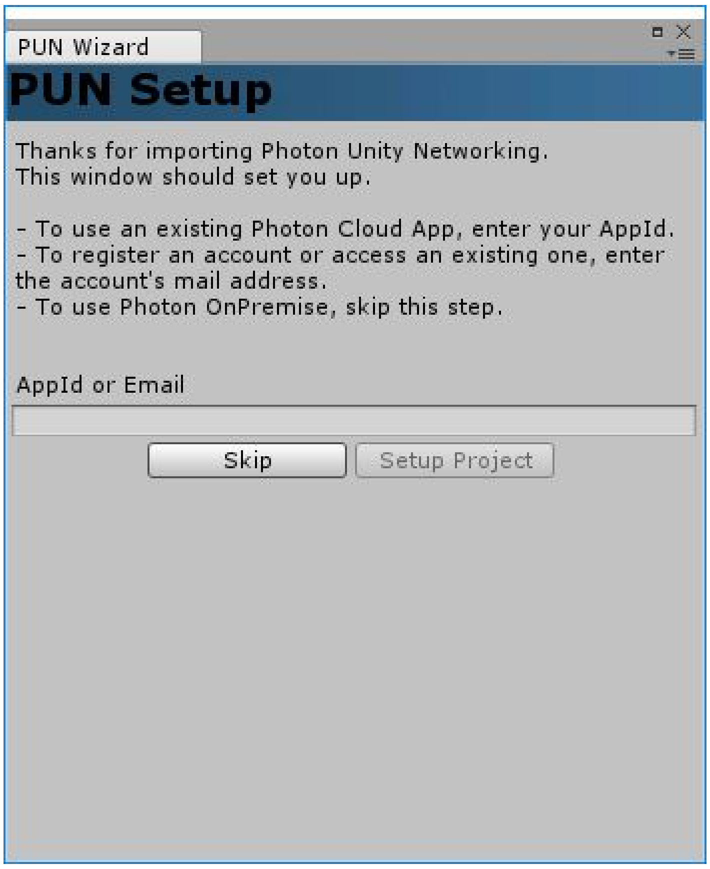
2D Online Platformer Tutorial Asset

Documentation

1.How to install and run the Asset

1) Sign up at https://www.photonengine.com/. Go to "Dashboard" - "Realtime" and memorize your App ID;
2) Press "Alt"+"P" in Unity - Setup Project or Window - Photon Unity Networking - PUN Wizard - Setup Project;
3) After that, you will be asked about your App ID, that you received in the first step.



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

2.How to use the asset

Controls :
1) left mouse button – fire weapons
2) left and right arrows – movement, down arrow – duck, up arrow – jump; the same is true for the keys: w, s, a, d

The first player, who connects to Photon, creates the room. All other players connect to him. (*ConnectRoom script*). If this player dies or if there is one alive player in the room, or if everyone dies (the last two players killed each other simultaneously), then all players again create prefab of the player player (*Remover script*). Nobody leaves the room

3.Short description of the tutorial

In this tutorial I will explain the concept of network in an abstract way (without detailed definitions and descriptions of the functions, variables, PUN classes) so that you understand the basis - how to add network logic to offline logic using a defined pattern of rules, which I offer you. The package, which I remade from single player to online, is 2D Platformer (Unity Essentials))
I deliberately simplified this pack, so that it is simpler to take down the added network code, so that there is nothing excess. Gameplay of this game – player vs. player on the small field (roughly speaking, the simplified gameplay of Duck Game). I deleted mobs, points, hp, HUD bombs: weapons are displayed in player hands, and weapon crates display the weapon, which player will take. There is no need for the camera movement script on such a small playing field.
In this tutorial I will stop on the main and difficult point - how to transfer the gameplay, the behavior of the player and their contacts with each other.

3.Four rules of the network (Photon Cloud)

The four rules for creation of online in the "arcade" genre on Photon Cloud.

Rule 1 - "I send the behaviour not only to myself, but also to other players".

All beauty of Photon Cloud is that it has a high degree of abstraction. We don't have to think about when I send a message of my behaviour to other players, first I will send the behaviour to the server, and then the server sends it to all other players. In Photon Cloud, we are introducing the transmission of messages in the player-to-player form. If you see that I address *PhotonNetwork*, almost in every script you see it I will be transferring data to other players. I need to transfer how I create player prefab to other players in this room. By a regular Unity Instantiate I will not be able to transfer everyone that I created a prefab: I will transfer it only to myself, but other players will not see it. In order to make them see the prefab creation, I need to create it through the *PhotonNetwork* class(*PhotonNetwork.Instantiate*); the same is true for deletion and sending the method to other players – *PunRPC*.

Rule 2 – “is this mine or yours?”

Whom do I control, and whom does my opponent control? Which player prefab? Who is the owner of this prefab?

There are two players in the room. As a result, we will enter the *PlayerControl script* of each other. And take away movement from each other. There are two ways to make this not happen -
1) Create a prefab with disabled scripts and during its creation we enable the script. Another player sees this script as disabled and does not enter *Update*, *FixedUpdate* of this script. However, it is important to understand if the script contains behaviours (*PunRPC*), which need to be transferred to other players, the player receiving this method can later refer to zero variables in it, therefore we can't initialize them in *Start*. Only in *Awake*. *Start* is not called for the disabled scripts.

2) You are in a room and another players enters it. You tell him: “This is my property, don't enter it; I created it and I control it!” - *if (!photonView.isMine) return;* when he enters the script of your object, he will always get into the *return*.

And here we again return to the useful features of Photon Cloud abstraction - distribution of IDs to objects without our knowledge, without direct intervention. Even when we change the owner of an object, we simply take the IDs of both ones and replace them one by another. There is no direct wording here – “change (assign) the id from one to another”.
https://doc.photonengine.com/en-us/pun/current/manuals-and-demos/ownership-transfer

Rule 3 - Time paradox.

The one, who transfers the data to others, is located in the future: that is why the ones, who receive data from him, see his past - they receive data with a delay (for example, the data on his movements (position points)). This basis totally changes some of the typical game logic decision without network, which can be solved by a several ways. For example, let us view the collisions topic in this project. Collision of a bullet with a player.

Two options can be used to implement this without network through collisions of OnTriggerEnter2D

1) the affected player accepts the bullet in his player body
2) the bullets accepts the player body into it.
Which version is the correct and workable one? Both. However, there are important moments in network logic, which you should be aware of.
Owner of the bullet will quicker find this collision than the one, who suffered from the bullet i.e. the owner of the bullet should only send data to the affected party ("I shot you - you should die").
In case owner of the bullet finds this collision quicker, the first option is no longer suitable, because there are bullets in our script - bullet deletes itself after it collides with someone (due to Photon rules - only owner of the bullet deletes his bullet). The player, whom we shoot, sometimes will not have time to get this collision, because he sees the past and this collision

did not yet happen for him. On the other hand, the shooting player had a collision and he deletes the bullet earlier than it meets for the affected party, as he receives data over the network (sees the movement of the bullet in past).

Second example of time paradox. Weapon with damage radius activation behaviour script – *Scripts/Weapon/Bomb* . Logic suggests that in order to perform this script easier, we should send the *PunRPC Explode* function not to all players, but only to *MasterClient* (there will be less activations, less foreach iterations). However, this is not suitable for the general example of the weapon with damage radius activation. For example, if we use this script to implement RPG that shoots mini-grenades, which immediately explode from any collision with anything, and if during this collision we will ask *MasterClient* to explode the grenade and will not explode it by ourselves, explosion will be with a delay. *MasterClient* – this is just a special player with additional capabilities, who has rights similar to any other players in the room (he has the smallest id and may spawn scene objects, which are not deleted after his exit, unlike spawn of the regular *PhotonNetwork.Instantiate*).

Example of time paradox in my other asset store package - http://u3d.as/ugv

https://www.youtube.com/watch?v=Grf2i3V5tGA

In this video, you can observe time paradox on 0:08. I deliberately did not remove it, so that we could review it in details. What happens here. I am moving in the future, and the opponent increases speed and rushes to me (according to the rules of the game, when a player glows blue (in the rush mode), all those who have touched him not in the rush mode, die. But he was flying in the future and he has already flew past me, when I encountered him. When I bump up against him, he is already behind me (and during the rapid movement he did not clash with me). If in my player body we write that we find collisions with other players in the dash mode, then I would have died very often from touching any possible things. It is better to dodge from death "in an awesome way" one more time than to not understand what you died from, from the touch you did not see.

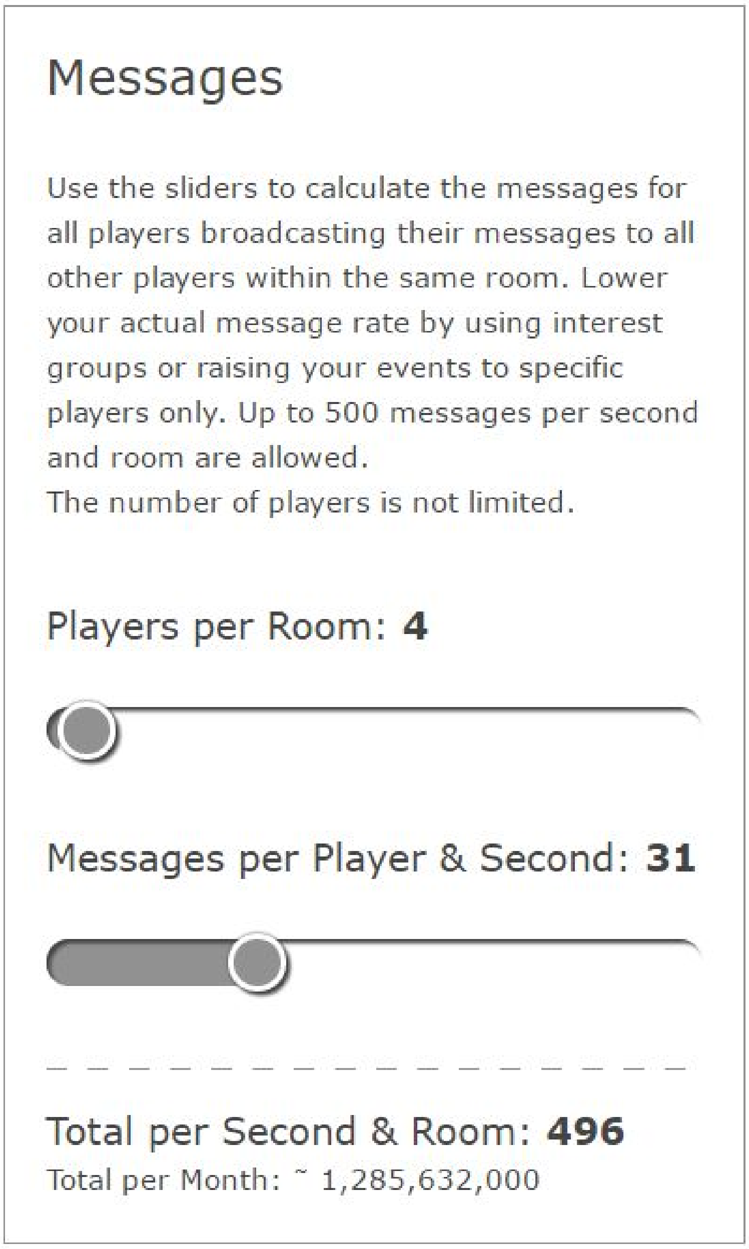
Rule 4 - Messages limit.

One of the main network rules in the arcade game genre: do not network code where it is not required. If you will receive too many messages from players, message buffer will be quickly flooded, a queue is formed and you will receive these messages with a delay - lags (jerky movement (missing or resending position points), later deletion, later spawn, later triggering of animation). Imagine a later spawn of a rocket launcher with jerky movement! This completely kills game dynamics! There is never excess network traffic for a dynamic arcade game. There is always not enough of it in Photon Cloud!

No need to synchronize ducks, buses - let them spawn for each player in their own way. If players could shoot the ducks and get loot out of them, then you had to synchronize the movements of these ducks on the network for all players.

The limit of the number of messages without a queue in the room equals to 500 per second.

https://www.photonengine.com/en-US/PUN/Pricing#plan-20



Messages

Use the sliders to calculate the messages for all players broadcasting their messages to all other players within the same room. Lower your actual message rate by using interest groups or raising your events to specific players only. Up to 500 messages per second and room are allowed.
The number of players is not limited.

Players per Room: **4**

Messages per Player & Second: **31**

Total per Second & Room: **496**
Total per Month: ~ 1,285,632,000

Let us look at how to use this very important data table! From this table we get the following: for the four players in the same room, each player may send other players a maximum of 31 messages without a delay. Even in my previous package (http://u3d.as/ugv) with simple gameplay, you can easily exceed this limit. There is a bonus of triple bullets. If we transfer movement of each of the bullets on the network, in total we get the number that exceeds limit at the moment, when all four players will move and will fire triple bullets at least once. (10*16 + 30*16 = 160+480=640>500).

Any transfer of smooth movement is the transfer of about 10 movement positions (10 messages) per second. *OnPhotonSerializeView*. 30 messages - transfer of three bullets movements. Why do we multiply by 16 and not by 4? In each second we send our 10 movement positions to four players and get 10 movement positions from three other players (3*40+1*40=120+40=160). Instead of sending three bullets, we send the one bullet, which has children (two bullets). In total, we send 10 messages instead of 30. And we get (10*16 + 10*16 = 160+160=320<500). In this point, we cut network traffic in the room BY HALF (640-320)! Well, I think you have already understood what minimum players limit should be in the room, where it is important to transfer movement without jerking and where there is rapid fire rate of the bullets, approximately 4-5 . As a result, the solution is to create dynamic games on a small field. In fact, you don't have to transmit bullet movements on the network, you have to transmit only the collisions. I have just shown an example of how easy it is to exceed the messages limit in Photon Cloud.

There is a reason why the fourth rule goes after the third one. These two rules complement each other in some way. Now let me clarify what I mean. Reducing the network traffic we get rid of the "excess" synchronization, but with little practice in network logic programming, by paying too much attention to this rule, you can by a mistake (or inadvertently) put the mandatory synchronization into the excess one and remove it. We don't need to transfer the missile movement points by the network, but we need to transfer the transition of the flying bomb. It seems that implementation is the same – spawn the bomb (missile) with a specified rotation by using the third argument of *PhotonNetwork.Instantiate* function. From the rotation in the *Awake* (*Start*) we define where the bomb (missile) will fly. And each player specified the same speed of the missile flight, the same strength of push for the bomb. Each time for those watching there will be the same spawn point and flight direction, but the bomb does not explode after touching! It explodes after a specified timing. Here players get the position differences in terms of a bomb explosion point. If I jumped and player threw a bomb at the same time, discrepancy may occur in this point. For one player this bomb flew past the player, and for other player he caught it with his body. At this point, indeed, the bomb explosion point will be in different locations for players. And this may happen. Without transfer of the bomb movement by the network.

4.Weapons implementation rules.

A new weapon crate needs to be added to the scene object – *pickupManager* into the *WeaponBox* variable of the *PickupSpawner script*. There should be the new last component on the new weapon crate (turned off script, which is inherited from Weapon).

Of course, you should never leave *spawn_point* equal to zero. For example, with this indicator of *spawn_point*, after throwing the bomb will be attached to the player and will not fly anywhere, it will remain at player's feet. At the same time, if you assign too low *spawn_point*, then the player will jump when crouching, he will touch the floor with his weapon during a throw.

Weapon behaviour classes, which are inherited from Weapon, should use *Update* for any non-animated weapon activations, and should use *FixedUpdate* for the animated ones (this is needed in order for Photon to transfer the animation activation *Trigger* in a proper way (it is too fast))

Of course, if you create the weapon with an entirely new behaviour (for example, a melee weapon), then you need to improve the base class *Weapon*.

I advise you to watch the second part of my tutorial. If the first part of the tutorial is theoretical, the second one is more specific, practical. The second part contains the description of the basic most frequently used capabilities of Photon Cloud and the discussion of an important errors, yellow warnings that may arise during practice on Photon Cloud with a described solutions(http://u3d.as/ugv). If you do not want to pay, please read the documentation on the official Photon Cloud website (exit games), download the exit games packs from the asset store. However, in this case nobody will tell you about the errors, which arise in practice =)