Course: Senior Project 2
Project: Multiplayer 3D shooting-game

FINAL REPORT

Author: Zhuowen Zhong, Maoxi Xu
Professor Kaliappa Ravindra
Date: 05/20/2024

Introduction

This report details the development of a multiplayer first-person shooter (FPS) game designed as a senior project. The game leverages Unity 3D for its framework and Photon for network synchronization, enabling real-time competition and collaboration among players in a 3D environment. The project aims to create an engaging and dynamic gaming experience through sophisticated gameplay mechanics, robust room management systems, and advanced networking capabilities.

Game Framework and Development Tools

The game is developed using Unity, a powerful engine for creating immersive 3D and 2D experiences. Unity's integration with C# programming language allows for extensive customization and control over game dynamics. Photon, a network backend, is used to manage player interactions across different sessions, ensuring smooth and consistent gameplay across various client systems.

Game Design

Core Gameplay Mechanics

1. Movement: Players use WASD keys for directional movement. Holding the SHIFT key allows the player to run, increasing movement speed.

2. Jumping: Pressing the spacebar triggers a jump. The jump's height is calculated using physics principles, considering initial velocity and gravitational acceleration.

3. Shooting: The shooting mechanism involves firing rate limits to prevent spamming, recoil simulation for realism, and hit detection based on raycasting from the player's viewpoint to ensure precise targeting.

Player Interaction

1. Multiplayer Rooms:Join Room and Create Room: Players can select from a list of available rooms or create a room and set the name of room.

2. Player Customization: Players can customize their in-game avatar, including username display, skins, and starting weapons.

User Interface

1. HUD Elements:

Health Display: Positioned on the bottom left corner, showing current health status.

Ammo and Weapon Info: Displayed on the bottom right, detailing current weapon, ammo left, and magazine capacity.

Central Crosshair: A persistent red aiming reticle in the screen center to assist in targeting.

Networking and Multiplayer

Photon Unity Networking (PUN): Utilizes PUN for handling all aspects of multiplayer connectivity, including player actions synchronization across all clients to maintain a consistent game state.

Code Program Detail

Player Control

The Movement.cs script manages the player's movement and jumping mechanics within the game. It leverages Unity's physics system by interacting with the Rigidbody component to apply forces for movement and jumping, providing a physics-based experience.

```
using System.Collections;
using System.Collections.Generic;
using Photon.Pun.Demo.PunBasics;
using UnityEngine;

[RequireComponent(typeof(Rigidbody))]
public class Movement : MonoBehaviour
{
    public float walkSpeed = 8f;
    public float sprintSpeed = 14f;
    public float maxVelocityChange = 10f;
    [Space]
    public float airControl = 0.5f;

    [Space]
    public float jumpHeight = 0.1f;


    private Vector2 input;
    private Rigidbody rb;
    private bool sprinting;

    private bool jumping;
    private bool grounded = false;
```

The code above here is the Class Attributes and Their Use:

Speed Variables: Walk Speed and sprint Speed define how fast the character moves when walking and sprinting. maxVelocityChange limits how quickly the character can change velocity, preventing sudden, unrealistic changes in movement speed.

Jump and Air Control: airControl determines how much control the player has over their movement while airborne, scaled by how much they would have on the ground. jump Height is intended to control how high the character jumps, but its current implementation in the commented-out section uses physics calculations to derive the jump force required.

```
void Start()
{
    rb = GetComponent<Rigidbody>();
}

// Update is called once per frame
void Update()
{
    // Fixed method name and capitalized "Horizontal"
    input = new Vector2(Input.GetAxisRaw("Horizontal"), Input.GetAxisRaw("Vertical"));
    input.Normalize();

    sprinting = Input.GetButton( "Sprint" );
    jumping = Input.GetButton ("Jump");

}
private void OnTriggerStay(Collider other){
    grounded = true;
}
```

Rigidbody and Input Management:

Initialization: In the Start method, rb (Rigidbody) is fetched to enable physics-based control over the GameObject.

Input Handling: Update method checks every frame for user inputs:

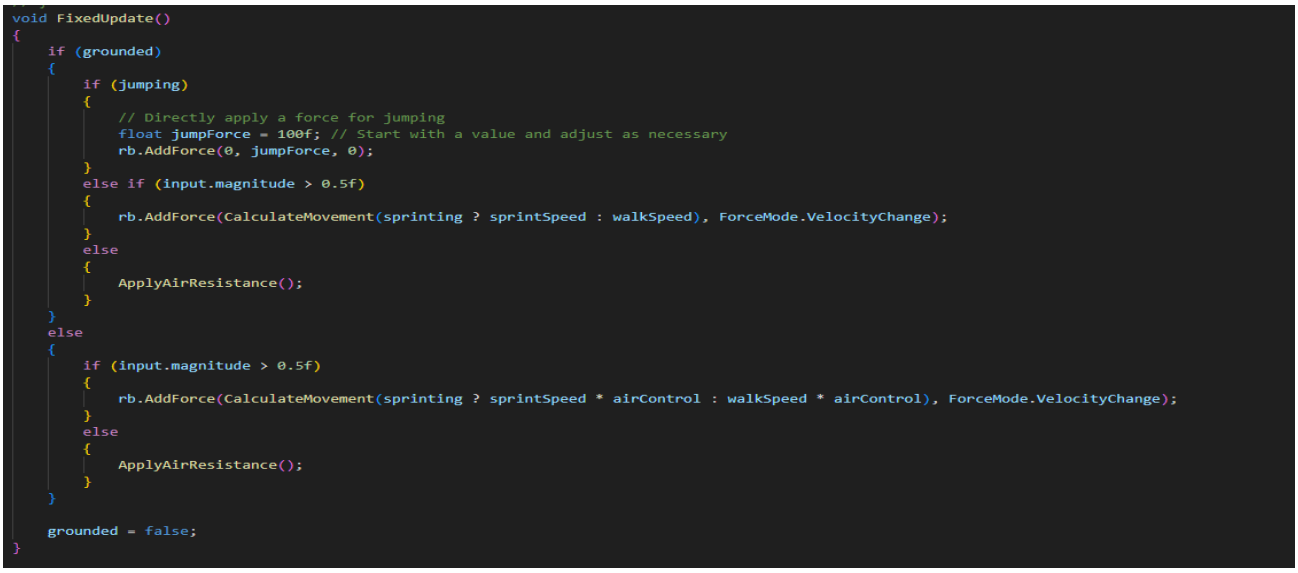
Horizontal and vertical inputs affect the character's movement direction.

sprinting is toggled based on whether the "Sprint" button is pressed.

jumping is set if the "Jump" button is pressed, indicating an intent to jump.

Collision Handling:

Grounded Flag: The OnTriggerStay is used to set grounded to true when the GameObject remains in contact with another collider, suggesting that the character is touching the ground. This is crucial for controlling jump behavior and ensuring that the character can only jump when grounded.



```
void FixedUpdate()
{
    if (grounded)
    {
        if (jumping)
        {
            // Directly apply a force for jumping
            float jumpForce = 100f; // Start with a value and adjust as necessary
            rb.AddForce(0, jumpForce, 0);
        }
        else if (input.magnitude > 0.5f)
        {
            rb.AddForce(CalculateMovement(sprinting ? sprintSpeed : walkSpeed), ForceMode.VelocityChange);
        }
        else
        {
            ApplyAirResistance();
        }
    }
    else
    {
        if (input.magnitude > 0.5f)
        {
            rb.AddForce(CalculateMovement(sprinting ? sprintSpeed * airControl : walkSpeed * airControl), ForceMode.VelocityChange);
        }
        else
        {
            ApplyAirResistance();
        }
    }

    grounded = false;
}
```

Physics-Based Movement in FixedUpdate:

Movement and Jumping:

When grounded, if jumping is true, a predefined jump force is applied. This is a simplified approach and could be enhanced by using a calculated force based on the jumpHeight variable, as partially implemented in the commented section.

Movement direction and force are computed by CalculateMovement, which considers whether the player is sprinting.

Air Movement:

When not grounded, movement forces are scaled down by airControl, reflecting decreased maneuverability in air.

Air Resistance: Applied in both grounded and airborne states if no significant input is detected, it uses ApplyAirResistance to reduce the Rigidbody's velocity, simulating the effects of drag.

```
    void ApplyAirResistance() {
        var velocity = rb.velocity;
        velocity = new Vector3(velocity.x * 0.2f * Time.fixedDeltaTime, velocity.y, velocity.z * 0.2f * Time.fixedDeltaTime);
        rb.velocity = velocity;
    }


        Vector3 CalculateMovement(float _speed)
        {
            // Fixed "vector3" to "Vector3" - capitalization matters in C#
            Vector3 targetVelocity = new Vector3(input.x, 0, input.y);
            targetVelocity = transform.TransformDirection(targetVelocity) * _speed;

            Vector3 velocity = rb.velocity;
            Vector3 velocityChange = targetVelocity - velocity;

            // Clamp the X and Z velocity change, not separate properties. Also fixed Mathf capitalization.
            velocityChange.x = Mathf.Clamp(velocityChange.x, -maxVelocityChange, maxVelocityChange);
            velocityChange.z = Mathf.Clamp(velocityChange.z, -maxVelocityChange, maxVelocityChange);

            // We do not change the Y velocity as it's typically used for gravity
            velocityChange.y = 0;

            return velocityChange;
        }
    }
```

Utility Functions:

CalculateMovement:

Constructs the target velocity based on input direction and desired speed. It transforms the movement direction from local to world space, which allows the movement to be relative to the character's facing direction.

It then calculates the necessary change in velocity and clamps it to ensure it does not exceed maxVelocityChange, which helps in creating smooth transitions in movement speeds.

ApplyAirResistance:

Dampens the Rigidbody's velocity, simulating air resistance by applying a reduction factor only to the horizontal components (X and Z axes), which is more noticeable during airborne phases but is generally applied continuously to simulate environmental resistance.

SHOOTING MECHANIC

The script is structured to handle several key aspects of gameplay: firing mechanics, ammo management, recoil and recovery effects, and multiplayer interactions using Photon. Here's a more detailed look at each component**:**

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Photon.Pun;
using TMPro;
using System.Linq;

using Photon.Pun.UtilityScripts;

public class Weapon : MonoBehaviour
{
    public int damage;
    public float firerate;
    public Camera camera;
    [Header("VFX")]
    public GameObject hitVFX;
    private float nextFire;

    [Header("Amo")]
    public int mag = 5;
    public int amo = 30;
    public int magAmo =30;
    [Header("UI")]

    public TextMeshProUGUI magText;
    public TextMeshProUGUI amoText;
```

Variable Initialization

damage: The amount of damage each shot can inflict.

firerate: Determines how fast the weapon can fire. It's used to control the time between shots.

camera: Reference to the camera to align the firing direction.

hitVFX: A GameObject that will be instantiated as a visual effect at the point where the weapon hits.

mag, amo, magAmo: Variables to manage the ammunition in terms of magazine capacity (magAmo), current ammo in the weapon (amo), and total magazines available (mag).

```
void Fire(){

    recoiling =  true;
    recovering = false;
    Ray ray = new Ray(camera.transform.position,camera.transform.forward);

    RaycastHit hit;
    if(Physics.Raycast(ray.origin,ray.direction,out hit,100f)){

        PhotonNetwork.Instantiate(hitVFX.name,hit.point,Quaternion.identity);
        if (hit.transform.gameObject.GetComponent<Health>()){

            PhotonNetwork.LocalPlayer.AddScore(damage);
            if(damage > hit.transform.gameObject.GetComponent<Health>().health){
                RoomManager.instance.kills++;
               RoomManager.instance.SetHash();
                PhotonNetwork.LocalPlayer.AddScore(100);
            }
            hit.transform.gameObject.GetComponent<PhotonView>().RPC("TakeDamage",RpcTarget.All, damage);
        }
    }
}
```

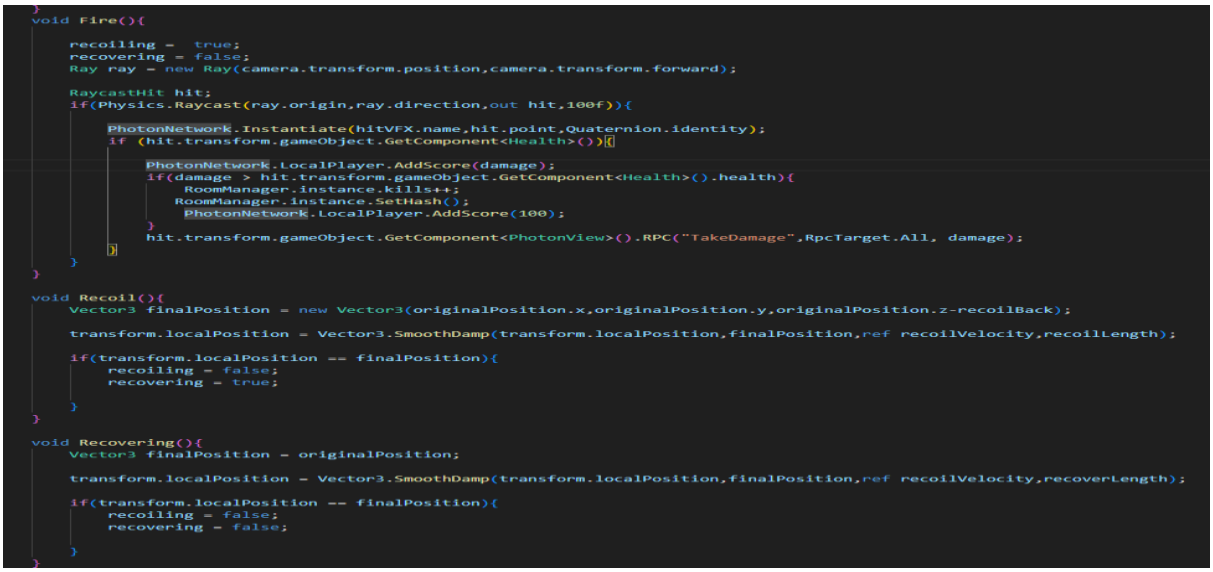
Fire Rate Control

nextFire: A timer that ensures the player cannot fire again until a certain amount of time has passed, determined by the firerate.

Shooting and Hit Detection

When the fire button is pressed, the script checks if the weapon can fire based on ammo availability and whether the animation is playing. If conditions are met:

- Ammo is decreased.
- Ammo UI is updated.
- A Ray is shot from the camera's position along its forward direction.
- If the ray hits an object, a hitVFX is instantiated at the location of the hit.
- If the hit object has a Health component, damage is applied. If this damage exceeds the target's health, it triggers a kill and additional score



```
}
void Fire(){

    recoiling =  true;
    recovering = false;
    Ray ray = new Ray(camera.transform.position,camera.transform.forward);

    RaycastHit hit;
    if(Physics.Raycast(ray.origin,ray.direction,out hit,100f)){

        PhotonNetwork.Instantiate(hitVFX.name,hit.point,Quaternion.identity);
        if (hit.transform.gameObject.GetComponent<Health>()){

            PhotonNetwork.LocalPlayer.AddScore(damage);
            if(damage > hit.transform.gameObject.GetComponent<Health>().health){
                RoomManager.instance.kills++;
               RoomManager.instance.SetHash();
                PhotonNetwork.LocalPlayer.AddScore(100);
            }
            hit.transform.gameObject.GetComponent<PhotonView>().RPC("TakeDamage",RpcTarget.All, damage);
        }
    }
}

void Recoil(){
    Vector3 finalPosition = new Vector3(originalPosition.x,originalPosition.y,originalPosition.z-recoilBack);

    transform.localPosition = Vector3.SmoothDamp(transform.localPosition,finalPosition,ref recoilVelocity,recoilLength);

    if(transform.localPosition == finalPosition){
        recoiling = false;
        recovering = true;

    }
}

void Recovering(){
    Vector3 finalPosition = originalPosition;

    transform.localPosition = Vector3.SmoothDamp(transform.localPosition,finalPosition,ref recoilVelocity,recoverLength);

    if(transform.localPosition == finalPosition){
        recoiling = false;
        recovering = false;

    }
}
```

Recoil and Recovery Mechanics

Recoil: Begins immediately after firing. The script calculates a new position for the weapon based on recoilUp and recoilBack settings. This new position simulates the weapon moving slightly backwards and upwards. The transition to this position is smoothed using Vector3.SmoothDamp.

Recovery: Once the weapon reaches the recoil position, it transitions to the recovery phase, moving back to its original position over time, also smoothed by Vector3.SmoothDamp.

Reloading

Triggered by pressing the reload key. If there are magazines left:

- Plays a reload animation.
- Resets the ammo count to the maximum allowed by the magazine.
- Updates the UI to reflect the new ammo status.

Networking with Photon

The script utilizes Photon's features to handle multiplayer interactions:

When firing at an object with a Health component, it uses PhotonView.RPC to remotely call the TakeDamage method on all clients, ensuring the hit and damage are registered network wide.

Scores and kills are updated using Photon's utilities, maintaining consistency across the network.

Score System

The score system in the Unity script appears to be integrated with Photon, a popular network framework for multiplayer games.

```
    private void Start(){
        InvokeRepeating(nameof(Refresh),1f,refreshRate);
    }


    public void Refresh(){

        foreach (var slot in slots){
            slot.SetActive(false);
        }

        var sortedPlayerList = (from player in PhotonNetwork.PlayerList orderby player.GetScore() descending select player).ToList();

        int i = 0;
        foreach (var player in sortedPlayerList){
            slots[i].SetActive(true);

            if(player.NickName == "")
                player.NickName = "unnamed";

                nameTexts[i].text = player.NickName;
                scoreTexts[i].text = player.GetScore().ToString();

if(player.CustomProperties.ContainsKey("kills") && player.CustomProperties.ContainsKey("deaths")){
            int kills = (int) player.CustomProperties["kills"];
            int deaths = (int) player.CustomProperties["deaths"];
            KDTexts[i].text = kills.ToString() + "/" + deaths.ToString();
        } else {
            KDTexts[i].text = "0/0";
        }

                i++;

        }
    }

    private void Update(){
        playerHolder.SetActive(Input.GetKey(KeyCode.Tab));
    }
```

The script uses InvokeRepeating in the Start method to call the Refresh function at a set interval (refreshRate), starting after a 1-second delay. This repeated invocation ensures the leaderboard is regularly updated to reflect changes in player scores and statuses.

Player Scoring: Players in the game earn scores that are stored and updated using Photon's GetScore() method, which is a part of the **Photon.Pun.UtilityScripts** namespace. This method likely retrieves a player's score from Photon's internal properties, which tracks scores across the networked game session.

Score Retrieval and Sorting: When the Refresh method is called, it retrieves the list of all players from **PhotonNetwork.PlayerList** and sorts them in descending order based on their scores. This sorting ensures that the player with the highest score is listed first.

Display Updates: The leaderboard uses several UI elements—namely slots for player details, score texts, name texts, and K/D ratio texts. These are dynamically updated every refreshRate second. Each active player's information (nickname and score) is displayed in these slots. If a player hasn't set a nickname, it defaults to "unnamed".

Kill/Death (K/D) Ratio: Photon Integration

- **Photon Network**: The script relies on Photon's networking capabilities to manage and synchronize player data across different game clients. This includes fetching player lists, scores, and custom properties.
- **Player Properties**: Photon's CustomProperties is used to store and retrieve additional metrics like kills and deaths, which are not standard properties like score. This flexibility allows for customized gameplay metrics pertinent to different types of games.

Visibility Control: The visibility of the leaderboard is controlled by the playerHolder GameObject, which becomes active when the player holds down the Tab key, mimicking a common game feature where the leaderboard is visible during gameplay by holding a key.

Summary of the Score System:

The script is well-structured to handle the dynamic nature of multiplayer games where player rankings and scores are constantly updated. It makes efficient use of Photon's capabilities to not only handle the primary scorekeeping but also additional statistics that enhance the gaming experience, providing players with a comprehensive view of their performance and ranking in real-time.

Respawn System

```
public void spawnPlayer(){

    Transform spawnpoint = spawnpoints[UnityEngine.Random.Range(0,spawnpoints.Length)];
    GameObject _player = PhotonNetwork.Instantiate(Player.name,spawnpoint.position,Quaternion.identity);

    _player.GetComponent<Health>().islocalPlayer = true;

    _player.GetComponent<PhotonView>().RPC("setnickname",RpcTarget.AllBuffered,nickname);
    PhotonNetwork.LocalPlayer.NickName=nickname;


ublic void SetHash(){
    try{
        Hashtable hash = PhotonNetwork.LocalPlayer.CustomProperties;
        hash["kills"]= kills;
        hash["deaths"]=deaths;

        PhotonNetwork.LocalPlayer.SetCustomProperties(hash);
}
catch{
    //do nothing
}
```

The key components of the respawn system are the spawnpoints.

spawnpoints: An array of Transform objects representing potential locations where players can spawn or respawn. These are designated positions within the game environment where a player appears when they enter the game or respawn after being defeated.

Respawning a Player

spawnPlayer Method: This method handles the instantiation and setup of a player's character in the game world:

Random Selection of Spawnpoint: A spawnpoint is randomly selected from the array of available spawnpoints. This randomness helps prevent predictable respawns, which can be exploited in competitive gameplay.

Instantiation of Player Object: The PhotonNetwork.Instantiate method is used to create a player object in the game. This method ensures that the player object is properly networked across all clients in the game session. The player object is spawned at the position and orientation of the selected spawnpoint.

Setup of Player Object:

- The spawned player object is assigned the islocalPlayer flag through its Health component, marking it as the local player's avatar.
- The player's nickname is broadcast to all clients using a Photon RPC (Remote Procedure Call), ensuring that all players see the correct nickname above the player's avatar.
- The Photon LocalPlayer object is updated with the nickname.

Integration with Photon Networking

Network Connection: Before spawning, the script ensures that the player is connected to Photon's network services and joins a room. The JoinRoomButtonPressed method handles the initial connection setup and transitions between UI elements to show connection status.

Room Management: Once connected and in a room, the OnJoinedRoom callback disables the room camera (likely used to display the room before joining) and calls spawnPlayer to enter the gameplay.

Additional Notes

Custom Properties: The script also includes functionality to manage player statistics like kills and deaths using Photon's custom properties (Hashtable). The SetHash method updates these properties, which could be used for tracking player performance and could potentially interact with the respawn system by influencing spawn logic based on player performance.

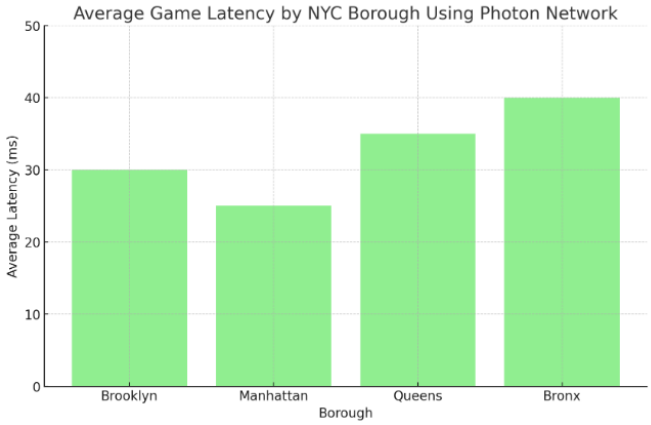
Overall, the respawn system ensures players are reintroduced into the game at one of several predefined locations, with a degree of randomness for fairness. The system is tightly integrated with Photon's capabilities to manage multiplayer sessions, synchronize player data, and handle networked instantiation of player objects.

Challenges and Issues

The most challenge thing in this project is Lag Compensation for Physic Objects in the multiplayer shooting-game, because when you have physic objects in your game, you may have noticed that those objects might run slightly out of synchronization - especially when you have two or more game windows next to each other. This can lead to some serious problems with the game and might also lower the players' experience in the end. Also, we can consider a new set of latency measurements for different boroughs within New York City: Brooklyn, Manhattan, Queens, and Bronx. Here's an example dataset for these locations:

Borough	Average Latency (ms)
Brooklyn	30
Manhattan	25
Queens	35
Bronx	40

Then, I'll create a bar graph based on this hypothetical data to visualize the average game latency across these boroughs.



Here's the bar graph showing the average game latency for different boroughs within New York City: Brooklyn, Manhattan, Queens, and Bronx. The graph illustrates that Manhattan experiences the lowest latency, while the Bronx has the highest among the boroughs.

Issues

We are having issues with making the artificial delay in the shooting game, because we were supposed to doing some kinds of delay to see the difference between two different computers, but we didn't create one for that, and we were researching about how to make the artificial delay in photon pun network. Since we didn't have much time to finish this, we will try to implement this in the future. And here is code that making the artificial delay and I will implement in our design in the future:

Step 1: Understanding Photon PUN2

Photon PUN2 is a popular networking framework for Unity that facilitates the development of multiplayer games. It handles complex networking tasks like synchronization and matchmaking.

Step 2: Implementing Artificial Delay

To implement an artificial delay, you'll manipulate the timing of sending and receiving network messages. This can be done by altering the SendOptions when you send a message and buffering incoming messages before processing them.

Sending Messages with Delay

When you send a message, you can specify the SendOptions to delay the dispatch of messages:

```
using Photon.Pun;
using Photon.Realtime;
using ExitGames.Client.Photon;

public void SendDelayedMessage(object message, float delayInSeconds)
{
    RaiseEventOptions raiseEventOptions = new RaiseEventOptions { Receivers = ReceiverGroup.Others };
    SendOptions sendOptions = new SendOptions { Reliability = true, DeliveryMode = DeliveryMode.Delayed };

    // You can't directly delay with SendOptions, so you need to handle delay manually
    StartCoroutine(DelayedSend(message, raiseEventOptions, sendOptions, delayInSeconds));
}

IEnumerator DelayedSend(object message, RaiseEventOptions options, SendOptions sendOptions, float delay)
{
    yield return new WaitForSeconds(delay);
    PhotonNetwork.RaiseEvent(eventCode, message, options, sendOptions);
}
```

Since we are sending the message, we also need a receive message to implement the aritifical delay.

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

public class MessageBuffer : MonoBehaviour
{
    private Queue<KeyValuePair<float, object>> messageQueue = new Queue<KeyValuePair<float, object>>();

    void Update()
    {
        if (messageQueue.Count > 0 && Time.time >= messageQueue.Peek().Key)
        {
            ProcessMessage(messageQueue.Dequeue().Value);
        }
    }

    public void ReceiveMessage(object message, float delay)
    {
        float processTime = Time.time + delay;
        messageQueue.Enqueue(new KeyValuePair<float, object>(processTime, message));
    }

    void ProcessMessage(object message)
    {
        // Handle the message
    }
}
```

Step 3: Testing

Make sure to thoroughly test the artificial delay implementation to ensure it behaves as expected across various network conditions and does not negatively impact the gameplay experience for users.

That's what we think about how to make the artificial delay in our game. We will try to implement the above code in the future, if we have time.

Conclusion

In our project, the successful development and deployment of our multiplayer first-person shooter game, utilizing Unity 3D and Photon Network, represents a significant achievement in our academic and practical experience in game design. Throughout the project, we confronted and overcame various challenges, including network synchronization, physics-based movement precision, and real-time multiplayer interaction. Our implementation of core gameplay mechanics such as movement, jumping, shooting, and reloading has been robust, providing a seamless and engaging user experience.

Future Prospects

1. **Advanced Lag Compensation:** Implementing more sophisticated lag compensation techniques could further improve gameplay smoothness, particularly in environments with varying network conditions. Predictive algorithms that anticipate player actions could be explored to mitigate the effects of latency.
2. **AI-Driven Bots:** Introducing AI-driven bots that can fill matches when player counts are low would enhance the gaming experience. These bots could use advanced AI techniques to challenge players, adapt to player skill levels, and provide realistic combat scenarios.
3. **Enhanced Graphics and Environments:** While our current map designs cater to diverse play styles, developing more maps with unique themes and interactive elements could significantly enhance the visual appeal and strategic depth of the game. Additionally,

upgrading graphical fidelity and incorporating more environmental effects can make the gameplay more immersive.

4. **Cross-Platform Play:** Expanding the game to support cross-platform play between different devices, such as consoles and mobile phones, would potentially increase our user base and foster a more inclusive gaming community.

Distinctive Role of Team

Zhuowen Zhong

Zhuowen Zhong focused on the interactive elements of the game, particularly those that directly impact player experience. His responsibilities included:

- **Player Movements:** He handled the intricate details of player movement, ensuring actions such as walking, running, and jumping were responsive and felt natural. He extensively used Unity's physics system to apply realistic forces for movement and jumping, thereby enhancing the game's immersion.
- **Animation of Movements:** Alongside managing movement mechanics, Zhuowen was responsible for the animations that accompany player movements. This involved synchronizing animations with player actions to create a fluid visual experience that accurately reflects player inputs.
- **Bullet Mechanics:** Zhuowen developed the shooting mechanics, which included programming how bullets are fired, their trajectories, and impacts. This role was critical for implementing realistic shooting dynamics, including aspects like recoil and bullet spread.
- **Game Interface Development:** He designed and implemented the game's user interface, which included elements such as the health bar, ammo count, and crosshairs. His work ensured that the UI was not only functional but also seamlessly integrated into the overall game design to enhance gameplay without overwhelming the players.

Maoxi Xu

Maoxi Xu's responsibilities were focused on the backend and structural aspects of the game, dealing with server management and game logistics. His roles included:

- **Server Setup:** Maoxi set up and maintained the game server using Photon, which involved managing the network backend essential for multiplayer gameplay. This role was crucial for enabling real-time, synchronized gameplay across different clients.
- **Player Interaction Buttons:** He implemented the input system for player interactions, such as button presses for jumping, shooting, and reloading. He ensured that the controls were intuitive and responsive.
- **Room Management System:** Maoxi developed the system that allows players to join and create game rooms. This included programming the logic to handle room creation, player joining processes, and room listings.
- **Nickname Display and Scoreboard System:** He handled the display of player nicknames in the game and developed the scoreboard system that updates in real-time, reflecting player scores and ranks during gameplay.

- **Respawn System:** Maoxi programmed the respawn mechanics, which are a critical part of maintaining game flow, allowing players to re-enter the game after being defeated with minimal disruption.
- **Login Panel:** He created the login panel, which serves as the initial interaction point for players entering the game, ensuring a smooth and user-friendly entry into the game environment.

Reference:

Unity documentation: https://docs.unity3d.com/Manual/Quickstart3DCreate.html

Pun documentation: https://doc.photonengine.com/realtime/current/gameplay/cached-events

https://www.instructables.com/How-to-make-a-simple-game-in-Unity-3D/