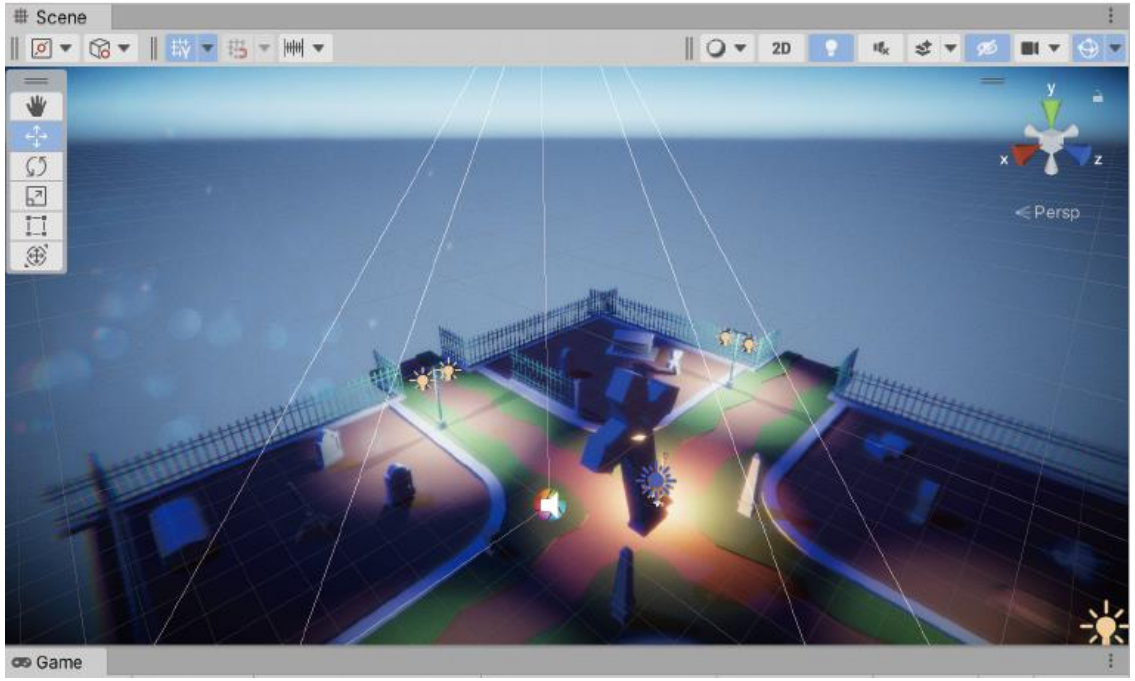
# 좀비 서바이버 :

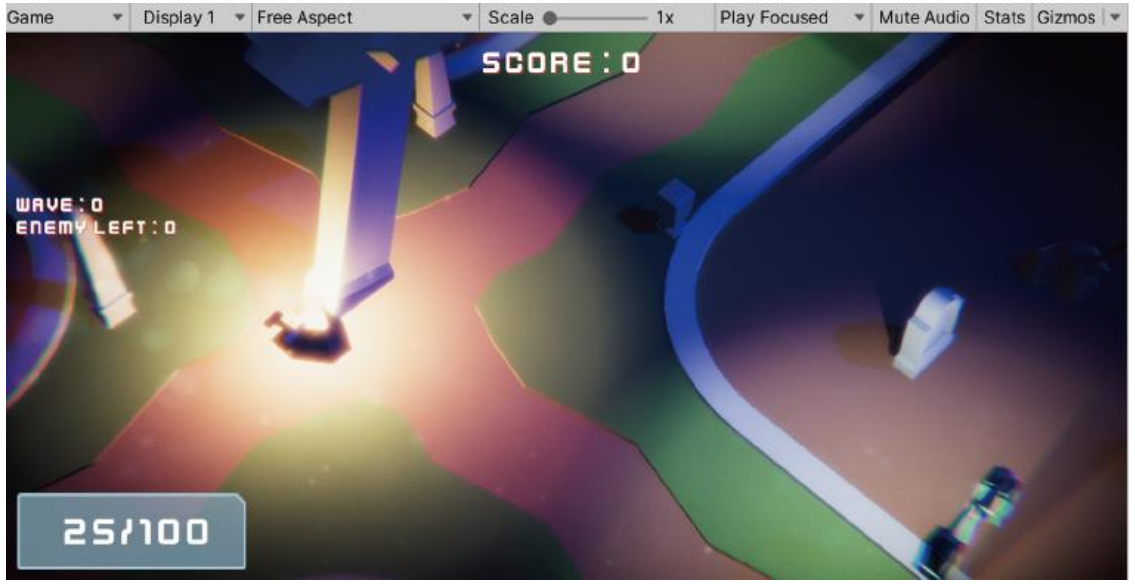
## 싱글플레이어를 멀티플레이어로 포팅

# 네트워크 플레이어 캐릭터 준비(1)

## [과정 01] Main 씬 준비하기

### ①프로젝트의 Scenes 폴더에 있는 Main 씬 열기



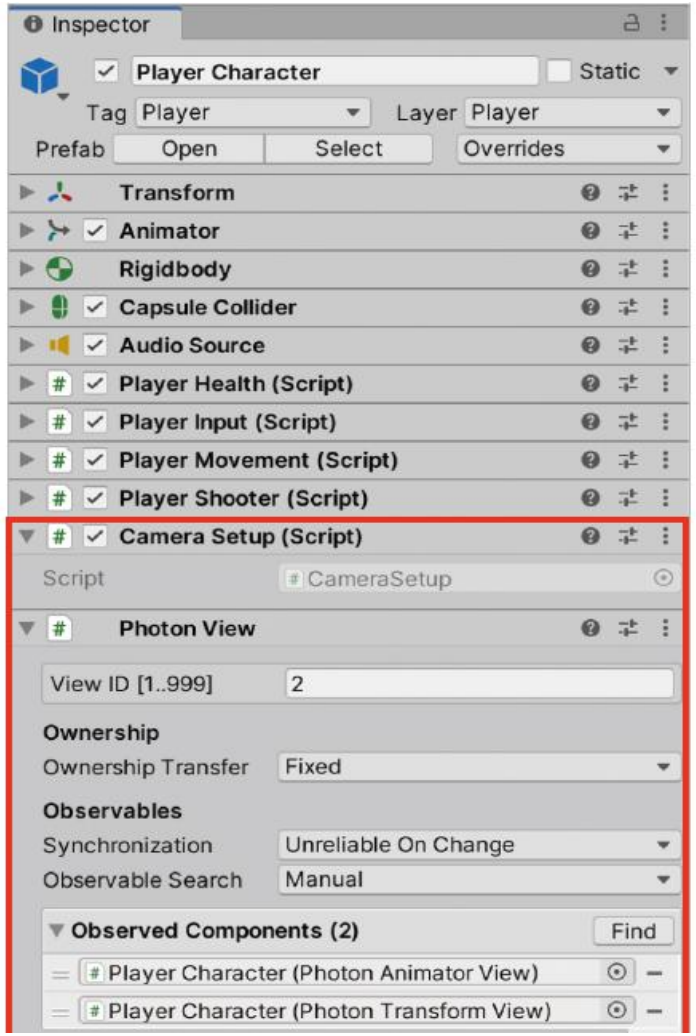

▶ 열린 Main 씬

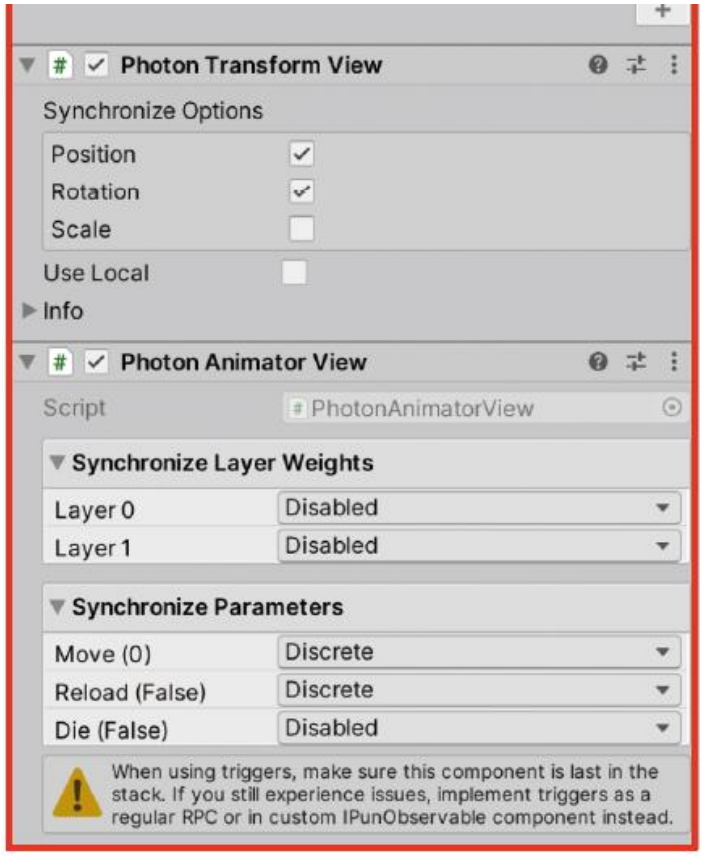
# 네트워크 플레이어 캐릭터 준비(2)

- Prefabs 폴더에서 Player Character 프리팹을 찾아 Main 씬에 Player Character를 생성

**[과정 01] Player Character 게임 오브젝트 준비**

**① Prefabs 폴더의 Player Character 프리팹을 하이어라키 창으로 드래그&드롭**



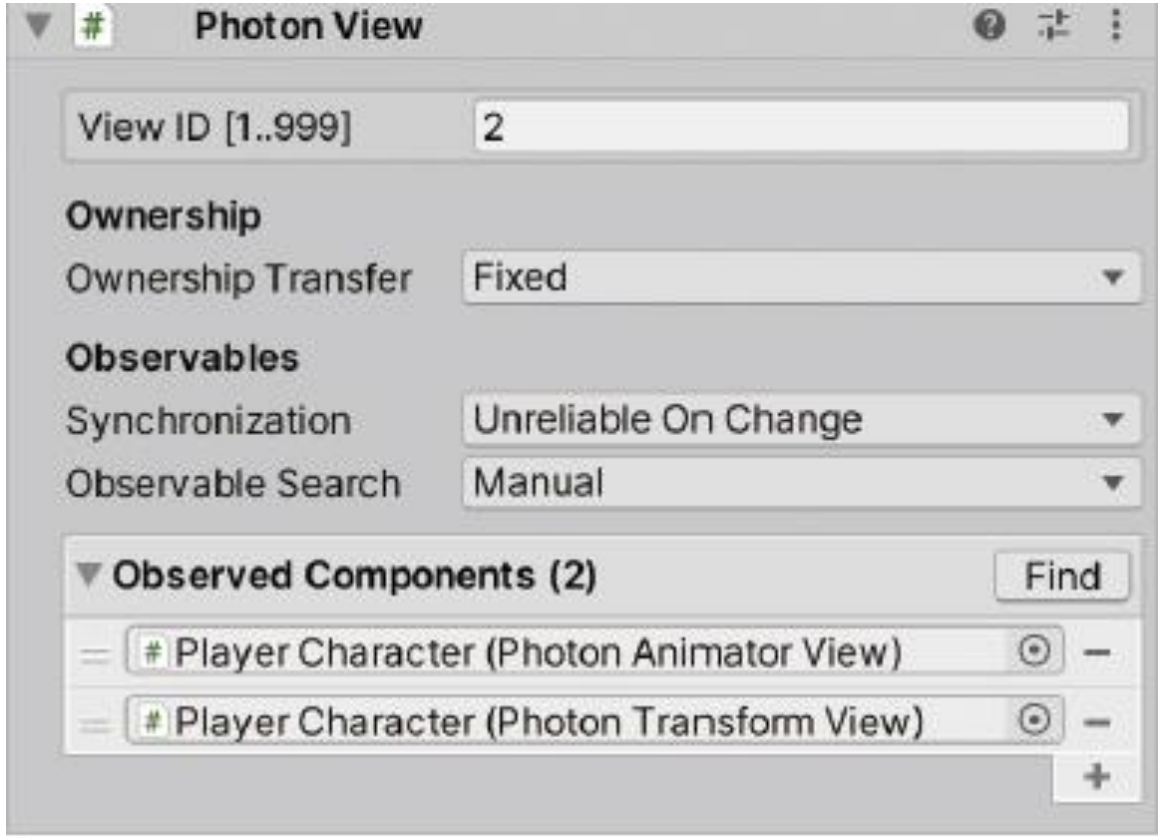




▶ 새로 추가된 컴포넌트

# 네트워크 플레이어 캐릭터 준비(3)

## Photon View 컴포넌트

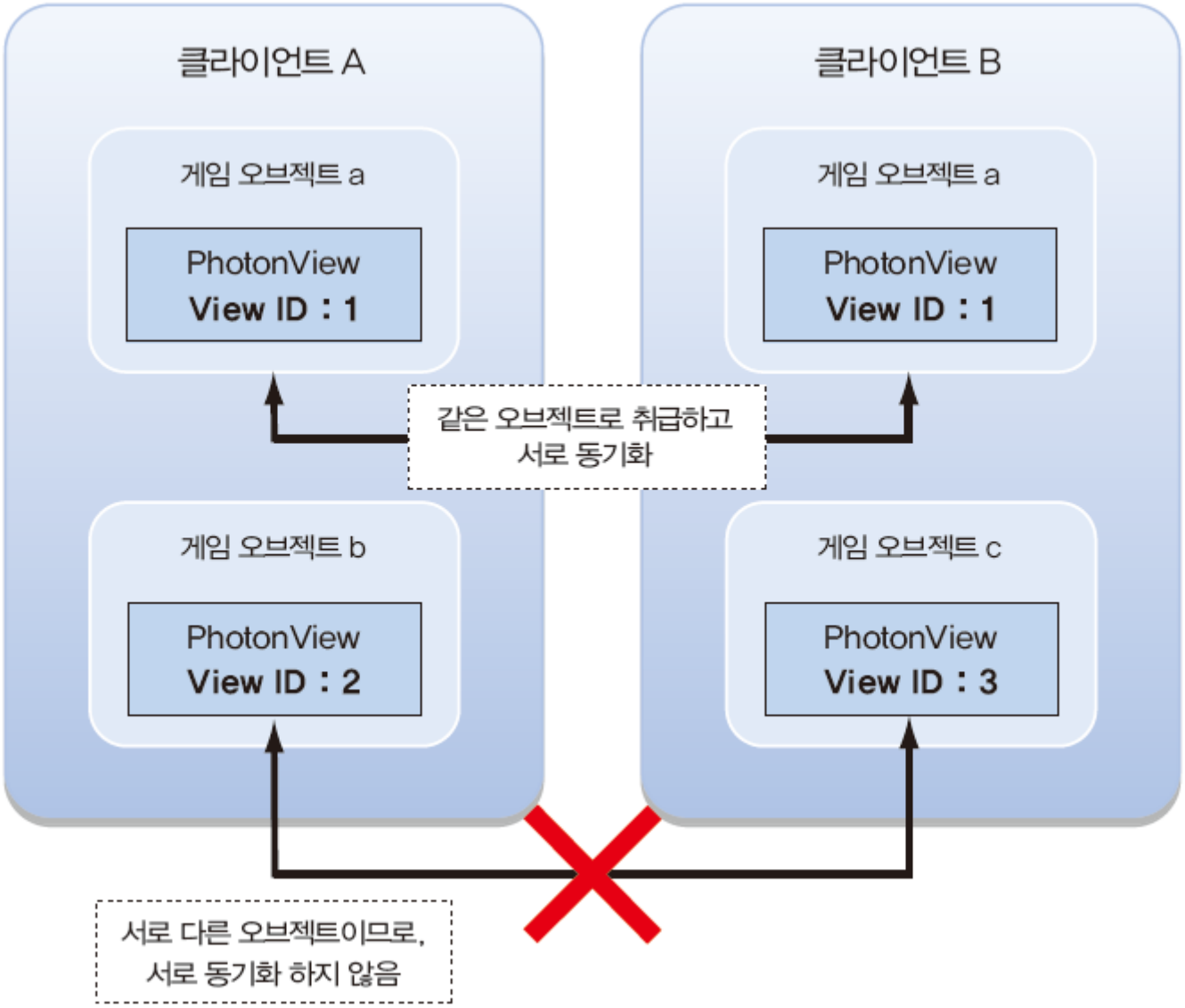
- 네트워크를 통해 동기화될 모든 게임 오브젝트는 Photon View 컴포넌트를 가져야 함
- Photon View 컴포넌트는 게임 오브젝트에 네트워크상에서 구별 가능한 식별자인 View ID를 부여
- 또한 Observed Components 리스트에 등록된 컴포넌트들의 변화한 수치를 관측하고, 네트워크를 넘어서 다른 클라이언트에 전달




▶ Photon View 컴포넌트

# 네트워크 플레이어 캐릭터 준비(4)

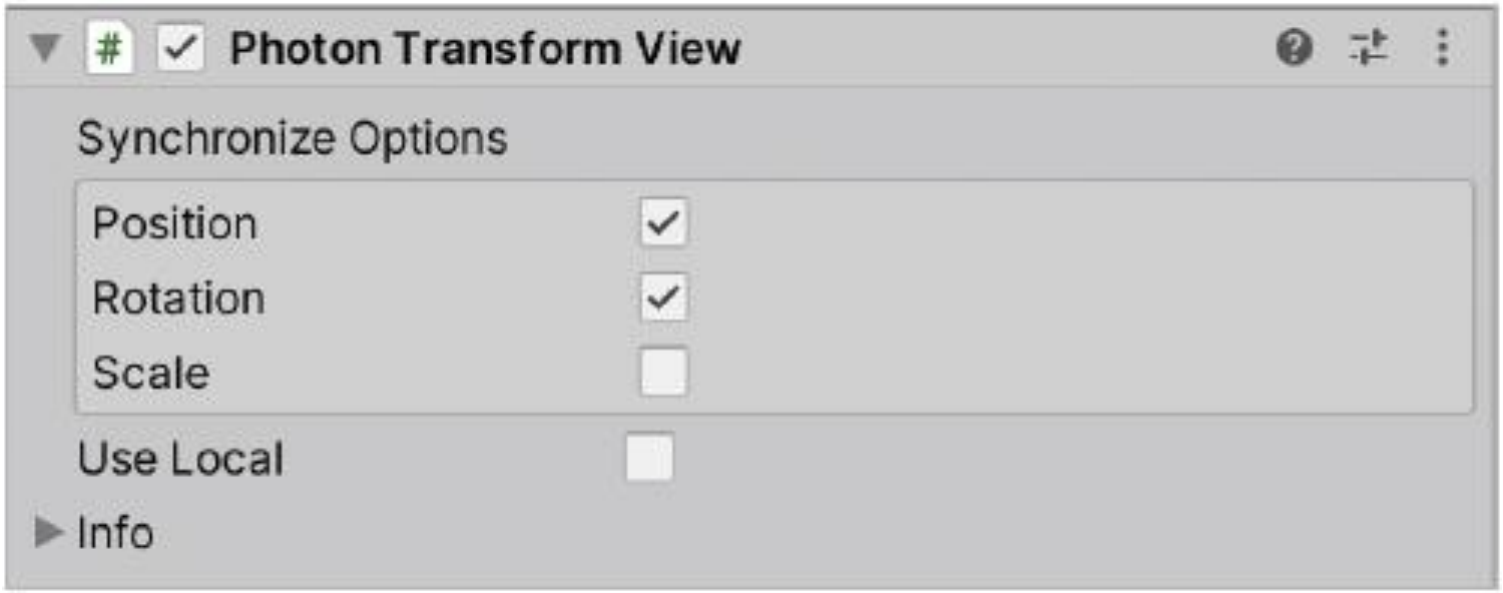
- 클라이언트 A가 Photon View 컴포넌트를 가진 게임 오브젝트 a를 생성한 다음 동기화를 통해 다른 클라이언트
  B에서도 게임 오브젝트 a를 생성하게 했다고 가정
  - Photon View 컴포넌트를 사용해 A 월드의 a와 B 월드의 a가 같은 네트워크 ID를 부여 받기 때문에 A 월드의 a와
    B 월드의 a를 같은 것으로 취급하고 둘을 동기화



# 네트워크 플레이어 캐릭터 준비(5)

## Photon Transform View 컴포넌트

- Photon Transform View 컴포넌트는 자신의 게임 오브젝트에 추가된 트랜스폼 컴포넌트 값의 변화를 측정하고, Photon View 컴포넌트를 사용해 동기화
- 현재 Player Character 게임 오브젝트의 Photon Transform View 컴포넌트는 트랜스폼의 위치와 회전을 동기화 하도록 설정

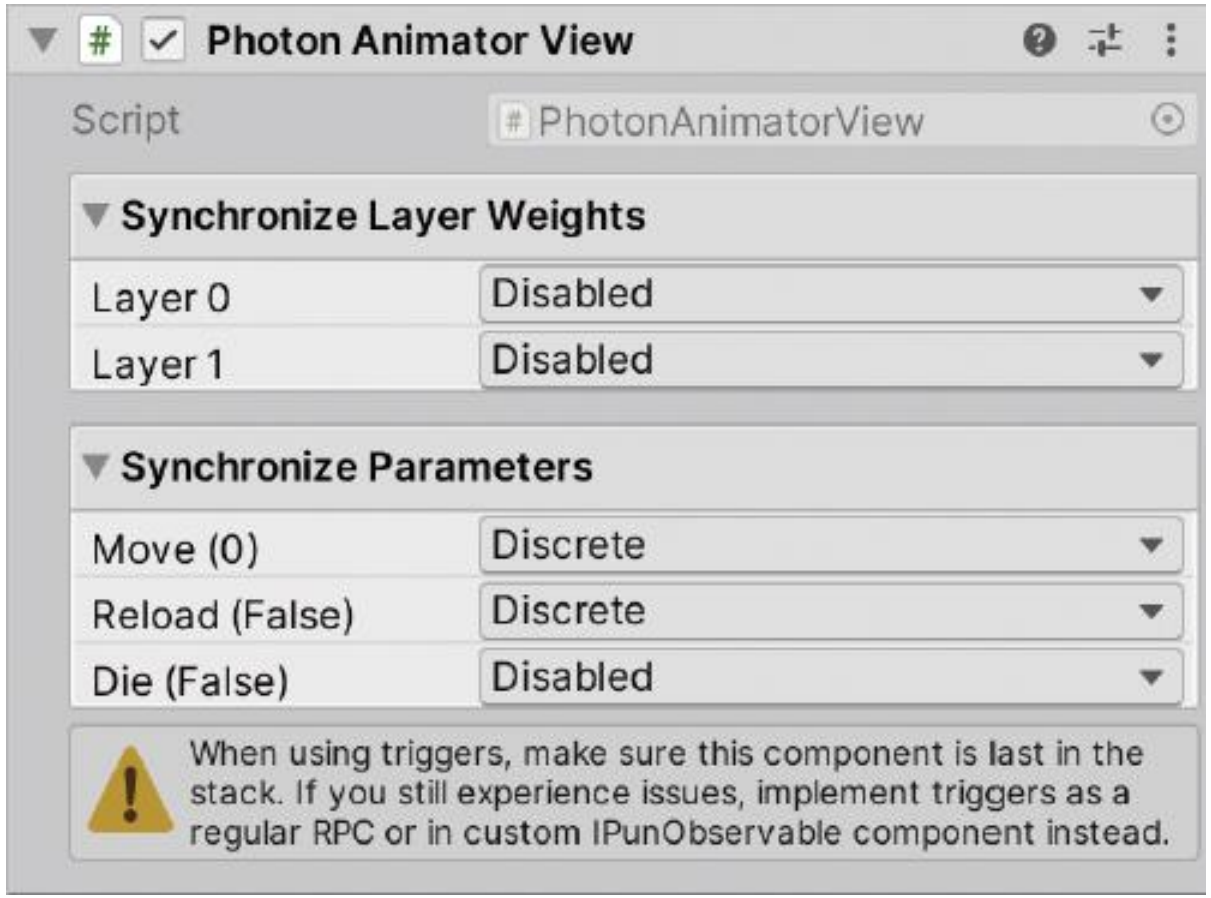



▶ Photon Transform View 컴포넌트

# 네트워크 플레이어 캐릭터 준비(6)

## Photon Animator View 컴포넌트

- Photon Animator View 컴포넌트는 네트워크를 넘어 로컬 게임 오브젝트와 리모트 게임 오브젝트 사이에서 애니메이터 컴포넌트의 파라미터를 동기화하여 서로 같은 애니메이션을 재생




▶ Photon Animator View 컴포넌트

# 네트워크 플레이어 캐릭터 준비(7)

## CameraSetup 스크립트

- CameraSetup 스크립트는 씬의 시네머신 가상 카메라가 로컬 플레이어만 추적하도록 설정
- Photon View 컴포넌트를 사용해 자신이 로컬인지 리모트인지 판별하고, 로컬이 맞다면 시네머신 가상 카메라가 자신을 추적하는 코드를 구현

**[과정 01] CameraSetup 스크립트 열기**
**Scripts 폴더에서 CameraSetup 스크립트 열기**

```
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() {

    }
}
```

**[과정 02] CameraSetup의 Start ( ) 메서드 완성하기**
**Start( ) 메서드를 다음과 같이 완성**

```
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 플레이어 캐릭터 준비(8)

- 새로 추가한 컴포넌트들의 역할
  - Photon View
    - Player Character 게임 오브젝트가 네트워크상에서 식별되게 함
    - 컴포넌트들의 값을 네트워크를 넘어 로컬-리모트 사이에서 동기화함
  - Photon Transform View
    - 로컬 Player Character의 트랜스폼 컴포넌트의 위치와 회전을 리모트 Player Character의 트랜스폼 컴포넌트에 동기화함
  - Photon Animator View
    - 로컬 Player Character의 애니메이터 컴포넌트의 파라미터를 리모트 Player Character의 애니메이터 컴포넌트에 동기화함
  - Camera Setup
    - 게임 오브젝트가 로컬 오브젝트면 시네머신 카메라가 자신을 추적하게 함

# 네트워크용 플레이어 캐릭터 컴포넌트(1)

## PlayerInput 스크립트

- 기존 기능 : 사용자 입력을 감지
- 변경된 기능 : 로컬 플레이어 캐릭터인 경우에만 사용자 입력 감지
- 기존 PlayerInput 스크립트에서 변경된 부분
  - using Photon.Pun; 추가
  - MonoBehaviour 대신 **MonoBehaviourPun** 사용
  - Update( ) 메서드 상단에 새로운 if 문 추가

```
if (!photonView.IsMine)
{
    return;
}
```

# 네트워크용 플레이어 캐릭터 컴포넌트(2)

## PlayerMovement 스크립트

- 기존 기능 : 사용자 입력에 따라 이동, 회전, 애니메이터 파라미터 지정
- 변경된 기능 : 로컬 플레이어 캐릭터인 경우에만 이동, 회전, 애니메이터 파라미터 지정
- PlayerMovement 스크립트의 주요 변경 사항
  - MonoBehaviour 대신 MonoBehaviourPun 사용
  - FixedUpdate( ) 메서드 상단에 로컬 여부를 검사하는 if 문 추가

```
if (!photonView.IsMine)
{
    return;
}
```

# 네트워크용 플레이어 캐릭터 컴포넌트(3)

## PlayerShooter 스크립트

- 기존 기능 : 사용자 입력에 따라 사격 실행 및 탄알 UI 갱신
- 변경된 기능 : 로컬 플레이어 캐릭터인 경우에만 사격 실행 및 탄알 UI 갱신
- 새로운 PlayerShooter 스크립트의 주요 변경 사항
  - MonoBehaviour 대신 MonoBehaviourPun 사용
  - Update( ) 메서드 상단에 로컬 여부를 검사하는 새로운 if 문 추가

```
if (!photonView.IsMine)
{
    return;
}
```

# 네트워크용 플레이어 캐릭터 컴포넌트(4)

## LivingEntity 스크립트

- 기존 기능 : 체력과 사망 상태 관리, 데미지 처리, 사망 처리
- 변경된 기능 : 호스트에서만 체력 관리와 대미지 처리 실행
- 새로운 LivingEntity 스크립트의 주요 변경 사항
  - MonoBehaviourPun 사용
  - 체력, 사망 상태 동기화를 위한 ApplyUpdatedHealth( ) 메서드 추가
  - OnDamage( ), RestoreHealth( )에 [PunRPC] 선언
  - OnDamage( )에서 대미지 처리는 호스트에서만 실행
  - RestoreHealth( )에서 체력 추가 처리는 호스트에서만 실행

# 네트워크용 플레이어 캐릭터 컴포넌트(5)

## [PunRPC]

- RPC를 구현하는 속성
- RPC를 통해 어떤 메서드를 다른 클라이언트에서 원격 실행할 때는 Photon View 컴포넌트의 RPC( ) 메서드를 사용
- RPC( ) 메서드는 입력으로 다음 값을 받음
  - 원격 실행할 메서드 이름(string 타입)
  - 원격 실행할 대상 클라이언트(RpcTarget 타입)
  - 원격 실행할 메서드에 전달할 값(필요한 경우)
- 자신의 Photon View 컴포넌트를 사용해 DoSomething( ) 메서드를 모든 클라이언트에서 원격 실행하는 코드

```
photonView.RPC("DoSomething", RpcTarget.All);
```

# 네트워크용 플레이어 캐릭터 컴포넌트(6)

## ApplyUpdateHealth( )

- 새로 추가된 ApplyUpdateHealth( ) 메서드는 [PunRPC] 속성으로 선언

```
[PunRPC]
public void ApplyUpdatedHealth(float newHealth, bool newDead) {
    health = newHealth;
    dead = newDead;
}
```

- ApplyUpdatedHealth ( )는 호스트 측 LivingEntity의 체력, 사망 상탯값을 다른 클라이언트의 LivingEntity에 전달하기 위해 사용

```
photonView.RPC("ApplyUpdatedHealth", RpcTarget.Others, health, dead);
```

# 네트워크용 플레이어 캐릭터 컴포넌트(7)

## OnDamage( )

- OnDamage( ) 내부 구현에서 추가한 코드는 호스트인 경우에만 대미지 수치를 적용하고, 그것을 호스트에서 다른 클라이언트로 전파하는 처리를 수행
  1. 호스트에서의 LivingEntity가 공격을 맞아 OnDamage( )가 실행됨
  2. 호스트에서 체력을 변경하고 클라이언트에 변경된 체력을 동기화
  3. 호스트가 다른 모든 클라이언트의 LivingEntity의 OnDamage( )를 원격 실행
- PhotonNetwork.IsMasterClient는 현재 코드를 실행하는 클라이언트가 마스터 클라이언트, 즉 호스트인지 반환하는 프로퍼티
- ApplyUpdatedHealth( ) 메서드를 원격 실행하여 호스트에서 변경된 체력을 다른 클라이언트에 적용

```
photonView.RPC("ApplyUpdatedHealth", RpcTarget.Others, health, dead);
```

- 다른 클라이언트에서도 OnDamage( ) 메서드를 원격 실행하는 처리가 이어짐

```
photonView.RPC("OnDamage", RpcTarget.Others, damage, hitPoint, hitNormal);
```

# 네트워크용 플레이어 캐릭터 컴포넌트(8)

## RestoreHealth( ) 메서드

- 변경된 RestoreHealth( ) 메서드는 [PunRPC] 속성이 선언되었으므로 어떤 클라이언트가 다른 클라이언트에서 원격 실행할 수 있음

```
// 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);
```

# 네트워크용 플레이어 캐릭터 컴포넌트(9)

## PlayerHealth 스크립트

- 기존 기능 : 플레이어 캐릭터의 체력 관리, 체력 UI 갱신
- 변경된 기능 : 리스폰 기능 추가, 아이템을 호스트에서만 사용
- 새로운 PlayerHealth 스크립트의 주요 변경 사항
  - RestoreHealth( ), OnDamage( )에 [PunRPC] 선언
  - Respawn( ) 메서드 추가
  - Die( ) 메서드 하단에서 Respawn( ) 실행
  - OnTriggerEnter( )의 item.Use( )를 if 문으로 감싸기

# 네트워크용 플레이어 캐릭터 컴포넌트(10)

## [PunRPC] 선언

- 오버라이드하는 측에서도 원본 메서드와 동일하게 [PunRPC] 속성을 선언해야 정상적으로 RPC를 통해 원격 실행
  - 따라서 PlayerHealth 스크립트의 RestoreHealth ( )와 OnDamage( )에도 동일한 [PunRPC] 속성을 선언
- 모든 클라이언트에서 PlayerHealth의 OnDamage( )가 동시에 실행된다고 가정했을 때 실제 대미지 적용은
  호스트에서만 실행
  - 나머지 클라이언트는 대미지를 입었을 때 겉으로 보이는 효과만 재생
  - PlayerHealth의 RestoreHealth( ) 메서드도 마찬가지

# 네트워크용 플레이어 캐릭터 컴포넌트(11)

## Die( ) 메서드

- 기존 Die( ) 메서드에 Invoke("Respawn", 5f);를 추가
  - Invoke( ) 메서드는 특정 메서드를 지연 실행하는 메서드
  - Invoke( ) 메서드는 지연 실행할 메서드의 이름과 지연시간을 입력받음
- 따라서 Die( ) 메서드가 실행되고 사망 후 5초 뒤에 Respawn( ) 메서드가 실행

# 네트워크용 플레이어 캐릭터 컴포넌트(12)

## Respawn( ) 메서드

- 새로 추가한 Respawn( ) 메서드는 사망한 플레이어 캐릭터를 부활시켜 재배치(리스폰)하는 메서드
- 부활 처리는 단순히 게임 오브젝트를 끄고 다시 켜는 간단한 방식으로 구현
- 자신의 게임 오브젝트 위치를 임의 위치로 옮기는 처리
  - 랜덤 위치는 반지름 5의 구 내부에서 임의 위치를 찾고, 높이 y 값을 0으로 변경하여 구현
  - 위치를 랜덤 지정하는 처리는 if (photonView.IsMine)에 의해 현재 게임 오브젝트가 로컬인 경우에만 실행

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

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

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

# 네트워크용 플레이어 캐릭터 컴포넌트(13)

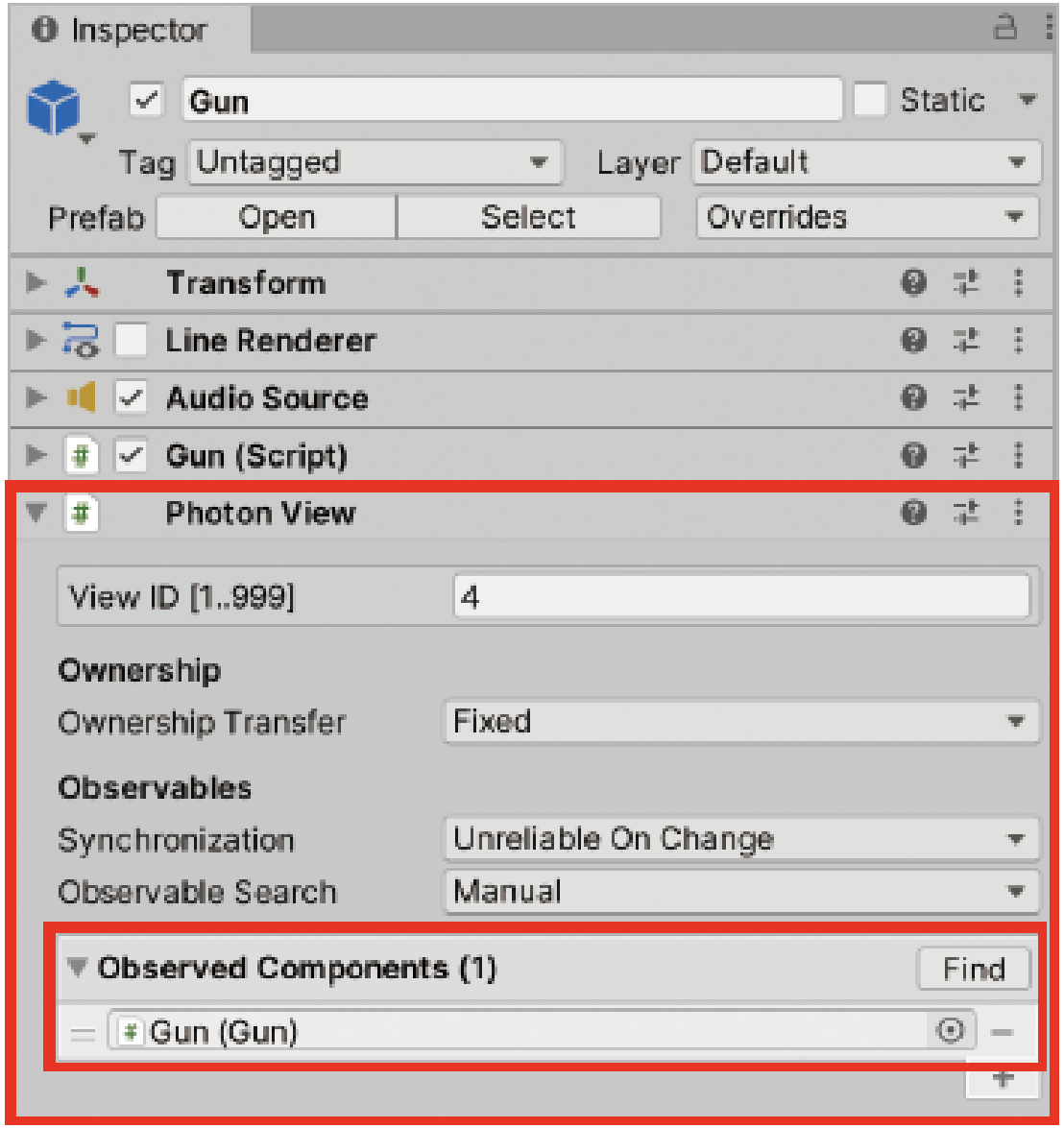
## OnTriggerEnter( ) 메서드

- 기존 OnTriggerEnter( ) 메서드는 충돌한 아이템을 감지하고 사용하는 처리를 구현
- 변경된 OnTriggerEnter( )는 기존 아이템 사용 처리 item.Use(gameObject);를 if 문으로 감싸서 호스트에서만 실행
  - 아이템을 먹는 효과음은 모든 클라이언트에서 실행되지만, 아이템 효과를 적용하는 부분은 호스트에서만 실행

```
if (PhotonNetwork.IsMasterClient)
{
    item.Use(gameObject);
}

playerAudioPlayer.PlayOneShot(itemPickupClip);
```

# 네트워크 Gun(1)

- Player Character 게임 오브젝트의 자식으로 추가된 Gun 게임 오브젝트와 Gun 스크립트의 변경 사항
  - Photon View 컴포넌트가 추가
  - Photon View 컴포넌트의 Observed Components 명단에 Gun 게임 오브젝트의 Gun 컴포넌트(스크립트)가 등록



# 네트워크 Gun(2)

## 변경된 Gun 스크립트 ➔ 이 부분은 수정된 파일을 배포했음

- 기존 기능 : 사격 실행, 사격 이펙트 재생, 재장전 실행, 탄알 관리
- 변경된 기능 : 실제 사격 처리 부분을 호스트에서만 실행, 상태 동기화
- 새로운 Gun 스크립트에 적용된 주요 변경 사항
  - MonoBehaviourPun 사용
  - IPunObservable 인터페이스 상속, OnPhotonSerializeView( ) 메서드 구현
  - 새로운 RPC 메서드 AddAmmo( ) 추가
  - Shot( )의 사격 처리 부분을 새로운 RPC 메서드 ShotProcessOnServer( )로 옮김
  - ShotEffect( )를 새로운 RPC 메서드 ShotEffectPocessOnClients( )로 감쌈

# 네트워크 Gun(3)

## IPunObservable 인터페이스와 OnPhotonSerializeView( ) 메서드

- Photon View 컴포넌트를 사용해 동기화를 구현할 모든 컴포넌트(스크립트)는 IPunObservable 인터페이스를 상속하고 OnPhotonSerializeView( ) 메서드를 구현

```
// 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();
    }
}
```

# 네트워크 Gun(4)

## AddAmmo( ) 메서드

- AddAmmo( ) 메서드는 탄알 추가 메서드
  - 호스트가 아이템을 사용하여 탄알을 추가했을 때 다른 클라이언트에서도 탄알이 추가되게 하는 RPC 메서드
- 호스트인 클라이언트 A는 모든 클라이언트 A와 B에서 b의 탄알이 증가하도록 RPC를 통해 모든 클라이언트에서 AddAmmo( )를 실행

# 네트워크 Gun(5)

## 호스트에 발사 처리 위임

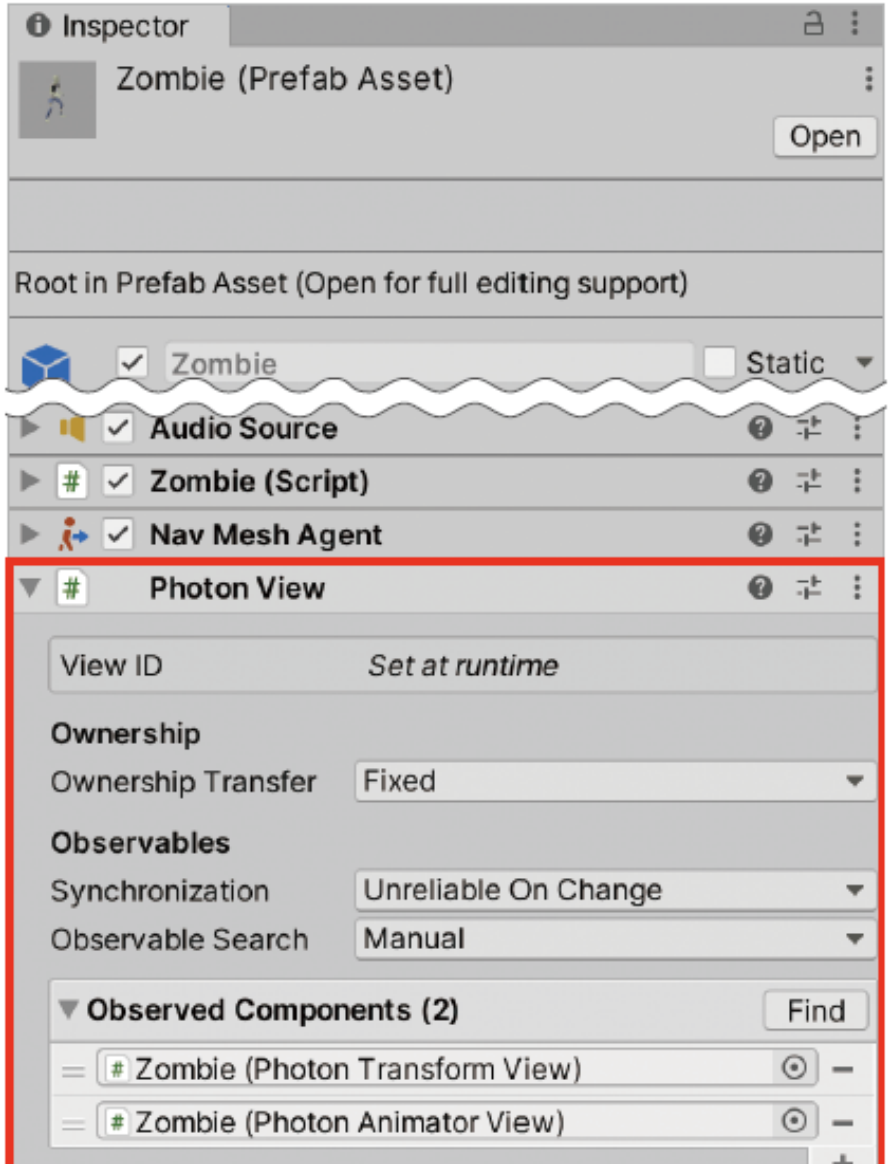
- Shot( )과 ShotEffect( )는 클라이언트의 발사 처리를 호스트에 맡기는 구조로 변경
  - 단, 발사 처리는 호스트에서만 실행해도 눈으로 보이는 사격 효과는 모든 클라이언트에서 실행
- 클라이언트 A, B, C가 존재하며 A가 호스트라고 가정했을 때 Shot( ) 메서드는 다음과 같이 실행
  1. 클라이언트 B의 로컬 플레이어 b의 총에서 Shot( ) 메서드 실행
  2. Shot( )에서 photonView.RPC("ShotProcessOnServer", RpcTarget.MasterClient); 실행
  3. 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
- 필요한 부분은 모두 확인했으므로, 씬에 남은 Player Character 게임 오브젝트 삭제
  ➔ 실제 게임에서는 프리펩을 사용해 생성할 것임

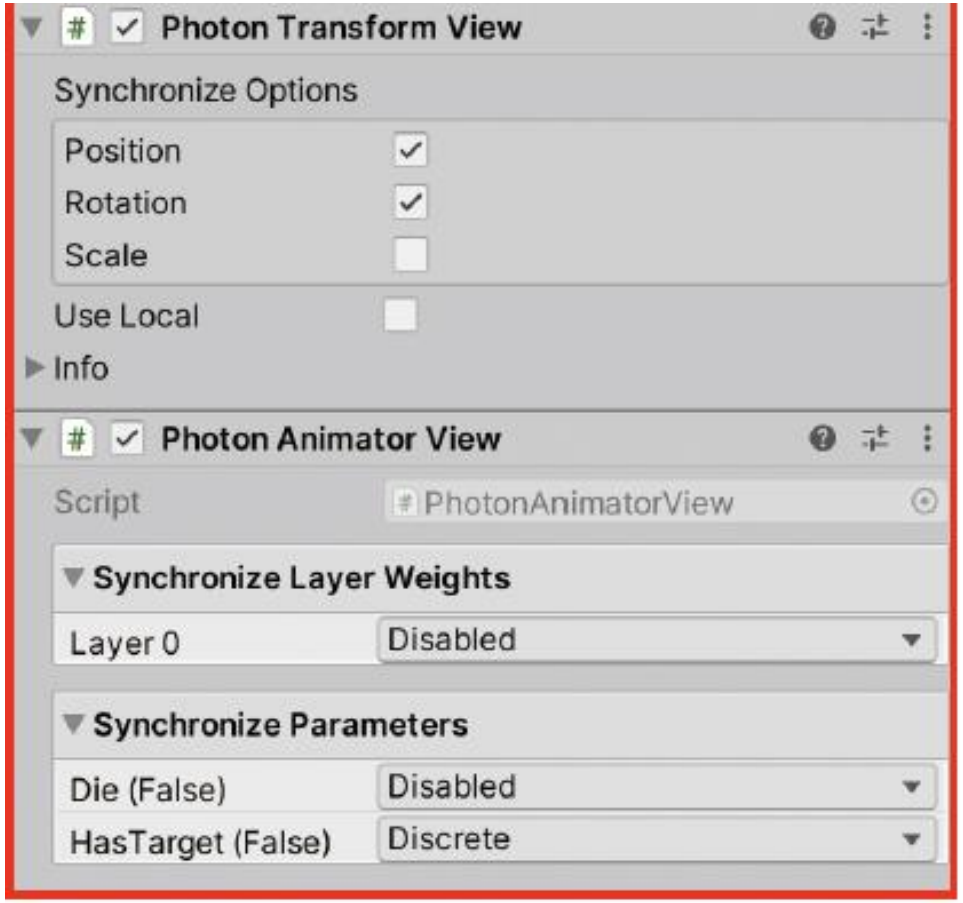
**[과정 01] 씬에 남은 Player Character 게임 오브젝트 삭제**

**①하이어라키 창에서 Player Character 게임 오브젝트 선택 > [Delete]키로 삭제**

# 네트워크 좀비(1)

- Zombie 프리팹에 Photon View, Photon Transform View, Photon Animator View 컴포넌트가 새로 추가되어짐



# 변경된 Zombie 스크립트

- 기존 기능 : 경로 계산, 목표 추적 및 공격
- 변경된 기능 : 호스트에서만 경로 계산, 추적, 공격을 실행

- 새로운 Zombie 스크립트는 기존 Zombie 스크립트에 다음과 같은 변경 사항이 적용
  - Setup( ), OnDamage( ) 메서드에 [PunRPC] 선언
  - Start( ), Update( ), OnTriggerStay( )를 호스트에서만 실행

# 네트워크 좀비(3)

## Setup( ), OnDamage( ) 메서드에 [PunRPC] 선언

- 생성한 좀비가 모든 클라이언트에서 동일한 능력치를 가지게 하려면 모든 클라이언트에서 좀비의 Setup( ) 메서드가 실행되어야 함
  - Setup( ) 메서드는 [PunRPC] 속성으로 선언
  - ZombieData를 통해 전달받던 체력, 공격력, 속도, 피부색만을 Setup( ) 메서드의 입력 파라미터로 직접 받음 ➔ 기존 방식은 불필요한 정보까지 클라이언트에 전송하므로 패킷 크기가 쓸데없이 커짐.. 네트워크 트래픽 초래
- 또한 'LivingEntity 스크립트'에서 OnDamage( ) 메서드에 이미 [PunRPC] 속성을 선언했지만 Zombie 스크립트에서 OnDamage( )를 오버라이드하면서 [PunRPC] 속성이 해지되었기 때문에 Zombie의 OnDamage( ) 메서드에서 [PunRPC] 속성을 다시 선언

# 네트워크 좀비(4)

## Start( ) 메서드를 호스트에서만 실행

- if 문을 추가하여 현재 코드를 실행 중인 클라이언트가 호스트가 아닌 경우에는 경로 계산을 시작하는 UpdatePath( ) 코루틴을 실행하지 못하도록 차단
  - 호스트의 Zombie 게임 오브젝트 위치를 다른 클라이언트의 Zombie 게임 오브젝트가 받아 적용하는 과정은 Photon View 컴포넌트에 의해 자동으로 이루어짐

```
if (!PhotonNetwork.IsMasterClient)
{
    return;
}
```

# 네트워크 좀비(5)

## Update( ) 메서드를 호스트에서만 실행

- 변경된 Update( ) 메서드 또한 if 문을 추가하여 클라이언트가 호스트가 아닌 경우에는 애니메이션 파라미터를 갱신하는 처리를 실행하지 못하게 함
- 물론 호스트에서만 좀비의 애니메이터 파라미터를 직접 갱신해도 Photon Animator View 컴포넌트에 의해 동기화되어 클라이언트에서도 같은 애니메이션이 재생되기 때문에 문제없음

```
if (!PhotonNetwork.IsMasterClient)
{
    return;
}
```

# 네트워크 좀비(6)

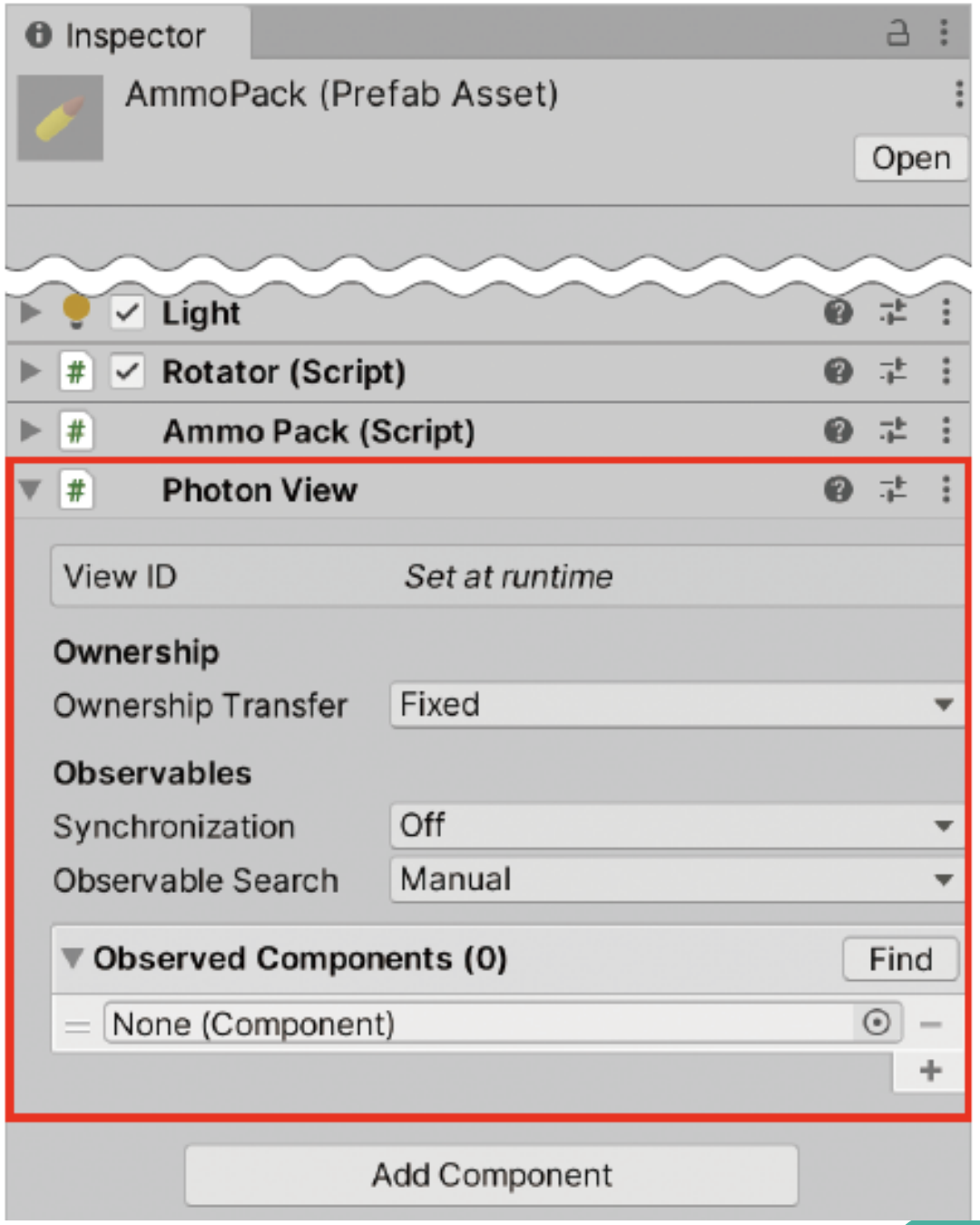
## OnTriggerStay( ) 메서드를 호스트에서만 실행

- 변경된 OnTriggerStay( ) 메서드는 최상단에 if 문을 추가하여 클라이언트가 호스트가 아닌 경우에는 공격을 실행하지 못하게 함
  - 즉, Zombie의 공격은 호스트에서만 이루어짐
  - 단, 공격을 받는 LivingEntity 타입은 'LivingEntity 스크립트'에서 살펴봤듯이 공격당한 결과를 다른 클라이언트에 RPC로 전파
  - 따라서 좀비가 플레이어 캐릭터를 공격한 결과는 호스트가 아닌 다른 클라이언트에도 무사히 적용

```
if (!PhotonNetwork.IsMasterClient)
{
    return;
}
```

# 네트워크 아이템(1)

- Prefabs 폴더의 AmmoPack 프리팹에 Photon View 컴포넌트가 추가
  - HealthPack 프리팹, Coin 프리팹에도 같은 변경 사항이 적용



# 네트워크 아이템(2)

## AmmoPack 스크립트 ➔ 수정된 코드로 배포했음

- 기존 기능 : 플레이어의 탄알 추가. 효과 적용 후 스스로를 파괴
- 변경된 기능 : 탄알 추가를 모든 클라이언트에서 실행. 모든 클라이언트에서 스스로를 파괴
- AmmoPack 스크립트는 다음과 같은 변경 사항이 적용
  - MonoBehaviourPun 사용
  - AddAmmo( )를 RPC로 원격 실행하여 탄알 추가
  - Destory( ) 대신 PhotonNetwork.Destroy( ) 메서드 사용
- 변경 사항의 역할 요약
  - 모든 클라이언트에서 탄알 추가
  - 네트워크상에서 동일 게임 오브젝트를 모두 파괴

# 19.5 네트워크 아이템(3)

## AddAmmo( ) 원격 실행

- 모든 클라이언트에서 원격으로 AddAmmo( ) 메서드가 실행되도록 코드를 변경
- 즉, 변경된 코드는 아이템 사용 자체는 호스트에서만 이루어지지만, 아이템을 사용하여 탄알이 증가하는 효과는 모든 클라이언트에서 동일하게 적용

```
// 총의 남은 탄환 수를 ammo 만큼 더합니다. 수정 전 코드
// playerShooter.gun.ammoRemain += ammo;


playerShooter.gun.photonView.RPC("AddAmmo", RpcTarget.All, ammo);
```

# 네트워크 아이템(4)

## PhotonNetwork.Destroy( ) 메서드 사용

- 네트워크상의 모든 클라이언트에서 동일하게 파괴되어야 하는 게임 오브젝트는 Destroy( ) 메서드 대신 PhotonNetwork.Destroy( ) 메서드를 사용
- PhotonNetwork.Destroy( )는 Photon View 컴포넌트를 가지고 있는 게임 오브젝트를 입력 받음
  - 입력된 게임 오브젝트는 모든 클라이언트에서 동시에 파괴
  - 즉, 사용된 탄알 아이템이 호스트에서 PhotonNetwork.Destroy(gameObject);를 실행하면 호스트를 포함한 모든 클라이언트에서 탄알 아이템 게임 오브젝트가 파괴

# 네트워크 아이템(5)

## HealthPack 스크립트➔ 수정된 코드로 배포했음

- 기존 기능 : 플레이어의 체력 추가. 효과 적용 후 스스로를 파괴
- 변경된 기능 : 모든 클라이언트에서 스스로를 파괴

- HealthPack은 AmmoPack과 달리 RPC를 사용하지 않음
  - 'LivingEntity 스크립트'에서 확인한 RestoreHealth( ) 메서드는 호스트에서 실행하면 자동으로 다른 클라이언트에서도 원격 실행되기 때문
- HealthPack 스크립트의 Use( ) 메서드 마지막에는 네트워크상의 모든 클라이언트에서 체력 아이템을 파괴하도록 PhotonNetwork.Destroy(gameObject);를 실행하도록 기존 코드 수정

# 네트워크 아이템(6)

**Coin 스크립트 ➔ 수정된 코드로 배포했음**

- 기존 기능 : 게임 점수 추가. 효과 적용 후 스스로를 파괴
- 변경된 기능 : 모든 클라이언트에서 스스로를 파괴
- Destroy( ) 메서드를 PhotonNetwork.Destroy( ) 메서드로 대체한 것 외에는 변경 사항이 없음

```
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 아이템(7)

**PhotonNetwork.Instantiate( ) : 매우 중요!!!**

- 자신의 게임 월드에서 어떤 게임 오브젝트를 생성하고, 같은 게임 오브젝트를 타인의 게임 월드에도 생성되게 하기위해서는 PhotonNetwork.Instantiate( ) 메서드를 사용
- PUN 구현 규칙으로 PhotonNetwork.Instantiate( )으로 생성된 프리팹들은 Resources 폴더에 있어야 함.
- 실시간으로 생성할 아이템 프리팹과 Player Character 프리팹, Zombie 프리팹을 Resources 폴더로 이동

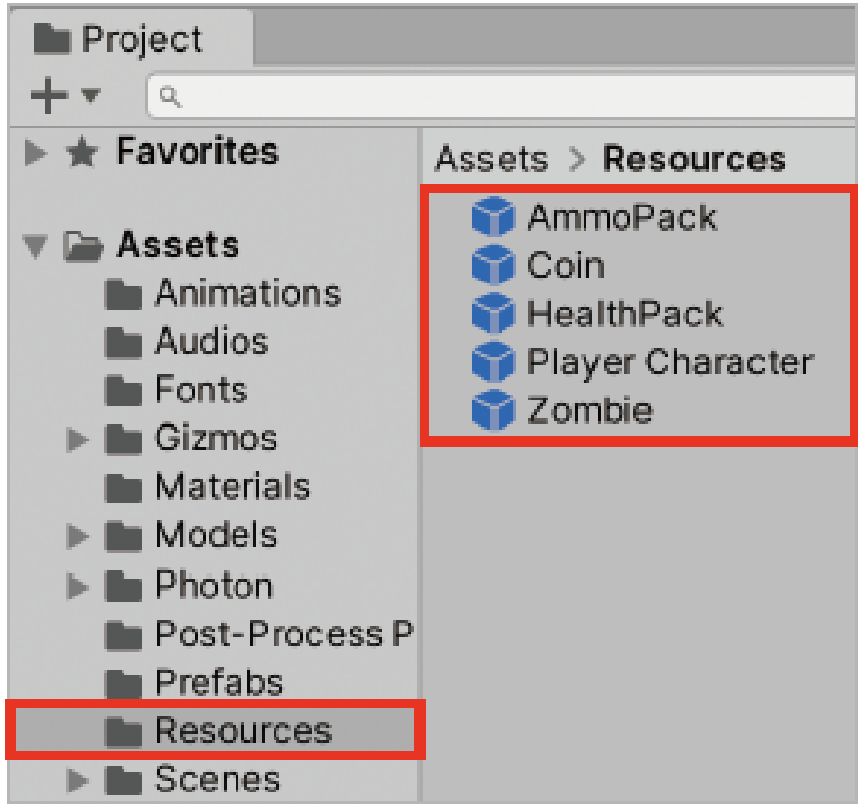
**[과정 01] Resources 폴더로 프리팹 옮기기**

①프로젝트에 Assets안에 Resources 폴더 생성

②프로젝트의 Prefabs 폴더에서 다음 프리팹 선택

- AmmoPack
- Coin
- HealthPack
- Player Character
- Zombie

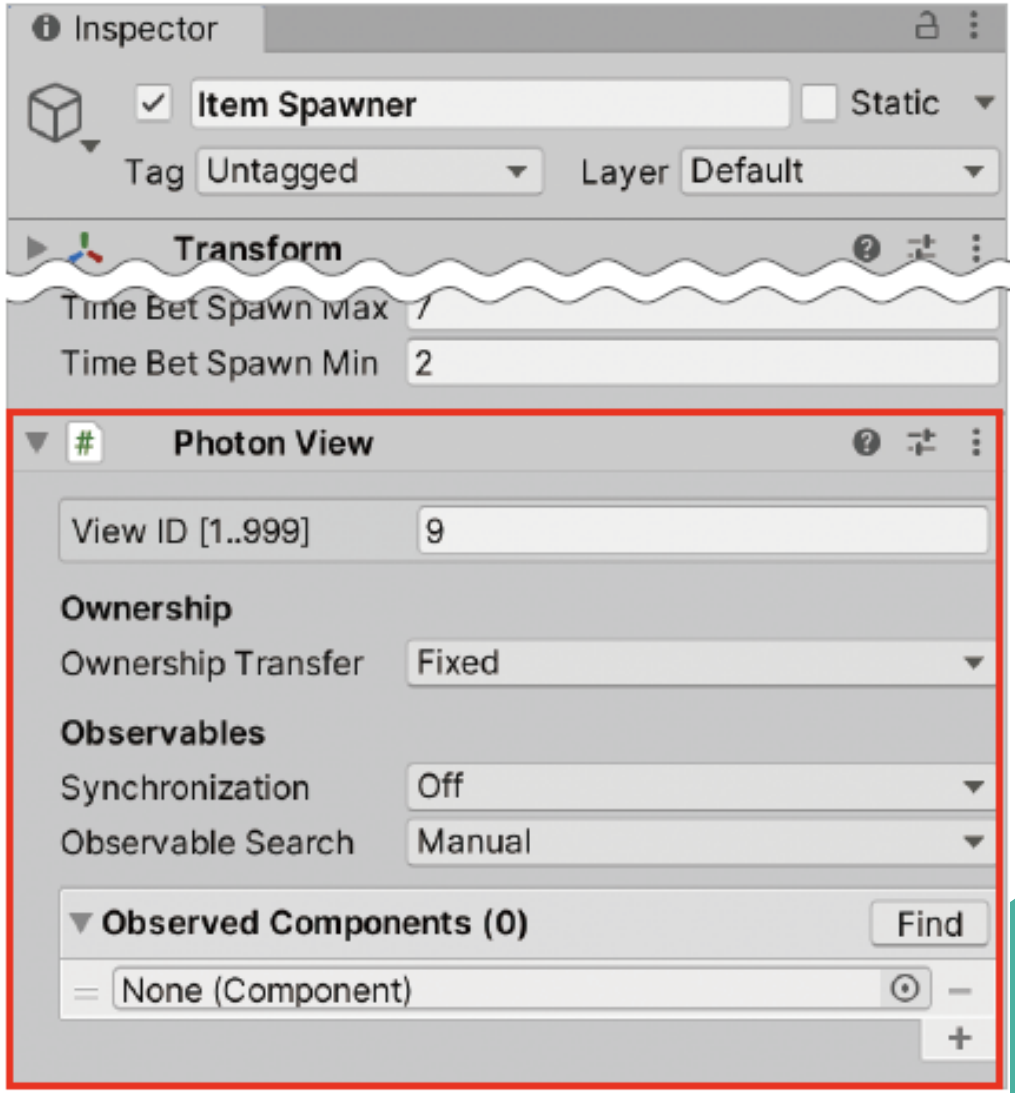
③선택한 프리팹을 Resource 폴더로 옮김



# 네트워크 아이템(8)

## ItemSpawner 스크립트

- 기존 기능 : 플레이어 캐릭터 근처에 아이템 생성
- 변경된 기능 : 맵 중심에 아이템 생성. 생성된 아이템을 일정 시간 후 모든 클라이언트에서 파괴
- 주요 변경 사항
  - 플레이어 캐릭터 위치를 사용하지 않음
  - 호스트에서만 아이템 생성
  - 아이템 생성은 Instantiate( ) 대신 PhotonNetwork.Instantiate( ) 사용
  - 아이템 파괴는 Destroy( ) 대신 PhotonNetwork.Destroy( ) 사용



## 플레이어 캐릭터 위치를 사용하지 않음

- 게임이 멀티플레이어로 변형되면서 플레이어 캐릭터가 둘 이상 존재하게 되었으므로 변경된 ItemSpawner는 아이템을 월드의 중심에서 maxDistance 반경 내의 랜덤 위치에 생성
- 따라서 기존 변수 playerTransform 선언을 삭제했으며, Spawn( ) 메서드에서 playerTransform.position을 사용한 부분을 다음과 같이 (0, 0, 0)에 대응하는 Vector3.zero로 변경

```
Vector3 spawnPosition = GetRandomPointOnNavMesh(Vector3.zero, maxDistance);
```

# 호스트에서만 아이템 생성

- 아이템 생성은 호스트에서 전담
- 따라서 Update( ) 메서드 상단에 다음 if 문을 삽입하여 호스트가 아닌 클라이언트에서는 Spawn( ) 실행에 도달하지 못하도록 변경

```
if (!PhotonNetwork.IsMasterClient)
{
    return;
}
```

# 네트워크 아이템(11)

## PhotonNetwork.Instantiate( ) 사용

- 호스트의 씬 뿐만 아니라 다른 클라이언트의 씬에서도 동일한 게임 오브젝트가 생성되고, 네트워크상에서
  동일한 게임 오브젝트로 취급되도록 하려면 PhotonNetwork.Instantiate( )를 사용
- 단, PhotonNetwork.Instantiate( ) 메서드는 프리팹을 직접 받지 못하고 프리팹의 이름을 받기 때문에 여러 개의
  아이템 프리팹 중 선택한 아이템 프리팹의 이름을 넣도록 Spawn( )의 코드를 수정

**기존 코드**

```
GameObject selectedItem = items[Random.Range(0, items.Length)];
GameObject item = Instantiate(selectedItem, spawnPosition, Quaternion.identity);
```

**수정된 코드**

```
GameObject selectedItem = items[Random.Range(0, items.Length)];

GameObject item = PhotonNetwork.Instantiate(selectedItem.name, spawnPosition,
    Quaternion.identity);
```

# 네트워크 아이템(12)

## PhotonNetwork.Destroy( ) 사용

- Destroy( ) 메서드 대신 PhotonNetwork.Destroy( ) 메서드를 사용
  - PhotonNetwork.Destroy( ) 메서드를 지연 생성하도록 감싸는 DestroyAfter( ) 코루틴을 추가

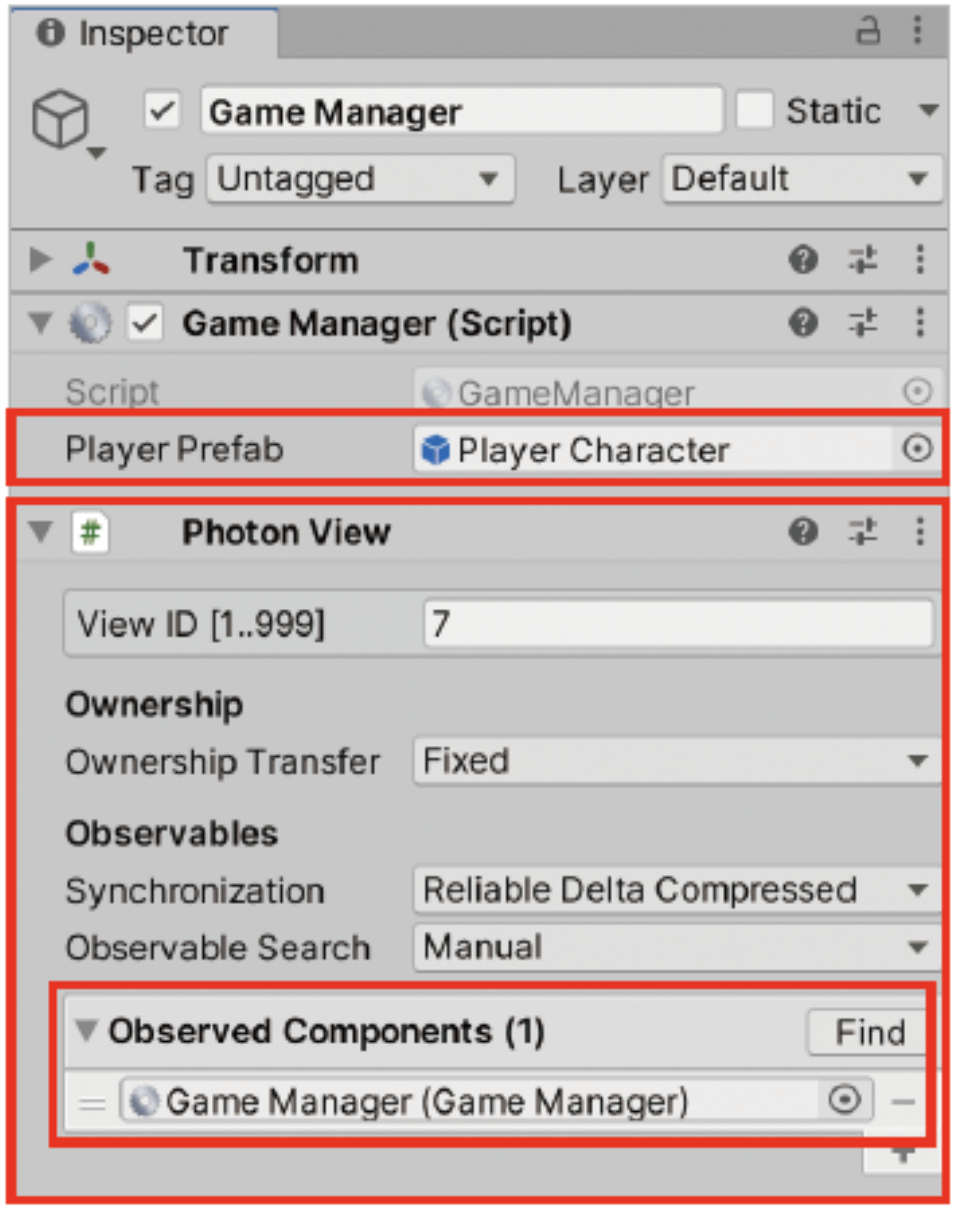
```
// 포톤의 PhotonNetwork.Destroy()를 지연 실행하는 코루틴
IEnumerator DestroyAfter(GameObject target, float delay) {
    // delay만큼 대기
    yield return new WaitForSeconds(delay);

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

- Spawn( ) 메서드에서 Destroy( ) 메서드 실행 부분을 DestroyAfter( ) 코루틴 실행으로 대체

```
StartCoroutine(DestroyAfter(item, 5f));
```

# 네트워크 게임 매니저(1)

- GameManager 컴포넌트에 이전에는 없던 Player Prefab 필드가 생겼으며, Player Character 프리팹이 할당
- Photon View 컴포넌트가 추가되었고, Observed Components에 GameManager 컴포넌트가 추가



# 네트워크 게임 매니저(2)

**GameManager 스크립트 ➔ 수정된 코드 배포하였음, 내용 확인할것**

- 기존 기능 : 게임 점수와 게임오버 상태 관리
- 변경된 기능 : 네트워크 플레이어 캐릭터 생성, 게임 점수 동기화, 룸 나가기 구현
- 주요 변경 사항
  - MonoBehaviourPunCallbacks 상속
  - 룸 나가기(로비로 돌아가기) 구현
  - IPunObservable 상속, OnPhotonSerializeView( ) 구현
  - Start( )에서 로컬 플레이어 캐릭터 생성

# 네트워크 게임 매니저(3)

## MonoBehaviourPunCallbacks 상속, 룸 나가기 구현

- OnLeftRoom( ) 메서드는 로컬 플레이어가 현재 게임 룸을 나갈 때 자동 실행
  - SceneManager.LoadScene("Lobby");에 의해 로컬 클라이언트의 씬만 Lobby 씬으로 변경되고, 다른 클라이언트는 여전히 룸에 접속된 상태

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

- GameManager 스크립트에 새로 추가된 Update ( ) 메서드에서는 키보드의 Esc 키 (KeyCode.Escape )를 눌렀을 때 네트워크 룸 나가기를 실행

```
private void Update() {
    if (Input.GetKeyDown(KeyCode.Escape))
    {
        PhotonNetwork.LeaveRoom();
    }
}
```

# 네트워크 게임 매니저(4)

## IPunObservable 상속, OnPhotonSerializeView( ) 구현

- IPunObservable 인터페이스를 상속하고 OnPhotonSerializeView( ) 메서드를 구현하여 로컬에서 리모트로의 점수 동기화를 구현하면 호스트에서 갱신된 점수가 다른 클라이언트에도 자동 반영

```
// 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);
    }
}
```

# 네트워크 게임 매니저(5)

## Start( )에서 로컬 플레이어 캐릭터 생성

- PhotonNetwork.Instantiate( ) 메서드를 실행해 자신의 로컬 플레이어 캐릭터를 네트워크상에서 생성
  - 즉, 자신의 입장에서는 로컬, 타인의 입장에서는 리모트인 플레이어 캐릭터가 생성
  - GameManager 스크립트의 Start( ) 메서드와 그 안의 PhotonNetwork.Instantiate( )는 각각의 클라이언트에서 따로 실행

```
private void Start() {
    // 생성할 랜덤 위치 지정
    Vector3 randomSpawnPos = Random.insideUnitSphere * 5f;
    // 위치 y 값은 0으로 변경
    randomSpawnPos.y = 0f;

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

# 네트워크 게임 매니저(6)

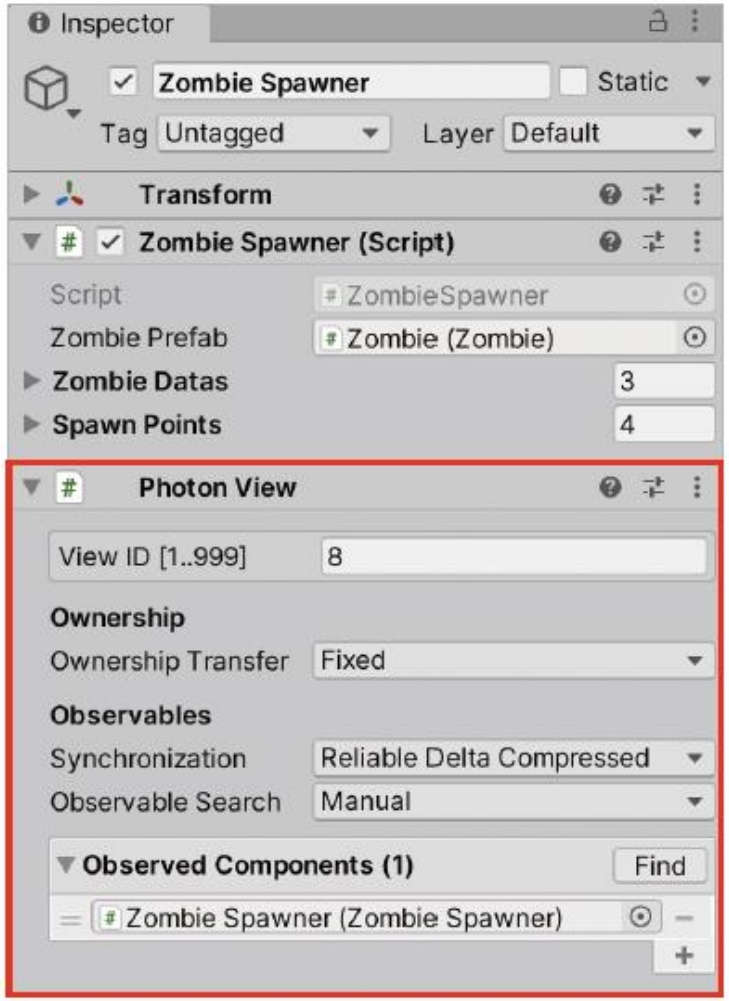
- 클라이언트 A와 B가 있다고 가정하고, A가 이미 접속한 상태에서 B가 나중에 룸에 접속했다고 가정
  1. 클라이언트 B가 룸에 접속 → B에서 GameManager의 Start( ) 실행
  2. PhotonNetwork.Instantiate( )에 의해 플레이어 캐릭터 b를 A와 B에 생성

**실행 순서**

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

# 좀비 생성기 포팅(1)

- Zombie Spawner 게임 오브젝트에 이전에는 없던 Photon View 컴포넌트 추가
- Photon View 컴포넌트의 Observed Components에는 ZombieSpawner 스크립트 추가



# 좀비 생성기 포팅(2)

## ZombieSpawner 스크립트

- 기존 기능 : 좀비 생성과 사망 시의 처리 등록. 남은 좀비를 UI로 표시
- 변경된 기능 : 네트워크상에서 좀비 생성. 남은 좀비 수 동기화
- 주요 변경 사항
  - zombieCount 변수 추가
  - IPunObservable 상속, OnPhotonSerializeView( ) 구현
  - CreateZombie( )에서 Zombie의 Setup( ) 메서드를 RPC로 원격 실행
  - DestoryAfter( ) 코루틴 메서드 추가
  - Awake( ) 메서드에서 Photon.Peer.RegisterType( ) 실행

# 1좀비 생성기 포팅(3)

## 웨이브 정보 동기화

```
// 남은 좀비 수을 위한 변수 추가
private int zombieCount = 0;
```

- 남은 좀비 수 zombieCount와 현재 웨이브 wave 값은 OnPhotonSerializeView( ) 메서드를 구현하여 동기화

```
// 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();
    }
}
```

# 좀비 생성기 포팅(4)

## Setup( ) 원격 실행

- 좀비 서바이버 멀티플레이어에서는 네트워크상의 모든 클라이언트에서 같은 좀비를 생성
- 따라서 CreateZombie ( ) 메서드에서 Instantiate ( )를 사용하여 zombiePrefab의 복제본을 생성하던 부분을 PhotonNetwork.Instantiate( )를 사용하도록 변경

**기존 코드**

```
Zombie zombie = Instantiate(zombiePrefab, spawnPoint.position, spawnPoint.rotation);
```

**변경된 코드**

```
GameObject createdZombie = PhotonNetwork.Instantiate(zombiePrefab.gameObject.name,
    spawnPoint.position,
    spawnPoint.rotation);

Zombie zombie = createdZombie.GetComponent<Zombie>();
```

# 좀비 생성기 포팅(5)

- 호스트뿐만 아니라 모든 클라이언트에서 생성된 좀비에 대해 Setup( ) 메서드를 원격 실행

**기존 코드**

```
zombie.Setup(zombieData);
```

**변경된 코드**

```
zombie.photonView.RPC("Setup", RpcTarget.All, zombieData.health, zombieData.damage,
    zombieData.speed, zombieData.skinColor);
```

# 좀비 생성기 포팅(6)

- CreateZombie( ) 메서드 마지막 부분에는 Zombie의 onDeath 이벤트에 생성한 좀비가 사망할 경우 실행될 메서드를 등록

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

# 좀비 생성기 포팅(7)

- PhotonNetwork.Destroy( ) 메서드를 지연하여 실행하는 코루틴 메서드를 만들어 기존 Destroy( ) 메서드를 대체

```
IEnumerator DestroyAfter(GameObject target, float delay) {
    // delay만큼 쉬고
    yield return new WaitForSeconds(delay);

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

**위 코드를 사용하여 기존 zombie.onDeath += ( ) => Destroy(zombie.gameObject, 10f);를 다음 코드로 대체**

```
zombie.onDeath += () => StartCoroutine(DestroyAfter(zombie.gameObject, 10f));
```

# 좀비 생성기 포팅(8)

## 직렬화와 역직렬화

- PUN은 RPC로 원격 실행할 메서드에 함께 첨부할 수 있는 입력 타입에 제약이 있음
  - RPC를 통해 다른 클라이언트로 전송 가능한 대표적인 타입으로는 byte, bool, int, float, string, Vector3, Quaternion 등
  - 이들은 직렬화/역직렬화가 PUN에 의해 자동으로 이루어짐
- PhotonPeer.RegisterType( ) 메서드를 실행하고, 원하는 타입을 명시하고, 어떻게 해당 타입을 직렬화(Serialize, 시리얼라이즈)/역직렬화(Deserialize, 디시리얼라이즈)할지 명시

```
PhotonPeer.RegisterType(타입, 번호, 직렬화 메서드, 역직렬화 메서드);
```

- 직렬화 - 어떤 오브젝트를 바이트 데이터로 변환하는 처리
- 역직렬화 - 바이트 데이터를 다시 원본 오브젝트로 변환하는 처리

# 좀비 생성기 포팅(9)

- PhotonPeer.RegisterType( ) 메서드에서 원하는 타입에 대한 직렬화와 역직렬화 메서드를 등록하면 PUN이 해당 메서드를 네트워크상에서 해당 타입을 주고받는 데 사용
- ZombieSpawner의 Awake( ) 메서드에 다음과 같은 처리를 추가

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

※ ColorSerialization.SerializeColor( )와 ColorSerialization.DeserializeColor( )는 저자가 미리 만들어둔 컬러 직렬화와 역직렬화 메서드

# 좀비 생성기 포팅(10)

## SerializeColor( )

- SerializeColor( ) 메서드는 들어온 오브젝트를 Color 타입으로 가정하고, 바이트 배열 데이터 byte[]로 직렬화
- 동시에 직렬화된 데이터의 길이를 short (int보다 더 적은 범위의 정수) 타입으로 반환

## DeserializeColor( )

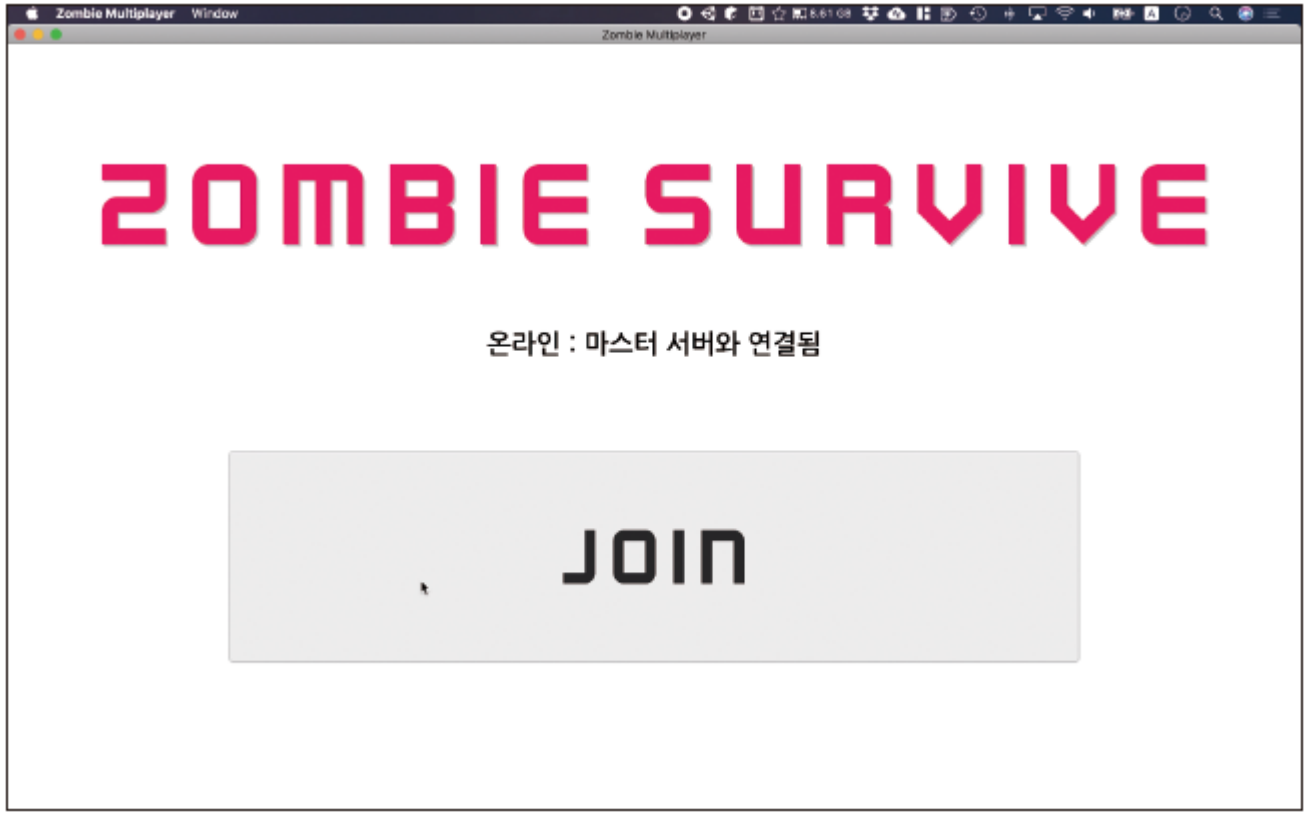
- DeserializeColor ( ) 메서드는 직렬화된 바이트 배열 데이터를 본래 타입인 Color 타입으로 변환

# 완성본 테스트(1)

- 포팅된 Zombie Multiplayer 프로젝트를 빌드하고 테스트

**[과정 01] 빌드하기**

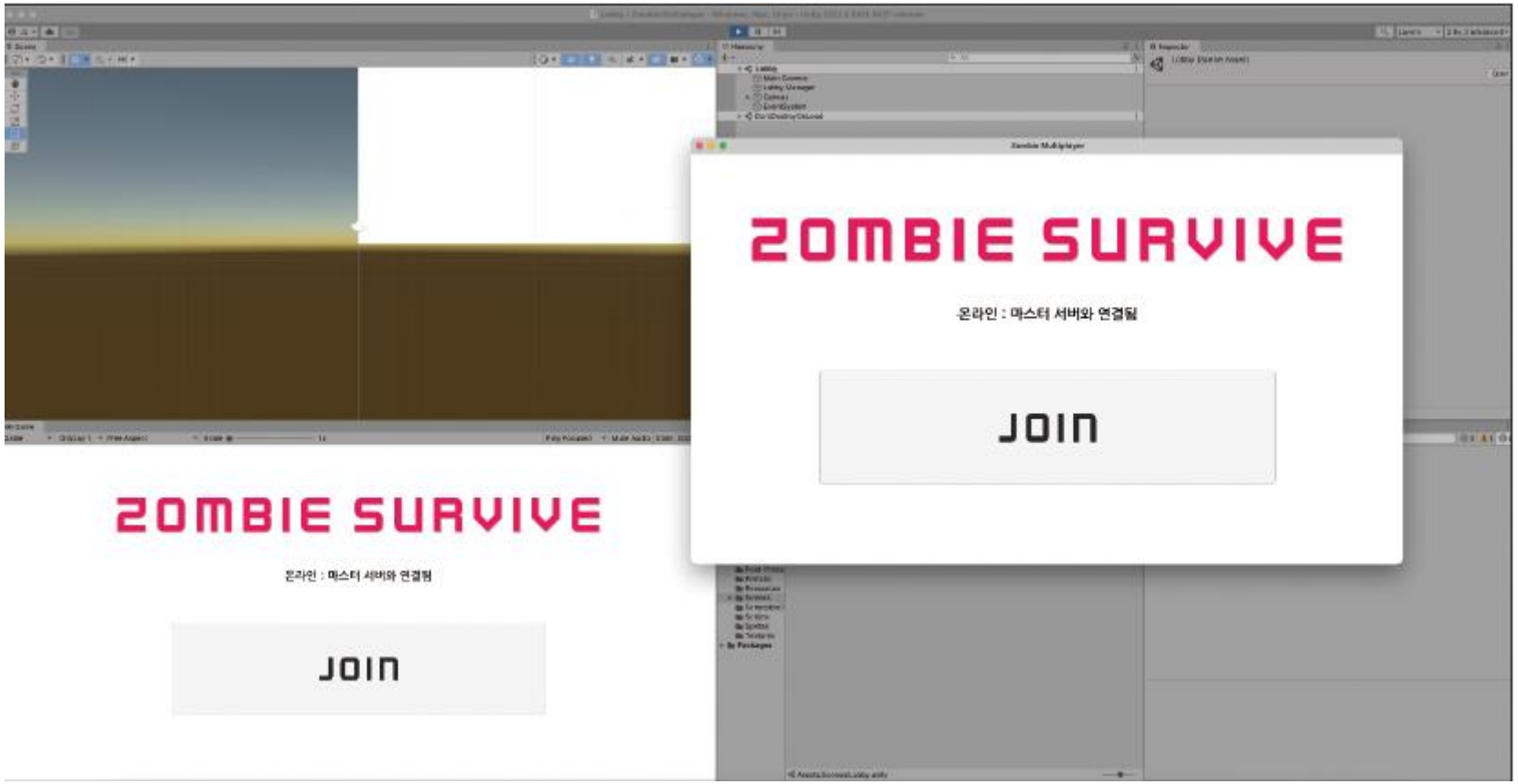
**① 적절한 경로에 빌드 및 실행(File > Build Settings > Build and Run)**




▶ 접속 준비된 로비

# 완성본 테스트(2)

- 멀티플레이어가 제대로 동작하는지 확인하려면 둘 이상의 클라이언트를 실행
  - 클라이언트 하나는 빌드된 프로그램을 창 모드로 띄워서 준비하고(창 모드 단축키 : 윈도우 [Alt+Enter], 맥 [Command+F] ), 다른 하나는 유니티 프로젝트를 사용




▶ 한 컴퓨터에서 두 개의 클라이언트 띄우기

# 완성본 테스트(3)

- 각 클라이언트에서 Join 버튼을 눌러 게임에 참가


▶ 여러 명의 플레이어와 함께 플레이