# Murder Mystery V2

## Abstract

Murder Mystery è un gioco per telefono multiplayer che prende spunto da l'omonima modalità presente in Minecraft ormai non più popolare e realizzato con lo scopo di onorarla e consiste nell'uccidere un assassino prima che lui faccia lo stesso.
Il progetto è scomposto in due parti: **Server e Client**.

## Indice

# Generalità

## Gioco

I player vengono istanziati in una lobby che coincide col terreno di gioco, al raggiungimento della quota minima di giocatori (3, massimo 10), inizia un countdown e al suo termine inizia il vero e proprio gioco, sullo schermo di ogni player verrà stampato il loro ruolo scelto casualmente tra:

- Innocent: player normale.
- Detective: velocità superiore agli Innocenti e possiede una pistola
- Murderer: possiede un coltello da lancio col quale uccide i player e aumenta la sua velocità temporaneamente dopo aver fatto ciò.

Nel gioco sono presenti dei cubi gialli che vengono istanziati casualmente per la mappa, prendendo i cubetti gialli fino a raggiungere una certa soglia un Innocente può ottenere una pistola monouso e un Murderer ottiene un nuovo coltello.
Il Murderer vince se uccide tutti i player e i player vincono se uccidono il Murderer.

## Target

Ragazzi dai 18+ anni con telefoni android (minimo android 5.1).

## Programmi utilizzati

- Blender: Modelli 3D
- Unity: Ambiente di sviluppo
- Visual Studio Code: Editor di testo
- GitHub: Pubblicazione progetto Open Source

## Collaboratori:

- **Giulio Biscontin : Hosting Server e Game Tester**
- Qing Qing Liu: Voice Actress
- Laxman Cozzarin : Audio Designer
- Teodoro Capacchione : Game Developer e 3d Designer

## Fonti utilizzate

Unity Networking series

## Tempo impiegato

Circa. 100 ore

## Bug

- Pistola non si distrugge quando colpisce innocente
- Pistola non viene istanziata quando si prende il Gold

- La visuale potrebbe scattare per una mal assegnazione degli ID

Questi bug verranno risolti in futuro in fase di rilascio e test da parte del pubblico.

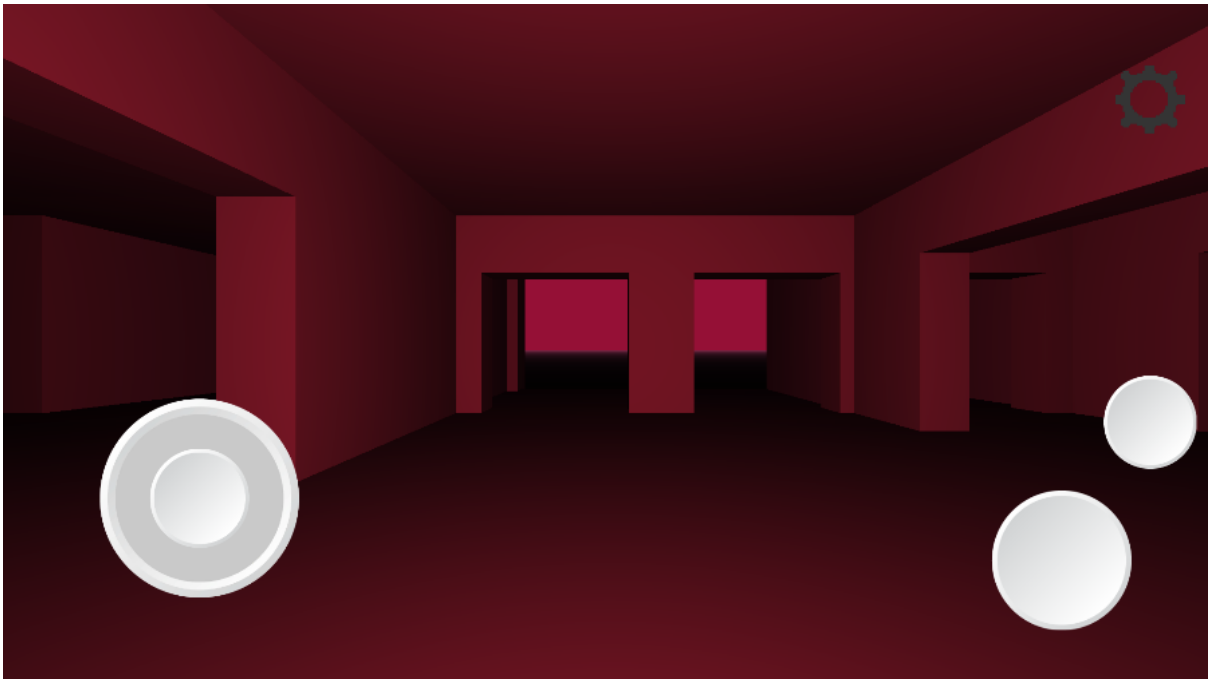
# Struttura

## Client

Il client è formato da 2 schermate principali:

- **Menu:**schermata adibita all’instaurazione della connessione col server,alla modifica delle opzioni e all'inserimento dati.

- **Gioco:**schermata nella quale si svolgono tutte le operazioni necessarie alla comunicazione col server e allo svolgimento del gioco.

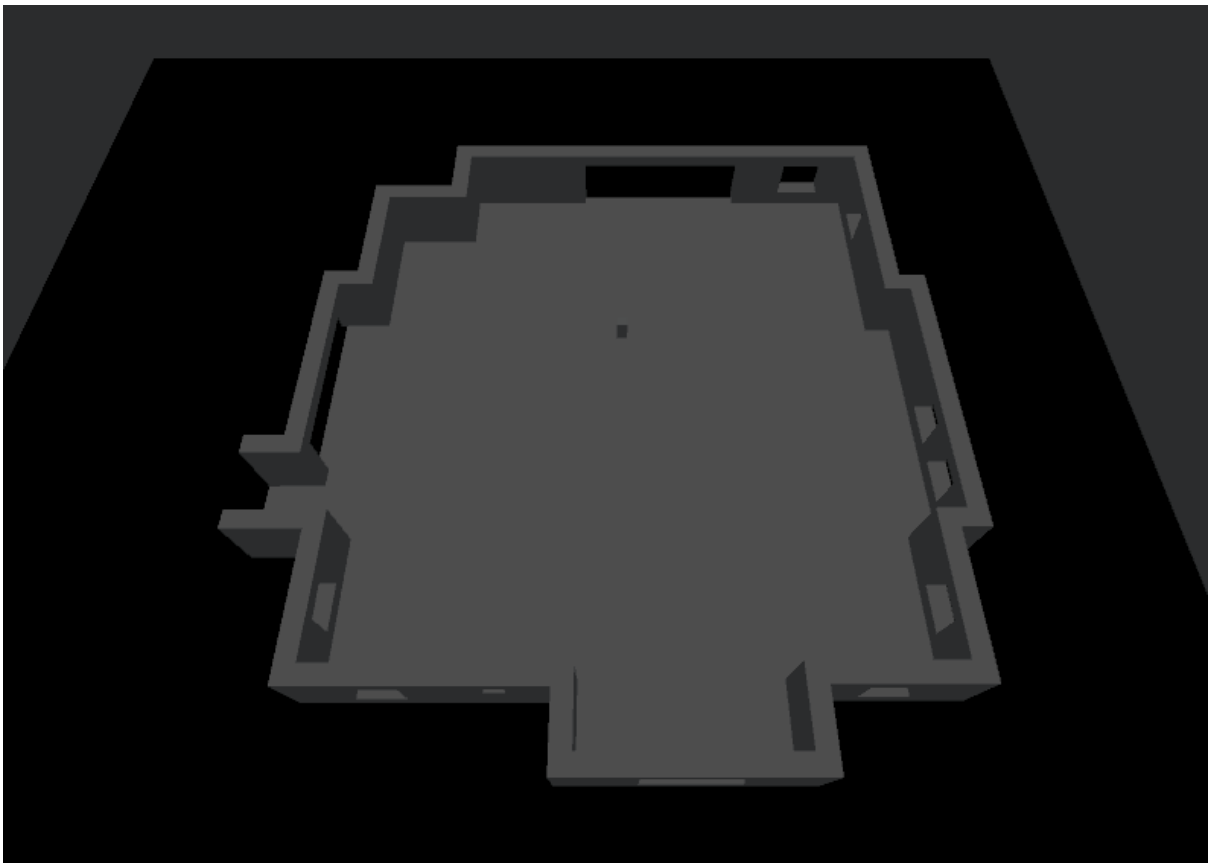


## Server

Il server è formato da una sola schermata:

- **Gioco:** al interno di questa schermata sono svolte tutte le operazioni necessarie alla comunicazione col client, all'analisi input del client e l'invio di pacchetti ai player oltre alle operazioni che si occupano delle meccaniche di gioco come la gravità o le

collisioni.



# Grafica

La grafica è in 3D ed è stata modellata attraverso Blender, un programma Open Source utilizzato per il modellamento 3D e la creazione di Film, per la maggior parte da Teodoro Capacchione della Cimut mentre la restante parte è stata  fornita da Unity.

## Modelli creati da Capacchione:

- Coltello
- Edificio
- Pistola

## Modelli già presenti su Unity:

- Cubo di default

## Modelli importati da Asset Store (Unity)

- Joystick

# Networking System

## Classi in comune

### Packet

Contiene una lista degli ID di tutti i possibili tipi di pacchetto

- enum ClientPackets: pacchetti inviabili dal Client
- enum ServerPackets: pacchetti inviabili dal Server

Un pacchetto ha come sua componente principale una lista di byte che contiene tutti i dati di uno specifico pacchetto.
La classe Packet fornisce oltretutto dei metodi per la scrittura(e quindi conversione) di diversi tipi di dato che vengono accodati alla lista di byte:
Non convertiti

- Byte
- Array di Byte

Convertiti

- short
- int
- long
- float
- bool

Array di dati per i quali deve stampare anche la lunghezza come un intero

- stringhe

Composti

- Posizione
- Rotazione

Allo stesso modo fornisce metodi per la lettura della lista di byte(e conversione se necessaria)

### ThreadManager

Il ThreadManager è un oggetto che viene utilizzato per l'esecuzione di azioni in modo asincrono.
La sua componente principale una lista di Action da eseguire e una lista di Action in esecuzione, a ogni frame la sua funzione Update viene chiamata e al suo interno viene copiata la lista di Action da eseguire nella lista in esecuzione e quindi viene fatta scorrere la lista e mandati in esecuzione ogni singola Action.
Per caricare la lista di Azioni si richiama il metodo
static void ExcecuteOnMainThread(Action _action), dove _action è una funzione richiamabile o Lambda.

## Lato Server

### Client

La classe Client rappresenta uno client connesso al server.

Al suo interno sono implementate delle funzioni che non possono essere eseguite sul player, come:

1. SendIntoGame istanzia il player e lo invia ai vari client, diviso in tre fasi
    a. Creazione Player su server e posizionamento
    b. Invio a tutti i Client un messaggio per l'istanziazione del player in locale
    c. Invio del messaggio d'istanziazione di tutti gli altri Player al Client del player appena entrato.
2. UpdatePlayer aggiorna il ruolo di un player eliminandolo e sostituendolo, funziona in modo analogo a SentIntoGame ma invia anche un messaggio di GameBegin in per attivare le animazioni d'inizio game sui clients.
3. Disconnect implementata in tre passaggi:
    a. Eliminazione Player dal server.
    b. Disconnessione del socket TCP e riassegnazione UDP endpoint.
    c. Invio di messaggio di disconnessione ai players rimanenti nel gioco.

Oltre alle funzioni la classe Client contiene:

1. Un ID
2. Istanza al Player al quale è collegato
3. Un'istanza della classe TCP e la sua definizione
4. Un'istanza della classe UDP e la sua definizione

```
public class Client
{
    //Dimensione Buffer Lettura/Scrittura
    public static int dataBufferSize = 4096;
    //ID di riconoscimento
    public int id;
    //Istanza del Player Server
    public Player player;
    //Classi Invio/Ricezione Dati
    public TCP tcp;
    public UDP udp;
    //Costruttore
    public Client(int _clientId)
    {
        id = _clientId;
        //Assegnazione ID per Debug
        tcp = new TCP(id);
        udp = new UDP(id);
    }
    //Definizione Classi TCP/UDP
    public class TCP...
    public class UDP...
    //Funzione usata per distruggere il player
    public void DestroyPlayer()...
    //Invio Player ai Client
    public void SendIntoGame()...
```

```
    //Modifica Ruolo di un player
    public void UpdatePlayer(int _newrole,Vector3 _position)...
    //Disconnessione Client
    public void Disconnect()...
}
```

## ServerHandle

Server handle è una classe di funzioni statiche che implementano l'analisi pacchetti e procedure a loro associate.
Prendono come parametro l'ID del mittente e il Packet da esso inviato(sono in quantità pari ai tipi di pacchetti definiti in ClienPackets).

```
public class ServerHandle
{
    public static void WelcomeReceived(int _fromClient, Packet _packet)...
    public static void Ready(int _fromClient,Packet _packet)...
    public static void PlayerMovement(int _fromClient, Packet _packet)...
    public static void PlayerShoot(int _fromClient, Packet _packet)...
}
```

## ServerSend

Server send è una classe di metodi statici che offre procedure per l'invio di pacchetti sia UDP che TCP ,oltre a implementare l'inizializzazione di quelli prossimi alla spedizione.
Quando un pacchetto viene caricato la prima cosa scritta al suo interno è il packetID (descritto in "enum ClientPackets") e l'ultima è la sua lunghezza, inserita in testa.
Qui sotto illustrato anche un esempio d'inizializzazione pacchetto.

```
public class ServerSend
{
    private static void SendTCPData(int _toClient, Packet _packet)
    {
        _packet.WriteLength();
        Server.clients[_toClient].tcp.SendData(_packet);
    }

    private static void SendUDPData(int _toClient, Packet _packet)
    {
        _packet.WriteLength();
        Server.clients[_toClient].udp.SendData(_packet);
    }
    //Invia un pacchetto TCP a tutti i client connessi
    private static void SendTCPDataToAll(Packet _packet)...
    //Invia un pacchetto TCP a tutti i client connessi tranne uno
    private static void SendTCPDataToAll(int _exceptClient, Packet _packet)...
    //Invia un pacchetto UDP a tutti i client connessi
```

```
private static void SendUDPDataToAll(Packet _packet)...
//Invia un pacchetto UDP a tutti i client connessi tranne uno
private static void SendUDPDataToAll(int _exceptClient, Packet _packet)...
//Pacchetto welcome
public static void Welcome(int _toClient, string _msg)
{
    using (Packet _packet = new Packet((int)ServerPackets.welcome))
    {
        _packet.Write(_msg);
        _packet.Write(_toClient);

        SendTCPData(_toClient, _packet);
    }
}
public static void SpawnPlayer(int _toClient, Player _player)...
public static void SpawnGhost(int _toClient, Player _player)...
public static void PlayerPosition(Player _player)...
public static void PlayerRotation(Player _player)...
public static void PlayerDisconnected(int _playerId)...
```

*Non sono illustrati tutte le procedure perché occuperebbero troppo spazio

## Server

Server è una classe di metodi e attributi statici che servono a gestire connessioni e pacchetti in arrivo.
Gli attributi essenziali per la gestione del server sono:

- clients: un static Dictionary<int,Client> utilizzato per tenere traccia dei vari Client e i loro ID.
- void PacketHandler(int _fromClient,Packet _packet): un delegate, definisce i parametri e il valore di ritorno di un tipo di funzioni, se paragonata a java si potrebbe definire quasi come un'interfaccia, che garantisce la presenza di un metodo specifico.
- packetHandlers: un static Dictionary<int,PacketHandler> utilizzato per raccogliere l'insieme di tipi di Packet ricevibili dal Client (listati in ClientPackets) e la funzione con la quale vengono processati.
- tcpListener: un TcpListener che si occupa di accettare le connessioni TCP.
- udpListener: un UdpClient (il nome non coincide con la classe), che si occupa di accettare e inviare pacchetti UDP.
- Port: un intero che contiene il valore della porta usata per la connessione.
- MaxPlayers: un intero che contiene il numero massimo di Client collegabili al server.

```
public static int MaxPlayers{ get; private set; }
public static int Port { get; private set; }
public static Dictionary<int, Client> clients = new Dictionary<int, Client>();
public delegate void PacketHandler(int _fromClient, Packet _packet);
public static Dictionary<int, PacketHandler> packetHandlers;
```

```
        private static TcpListener tcpListener;
        private static UdpClient udpListener;
```

I metodi statici usati per l'accettazione di connessioni e pacchetti, vengono lanciati come coroutine:

- TCPConnectCallback
- UDPReceiveCallback

mentre gli altri sono richiamati solo quando necessario:

- Start
- Stop
- SendUDPData
- InitializeServerData

### Start

Un metodo void che riceve come parametri il numero massimo di giocatori e la porta sulla quale si effettua la connessione coi vari client.
Esso (insieme a InitializeServerData)serve a inizializzare tutti gli attributi statici della classe, oltre a far partire il TcpListener e a dare inizio alle coroutine di accettazione client TCP e pacchetti UDP.

```
public static void Start(int _maxPlayers, int _port)
    {
        try{
            Server.Stop();
        }
        catch{
        }
        MaxPlayers = _maxPlayers;
        Port = _port;

        Debug.Log("Starting server...");
        InitializeServerData();

        tcpListener = new TcpListener(IPAddress.Any, Port);
        tcpListener.Start();
        tcpListener.BeginAcceptTcpClient(TCPConnectCallback, null);

        udpListener = new UdpClient(Port);
        udpListener.BeginReceive(UDPReceiveCallback, null);

        Debug.Log($"Server started on port {Port}.");
    }
```

### InizializeServerData

Un metodo void chiamato solo in Start che serve a inizializzare i due dizionari statici della classe Server:

- clients: viene caricato con un numero uguale a MaxPlayers di Client e gli si assegna un ID univoco (ai client già inizializzati vengono associati i socket TCP accettati in seguito).
- packetHandlers: viene caricato con tutti i valori di ClientPackets e i loro PacketHandler associati.

```
private static void InitializeServerData()
    {
        clients.Clear();
        for (int i = 1; i <= MaxPlayers; i++)
        {
            clients.Add(i, new Client(i));
        }

        packetHandlers = new Dictionary<int, PacketHandler>()
        {
            { (int)ClientPackets.welcomeReceived, ServerHandle.WelcomeReceived },
            { (int)ClientPackets.playerMovement, ServerHandle.PlayerMovement },
            { (int)ClientPackets.playerShoot, ServerHandle.PlayerShoot },
            { (int)ClientPackets.ready, ServerHandle.Ready }
        };
        Debug.Log("Initialized packets.");
    }
```

### TcpConnectCallback

Un metodo void che prende come parametro un IAsyncResult, viene richiamata come coroutine ogni volta che viene accettata una connessione.
Viene utilizzata per l'accettazione di connessioni TCP che verranno assegnate al primo Client libero.

```
private static void TCPConnectCallback(IAsyncResult _result)
    {
            //Connessione arrivata
        TcpClient _client = tcpListener.EndAcceptTcpClient(_result);
            //Attesa connessione
        tcpListener.BeginAcceptTcpClient(TCPConnectCallback, null);
        Debug.Log($"Incoming connection from {_client.Client.RemoteEndPoint}...");

        for (int i = 1; i <= MaxPlayers; i++)
        {
            if (clients[i].tcp.socket == null)
            {
                Debug.Log("Second found");
                clients[i].tcp.Connect(_client);
                return;
```

```
        }
    }
    Debug.Log($"{_client.Client.RemoteEndPoint} failed to connect: Server full!");
}
```

## UDPReceiveCallback

Un metodo void che prende come parametro un IAsyncResult, viene usato per la ricezione dei pacchetti e per l'assegnazione di endPoint ai Client.

```
private static void UDPReceiveCallback(IAsyncResult _result) {
    try{
        //Ricevo un pacchetto e mi rimetto in attesa
        IPEndPoint _clientEndPoint = new IPEndPoint(IPAddress.Any, 0);
        byte[] _data = udpListener.EndReceive(_result, ref _clientEndPoint);
        udpListener.BeginReceive(UDPReceiveCallback, null);
        //Lettura Pacchetto
        if (_data.Length < 4)return;
        using (Packet _packet = new Packet(_data))
        {
            int _clientId = _packet.ReadInt();
            if (_clientId == 0)return;
        //Se ( EndPoint _clientID è null ) lo assegno
            if (clients[_clientId].udp.endPoint == null)
            {
                clients[_clientId].udp.Connect(_clientEndPoint);
                return;
            }
            if (clients[_clientId].udp.endPoint.ToString() == _clientEndPoint.ToString())
            {
                clients[_clientId].udp.HandleData(_packet);
            }
        }
    }
    catch (Exception _ex){
        Debug.Log($"Error receiving UDP data: {_ex}");
    }
}
```

## SendUDPData

Un metodo void che prende come parametri l'IPEndPoint del destinatario e il Packet da inviare.
Serve ad inviare i pacchetti UDP.

```
public static void SendUDPData(IPEndPoint _clientEndPoint, Packet _packet){
    try {
        if (_clientEndPoint != null){
```

```
            udpListener.BeginSend(_packet.ToArray(), _packet.Length(), _clientEndPoint, null, null);
        }
    }
    catch (Exception _ex){
        Debug.Log($"Error sending data to {_clientEndPoint} via UDP: {_ex}");
    }
}
```

### Stop

Metodo void che serve a disconnettere tutti i Client collegati al server e chiudere il TcpListener e UdpClient.

```
public static void Stop(){
    foreach(Client _client in clients.Values){
        _client.Disconnect();
    }
    tcpListener.Stop();
    udpListener.Close();
}
```

## Lato Client

### Client

La classe Client rappresenta il client ed è unica all'interno del gioco, essa serve per instaurare connessioni TCP e ricevere pacchetti UDP(Molto simile alle classi Client e Server della versione Server del programma).
Gli attributi necessari per il funzionamento del Client sono:

- instance: un oggetto statico di tipo Client che serve come riferimento per la classe(se vengono creati 2 client in automatico uno verrà eliminato).
- dataBufferSize: un intero che contiene la dimensione massima di un flusso dati.
- defaultip e ip: rispettivamente l'ip di default e l'ip utilizzato, esistono per implementare il cambio ip dell'utente.
- port: un intero che contiene il valore della porta del server a cui collegarsi.
- myID: un intero assegnato dal Server che identifica il client
- TCP e UDP: due classi definite dentro la classe Client che si occupano dell'invio e ricezione dati tramite TCP e UDP.
- isConnected: una variabile booleana che contiene lo stato della connessione del client.
- void PacketHandler(Packet _packet): un delegate che definisce i parametri e il valore di ritorno di tutte le funzioni che analizzano i Packet.
- packetHandlers: un dictionary statico che contiene l'elenco di tutte le funzioni appartenenti a ClientHandle associate al loro corrispettivo valore ServerPackets.

```
public static Client instance;
    public static int dataBufferSize = 4096;
    public string defaultip = "127.0.0.1";
```

```
public string ip = "127.0.0.1";
public int port = 26950;
public int myId = 0;
public TCP tcp;
public UDP udp;

public bool isConnected = false;
private delegate void PacketHandler(Packet _packet);
private static Dictionary<int, PacketHandler> packetHandlers;
```

## Funzioni Client

### Awake

Viene avviata solo quando l'oggetto di unity al quale Client è associato viene creato, serve a controllare che non ci siano altre istanze di Client nel gioco e in quel caso le elimina.

```
private void Awake()
    {
        DontDestroyOnLoad(gameObject);
        if (instance == null)
        {
            instance = this;
            ip = defaultip;
        }
        else if (instance != this)
        {
            Debug.Log("Instance already exists, destroying object!");
            Destroy(this);
        }
    }
```

### ConnetToServer e Disconnect

Sono le funzioni utilizzate per la connessione e disconnessione del client dal server si occupano anche del inizializzazione degli attributi di Client.
La disconnessione viene chiamata in automatico quando il gioco viene chiuso.

```
public bool ConnectToServer(){
        tcp = new TCP();
        udp = new UDP();

        InitializeClientData();

        // Connetto al server
```

```
        return tcp.Connect();
    }

public void Disconnect()
    {
        if(tcp.socket.Connected){
            if (isConnected)
            {
                Debug.Log("Improvvisazione");
                isConnected = false;
                tcp.socket.Close();
                udp.socket.Close();
                GameManager.instance.endGame=true;
            }
        }
    }
```

### InitializeClientData

Serve ad inizializzare packetHandlers con tanti PackeHandler quanti i tipi di pacchetto definiti in ServerPackets.

```
private void InitializeClientData()
{
        packetHandlers = new Dictionary<int, PacketHandler>()
        {
            { (int)ServerPackets.welcome, ClientHandle.Welcome },
            { (int)ServerPackets.spawnPlayer, ClientHandle.SpawnPlayer },
            { (int)ServerPackets.playerPosition, ClientHandle.PlayerPosition },
            { (int)ServerPackets.playerRotation, ClientHandle.PlayerRotation },
            { (int)ServerPackets.playerDisconnected, ClientHandle.PlayerDisconnected },
             *per mancanza di spazio non sono inserite tutte le funzioni in questa documentazione
        };
        Debug.Log("Initialized packets.");
    }
```

## ClientHandle

Client handle è una classe di funzioni statiche che implementano l'analisi pacchetti e procedure a loro associate, prendono come parametro un Packet inviato dal Server e sono in quantità proporzionale ai tipi di pacchetti definiti in ServerPackets.

```
public class ClientHandle : MonoBehaviour{

    public static void Welcome(Packet _packet)...
    public static void SpawnPlayer(Packet _packet)...
```

```
    public static void PlayerPosition(Packet _packet)...
    public static void PlayerRotation(Packet _packet)...
    public static void PlayerDisconnected(Packet _packet)...
*non sono elencate tutte le funzioni per mancanza di spazio
}
```

## ClientSend

Client send è una classe di metodi statici che offre procedure per l'invio di pacchetti sia UDP che TCP oltre a implementare l'inizializzazione di quelli prossimi alla spedizione.
Quando un pacchetto viene caricato la prima cosa scritta al suo interno è il packetID (descritto in "enum ClientPackets") e l'ultima è la sua lunghezza inserita in testa.
Qui sotto illustrato anche un esempio d'inizializzazione pacchetto.

```
public class ClientSend : MonoBehaviour
{
    private static void SendTCPData(Packet _packet)
    {
        _packet.WriteLength();
        Client.instance.tcp.SendData(_packet);
    }

    private static void SendUDPData(Packet _packet)
    {
        _packet.WriteLength();
        Client.instance.udp.SendData(_packet);
    }

    public static void WelcomeReceived(){
        using (Packet _packet = new Packet((int)ClientPackets.welcomeReceived))
        {
            _packet.Write(Client.instance.myId);

            SendTCPData(_packet);
        }
    }
    public static void PlayerMovement(float[] _axes,bool[] _inputs)...
    public static void PlayerShoot()...
    public static void PlayerThrowItem(Vector3 _facing)...
    public static void Ready()...
}
```

# Meccaniche di Gioco

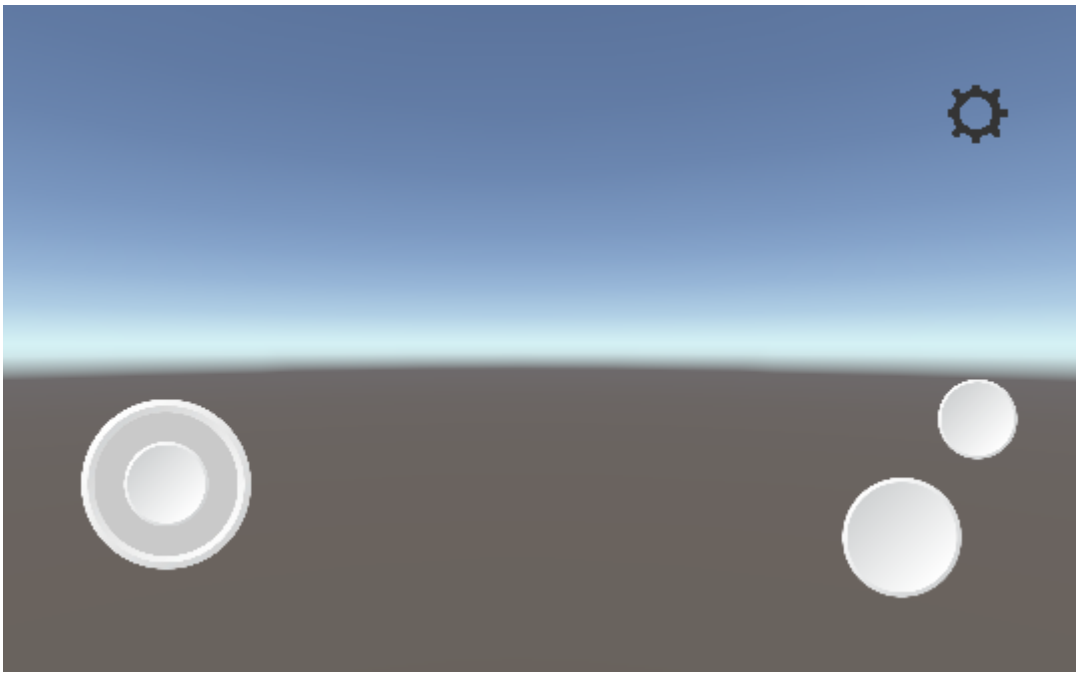
*Certe meccaniche di gioco potrebbero essere semplificate per coglierne meglio l'essenza

Lettura grafici:

- Entità Rettangolare = Entità presente sul Client.
- Entità Rettangolare Arrotondata = Entità presente sul Server

Se una freccia collega due rettangoli di tipo diverso sottintende tutti i processi elencati nella sezione Networking di comunicazione tra client e server.

## Player Interface



L'interfaccia reagisce in base all'input del player, partendo da sinistra

- il Joystick per il movimento: salva dentro a delle variabili statiche, contenute nella classe Joystick, la posizione del handle(parte più interna).
- Pulsante per il salto: è il pulsante più spazioso, quando premuto setta la variabile statica "pressed", contenuta dentro la classe Jump_Button, a true e false quando rilasciato(viene uttilizzato sia per il salto che per il volo verticale dei fantasmi).
- Pulsante per lo sparo:è il pulsante più piccolo, quando premuto manda un segnale di volo verticale verso il basso e quando rilasciato manda un segnale al server per richiedere lo sparo dell'arma del player che può essere un coltello o una pistola(Il bottone manda il segnale di sparo quando rilasciato permettendo ai player di mirare il bersaglio prima di sparare).
- Pulsante opzioni: è il pulsante con come simbolo un ingranaggio(pixel-art),serve per aprire il menu opzioni del player durante una partita che permette di uscire dalla partita.

# Player

## Server

La classe player del server rappresenta il personaggio controllato da un Client e racchiude tutte le informazioni e metodi utili a esso.
Player è l'oggetto soggetto a gravita, collisioni e altre procedure; la conseguenza di queste procedure (es: morte, movimento, etc...)viene inviata al Client che viene aggiornato costantemente.
Questa tecnica è stata utilizzata per rendere più complesso il bug abusing(sfruttamento di bug di gioco per concedere vantaggi a se stessi).

```
public class Player : MonoBehaviour{
    public int id;
    public CharacterController controller;//Componente usato per muovere il player
    public int gold;
    public int role = 0;//Ruolo
    public float gravity = -9.81f;
    public float moveSpeed = 5f;
    public float jumpSpeed = 10f;
    public float throwForce = 30f;
    public bool isRunning=false;
    private bool[] inputs;
    private float[] axes;
    private float yVelocity = 0;
    public Transform vision;//Telecamera
    public Transform weaponManager;//Fodero dell'arma
    public bool ghost=false;
    public LayerMask deathGround;

    private void Start()...
    public void Initialize(int _id)...
    //funzione richiamata in loop
    public void FixedUpdate()..
    private void Move(Vector2 _inputDirection)...
    public void SetInput(float[] _axes,bool[] _inputs, Quaternion _rotation,Quaternion _vision)...
    public void Shoot()...
    public void GetGold(int _gold)...
    public bool AttemptPickupItem()...
    public void Die()...
    public void Teleport(Vector3 _position)...
}
```

## Client

### PlayerController

La classe che si occupa della gestione input del giocatore che viene inviato al Server per l analisi.

```
public class PlayerController : MonoBehaviour
{
    Joystick joystick;
    Shoot_Button shoot_Button;
    Jump_Button jump_Button;
    public Transform camTransform;//Posizione telecamera
    bool shooted=false;
    private void Start()...
    private void FixedUpdate()... //Loop che richiama SendInputToServer
    private void SendInputToServer()...
}
```

### PlayerManager

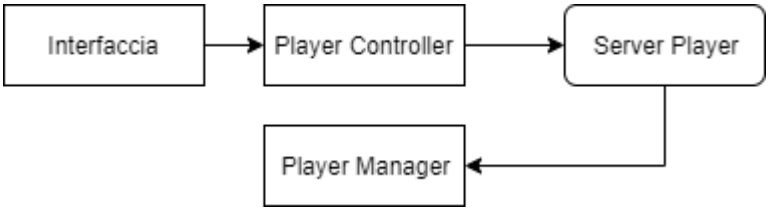
La classe che riceve input da parte del server, essa si occupa di posizionare il player e di attivare tutte le procedure grafiche oltre che a contenere tutti i dati più importanti della sua controparte sul server, i quali vengono costantemente aggiornati ad ogni modifica.

```
public class PlayerManager : MonoBehaviour
{
    public int id;
    public MeshRenderer model;//Grafica
    public int gold;
    public Transform weaponManager;//Fodero del arma
    public int role;
    public UI_Player ui;//GUI del Utente
    public void Initialize(int _id)...
    public void InitializeGhost(int _id)...
    public void RemoveWeapon()...
    public void PickUpKnife()...
    public void Gold(int _gold)...
    public void Die()...
    public void ShootEffect()...
    public void DestroyGun()...
    public void SpawnPistol()...
}
```

## Movimento e Salto

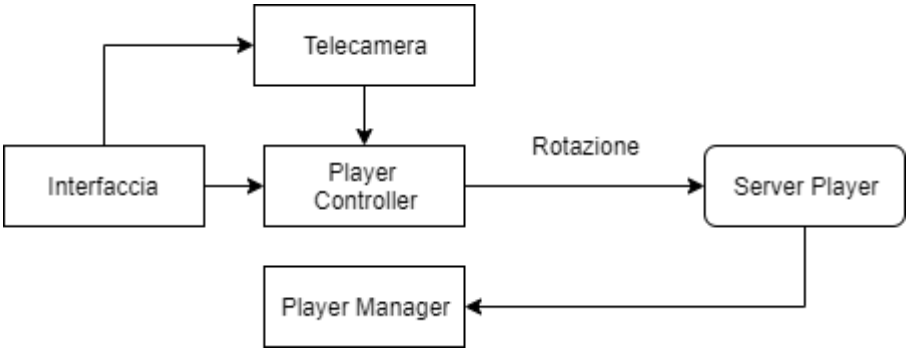
Il movimento del Player è implementato attraverso la sua interfaccia grafica che acquisisce l'input ,dal Player Controller che lo analizza e lo invia al Server che lo analizza e quindi

ritorna la posizione del player a tutti i Client che posizioneranno il giocatore nella loro istanza di gioco.




## Rotazione

La rotazione avviene in locale sul client e viene comunicata al server ,quindi a tutti i players collegati ad esso.
Lo schermo prende l'input di ogni tocco del player e in base al cambiamento di posizione del dito ruota il personaggio (orizzontalmente) e la telecamera (verticalmente).




## Schermo

Lo schermo si differenzia dall'interfaccia in quanto non è un oggetto come può esserlo un Panel o un Bottone ma è un sistema di rilevamento touch screen implementato per le piattaforme android e IOS che raccoglie dati sui tocchi quali numero,posizione e ID
(Ma per comodità viene trattato come Interfaccia).

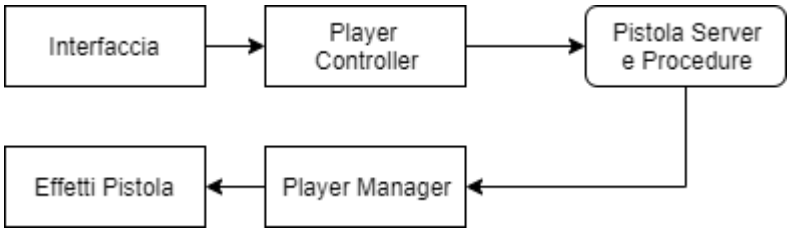
## PlayerController (Mouse Script)

Il Mouse Script(chiamato così perché la prima versione di Murder Mystery era esclusivamente per computer a scopo di progettazione) analizza l'input dello schermo in modo da limitare l'area di tocco (scartando i touch),da regolare il numero di tocchi e da trasformare il movimento del dito che scorre sullo schermo in rotazione del personaggio.

## Pistola

La pistola è l'arma principale del personaggio Detective, gli permette di sparare e uccidere sul colpo chiunque ed è ottenibile dai player appena presi abbastanza **Gold.**
Nel client l'Interfaccia riceve l'input direttamente dal Player, richiede il permesso di sparare al server e se lo ottiene, sul server vengono attivate le dovute procedure come la morte di un

personaggio, e alla fine il Client riceve il permesso di attivare gli effetti speciali dell'arma come particelle.




## Pistola Server

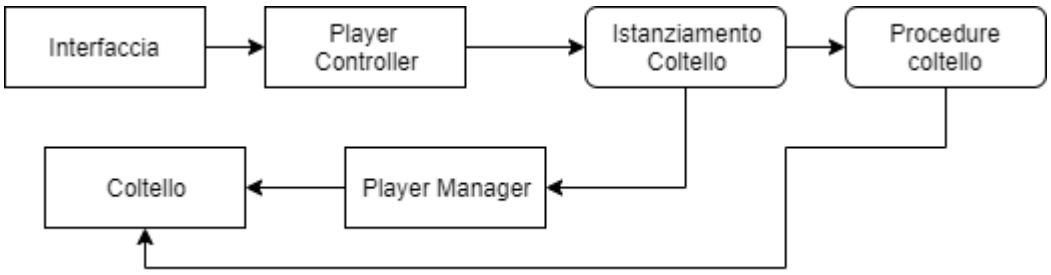
La pistola sul server analizza le condizioni di se stessa e se sono idonee procede allo sparo,e quindi invia il permesso al client di attivare gli effetti, lo sparo è un raggio che parte dalla telecamera del player, in una direzione dettata da essa e se questo raggio interseca un oggetto di gioco allora da inizio ad una procedura (come la morte).

## Pistola

Pistola contiene solo un metodo che gli permette di attivare gli effetti.

## Coltello

Il coltello è l'arma peculiare del personaggio Murderer,esso non è un raggio istantaneo come quello della pistola ma un oggetto fisico che necessita di instaziazione sia su server che su client oltre che metodi per essere raccolto da terra (l'insieme di questi metodi viene riassunto in Procedure Coltello).




## Coltello

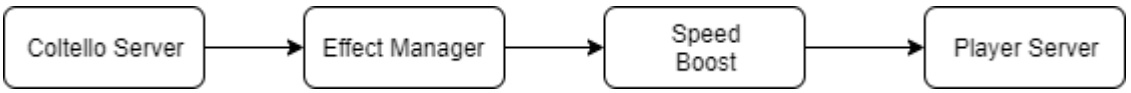
Sia sul server che sui client consiste nell'insieme di 3 oggetti:

- Coltello estetico: tenuto in mano
- Coltello lanciato: esiste solo finché non collide con un oggetto
- Coltello spawner: serve per raccogliere il coltello

Può esistere solo 1 di questi oggetti alla volta sia su server che sui client poiché l'esistenza di uno implica l'assenza degli altri.

## Adrenalina

L’adrenalina è ciò che fa muovere a velocità doppia e temporaneamente il Murderer dopo che egli uccide un altro player.
Un coltello contiene al suo interno l’ID del suo lanciatore, quando il coltello uccide un player sul server duplica la velocità del lanciatore assegnandogli un oggetto SpeedBoost con autodistruzione impostata a 3 secondi dopo la quale la velocità verrà dimezzata.




### Effect Manager

Contiene la lista di funzioni static per l'assegnazione effetti ai players.

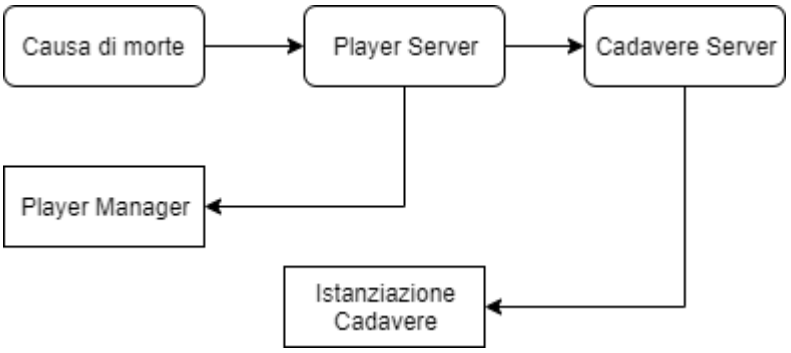
### Speed Boost

Un oggetto che quando associato ad un player ne duplica la velocità e la dimezza quando distrutto.

## Morte

La morte è un effetto chiamato sul Player da Pistola e Coltello.
La morte consiste in 2 fasi parallele:

- Trasformazione Player in Ghost:un player diventa invisibile e ottiene la capacità di volare e passare attraverso i muri.
- Istanziazione del Cadavere: un cadavere viene disteso a terra a simboleggiare il punto di morte di un player.




### Cadavere Server

Contiene un ID che lo identifica e una volta creato invia un messaggio ai Client per poterlo istanziare anche nelle loro schermate.

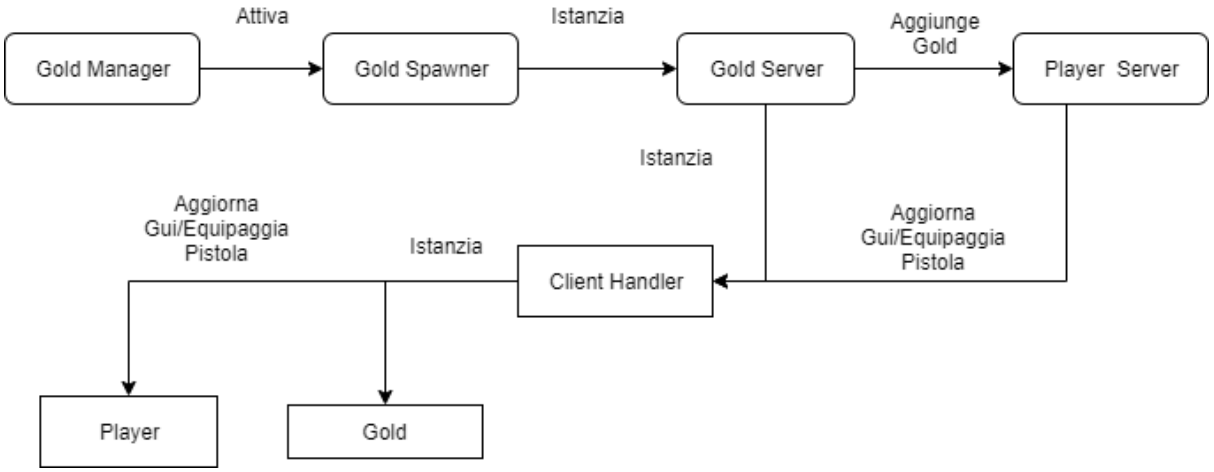
### Cadavere

E’ solo un oggetto grafico che viene posizionato in base alla posizione del Cadavere Server

## Gold

Il sistema che gestisce il Gold è formato da 3 parti:

- Gold manager : è uno script che decide dove il Gold verrà istanziato e con quale frequenza,esso viene attivato ad inizio Game
- Gold Spawner serve ad indicare la posizione nella quale un Gold verrà istanziato., sono sparsi per la mappa.
- Gold: è l'unico dei 3 oggetti che viene istanziato anche sul client consiste in un oggetto con sensore di avvicinamento che quando viene attivato aggiunge 1 gold al inventario di un player,quando un player non Murderer arriva a possedere 3 gold una pistola verrà equipaggiata ad esso.




## Gold Manager

E' uno script presente solo sul server che quando attivato seleziona dei gold spawner e istanzia del Gold alle loro posizioni.

## Gold spawner

Sono identificati con degli ID e contengono un metodo per istanziare il Gold.

## Gold Server

E' un oggetto del server con sensore di prossimità che attivato da inizio all'autodistruzione ed invia un messaggio al Player Server per l aumento del Gold e al Client per eliminare la parte grafica.