**본 글은 엔픽셀 인턴십 중 작성한 문서로서 회사의 허락을 받아 공개합니다.**

**Photon Network 101**

**Photon Unity Network : PUN**

**Network Connecting**

PUN2 에는 기본적으로 네트워크에 연결하고 룸 생성 및 참여하는 메서드를 제공합니다.

PhotonNetwork.**ConnectUsingSettings**(); *//포톤네트워크에 연결하는 메서드*

PhotonNetwork.**CreateRoom**(**string**); *//string이름의 룸을 생성하는 메서드*

PhotonNetwork.**JoinRoom**(**string**); *//string이름의 룸에 참여하는 메서드*

PhotonNetwork.**JoinRandomRoom**(); *//생성된 랜덤한 룸에 참여하는 메서드*

룸 생성 시 RoomOptions를 통해 룸 옵션을 설정할 수 있습니다.

RoomOptions roomOptions = **new** **RoomOptions**();

roomOptions.IsVisible = **true**;

roomOptions.MaxPlayers = 4;

PhotonNetwork.**JoinOrCreateRoom**(roomName, roomOptions);

네트워크 연결 시, 룸 입장 및 퇴장 시 자동으로 호출되는 다양한 콜백함수 또한 존재합니다.

**public** **override** **void** **OnConnectedToMaster**(); *//네트워크에 연결 시 호출*

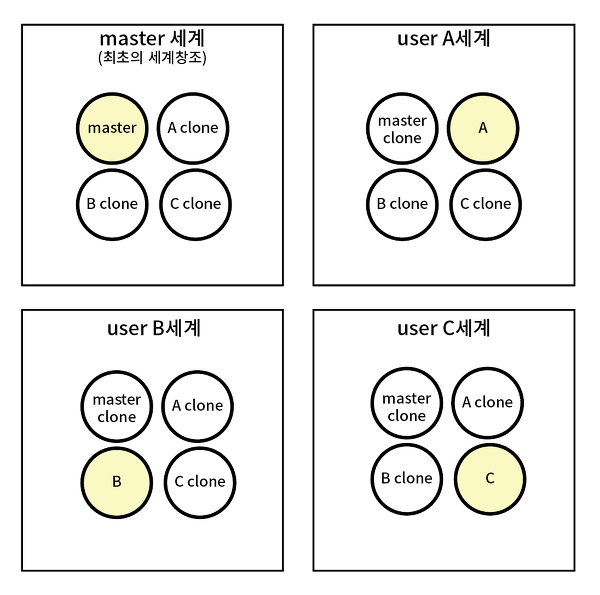
**public** **override** **void** **OnJoinedRoom**(); *//룸에 입장 시 호출*

**public** **override** **void** **OnPlayerLeftRoom**(Player otherPlayer); *//룸에서 퇴장 시 호출*

**public** **override** **void** **OnJoinRandomFailed**(**short** returnCode, **string** message); *//랜덤룸에 입장 실패 시 호출*

**PUN Overview**

In Game에 입장 시 PUN에서 각 클라이언트들의 상태는 아래와 같습니다.



PhotonView 컴포넌트를 가지는 오브젝트는, PhotonNetwork.Instantiate()시 각각의 클라이언트(세계)에서 동일한 viewID를 부여받게 됩니다. PhotonNetwork를 통해 생성되지 않는 경우 사용자가 직접 viewID를 부여하기도 합니다. 단, 한 씬에서 같은 viewID를 가지는 오브젝트가 존재해서는 안됩니다.

Ex) master세계의 A clone의 viewID가 2001이라면, userA의 세계, userB의 세계, userC의 세계에서 A clone은 모두 2001의 viewID를 가지게 됩니다.

Ex) userB세계의 B의 viewID가 1001이라면, master 세계, userA의 세계, userC의 세계에서 B clone은 모두 1001의 viewID를 가지게 됩니다.

PUN에서 내 오브젝트와 상대 오브젝트를 구별하는 것이 중요한데, photonView의 isMine변수로 구분하게 됩니다. 기본적으로 씬의 하이어라키에 생성한 오브젝트는 MasterCliinet가 주인이 됩니다.

Ex) master세계의 master오브젝트는 photonView.isMine = true, 나머지 오브젝트는 false입니다.

Ex) userB세계의 B오브젝트는 photonView.isMine = true, 나머지 오브젝트는 false입니다.

이는 각 클라이언트에서 Instantiate한 오브젝트로 구분할 수 있는데, 기본적으로 맵에 생성한 오브젝트는 모두 Master의 오브젝트로 생성됩니다.

Ex) A 클라이언트에서 Alpha 오브젝트를 Instantiate하면 A 세계에서 Alpha는 isMine이 true이며, 다른 세계에서는 isMine이 false가 됩니다.

PUN에서 동기화 하는 작업은 크게 아래 세 가지 경우로 나뉘게 됩니다.

1. PhotonView 컴포넌트를 가지는 오브젝트에 Photon Transform View 혹은 Photon Rigidbody View를 추가 해주는 경우

2. photonView의 RPC함수를 사용하는 경우

3. OnPhotonSerializeView() 함수를 사용하는 경우

**1. Photon Transform View, Photon Rigidbody View**

Photon Transform View, Photon Rigidbody View를 사용하는 경우, 동기화할 오브젝트는 그 오브젝트의 주인이 조작해야 동기화가 처리됩니다. (다른 클라이언트의 scene view에서 다른 플레이어 오브젝트의 Transform을 움직여도 나머지 클라이언트에서 동기화 처리는 되지 않습니다.)

Transform View의 경우 position, rotation, scale값을 동기화 해주며, Rigidbody View의 경우 velocity와 같은 Rigidbody 관련 변수들을 동기화 처리 해줍니다.

유저의 입력을 받아서 빠르게 처리해야 하는 Moving, Rotating와 같은 연산에 적합합니다.

Ownership Transfer를 통해 다른 클라이언트에서 생성한 오브젝트의 소유 권한을 가져올 수 있습니다.

photonView의 void TransferOwnership(Player player)함수는 해당 오브젝트의 권한을 player에게 넘기는 기능을 가지고 있습니다. 예시로 든 코드는 weapon의 photonView에서 TransferOwnership함수를 호출하여 현재 턴의 플레이어에게 weapon의 권한을 넘기는 기능입니다. 이렇게 처리하면 다른 플레이어에서 생성된 오브젝트라도 권한을 넘겨받은 플레이어가 조작 시 photonView의 동기화 처리가 정상적으로 작동합니다.

**2. photonView의 RPC함수**

RPC란 Romote Procedure Control 의 약자로, 원격 제어를 위한 코딩 없이 다른 주소 공간에서 함수나 프로시저를 실행할 수 있게하는 프로세스 간 통신 기술을 의미합니다. photonView의 RPC함수를 사용하는 경우, RPC함수는 각 클라이언트에서 같은 viewID를 가지는 오브젝트에서 호출합니다. 오브젝트의 상태를 변경하거나, 어떤 오브젝트를 생성하고 처리하는 연산에 적합합니다.



Ex) B세계의 viewID가 1001인 오브젝트가 RPC함수를 호출함 -> 나머지 세계에서 viewID가 1001인 오브젝트들이 해당 함수를 실행함

RpcTarget으로 해당 함수를 실행할 클라이언트를 설정할 수 있습니다. 자주 사용되는 종류는 아래와 같습니다.

RpcTarget.All : 자기를 포함한 모두에게 호출. 자기가 제일 빨리 실행함

Rpctarget.AllBufferedViaServer : 자기를 포함한 모두에게 호출. 자기와 다른 사람들의 실행 타이밍이 같음

RpcTarget.Others : 자기를 제외한 모두에게 호출

RpcTarget.MasterClient : MasterClient에서만 호출

호출하고자 하는 함수는 반드시 [PunRPC] Attritube가지고 있어야 합니다. 해당 함수를 RPC호출 하는 경우 함수 내 멤버변수들은 **함수를 직접 실행하는 클라이언트의 값**임을 인지해야 합니다. 함수를 실행하라고 호출하는 클라이언트의 멤버변수 값이 그대로 사용되지 않습니다.

RPC함수를 호출하는 모습 입니다. "SetTurnTimer"은 호출할 함수의 이름입니다. RpcTarget은 해당 함수를 호출할 대상을 정의합니다. 이후로 들어가는 값들은 RPC로 호출할 함수의 매개변수 입니다. 이 때 넘겨주는 매개변수 값은 해당 RPC함수를 호출하는 클라이언트의 값입니다. 주의해야 할 점은 어떤 클래스의 RPC함수가 private선언 되어 있다면, 이 클래스를 상속받은 자식 클래스에서 해당 함수를 RPC호출 할 수 없으며 protected나 public선언으로 접근이 가능하도록 해주어야 합니다.

**3. OnPhotonSerializeView()**

Photon Transform View, Photon Rigidbody View등의 컴포넌트를 사용한 동기화 처리는, 편리하지만 동기화가 제대로 처리 안 될 수도 있습니다. 이럴 때 OnPhotonSerializeView를 사용하여 사용자가 동기화 처리에 관여할 수 있습니다. 우선 OnPhotonSerializeView를 사용할 클래스에 IPunObserable 인터페이스를 상속합니다. 이후 아래 함수를 구현합니다.

**public** **void** **OnPhotonSerializeView**(PhotonStream stream, PhotonMessageInfo info)

{

**if** (stream.IsWriting)

{

*//내가 데이터를 보내는 중 일때 처리...*

}

**else**

{

*//내가 데이터를 받는 중 일때 처리...*

}

}

아래는 OnPhotonSerializeView를 사용하여 플레이어 위치를 보간 후 동기화하는 코드입니다.

**void** **Update**()

{

**if** (!photonView.IsMine)

{

transform.position = Vector3.**Lerp**(transform.position, remotePos, 10 \* Time.deltaTime);

}

}

**public** **void** **OnPhotonSerializeView**(PhotonStream stream, PhotonMessageInfo info)

{

**if** (stream.IsWriting)

{

stream.**SendNext**(transform.position);

}

**else**

{

remotePos = (Vector3)stream.**ReceiveNext**();

}

}

이 때 보내는 값과 받는 값의 순서가 일치하도록 코드를 작성해야 합니다.

**ActorNumber vs ViewID**

ViewID는 PhotonView 컴포넌트를 가지는 모든 오브젝트에게 부여되는 ID입니다. 하이어라키에 미리 생성된 오브젝트들은 1 ~ 1000까지의 번호를 사용할 수 있으며, PhotonNetwork.Instantiate()로 생성되는 오브젝트들은 master부터 1001 ~ 2000번의 번호를 자동으로 부여하고 이후 플레이어들은 2001 ~ 3000, 3001 ~ 4000...순서로 부여해 줍니다. 한 씬에 같은 ViewID를 가지는 오브젝트는 존재할 수 없으며, 이전 씬에서 DontDestroy로 넘어온 오브젝트가 중복된 ViewID를 가지는 경우 에러 메시지를 출력합니다.

ActorNumber는 PhotonNetwork에서 Room에 플레이어가 입장 시 각 플레이어를 식별하기 위해 플레이어에게 부여하는 ID입니다. 이 ActorNumber는 Master부터 1, 2, 3...순으로 순차적으로 부여하며, 플레이어가 룸을 이탈 시 해당 플레이어가 가지고 있던 번호는 다시 사용하지 않습니다.

Ex) Player1 Room 생성 -> ActorNumberList { 1 }

Player2 Room 입장 -> ActorNumberList { 1, 2 }

Player2 Room 이탈 -> ActorNumberList { 1 }

Player3 Room 입장 -> ActorNumberList { 1, 3 }

Player2 Room 재입장 -> ActorNumberList {1, 3, 4 }

ViewID는 오브젝트마다 가지는 ID 이지만, ActorNumber는 플레이어마다 가지는 ID라는 차이점이 존재합니다. ActorNumber를 사용하여 각 오브젝트의 Owner Player를 구분할 수 있으며 이는 Photon에서 오브젝트를 다루는 것과 어떤 플레이어의 Input를 받을건지 구분함에 편리함을 제공합니다.

**Player CustomProperty**

Photon에서 Player마다 가질 수 있는 Property를 제공합니다. 기본적으로 ActorNumber, NickName과 같은 Property가 있으며, 여기서 추가로 개발자가 Property를 더해주고 싶은 경우 CustomProperty기능을 사용할 수 있습니다.

CustomProperty는 Hashtable의 형태로 구성됩니다. 따라서 CustomProperty를 추가할 경우 Photon에서 제공되는 Hashtable 자료구조를 사용해야 합니다.

**using** Hashtable = ExitGames.Client.Photon.Hashtable;

CustomProperty를 추가해주는 스크립트의 상단에 해당 코드를 삽입하여 Hashtable 자료구조를 ExitGames에서 제공하는 Hashtable로 사용합니다.

Hashtable \_playerProperty = **new** **Hashtable**();

\_playerProperty["TeamColor"] = "Red";

\_playerProperty["KillCnt"] = 0;

\_playerProperty["DeathCnt"] = 0;

PhotonNetwork.LocalPlayer.CustomProperties = \_playerProperty;

이후 위와 같은 형태로 player에게 CustomProperty를 추가할 수 있습니다. 상단의 코드는 클라이언트의 로컬 플레이어에게 TeamColor, KillCnt, DeathCnt의 Property를 추가한 모습 입니다.

추가된 Custom Property를 사용하는 것은 일반적인 Hashtable과 같이 사용할 수 있습니다.

**private** **void** **SetPrevTeamDeathCnt**()

{

**if** ((**string**)\_turnManager.CurrentPlayer.CustomProperties["TeamColor"] == "Red")

{

\_prevTeamDeathCnt = \_redTeamDeathCnt;

}

**else** **if** ((**string**)\_turnManager.CurrentPlayer.CustomProperties["TeamColor"] == "Blue")

{

\_prevTeamDeathCnt = \_blueTeamDeathCnt;

}

}

**private** **void** **UpdatePlayerKillCnt**()

{

**if** ((**string**)\_turnManager.CurrentPlayer.CustomProperties["TeamColor"] == "Red")

{

\_turnManager.CurrentPlayer.CustomProperties["KillCnt"] = (**int**)\_turnManager.CurrentPlayer.CustomProperties["KillCnt"] + (\_prevTeamDeathCnt - \_redTeamDeathCnt);

}

**else** **if** ((**string**)\_turnManager.CurrentPlayer.CustomProperties["TeamColor"] == "Blue")

{

\_turnManager.CurrentPlayer.CustomProperties["KillCnt"] = (**int**)\_turnManager.CurrentPlayer.CustomProperties["KillCnt"] + (\_prevTeamDeathCnt - \_blueTeamDeathCnt);

}

**if** (\_turnManager.CurrentCharacter.GetComponent<DefaultCharacter>().\_isDeath)

{

\_turnManager.CurrentPlayer.CustomProperties["KillCnt"] = (**int**)\_turnManager.CurrentPlayer.CustomProperties["KillCnt"] - 1;

}

}

Custom Property의 프로퍼티 리턴 값의 자료형은 Hashtable이므로 해당 데이터를 사용할 때는 사용할 데이터의 자료형으로 변환해야 합니다.

Custom Property를 사용할 때 주의할 점은, Player역시 Object들과 마찬가지로 클라이언트마다 독립된 개체로 인식된다는 것 입니다. Player1 세계의 Player1의 Custom Property값을 바꾼다고 해서 Player2 세계의 Player1의 Custom Property값도 바뀌지 않습니다. Custom Property가 바뀔 때 실행되는 Call-back 함수를 이용하여 다른 클라이언트들의 값을 바꿔줄 수 있습니다.

Hashtable hash = **new** **Hashtable**();

hash["CustomProp"] = "something";

player.**SetCustomProperties**(hash);

->

**public** **override** **void** **OnPlayerPropertiesUpdate**(Player target, Hashtable changedProps)

{

*//Write something you expect*

}

또한 Lobby에서 Room생성 시 RoomOptions.BroadcastPropsChangeToAll값을 true로 설정 해주어야 Custom Property의 변화를 다른 클라이언트에 알릴 수 있습니다.