게임엔진프로그래밍응용

13. 네트워크 협동 게임 2

청강문화산업대학교 게임콘텐츠스쿨

반 경 진

공지사항

공지사항

- 2023학년도 1학기 방역 및 학사운영 방안
  https://www.ck.ac.kr/archives/193175
- 2023학년도 1학기 국가공휴일 및 대학 행사 수업 대체 일정 공지
  https://www.ck.ac.kr/archives/193109

온라인 수업 저작권 유의 사항

온라인 수업 저작권 유의 사항

온라인수업 저작권 유의사항 안내

강의 저작물을 다운로드, 캡처하여
교외로 유출하는 행위는
불 법 입 니 다

저작권자의 허락 없이 저작물을 복제, 공중송신 또는 배포하는 것은
저작권 침해에 해당하며 저작권법에 처벌받을 수 있습니다.

강의 동영상과 자료 일체는 교수 및 학교의 저작물로서 저작권이 보호됩니다.
수업자료를 무단 복제 또는 배포, 전송 시 민형사상 책임을 질 수 있습니다.

Index

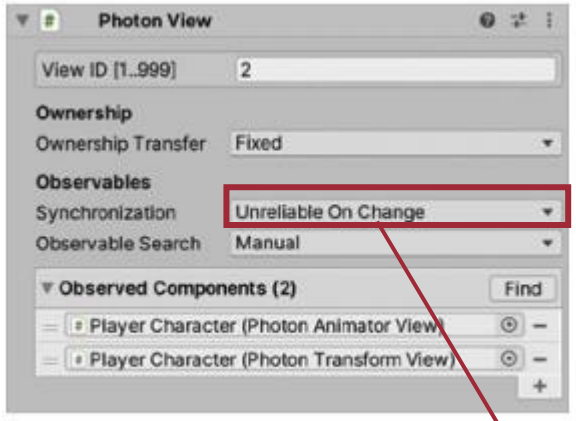
Photon 동기화

Photon 동기화

- **PhotonView**
  - 네트워크를 통해 동기화될 모든 게임 오브젝트는 Photon View 컴포넌트를 가져야함
  - 게임 오브젝트에 네트워크상에서 구별 가능한 식별자인 View ID를 부여
  - Observed Components 리스트에 등록된 컴포넌트들의 변화한 수치를 관측하고, 네트워크를 넘어서 다른 클라이언트에 전달
  - 로컬과 리모트의 구분 가능(PhotonView IsMine)
  - IPunObservable 인터페이스를 상속한 컴포넌트 관측 및 동기화 가능
  - RPC(Remote Procedure Call)
  - MonoBehaviourPun를 상속받으면 PhotonView를 멤버로 사용할 수 있다.

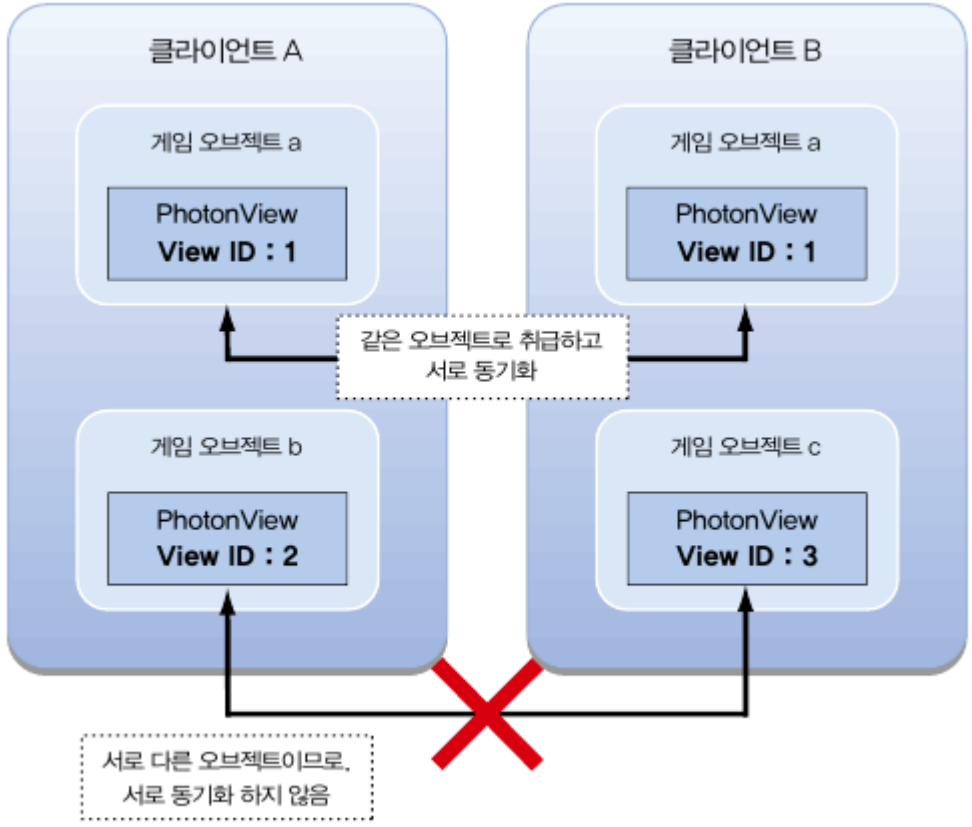
Photon 동기화

- PhotonView



▶ Photon View 컴포넌트

- Off : 동기화하지 않습니다.
- Reliable Delta Compressed : 상대방이 최근에 수신한 값과 동일한 값은 송신하지 않습니다.
- Unreliable : 패킷의 수신 여부를 검사하지 않고 지속적으로 송신합니다.
- Unreliable On Change : Unreliable과 동일하나 값의 변화가 감지될 때만 송신합니다.



▶ View ID를 이용해 같은 게임 오브젝트를 식별

Photon 동기화

- PhotonView

```
void PhotonView.RPC ( string          methodName,
                      PhotonTargets   target,
                      params object[] parameters
                    )
```

접속한 룸안의 리모트 클라이언트(들)로 이 게임 오브젝의 RPC 메소드를 호출

Photon 동기화

- PhotonView

enum PhotonTargets

RPC 의 "target" 옵션들입니다. 어떤 원격 클라이언트들이 RPC 호출을 수신 할지를 정의 합니다.

Enumerator	
All	RPC를 모두에게 전송하고 클라이언트에서 즉시 수행 합니다. 나중에 참여한 플레이어는 이 RPC를 수행하지는 않습니다.
Others	RPC를 모두에게 전송합니다. 클라이언트는 RPC를 수행하지 않습니다. 나중에 참여한 플레이어는 이 RPC를 수행하지는 않습니다.
MasterClient	RPC를 MasterClient에게만 전송 합니다. 주의: MasterClient는 RPC를 수행하기전에 연결해제가 되어 RPC가 없어지는 원인이 될 수 있습니다.
AllBuffered	RPC를 모두에게 전송하고 클라이언트에서 즉시 수행 합니다. 버퍼로 기록 되기 때문에 새로운 플레이어가 참여할 때 이 RPC를 받게 됩니다(이 클라이언트가 떠나기 전까지).
OthersBuffered	RPC를 모두에게 전송합니다. 이 클라이언트는 RPC를 수행하지 않습니다.버퍼로 기록 되기 때문에 새로운 플레이어가 참여할 때 이 RPC를 받게 됩니다(이 클라이언트가 떠나기 전까지).
AllViaServer	서버를 통해 이 클라이언트를 포함한 모두에게 RPC를 전송 합니다. 이 클라이언트는 다른 것과 같이 서버에서 수신 되었을 때 RPC를 실행 합니다. 잇점: 서버의 RPC 전송순서는 모든 클라이언트에 동일 합니다.
AllBufferedViaServer	이 클라이언트를 포함한 모두에게 RPC 를 서버를 통하여 전송하고 나중에 참여할 플레이어를 위해서 버퍼화 합니다. 이 클라이언트는 다른 것과 같이 서버에서 수신 되었을 때 RPC를 실행 합니다. 잇점: 서버의 RPC 전송순서는 모든 클라이언트에 동일 합니다.

Photon 동기화

- **IPunObservable**
  - 관찰 할 수 있는 스크립트들의 정확한 구현을 쉽게 하기 위하여 OnPhotonSerializeView 메소드를 정의
  - PUN 에 의해서 초당 여러번 호출. 따라서 스크립트에서 PhotonView 의 동기화 데이터를 읽고 쓸 수 있음
  - 다른 콜백과 달리 OnPhotonSerializeView 는 PhotonView.observed 스크립트로 PhotonView 에 지정되어 있을 때만 호출

Photon 동기화

- IPunObservable

```
/// <summary>Defines the OnPhotonSerializeView method to make it easy to implement correctly for observable scripts.</summary>
/// \ingroup callbacks
참조 19개
public interface IPunObservable
{
    /// <summary>
    /// Called by PUN several times per second, so that your script can write and read synchronization data for the PhotonView.
    /// </summary>
    /// <remarks>
    /// This method will be called in scripts that are assigned as Observed component of a PhotonView.<br/>
    /// PhotonNetwork.SerializationRate affects how often this method is called.<br/>
    /// PhotonNetwork.SendRate affects how often packages are sent by this client.<br/>
    ///
    /// Implementing this method, you can customize which data a PhotonView regularly synchronizes.
    /// Your code defines what is being sent (content) and how your data is used by receiving clients.
    ///
    /// Unlike other callbacks, <i>OnPhotonSerializeView only gets called when it is assigned
    /// to a PhotonView</i> as PhotonView.observed script.
    ///
    /// To make use of this method, the PhotonStream is essential. It will be in "writing" mode" on the
    /// client that controls a PhotonView (PhotonStream.IsWriting == true) and in "reading mode" on the
    /// remote clients that just receive that the controlling client sends.
    ///
    /// If you skip writing any value into the stream, PUN will skip the update. Used carefully, this can
    /// conserve bandwidth and messages (which have a limit per room/second).
    ///
    /// Note that OnPhotonSerializeView is not called on remote clients when the sender does not send
    /// any update. This can't be used as "x-times per second Update()".
    /// </remarks>
    /// \ingroup publicApi
    참조 14개
    void OnPhotonSerializeView(PhotonStream stream, PhotonMessageInfo info);
}
```

Photon 동기화

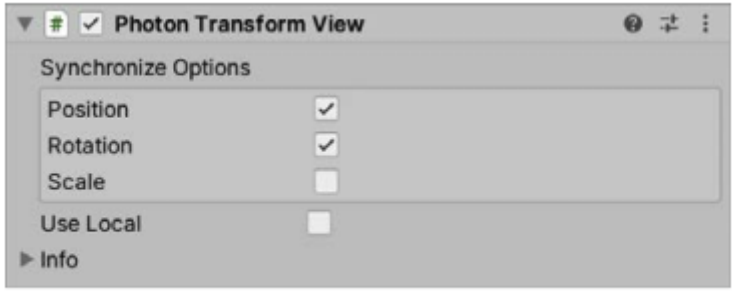
- IPunObservable

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

/// <summary>
/// 점수와 게임 오버 여부를 관리하는 게임 매니저
/// </summary>
Unity 스크립트(자산 참조 1개)|참조 9개
public class GameManager : MonoBehaviourPunCallbacks, IPunObservable
{
```

```
참조 3개
public void OnPhotonSerializeView(PhotonStream stream, PhotonMessageInfo info)
{
    if (stream.IsWriting)
    {
        stream.SendNext(score);
    }
    else
    {
        score = (int) stream.ReceiveNext();

        UIManager.instance.UpdateScoreText(score);
    }
}
```

Photon 동기화

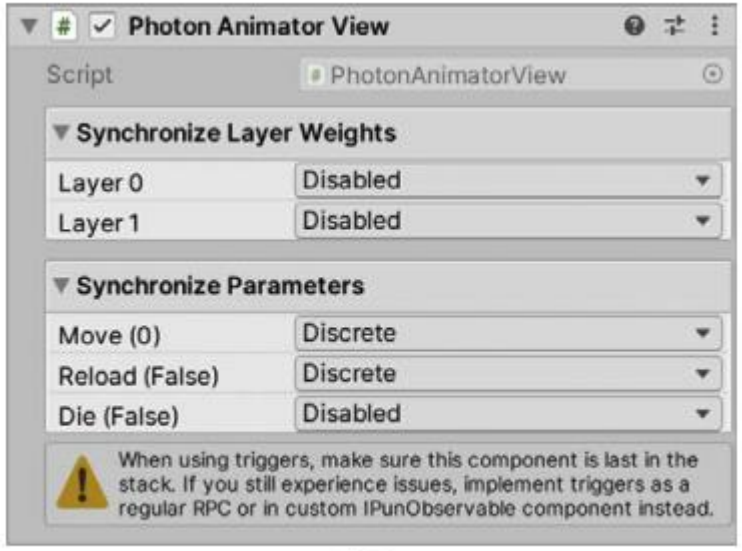
- **PhotonTransformView**
  - 자신의 게임 오브젝트에 추가된 트랜스폼 컴포넌트의 값의 변화를 측정하고, Photon View 컴포넌트를 사용해 동기화
  - 자신이 로컬이라면 트랜스폼의 속성값을 감지하고 리모트에 전송
  - 자신이 리모트라면 송신된 로컬의 값을 받아 자신의 트랜스폼 컴포넌트에 적용
  - Photon View 컴포넌트 없이는 동작 안함



Photon Transform View 컴포넌트

Photon 동기화

- **PhotonAnimatiorView**
  - 로컬 게임 오브젝트와 리모트 게임 오브젝트 사이에서 애니메이터 컴포넌트의 파라미터를 동기화
  - Discrete - 이산적으로 동기화(Continuous 보다 적은 대역폭 사용)
  - Continuous - 연속적인 변화를 동기화
  - Disabled - 동기화 안함



Photon Animator View 컴포넌트

Photon 타입등록(Serialization)

Photon 타입등록(Serialization)

- Photon 에서 지원하는 데이터 타입

타입 (C#)	크기 [bytes]	설명
byte	2	8 bit unsigned
boolean	2	true or false
short	3	16 bit
int	5	32 bit
long	9	64 bit
float	5	32 bit
double	9	64 bit
string	3 + size(UTF8.GetBytes(string))	< short.MaxValue length
byte-array	5 + 1 * length	< int.MaxValue length
int-array	5 + 4 * length	< int.MaxValue length
<type> 배열	4 + size(entries) - count(entries)	< short.MaxValue length
hashtable	3 + size(keys) + size(values)	< short.MaxValue pairs
dictionary	3 + size(keys) + size(values)	< short.MaxValue pairs K- 또는 V-타입이 객체인 경우 추가적인 크기

Photon 타입등록(Serialization)

- **Photon 유니티 네트워킹의 추가 타입**

타임 (C#)	크기 [bytes]	설명
Vector2	12	2 floats
Vector3	16	3 floats
Quaternion	20	4 floats
PhotonPlayer	8	integer PhotonPlayer.ID

Photon 타입등록(Serialization)

- **커스텀 타입**
  - 직렬화/비직렬화
  - 기본적인 개념은 두 개의 메소드를 통해 클래스를 byte-배열로 그리고 byte-배열을 클래스로 해주는 것을 구현
  - 스트림 버퍼를 사용한 직렬화/비직렬화
  - 이 클래스를 Photon API 에 등록

Photon 타입등록(Serialization)

- **커스텀 타입**
  - Byte Array Method

```
public delegate byte[] SerializeMethod(object customObject);
public delegate object DeserializeMethod(byte[] serializedCustomObject);
```

```
PhotonPeer.RegisterType(Type customType, byte code, SerializeMethod serializeMethod,
                        DeserializeMethod deserializeMethod)
```

Photon 타입등록(Serialization)

```
static bool ExitGames.Client.Photon.PhotonPeer.RegisterType ( Type               customType,
                                                              byte               code,
                                                              SerializeMethod    serializeMethod,
                                                              DeserializeMethod  constructor
                                                            )
```

Registers new types/classes for de/serialization and the fitting methods to call for this type.

SerializeMethod and DeserializeMethod are complementary: Feed the product of serializeMethod to the constructor, to get a comparable instance of the object.

After registering a Type, it can be used in events and operations and will be serialized like built-in types.

Parameters

customType	Type (class) to register.
code	A byte-code used as shortcut during transfer of this Type.
serializeMethod	Method delegate to create a byte[] from a customType instance.
constructor	Method delegate to create instances of customType's from byte[].

Returns

If the Type was registered successfully.

Photon 타입등록(Serialization)

```
public class MyCustomType
{
  public byte Id { get; set; }

  public static object Deserialize(byte[] data)
  {
    var result = new MyCustomType();
    result.Id = data[0];
    return result;
  }

  public static byte[] Serialize(object customType)
  {
    var c = (MyCustomType)customType;
    return new byte[] { c.Id };
  }
}
```

```
PhotonPeer.RegisterType(typeof(MyCustomType), myCustomTypeCode, MyCustomType.Serialize,
                        MyCustomType.Deserialize);
```

Photon 타입등록(Serialization)

- **커스텀 타입**
  - StreamBuffer Method

```
public delegate short SerializeStreamMethod(StreamBuffer outStream, object customobject);
public delegate object DeserializeStreamMethod(StreamBuffer inStream, short length);
```

```
RegisterType(Type customType, byte code, SerializeStreamMethod serializeMethod,
             DeserializeStreamMethod deserializeMethod)
```

Photon 타입등록(Serialization)

```
static bool ExitGames.Client.Photon.PhotonPeer.RegisterType ( Type                    customType,
                                                              byte                    code,
                                                              SerializeStreamMethod   serializeMethod,
                                                              DeserializeStreamMethod constructor
                                                            )                                          inline static
```

Photon 타입등록(Serialization)

```
public static readonly byte[] memVector2 = new byte[2 * 4];
private static short SerializeVector2(StreamBuffer outStream, object customobject)
{
    Vector2 vo = (Vector2)customobject;
    lock (memVector2)
    {
        byte[] bytes = memVector2;
        int index = 0;
        Protocol.Serialize(vo.x, bytes, ref index);
        Protocol.Serialize(vo.y, bytes, ref index);
        outStream.Write(bytes, 0, 2 * 4);
    }

    return 2 * 4;
}
```

Photon 타입등록(Serialization)

```
private static object DeserializeVector2(StreamBuffer inStream, short length)
{
    Vector2 vo = new Vector2();
    lock (memVector2)
    {
        inStream.Read(memVector2, 0, 2 * 4);
        int index = 0;
        Protocol.Deserialize(out vo.x, memVector2, ref index);
        Protocol.Deserialize(out vo.y, memVector2, ref index);
    }

    return vo;
}
```

```
PhotonPeer.RegisterType(typeof(Vector2), (byte)'W', SerializeVector2, DeserializeVector2);
```

Photon 타입등록(Serialization)

- **커스텀 타입**
  - UnityEngine.Color
  - ColorSerialization.cs - static method 사용

Photon 타입등록(Serialization)

```
using ExitGames.Client.Photon;
using UnityEngine;

참조 2개
public class ColorSerialization
{
    private static byte[] colorMemmory = new byte[4 * 4];

    참조 1개
    public static short SerializeColor(StreamBuffer outStream, object targetObject)
    {
        Color color = (Color) targetObject;

        lock (colorMemmory)
        {
            byte[] bytes = colorMemmory;
            int index = 0;

            Protocol.Serialize(color.r, bytes, ref index);
            Protocol.Serialize(color.g, bytes, ref index);
            Protocol.Serialize(color.b, bytes, ref index);
            Protocol.Serialize(color.a, bytes, ref index);
            outStream.Write(bytes, 0, 4 * 4);
        }

        return 4 * 4;
    }
```

Photon 타입등록(Serialization)

```
참조 1개
public static object DeserializeColor(StreamBuffer inStream, short length)
{
    Color color = new Color();

    lock (colorMemmory)
    {
        inStream.Read(colorMemmory, 0, 4 * 4);
        int index = 0;

        Protocol.Deserialize(out color.r, colorMemmory, ref index);
        Protocol.Deserialize(out color.g, colorMemmory, ref index);
        Protocol.Deserialize(out color.b, colorMemmory, ref index);
        Protocol.Deserialize(out color.a, colorMemmory, ref index);
    }

    return color;
}
}
```

```
Unity 메시지 | 참조 0개
private void Awake()
{
    PhotonPeer.RegisterType(typeof(Color), 128, ColorSerialization.SerializeColor,
        ColorSerialization.DeserializeColor);
}
```

ZombieSpawner

Photon RPC

Photon RPC

- **RPC**
  - 원격 프로시져 호출은 이름 그대로 (같은 룸에 있는)원격 클라이언트에 있는 메소드를 호출
    메소드의 이름을 문자열로 호출
  - 메소드에 대해서 원격 호출을 사용하려면 [PunRPC] 속성을 적용
  - 스크립트가 MonoBehaviourPun를 상속 하면 this.photonView.RPC() 를 사용 할 수 있음
  - Shotcut - 문자열은 네트워크를 통한 전송에서 가장 비효율적이기 때문에 PUN은 문자열을 줄여서 전송하는 트릭을 사용
    RPC 목록은 PhotonServerSettings 를 통해서 저장되고 관리

Photon RPC

```
[PunRPC]
void ChatMessage(string a, string b)
{
    Debug.Log(string.Format("ChatMessage {0} {1}", a, b));
}
```

```
PhotonView photonView = PhotonView.Get(this);
photonView.RPC("ChatMessage", RpcTarget.All, "jup", "and jup!");
```

```
[PunRPC]
void ChatMessage(string a, string b, PhotonMessageInfo info)
{
    // the photonView.RPC() call is the same as without the info parameter.
    // the info.Sender is the player who called the RPC.
    Debug.Log(string.Format("Info: {0} {1} {2}", info.Sender, info.photonView, info.timestamp));
}
```

Photon RPC

- **RPC의 타이밍과 로딩 레벨**
  - RPC는 특정 PhotonViews에서 호출 되며 수신측에서 매칭되는 하나의 클라이언트가 목표 지점
    만약 원격 클라이언트가 로드되지 않았거나 아직 매칭되는 PhotonView 생성을 하지 못했으면
    RPC 는 손실
  - RPC 손실에 대한 전형적인 원인은 클라이언트가 새로운 신(scene)을 로드 할 때
    접속 전에 PhotonNetwork.automaticallySyncScene = true 로 설정하고 룸의 마스터 클라이언트
    에게 PhotonNetwork.LoadLevel() 를 사용

Photon 오브젝트 생성, 삭제

Photon 오브젝트 생성, 삭제

- **게임 오브젝트의 생명주기 관리**

  - Instantiate
    네트워크 객체를 생성하기 위해서는 유니티의 Instantiate 대신 PhotonNetwork.Instantiate 를 사용
    PUN은 내부적으로 PhotonNetwork.PrefabPool 에서 GameObject를 가져와, 네트워크용으로 설정하고 사용
    모든 프리팹은 PhotonView 컴포넌트가 있어야됨
    Resources 폴더에서 프리팹을 로드하고 나중에 GameObject를 파괴하는 DefaultPool을 사용
    좀 더 복잡한 IPunPrefabPool 구현은 파괴할 때 객체를 반환할 수 있고 인스턴스 생성에 재사용할 수 있음. 이런 경우에, GameObjects는 Instantiate 에서 실제 생성되지 않으며, 이 의미는 이러한 경우에 있어 유니티가 Start() 를 호출하지 않는다는 의미.
    이로 인하여, 네트워크 게임 객체들에 있는 스크립트들은 OnEnable 과 OnDisable 을 반드시 구현

  - Destroy

Photon 오브젝트 생성, 삭제

- 게임 오브젝트의 생명주기 관리

```
static GameObject PhotonNetwork.Instantiate ( string      prefabName,
                                              Vector3     position,
                                              Quaternion  rotation,
                                              int         group
                                            )
```

static

네트워크 상에서 prefab 의 인스턴스를 생성 합니다. 프리팹은 루트의 "Resources" 폴더에 있어야 합니다.

Resources 폴더에서 프리팹을 사용하는 대신에 수동으로 인스턴스 생성을 하고 PhotonView 를 할당 할 수 있습니다. 문서를 살펴보세요.

Parameters

prefabName 인스턴스화 할 프리팹 이름.

position 인스턴스에 적용될 Vector3 Position.

rotation 인스턴스에 적용될 Rotation Quaternion.

group 이 PhotonView 의 그룹.

Returns

PhotonView 가 초기화된 새로운 게임오브젝트의 인스턴스.

Photon 오브젝트 생성, 삭제

- 게임 오브젝트의 생명주기 관리

static void PhotonNetwork.Destroy (GameObject targetGo) static

static 이 아니거나 이 클라이언트의 통제하에 있지 않은 게임오브젝트 네트워크-제거.

네트워크 게임오브젝트를 제거하면 다음도 포함됩니다:

- 서버의 룸 버퍼로 부터 오는 인스턴스생성 호출 제거.
- **PhotonNetwork.Instantiate** 호출로 비간접적으로 생성된 PhotonViews의 RPC 버퍼 제거.
- 다른 클라이언트들에게 게임 오브젝트를 제거 하라는 메시지 전송(네트워크 Lag 에 영향을 미침).

일반적으로 룸을 떠날 때 게임오브젝트들은 자동으로 제거 됩니다. 룸에 있지 않을 동안 게임 오브젝트를 제거해야 한다면 Destroy 는 로컬에서만 수행 됩니다.

네트워크 객체들을 제거 하는 것은 **PhotonNetwork.Instantiate()** 에 의해서 생성된 것에만 동작 합니다. 씬에서 로드된 객체들은 **PhotonView** 컴포넌트를 가지고 있어도 무시됩니다.

게임오브젝트는 이 클라이언트의 제어하에 있어야 합니다:

- 이 클라이언트에 의해 인스턴스가 생성되고 소유 됨.
- 룸을 나간 플레이어들의 인스턴스생성된 객체들은 마스터 클라이언트에 의해 제어 됩니다.
- 씬이 소유한 게임오브젝트들은 마스터 클라이언트에 의해 제어 됩니다.
- 클라이언트가 룸에 없는 동안 게임오브젝트는 제거 될 수 있습니다.

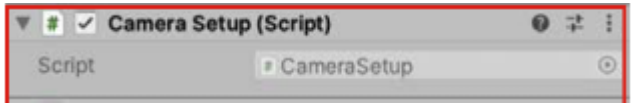
Returns

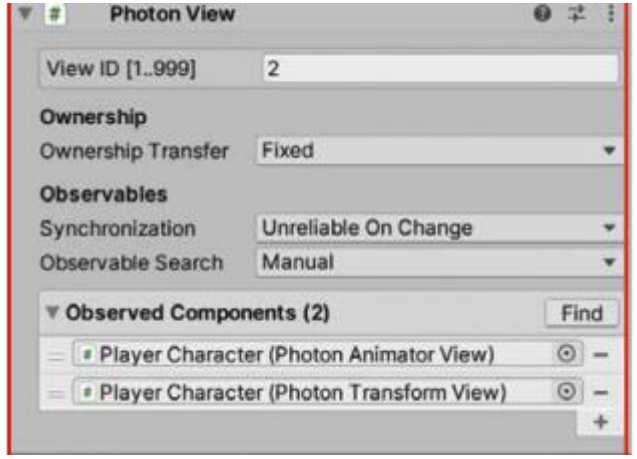
없습니다. 이슈가 발생 했을 때 에러 디버그 로그를 확인 해 주세요.

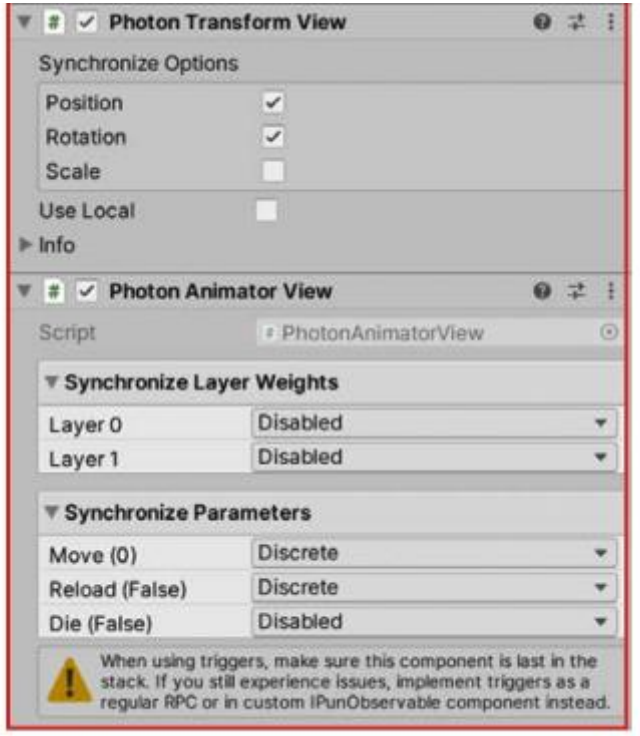
네트워크 Player, Follow Cam

네트워크 Player, Follow Cam

- Player Character







▶ 새로 추가된 컴포넌트

네트워크 Player, Follow Cam

- **CameraSetup**
  시네머신 가상 카메라가 로컬 플레이어만 추적
- **PlayerInput**
  로컬 플레이어 캐릭터인 경우에만 사용자 입력 감지
- **PlayerMovement**
  로컬 플레이어 캐릭터인 경우에만 이동, 회전, 애니메이션
- **PlayerShooter**
  로컬 플레이어 캐릭터인 경우에만 사격 및 탄알 UI 갱신
- **LivingEntity**
  호스트에서만 체력 관리와 대미지 처리 (RPC 이용 동기화)
- **PlayerHealth**
  리스폰 기능 추가, 체력 아이템 호스트에서만 사용 (RPC 이용 동기화)
- **Gun**
  실제 사격 처리 부분을 호스트에서만 실행, 상태 동기화 (RPC 이용 동기화)

네트워크 Player, Follow Cam

- CameraSetup

```
using Cinemachine; // 시네머신 관련 코드
using Photon.Pun; // PUN 관련 코드
using UnityEngine;

// 시네머신 카메라가 로컬 플레이어를 추적하도록 설정
public class CameraSetup : MonoBehaviourPun {
    void Start() {
        // 만약 자신이 로컬 플레이어라면
        if (photonView.IsMine)
        {
            // 씬에 있는 시네머신 가상 카메라를 찾고
            CinemachineVirtualCamera followCam =
                FindObjectOfType<CinemachineVirtualCamera>();
            // 가상 카메라의 추적 대상을 자신의 트랜스폼으로 변경
            followCam.Follow = transform;
            followCam.LookAt = transform;
        }
    }
}
```

네트워크 Player, Follow Cam

- PlayerInput

```
using Photon.Pun;
using UnityEngine;

// 플레이어 캐릭터를 조작하기 위한 사용자 입력을 감지
// 감지된 입력값을 다른 컴포넌트가 사용할 수 있도록 제공
public class PlayerInput : MonoBehaviourPun {
    // 변수 선언부 생략(기존 코드와 동일함)

    // 매 프레임 사용자 입력을 감지
    private void Update() {
        // 로컬 플레이어가 아닌 경우 입력을 받지 않음
        if (!photonView.IsMine)
        {
            return;
        }
        // 게임오버 상태에서는 사용자 입력을 감지하지 않음
        if (GameManager.instance != null
            && GameManager.instance.isGameover)
        {
            move = 0;
            rotate = 0;
            fire = false;
            reload = false;
            return;
        }

        // move에 관한 입력 감지
        move = Input.GetAxis(moveAxisName);
        // rotate에 관한 입력 감지
        rotate = Input.GetAxis(rotateAxisName);
        // fire에 관한 입력 감지
        fire = Input.GetButton(fireButtonName);
        // reload에 관한 입력 감지
        reload = Input.GetButtonDown(reloadButtonName);
    }
}
```

네트워크 Player, Follow Cam

- PlayerMovement

```
using Photon.Pun;
using UnityEngine;

// 플레이어 캐릭터를 사용자 입력에 따라 움직이는 스크립트
public class PlayerMovement : MonoBehaviourPun {
    // 변수 선언부 생략(기존 코드와 동일함)

    private void Start() {
        // 코드 생략(기존 코드와 동일함)
    }

    // FixedUpdate는 물리 갱신 주기에 맞춰 실행됨
    private void FixedUpdate() {
        // 로컬 플레이어만 직접 위치와 회전 변경 가능
        if (!photonView.IsMine)
        {
            return;
        }
        // 회전 실행
        Rotate();
        // 움직임 실행
        Move();

        // 입력값에 따라 애니메이터의 Move 파라미터값 변경
        playerAnimator.SetFloat("Move", playerInput.move);
    }

    // 입력값에 따라 캐릭터를 앞뒤로 움직임
    private void Move() {
        // 코드 생략(기존 코드와 동일함)
    }

    // 입력값에 따라 캐릭터를 좌우로 회전
    private void Rotate() {
        // 코드 생략(기존 코드와 동일함)
    }
}
```

네트워크 Player, Follow Cam

- PlayerShooter

```
using Photon.Pun;
using UnityEngine;

// 주어진 Gun 오브젝트를 쏘거나 재장전
// 알맞은 애니메이션을 재생하고 IK를 사용해 캐릭터 양손이 총에 위치하도록 조정
public class PlayerShooter : MonoBehaviourPun {
    // 변수 선언부 생략(기존 코드와 동일함)

    private void Start() {
        // 코드 생략(기존 코드와 동일함)
    }

    private void OnEnable() {
        // 코드 생략(기존 코드와 동일함)
    }

    private void OnDisable() {
        // 코드 생략(기존 코드와 동일함)
    }
```

```
private void Update() {
    // 로컬 플레이어만 총을 직접 사격. 탄알 UI 갱신 가능
    if (!photonView.IsMine)
    {
        return;
    }

    // 입력을 감지하여 총을 발사하거나 재장전
    if (playerInput.fire)
    {
        // 발사 입력 감지 시 총 발사
        gun.Fire();
    }
    else if (playerInput.reload)
    {
        // 재장전 입력 감지 시 재장전
        if (gun.Reload())
        {
            // 재장전 성공 시에만 재장전 애니메이션 재생
            playerAnimator.SetTrigger("Reload");
        }
    }

    // 남은 탄알 UI 갱신
    UpdateUI();
}
```

네트워크 Player, Follow Cam

- LivingEntity

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

// 생명체로 동작할 게임 오브젝트를 위한 뼈대 제공
// 체력, 대미지 받아들이기, 사망 기능, 사망 이벤트 제공
public class LivingEntity : MonoBehaviourPun, IDamageable {
    // 변수 선언부 생략(기존 코드와 동일함)

    // 호스트->모든 클라이언트 방향으로 체력과 사망 상태를 동기화하는 메서드
    [PunRPC]
    public void ApplyUpdatedHealth(float newHealth, bool newDead) {
        health = newHealth;
        dead = newDead;
    }

    // 생명체가 활성화될 때 상태 리셋
    protected virtual void OnEnable() {
        // 사망하지 않은 상태로 시작
        dead = false;
        // 체력을 시작 체력으로 초기화
        health = startingHealth;
    }
```

```
    // 대미지 처리
    // 호스트에서 먼저 단독 실행되고, 호스트를 통해 다른 클라이언트에서 일괄 실행됨
    [PunRPC]
    public virtual void OnDamage(float damage, Vector3 hitPoint, Vector3 hitNormal) {
        if (PhotonNetwork.IsMasterClient)
        {
            // 대미지만큼 체력 감소
            health -= damage;

            // 호스트에서 클라이언트로 동기화
            photonView.RPC("ApplyUpdatedHealth", RpcTarget.Others, health, dead);

            // 다른 클라이언트도 OnDamage를 실행하도록 함
            photonView.RPC("OnDamage", RpcTarget.Others, damage, hitPoint, hitNormal);
        }

        // 체력이 0 이하 && 아직 죽지 않았다면 사망 처리 실행
        if (health <= 0 && !dead)
        {
            Die();
        }
    }
```

네트워크 Player, Follow Cam

- LivingEntity

```
// 체력을 회복하는 기능
[PunRPC]
public virtual void RestoreHealth(float newHealth) {
    if (dead)
    {
        // 이미 사망한 경우 체력을 회복할 수 없음
        return;
    }

    // 호스트만 체력을 직접 갱신 가능
    if (PhotonNetwork.IsMasterClient)
    {
        // 체력 추가
        health += newHealth;
        // 서버에서 클라이언트로 동기화
        photonView.RPC("ApplyUpdatedHealth", RpcTarget.Others, health, dead);

        // 다른 클라이언트도 RestoreHealth를 실행하도록 함
        photonView.RPC("RestoreHealth", RpcTarget.Others, newHealth);
    }
}
```

```
    public virtual void Die() {
        // onDeath 이벤트에 등록된 메서드가 있다면 실행
        if (onDeath != null)
        {
            onDeath();
        }

        // 사망 상태를 참으로 변경
        dead = true;
    }
}
```

네트워크 Player, Follow Cam

- PlayerHealth

```
using Photon.Pun;
using UnityEngine;
using UnityEngine.UI; // UI 관련 코드

// 플레이어 캐릭터의 생명체로서의 동작 담당
public class PlayerHealth : LivingEntity {
    // 변수 선언부 생략(기존 코드와 동일함)

    private void Awake() {
        // 코드 생략(기존 코드와 동일함)
    }

    protected override void OnEnable() {
        // 코드 생략(기존 코드와 동일함)
    }

    // 체력 회복
    [PunRPC]
    public override void RestoreHealth(float newHealth) {
        // 코드 생략(기존 코드와 동일함)
    }
```

```
    // 대미지 처리
    [PunRPC]
    public override void OnDamage(float damage, Vector3 hitPoint,
        Vector3 hitDirection) {
        // 코드 생략(기존 코드와 동일함)
    }

    public override void Die() {
        // LivingEntity의 Die() 실행(사망 적용)
        base.Die();

        // 체력 슬라이더 비활성화
        healthSlider.gameObject.SetActive(false);

        // 사망음 재생
        playerAudioPlayer.PlayOneShot(deathClip);

        // 애니메이터의 Die 트리거를 발동시켜 사망 애니메이션 재생
        playerAnimator.SetTrigger("Die");

        // 플레이어 조작을 받는 컴포넌트 비활성화
        playerMovement.enabled = false;
        playerShooter.enabled = false;

        // 5초 뒤에 리스폰
        Invoke("Respawn", 5f);
    }
```

네트워크 Player, Follow Cam

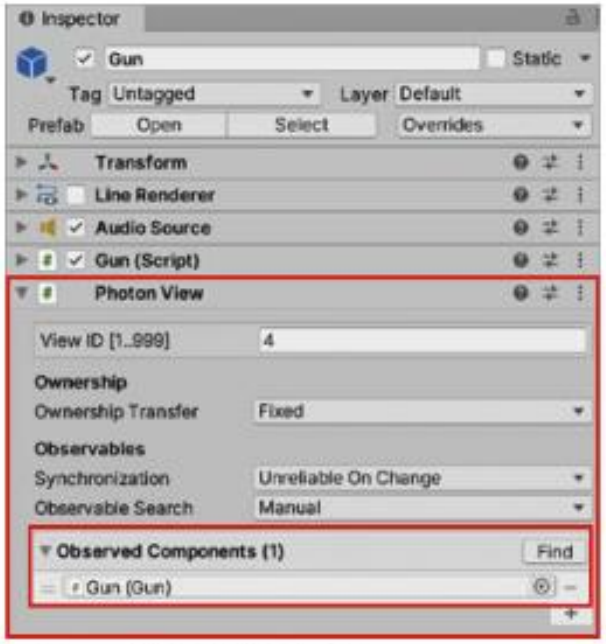
- PlayerHealth

```
private void OnTriggerEnter(Collider other) {
    // 아이템과 충돌한 경우 해당 아이템을 사용하는 처리
    // 사망하지 않은 경우에만 아이템 사용 가능
    if (!dead)
    {
        // 충돌한 상대방으로부터 Item 컴포넌트 가져오기 시도
        IItem item = other.GetComponent<IItem>();

        // 충돌한 상대방으로부터 Item 컴포넌트 가져오는 데 성공했다면
        if (item != null)
        {
            // 호스트만 아이템 직접 사용 가능
            // 호스트에서는 아이템 사용 후 사용된 아이템의 효과를 모든 클라이언트에 동기화시킴
            if (PhotonNetwork.IsMasterClient)
            {
                // Use 메서드를 실행하여 아이템 사용
                item.Use(gameObject);
            }

            // 아이템 습득 소리 재생
            playerAudioPlayer.PlayOneShot(itemPickupClip);
        }
    }
}
```

```
    // 부활 처리
    public void Respawn() {
        // 로컬 플레이어만 직접 위치 변경 가능
        if (photonView.IsMine)
        {
            // 원점에서 반경 5유닛 내부의 랜덤 위치 지정
            Vector3 randomSpawnPos = Random.insideUnitSphere * 5f;
            // 랜덤 위치의 y 값을 0으로 변경
            randomSpawnPos.y = 0f;

            // 지정된 랜덤 위치로 이동
            transform.position = randomSpawnPos;
        }

        // 컴포넌트를 리셋하기 위해 게임 오브젝트를 잠시 껐다가 다시 켜기
        // 컴포넌트의 OnDisable(), OnEnable() 메서드가 실행됨
        gameObject.SetActive(false);
        gameObject.SetActive(true);
    }
}
```

네트워크 Player, Follow Cam

- Gun



▶ Gun 게임 오브젝트의 모습

네트워크 Player, Follow Cam

- Gun

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

// 총을 구현
public class Gun : MonoBehaviourPun, IPunObservable {
    // 변수 선언부 생략(기존 코드와 동일함)

    // 주기적으로 자동 실행되는 동기화 메서드
    public void OnPhotonSerializeView(PhotonStream stream, PhotonMessageInfo info) {
        // 로컬 오브젝트라면 쓰기 부분이 실행됨
        if (stream.IsWriting)
        {
            // 남은 탄알 수를 네트워크를 통해 보내기
            stream.SendNext(ammoRemain);
            // 탄창의 탄알 수를 네트워크를 통해 보내기
            stream.SendNext(magAmmo);
            // 현재 총의 상태를 네트워크를 통해 보내기
            stream.SendNext(state);
        }
        else
        {
            // 리모트 오브젝트라면 읽기 부분이 실행됨
            // 남은 탄알 수를 네트워크를 통해 받기
            ammoRemain = (int) stream.ReceiveNext();
            // 탄창의 탄알 수를 네트워크를 통해 받기
            magAmmo = (int) stream.ReceiveNext();
            // 현재 총의 상태를 네트워크를 통해 받기
            state = (State) stream.ReceiveNext();
        }
    }

    // 남은 탄알을 추가하는 메서드
    [PunRPC]
    public void AddAmmo(int ammo) {
        ammoRemain += ammo;
    }

    private void Awake() {
        // 코드 생략(기존 코드와 동일함)
    }
```

네트워크 Player, Follow Cam

- Gun

```
private void OnEnable() {
    // 코드 생략(기존 코드와 동일함)
}

// 발사 시도
public void Fire() {
    // 코드 생략(기존 코드와 동일함)
}

private void Shot() {
    // 실제 발사 처리는 호스트에 대리
    photonView.RPC("ShotProcessOnServer", RpcTarget.MasterClient);

    // 남은 탄환 수를 -1
    magAmmo--;
    if (magAmmo <= 0)
    {
        // 탄창에 남은 탄알이 없다면 총의 현재 상태를 Empty로 갱신
        state = State.Empty;
    }
}
```

```
// 호스트에서 실행되는 실제 발사 처리
[PunRPC]
private void ShotProcessOnServer() {
    // 레이캐스트에 의한 충돌 정보를 저장하는 컨테이너
    RaycastHit hit;
    // 탄알이 맞은 곳을 저장할 변수
    Vector3 hitPosition = Vector3.zero;

    // 레이캐스트(시작 지점, 방향, 충돌 정보 컨테이너, 사정거리)
    if (Physics.Raycast(fireTransform.position,
        fireTransform.forward, out hit, fireDistance))
    {
        // 레이가 어떤 물체와 충돌한 경우

        // 충돌한 상대방으로부터 IDamageable 오브젝트 가져오기 시도
        IDamageable target =
            hit.collider.GetComponent<IDamageable>();

        // 상대방으로부터 IDamageable 오브젝트를 가져오는 데 성공했다면
        if (target != null)
        {
            // 상대방의 OnDamage 함수를 실행시켜 상대방에게 대미지 주기
            target.OnDamage(damage, hit.point, hit.normal);
        }

        // 레이가 충돌한 위치 저장
        hitPosition = hit.point;
    }
```

네트워크 Player, Follow Cam

- Gun

```
    else
    {
        // 레이가 다른 물체와 충돌하지 않았다면
        // 탄알이 최대 사정거리까지 날아갔을 때의 위치를 충돌 위치로 사용
        hitPosition = fireTransform.position +
                      fireTransform.forward * fireDistance;
    }

    // 발사 이펙트 재생. 이펙트 재생은 모든 클라이언트에서 실행
    photonView.RPC("ShotEffectProcessOnClients", RpcTarget.All, hitPosition);
}

// 이펙트 재생 코루틴을 랩핑하는 메서드
[PunRPC]
private void ShotEffectProcessOnClients(Vector3 hitPosition) {
    StartCoroutine(ShotEffect(hitPosition));
}

// 발사 이펙트와 소리를 재생하고 탄알 궤적을 그림
private IEnumerator ShotEffect(Vector3 hitPosition) {
        // 코드 생략(기존 코드와 동일함)
    }

    // 재장전 시도
    public bool Reload() {
        // 코드 생략(기존 코드와 동일함)
    }

    // 실제 재장전 처리를 진행
    private IEnumerator ReloadRoutine() {
        // 코드 생략(기존 코드와 동일함)
    }
}
```

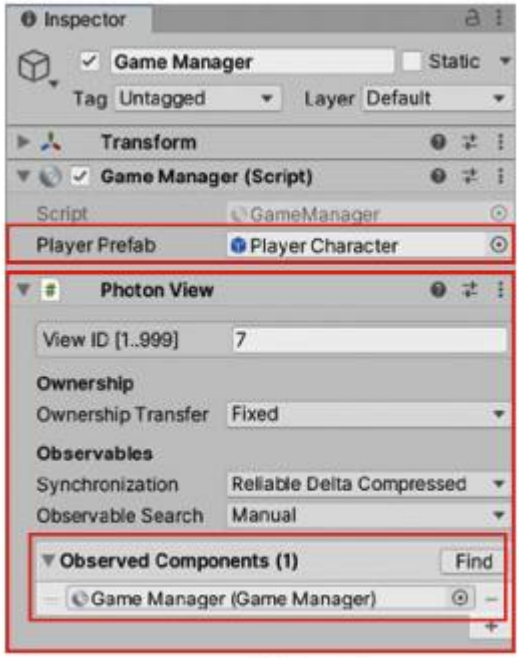
네트워크 Player, Follow Cam

- Gun

1. 클라이언트 B의 로컬 플레이어 b의 총에서 Shot() 메서드 실행
2. Shot()에서 photonView.RPC("ShotProcessOnServer", RpcTarget.MasterClient); 실행
3. 실제 사격 처리를 하는 ShotProcessOnServer()는 호스트 클라이언트 A에서만 실행
4. ShotProcessOnServer()에서 photonView.RPC("ShotEffectProcessOnClients", RpcTarget.All, hitPosition); 실행
5. 사격 효과 재생인 ShotEffectProcessOnClients()는 모든 클라이언트 A, B, C에서 실행됨

네트워크 Player, Follow Cam

- **GameManager**
  네트워크 플레이어 캐릭터 생성, 점수 동기화, 룸 나가기 구현



▶ Game Manager 게임 오브젝트

네트워크 Player, Follow Cam

- GameManager

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

// 점수와 게임오버 여부, 게임 UI를 관리하는 게임 매니저
public class GameManager : MonoBehaviourPunCallbacks, IPunObservable {
    // 외부에서 싱글턴 오브젝트를 가져올 때 사용할 프로퍼티
    public static GameManager instance
    {
        // 코드 생략(기존 코드와 동일함)
    }

    private static GameManager m_instance; // 싱글턴이 할당될 static 변수

    public GameObject playerPrefab; // 생성할 플레이어 캐릭터 프리팹

    private int score = 0; // 현재 게임 점수
    public bool isGameover { get; private set; } // 게임오버 상태
```

```
// 주기적으로 자동 실행되는 동기화 메서드
public void OnPhotonSerializeView(PhotonStream stream, PhotonMessageInfo info) {
    // 로컬 오브젝트라면 쓰기 부분이 실행됨
    if (stream.IsWriting)
    {
        // 네트워크를 통해 score 값 보내기
        stream.SendNext(score);
    }
    else
    {
            // 리모트 오브젝트라면 읽기 부분이 실행됨

            // 네트워크를 통해 score 값 받기
            score = (int) stream.ReceiveNext();
            // 동기화하여 받은 점수를 UI로 표시
            UIManager.instance.UpdateScoreText(score);
      }
    }


    private void Awake() {
        // 코드 생략(기존 코드와 동일함)
    }
```

네트워크 Player, Follow Cam

- GameManager

```
// 게임 시작과 동시에 플레이어가 될 게임 오브젝트 생성
private void Start() {
    // 생성할 랜덤 위치 지정
    Vector3 randomSpawnPos = Random.insideUnitSphere * 5f;
    // 위치 y 값은 0으로 변경
    randomSpawnPos.y = 0f;

    // 네트워크상의 모든 클라이언트에서 생성 실행
    // 해당 게임 오브젝트의 주도권은 생성 메서드를 직접 실행한 클라이언트에 있음
    PhotonNetwork.Instantiate(playerPrefab.name, randomSpawnPos, Quaternion.identity);
}

// 점수를 추가하고 UI 갱신
public void AddScore(int newScore) {
    // 게임오버가 아닌 상태에서만 점수 추가 가능
    if (!isGameover)
    {
        // 점수 추가
        score += newScore;
        // 점수 UI 텍스트 갱신
        UIManager.instance.UpdateScoreText(score);
    }
}
```

```
    // 게임오버 처리
    public void EndGame() {
        // 코드 생략(기존 코드와 동일함)
    }

    // 키보드 입력을 감지하고 룸을 나가게 함
    private void Update() {
        if (Input.GetKeyDown(KeyCode.Escape))
        {
            PhotonNetwork.LeaveRoom();
        }
    }

    // 룸을 나갈 때 자동 실행되는 메서드
    public override void OnLeftRoom() {
        // 룸을 나가면 로비 씬으로 돌아감
        SceneManager.LoadScene("Lobby");
    }
}
```

네트워크 Player, Follow Cam

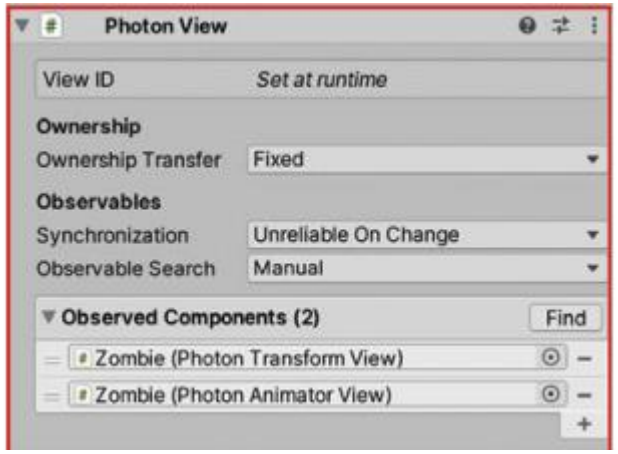
- GameManager

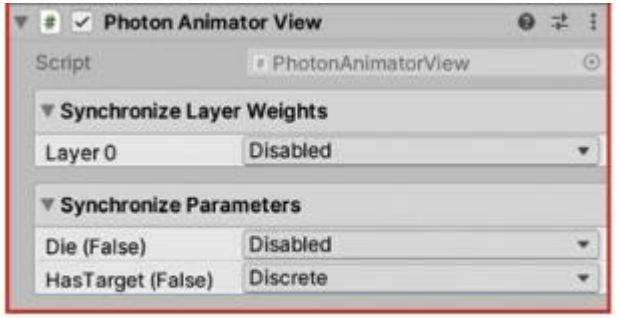
1. 클라이언트 A가 룸을 생성하고 접속
2. A에서 GameManager의 Start() 실행
3. A가 PhotonNetwork.Instantiate()에 의해 플레이어 캐릭터 a를 A에 생성
4. 클라이언트 B가 룸에 접속 → 클라이언트 B에 자동으로 a가 생성됨
5. B에서 GameManager의 Start() 실행
6. B가 PhotonNetwork.Instantiate()에 의해 플레이어 캐릭터 b를 A와 B에 생성

네트워크 Zombie, Item

네트워크 Zombie, Item

- **Zombie**
  호스트에서만 경로 계산, 추적, 공격을 실행 (RPC 이용 동기화)
  위치, 회전, 애니메이션 동기화





▶ Zombie 프리팹

네트워크 Zombie, Item

- Zombie

```
using System.Collections;
using Photon.Pun;
using UnityEngine;
using UnityEngine.AI; // AI, 내비게이션 시스템 관련 코드 가져오기

// 좀비 AI 구현
public class Zombie : LivingEntity {
    // 변수 선언부 생략(기존 코드와 동일함)

    // 추적할 대상이 존재하는지 알려주는 프로퍼티
    private bool hasTarget
    {
        // 코드 생략(기존 코드와 동일함)
    }

    private void Awake() {
        // 코드 생략(기존 코드와 동일함)
    }
```

```
// 좀비 AI의 초기 스펙을 결정하는 셋업 메서드
[PunRPC]
public void Setup(float newHealth, float newDamage,
    float newSpeed, Color skinColor) {
    // 체력 설정
    startingHealth = newHealth;
    health = newHealth;
    // 공격력 설정
    damage = newDamage;
    // 내비메시 에이전트의 이동 속도 설정
    navMeshAgent.speed = newSpeed;
    // 렌더러가 사용 중인 머테리얼의 컬러를 변경, 외형색이 변함
    zombieRenderer.material.color = skinColor;
}

private void Start() {
    // 호스트가 아니라면 AI의 추적 루틴을 실행하지 않음
    if (!PhotonNetwork.IsMasterClient)
    {
        return;
    }

    // 게임 오브젝트 활성화와 동시에 AI의 추적 루틴 시작
    StartCoroutine(UpdatePath());
}
```

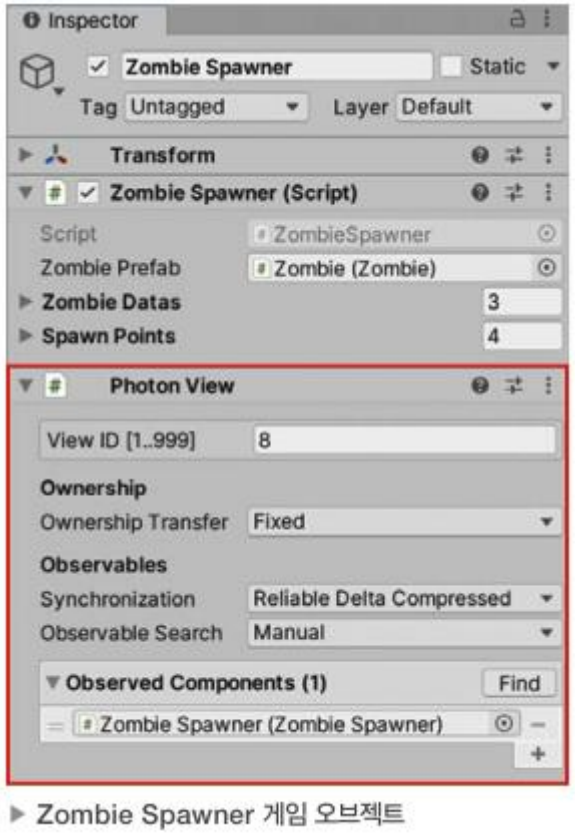
네트워크 Zombie, Item

- Zombie

```
private void Update() {
    // 호스트가 아니라면 애니메이션의 파라미터를 직접 갱신하지 않음
    // 호스트가 파라미터를 갱신하면 클라이언트에 자동으로 전달되기 때문
    if (!PhotonNetwork.IsMasterClient)
    {
        return;
    }

    // 추적 대상의 존재 여부에 따라 다른 애니메이션 재생
    zombieAnimator.SetBool("HasTarget", hasTarget);
}

// 주기적으로 추적할 대상의 위치를 찾아 경로 갱신
private IEnumerator UpdatePath() {
    // 코드 생략(기존 코드와 동일함)
}


// 대미지를 입었을 때 실행할 처리
[PunRPC]
public override void OnDamage(float damage, Vector3 hitPoint, Vector3 hitNormal) {
    // 코드 생략(기존 코드와 동일함)
}

// 사망 처리
public override void Die() {
    // 코드 생략(기존 코드와 동일함)
}

private void OnTriggerStay(Collider other) {
    // 호스트가 아니라면 공격 실행 불가
    if (!PhotonNetwork.IsMasterClient)
    {
        return;
    }

    // 자신이 사망하지 않았으며,
    // 최근 공격 시점에서 timeBetAttack 이상 시간이 지났다면 공격 가능
    if (!dead && Time.time >= lastAttackTime + timeBetAttack)
    {
        // 코드 생략(기존 코드와 동일함)
    }
}
}
```

네트워크 Zombie, Item

- **ZombieSpawner**
  네트워크상에서 좀비 생성, 남은 좀비 수 동기화



네트워크 Zombie, Item

- ZombieSpawner

```
using System.Collections;
using System.Collections.Generic;
using ExitGames.Client.Photon;
using Photon.Pun;
using UnityEngine;

// 좀비 게임 오브젝트를 주기적으로 생성
public class ZombieSpawner : MonoBehaviourPun, IPunObservable {
    public Zombie zombiePrefab; // 생성할 좀비 원본 프리팹

    public ZombieData[] zombieDatas; // 사용할 좀비 셋업 데이터
    public Transform[] spawnPoints; // 좀비 AI를 소환할 위치

    private List<Zombie> zombies = new List<Zombie>(); // 생성된 좀비를 담는 리스트

    private int zombieCount = 0; // 남은 좀비 수
    private int wave; // 현재 웨이브
```

```
// 주기적으로 자동 실행되는 동기화 메서드
public void OnPhotonSerializeView(PhotonStream stream, PhotonMessageInfo info) {
    // 로컬 오브젝트라면 쓰기 부분이 실행됨
    if (stream.IsWriting)
    {
        // 남은 좀비 수를 네트워크를 통해 보내기
        stream.SendNext(zombies.Count);
        // 현재 웨이브를 네트워크를 통해 보내기
        stream.SendNext(wave);
    }
    else
    {
        // 리모트 오브젝트라면 읽기 부분이 실행됨
        // 남은 좀비 수를 네트워크를 통해 받기
        zombieCount = (int) stream.ReceiveNext();
        // 현재 웨이브를 네트워크를 통해 받기
        wave = (int) stream.ReceiveNext();
    }
}

private void Awake() {
```

네트워크 Zombie, Item

- ZombieSpawner

```
    PhotonPeer.RegisterType(typeof(Color), 128, ColorSerialization.SerializeColor,
        ColorSerialization.DeserializeColor);
}

private void Update() {
    // 호스트만 좀비를 직접 생성할 수 있음
    // 다른 클라이언트는 호스트가 생성한 좀비를 동기화를 통해 받아옴
    if (PhotonNetwork.IsMasterClient)
    {
        // 게임오버 상태일 때는 생성하지 않음
        if (GameManager.instance != null && GameManager.instance.isGameover)
        {
            return;
        }

        // 좀비를 모두 물리친 경우 다음 스폰 실행
        if (zombies.Count <= 0)
        {
            SpawnWave();
        }
    }
```

```
    // UI 갱신
    UpdateUI();
}

// 웨이브 정보를 UI로 표시
private void UpdateUI() {
    if (PhotonNetwork.IsMasterClient)
    {
        // 호스트는 직접 갱신한 좀비 리스트를 이용해 남은 좀비 수 표시
        UIManager.instance.UpdateWaveText(wave, zombies.Count);
    }
    else
    {
        // 클라이언트는 좀비 리스트를 갱신할 수 없으므로
        // 호스트가 보내준 zombieCount를 이용해 좀비 수 표시
        UIManager.instance.UpdateWaveText(wave, zombieCount);
    }
}
```

네트워크 Zombie, Item

- ZombieSpawner

```
// 현재 웨이브에 맞춰 좀비 생성
private void SpawnWave() {
    // 코드 생략(기존 코드와 동일함)
}

// 좀비 생성
private void CreateZombie() {
    // 사용할 좀비 데이터 랜덤으로 결정
    ZombieData zombieData = zombieDatas[Random.Range(0, zombieDatas.Length)];

    // 생성할 위치를 랜덤으로 결정
    Transform spawnPoint = spawnPoints[Random.Range(0, spawnPoints.Length)];

    // 좀비 프리팹으로부터 좀비 생성. 네트워크상의 모든 클라이언트에 생성됨
    GameObject createdZombie = PhotonNetwork.Instantiate(zombiePrefab.gameObject.name,
        spawnPoint.position,
        spawnPoint.rotation);

    // 생성한 좀비를 셋업하기 위해 Zombie 컴포넌트를 가져옴
    Zombie zombie = createdZombie.GetComponent<Zombie>();
    // 생성한 좀비의 능력치 설정
    zombie.photonView.RPC("Setup", RpcTarget.All, zombieData.health, zombieData.damage,
        zombieData.speed, zombieData.skinColor);

    // 생성된 좀비를 리스트에 추가
    zombies.Add(zombie);

    // 좀비의 onDeath 이벤트에 익명 메서드 등록
    // 사망한 좀비를 리스트에서 제거
    zombie.onDeath += () => zombies.Remove(zombie);
    // 사망한 좀비를 10초 뒤에 파괴
    zombie.onDeath += () => StartCoroutine(DestroyAfter(zombie.gameObject, 10f));
    // 좀비 사망 시 점수 상승
    zombie.onDeath += () => GameManager.instance.AddScore(100);
}

// 포톤의 Network.Destroy()는 지연 파괴를 지원하지 않으므로 지연 파괴를 직접 구현함
IEnumerator DestroyAfter(GameObject target, float delay) {
    // delay만큼 쉬고
    yield return new WaitForSeconds(delay);
```

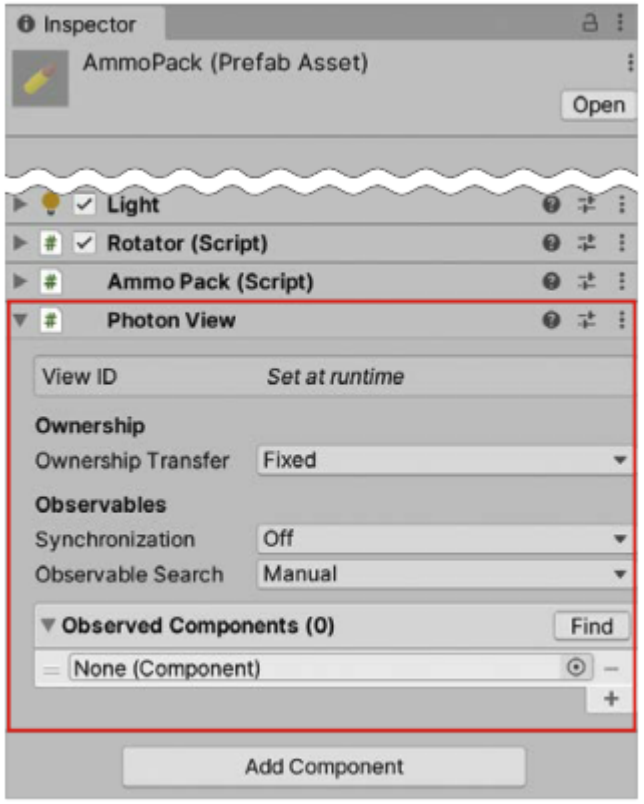
네트워크 Zombie, Item

- ZombieSpawner

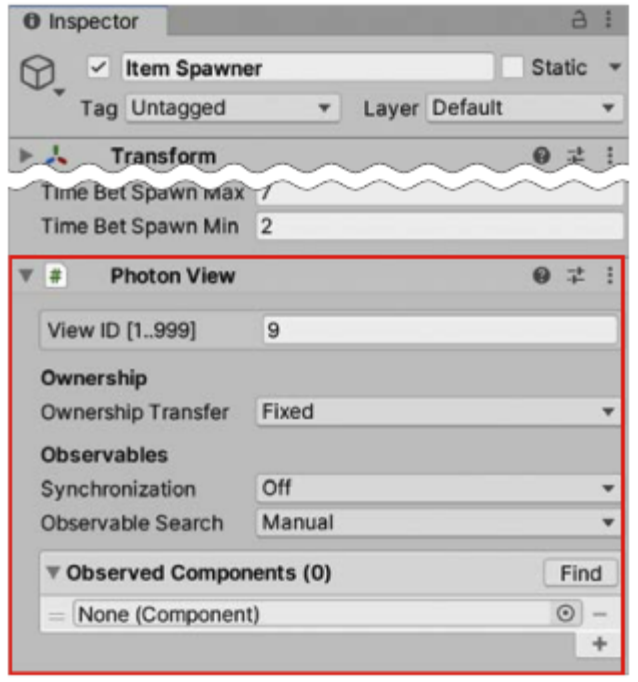
```
        // target이 아직 파괴되지 않았다면
        if (target != null)
        {
            // target을 모든 네트워크상에서 파괴
            PhotonNetwork.Destroy(target);
        }
    }
}
```

네트워크 Zombie, Item

- **AmmoPack, HealthPack, Coin**
  모든 클라이언트에서 실행
- **ItemSpawner**
  맵 중심에 아이템 생성(호스트에서)
  모든 클라이언트에서 동기화



▶ AmmoPack 프리팹



▶ Item Spawner 게임 오브젝트

네트워크 Zombie, Item

- AmmoPack

```
using Photon.Pun;
using UnityEngine;

// 탄알을 충전하는 아이템
public class AmmoPack : MonoBehaviourPun, IItem {
    public int ammo = 30; // 충전할 탄알 수

    public void Use(GameObject target) {
        // 전달받은 게임 오브젝트로부터 PlayerShooter 컴포넌트 가져오기 시도
        PlayerShooter playerShooter = target.GetComponent<PlayerShooter>();

        // PlayerShooter 컴포넌트가 있고 총 오브젝트가 존재하면
        if (playerShooter != null && playerShooter.gun != null)
        {
            // 총의 남은 탄환 수를 ammo만큼 더하기. 모든 클라이언트에서 실행
            playerShooter.gun.photonView.RPC("AddAmmo", RpcTarget.All, ammo);
        }

        // 모든 클라이언트에서 자신 파괴
        PhotonNetwork.Destroy(gameObject);
    }
}
```

네트워크 Zombie, Item

- HealthPack

```
using Photon.Pun;
using UnityEngine;

// 체력을 회복하는 아이템
public class HealthPack : MonoBehaviourPun, IItem {
    public float health = 50; // 체력을 회복할 수치

    public void Use(GameObject target) {
        // 전달받은 게임 오브젝트로부터 LivingEntity 컴포넌트 가져오기 시도
        LivingEntity life = target.GetComponent<LivingEntity>();

        // LivingEntity컴포넌트가 있다면
        if (life != null)
        {
            // 체력 회복 실행
            life.RestoreHealth(health);
        }

        // 모든 클라이언트에서 자신 파괴
        PhotonNetwork.Destroy(gameObject);
    }
}
```

네트워크 Zombie, Item

- Coin

```
using Photon.Pun;
using UnityEngine;

// 게임 점수를 증가시키는 아이템
public class Coin : MonoBehaviourPun, IItem {
    public int score = 200; // 증가할 점수

    public void Use(GameObject target) {
        // 게임 매니저로 접근해 점수 추가
        GameManager.instance.AddScore(score);
        // 모든 클라이언트에서 자신 파괴
        PhotonNetwork.Destroy(gameObject);
    }
}
```

네트워크 Zombie, Item

- ItemSpawner

```
using System.Collections;
using Photon.Pun;
using UnityEngine;
using UnityEngine.AI; // 내비메시 관련 코드

// 주기적으로 아이템을 플레이어 근처에 생성하는 스크립트
public class ItemSpawner : MonoBehaviourPun {
    public GameObject[] items; // 생성할 아이템

    public float maxDistance = 5f; // 플레이어 위치에서 아이템이 배치될 최대 반경

    public float timeBetSpawnMax = 7f; // 최대 시간 간격
    public float timeBetSpawnMin = 2f; // 최소 시간 간격

    private float timeBetSpawn; // 생성 간격
    private float lastSpawnTime; // 마지막 생성 시점

    private void Start() {
        // 코드 생략(기존 코드와 동일함)
    }
```

```
// 주기적으로 아이템 생성 처리 실행
private void Update() {
    // 호스트에서만 아이템 직접 생성 가능
    if (!PhotonNetwork.IsMasterClient)
    {
        return;
    }

    if (Time.time >= lastSpawnTime + timeBetSpawn)
    {
        // 마지막 생성 시간 갱신
        lastSpawnTime = Time.time;
        // 생성 주기를 랜덤으로 변경
        timeBetSpawn = Random.Range(timeBetSpawnMin, timeBetSpawnMax);
        // 실제 아이템 생성
        Spawn();
```

네트워크 Zombie, Item

- ItemSpawner

```
    }
}

// 실제 아이템 생성 처리
private void Spawn() {
    // (0, 0, 0)을 기준으로 maxDistance 안에서 내비메시 위의 랜덤 위치 지정
    Vector3 spawnPosition = GetRandomPointOnNavMesh(Vector3.zero, maxDistance);
    // 바닥에서 0.5만큼 위로 올리기
    spawnPosition += Vector3.up * 0.5f;

    // 생성할 아이템을 무작위로 하나 선택
    GameObject selectedItem = items[Random.Range(0, items.Length)];

    // 네트워크의 모든 클라이언트에서 해당 아이템 생성
    GameObject item = PhotonNetwork.Instantiate(selectedItem.name, spawnPosition,
        Quaternion.identity);

    // 생성한 아이템을 5초 뒤에 파괴
    StartCoroutine(DestroyAfter(item, 5f));
}
```

```
    // 포톤의 PhotonNetwork.Destroy()를 지연 실행하는 코루틴
    IEnumerator DestroyAfter(GameObject target, float delay) {
        // delay만큼 대기
        yield return new WaitForSeconds(delay);

        // target이 파괴되지 않았으면 파괴 실행
        if (target != null)
        {
            PhotonNetwork.Destroy(target);
        }
    }


    // 네브메시 위의 랜덤한 위치를 반환하는 메서드
    // center를 중심으로 distance 반경 안에서 랜덤한 위치를 찾음
    private Vector3 GetRandomPointOnNavMesh(Vector3 center, float distance) {
        // 코드 생략(기존 코드와 동일함)
    }
}
```

네트워크 Zombie, Item

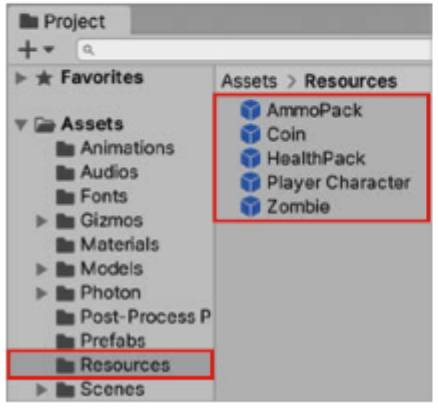
[과정 01] Resources 폴더로 프리팹 옮기기

① 프로젝트에 **Resources** 폴더 생성

② 프로젝트의 **Prefabs** 폴더에서 다음 프리팹 선택

- **AmmoPack**
- **Coin**
- **HealthPack**
- **Player Character**
- **Zombie**

③ 선택한 프리팹을 **Resource** 폴더로 옮김



▶ Resources 폴더로 프리팹 옮기기