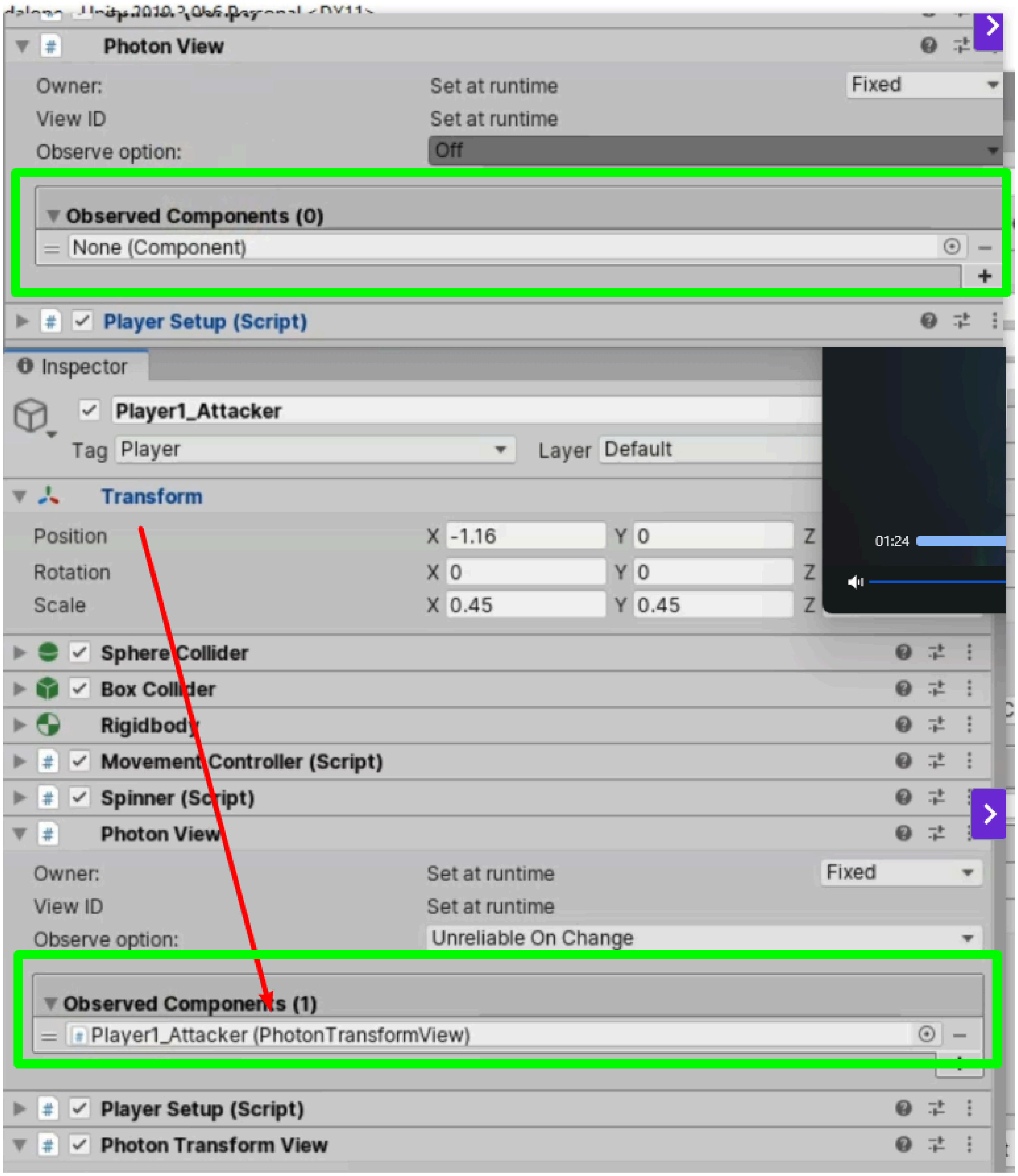
how recieve next is taking count of which recieve next recieve which one sent data

In **Photon PUN**, the `ReceiveNext()` method retrieves values from the stream in the exact order they were sent using `SendNext()`. The order of `SendNext()` calls in the **writing section** (when `stream.IsWriting` is `true`) determines the order in which values must be read in the **reading section** (when `stream.IsWriting` is `false`).

How ReceiveNext() Matches SendNext()

- `SendNext()` pushes values onto a queue (like a list).
- `ReceiveNext()` reads them in the same sequence.



Photon View

Owner: Set at runtime Fixed

View ID Set at runtime

Observe option: Off

Observed Components (0)

None (Component)

Player Setup (Script)

Inspector

Player1_Attacker

Tag Player Layer Default

Transform

Position	X	-1.16	Y	0	Z
Rotation	X	0	Y	0	Z
Scale	X	0.45	Y	0.45	Z

Sphere Collider

Box Collider

Rigidbody

Movement Controller (Script)

Spinner (Script)

Photon View

Owner: Set at runtime Fixed

View ID Set at runtime

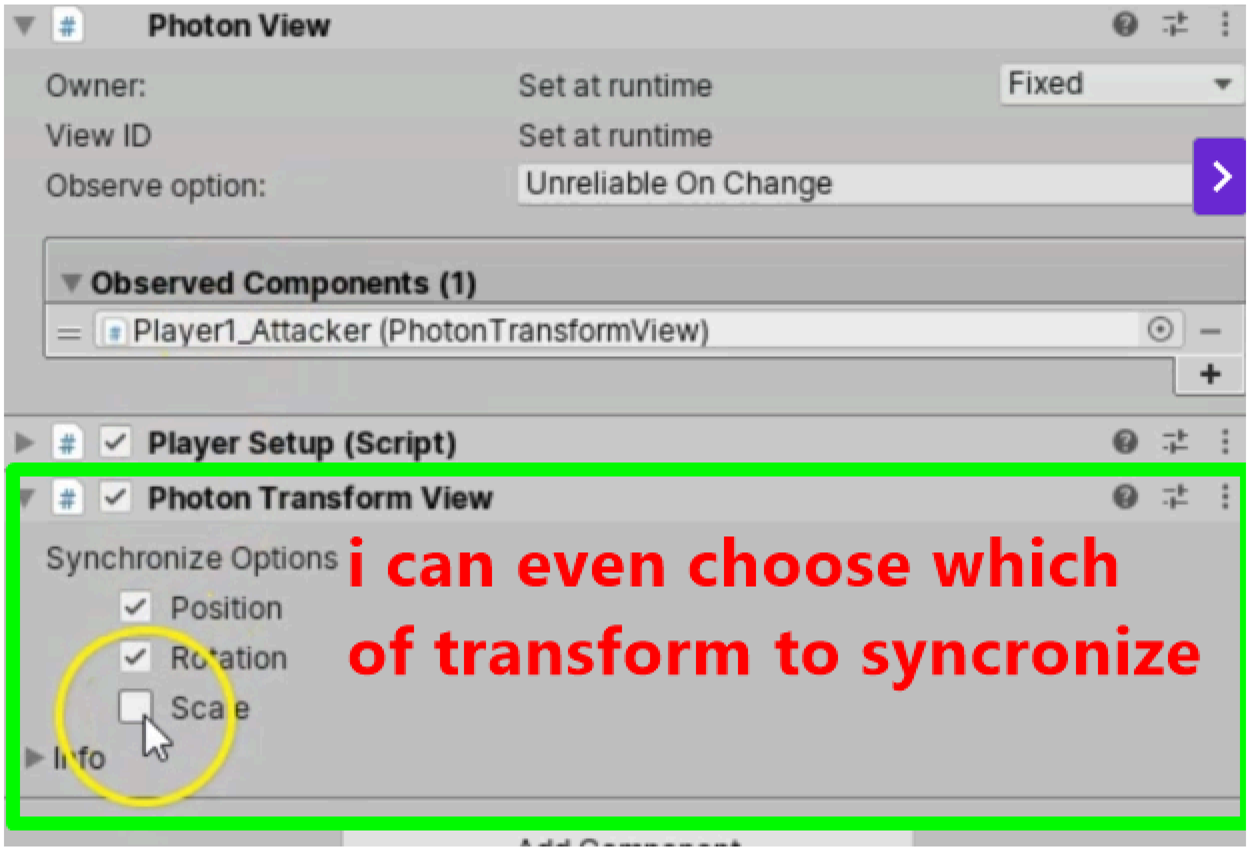
Observe option: Unreliable On Change

Observed Components (1)

Player1_Attacker (PhotonTransformView)

Player Setup (Script)

Photon Transform View



Photon View

Owner: Set at runtime Fixed

View ID Set at runtime

Observe option: Unreliable On Change

Observed Components (1)

Player1_Attacker (PhotonTransformView)

Player Setup (Script)

Photon Transform View

Synchronize Options

✓ Position

✓ Rotation

Scale

Info

i can even choose which of transform to syncronize

PhotoView Component

In **Photon Unity Networking (PUN)**, `PhotonView` is a core component that enables networking functionality by synchronizing GameObjects across different clients. It ensures that the same object behaves consistently for all players in a multiplayer game.

Key Features of PhotonView

1. **Unique View ID**
   - Each `PhotonView` has a unique `ViewID`, which is used to identify and synchronize objects across the network.
   - The `ViewID` is assigned either manually or automatically.
2. **Ownership**
   - Each `PhotonView` has an **owner** (a player who instantiated or owns the object).
   - `PhotonView.IsMine` checks if the local player owns the object.
   - Ownership can be transferred using `TransferOwnership(int playerID)`.
3. **Synchronization**

    - Used to sync position, rotation, animations, and other properties.
    - Works alongside `PhotonTransformView`, `PhotonAnimatorView`, or custom scripts.
4. **Remote Procedure Calls (RPCs)**
    - Enables calling functions across the network.
    - Example: `photonView.RPC("SomeFunction", RpcTarget.All, parameters);`

Basic Usage Example

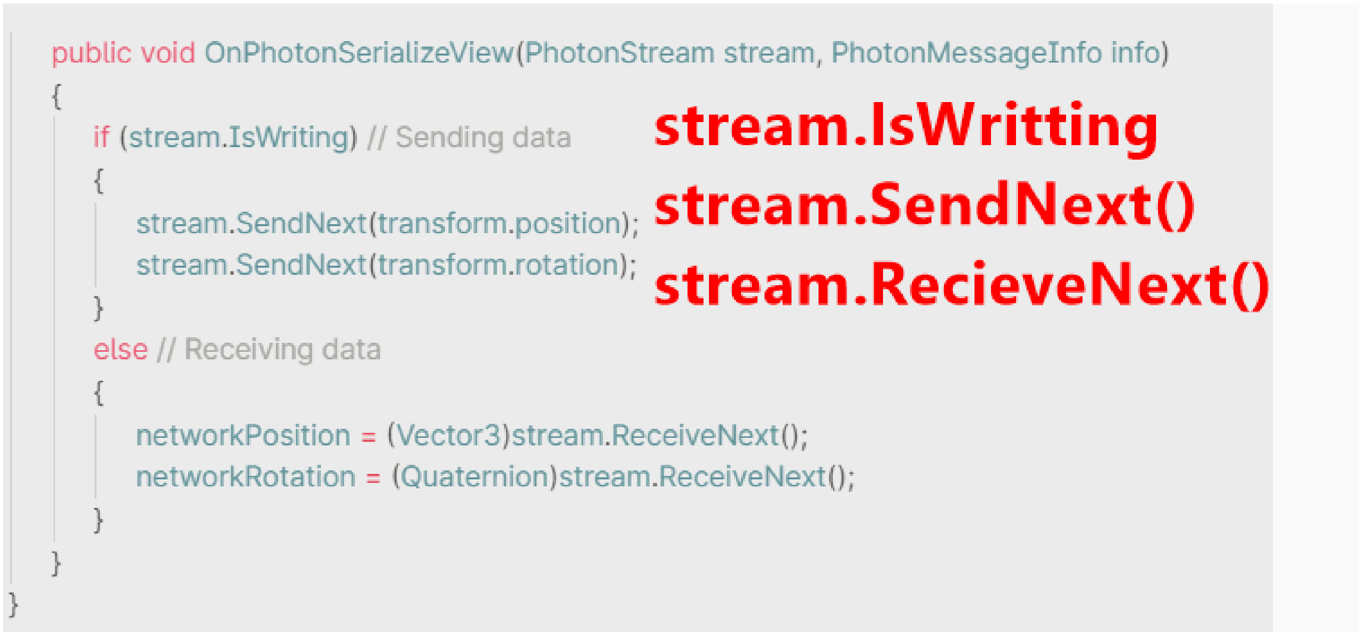
1. Checking Ownership**

```
using Photon.Pun;

public class PlayerController : MonoBehaviour
{
    private PhotonView photonView;

    void Start()
    {
        photonView = GetComponent<PhotonView>();

        if (photonView.IsMine)
        {
            // This object belongs to the local player
            Debug.Log("I own this object!");
        }
        else
        {
            // Another player owns this object
            Debug.Log("Another player owns this object.");
        }
    }
}
```

2. Synchronizing Position and Rotation** (GOAT)

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

```
public class PlayerMovement : MonoBehaviourPun, IPunObservable
{
    public float speed = 5f;
    private Vector3 networkPosition;
    private Quaternion networkRotation;

    void Update()
    {
        if (photonView.IsMine)
        {
            float moveX = Input.GetAxis("Horizontal") * speed * Time.deltaTime;
            float moveZ = Input.GetAxis("Vertical") * speed * Time.deltaTime;
            transform.Translate(new Vector3(moveX, 0, moveZ));
        }
        else
        {
            // Smoothly update position for remote players
            transform.position = Vector3.Lerp(transform.position, networkPosition,
Time.deltaTime * 10);
            transform.rotation = Quaternion.Lerp(transform.rotation, networkRotation,
Time.deltaTime * 10);
        }
    }

    public void OnPhotonSerializeView(PhotonStream stream, PhotonMessageInfo info)
    {
        if (stream.IsWriting) // Sending data
        {
            stream.SendNext(transform.position);
            stream.SendNext(transform.rotation);
        }
        else // Receiving data
        {
            networkPosition = (Vector3)stream.ReceiveNext();
            networkRotation = (Quaternion)stream.ReceiveNext();
        }
    }
}
```



```
public void OnPhotonSerializeView(PhotonStream stream, PhotonMessageInfo info)
{
    if (stream.IsWriting) // Sending data
    {
        stream.SendNext(transform.position);
        stream.SendNext(transform.rotation);
    }
    else // Receiving data
    {
        networkPosition = (Vector3)stream.ReceiveNext();
        networkRotation = (Quaternion)stream.ReceiveNext();
    }
}
}
```

3. Using RPCs**

```
using Photon.Pun;

public class PlayerActions : MonoBehaviourPun
{
   void Update()
   {
      if (Input.GetKeyDown(KeyCode.Space) && photonView.IsMine)
      {
         photonView.RPC("Jump", RpcTarget.All);
      }
   }

   [PunRPC]
   void Jump()
   {
      Debug.Log("Player jumped!");
      GetComponent<Rigidbody>().AddForce(Vector3.up * 5f, ForceMode.Impulse);
   }
}
```

Common Issues & Solutions

1. **PhotonView ViewID is `0`**
    - This happens when the object isn't properly instantiated over the network.
    - Use `PhotonNetwork.Instantiate("PrefabName", position, rotation)` instead of `Instantiate()`.
2. **RPC Not Executing**
    - Ensure the method has `[PunRPC]` attribute.
    - Check if `PhotonView` is on the same GameObject as the script.
3. **Synchronization Lag / Jitter**
    - Use interpolation (`Vector3.Lerp`) for smoother movement.
    - Optimize data sent through `OnPhotonSerializeView`.

IPunObservable in Photon Unity Networking (PUN)

`IPunObservable` is an interface used in **Photon Unity Networking (PUN)** for synchronizing data across the network. It is primarily used with `OnPhotonSerializeView`, allowing custom data (like position, health, ammo, etc.) to be sent and received between players.

How IPunObservable Works

1. **Sending Data (Writing)**
    - If the local player owns the `PhotonView`, they can send data to other players.
2. **Receiving Data (Reading)**
    - Other players receive this data and apply it to their copies of the object.

Basic Implementation

To use `IPunObservable`, implement it in your script and define `OnPhotonSerializeView`.

Using IPunObservable for Health Synchronization

Example of syncing player health across the network:

```
using Photon.Pun;

public class PlayerHealth : MonoBehaviourPun, IPunObservable
{
    public int health = 100;

    public void TakeDamage(int damage)
    {
        if (photonView.IsMine) // Only the owner should modify health
        {
            health -= damage;
            Debug.Log("Health: " + health);
        }
    }

    public void OnPhotonSerializeView(PhotonStream stream, PhotonMessageInfo info)
    {
        if (stream.IsWriting) // Sending health to others
        {
            stream.SendNext(health);
        }
        else // Receiving health from the owner
        {
            health = (int)stream.ReceiveNext();
        }
    }
}
```

🔹 **Key Points:**

- Only the player who owns the `PhotonView` updates their health.
- Other players receive the updated health value in real time.

Best Practices for IPunObservable

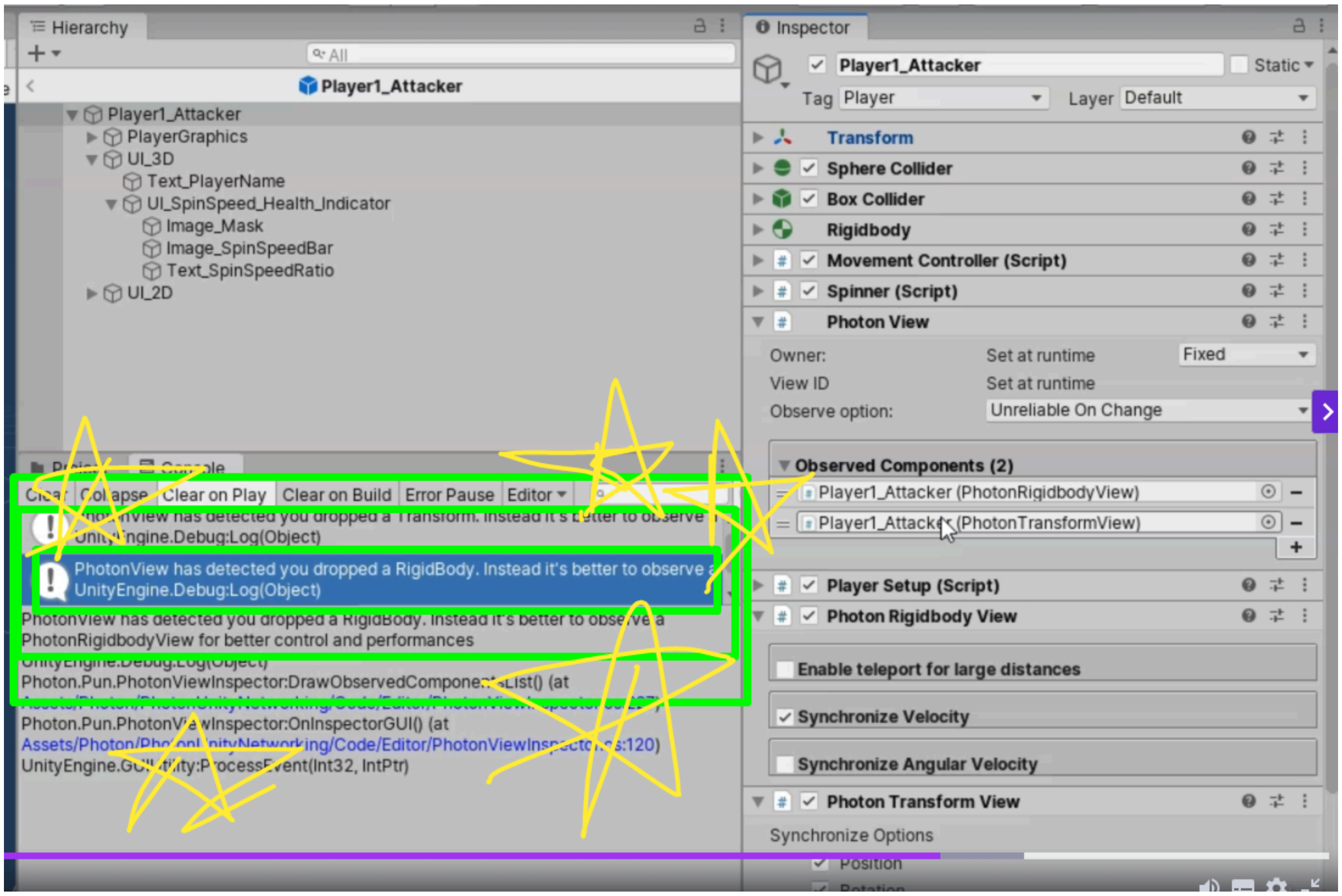
✅ Use Lerp/Slerp for smooth movement interpolation ✅ Only sync essential data to optimize network performance ✅ Ensure the object has a `PhotonView` component attached ✅ Avoid syncing physics-heavy objects frequently (use interpolation instead) ✅ Use `PhotonNetwork.Instantiate()` instead of `Instantiate()` for networked objects

Common Issues & Fixes

Issue	Solution
`OnPhotonSerializeView` not being called	Ensure the GameObject has a `PhotonView` component.
Laggy movement	Use interpolation (`Vector3.Lerp`) instead of directly setting the position.
View ID is `0` (Not syncing)	Ensure objects are instantiated with `PhotonNetwork.Instantiate()`.
Data is not updating	Make sure the data is properly sent/received using `stream.SendNext()` and `stream.ReceiveNext()`.

`OnPhotonSerializeView(PhotonStream stream, PhotonMessageInfo info)`

It's much better to add `rigidbody` component over `transform` transform component in monitoring



Player1_Attacker
- Player1_Attacker
  - PlayerGraphics
  - UI_3D
    - Text_PlayerName
    - UI_SpinSpeed_Health_Indicator
      - Image_Mask
      - Image_SpinSpeedBar
      - Text_SpinSpeedRatio
  - UI_2D

Player1_Attacker — Static
Tag: Player — Layer: Default

- Transform
- ✓ Sphere Collider
- ✓ Box Collider
- Rigidbody
- ✓ Movement Controller (Script)
- ✓ Spinner (Script)
- Photon View

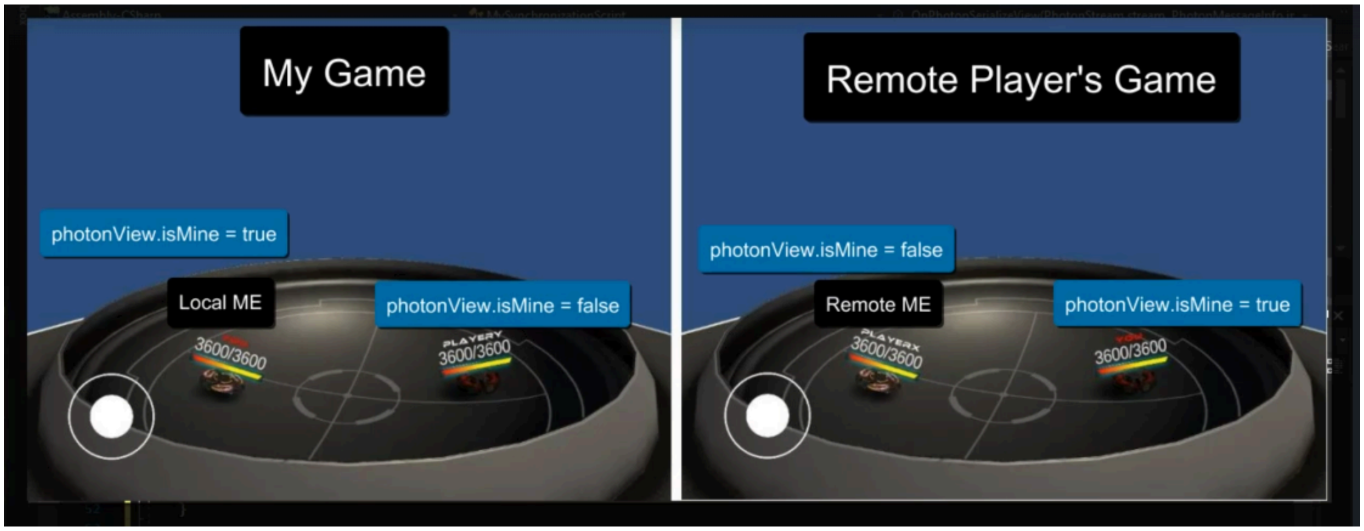
Owner:	Set at runtime	Fixed
View ID	Set at runtime	
Observe option:	Unreliable On Change	

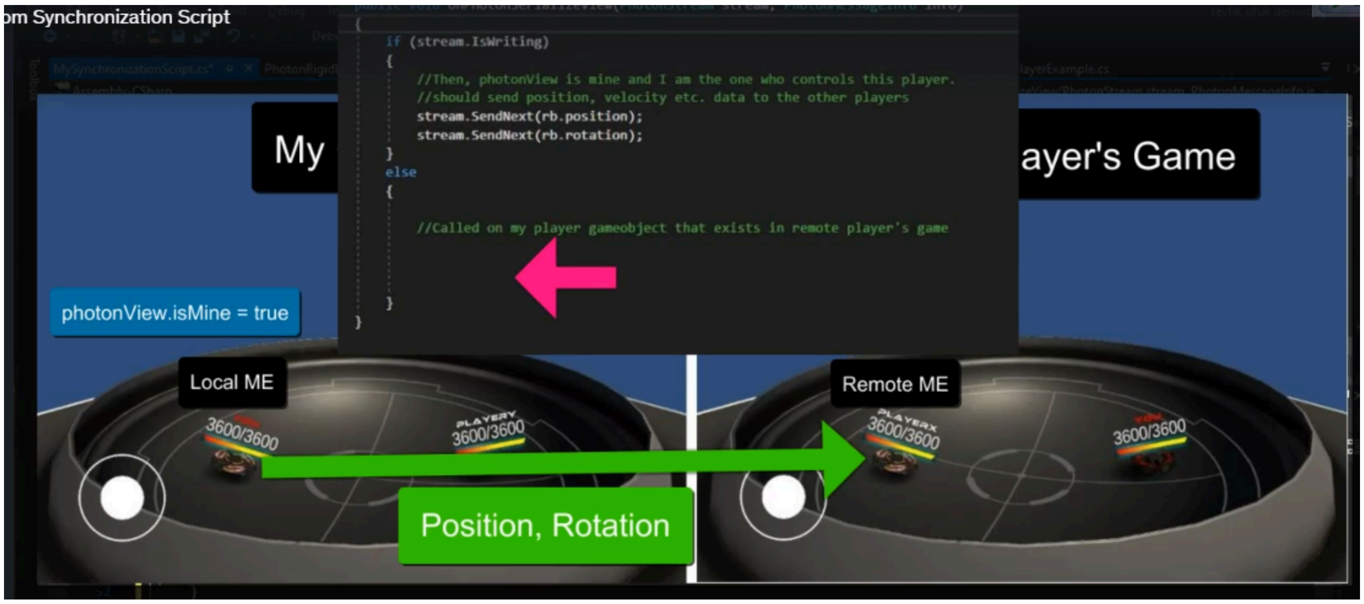
Observed Components (2)
- Player1_Attacker (PhotonRigidbodyView)
- Player1_Attacker (PhotonTransformView)

- ✓ Player Setup (Script)
- ✓ Photon Rigidbody View
  - Enable teleport for large distances
  - ✓ Synchronize Velocity
  - Synchronize Angular Velocity
- ✓ Photon Transform View
  - Synchronize Options
  - ✓ Position
  - ✓ Rotation

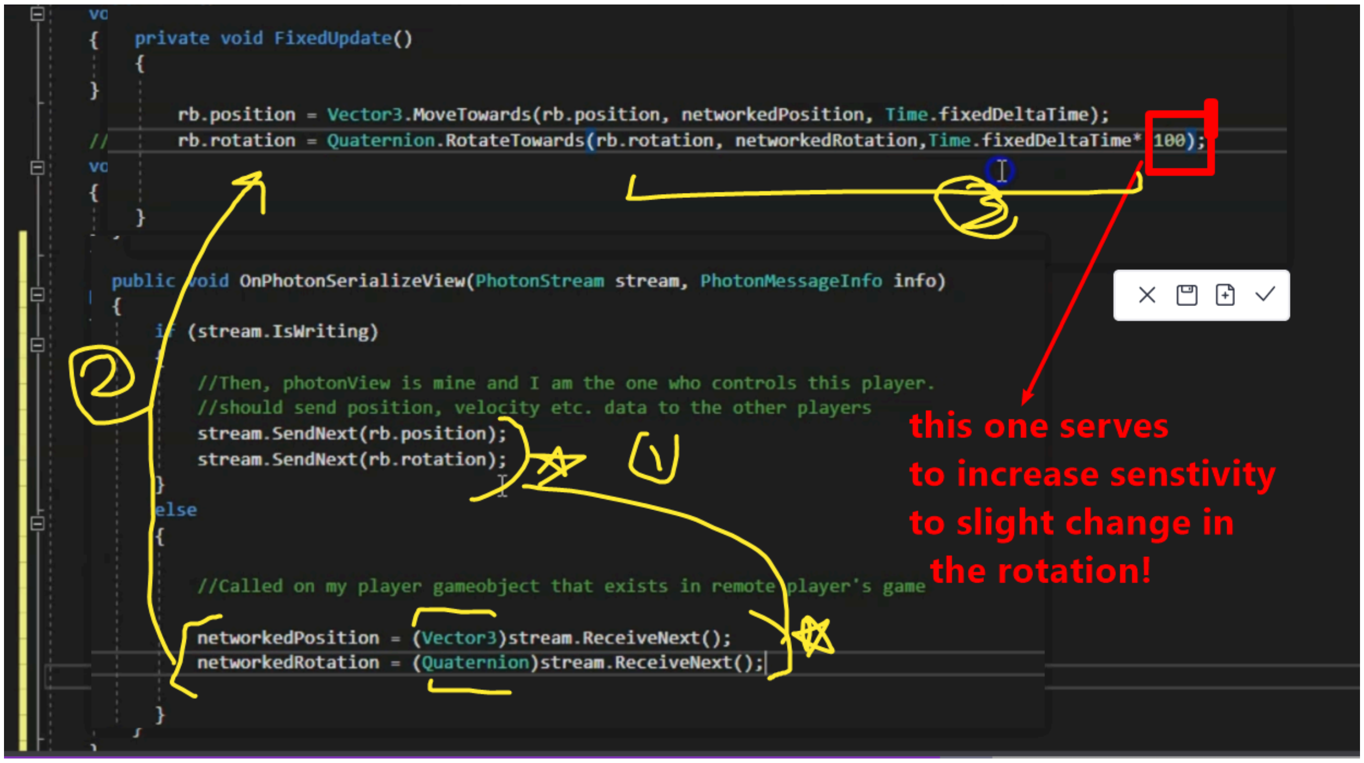
```
PhotonView has detected you dropped a Transform. Instead it's better to observe
UnityEngine.Debug:Log(Object)
PhotonView has detected you dropped a RigidBody. Instead it's better to observe a
UnityEngine.Debug:Log(Object)

PhotonView has detected you dropped a RigidBody. Instead it's better to observe a
PhotonRigidbodyView for better control and performances
UnityEngine.Debug:Log(Object)
Photon.Pun.PhotonViewInspector:DrawObservedComponentsList() (at
Assets/Photon/PhotonUnityNetworking/Code/Editor/PhotonViewInspector.cs:207)
Photon.Pun.PhotonViewInspector:OnInspectorGUI() (at
Assets/Photon/PhotonUnityNetworking/Code/Editor/PhotonViewInspector.cs:120)
UnityEngine.GUIUtility:ProcessEvent(Int32, IntPtr)
```

we'll write our own synchronization script over PLAINLY just PASTING RIGID-BODY or TRANSFORM there, observable-component cuz we have limited data restriction on free-plan



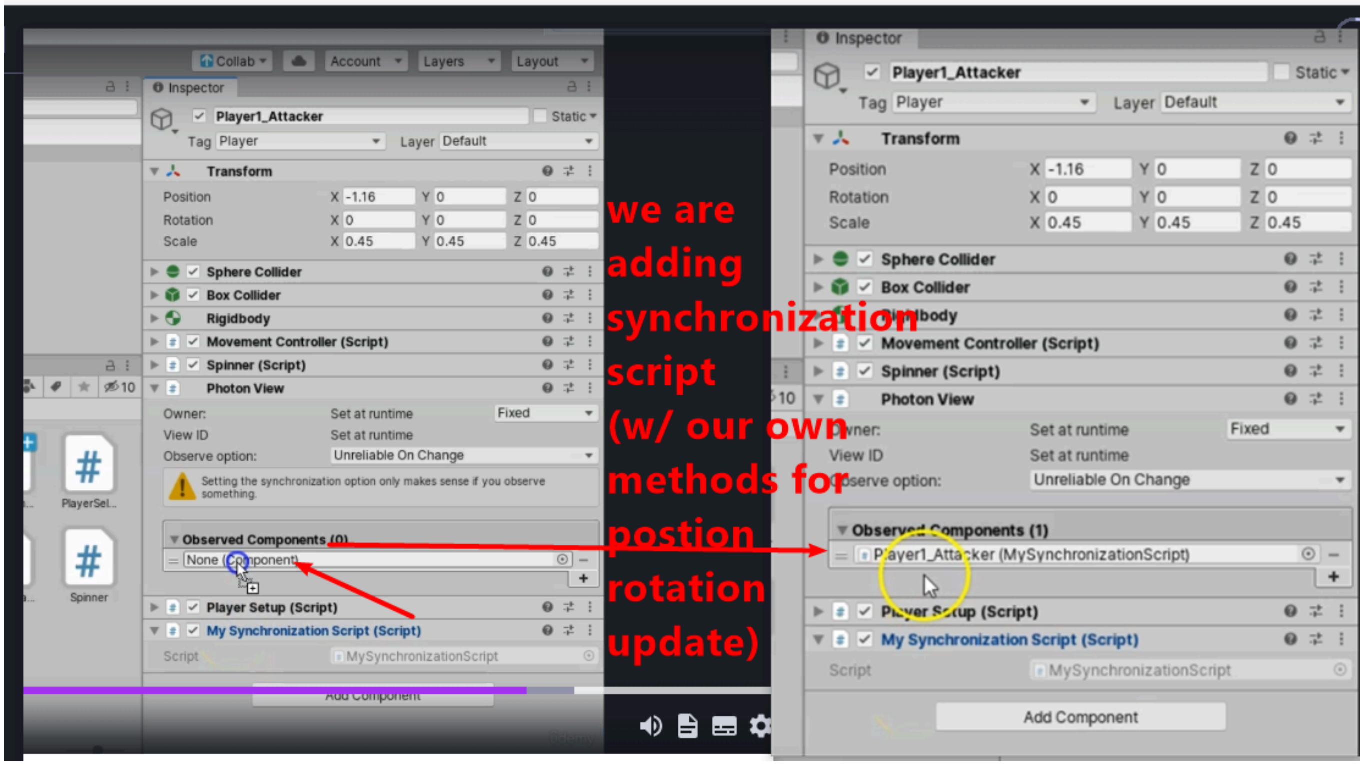


```
{
    if (stream.IsWriting)
    {
        //Then, photonView is mine and I am the one who controls this player.
        //should send position, velocity etc. data to the other players
        stream.SendNext(rb.position);
        stream.SendNext(rb.rotation);
    }
    else
    {

        //Called on my player gameobject that exists in remote player's game


    }
}
```



```
vo
{   private void FixedUpdate()
    {
}
        rb.position = Vector3.MoveTowards(rb.position, networkedPosition, Time.fixedDeltaTime);
//      rb.rotation = Quaternion.RotateTowards(rb.rotation, networkedRotation,Time.fixedDeltaTime* 100);
vo
{
    }

public void OnPhotonSerializeView(PhotonStream stream, PhotonMessageInfo info)
{
    if (stream.IsWriting)
    {
        //Then, photonView is mine and I am the one who controls this player.
        //should send position, velocity etc. data to the other players
        stream.SendNext(rb.position);
        stream.SendNext(rb.rotation);
    }
    else
    {

        //Called on my player gameobject that exists in remote player's game

        networkedPosition = (Vector3)stream.ReceiveNext();
        networkedRotation = (Quaternion)stream.ReceiveNext();

    }
```

2
1
this one serves to increase senstivity to slight change in the rotation!



Inspector

Player1_Attacker — Static
Tag Player — Layer Default

Transform
Position X -1.16 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 0.45 Y 0.45 Z 0.45

Sphere Collider
Box Collider
Rigidbody
Movement Controller (Script)
Spinner (Script)
Photon View
Owner: Set at runtime — Fixed
View ID: Set at runtime
Observe option: Unreliable On Change
Setting the synchronization option only makes sense if you observe something.
Observed Components (0)
None (Component)
Observed Components (1)
Player1_Attacker (MySynchronizationScript)
Player Setup (Script)
My Synchronization Script (Script)
Script: MySynchronizationScript
Add Component

we are adding synchronization script (w/ our own methods for postion rotation update)

LAG COMPENSATION

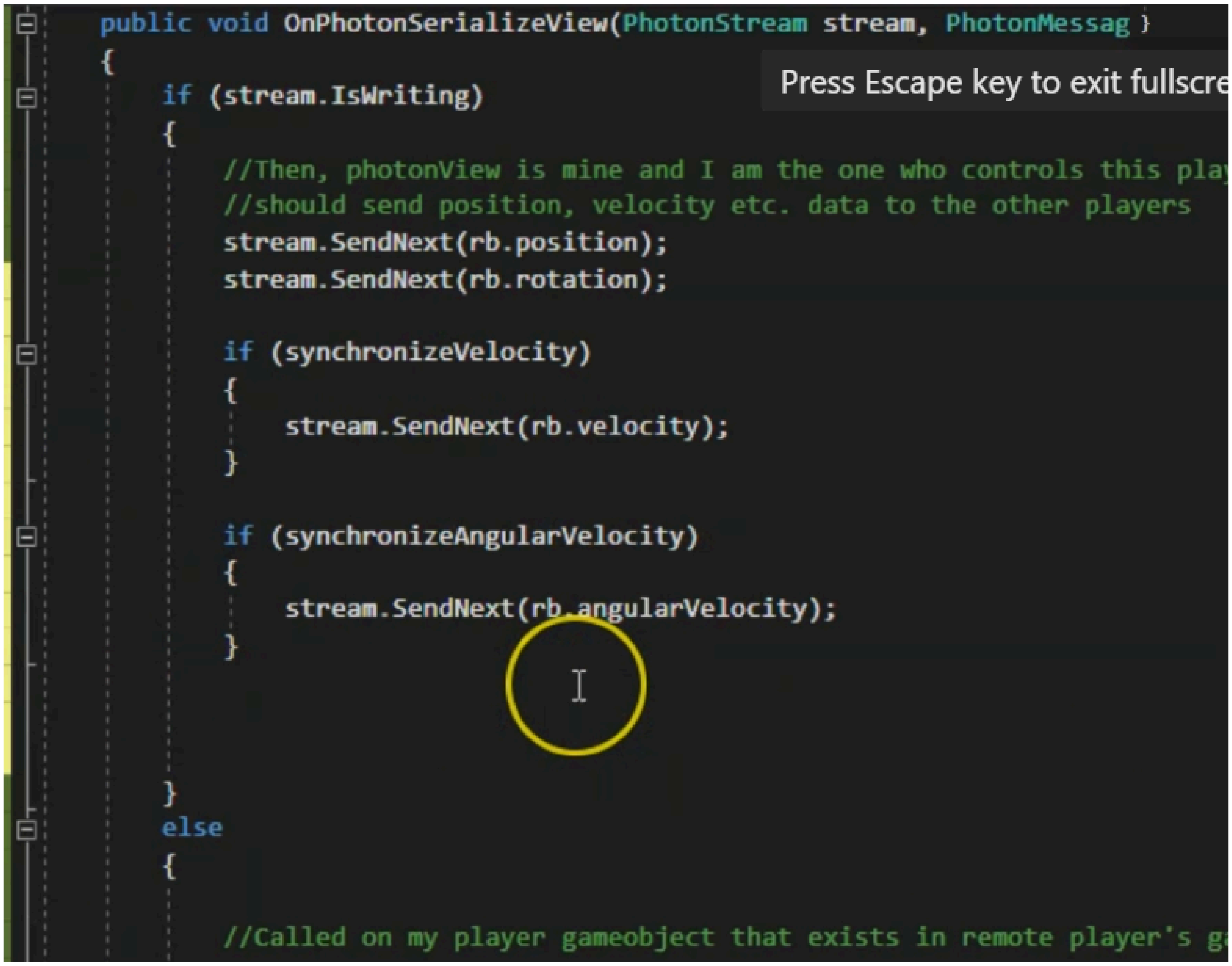
Ways we'll do this.

- Compute **Lag** = `Mathf.Abs((float)(PhotonNetwork.Time -info.SentServerTime))`
- **Predict Expected Position** - Acc To `PrevVelocityRecievedData`, by doing

`distancelag=velocity*lag`

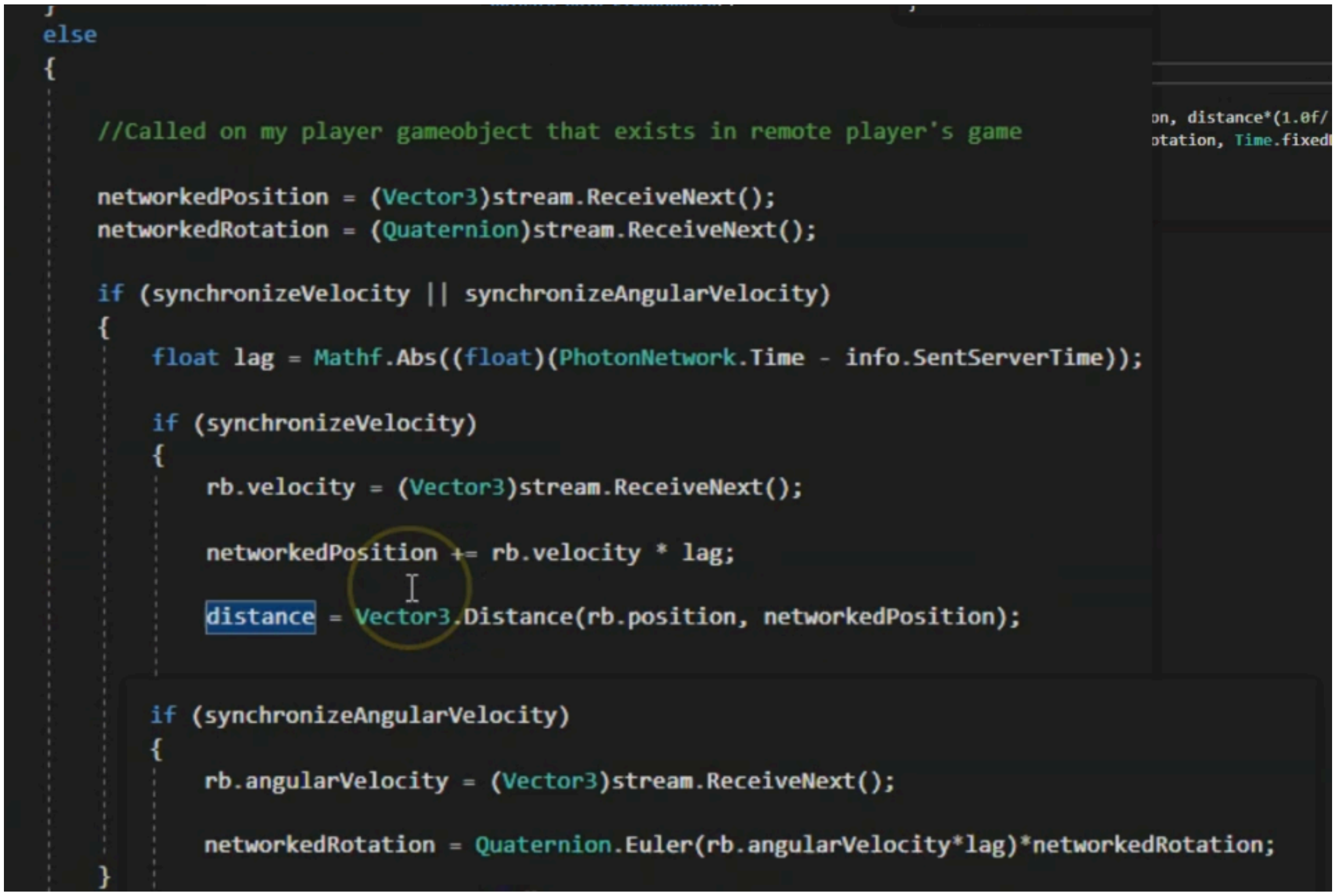
- TeleportCharacter if *difference b/w distance NOW_RECIEVED vs PREVIOUS* is beyond MinDistanceToTeleport
- Instead of Using `time.DeltaTime` - do - MoveTowards= `distancelag*(1.0f/PhotonNetwork.SerializationRate);` RotateTowards= `anglelag*(1.0f/PhotonNetwork.SerializationRate)*senstivity;` **What this does is, HIGHER THE `distancelag` / `anglelag` faster it'll move THE LERP**

Sending Data



```
public void OnPhotonSerializeView(PhotonStream stream, PhotonMessag }
{
    if (stream.IsWriting)
    {
        //Then, photonView is mine and I am the one who controls this play
        //should send position, velocity etc. data to the other players
        stream.SendNext(rb.position);
        stream.SendNext(rb.rotation);

        if (synchronizeVelocity)
        {
            stream.SendNext(rb.velocity);
        }

        if (synchronizeAngularVelocity)
        {
            stream.SendNext(rb.angularVelocity);
        }


    }
    else
    {

        //Called on my player gameobject that exists in remote player's g
```

Receiving Data

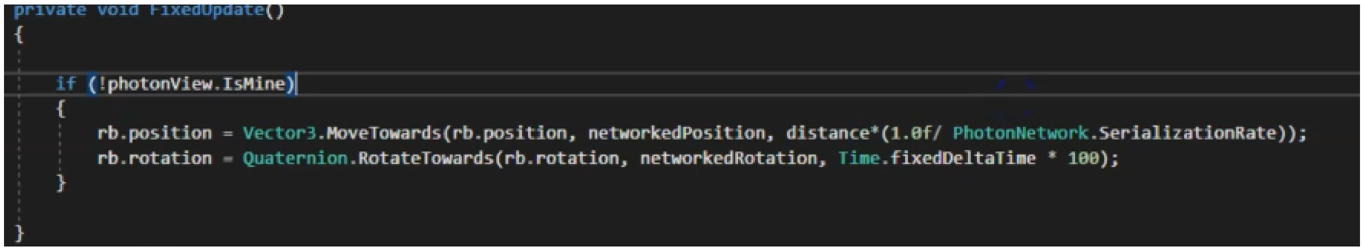


```
}
else
{

    //Called on my player gameobject that exists in remote player's game

    networkedPosition = (Vector3)stream.ReceiveNext();
    networkedRotation = (Quaternion)stream.ReceiveNext();

    if (synchronizeVelocity || synchronizeAngularVelocity)
    {
        float lag = Mathf.Abs((float)(PhotonNetwork.Time - info.SentServerTime));

        if (synchronizeVelocity)
        {
            rb.velocity = (Vector3)stream.ReceiveNext();

            networkedPosition += rb.velocity * lag;

            distance = Vector3.Distance(rb.position, networkedPosition);

        if (synchronizeAngularVelocity)
        {
            rb.angularVelocity = (Vector3)stream.ReceiveNext();

            networkedRotation = Quaternion.Euler(rb.angularVelocity*lag)*networkedRotation;
    }
```

```
on, distance*(1.0f/
otation, Time.fixed
```

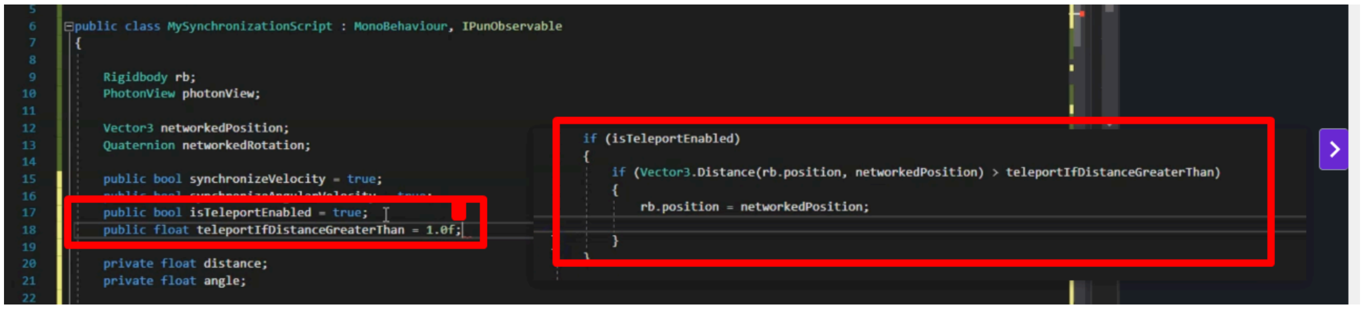
`distance` ain't used in `Vector3.MoveTowards()`, it's for `TeleportLogic`!!

Implementing Data



```
private void FixedUpdate()
{

    if (!photonView.IsMine)
    {
        rb.position = Vector3.MoveTowards(rb.position, networkedPosition, distance*(1.0f/ PhotonNetwork.SerializationRate));
        rb.rotation = Quaternion.RotateTowards(rb.rotation, networkedRotation, Time.fixedDeltaTime * 100);
    }

}
```

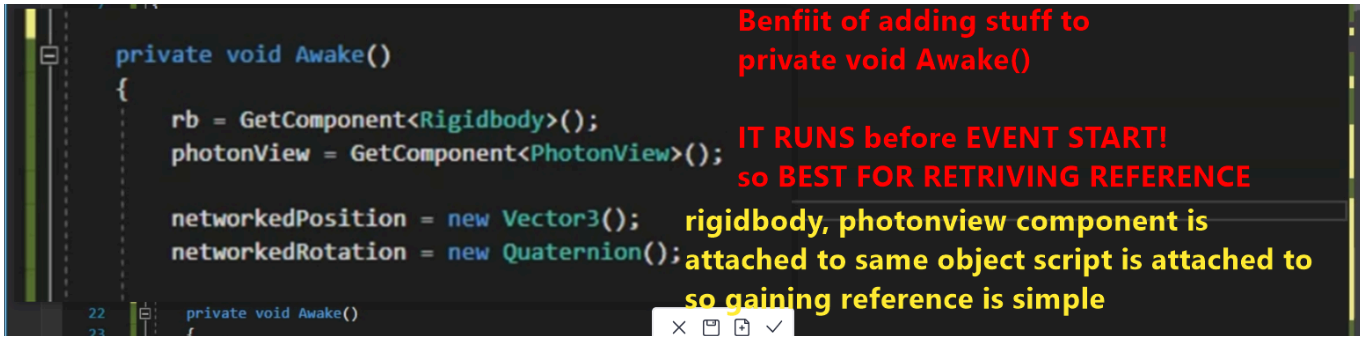
Teleport Logic



```
public class MySynchronizationScript : MonoBehaviour, IPunObservable
{

    Rigidbody rb;
    PhotonView photonView;

    Vector3 networkedPosition;
    Quaternion networkedRotation;

    public bool synchronizeVelocity = true;
    public bool synchronizeAngularVelocity = true;
    public bool isTeleportEnabled = true;
    public float teleportIfDistanceGreaterThan = 1.0f;

    private float distance;
    private float angle;
```

```
if (isTeleportEnabled)
{
    if (Vector3.Distance(rb.position, networkedPosition) > teleportIfDistanceGreaterThan)
    {
        rb.position = networkedPosition;

    }
}
```

USING VOID AWAKE BENIFITS



```
private void Awake()
{
    rb = GetComponent<Rigidbody>();
    photonView = GetComponent<PhotonView>();

    networkedPosition = new Vector3();
    networkedRotation = new Quaternion();
```

```
private void Awake()
```

Benfiit of adding stuff to
private void Awake()

IT RUNS before EVENT START!
so BEST FOR RETRIVING REFERENCE

rigidbody, photonview component is
attached to same object script is attached to
so gaining reference is simple