Link to Lab Drive

https://drive.google.com/drive/folders/1jgFTWCJ5JD-JnIY7n_L7s2n2vA1F9nNX?usp=sharing

OR

https://tinyurl.com/ta-comp476-daniel

Photon Unity Networking (PUN)

PUN

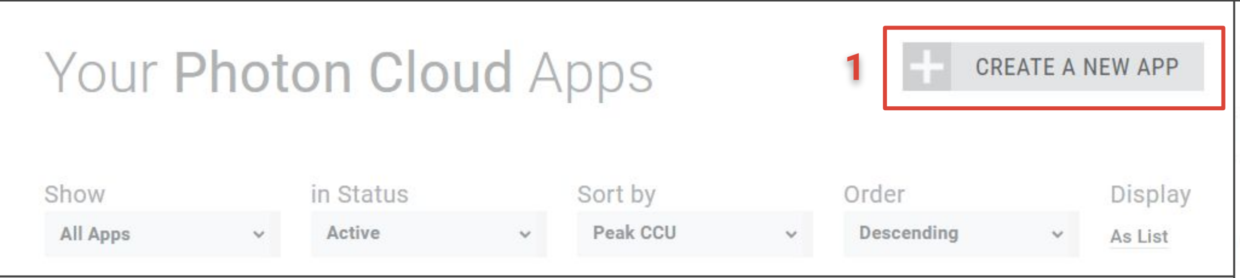
Photon is a real-time multiplayer game development framework that is fast, lean and flexible. Photon consists of a server and multiple client SDKs for major platforms.

Photon Unity Network (PUN) is a Unity specific, high-level solution: Matchmaking, easy to use callbacks, components to synchronize GameObjects, Remote Procedure Calls (RPCs) and similar features provide a great start. Beyond that is a solid, extensive API for more advanced control.

There is an Online Documentation, which is considered a manual for PUN. This might become your primary source for information. It summarizes the most important classes in the Public API module and explains each class, method and field individually. This is generated from the source of PUN and should be used to look up details on usage and parameters.

Before you get started with Photon, make sure to create a free Photon account here :
https://id.photonengine.com/en-US/Account/SignUp

PUN: Photon App Setup



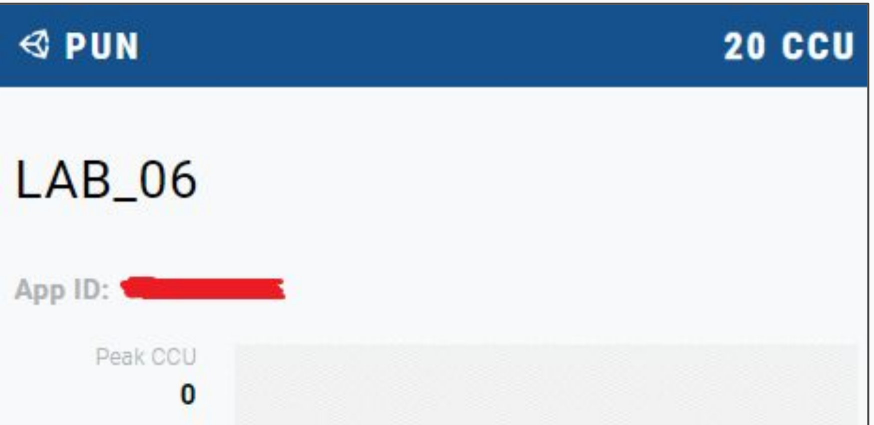
Your Photon Cloud Apps

1 CREATE A NEW APP

Show	in Status	Sort by	Order	Display
All Apps	Active	Peak CCU	Descending	As List

After registering and signing in, head to your Photon dashboard to setup a new Photon app: https://dashboard.photonengine.com/en-US/

After creating a new app, take note of your App ID as you will use it later in the Unity editor. You need to create an App for each networked game you create.

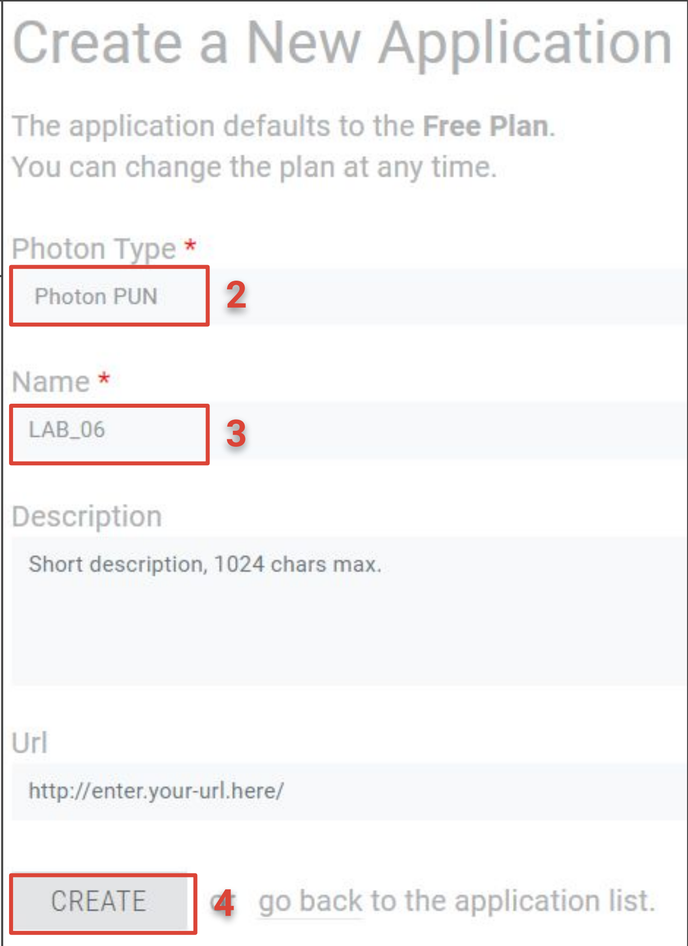


PUN 20 CCU

LAB_06

App ID: [redacted]

Peak CCU
0



Create a New Application

The application defaults to the **Free Plan**.
You can change the plan at any time.

Photon Type *
Photon PUN 2

Name *
LAB_06 3

Description
Short description, 1024 chars max.

Url
http://enter.your-url.here/

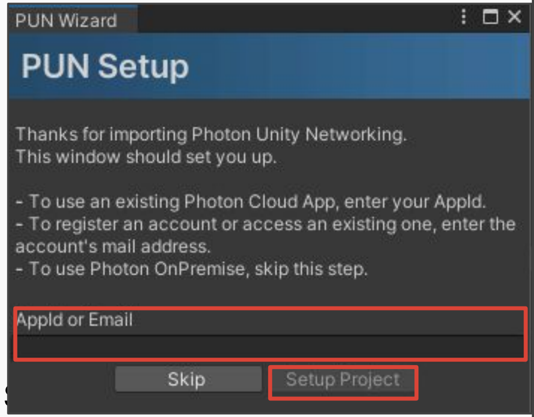
CREATE 4 go back to the application list.

PUN: Unity Setup

After creating a new project in Unity, go to the Unity asset store and get the free PUN 2 package which can support a maximum of 20 players:
https://assetstore.unity.com/packages/tools/network/pun-2-free-119922

In Unity, go to **Window > Package Manager**. In the Package Manager Window, select **My Assets** under the packages dropdown and you should see "PUN 2 - FREE" in the list of packages. Select it and in the bottom right corner of the window press the "download" button and then press import. This should import all the required packages and group them under a folder called "Photon" in the root of your Assets folder.

Once imported, the Photon Project Setup Wizard should appear. Enter your App ID that you made a note of earlier in the text field and press "Setup Project". This should create and setup a Photon Server Settings object that will contain various settings about your PUN application.



PUN Wizard

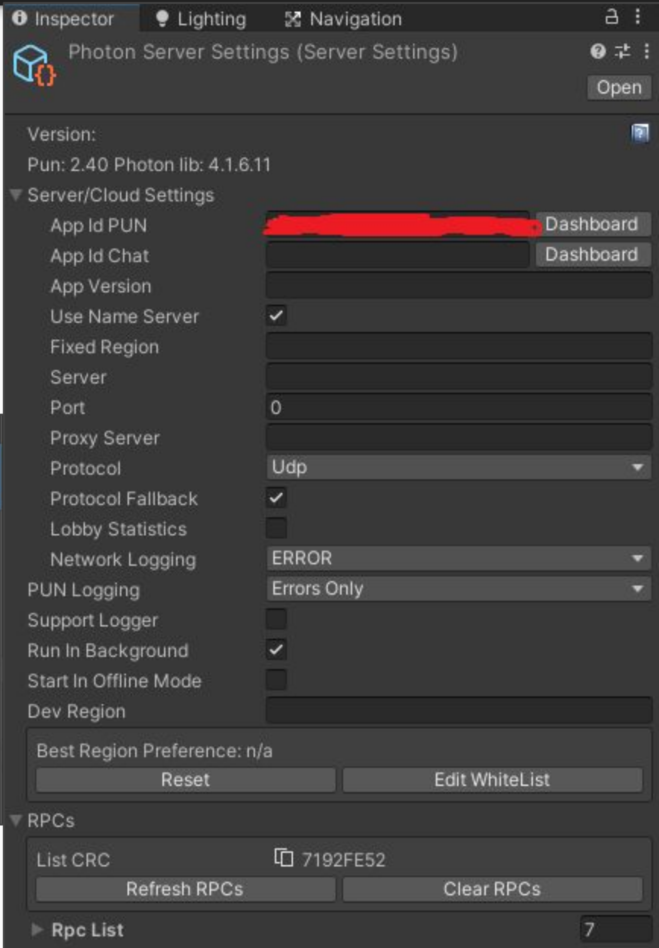
PUN Setup

Thanks for importing Photon Unity Networking.
This window should set you up.

- To use an existing Photon Cloud App, enter your AppId.
- To register an account or access an existing one, enter the account's mail address.
- To use Photon OnPremise, skip this step.

AppId or Email

Check the documentation for more info on Photon
https://doc.photonengine.com/en-us/pun/v2/getting-started/initial-setup



Photon Server Settings (Server Settings)

Version:
Pun: 2.40 Photon lib: 4.1.6.11

Server/Cloud Settings

App Id PUN	[redacted]
App Id Chat	
App Version	
Use Name Server	✓
Fixed Region	
Server	
Port	0
Proxy Server	
Protocol	Udp
Protocol Fallback	✓
Lobby Statistics	
Network Logging	ERROR
PUN Logging	Errors Only
Support Logger	
Run In Background	✓
Start In Offline Mode	
Dev Region	

Best Region Preference: n/a

RPCs

List CRC 7192FE52

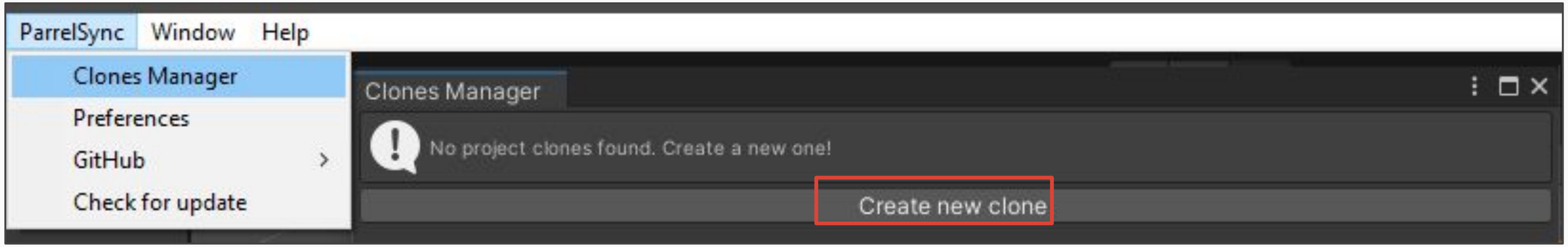
Rpc List 7

ParrelSync

ParrelSync is a Unity package that is useful for testing multiplayer features in your project. You can download the source code via the github repo here: https://github.com/VeriorPies/ParrelSync

You can also download the latest .unitypackage file from here : https://github.com/VeriorPies/ParrelSync/releases
Then you just need to drag and drop the .unitypackage file into the UnityEditor and it should prompt you for the import.

After importing ParrelSync into the project, you can now open the clones manager window and create and then open the cloned project.



No project clones found. Create a new one!

This will clone the entire project and keep the synced project synced up with the current one. Why do we do this? Because the alternative is to make a new build of your game each time you make changes to the code. This is a very lengthy procedure that wastes a lot of time.

Connection And Callbacks

In order to connect to photon servers using the PhotonServerSettings asset, you should call the **ConnectUsingSettings()** function.

```
void ConnectToPhotonExample() {
    PhotonNetwork.AutomaticallySyncScene = true // sync the current scene across all clients in the same room
    PhotonNetwork.ConnectUsingSettings();
}
```

When you load a scene, Unity usually destroys all GameObjects in the current scene before loading the new one. This includes networked objects (we will discuss these soon), which can lead to issues on the network. To avoid network issues with loading scenes, you can set **PhotonNetwork.AutomaticallySyncScene** to true and use **PhotonNetwork.LoadLevel()** to sync and load a scene across all clients in a room.

PUN uses **callbacks** to let you know when the client established the connection, joined a room, etc…
For convenience, PUN has the MonoBehaviourPunCallbacks class which inherits from MonoBehaviour. It implements all the important callback interfaces and registers itself automatically, so you can inherit it and just override specific callback methods.

```
public class YourClass : MonoBehaviourPunCallbacks {
    public override void OnConnectedToMaster() {
        Debug.Log("OnConnectedToMaster() was called by PUN.");
        PhotonNetwork.JoinRandomRoom();
    }
}
```

Alternatively you can implement the interface **IConnectionCallbacks** in any class and then register instances for callbacks via the static method **PhotonNetwork.AddCallbackTarget()**.

Matchmaking

The **OnConnectedToMaster() callback** will be invoked when the client is connected to the Master Server and ready for matchmaking and other tasks. At this point, you could try to join an existing game session or create your own. The following code snippets show possible method calls to start or join game sessions (**Rooms**).

```
// Join room "someRoom"
PhotonNetwork.JoinRoom("someRoom");
  // Fails if "someRoom" does not exist, closed or full. Error callback: IMatchmakingCallbacks.OnJoinRoomFailed
```

```
// Tries to join any random room:
PhotonNetwork.JoinRandomRoom();
  // Fails if there are no open rooms. Error callback: IMatchmakingCallbacks.OnJoinRandomFailed
```

```
// Create this room.
PhotonNetwork.CreateRoom("MyMatch");
  // Fails if "MyMatch" room already exists and calls: IMatchmakingCallbacks.OnCreateRoomFailed
```

When friends want to play together and have a way to communicate outside of PUN (e.g. with Discord), they can make up a room name and enter it in the game. Then the code can invoke **JoinOrCreateRoom(roomName, roomOptions, lobby)**. If nobody else should be matched into this room, you can also make it invisible for matchmaking.

```
RoomOptions roomOptions = new RoomOptions();
roomOptions.IsVisible = false; // make this room invisible for matchmaking
roomOptions.MaxPlayers = 4;
PhotonNetwork.JoinOrCreateRoom(roomName, roomOptions, TypedLobby.Default);
```

With **JoinOrCreateRoom(...)**, the room gets created on demand, so it doesn't matter who is first. If it's full, IMatchmakingCallbacks.OnJoinRoomFailed() gets invoked. For more info: MatchMaking Guide

Networked Instantiate

To create networked GameObjects, use **PhotonNetwork.Instantiate()** instead of Unity's Object.Instantiate. Any client in a room can call this to create objects which it will control.

```
PhotonNetwork.Instantiate("MyPrefabName", new Vector3(0, 0, 0), Quaternion.identity);
```

The **prefab must have a PhotonView component** attached. This component stores and manages a ViewID (the identifier for network messages), who owns the object, which scripts will read and write network updates and how those updates are sent. Check the inspector to setup a PhotonView component via the UnityEditor. A networked object should only have 1 PhotonView component attached to it.

By default, PUN instantiate uses a DefaultPool implementation, which loads prefabs from "Resources" folders and Destroys the GameObject later on.

By default, GameObjects created with PhotonNetwork.Instantiate exist as long as the creator is in the room. When you swap rooms, objects don't carry over, just like when you switch a scene in Unity.

When a client leaves a room, the remaining players will destroy the GameObjects created by the leaving player. If this doesn't fit your game logic, you can disable this: Set the RoomOptions.CleanupCacheOnLeave to false, when you create a room.

The Master Client can create GameObjects that have the lifetime of the room by using PhotonNetwork.InstantiateRoomObject(). Note: The object is not associated with the Master Client but the room. By default, the Master Client controls these objects but you can pass on control with photonView.TransferOwnership().

Networked Destroy

You can also manually and explicitly destroy networked objects, using PhotonNetwork.Destroy().

As previously mentioned, when you leave a room the networked GameObjects get destroyed automatically. However, if you are connected and joined to a room and you want to **destroy a networked GameObject** that was created using PhotonNetwork.Instantiate(...), you should use **PhotonNetwork.Destroy(...)**. This will handle some cleanup tasks and destroy the gameobject across the network.

In order for PhotonNetwork.Destroy(...) to succeed, the GameObject to be destroyed must meet the following conditions:

- The GameObject was instantiated at runtime using PhotonNetwork.Instantiate() method call.
- If the client is joined to an online room, the GameObject's PhotonView must be owned or controlled by the same client.

Note: GameObjects can be destroyed locally using PhotonNetwork.Destroy(...) without issues if the client is not joined to a room or joined in offline mode (more on this at the end).

For more on Instantiation in PUN, see https://doc.photonengine.com/en-us/pun/current/gameplay/instantiation

Networked Game Logic

As previously mentioned, GameObjects can be instantiated as "networked GameObjects" by attaching a **PhotonView** component. It identifies the object and the owner across the network using a viewID. The photon player who's in control, is the one that will update everyone of changes to this object.

Typically, you would add a PhotonView component to a prefab and use **PhotonNetwork.Instantiate()** to create an networked instance of that object. The prefab that you wish to instantiate must be located within a folder called “Resources” as per the default implementation of the Photon PrefabPool. Recall that Unity discourages and advises against using the Resources folder. Luckily for us there is a way around this which involves implementing the IPunPrefabPool interface. This will be shown in a later slide.

The PhotonView manages a list of observed components. By default this only includes itself. A custom observed component is in charge of writing and reading the state of the networked object several times a second. To do so, a script must implement the **IPunObservable** interface, which defines the **OnPhotonSerializeView(PhotonStream stream, PhotonMessageInfo info)** method.

```
// used as Observed component in a PhotonView, this only reads/writes the position
public void OnPhotonSerializeView(PhotonStream stream, PhotonMessageInfo info) {
    if (stream.IsWriting) // write properties to stream
    {
        Vector3 pos = transform.localPosition;
        stream.Serialize(ref pos);
    }
    else  // read properties from stream
    {
        Vector3 pos = Vector3.zero;
        stream.Serialize(ref pos);  // pos gets “filled-in”.
    }
}
```

Serialization and Deserialization must be done in the same order.

This way of synchronizing state is useful and should be used for data that gets updated **very frequently**.

Normally, if all you need to do is sync the Transform component data then you just need to add a **PhotonTransformView** component to the gameobject and drag it into the observed components list of the PhotonView component.

Networked Game Logic

Clients can also perform Remote Procedure Calls (RPC) on specific networked objects for anything that happens **infrequently**:

```
// defining a method that can be called by other clients:
[PunRPC]
public void ExampleMethodRPC(byte myParameter)
{
    Debug.LogFormat("RPC call ‘ExampleMethodRPC’ : {0}", myParameter);
}
```

```
// calling the RPC somewhere else in your code
photonView.RPC("ExampleMethodRPC", RpcTarget.All, (byte)1); // need to pass something, if nothing needed then pass 1 byte
```

RPC Targets, Buffering And Order

You can define which clients execute an RPC. Most of the time we just send the event to everyone using **RPCTarget.All**.

- **MasterClient** : Sends the RPC to the master client (host) only.
- Others values [**Others**, **OthersBuffered**] : Sends the RPC to everyone else. The sending client does not execute the RPC.
- Buffered values [**AllBuffered**, **AllBufferedViaServer**, **OthersBuffered**] : The server will remember your RPCs and when a new player joins, it gets the RPC event even though it happened earlier (unless the sending client leaves). Use this with care, as a long buffer list causes longer join times.
- ViaServer values [**AllViaServer**, **AllBufferedViaServer**] : Usually, when the sending client has to execute an RPC, it does so immediately, then sends it to all other clients. This, however, affects the order of events, because there is no lag when calling a method locally. This is especially interesting when RPCs should be done in order. RPCs sent via the server are executed in the same order by all receiving clients. It is the order of arrival on the server.
- Alternatively, you can call an RPC for a specific player in the room. Use the overloaded function with the target Player as second parameter. If you directly target the local player then this will be executed locally and won't go through the server.

Networked Game Logic

RPC Considerations

- By design, the script that has the RPC methods needs to be attached to the exact same GameObject as the PhotonView not its parent nor its child.
- The RPC method cannot be static.
- Generic methods are not supported as PUN RPCs.
- An RPC method that has a return value other than void can be called but the return value will not be used. Unless you need the return value in other explicit direct calls to the same method, always use void as a return value.
- If the RPC method is overridden, do not forget to add the [PunRPC] attribute to it. Otherwise, inherited methods not overridden that have [PunRPC] attribute on the base class can be used.
- Do not attach more than one component of the same type and that have RPC methods on the same GameObject as a PhotonView. If you have a class called MyClass that implements a method MyRPC marked as PunRPC, you should not attach more than one instance of it to the same GameObject. You can even make use of Unity's [DisallowMultipleComponent] class attribute.
- Make sure that you use a unique name per RPC method. Do not create overloaded RPC methods.
- It is not recommended to use optional parameters in RPC methods. If necessary, pass all parameters including optional ones during the RPC call. Otherwise, the receiving clients will not be able to find and process the incoming RPC.
- **Not all data types are automatically serializable by Photon**. Try to use simple types like Vector3, int, float, bool, string. For a full list of supported data types visit: https://doc.photonengine.com/en-us/realtime/current/reference/serialization-in-photon
- If you want to send an object array as a parameter of an RPC method, you need to cast it to object type first.

Networked Game Logic

Independent from GameObjects, you can **raise your own events** and send them without any relation to some networked object:

```
PhotonNetwork.RaiseEvent(eventCode, eventContent, RaiseEventOptions.Default, SendOptions.SendReliable);
```

- Events are described by using an unique identifier, the **eventCode**. In Photon this event code is described as a byte value, which allows up to 256 different events. However some of them are already used by Photon itself, so you can't use all of them for custom events. After excluding all of the built-in events, you still have the possibility to use up to 200 custom event codes [0..199]. The event code 0 should be avoided if you are going to use advanced events cache manipulation (see link below).
- The **eventContent** can be anything that PUN can serialize (see the bottom of the previous slide for more info on what data types Photon can serialize).
- The third parameter describes the **RaiseEventOptions**. With these options, you define which clients get the event, if it's buffered, etc.
- The last parameter describes the **SendOptions**. With these options, you can choose if this event is sent reliable or unreliable or choose if the message should be encrypted. In our example we just want to make sure that our event is sent reliably.

Read more about PUN's RPCs and RaiseEvent() .

Networked Game Logic

Full Example using RaiseEvent():

```
public class SendEventExample {
// If you have multiple custom events, it is recommended to define them in the used class
    public const byte MoveUnitsToTargetPositionEventCode = 1;

    private void SendMoveUnitsToTargetPositionEvent() {
        // Array contains the target position and the IDs of the selected units
        object[] content = new object[] { new Vector3(10.0f, 2.0f, 5.0f), 1, 2, 5, 10 };
        // You would have to set the Receivers to All in order to receive this event on the local client as well
        RaiseEventOptions raiseEventOptions = new RaiseEventOptions() { Receivers = ReceiverGroup.All };

        PhotonNetwork.RaiseEvent(MoveUnitsToTargetPositionEventCode, content, raiseEventOptions, SendOptions.SendReliable);
    }
}
```

```
public class ReceiveEventExample : MonoBehaviour, IOnEventCallback {
    private void OnEnable() { PhotonNetwork.AddCallbackTarget(this); }
    private void OnDisable() { PhotonNetwork.RemoveCallbackTarget(this); }

    public void OnEvent(EventData photonEvent) {
        byte eventCode = photonEvent.Code;
        if (eventCode == SendEventExample.MoveUnitsToTargetPositionEvent) {
            object[] data = (object[])photonEvent.CustomData;
            Vector3 targetPosition = (Vector3)data[0];
            for (int index = 1; index < data.Length; ++index) {
                int unitId = (int)data[index];
                UnitList[unitId].TargetPosition = ...
            }
        }
    }
}
```

Offline Mode

Offline mode is a feature to be able to reuse your multiplayer code in single player game modes as well.
The most common features that you'll want to reuse in single player are sending RPCs and using PhotonNetwork.Instantiate(). The main goal of offline mode is to disable null references and other errors when using PhotonNetwork functionality while not connected. You would still need to keep track of the fact that you're running a single player game, to set up the game etc. However, while running the game, all code should be reusable.You need to manually enable offline mode, as PhotonNetwork needs to be able to distinguish erroneous from intended behaviour.
Enabling this feature is very easy:

```
PhotonNetwork.OfflineMode = true;
```

Once set to true, Photon will invoke OnConnectedToMaster() and then you can create a room, this room will be of course Offline too.
You can now reuse certain multiplayer methods without generating any connections or errors. Furthermore there is no noticeable overhead.
Below follows a list of PhotonNetwork functions and variables and their results during offline mode:

- The actor number returned by PhotonNetwork.LocalPlayer is always -1.
- PhotonNetwork.PlayerList contains only the local player and PhotonNetwork.PlayerListOthers is always empty.
- PhotonNetwork.Time: returns Time.time.
- PhotonNetwork.IsMasterClient: Always true.
- PhotonNetwork.Instantiate(), PhotonNetwork.Destroy(), PhotonNetwork.NickName, PhotonView.RPC(), PhotonNetwork.AllocateViewID() all work as expected on the current client. RPC calls however, will not be buffered and/or carried over if you later decide to go online.
- PhotonNetwork.RemoveRPCs/RemoveRPCsInGroup/SetInterestGroups/SetSendingEnabled/SetLevelPrefix: While these make no sense in single player, they should not cause any problems either.

Note that using properties or methods other than the ones above can yield unexpected results and some will simply do nothing. If you intend on starting a game in single player, but move it to multiplayer at a later stage, you might want to consider hosting a single player game instead.

Either set **PhotonNetwork.OfflineMode = false or** simply call **Connect()** to **stop offline mode**.

IPunPrefabPool

Instead of using the Resources system to instantiate prefabs across the network which is the default behavior, a better implementation is to create your own custom prefab pool and tell Photon to use it instead of the default by implementing the IPunPrefabPool interface :

```
// IPunPrefabPool defines an interface for object pooling, used with PhotonNetwork.Instantiate(...) and PhotonNetwork.Destroy(...)
public class NetworkPrefabPool : MonoBehaviour, IPunPrefabPool
{
    [SerializeField] private List<PhotonView> prefabs;
    private Dictionary<string, GameObject> poolDict;

    private void Awake() {
        poolDict = new Dictionary<string, GameObject>();
        foreach (PhotonView p in prefabs)
            poolDict.Add(p.name, p.gameObject);
    }

    public void Destroy(GameObject gameObject) => GameObject.Destroy(gameObject);

    public GameObject Instantiate(string prefabId, Vector3 position, Quaternion rotation) {
        if (!poolDict.ContainsKey(prefabId)) {
            Debug.LogError("Missing prefab '" + prefabId + "' in NetworkPrefabPool.");
            return null;
        }
        else {
            GameObject obj = poolDict[prefabId];
            GameObject instance = GameObject.Instantiate(obj, position, rotation) as GameObject;

            return instance;
        }
    }
}
```

```
PhotonNetwork.PrefabPool = new NetworkPrefabPool(); // Tell Photon to use our custom implementation instead
```

Tasks

Multiplayer Asteroids Game

Modify the following scripts as follows (you can search the files for the word “TODO”).

LobbyMainPanel.cs :

- Create a new room using the roomName entered by the user. For the RoomOptions, set MaxPlayers to the value entered by the user and PlayerTtl to 10000 ms.
- Join a random room when the "Join Random Room" button is clicked
- Leave the room when the "Leave Game" button is clicked
- When the "Login" button is clicked, set the LocalPlayer's NickName to playerName and then connect using the configured PhotonNetworkSettings asset
- When the "Start Game" button is clicked, close the current room by setting the PhotonNetwork.CurrentRoom.IsOpen variable, make the current room invisible by setting the PhotonNetwork.CurrentRoom.IsOpen variable, and finally load the level "Asteroids-GameScene" across all clients
- Observe and discuss the methods in the #region PUN CALLBACKS

AsteroidsGameManager.cs :

- When the game is started (in StartGame()), Instantiate the prefab with the name "Spaceship"
- When a specific player leaves the room, then they should also disconnect from the network. Use OnLeftRoom() callback.
- When a specific player disconnects, they should locally load the "Asteroids-LobbyScene" Scene. Use OnDisconnected() callback.
- When any player leaves the room, then all other clients should call the CheckEndOfGame() function. Use OnPlayerLeftRoom() callback.
- Observe and discuss the methods in the #region PUN CALLBACKS

PlayerOverviewPanel.cs :

- When any player leaves the room, then all other clients should destroy their respective PlayerOverviewEntry gameobject and remove the entry from the playerListEntries dictionary.
- Observe and discuss the methods in the #region PUN CALLBACKS

Spaceship.cs :

- Mark the following methods as RPC methods: DestroySpaceship(), Fire(...), RespawnSpaceship()
- Observe and discuss the methods in the #region PUN CALLBACKS

Thank You

Link to Lab Drive

https://drive.google.com/drive/folders/1jgFTWCJ5JD-JnIY7n_L7s2n2vA1F9nNX?usp=sharing

OR

https://tinyurl.com/ta-comp476-daniel

Other Links

https://docs.unity3d.com/2020.3/Documentation/Manual/index.html

https://docs.unity3d.com/2020.3/Documentation/Manual/ExecutionOrder.html

https://doc.photonengine.com/en-us/pun/current/getting-started/pun-intro

https://doc-api.photonengine.com/en/pun/v2/index.html