragrafo 1.3.1 in cui ne ho elencato le caratteristiche e le regole di gioco.

In questo capitolo andrò a spiegare come tale proposta è stata implementata, quali scelte sono state fatte e quali componenti sono state utilizzate per la realizzazione dell'elaborato.

3.1 Struttura Dati

Nel paragrafo 1.3 vengono evidenziate le caratteristiche principali in merito a BattleShip. Per far sì che il gioco funzioni correttamente abbiamo bisogno che i giocatori non vengano a conoscenza della posizione delle navi avversarie ma tale posizione deve essere nota alla tavola su cui il giocatore sta attaccando.

La struttura dati da utilizzare dev'essere efficiente e tener traccia di tutto quello che succede sulla tavola di un giocatore e rispecchiarlo anche su quella dell'avversario "filtrando" però l'informazione in modo da mostrare solo il necessario.

Essendo C# un linguaggio Object-oriented esso ci permette di realizzare oggetti e di utilizzarli come una struttura dati.

Ripensando al gioco tradizionale vediamo che ogni giocatore ha dinanzi a sé 2 griglie con 8x8 caselle ognuna, per un totale di 4 griglie per una partita.

Rispettando il gioco cartaceo la mia scelta è stata di utilizzare come struttura dati una classe contenente 4 matrici 10x10 composte da oggetti Casella rappresentanti le singole caselle ed i loro relativi stati in base alle azioni dei vari Player.

3.1.1 Casella

La classe Casella costituisce l'unità principale della struttura dati utilizzata.

Consiste in uno Script C# integrato all'interno del Prefab Casella che ne gestisce le varie modifiche in merito alle informazioni che contiene. Per rendere visibili a tutti i Client nella stanza le varie modifiche il Prefab Casella è dotato della componente PhotonView.



Figura 3.1 : Prefab Casella e la componente Script Casella con tutte le informazioni

```
public class Casella : MonoBehaviourPunCallbacks, IPunObservable
{
    #region Public Fields
        public int table;  < Unchanged
        public int riga = 0;  < Unchanged
        public int colonna = 0;  < Unchanged
        public bool naveposizionataP1;  < Unchanged
        public bool colpitaP1;  < Unchanged
        public bool naveposizionataP2;  < Unchanged
        public bool colpitaP2;  < Unchanged
        public bool p2affondaP1;  < Unchanged
        public bool p1affondaP2;  < Unchanged
        public int player;  < Unchanged

        private PhotonView pv;
    #endregion
}
```

Figura 3.2 : Script Casella e la dichiarazione delle variabili


Come possiamo vedere nelle figure all'interno di un oggetto casella vi sono informazioni in merito:

- Posizione della Casella:
 - *Table*: identifica la tavola di appartenenza della Casella;
 - *Riga*: numero di riga;
 - *Colonna*: numero colonna;
 - *Player*: identificativo del Player che possiede tale casella.
- Azioni compiute dai Player:
 - *naveposizionataP1*: booleano che indica la presenza o meno di una nave del Player1 sulla casella;
 - *naveposizionataP2*: booleano che indica la presenza o meno di una nave del Player2 sulla casella;

- *colpitaP1*: Booleano indicante se il Player2 ha colpito tale casella del Player1;
- *colpitaP2*: Booleano indicante se il Player1 ha colpito tale casella del Player 2;
- *p1affondap2*: Booleano indicante se il Player1 ha affondato una nave del Player2;
- *p2affondap1*: Booleano indicante se il Player2 ha affondato una nave del Player1.

3.1.2 Tavola

La classe Tavola è uno Script, presente nell'Asset di Unity, che non è associato a nessun GameObject ma in essa vi troviamo le principali strutture dati del gioco, ossia, quattro matrici di tipo Casella, due per ogni Player.



```
public class Tavola : MonoBehaviour, IPunObservable
{
    #region Public Fields

    //Dichiaro le 4 tavole di gioco
    public static Casella[,] table1Player1 = new Casella[11,11];
    public static Casella[,] table2Player1 = new Casella[11,11];
    public static Casella[,] table1Player2 = new Casella[11,11];
    public static Casella[,] table2Player2 = new Casella[11,11];

    #endregion
}
```

Figura 3.3 : Script Tavola in cui vengono dichiarate le 4 matrici di tipo Casella

Secondo la logica del gioco ad ogni giocatore sono affidate due tavole, Table 1 e 2 rispettivamente.

La prima tavola viene utilizzata per posizionare le proprie navi e verrà aggiornata mostrando quali caselle sono state colpite dall'avversario. La seconda tavola, invece, viene utilizzata per selezionare le caselle da colpire dell'avversario nel tentativo di affondare le sue navi.

Tale logica ci porta a pensare che la Table1 di ogni Player corrisponderà alla Table2 del suo avversario.

3.1.3 Creazione Tavole

Come abbiamo detto nel paragrafo precedente le informazioni contenute nella Tavola1 di un Player e quella della Tavola2 del suo avversario saranno equivalenti, e ciò potrebbe portarci a pensare che sarebbe ottimale ridurre le 4 tavole a due.

La scelta di utilizzare ben 4 tavole ricade sul riferimento agli oggetti che ognuna di essa contiene.



```
//Istanziamo la Prima Tabella di Gioco
for (x = 1; x < 11; x++)
{
    for (y = 1; y < 11; y++)
    {
        casellaPlayer = PhotonNetwork.Instantiate( prefabName: Path.Combine("PhotonPrefabs", "Casella"),
            position: new Vector3( x: -500 + (x * 90), y: 1, z: -1000 + (y * 90)), Quaternion.identity, group: 0);

        setCasella(casellaPlayer, x, y, numtable: 1);

        Tavola.table1Player1[x, y] = casellaPlayer.GetComponent<Casella>();
    }
}

? usages ? overrides ? ext methods ? exposing APIs
public void setCasella (GameObject casellaPlayer, int x, int y, int numtable)
{
    casellaPlayer.GetComponent<Casella>().SetRiga(x);
    casellaPlayer.GetComponent<Casella>().SetColonna(y);
    casellaPlayer.GetComponent<Casella>().SetTable(numtable);
    casellaPlayer.GetComponent<Casella>().SetPlayerTable( num: 1);
}
```

Figura 3.4: Script PlayerManager in cui vengono istanziati i singoli Prefab Casella

Nella figura 3.4 vediamo che al momento dell'istanziamento di un Prefab Casella, tramite il comando `GetComponent<Casella>`, estraiamo dal Prefab lo Script.

Tramite la funzione `setCasella` vengono inserite, in tale Script, le informazioni:

- Riga in cui è posizionata;
- Colonna in cui è posizionata;
- Tavola in cui è posizionata;
- Player che la istanzia e a cui appartiene.

In questo modo possiamo vedere che all'interno di ogni matrice, definita nella classe Tavola, avremo uno Script corrispondente ad un Prefab Casella istanziato nella scena. Avere tale corrispondenza è utile per ogni modifica da effettuare, poiché tramite script possiamo facilmente risalire al GameObject a cui appartiene, senza doverne necessariamente salvare la posizione all'interno di una scena.

3.2 Scene di Gioco

Le scene in Unity contengono gli oggetti del gioco. Possono essere utilizzate per creare menu principale, livelli individuali e tanto altro. Identifichiamo ogni scena come un livello unico. È possibile avere all'interno di un unico progetto Unity più Scene di gioco.

In ogni scena sarà possibile inserire GameObject che realizzano ambienti, ostacoli e decorazioni, progettando e costruendo il gioco in pezzi.

Alla creazione di un nuovo progetto Unity verrà mostrata una scena senza titolo e non salvata. La scena sarà vuota tranne che per gli oggetti di default: una luce direzionale e una telecamera ortografica o prospettica, a seconda che il progetto sia stato avviato in modalità 2D o 3D.

Per la realizzazione del gioco sono state utilizzate 3 Scene:

- HomeMenu;
- Room for 1;
- Room for 2.

3.2.1 HomeMenu

HomeMenu è la Scena di base, quella che ci viene presentata all'avvio del gioco.

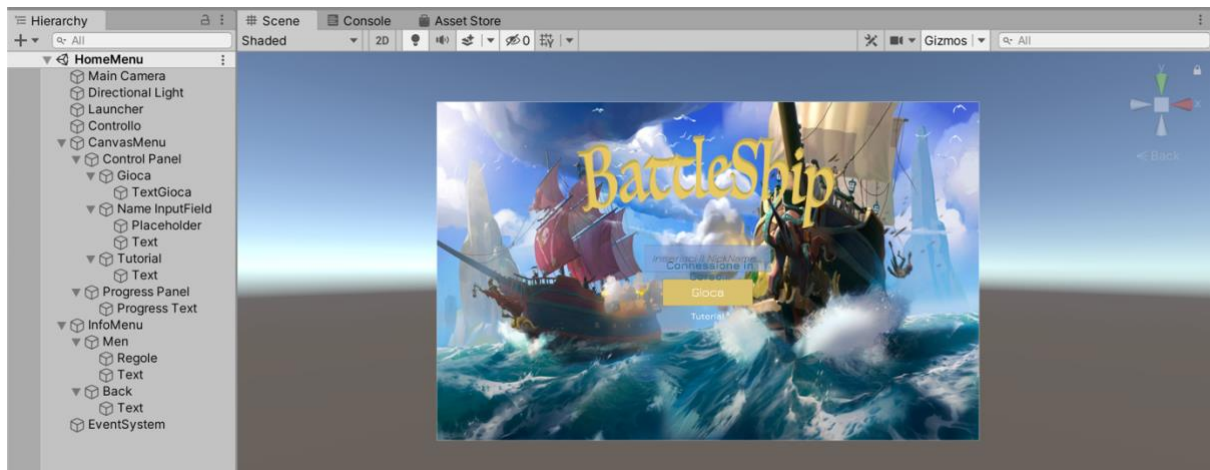


Figura 3.5 : Scena HomeMenu, sulla sinistra è possibile vederne la gerarchia.

Gli unici GameObject presenti nella scena sono Launcher e Controllo, entrambi Empty GameObject. Launcher avrà un'unica componente, lo Script Launcher, incaricato di connettere il giocatore con il Server. Controllo, invece, grazie lo script ControlloMenu consentirà al Giocatore di poter visualizzare la schermata Tutorial.

La maggioranza dei GameObject si concentra nella User Interface di Canvas Menu. Possiamo individuare tre differenti Menù:

- Control Panel;
- Progress Panel;
- Info Menu.

Control Panel permette al giocatore di visualizzare una schermata costituita da un InputField (in cui potrà inserire un nome), un pulsante Gioca e una voce che porterà l'utente a visionare il Tutorial.

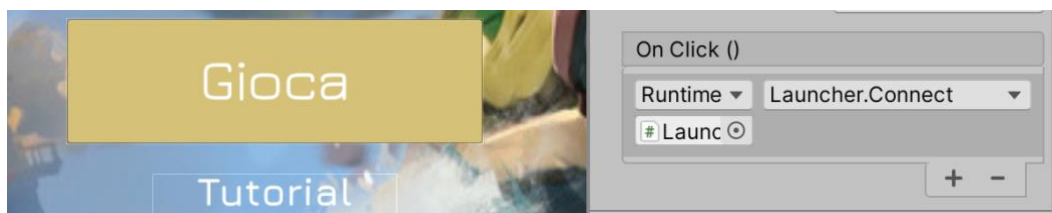


Figura 3.6 : Pulsante Gioca e Tutorial, sulla sinistra la funzione che verrà lanciata al OnClick del pulsante.

Progress Panel è il pannello che mostrerà una schermata con scritto “Caricamento in Corso ...” e verrà visionato subito dopo aver cliccato “Gioca”. In questa fase Photon sta controllando la connessione e cercando o creando una stanza.

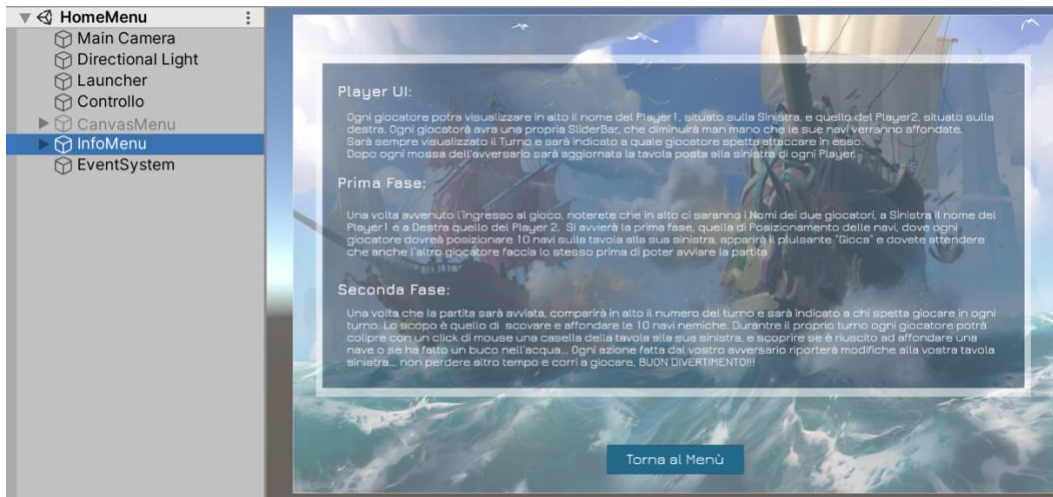


Figura 3.7: Pannello InfoMenu

Info Menu è il pannello che mostrerà al giocatore le regole del gioco e contiene un pulsante che riporterà in qualsiasi momento il giocatore a visualizzare il Control Panel.

3.2.2 Room for 1

Ho scelto di utilizzare Room for 1 per distinguere le fasi di gioco. Come sappiamo BattleShip è un gioco per due giocatori, dunque non ha molto senso iniziare a mostrare la schermata di gioco ad un giocatore che è ancora in attesa del suo avversario.

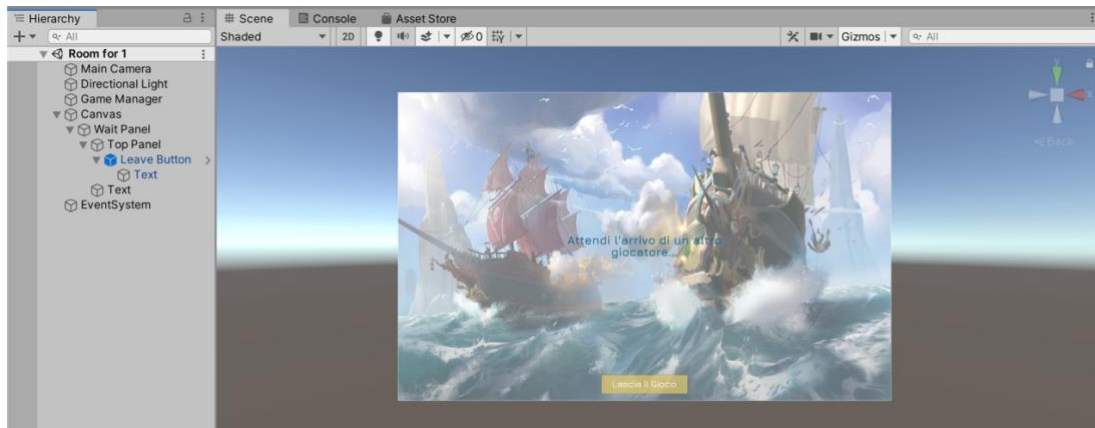


Figura 3.8 : Room for 1 con la sua gerarchia.

La scena Room for 1 è un “intrattenimento” per il singolo giocatore in cui gli viene comunicato che il suo avversario ancora non è connesso.

Se l’attesa si fa troppo lunga, il giocatore può scegliere di abbandonare la schermata in qualsiasi momento grazie al Prefab Leave Button. Il pulsante è stato realizzato come Prefab per evitarne inutili repliche in altre scene.

3.2.3 Room for 2

La scena Room for 2 è dove si svolgerà il gioco.

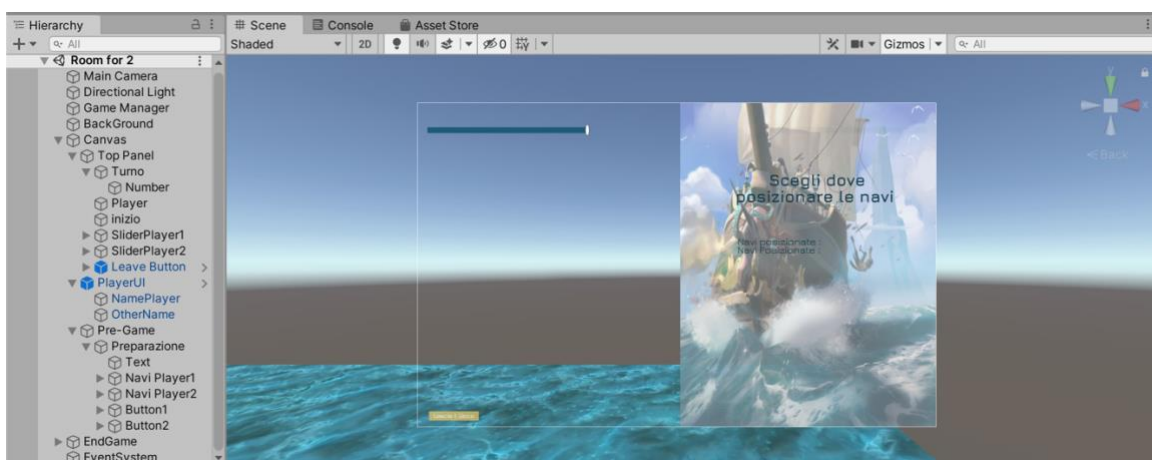


Figura 3.9 : Room for 2 con apposita gerarchia

Il principale GameObject presente nella scena è Background, un grande pannello con Texture marina, sopra il quale verranno istanziate le tavole di gioco in modo da dare la sensazione, all'utente, di una vera battaglia navale.

GameManager è un Empty GameObject contenente lo Script di controllo del gioco con tutte le informazioni principali circa le istanze dei giocatori entrati all'interno della Scena.

Anche in questa scena possiamo notare una ricca User Interface nuovamente suddivisa in 3 differenti menu:

- Top Panel;
- PlayerUI;
- Pre-Game.

Top Panel è il pannello che mostrerà al giocatore le informazioni sul turno attuale e la Slider Bar in merito al numero di navi affondate.

Contiene, inoltre, il Prefab Leave Game per dare la possibilità ai giocatori di lasciare il gioco in qualsiasi momento.

PlayerUI è un Prefab che contiene i Nickname dei due Player collegati. In alto a sinistra quello del Player1 mentre a destra quello del Player2.



Figura 3.10 : Slider Bar e Nickname del Player sulla sinistra per il Player1 e sulla destra per il Player2.

Al centro le informazioni in merito al turno

Pre-Game è un pannello che verrà visualizzato solo su metà schermo per la prima fase del gioco. Inizialmente i giocatori dovranno posizionare sulla loro tavola sinistra le 10 navi.

La loro tavola destra sarà coperta da tale pannello dove verranno contate e visualizzate le navi posizionate dal Player, ed una volta arrivate a 10, comparirà il pulsante "Gioca" che consentirà ai giocatori di dare inizio alla partita e al pannello Pre-Game di scomparire.

3.3 Avvio della Connessione

Nello Script Launcher è stata realizzata la connessione al Server Photon Cloud per far sì che un Player al momento del “Log in” venga portato in una stanza già presente o se necessario in una nuova stanza creata da Photon.

```
asset usage ? usages ? overrides ? ext methods ? exposing APIs
public void Connect()
{
    if (PhotonNetwork.IsConnected)
    {
        //Tentiamo di entrare in una stanza casuale,
        //se falliamo riceviamo una notifica in OnJoinRandomFailed() e ne creiamo una
        PhotonNetwork.JoinRandomRoom();
    }
    else
    {
        //dobbiamo connetterci a Photon Online Server.
        isConnecting = PhotonNetwork.ConnectUsingSettings();
        PhotonNetwork.GameVersion = gameVersion;
    }

    controlPanel.SetActive(false);
    progressLabel.SetActive(true);
}
```

Figura 3.11: Launcher Script metodo Connect

Come abbiamo visto nella figura 3.6, associato al pulsante “Gioca” vi è associata la principale funzione della classe Launcher, “Connect”, grazie alla quale il giocatore viene connesso al Server Photon.

Se il giocatore era già connesso verrà inserito in una stanza tramite la funzione JoinRandomRoom().

Se il giocatore non era ancora connesso al Server verrà creata una connessione tramite ConnectUsingSettings() che rappresenta il punto di partenza per connettersi a Photon Cloud.

Lo Script Launcher estenderà la classe MonoBehaviourPunCallbacks poiché espone proprietà specifiche con metodi virtuali utilizzabili e sovrascrivibili.

Tra i vari metodi di Callbacks troviamo:

- IConnectionCallbacks: richiamate relative alla connessione;
- IInRoomCallbacks: richiami che avvengono all'interno della stanza;
- ILobbyCallbacks: richiamate relative alla lobby;
- IPunObservable: Callback di serializzazione PhotonView.

```

? usages ? overrides ? ext methods ? exposing APIs
public override void OnJoinRandomFailed(short returnCode, string message)
{
    //base.OnJoinRandomFailed(returnCode, message);
    Debug.Log( message: "PUN/Launcher: OnJoinRandomFailed() è stata chiamata dal PUN, non vi erano stanze disponibili," +
        " ne abbiamo creata un'altra chiamando PhotonNetwork.CreateRoom");
    //non siamo riusciti ad unirci ad una stanza, forse non ne esisteva una oppure erano piene. Ne creiamo una
    PhotonNetwork.CreateRoom( roomName: null, new RoomOptions { MaxPlayers = maxPlayerForRoom });
}

? usages ? overrides ? ext methods ? exposing APIs
public override void OnJoinedRoom()
{
    //base.OnJoinedRoom();
    Debug.Log( message: "PUN/Launcher: OnJoinedRoom() è stata chiamata dal PUN. ora questo client è nella Room");
    Debug.Log( message: "Ci Siamo Uniti alla stanza Room For 1");
    //carichiamo la stanza
    PhotonNetwork.LoadLevel("Room for 1");
}

```

Figura 3.12: Launcher Script, metodi OnJoinedRandomFailed e OnJoinedRoom

Le funzioni Callbacks realizzate vengono utilizzate per istruire Photon sulla distribuzione dei Player nelle stanze virtuali e sulle scene da mostrare loro.

Con OnJoinedRoom indichiamo in quale stanza Photon deve mandare il nostro Player e la scena da mostrargli. Ricordiamo che nel Paragrafo 3.2 abbiamo illustrato la suddivisione delle scene. Tale suddivisione semplifica il lavoro di Photon, poiché quando ci sarà un solo Player all'interno della connessione verrà indirizzato nella stanza che mostrerà Room for 1. Nel momento in cui un secondo Player effettuerà l'accesso entrambi i Player verranno portati nella stanza che mostrerà loro Room for 2. Dato il vincolo di MaxPlayerForRoom, tale stanza virtuale risulterà piena.

Se un terzo Player proverà ad effettuare l'accesso verrà invocato il metodo OnJoinRandomFailed dove viene specificato che quando l'ingresso di un Player all'interno di una stanza Photon fallisce, Photon creerà un'altra stanza virtuale per tale Player.

Nel nostro caso egli si ritroverà in una stanza che mostrerà la scena Room for 1. Possiamo notare che nella creazione della stanza tramite il metodo CreateRoom compare la variabile maxPlayerForRoom in cui bisogna specificare l'intero che indica il numero massimo di Player all'interno della stanza che verrà creata.

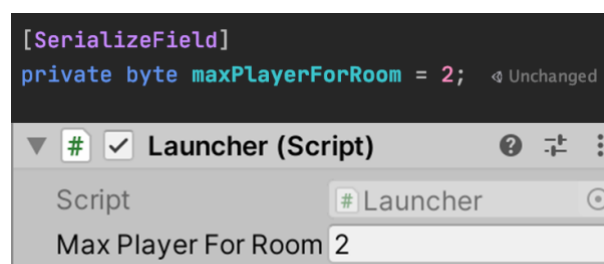


Figura 3.13: Launcher Script, maxPlayerForRoom

È molto importante definire un variabile per maxPlayerForRoom. Tale valore può essere deciso al momento di creazione di una stanza ma dichiarandolo all'interno dello Script, esso comparirà come valore anche all'interno della Component dell'oggetto che contiene lo Script Launcher e in questo modo sarà possibile, in qualsiasi momento, modificare il numero massimo di giocatori.

Capitolo 4

Player e Sincronizzazione

4.1 Gestione dei Player

Nel paragrafo 3.3 abbiamo parlato della connessione al gioco e di cosa avviene quando un Player effettua il “Log in”. Una volta entrato nella stanza sarà lo script GameManager a prendere il controllo del gioco.

```
#region Photon Callbacks
/// <summary>
/// vengono chiamate quando il giocatore locale lascia la stanza. Dobbiamo caricare la scena Luncher.
/// </summary>
? usages ? overrides ? ext methods ? exposing APIs
public override void OnLeftRoom()
{
    //base.OnLeftRoom();
    SceneManager.LoadScene(0);
}

? usages ? overrides ? ext methods ? exposing APIs
public override void OnPlayerEnteredRoom(Player other)
{
    //base.OnPlayerEnteredRoom(other);
    Debug.LogFormat("OnPlayerEnteredRoom() {0} ", other.NickName);

    if(PhotonNetwork.IsMasterClient)
    {
        Debug.LogFormat("OnPlayerEnteredRoom IsMasterClient {0}", PhotonNetwork.IsMasterClient);
        LoadArena();
    }
}

? usages ? overrides ? ext methods ? exposing APIs
public override void OnPlayerLeftRoom(Player otherPlayer)
{
    //base.OnPlayerLeftRoom(otherPlayer);
    Debug.LogFormat("OnPlayerLeftRoom() {0}", otherPlayer.NickName);
    if (PhotonNetwork.IsMasterClient)
    {
        Debug.LogFormat("OnPlayerLeftRoom IsMasterClient {0}", PhotonNetwork.IsMasterClient);
        LoadArena();
    }
}
}
#endregion
```

Figura 4.1: Chiamate ai metodi PunCallbacks in GameManager

```
? usages ? overrides ? ext methods ? exposing APIs
void LoadArena()
{
    if (!PhotonNetwork.IsMasterClient)
    {
        Debug.LogError( message: "PhotonNetwork: Ho provato ad entrare in un livello ma non sono il MasterClient ");
    }
    Debug.LogFormat("PhotonNetwork: Livello Caricato : {0}", PhotonNetwork.CurrentRoom.PlayerCount);
    PhotonNetwork.LoadLevel("Room for " + PhotonNetwork.CurrentRoom.PlayerCount);
}
}
```

Figura 4.2: GameManager metodo LoadArena.

GameManager si occupa dell'entrata e dell'uscita dei Player all'interno della stanza tramite l'override dei metodi PUN OnPlayerEnteredRoom e OnPlayerLeftRoom.

Tali metodi richiameranno LoadArena che caricherà il livello di gioco mostrando la Scena Room for 2.

```
// Start is called before the first frame update
// Event function  ? usages  ? overrides  ? ext methods  ? exposing APIs
void Start()
{
    Debug.LogFormat("We are Instantiating LocalPlayer from {0}", SceneManagerHelper.ActiveSceneName);

    #region Istantiate Player
    if (PhotonNetwork.IsMasterClient)
    {
        // siamo in una stanza. genera un personaggio per il giocatore locale.
        // viene sincronizzato utilizzando PhotonNetwork.Instantiate
        playerPrefab = PhotonNetwork.Instantiate( prefabName: Path.Combine("PhotonPrefabs", "Plane"),
            position: new Vector3( x: 0f, y: 0f, z: 0f), Quaternion.identity, group: 0);
        DontDestroyOnLoad(playerPrefab);
    }
    else
    {
        // siamo in una stanza. genera un personaggio per il giocatore locale.
        // viene sincronizzato utilizzando PhotonNetwork.Instantiate
        playerPrefab = PhotonNetwork.Instantiate( prefabName: Path.Combine("PhotonPrefabs", "PlanePlayer2"),
            position: new Vector3( x: 1200f, y: 0f, z: 0f), Quaternion.identity, group: 0);
        DontDestroyOnLoad(playerPrefab);
    }
    #endregion
}
```

Figura 4.3: GameManager metodo Start.

Una volta che la scena viene caricata sarà eseguito il metodo Start della classe.

Se il giocatore entrato è il MasterClient allora verrà istanziato il Prefab Plane altrimenti verrà istanziato il Prefab PlanePlayer2.

I due pannelli vengono istanziati in punti differenti della scena in modo tale da non sovrapporsi tra loro e da simulare il gioco cartaceo in cui i due giocatori siedono l'uno di fronte all'altro.

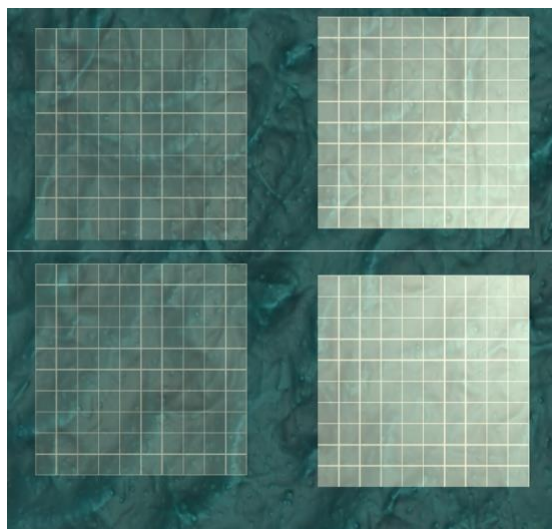


Figura 4.4: Plane e PlanePlayer2 con le rispettive tavole

I Prefab Plane e PlanePlayer2 sono dotati di Script PlayerManager ed hanno nella gerarchia una Camera. Lo Script PlayerManager sarà incaricato di realizzare le due tavole di gioco del giocatore istanziandone le caselle.

Un altro importante compito di tale classe sarà disabilitare la MainCamera della scena passando la visuale alla Camera figlia del Plane. In questo modo ogni giocatore avrà come punto di vista l'inquadratura della Camera associata al suo Plane.

```
//Istanziamo la Seconda tabella di Gioco
for (int x = 1; x < 11; ++x)
{
    for (int y = 1; y < 11; ++y)
    {
        casellaPlayer = PhotonNetwork.Instantiate( prefabName: Path.Combine("PhotonPrefabs", "Casella"),
            position: new Vector3( x: -500 + (x * 90), y: 1, z: 0 + (y * 90)), Quaternion.identity, group: 0);

        setCasella(casellaPlayer, x, y, numtable: 2);

        Tavola.table2Player1[x, y] = casellaPlayer.GetComponent<Casella>();
    }
}

#endregion
}
}
else
{
    //Facciamo in modo che il Player utilizzi la Camera che ha istanziato come figlio
    Destroy(mycam);
    Destroy(myal);
}
```

Figura 4.5: PlayerManager realizzazione delle tavole e passaggio di Camera

4.2 Gestione della Sincronizzazione

Nel paragrafo precedente abbiamo parlato della gestione dei Player mediante la classe GameManager che, oltre al “Log in”, si occupa anche dell’istanziamento del Plane di gioco. Abbiamo anche visto che nello Script PlayerManager verranno realizzate le tavole e verrà modificata la camera utilizzata dal Player.

Tali modifiche vengono eseguite indipendentemente da ogni Player. Nella fase di login, infatti, ogni Player avrà a disposizione la sua “copia” della classe GameManager che istanzierà nella scena il Prefab Plane, nel caso si tratti del MasterClient, o il PlanePlayer2.

Seguendo tale logica entrambi i Player avranno la propria copia locale del piano e delle tavole di gioco e potranno posizionare le navi sulla loro tavola aggiornando così le informazioni sulle caselle.

Ma come potranno tali informazioni arrivare all’altro giocatore se le modifiche vengono eseguite in locale?

Per risolvere questo problema sfruttiamo una componente di Photon, la PhotonView.

Come abbiamo già detto nel paragrafo 3.1.1 ogni casella è dotata di una componente PhotonView, grazie alla quale sarà possibile rendere visibile nella scena di ambo i giocatori gli oggetti dotati di tale componente e le loro relative informazioni.

Le PhotonView sono componenti estremamente utili quanto problematiche.

Ci permettono di far sì che un oggetto venga sincronizzato nella scena di gioco in modo da essere visto da tutti i Client connessi e non solo dal proprietario.

Tutte le modifiche devono però essere effettuate espressamente su tale oggetto e le variabili che verranno modificate negli script non potranno essere statiche, ovvero non potranno né essere chiamate in altre classi al di fuori della classe che le contiene né essere utilizzate in metodi statici che potrebbero essere chiamati da altre classi.

Possiamo, dunque, considerare l'utilizzo della PhotonView come una "simulazione di sincronizzazione" sfruttando le PunRPC.

Nel paragrafo 2.4.1 abbiamo spiegato la funzionalità delle chiamate PunRPC.

Un metodo chiamato in questo modo può essere eseguito da tutti i Client nella scena.

Nel momento in cui un Client qualsiasi chiama un metodo PunRPC su un oggetto, con RpcTarget.all, ogni Client presente nella stanza eseguirà tale modifica sulla sua istanza locale dell'oggetto presente nella scena.

Rendiamolo più chiaro mostrando cosa accade quando un Player posiziona una nave su una Casella.

```
//Posizionamento Nave
if (GameController.naviP1 < 10 && this.naveposizionataP1 == false && this.table == 1)
{
    GameController.naviP1 = GameController.naviP1 + 1;
    this.gameObject.GetComponent<MeshRenderer>().material = nave;
    this.pv.RPC( methodName: "PosizionaNaveP1", RpcTarget.All, params parameters: riga,colonna);
}

//Il Player1 posiziona una nave
[PunRPC]
? usages ? overrides ? ext methods ? exposing APIs
public void PosizionaNaveP1(int riga, int colonna)
{
    if (PhotonNetwork.IsMasterClient)
    {
        Tavola.table1Player1[riga,colonna].PosizionaNaveP1();
    }
    else
    {
        Tavola.table2Player2[riga,colonna].PosizionaNaveP1();
    }
}
```

Figura 4.6: Script Casella. In alto la chiamata a metodo RPC "PosizionaNaveP1" in basso il corpo del metodo.

Il Player1 con un click del mouse posizionerà una nave su una casella. La texture della Casella verrà aggiornata con l'immagine di una nave e il Booleano corrispondente a naveposizionata1 verrà settato a True.

Come possiamo vedere nella figura 4.6 anziché semplicemente settare il Booleano a True viene chiamato il metodo PunRPC, PosizionaNaveP1.

Tale metodo farà in modo che simultaneamente i Player settino il Booleano naveposizionataP1 a True, rispettando l'idea sulla condivisione delle informazioni delle tavole precedentemente espressa nel paragrafo 3.1.2.

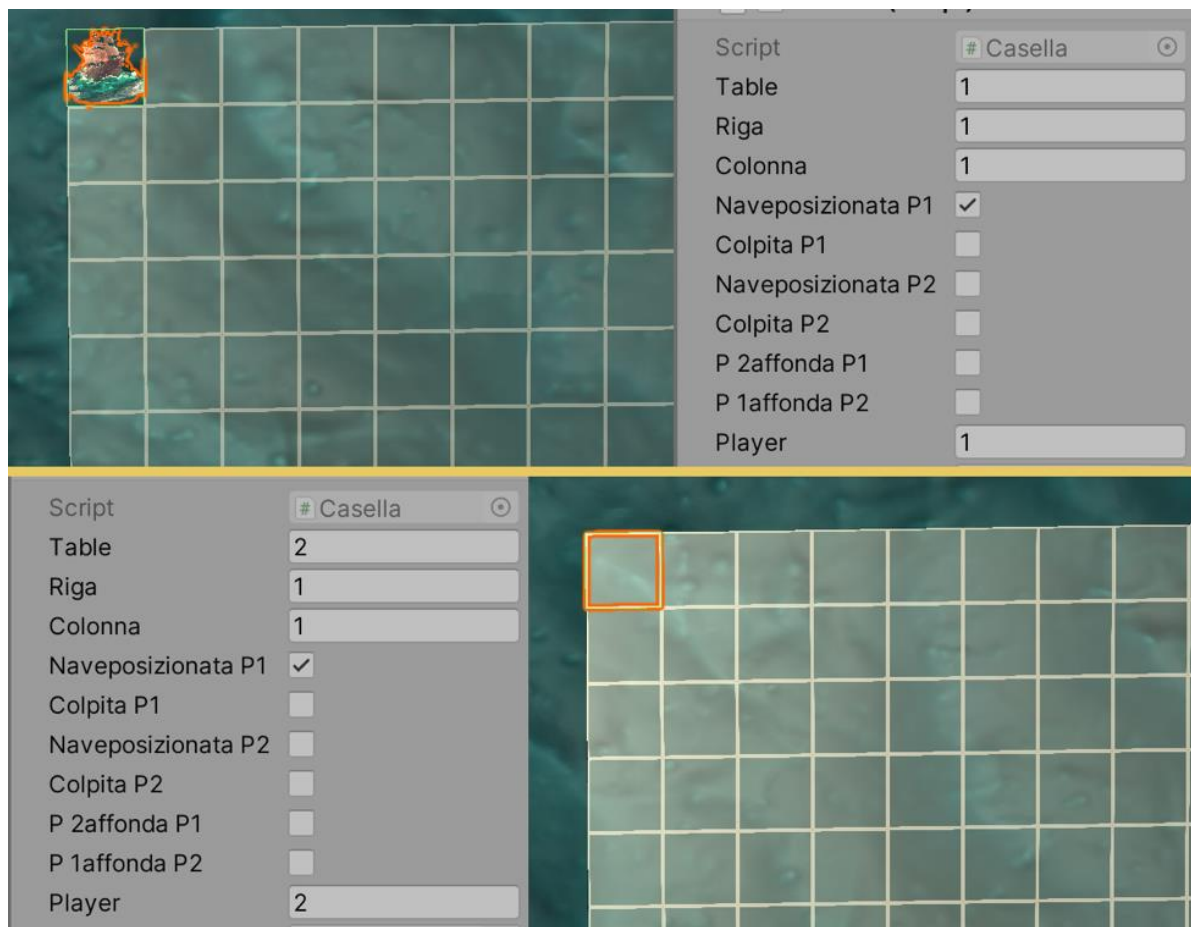


Figura 4.7: In alto la schermata e le informazioni del Player1
in basso la schermata e le informazioni del Player 2

Come possiamo vedere nella figura 4.7 anche se le informazioni vengono "sincronizzate" le texture sono differenti, dunque, il Player1 vedrà che su quella casella ha posizionato una nave, mentre il Player2 riceve solo l'informazione, com'è giusto che sia.

4.3 Gestione dei Turni

BattleShip è un gioco caratterizzato da periodi di tempo limitati che si susseguono in modo alternato in cui ogni giocatore deve attendere il proprio per poter effettuare una mossa.

Nel paragrafo 3.2.3 notiamo come le informazioni circa il Turno vengono sempre mostrate nel Top Panel.

Top Panel contiene lo Script GameController che racchiude tutte le informazioni in merito al turno e alle navi in gioco per ogni giocatore.

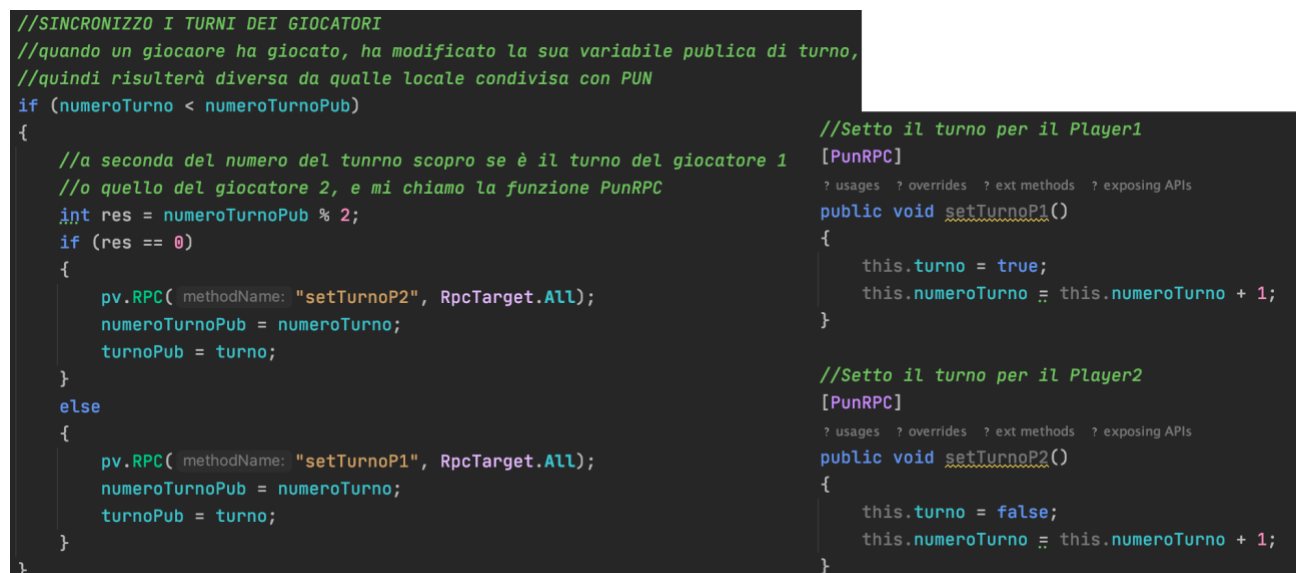
Nel paragrafo 4.2 ho illustrato le complessità in merito alla PhotonView e all'importanza di chiamare i metodi RPC dallo Script dell'oggetto che possiede tale PhotonView.

Secondo tali vincoli la gestione del turno rimane vincolata alla User Interface Top Panel e il metodo RPC che si occupa di modificare tale informazione risiede nello Script GameController.

Ma per avere un gioco più fluido e coerente subito dopo che un Player ha effettuato la sua mossa il turno dovrebbe passare al suo avversario.

Quindi come possiamo effettuare tale modifica sapendo che la variabile da modificare non si trova nelle caselle e che siamo impossibilitati a chiamare variabili o metodi RPC da classi esterne per i vincoli precedentemente visti di PhotonView?

Per risolvere tale problema ho scelto di utilizzare variabili temporanee statiche.



```
//SINCRONIZZO I TURNI DEI GIOCATORI
//quando un giocatore ha giocato, ha modificato la sua variabile pubblica di turno,
//quindi risulterà diversa da quella locale condivisa con PUN
if (numeroTurno < numeroTurnoPub)
{
    //a seconda del numero del turno scopro se è il turno del giocatore 1
    //o quello del giocatore 2, e mi chiamo la funzione PunRPC
    int res = numeroTurnoPub % 2;
    if (res == 0)
    {
        pv.RPC( methodName: "setTurnoP2", RpcTarget.All);
        numeroTurnoPub = numeroTurno;
        turnoPub = turno;
    }
    else
    {
        pv.RPC( methodName: "setTurnoP1", RpcTarget.All);
        numeroTurnoPub = numeroTurno;
        turnoPub = turno;
    }
}

//Setto il turno per il Player1
[PunRPC]
public void setTurnoP1()
{
    this.turno = true;
    this.numeroTurno = this.numeroTurno + 1;
}

//Setto il turno per il Player2
[PunRPC]
public void setTurnoP2()
{
    this.turno = false;
    this.numeroTurno = this.numeroTurno + 1;
}
```

Figura 4.8: GameController. Sulla sinistra la modifica del turno e la chiamata al metodo RPC sulla destra il corpo del metodo RPC

Utilizzando una variabile statica, nel mio caso turnoPub, essa potrà essere modificata al di fuori dello script GameController.

Una volta che il Player eseguirà la sua mossa verranno effettuate operazioni su una Casella tra cui la modifica alla variabile turnoPub contenuta in GameController.

Il GameController verificherà se i valori di turno e turnoPub coincidono e in caso contrario bisognerà cambiare turno poiché un giocatore ha effettuato la sua mossa.


Verrà eseguita, dunque, una chiamata al metodo RPC che farò in modo che entrambi i giocatori modificheranno la loro variabile turno.

Tale strategia, utilizzata sul Booleano turno che con True indica il turno del Player1 e con False quella del Player2, viene estesa anche al numero del turno e al numero di navi rimaste per ogni Player.

4.4. User Interface

Quando ci si ritrova ad iniziare un gioco, che sia da tavolo o digitale, una delle caratteristiche più importanti per identificare e rendere unico un giocatore è l'uso di un Nickname.

Il paragrafo 3.2.3 elenca i differenti Panel presenti nella schermata di gioco e le caratteristiche appartenenti ai Player che esse contengono. Lo Script PlayerUI presente nel Prefab Panel PlayerUI consente di tener traccia del Nickname scelto da ogni giocatore e di mostrarlo ad entrambi i Player tenendo sempre sulla sinistra quello del Player1 e sulla destra quello del Player2. Tale Nickname viene memorizzato in modo tale che in un successivo accesso del giocatore egli avrà già a disposizione il Nickname da lui scelto in una precedente partita.



```

? usages ? overrides ? ext methods ? exposing APIs
public void SetTarget(PlayerManager target)
{
    if(_target == null)
    {
        Debug.LogError( message: "<Color=Red><a>Missing</a></Color> " +
            "PlayMakerManager target for PlayerUI.SetTarget.", context: this);
        return;
    }
    target = _target;
    if (playerNameText != null)
    {
        playerNameText.text = target.photonView.Owner.NickName;
    }
}

? usages ? overrides ? ext methods ? exposing APIs
void OnLevelWasLoaded(int level)
{
    //Settiamo il NickName sull'interfaccia
    GameObject uiGo = Instantiate(this.PlayerUIPrefab);
    uiGo.SendMessage( methodName: "SetTarget", value: this, SendMessageOptions.RequireReceiver);
}

```

Figura 4.9: In alto PlayerUI Script che mostra il corpo del metodo setTarget
in basso GameManager in cui viene chiamato il metodo setTarget

Il PlayerManager oltre ad istanziare le caselle di gioco e ad “appropriarsi” della visuale, togliendola alla MainCamera, si occupa anche di istanziare il Prefab PlayerUI. Come possiamo vedere nella figura 4.9 il PlayerManager dopo aver istanziato il Prefab PlayerUI ne chiama il metodo setTarget.

Il metodo setTarget prende come parametro un PlayerManager e ne estrae il Nickname che setta come testo nella sua interfaccia.

Durante una partita è molto importante tener traccia dello stato attuale dei giocatori. Ho scelto di inserire delle Slider Bar per svolgere tale ruolo.

All'interno del Top Panel possiamo trovare lo Script Text responsabile del testo relativo alle informazioni del turno e delle Slider Bar dei Player.

```
//Modifico il numero dei turni
numeroturno.GetComponent<Text>().enabled = true;
numeroturno.text = " " + GameController.numeroTurnoPub;

//A seconda del turno, setto la scritta al ogni giocatore
if (GameController.turnoPub)
{
    playerturno.text = "è il tuo turno";
}
else
{
    playerturno.text = "è il turno dell'avversario";
}

sliderPlayer1.value = GameController.nv1;
sliderPlayer2.value= GameController.naviaffondate2;
```

Figura 4.10: Script Text che mostra come vengono modificate le scritte in merito al turno e le SliderBar dei Player



Figura 4.11: User Interface

Come abbiamo detto nel paragrafo precedente GameController si occuperà di tenere aggiornate e sincronizzate le informazioni in merito al turno e al numero di navi sfruttando variabili statiche temporanee.

Dichiarando tali variabili statiche temporanee esse saranno equivalenti alle variabili sincronizzate dai metodi RPC.

Possiamo, infatti, richiamare le variabili statiche in Text così da avere sempre in mostra nella User Interface lo status attuale del gioco e dei Player.

Capitolo 5

Conclusioni e Sviluppi Futuri

5.1 Come giocare a BattleShip

5.1.1 Design del Gioco

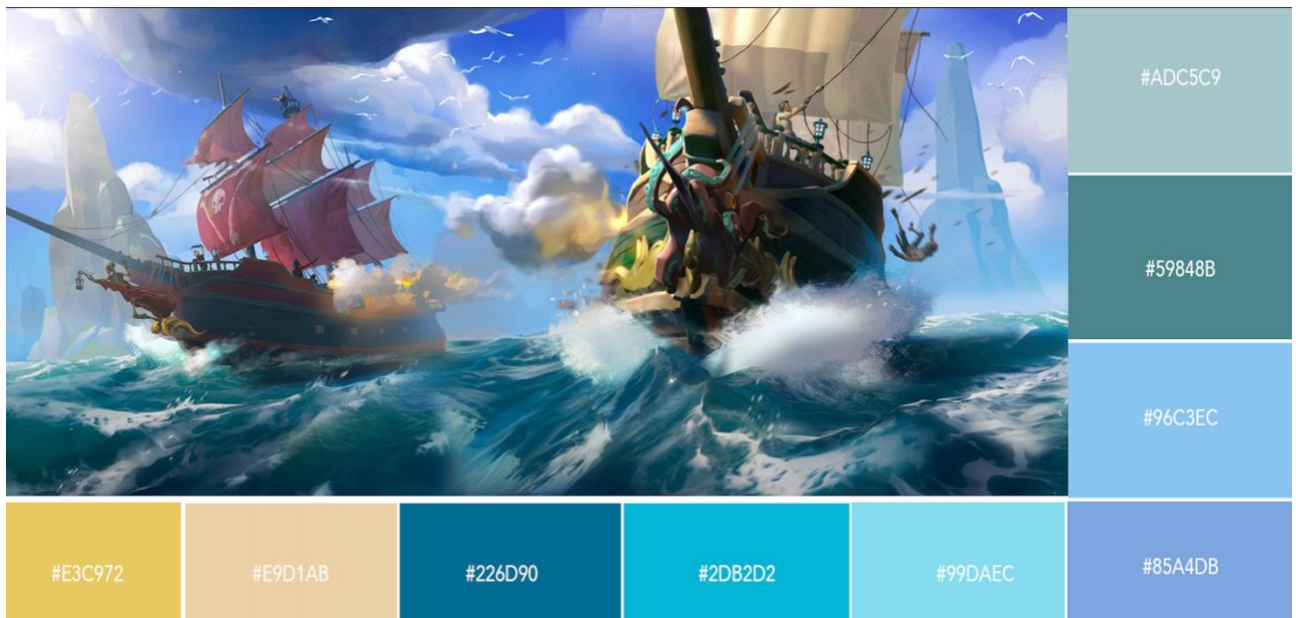


Figura 5.1: Sfondo delle HomeMenu con la Palette

Come sfondo della HomeMenu è stata scelta un'immagine con due Navi intente a scambiarsi cannonate richiamando il tema della BattleShip.

Le varie sfumature di colori tra il cielo, il mare e i decori delle navi hanno definito la Palette di colori che è stata utilizzata in tutti i Panel della User Interface così da mantenere coerente lo stile del gioco, senza distaccarsi dal tema.

5.1.2 Prima fase del gioco

Nel paragrafo 3.2 viene mostrata la schermata iniziale in cui un giocatore può scegliere il proprio Nickname e avviare la partita.

Una volta che anche il suo avversario avrà fatto lo stesso entrambi si ritroveranno nella prima fase della partita, quella del posizionamento delle navi.



Figura 5.2: Screen di BattleShip. Prima fase del gioco, posizionamento navi.

Ogni giocatore vedrà la scena mostrata nella figura 5.2, una tavola su cui potrà posizionare le sue 10 navi. Possiamo notare che il pannello sulla destra richiama parte dell'immagine usata come sfondo nel HomeMenu.

Lo stile delle navi mostrate nel menù è analogo a quello delle navi posizionate dal Player sulle caselle così da uniformare lo stile del gioco.

5.1.3 Seconda fase del gioco

Dopo aver posizionato le 10 navi apparirà il pulsante "Gioca" il quale permette di capire che il Player è pronto ad iniziare la partita.



Figura 5.3: Screen BattleShip di una Partita

Ogni giocatore potrà visualizzare in alto il proprio turno e, dunque, se potrà attaccare o attendere la mossa dell'avversario.

Ogni azione compiuta viene notificata ad entrambi i Player. Osservando la figura 5.3 possiamo notare che ogni qualvolta il Player1 attacca il suo avversario le caselle cambiano. La Texture di una casella, infatti, diventerà l'immagine dell'oceano increspato nel caso in cui il colpo sia andato a vuoto o di una nave che sta esplodendo nel caso egli abbia colpito ed affondato una nave nemica.

Tali modifiche di texture vengono applicate anche alle caselle della tavola sinistra, ovvero la tavola dove i Player hanno posizionato le loro navi, in modo da rappresentare le mosse eseguite dall'avversario.

In questo modo ogni Player potrà visionare:

- Tavola destra:
 - Caselle da lui colpite;
 - Navi da lui affondate.
- Tavola sinistra:
 - Navi da lui posizionate;
 - Caselle che l'avversario ha colpito;
 - Navi che l'avversario ha affondato.

La Slider bar inizialmente azzurra, decrementando, si colorerà di giallo man mano che le navi del Player alla quale è collegata vengono affondate. Lo scopo del gioco è appunto quello di affondare la flotta nemica e il primo Player che ci riuscirà sarà il vincitore della partita.



Figura 5.4: Screen BattleShip in alto la schermata della Vittoria in basso quella della Sconfitta

5.2 Conclusioni

Unity è un ottimo motore grafico multiplatforma utilizzato in molti ambiti tra cui quello dello sviluppo videoludico.

Per la sua semplicità risulta molto intuitivo. Rende piacevole al programmatore l'esperienza della manipolazione degli oggetti e della creazione di una scena.

Permette, inoltre, di dar libero sfogo alla propria fantasia semplificando l'implementazione di un qualsiasi tipo di videogioco.

Realizzare BattleShip è stato un utile esperimento di utilizzo della comunicazione multiplayer sfruttando le proprietà di Photon.

Photon è una piattaforma multiplayer pensata con lo scopo di semplificare la realizzazione di giochi Multiplayer. Grazie alle sue molteplici componenti è possibile realizzare una chat, sia composta da soli messaggi e sia vocale, instaurare una comunicazione diretta tra un Player e l'altro, realizzare giochi d'azione e tante altre funzionalità.

Utilizzare Photon non è semplice e immediato ma gode di guide, tutorial online e community che agevolano il programmatore.

Il progetto di tesi mi ha permesso di studiare una nuova piattaforma arricchendo il mio bagaglio culturale in merito ad uno degli aspetti legati ai videogame che da sempre ha suscitato il mio interesse, ovvero la comunicazione Multiplayer.

Ritengo che oggi avere la possibilità di giocare in maniera multimediale a giochi tradizionali, e non solo, sia molto importante e utile in circostanze in cui, come l'attuale situazione, non sia possibile riunirsi fisicamente con amici e parenti.

5.3 Sviluppi Futuri

Essendo ispirato al tradizionale gioco da tavola BattleShip risulta essere molto fedele tale versione. La grafica è molto minimale in modo da non sovraccaricare e stancare l'occhio del giocatore.

Tra i possibili sviluppi futuri suggerirei di integrare elementi multimediali come una musica di sottofondo o animazioni che mostrino un cannone che affonda una nave o colpisce l'acqua ogni qualvolta un giocatore esegue una mossa.

Nel paragrafo 1.3.1 vengono elencate le caratteristiche della versione del gioco da me proposta in cui si evince che le navi posizionate dai vari giocatori hanno la dimensione di un'unica casella.

Un altro possibile sviluppo futuro, dunque, potrebbe essere l'integrazione di navi di dimensioni variabili.

Altri possibili sviluppi riguardano la presenza di una Chat tramite la quale i giocatori potranno comunicare tra loro.

Prendendo spunto dalle versioni viste nel paragrafo 1.4 si potrebbe estendere il gioco da 2 a 4 giocatori, introducendo nuove regole e modalità di gioco.

Codici Citati

In questa appendice sono presenti tutti i codici, riportati per intero, citati nella tesi.

Casella

```
using Photon.Pun;
using UnityEngine;

[RequireComponent(typeof(PhotonView))]
public class Casella : MonoBehaviourPunCallbacks, IPunObservable
{
    #region Public Fields
        public int table;
        public int riga = 0;
        public int colonna = 0;
        public bool naveposizionataP1;
        public bool colpitaP1;
        public bool naveposizionataP2;
        public bool colpitaP2;
        public bool p2affondaP1;
        public bool p1affondaP2;
        public int player;

        public Texture2D mirino;
        public Texture2D mouse;
        public Material nave;
        public Material colpita;
        public Material affondata;

        private PhotonView pv;
    #endregion

    void Update()
    {
        //Modifichiamo l'aspetto delle caselle del Player1 nel caso esse
        //vengano colpite e/o affondate
        if (table == 1 && player == 1)
        {
            if (p2affondaP1 && colpitaP1)
            {
                this.gameObject.GetComponent<MeshRenderer>().material =
                affondata;
            } else if (colpitaP1)
            {
                this.gameObject.GetComponent<MeshRenderer>().material =
                colpita;
            }
        }

        //Modifichiamo l'aspetto delle caselle del Player2 nel caso esse
        //vengano colpite e/o affondate
    }
```

```

if (table == 1 && player == 2)
{
    if (plaffondaP2 && colpitaP2)
    {
        this.gameObject.GetComponent<MeshRenderer>().material =
affondata;
    } else if (colpitaP2)
    {
        this.gameObject.GetComponent<MeshRenderer>().material =
colpita;
    }
}

}

#region OnMouse Function

    #region Mouse Icon

        //facciamo in modo che quando il Mouse passi sopra le caselle,
        esso divento un mirino
        private void OnMouseEnter()
        {
            Cursor.SetCursor(mirino, Vector2.zero, CursorMode.Auto);
        }

        private void OnMouseExit()
        {
            Cursor.SetCursor(mouse, Vector2.zero, CursorMode.Auto);
        }

    #endregion

public void OnMouseDown()
{
    //Operazioni del Giocatore 1
    #region Player1

        if (this.player == 1 )
        {
            //Posizionamento Nave
            if (GameController.naviP1 < 10 && this.naveposizionataP1 ==
false && this.table == 1)
            {
                GameController.naviP1 = GameController.naviP1 + 1;
                this.gameObject.GetComponent<MeshRenderer>().material =
nave;

                this.pv.RPC("PosizionaNaveP1", RpcTarget.All, riga, colonna);
            }

            //ProvaColpo
            if ( this.table == 2 && GameController.iniziogioco &&
GameController.turnoPub && this.colpitaP2 == false)
            {
                //controllo se nella casella dove sto colpendo
                l'avversario ha posizionato una nave
                if (naveposizionataP2)
                {

```



```

this.gameObject.GetComponent<MeshRenderer>().material = affondata;
//devo effettuare le modifiche per il numero di
navi e il turno

this.pv.RPC("plaffondanavedip2",RpcTarget.All,riga,colonna);
    GameController.numeroTurnoPub =
GameController.numeroTurnoPub + 1;
    GameController.turnoPub = false;
    GameController.naviaffondate2 += 1;
    Debug.LogWarning("Player1 : Navi P2 Affondate
:"+GameController.naviaffondate2);
    }
    else
    {

this.gameObject.GetComponent<MeshRenderer>().material = colpita;
//devo effettuare le modifiche per il turno

this.pv.RPC("p1ColpisceCasellaP2",RpcTarget.All,riga,colonna);
    GameController.numeroTurnoPub =
GameController.numeroTurnoPub + 1;
    GameController.turnoPub = false;
    }
    }
}

#endregion

//Operazioni del Giocatore 2
#region Player2

    if (this.player == 2 )
    {
        //Posizionamento Nave
        if ( GameController.naviP2<10 && this.naveposizionataP2 ==
false && this.table == 1 && GameController.iniziogioco == false)
        {
            GameController.naviP2 = GameController.naviP2 + 1;
            this.gameObject.GetComponent<MeshRenderer>().material =
nave;

this.pv.RPC("PosizionaNaveP2",RpcTarget.All,riga,colonna);
        }

        //ProvaColpo
        if ( this.table == 2 && GameController.turnoPub == false &&
GameController.iniziogioco && this.colpitaP1 == false)
        {
            //controllo se nella casella dove sto colpendo
l'avversario ha posizionato una nave
            if (naveposizionataP1)
            {

this.gameObject.GetComponent<MeshRenderer>().material = affondata;
//devo effettuare le modifiche per il numero di
navi e il turno

this.pv.RPC("p2affondanavedip1",RpcTarget.All,riga,colonna);
GameController.numeroTurnoPub=GameController.numeroTurnoPub + 1;

```

```

        GameController.turnoPub = true;
        GameController.nv1 = GameController.nv1 + 1;
        Debug.LogWarning("PLAYER2 : Ho affondato una nave
nemica, Navi affondate = "+GameController.nv1);
    }
    else
    {

this.gameObject.GetComponent<MeshRenderer>().material = colpita;
        //devo effettuare le modifiche per il turno

this.pv.RPC("p2ColpisceCasellaP1",RpcTarget.All,riga,colonna);
        GameController.numeroTurnoPub =
GameController.numeroTurnoPub + 1;
        GameController.turnoPub = true;
    }
}

#endregion

}

#endregion

#region PunRPCMethods

#region Posizionamento e Rimozione di una Nave

    //Il Player1 posiziona una nave
    [PunRPC]
    public void PosizionaNaveP1(int riga, int colonna)
    {
        if (PhotonNetwork.IsMasterClient)
        {
            Tavola.table1Player1[riga,colonna].PosizionaNaveP1();
        }
        else
        {
            Tavola.table2Player2[riga,colonna].PosizionaNaveP1();
        }
    }

    //Il Player 2 Posiziona una nave
    [PunRPC]
    public void PosizionaNaveP2(int riga, int colonna)
    {
        if (PhotonNetwork.IsMasterClient)
        {
            Tavola.table2Player1[riga,colonna].PosizionaNaveP2();
        }
        else
        {
            Tavola.table1Player2[riga,colonna].PosizionaNaveP2();
        }
    }
}

#endregion

#region Colpisci e/o Affonda
    //Il Player 1 Affonda una Nave del Player2

```



```
[PunRPC]
public void plaffondanavedip2(int riga, int colonna)
{
    if (PhotonNetwork.IsMasterClient)
    {
        Tavola.table2Player1[riga,colonna].AffondaP2();
        Tavola.table2Player1[riga,colonna].ColpisciP2();
    }
    else
    {
        Tavola.table1Player2[riga,colonna].AffondaP2();
        Tavola.table1Player2[riga,colonna].ColpisciP2();
    }
}

//Il Player 2 Affonda una Nave del Player1
[PunRPC]
public void p2affondanavedip1(int riga, int colonna)
{
    if (PhotonNetwork.IsMasterClient)
    {
        Tavola.table1Player1[riga,colonna].AffondaP1();
        Tavola.table1Player1[riga,colonna].ColpisciP1();
    }
    else
    {
        Tavola.table2Player2[riga,colonna].AffondaP1();
        Tavola.table2Player2[riga,colonna].ColpisciP1();
    }
}

//Il Player 1 Colpisce
[PunRPC]
public void p1ColpisceCasellaP2(int riga, int colonna)
{
    if (PhotonNetwork.IsMasterClient)
    {
        Tavola.table2Player1[riga,colonna].ColpisciP2();
    }
    else
    {
        Tavola.table1Player2[riga,colonna].ColpisciP2();
    }
}

//Il Player 2 Colpisce
[PunRPC]
public void p2ColpisceCasellaP1(int riga, int colonna)
{
    if (PhotonNetwork.IsMasterClient)
    {
        Tavola.table1Player1[riga,colonna].ColpisciP1();
    }
    else
    {
        Tavola.table2Player2[riga,colonna].ColpisciP1();
    }
}

#endregion
```

```

#endregion

#region PhotonView Observer

    public void OnPhotonSerializeView(PhotonStream stream,
PhotonMessageInfo info)
    {
        if (stream.IsWriting)
        {
            // Possediamo questo giocatore: invia queste informazioni
agli altri i nostri dati
            stream.SendNext(table);
            stream.SendNext(riga);
            stream.SendNext(colonna);
            stream.SendNext(naveposizionataP1);
            stream.SendNext(naveposizionataP2);
            stream.SendNext(colpitaP1);
            stream.SendNext(colpitaP2);
            stream.SendNext(plaffondaP2);
            stream.SendNext(p2affondaP1);
            stream.SendNext(player);
        }
        else
        {
            //Lettore di rete, ricevi dati
            this.table = (int)stream.ReceiveNext();
            this.riga = (int)stream.ReceiveNext();
            this.colonna = (int)stream.ReceiveNext();
            this.naveposizionataP1 = (bool)stream.ReceiveNext();
            this.naveposizionataP2 = (bool)stream.ReceiveNext();
            this.colpitaP1 = (bool)stream.ReceiveNext();
            this.colpitaP2 = (bool)stream.ReceiveNext();
            this.plaffondaP2 = (bool)stream.ReceiveNext();
            this.plaffondaP2 = (bool)stream.ReceiveNext();
            this.player = (int)stream.ReceiveNext();
        }
    }

#endregion
}

```

GameController

```
using System;
using System.Collections;
using System.Collections.Generic;
using Photon.Pun;
using UnityEngine;
using UnityEngine.UI;
using UnityStandardAssets.Characters.ThirdPerson.PunDemos;

[RequireComponent(typeof(PhotonView))]
public class GameController : MonoBehaviourPunCallbacks, IPunObservable
{
    #region fields

    public static int naviP1 = 0;
    public static int naviP2 = 0;
    public bool startP1 = false;
    public bool startP2 = false;
    public static bool iniziogioco = false;
    public int n2=0;
    public int n1=0;
    public int numeroTurno = 1;
    public static int numeroTurnoPub = 1;
    public static bool turnoPub = true;
    public bool turno = true;
    public static int nv1 = 0;
    public static int naviaffondate2 = 0;
    private PhotonView pv;

    #endregion

    void Start()
    {
        pv = GetComponentInParent<PhotonView>();
        naviP1 = 0;
        naviP2 = 0;
        startP1 = false;
        startP2 = false;
        iniziogioco = false;
        turno = true;
        turnoPub = true;
        numeroTurno = 1;
        numeroTurnoPub = numeroTurno;
        nv1 = 0;
        naviaffondate2 = 0;
    }

    //Risetto i parametri iniziali nel caso il gioco venga riavviato
    public void Awake()
    {
        pv = GetComponentInParent<PhotonView>();
        naviP1 = 0;
        naviP2 = 0;
        startP1 = false;
        startP2 = false;
        iniziogioco = false;
    }
}
```

```

        turno = true;
        turnoPub = turno;
        numeroTurno = 1;
        numeroTurnoPub = numeroTurno;
        nv1 = 0;
        naviaffondate2 = 0;
    }

    #region PunRPC Metods

    //Il Player1 è pronto setta a true il suo booleano
    [PunRPC]
    public void gioco1(bool si)
    {
        this.startP1 = si;
        Debug.Log("Ho avviato la funzione INIZIO GIOCO 1, il valore di
startP1 è : "+startP1+ "  P2StartP1 : "+startP2);
    }

    //Il Player2 è pronto e setta a true il suo booleano
    [PunRPC]
    public void gioco2(bool si)
    {
        this.startP2 = si;
        Debug.Log("Ho avviato la funzione INIZIO GIOCO 2, il valore di
startP1 è : "+startP1+ "  P2StartP1 : "+startP2);
    }

    //Setto il turno per il Player1
    [PunRPC]
    public void setTurnoP1()
    {
        this.turno = true;
        this.numeroTurno = this.numeroTurno + 1;
    }

    //Setto il turno per il Player2
    [PunRPC]
    public void setTurnoP2()
    {
        this.turno = false;
        this.numeroTurno = this.numeroTurno + 1;
    }

    [PunRPC]
    public void SettaNavi1()
    {
        this.n1 = this.n1 + 1;
        Debug.LogWarning("SONO NALLA PUN SETTO NAVI 1 :le navi sono "+ n1);
    }

    [PunRPC]
    public void Navi2()
    {
        this.n2 = this.n2 + 1;
        Debug.LogWarning("SONO NALLA PUN SETTO NAVI 2 :le navi sono "+ n2);
    }

    #endregion

```

```

public void setgioco1()
{
    pv.RPC("gioco1", RpcTarget.All, true);
}

public void setgioco2()
{
    pv.RPC("gioco2",RpcTarget.All, true);
}

// Update is called once per frame
void Update()
{
    if (PhotonNetwork.IsMasterClient)
    {
        if (startP1 == true && startP2 == true)
        {
            //entremabi i giocatori hanno cliccato su "Gioca" quindi
            sono pronti ad iniziare
            iniziogioco = true;
        }

        //SINCRONIZZO I TURNI DEI GIOCATORI
        //quando un giocaore ha giocato, ha modificato la sua variabile
        pubblica di turno,
        //quindi risulterà diversa da quelle locale condivisa con PUN
        if (numeroTurno < numeroTurnoPub)
        {
            //a seconda del numero del tunrno scopro se è il turno del
            giocatore 1
            //o quello del giocatore 2, e mi chiamo la funzione PunRPC
            int res = numeroTurnoPub % 2;
            if (res == 0)
            {
                pv.RPC("setTurnoP2", RpcTarget.All);
                numeroTurnoPub = numeroTurno;
                turnoPub = turno;
            }
            else
            {
                pv.RPC("setTurnoP1", RpcTarget.All);
                numeroTurnoPub = numeroTurno;
                turnoPub = turno;
            }
        }
        else if (numeroTurnoPub < numeroTurno)
        {
            //Se l'altro giocatore ha modificato il turno, la variabile
            locale PUN sarà diversa, quindi setto i miei valori uguali e aggiornno il
            turno
            numeroTurnoPub = numeroTurno;
            turnoPub = turno;
        }

        if (n1 != nv1 || n2 != naviaffondate2)
        {
            //Controllo che il numero locale e il numero PUN di navi
            del Player1 sia uguale e nel caso lo modifico

```

```

        if (n1 < nv1)
        {
            pv.RPC("SettaNavi1", RpcTarget.All);
            nv1 = n1;
            Debug.LogWarning("SONO USCITA DALLA PUN SETTO NAVI 1
:le navi sono "+ n1 +" le PUB sono : "+nv1);
        } else if (nv1 < n1)
        {
            nv1 = n1;
        }

        //Controllo che il numero locale e il numero PUN di navi
del Player2 sia uguale e nel caso lo modifico
        if (n2 < naviaffondate2)
        {
            pv.RPC("Navi2", RpcTarget.All);
            naviaffondate2 = n2;
        } else if (naviaffondate2 < n2)
        {
            naviaffondate2 = n2;
        }
    }
}
else
{
    if (startP1 == true && startP2 == true)
    {
        //entremabi i giocatori hanno cliccato su "Gioca" quindi
sono pronti ad iniziare
        iniziogioco = true;
    }

    //SINCRONIZZO I TURNI DEI GIOCATORI
    //quando un giocaore ha giocato, ha modificato la sua variabile
publica di turno, quindi risulterà diversa da qualle locale condivisa con
PUN
    if (numeroTurno < numeroTurnoPub)
    {
        //a seconda del numero del tunrno scopro se è il turno del
giocatore 1 o quello del giocatore 2, e mi chiamo la funzione PunRPC
        int res = numeroTurnoPub % 2;
        if (res == 0)
        {
            pv.RPC("setTurnoP2", RpcTarget.All);
            numeroTurnoPub = numeroTurno;
            turnoPub = turno;
        }
        else
        {
            pv.RPC("setTurnoP1", RpcTarget.All);
            numeroTurnoPub = numeroTurno;
            turnoPub = turno;
        }
    }
    else if (numeroTurnoPub < numeroTurno)
    {
        //Se l'altro giocatore ha modificato il turno, la variabile
locale PUN sarà diversa, quindi setto i miei valori uguali e aggiornno il
turno
        numeroTurnoPub = numeroTurno;
    }
}

```

```

        turnoPub = turno;
    }

    if (n1 != nv1 || n2 != naviaffondate2)
    {
        //Controllo che il numero locale e il numero PUN di navi
del Player1 sia uguale e nel caso lo modifico
        if (n1 < nv1)
        {
            pv.RPC("SettaNavi1", RpcTarget.All);
            nv1 = n1;
            Debug.LogWarning("SONO USCITA DALLA PUN SETTO NAVI 1
:le navi sono "+ n1 +" le PUB sono : "+nv1);
        } else if (nv1 < n1)
        {
            nv1 = n1;
        }

        //Controllo che il numero locale e il numero PUN di navi
del Player2 sia uguale e nel caso lo modifico
        if (n2 < naviaffondate2)
        {
            pv.RPC("Navi2", RpcTarget.All);
            naviaffondate2 = n2;
        } else if (naviaffondate2 < n2)
        {
            naviaffondate2 = n2;
        }
    }
}

}

public void OnPhotonSerializeView(PhotonStream stream,
PhotonMessageInfo info)
{
    if (stream.IsWriting)
    {
        // Possediamo questo giocatore: invia queste informazioni agli
altri i nostri dati
        stream.SendNext(startP1);
        stream.SendNext(startP2);
        stream.SendNext(iniziogioco);
        stream.SendNext(n1);
        stream.SendNext(n2);
    }
    else
    {
        //Lettore di rete, ricevi dati
        startP1 = (bool)stream.ReceiveNext();
        startP2 = (bool)stream.ReceiveNext();
        iniziogioco = (bool)stream.ReceiveNext();
        n1 = (int)stream.ReceiveNext();
        n2 = (int)stream.ReceiveNext();
    }
}
}
}

```

GameManager

```
using UnityEngine;
using UnityEngine.SceneManagement;
using Photon.Pun;
using Photon.Realtime;
using System.IO;
using System;

public class GameManager : MonoBehaviourPunCallbacks
{
    #region Public Fields

    [Tooltip("Prefab da usare per istanziare le piattaforme di gioco")]
    GameObject playerPrefab;

    public Canvas menu;
    public Canvas endgame;

    #endregion

    #region Photon Callbacks
    /// vengono chiamate quando il giocatore locale lascia la stanza.
    public override void OnLeftRoom()
    {
        SceneManager.LoadScene(0);
    }

    public override void OnPlayerEnteredRoom(Player other)
    {
        if (PhotonNetwork.IsMasterClient)
        {
            LoadArena();
        }
    }

    public override void OnPlayerLeftRoom(Player otherPlayer)
    {
        Debug.LogFormat("OnPlayerLeftRoom() {0}", otherPlayer.NickName);
        if (PhotonNetwork.IsMasterClient)
        {
            LoadArena();
        }
    }
    #endregion

    #region Public Methods

    public void LeaveRoom()
    {
        PhotonNetwork.LeaveRoom();
    }
    #endregion

    #region Private Methods
    void LoadArena()
    {
        PhotonNetwork.LoadLevel("Room for " +
        PhotonNetwork.CurrentRoom.PlayerCount);
    }
}
```



```

#endregion
void Start()
{
    #region Istantiate Player
    if (PhotonNetwork.IsMasterClient)
    {
        // siamo in una stanza. genera un personaggio per il giocatore locale.
        // viene sincronizzato utilizzando PhotonNetwork.Instantiate
        playerPrefab =
PhotonNetwork.Instantiate(Path.Combine("PhotonPrefabs", "Plane"), new
Vector3(0f, 0f, 0f), Quaternion.identity, 0);
        DontDestroyOnLoad(playerPrefab);

    }
    else
    {
        // siamo in una stanza. genera un personaggio per il giocatore locale.
        // viene sincronizzato utilizzando PhotonNetwork.Instantiate
        playerPrefab = PhotonNetwork.Instantiate(Path.Combine("PhotonPrefabs",
"PlanePlayer2"), new Vector3(1200f, 0f, 0f), Quaternion.identity, 0);
        DontDestroyOnLoad(playerPrefab);

    }
    #endregion
}

// Update is called once per frame
void Update()
{
    if (PhotonNetwork.PlayerList.Length == 2)
    {
        if (GameController.iniziogioco )
        {
            menu.GetComponent<Canvas>().enabled = false;
        }
    }

    if (GameController.nv1 == 10 || GameController.naviaffondate2 ==
10)
    {
        GameController.iniziogioco = false;
        endgame.GetComponent<Canvas>().enabled = true;
    }
}
}

```

Launcher

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Photon.Pun;
using Photon.Realtime;

namespace BattleShip
{
    public class Launcher : MonoBehaviourPunCallbacks
    {
        #region Private Serializable Fields
        ///definiamo il numero massimo di giocatori, così che quando una stanza sarà piena, ne sarà creata un'altra
        [Tooltip("definiamo il numero massimo di giocatori, così che quando una stanza sarà piena, ne sarà creata un'altra")]
        [SerializeField]
        private byte maxPlayerForRoom = 2;

        #endregion

        #region Private Fields
        /// Numero di versione del client. Gli utenti sono separati da gameVersion
        string gameVersion = "1";

        [Tooltip("Panrel UI per il nome dell'utente")]
        [SerializeField]
        private GameObject controlPanel;
        [Tooltip("Panel UI per informare l'utente della connessione")]
        [SerializeField]
        private GameObject progressLabel;

        /// Teniamo traccia del processo in corso. Essendo la connessione asincrona è basata su differenti Callbacks. Dobbiamo tenere traccia delle Callback per regolare correttamente il comportamento quando riceviamo una chiamata In generale viene utilizzato per il Callback OnConnectionToMaster()
        bool isConnecting;

        #endregion

        #region MonoBehaviour CallBacks
        ///Chiamato sui GameObject durante la fase iniziale di Inizializzazione
        void Awake()
        {
            ///possiamo usare la chiamata sul ClientMaster e tutti i client nella stanza sincronizzano automaticamente il loro livello
            PhotonNetwork.AutomaticallySyncScene = true;
        }

        void Start()
        {
            progressLabel.SetActive(false);
            controlPanel.SetActive(true);
        }

        #endregion
        #region Public Methods
        ///Iniziamo il processo di connessione: Se siamo già connessi proviamo ad entrare in una stanza casuale, altrimenti connessi quest'istanza a Photon

```

```

Cloud Network
public void Connect()
{
    if (PhotonNetwork.IsConnected)
    {
        //Tentiamo di entrare in una stanza casuale, se falliamo riceviamo una
        //notifica in OnJoinRandomFailed() e ne creiamo una
        PhotonNetwork.JoinRandomRoom();
    }
    else
    {
        //dobbiamo connetterci a Photon Online Server.
        isConnecting = PhotonNetwork.ConnectUsingSettings();
        PhotonNetwork.GameVersion = gameVersion;
    }

    controlPanel.SetActive(false);
    progressLabel.SetActive(true);
}
#endregion

#region MonoBehaviourPunCallbacks Callbacks
public override void OnConnectedToMaster()
{
    //base.OnConnectedToMaster();
    Debug.Log("PUN/Launcher: OnConnectionToMaster() è stato
chiamato dalla PUN");
    if(isConnecting)
    {
        PhotonNetwork.JoinRandomRoom();
        isConnecting = false;
    }
}

public override void OnDisconnected(DisconnectCause cause)
{
    controlPanel.SetActive(true);
    progressLabel.SetActive(false);
    isConnecting = false;
}

public override void OnJoinRandomFailed(short returnCode, string
message)
{
    //non siamo riusciti ad unirci ad una stanza, forse non ne
    //esisteva una oppure erano piene. Ne creiamo una
    PhotonNetwork.CreateRoom(null, new RoomOptions { MaxPlayers =
maxPlayerForRoom });
}

public override void OnJoinedRoom()
{
    PhotonNetwork.LoadLevel("Room for 1");
}
#endregion
}

```

PlayerManager

```

using System.IO;
using Photon.Pun;
using UnityEngine;

public class PlayerManager : MonoBehaviourPun
{
    #region Public Fields

    [Tooltip("Prefabbricato del UI del giocatore")]
    [SerializeField]
    public GameObject PlayerUIPrefab;

    public GameObject casellaPlayer;

    public Camera mycam;
    public AudioListener myal;

    PhotonView pv;
    int x, y;

    public static GameObject LocalPlayerInstance;

    #endregion

    void Start()
    {
        //Istanzio L'interfaccia del Player1
        if (PlayerUIPrefab != null)
        {
            GameObject _uiGo = Instantiate(PlayerUIPrefab);
            _uiGo.SendMessage("SetTarget", this, SendMessageOptions.RequireReceiver);
        }

        pv = GetComponent<PhotonView>();
        if (pv.IsMine)
        {
            if (PhotonNetwork.IsMasterClient)
            {
                #region Tavole di Gioco

                //Istanziamo la Prima Tabella di Gioco
                for (x = 1; x < 11; x++)
                {
                    for (y = 1; y < 11; y++)
                    {
                        casellaPlayer =
                        PhotonNetwork.Instantiate(Path.Combine("PhotonPrefabs", "Casella"), new
                        Vector3(-500 + (x * 90), 1, -1000 + (y * 90)), Quaternion.identity, 0);

                        setCasella(casellaPlayer, x, y, 1);

                        Tavola.table1Player1[x, y] = casellaPlayer.GetComponent<Casella>();
                    }
                }
                //Istanziamo la Seconda tabella di Gioco
                for (int x = 1; x < 11; ++x)
                {
                    for (int y = 1; y < 11; ++y)
                    {

```

```

casellaPlayer = PhotonNetwork.Instantiate(Path.Combine("PhotonPrefabs",
"Casella"), new Vector3(-500 + (x * 90), 1, 0 + (y * 90)),
Quaternion.identity, 0);

        setCasella(casellaPlayer, x, y, 2);

Tavola.table2Player1[x, y] = casellaPlayer.GetComponent<Casella>();
    }
}

#endregion
}
}
else
{
//Facciamo in modo che il Player utilizzi la Camera del figlio
    Destroy(mycam);
    Destroy(myal);
}
}

//Prendiamo lo script presente nel GameObject Casella che abbiamo
istanziato e vi inseriamo le informazioni di base
public void setCasella (GameObject casellaPlayer, int x, int y, int
numtable)
{
    casellaPlayer.GetComponent<Casella>().SetRiga(x);
    casellaPlayer.GetComponent<Casella>().SetColonna(y);
    casellaPlayer.GetComponent<Casella>().SetTable(numtable);
    casellaPlayer.GetComponent<Casella>().SetPlayerTable(1);
}

void Awake()
{
    //teniamo traccia dell'istanza del giocatore Locale per impedire la
    creazione di istanze quando i livelli sono sincronizzati
    if(photonView.IsMine)
    {
        PlayerManager.LocalPlayerInstance = this.gameObject;
        DontDestroyOnLoad(this.gameObject);
    }
}

void Update()
{
    if (photonView.IsMine == false && PhotonNetwork.IsConnected == true)
    {
        return;
    }
}

void CalledOnLevelWasLoaded(int level)
{
    //Settiamo il NickName sull'interfaccia
    GameObject _uiGo = Instantiate(this.PlayerUIPrefab);
    _uiGo.SendMessage("SetTarget", this,
SendMessageOptions.RequireReceiver);
}
}

```

PlayerUI

```
using UnityEngine;
using UnityEngine.UI;

public class PlayerUI : MonoBehaviour
{
    #region Private Fields

    [Tooltip("UI: Testo del Nome Player sul Display")]
    [SerializeField]
    private Text playerNameText;

    [Tooltip("UI: Testo del Nome Player sul Display")]
    [SerializeField]
    private Text playerNameText2;

    #endregion

    private PlayerManager target;
    private PlayerManagerP2 target2;

    #region MonoBehaviour Callbacks

    void Awake()
    {
        this.transform.SetParent(GameObject.Find("Canvas").GetComponent<Transform>(), false);
    }

    #endregion

    #region Public Methods

    public void SetTarget(PlayerManager _target)
    {
        if(_target == null)
        { return; }
        target = _target;
        if (playerNameText != null)
        {
            playerNameText.text = target.photonView.Owner.NickName;
        }
    }

    public void SetTarget2(PlayerManagerP2 pm2)
    {
        if (pm2 == null)
        { return; }
        target2 = pm2;
        if (playerNameText2 != null)
        {
            playerNameText2.text = target2.photonView.Owner.NickName;
        }
    }

    #endregion
}
```

Tavola

```

using Photon.Pun;
using UnityEngine;

public class Tavola : MonoBehaviour, IPunObservable
{
    #region Public Fields

    //Dichiaro le 4 tavole di gioco
    public static Casella[,] table1Player1 = new Casella[11,11];
    public static Casella[,] table2Player1 = new Casella[11,11];
    public static Casella[,] table1Player2 = new Casella[11,11];
    public static Casella[,] table2Player2 = new Casella[11,11];

    #endregion

    void Update()
    {
        if (PhotonNetwork.PlayerList.Length == 2)
        {
            Controllo se il Player2 ha posizionato una nave, e modifico anche tavola
            for (int i = 1; i < 11; i++)
            {
                for (int j = 1; j < 11; j++)
                {
                    #region Player1

                    //Posizionamento Nave - Controllo se i valori coincidono
                    if (Tavola.table1Player2[i, j].naveposizionataP2 !=
                        Tavola.table2Player1[i, j].naveposizionataP2)
                    {
                        //Controllo se il Player2 ha posizonato una nave e l'aggiungo anche al
                        //Player1
                        if(Tavola.table1Player2[i,j].naveposizionataP2)
                            Tavola.table2Player1[i,j].PosizionaNaveP2();
                    }

                    //Controllo se qualche mia nave è stata affondata
                    if (Tavola.table2Player2[i, j].p2affondaP1 !=
                        Tavola.table1Player1[i, j].p2affondaP1)
                    {
                        //Controllo se il Player2 ha affondato una mia nave e aggiornno tavola
                        if (Tavola.table2Player2[i, j].p2affondaP1)
                        {
                            Tavola.table1Player1[i, j].p2affondaP1 = true;
                            Tavola.table1Player1[i, j].colpitaP1 = true;
                        }
                    }

                    //Controllo se qualche mia casella è stata colpita
                    if (Tavola.table2Player2[i, j].colpitaP1 !=
                        Tavola.table1Player1[i, j].colpitaP1)
                    {
                        //Controllo se il Player2 ha colpito una mia casella e aggiornno tavola
                        if (Tavola.table2Player2[i, j].colpitaP1)
                            Tavola.table1Player1[i, j].colpitaP1 = true;
                    }

                    #endregion
                }
            }
        }
    }
}

```

```

        #region Player2

        //Posizionamento Nave - Controllo se i valori coincidono
        if (Tavola.table1Player1[i, j].naveposizionataP1 !=
            Tavola.table2Player2[i, j].naveposizionataP1)
        {
            //Controllo se il Player1 ha posizonato una nave e l'aggiungo anche al
            Player1
            if (Tavola.table1Player1[i, j].naveposizionataP1)
                Tavola.table1Player2[i, j].PosizionaNaveP1();
        }
        //Controllo se qualche mia nave è stata affondata
        if (Tavola.table2Player1[i, j].plaffondaP2 !=
            Tavola.table1Player2[i, j].plaffondaP2)
        {
            //Controllo se il Player1 ha affondato una mia nave e aggiorno tavola
            if (Tavola.table2Player1[i, j].plaffondaP2)
            {
                Tavola.table1Player2[i, j].plaffondaP2 = true;
                Tavola.table1Player2[i, j].colpitaP2 = true;
            }
        }

        //Controllo se qualche mia casella è stata colpita
        if (Tavola.table2Player1[i, j].colpitaP2 !=
            Tavola.table1Player2[i, j].colpitaP2)
        {
            //Controllo se il Player1 ha colpito una mia casella e aggiorno tavola
            if (Tavola.table2Player1[i, j].colpitaP2)
                Tavola.table1Player2[i, j].colpitaP2 = true;
        }
        #endregion
    }
}

}

public void OnPhotonSerializeView(PhotonStream stream,
PhotonMessageInfo info)
{
    if (stream.IsWriting)
    {
        stream.SendNext(table2Player1);
        stream.SendNext(table2Player2);
        stream.SendNext(table1Player1);
        stream.SendNext(table1Player2);
    }
    else
    {
        //Lettore di rete, ricevi dati
        Tavola.table1Player2 = (Casella[,])stream.ReceiveNext();
        Tavola.table2Player2 = (Casella[,])stream.ReceiveNext();
        Tavola.table1Player1 = (Casella[,])stream.ReceiveNext();
        Tavola.table2Player1 = (Casella[,])stream.ReceiveNext();
    }
}
}
}

```


TextUI

```
using Photon.Pun;
using UnityEngine;
using UnityEngine.UI;

public class TextUI : MonoBehaviour
{
    #region Public Fields

    public Text turno;
    public Text numeroturno;
    public Text playerturno;
    public Text startgame;
    float tm = 300;

    public Slider sliderPlayer1;
    public Slider sliderPlayer2;

    #endregion
    // Start is called before the first frame update
    void Start()
    {
        numeroturno.text = " ";
    }

    // Update is called once per frame
    void Update()
    {
        //Se il gioco è iniziato, faccio comparire le scritte in merito al
        turno
        if (GameController.iniziogioco)
        {
            turno.GetComponent<Text>().enabled = true;
            playerturno.GetComponent<Text>().enabled = true;
            //textiniziogioco();
        }

        //Modifico il contenuto delle scritte in base al giocatore che le
        visualizza
        if (PhotonNetwork.IsMasterClient)
        {
            //Modifico il numero dei turni
            numeroturno.GetComponent<Text>().enabled = true;
            numeroturno.text = " " + GameController.numeroTurnoPub;

            //A seconda del turno, setto la scritta al ogni giocatore
            if (GameController.turnoPub)
            {
                playerturno.text = "è il tuo turno";
            }
            else
            {
                playerturno.text = "è il turno dell'avversario";
            }

            sliderPlayer1.value = GameController.nv1;
            sliderPlayer2.value= GameController.naviaffondate2;
        }
        else
        {

```

```
//Modifico il numero dei turni
numeroturno.GetComponent<Text>().enabled = true;
numeroturno.text = " " + GameController.numeroTurnoPub;

//A seconda del turno, setto la scritta al ogni giocatore
if (!GameController.turnoPub)
{
    playerturno.text = "è il tuo turno";
}
else
{
    playerturno.text = "è il turno dell'avversario";
}

sliderPlayer1.value = GameController.nv1;
sliderPlayer2.value= GameController.naviaffondate2;
    }
}
}
```

Bibliografia

- [1] BattleShip game. *Wikipedia-en BattleShip Game* [https://en.wikipedia.org/wiki/Battleship_\(game\)](https://en.wikipedia.org/wiki/Battleship_(game))
- [2] Unity Technologies. *Scripting API: GameObject.FindGameObjectsWithTag*. <https://docs.unity3d.com/ScriptReference/GameObject.FindGameObjectsWithTag.html>. 2020.
- [3] Unity Technologies. *Scripting API: GameObject.FindWithTag*. <https://docs.unity3d.com/ScriptReference/GameObject.FindWithTag.html>. 2020.
- [4] Unity Technologies. *Scripting API: GameObject.layer*. <https://docs.unity3d.com/ScriptReference/GameObject-layer.html>. 2020.
- [5] Unity Technologies. *Scripting API: GameObject.tag*. <https://docs.unity3d.com/ScriptReference/GameObject-tag.html>. 2020.
- [6] Unity Technologies. *Scripting API: NetworkDiscovery.OnReceivedBroadcast*. <https://docs.unity3d.com/2017.3/Documentation/ScriptReference/Networking.NetworkDiscovery.OnReceivedBroadcast.html>. 2020.
- [7] Unity Technologies. *Scripting API: NetworkDiscovery.StartAsClient*. <https://docs.unity3d.com/560/Documentation/ScriptReference/Networking.NetworkDiscovery.StartAsClient.html>. 2020.
- [8] Unity Technologies. *Scripting API: NetworkDiscovery.StartAsServer*. <https://docs.unity3d.com/2017.1/Documentation/ScriptReference/Networking.NetworkDiscovery.StartAsServer.html>. 2020.
- [9] Unity Technologies. *Scripting API: NetworkDiscovery.StopBroadcast*. <https://docs.unity3d.com/2017.3/Documentation/ScriptReference/Networking.NetworkDiscovery.StopBroadcast.html>. 2020.
- [10] Unity Technologies. *Scripting API: NetworkManager.StartClient*. <https://docs.unity3d.com/2018.1/Documentation/ScriptReference/Networking.NetworkManager.StartClient.html>. 2020.
- [11] Unity Technologies. *Scripting API: NetworkManager.StartHost*. <https://docs.unity3d.com/2018.1/Documentation/ScriptReference/Networking.NetworkManager.StartHost.html>. 2020.
- [12] Unity Technologies. *Unity Manual: Advanced operations: Using the LLAPI*. <https://docs.unity3d.com/Manual/UnityWebRequest-LLAPI.htm>. 2020.
- [13] Unity Technologies. *Unity Manual: AssetBundles*. <https://docs.unity3d.com/Manual/AssetBundlesIntro.html>. 2020.
- [14] Unity Technologies. *Unity Manual: Camera*. <https://docs.unity3d.com/Manual/class-Camera.html>. 2020.

- [15] Unity Technologies. *Unity Manual: GameObject*. <https://docs.unity3d.com/Manual/class-GameObject.html>. 2020.
- [16] Unity Technologies. *Unity Manual: Introduction to components*. <https://docs.unity3d.com/Manual/Components.html>. 2020.
- [17] Photon Engine. *Photon Engine*. <https://www.photonengine.com>. 2020.
- [18] Photon Engine. *Realtime*. <https://www.photonengine.com/en-US/Realtime>.
- [19] Photon Engine. *Photon Unity Network – PUN*. <https://www.photonengine.com/en-US/PUN>
- [20] Photon Engine. *Bolt* <https://www.photonengine.com/en-US/BOLT>
- [21] Photon Engine. *Quantum* <https://www.photonengine.com/en-US/Quantum>
- [22] Photon Engine. *Chat* <https://www.photonengine.com/en-US/Chat>
- [23] Photon Engine. *Voice* <https://www.photonengine.com/en-US/Voice>
- [24] Photon Engine. *Server* <https://www.photonengine.com/en-US/Server>
- [25] Photon Engine. *PUN Tutorial* <https://doc.photonengine.com/en-us/pun/v2/demos-and-tutorials/pun-basics-tutorial/intro>
- [26] Photon Engine. *PUN Glossary* <https://doc.photonengine.com/en-us/realtime/current/reference/glossary>
- [27] Photon Engine. *PhotonNetwork Class Reference* https://doc-api.photonengine.com/en/pun/v1/class_photon_network.html#af498064a6019a6c69e875bd64db40216

*“Happiness can be found, even in the darkest of times,
if one only remembers to turn on the light.”*

- Albus Percival Wulfric Brian Dumbledore

Ringraziamenti

*Il primo ringraziamento, forse il più importante, va a **me stessa**.
Sin dal primo giorno di Università ho sempre immaginato questo giorno in maniera
differente ma l'importante è essere arrivata, in un modo o nell'altro, a questo ambito
traguardo. Le persone come me tendono a ricordare ogni singolo momento o data
associandone un significato speciale e un ricordo indelebile.
Ricordo ogni avvenimento di questo percorso,
partendo dal giorno in cui ero una matricola sola e impaurita.
Impaziente di intraprendere questa nuova avventura,
terrorizzata dall'idea di non essere all'altezza o di non essere abbastanza.
Questo percorso mi ha portato a compiere molte esperienze e a crescere professionalmente e
personalmente, rendendomi una donna forte e consapevole del suo valore.*

*Ringrazio la mia **famiglia**,
che mi ha sempre sostenuta, in particolare, mia **mamma** che è sempre stata al mio fianco, mia
sorella **Roberta** per essere stata una sorella buona e comprensiva, la mia valvola di sfogo in
alcuni momenti di tristezza, e la piccola **Birba** per avermi strappato un sorriso e ridotto lo
stress nei momenti peggiori. Grazie perché avete sempre creduto in me.*

*Ringrazio i miei amici **Ciro, Chicca, Rachele, Claudia e Francesca**,
siete state le prime persone che ho conosciuto, avete accolto questa matricola impaurita e avete
condiviso con lei ore ed ore di studio, sofferenze e lacrime. Ma anche tanti bei ricordi.
Soprattutto caffè...*

*Ringrazio i miei amici "quasi normali" **Elio, Salvatore e Simone**,
per aver ascoltato tutti i miei drammi e le mie lamentele, per aver condiviso con me gioie e
dolori, soprattutto aperitivi, e per esserci stati sempre nonostante liti, urla e distanza.*

*Ringrazio i miei compagni di progetto **Alfredo, Biagio, Bomba, Eugenio e Nicola**
Per aver condiviso con me l'ansia in merito ad uno degli esami più impegnativi di tutta la
carriera, per le risate e per i bei momenti vissuti insieme. Grazie a voi ho capito l'importanza
di lavorare in un team e che quando si è in ottima compagnia il lavoro risulta meno faticoso
... o quasi.*

Ringraziamenti

Ringrazio **Rosa e Anna**,
*per aver subito tutte le mie lamentele, le mie paure e le mie ansie,
pur non sapendo di cosa io stessi parlando.*

Ringrazio **Falletti**,
*per essermi stato vicino, per essere stato la mia spalla su cui piangere, per essersi subito tutti i
miei drammi e per tutti gli elogi e le parole di conforto che mi ha dato.*

Ringrazio **Federico**,
*per essermi stato vicino quando ne avevo più bisogno, per essere stato comprensivo
e per non avermi fatto perdere la sanità mentale durante la quarantena
grazie alle recensioni delle puntate di SAO.*

Ringrazio **Roberto**,
*per essere la persona più buona, intelligente e disponibile che io conosca,
per la tua infinita pazienza nei miei confronti e per il tuo costante aiuto.*

Ringrazio **Simona**,
*per aver sempre creduto in me, per il continuo sostegno e
per essermi stata vicino ed avermi supportata sempre.*

Ringrazio **Matilde**,
*poiché mi ricordi costantemente che l'amicizia va ben oltre al vedersi tutti i giorni.
Cinque anni a condividere avventure e sventure creano un legame indistruttibile.*

Un ringraziamento speciale va a **Lorenzo**,
*Sei la persona che più di tutte ha notato il mio valore
ed è riuscita a spronarmi facendolo notare anche a me.
Con il tuo amore e i tuoi modi di fare sei riuscito a darmi la forza di abbattere
le barriere che mi ero creata e superare quei limiti oltre quali non mi ero mai spinta,
conquistando traguardi che mai avrei sognato di raggiungere.*

E a te, se sei rimasto con me fin proprio alla fine.