Net Positive Tutorial
Building a Networked Game

What we'll be doing

We're going to make a ultra-simple MMO (moderately multiplayer online) dungeon crawler. Where they can run around with their friends, attacking, enemies, each other.

Before we start

Please make sure you have Unity 2017 installed

This document is contains both instructions, and explanations. Steps for you to perform start with a **(!)**, and are **highlighted in purple**.

We will start from empty and incomplete scripts, and incrementally build up functionality. It's OK if you are new to coding in Unity. There will be links to live google docs for each relevant script in the project for you to paste into your project.

The existing Photon documentation is quite good, and my tutorial loosely follows their basic tutorial:
https://doc.photonengine.com/en/pun/current/tutorials/pun-basics-tutorial/intro

Photon Setup

Create Photon Account

(!) Register for a Photon account at

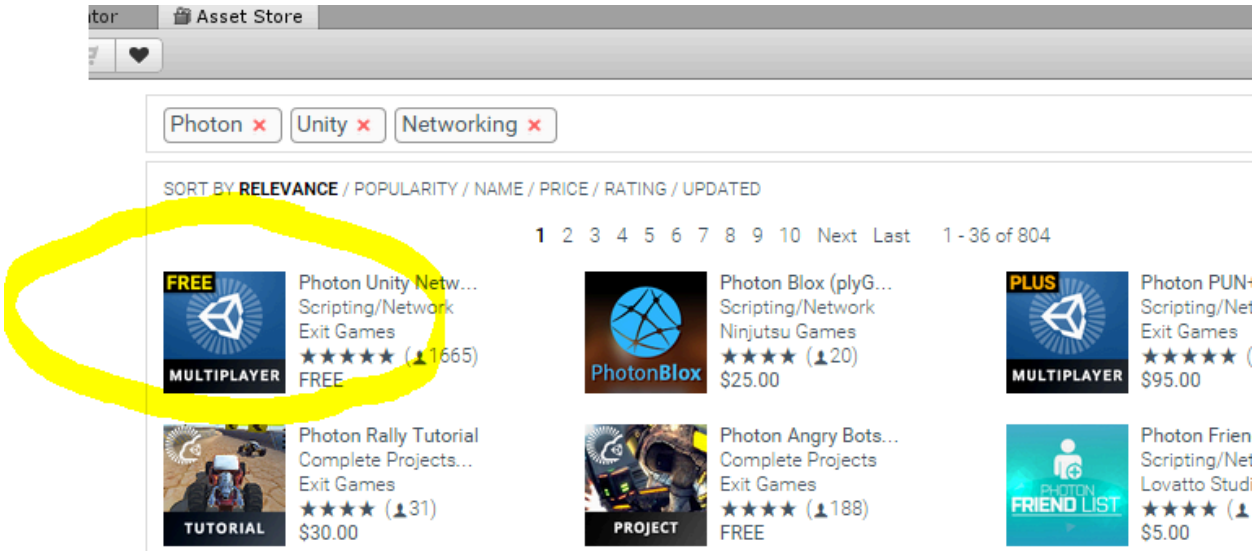
https://www.photonengine.com/en/Account/SignIn

Download Template Project

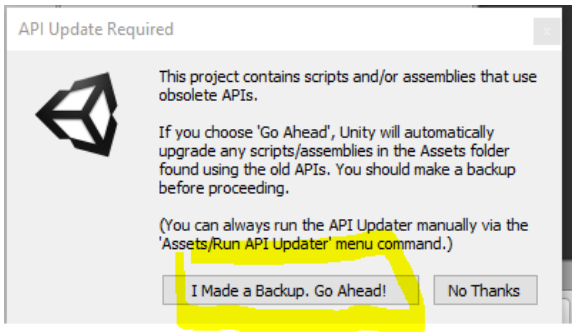
(!) download the template project

(!) unzip and open with Unity

(!) Download and import '***Photon Unity Networking'*** from the Asset Store.



It's this free one



Hit this button if it appears

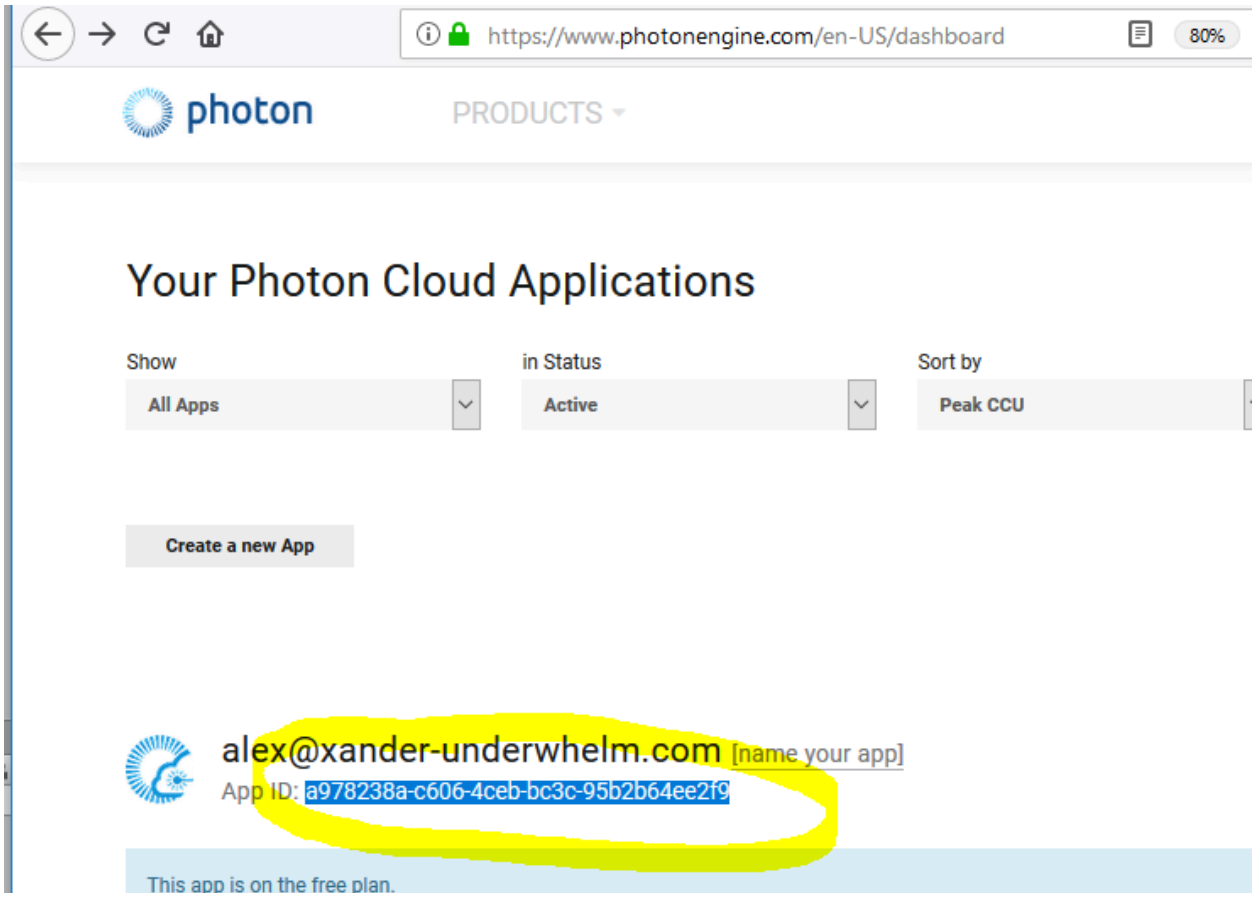
Link Project with your Photon Account

(!) Sign in to your photon account

https://www.photonengine.com/en-US/Photon

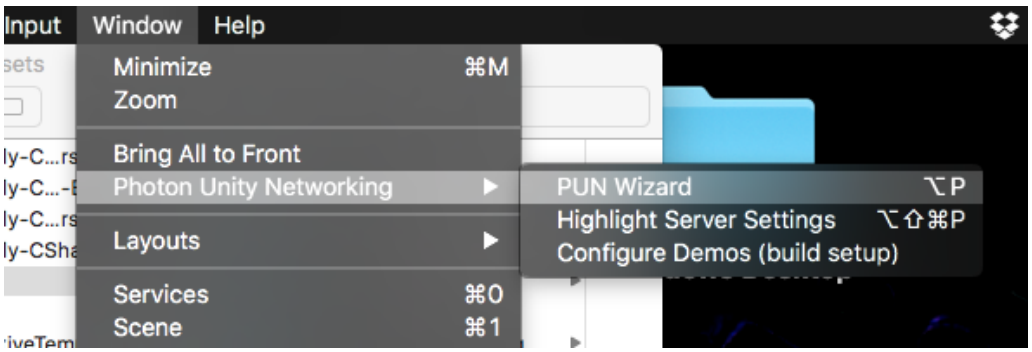
It should automatically take you to your dashboard

(!) Copy your **'App ID'** from the dashboard...

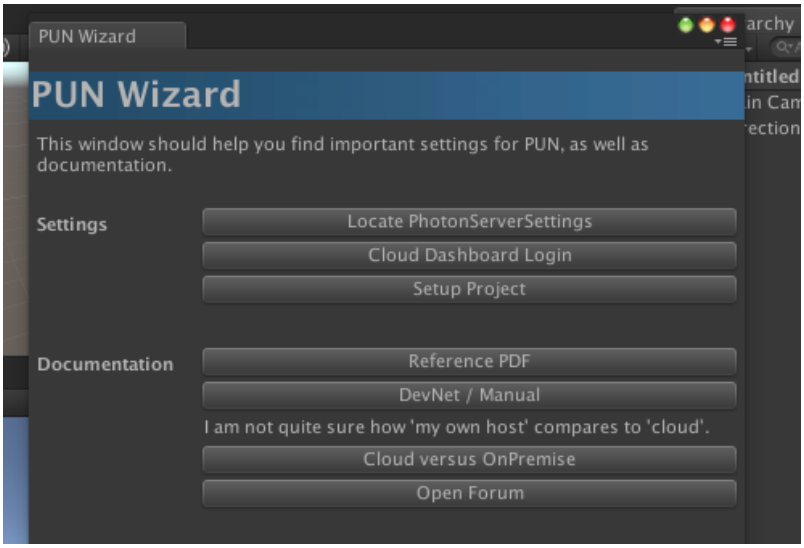


Here's the Dashboard, with the App ID circled

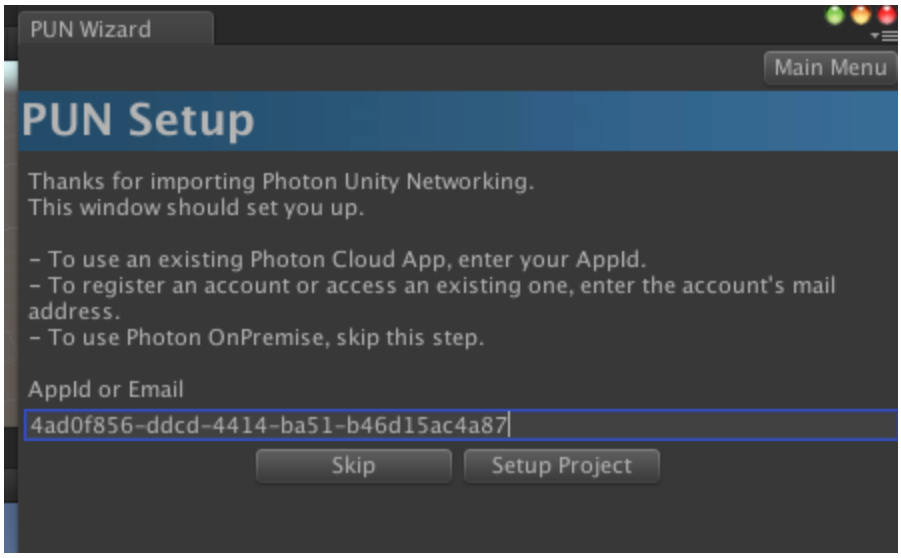
(!) Go to **Window->Photon Unity Networking->PUN Wizard**



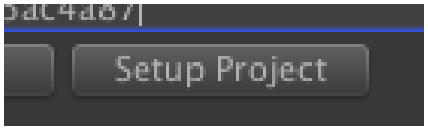
(!) Hit the **‘Setup Project’** button



(!) Paste in your app-ID

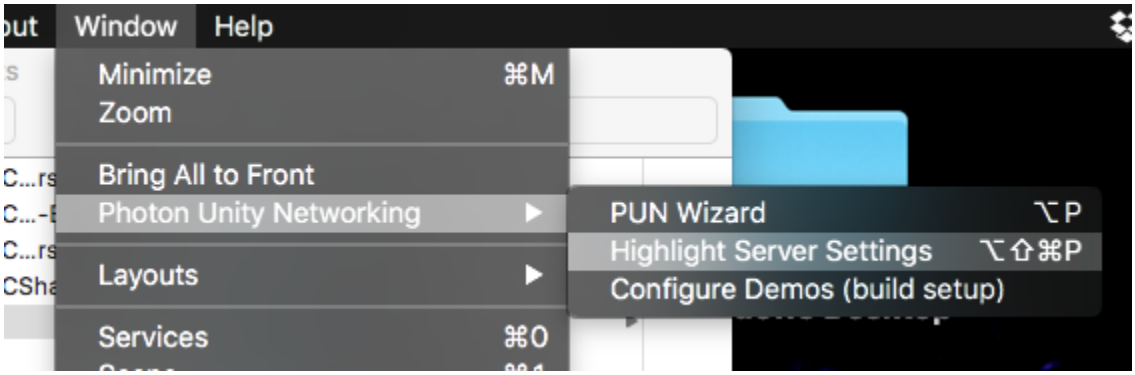


(!) Hit ‘Setup Project’

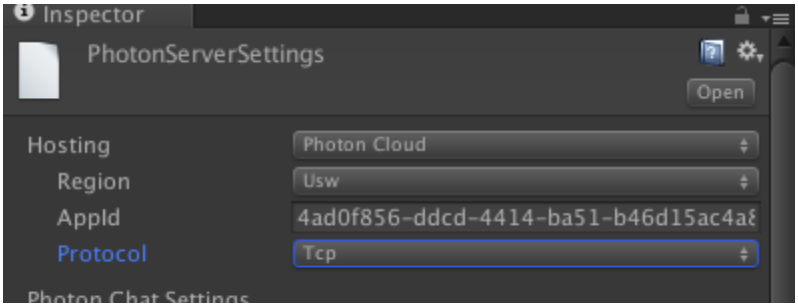


Set up the Server and Demos

(!) Go to **Window->Photon Unity Networking->Highlight Server Settings**

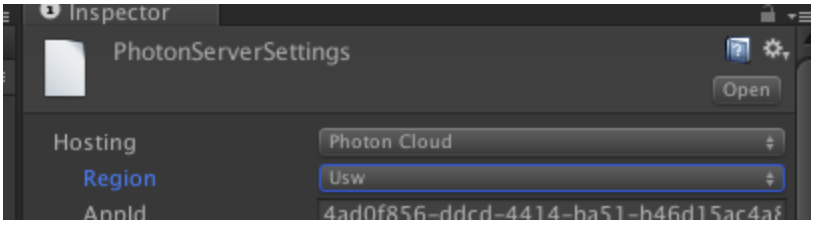


(!) **IMPORTANT!** In the Inspector, set **'Protocol'** to **'Tcp'**



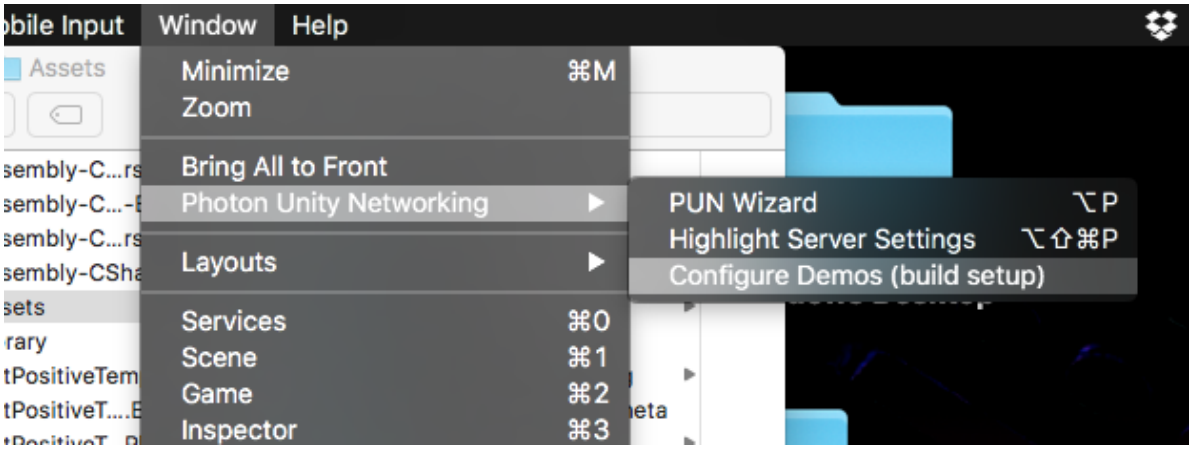
It won't work if you miss this step!

(!) Also change **Region** to ***'Usw'*.**



Go through a Photon server on the west coast instead of Europe

(!) Go to **Window->Photon Unity Networking->Configure Demos (build setup)**

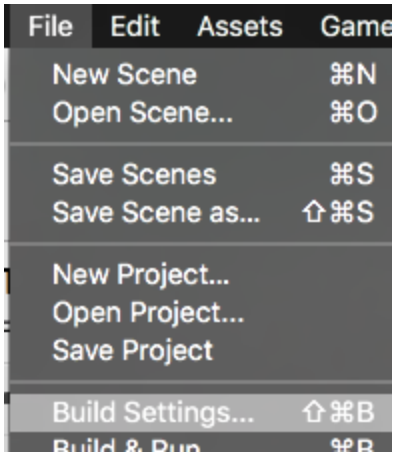


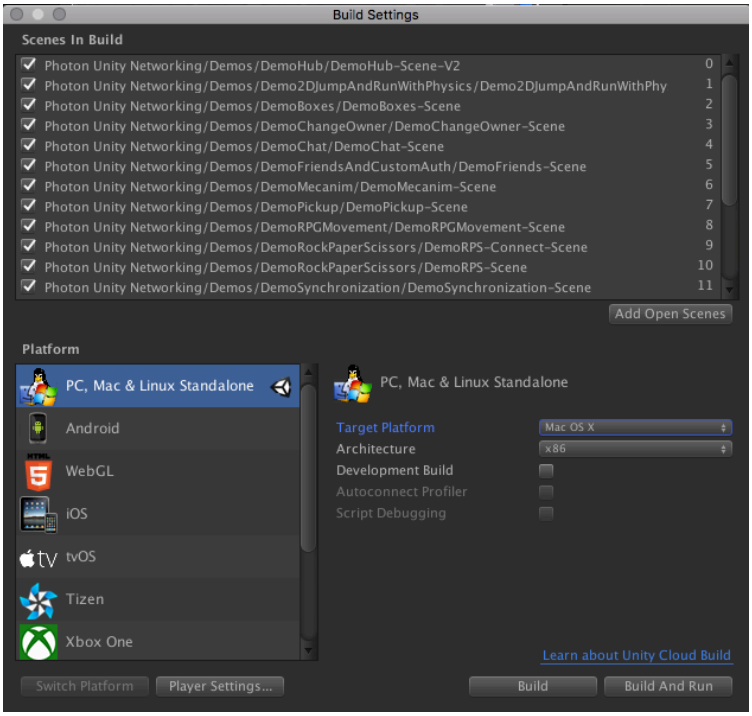
Now we're ready to try Photon's built in demos, and start developing for ourselves

Run the built-in demos, and Verify Photon is working

We're building a multiplayer game, so we need to run multiple copies of our game. One instance of the game can be in the editor. We make a build and run that 1 or more times to add additional players.

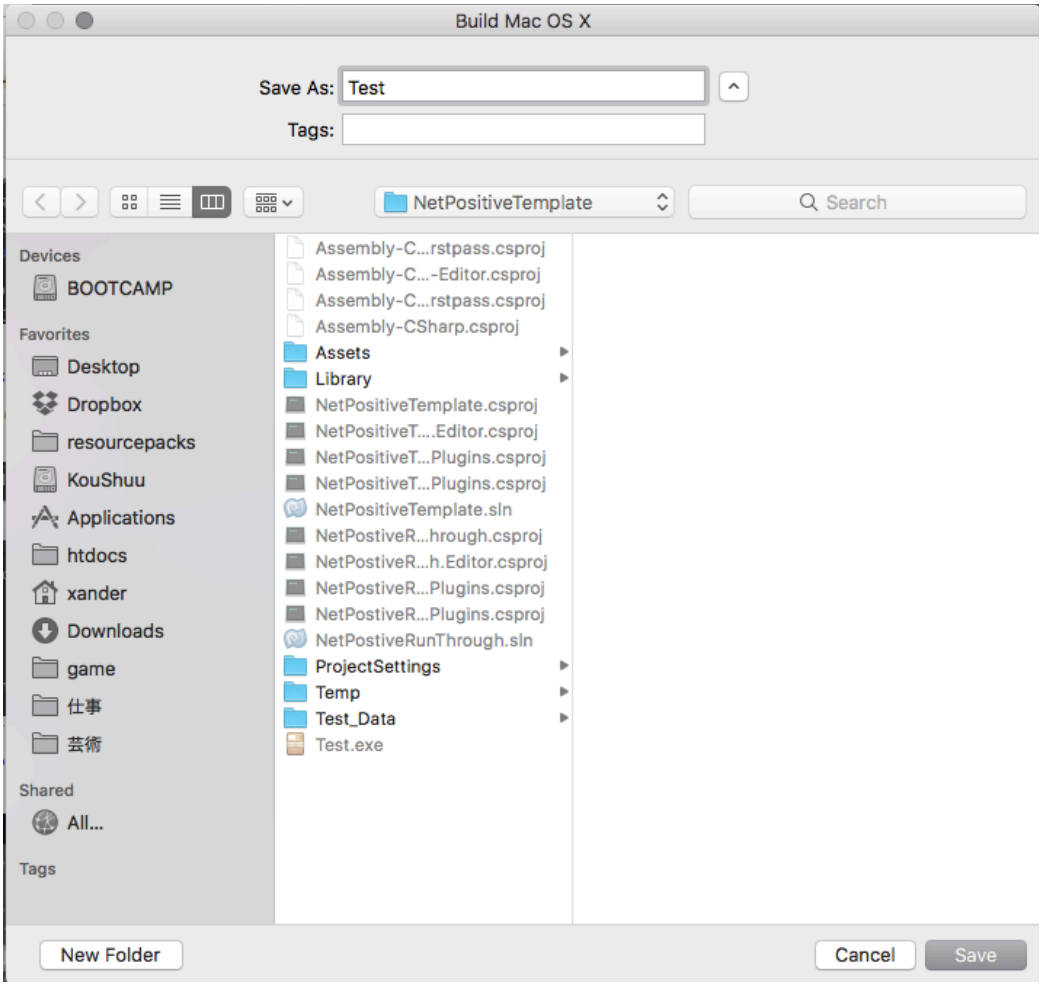
(!) Go to **'File->Build Settings…'**





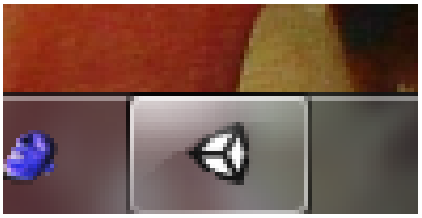
(it should look like this, with a bunch of scenes added)

(!) Build a Mac, or PC version Giving it the name “test”



I’m working on Mac here, so I’m making a Mac build.

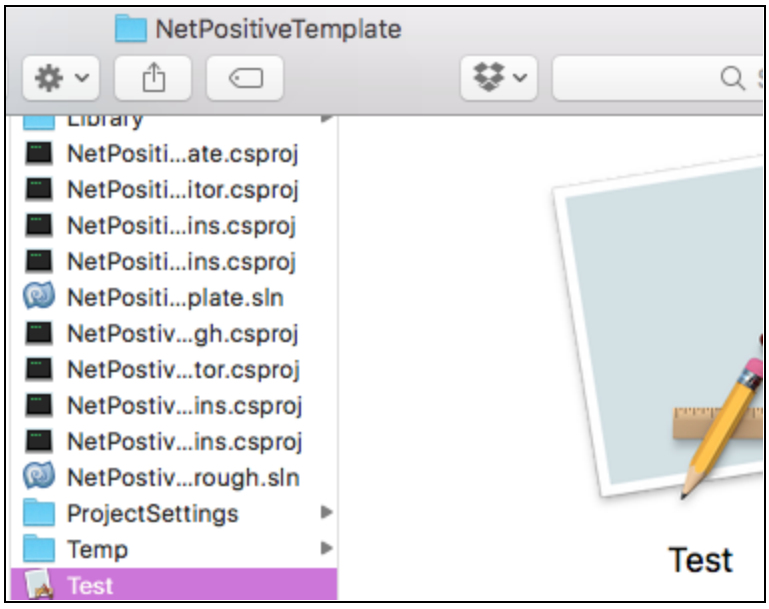
It’s also a good idea to put the EXE in your taskbar (Windows), or on the dock (Mac). That way you can quickly start a second copy of the game.



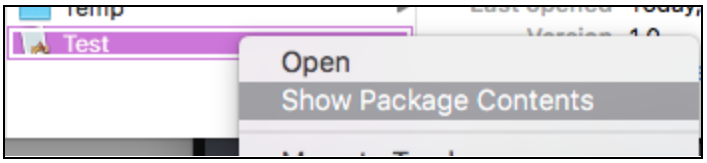
Here’s the build on the OSX dock, and the Windows taskbar respectively

Important note for Mac Users:

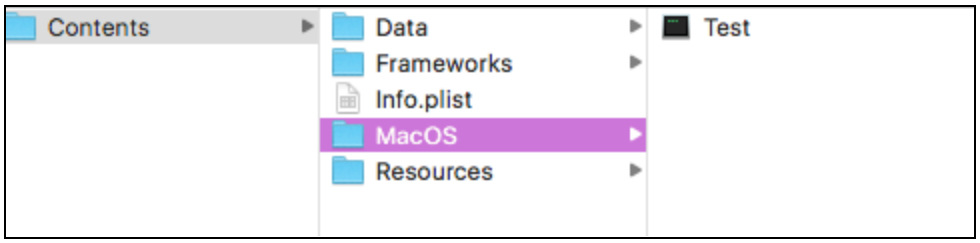
If you want to run multiple copies of your built game (to test more than 2 players), need to run



Once your build is finished

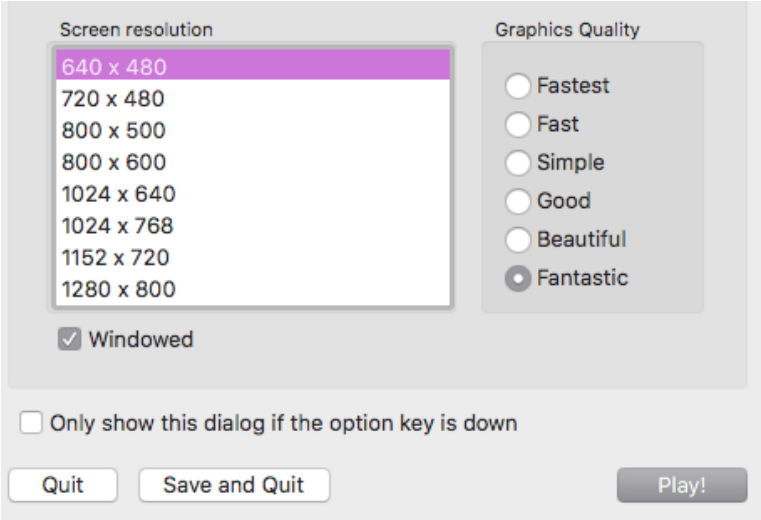


2-finger click on the build and choose 'Show Package Contents'



Make an Alias of the file with the Black Icon in ***Contents/MacOS*****,** or put it on the right side of your dock:

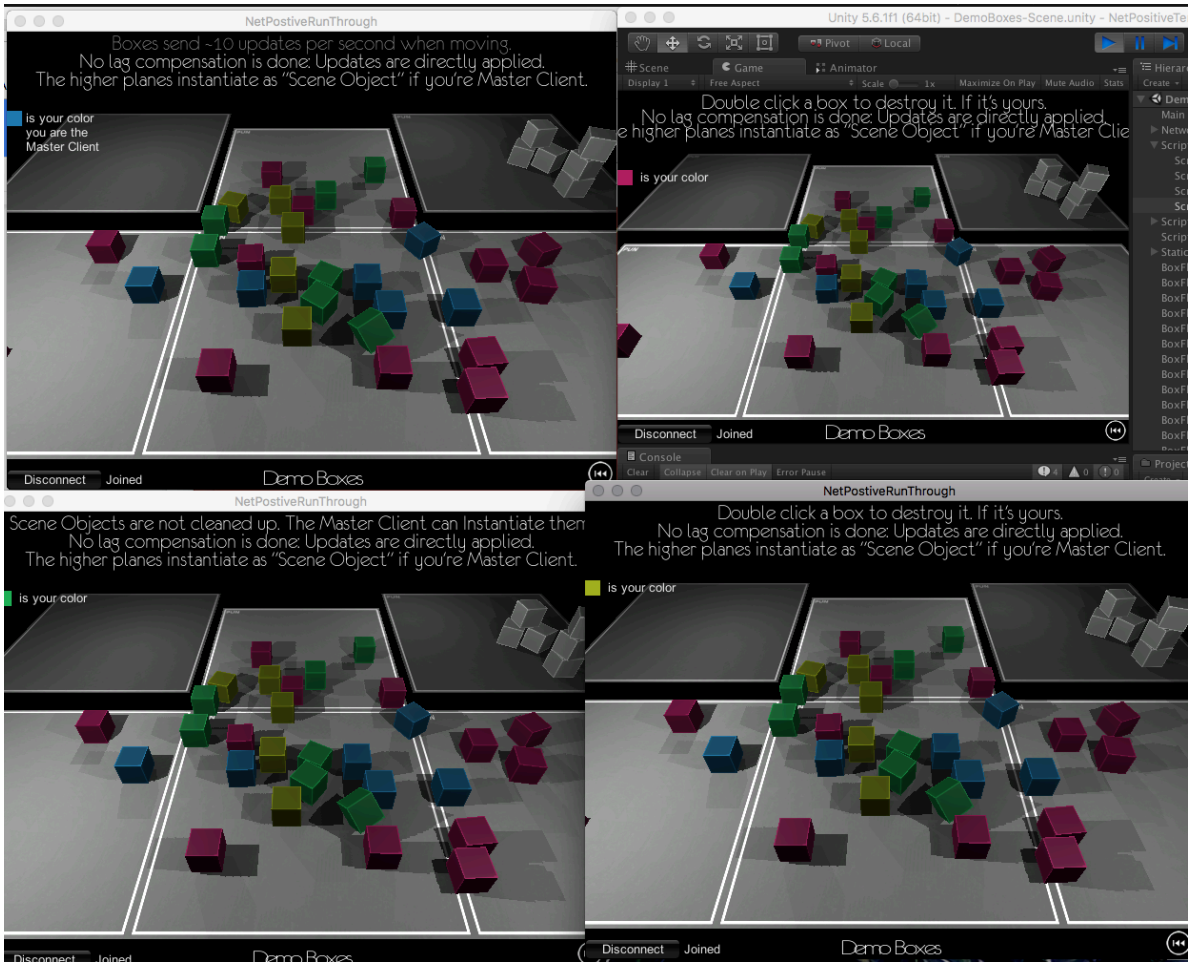
(!) Once it's finished building, run the built game.



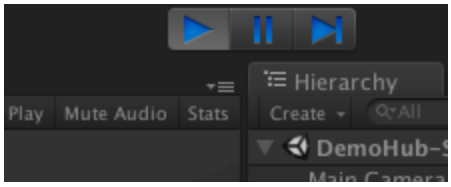
(!) In the resolution dialog, check **"windowed"**, and choose a small resolution. (like, 640x480)

DON'T run the game full screen, or at a high resolution. We must run one copy of the game for each player to test the networking (one copy of the game can be the editor), and we really need to be able to see all the players' game screens.

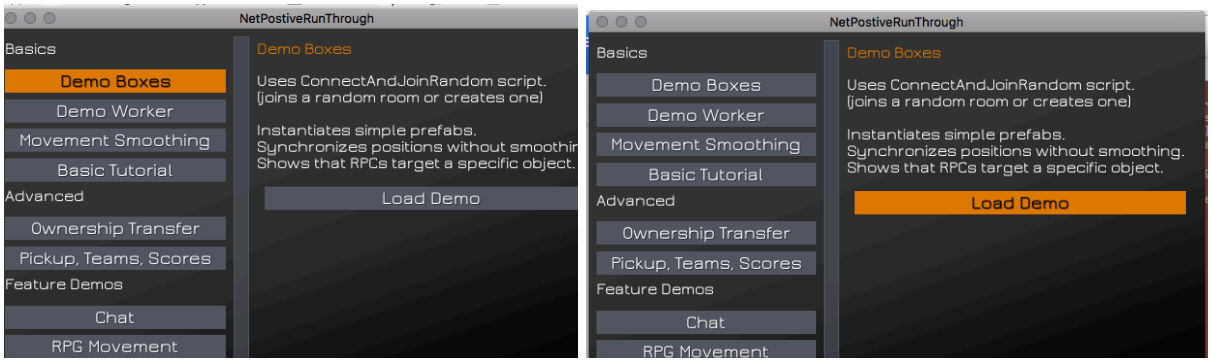
E.g. Below, we're running 3 copies of the demo, and 1 copy in the editor



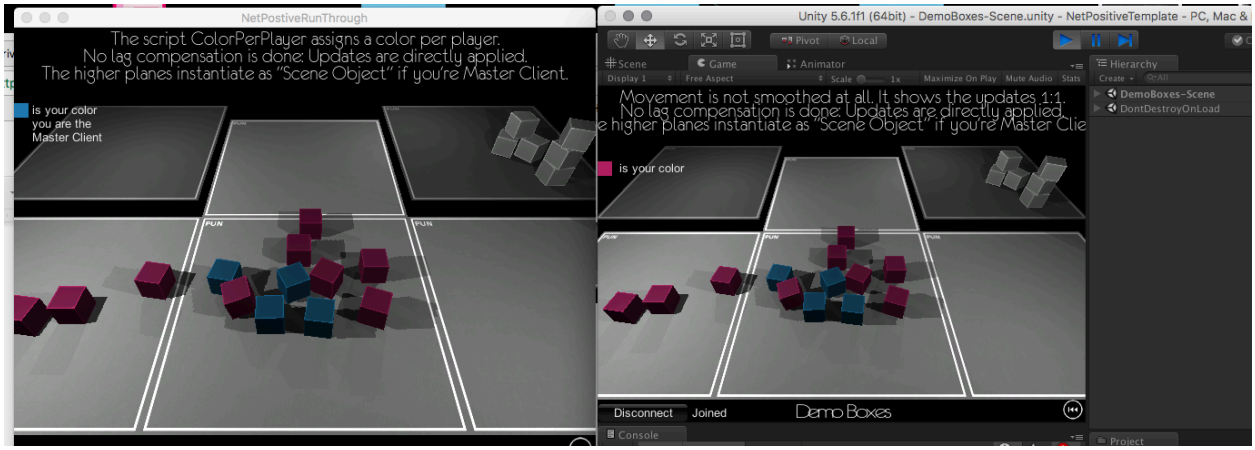
(!) Once the build is running, also run the scene in the editor.



(!) Start, the '**Demo Boxes**' in both the build, and the editor



Choose demo Boxes, then Load Demo in both the build, and the editor

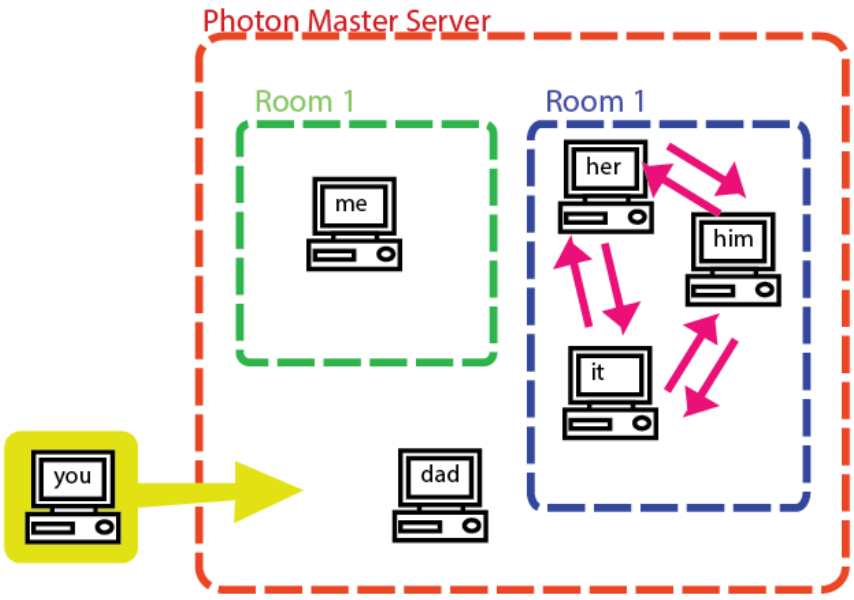


You should be able to click and see boxes appear in both windows

Making a new game, from scratch(ish)

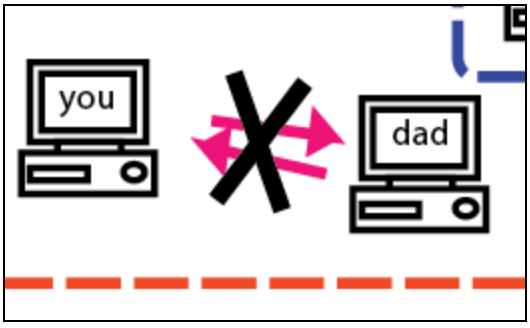
Connecting to Photon

To participate in a photon game, first must do 2 connection steps. a new player must first connect to the Photon *MASTER SERVER*.

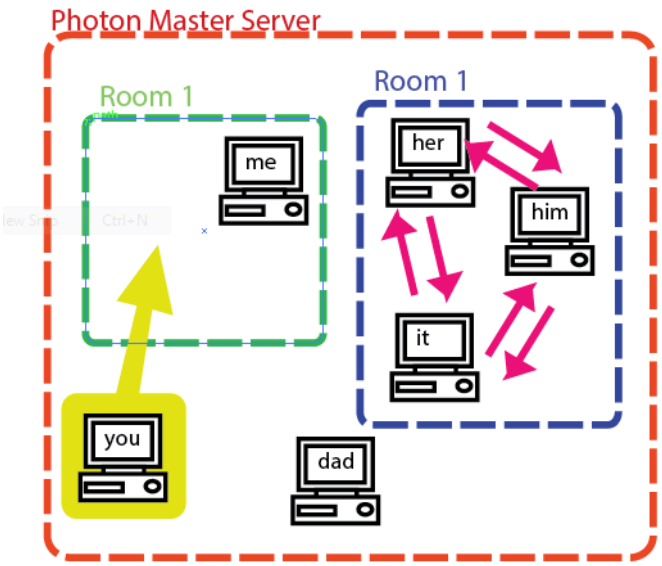


1st, you must connect to the Photon Master Server

Once connected to the master server, a player must join a *ROOM*.
In our project, there will just be 1 room that all players join.

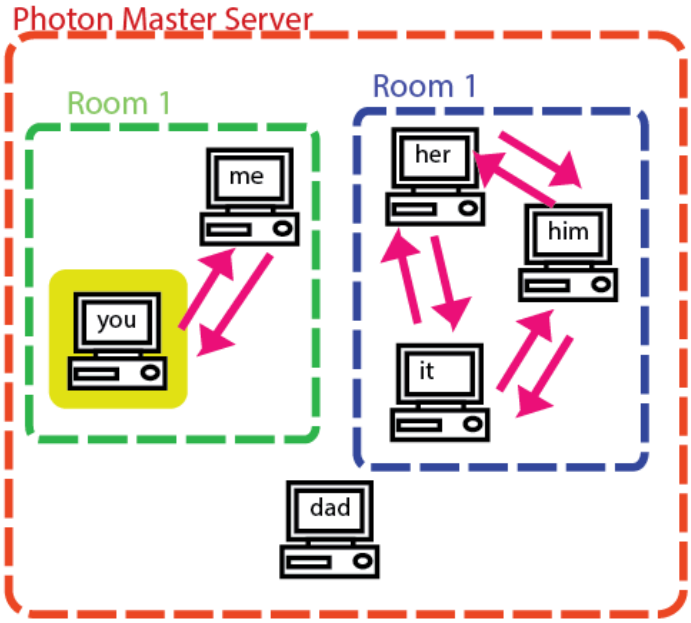


Can't talk to dad unless we're both in the same room (Both you and dad are not in a room yet!)



Once connect to the master server, if you want to actually communicate/play a game with other players, you must enter a room

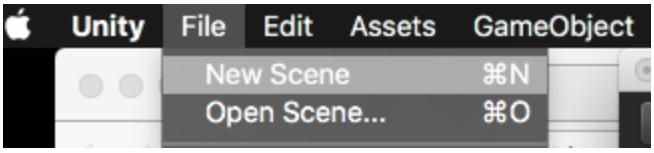
ROOMS are like individual matches of a game, and players can only send and receive messages to other players in the same *ROOM***.* Player can freely create, join, and leave rooms. Photon also provides functionality to restrict who can enter a room, and when they can enter, or to find a random available room to join. (Think, random matchmaking). In an MMO, there might be individual rooms individual dungeons or towns.



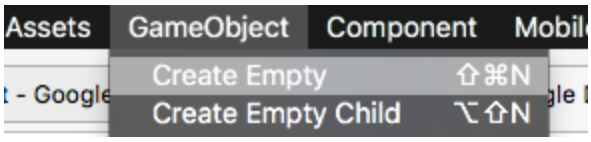
NOTE: Photon's own tutorial has more info about auto-joining a free room & matchmaking:
https://doc.photonengine.com/en/pun/current/tutorials/pun-basics-tutorial/intro

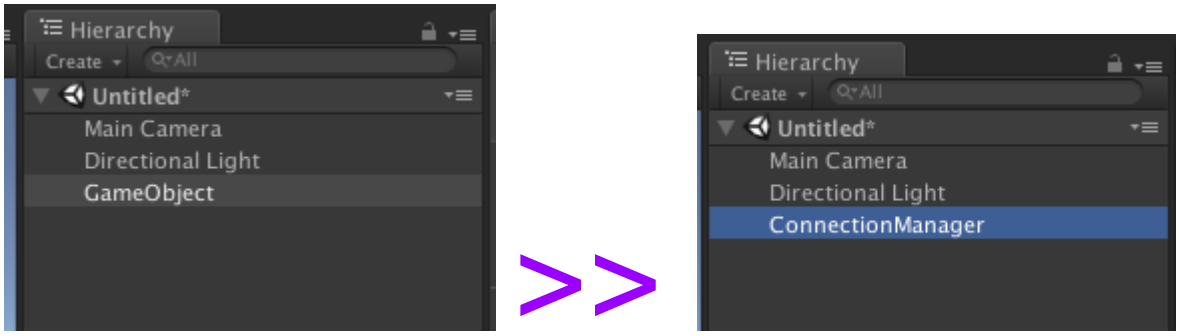
Our first Photon Game

(!) Create a new scene

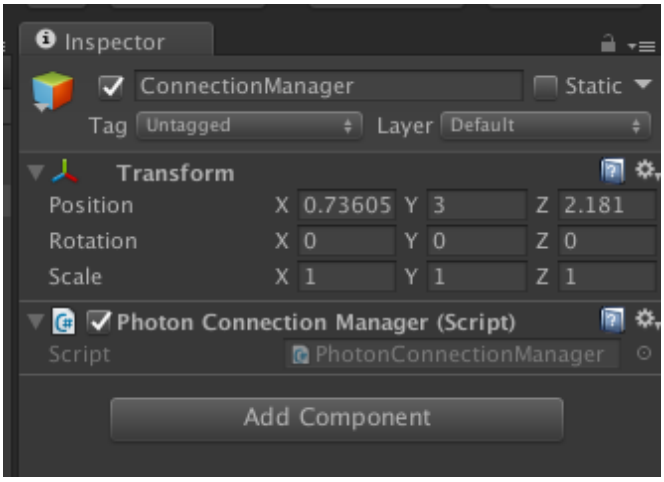


(!) Create an empty gameobject, and name it ***ConnectionManager***

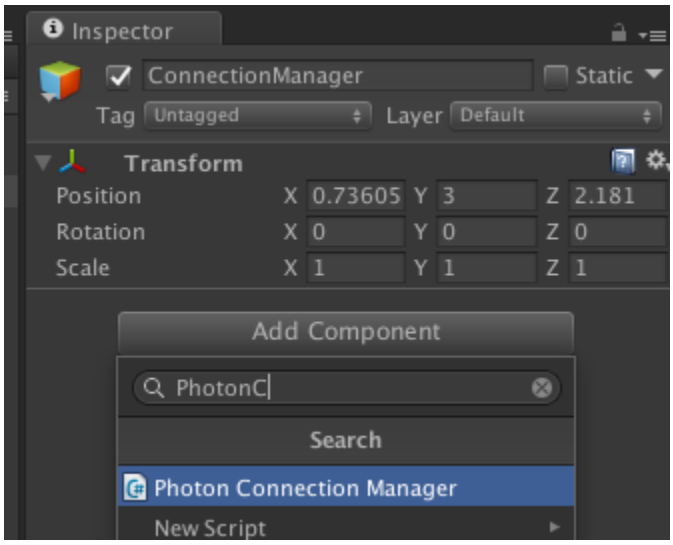




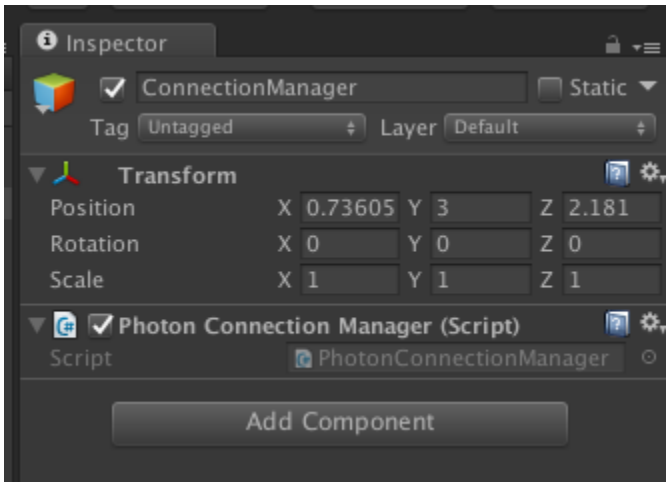
(!) Add a ConnectionManagerIncomplete script to the ***ConnectionManager*** *object*



...

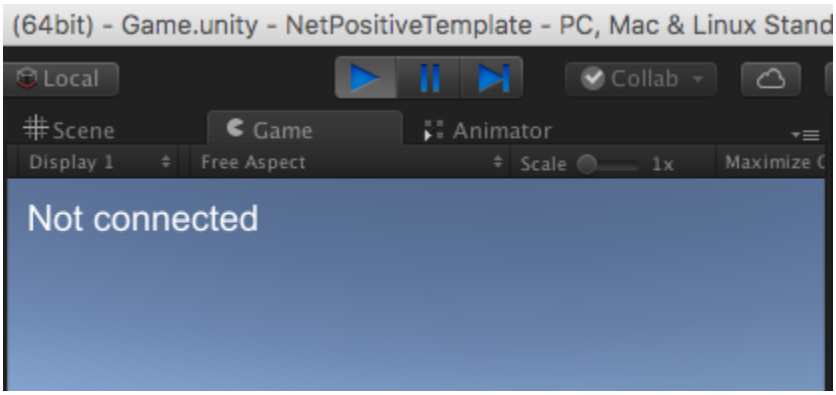

Inspector
ConnectionManager
Static
Tag
Untagged
Layer
Default
Transform
Position
X 0.73605
Y 3
Z 2.181
Rotation
X 0
Y 0
Z 0
Scale
X 1
Y 1
Z 1
Add Component
PhotonC
Search
Photon Connection Manager
New Script

...


Inspector
ConnectionManager
Static
Tag
Untagged
Layer
Default
Transform
Position
X 0.73605
Y 3
Z 2.181
Rotation
X 0
Y 0
Z 0
Scale
X 1
Y 1
Z 1
Photon Connection Manager (Script)
Script
PhotonConnectionManager
Add Component

...

(!) Run the game in the editor



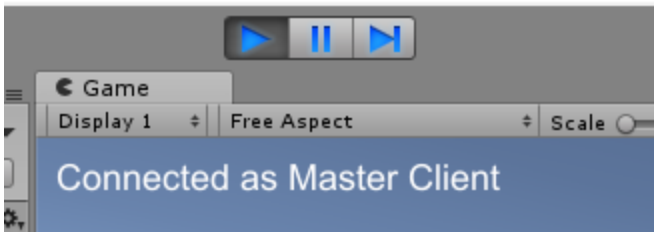
I gave you an incomplete script! we'll fix it below.

(!) **CODE-ALONG** : Update your script to match PhotonConnectionManagerIncomplete

Warning! If you're skipping ahead, the above won't be ready yet.
As a hint, you need to connect to the master server, then join/create a room

<Is this where this should go?>
Here's more information about callbacks like *OnConnectedToMaster()*, and *OnJoinRoom()*
https://doc-api.photonengine.com/en/pun/current/interface_i_pun_callbacks.html

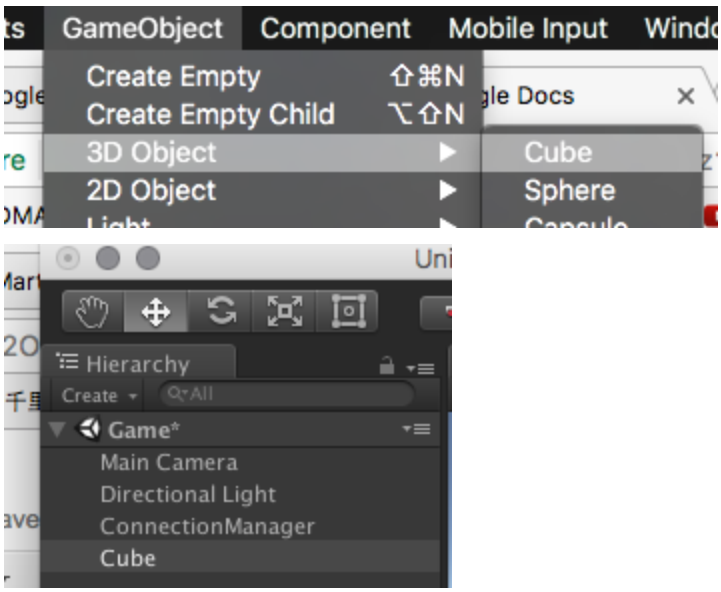
(!) Once the above code-along is done, Run the scene, and verify that the game view shows you are **'Connected as Master Client'**

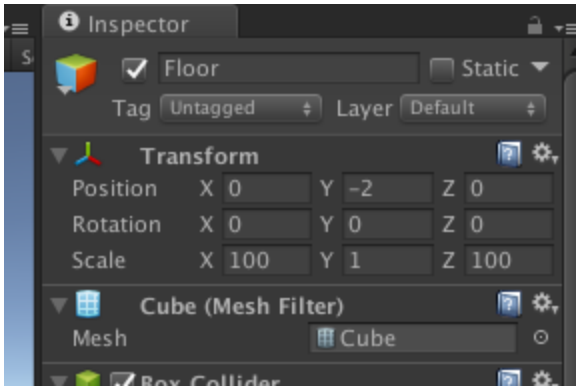


You should see this,

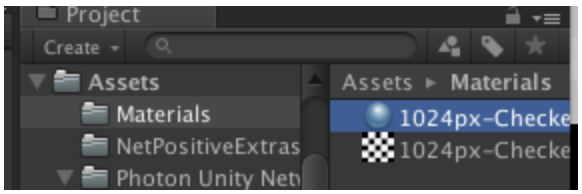
Quickly Add a floor

(!) Create a big cube for the floor, put on a checkerboard. Call it "**floor**".





I Recommend you make its scale (100,1,100) and its position it at (0,-2,0)

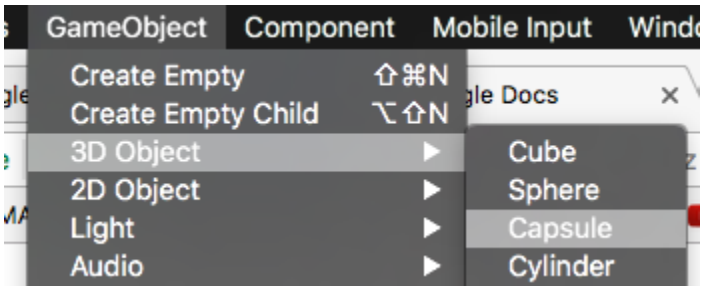


I also recommend you put a texture on it

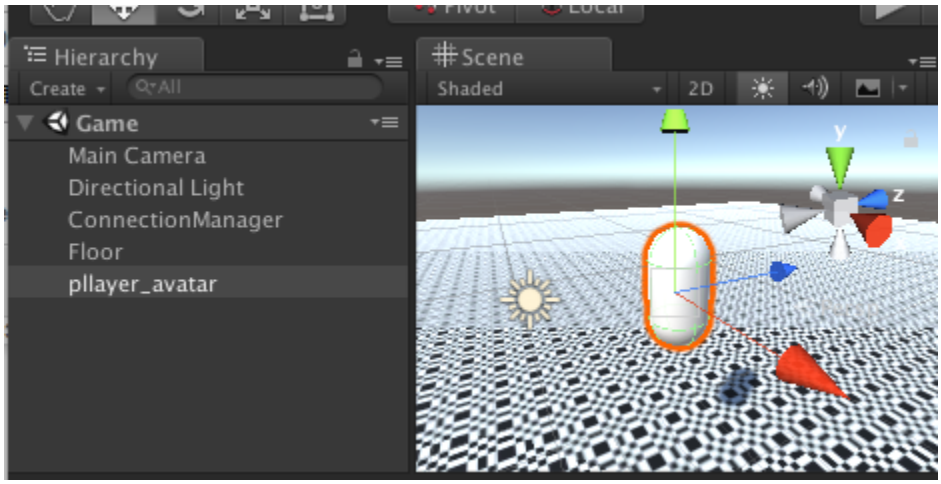
Creating a networked object

To start we'll create a networked capsules whose positions and rotation will be synced across the network. We'll eventually turn this object into a prefab for player's avatar.

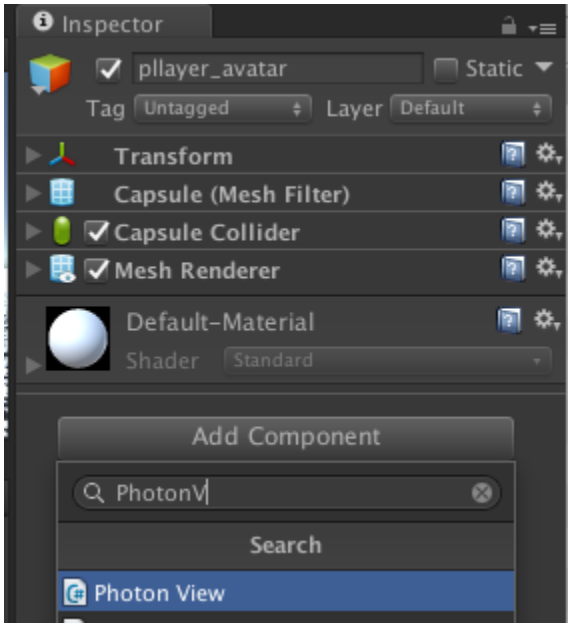
(!) Create a capsule, (it will eventually be the player's avatar)



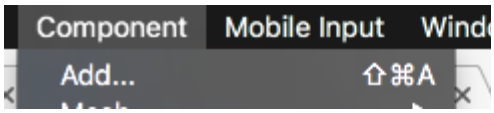
(!) Name it **"player_avatar"**



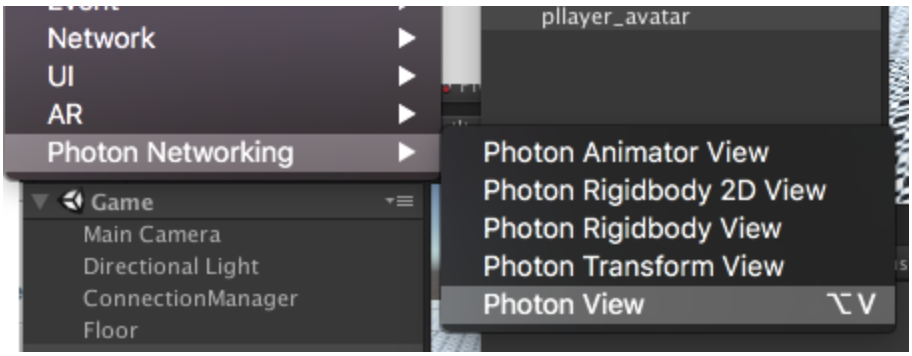
(!) Add a ***PhotonView*** component to “player_avatar”



*You can use the **‘Add Component’ button** in the inspector…*
OR



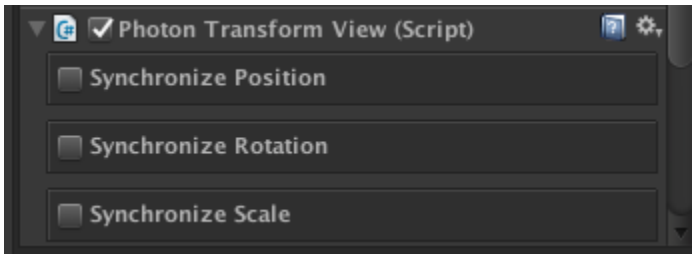
…



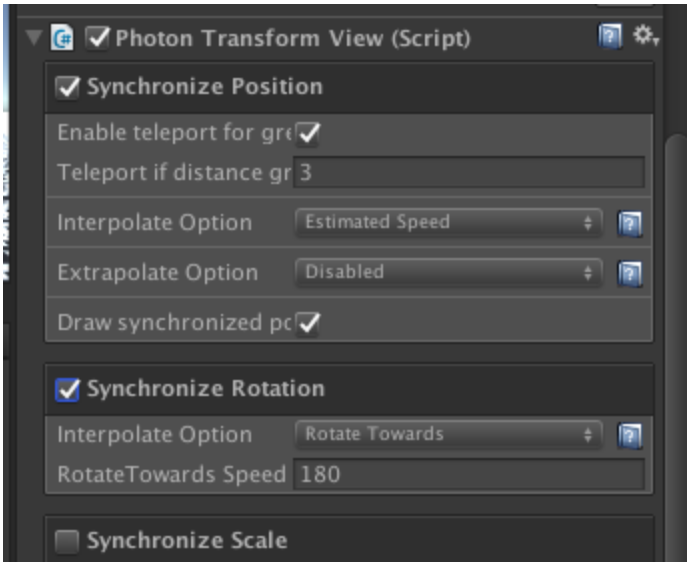
*Use the Menu **‘Component->Photon Networking->Photon View’***

(!) Also add a ***PhotonTransformView*** component

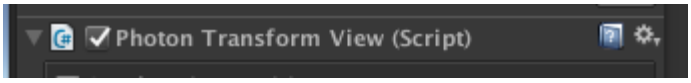
You can use the same menu, or inspector button

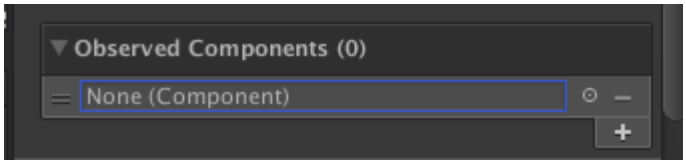


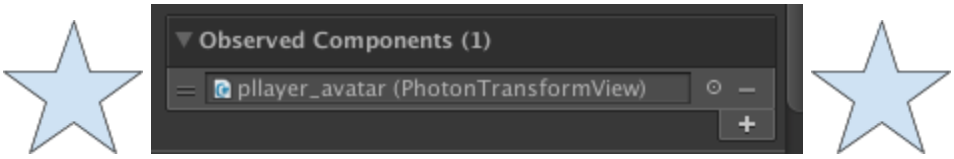
(!) Check *'Synchronize Position'* &*'Synchronize Rotation'* on the ***PhotonTransformView***



(!) Drag the ***PhotonTransformView*** component onto the ***Observed Components*** slot on the ***PhotonView.***

Drag this… 

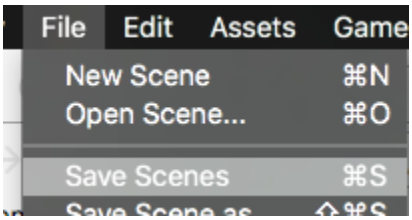
On to this! 



Should look like this once dragged

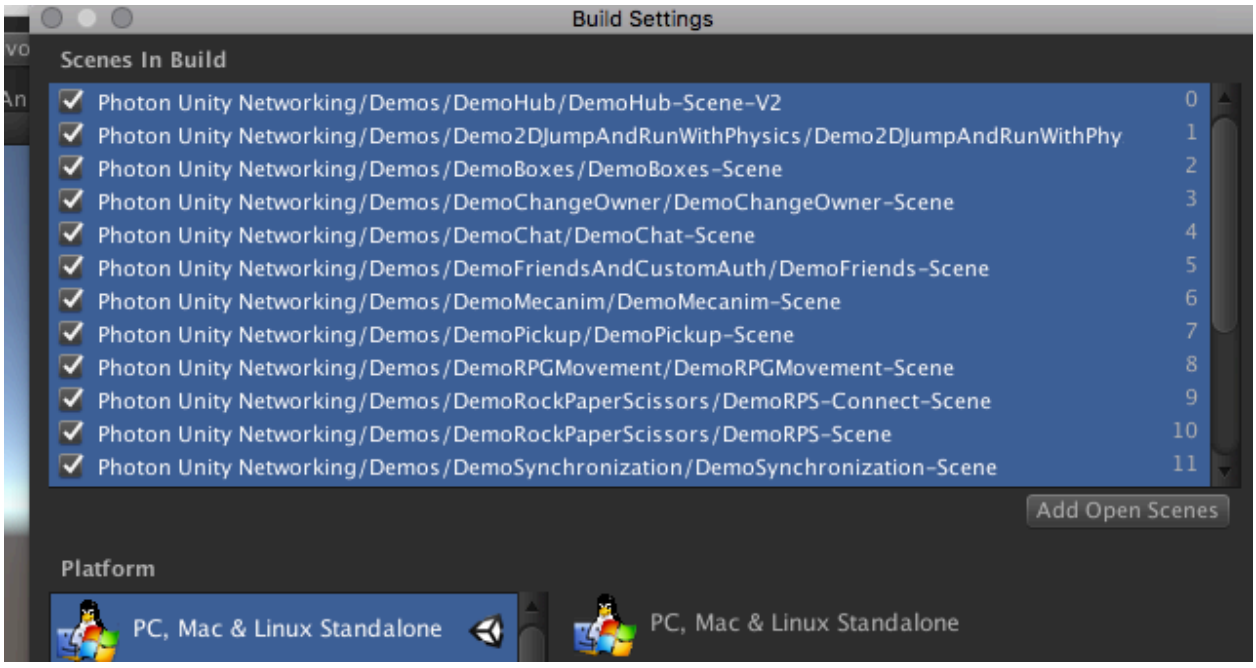
Moving a networked Object

(!) Save scene and call "**Game**".



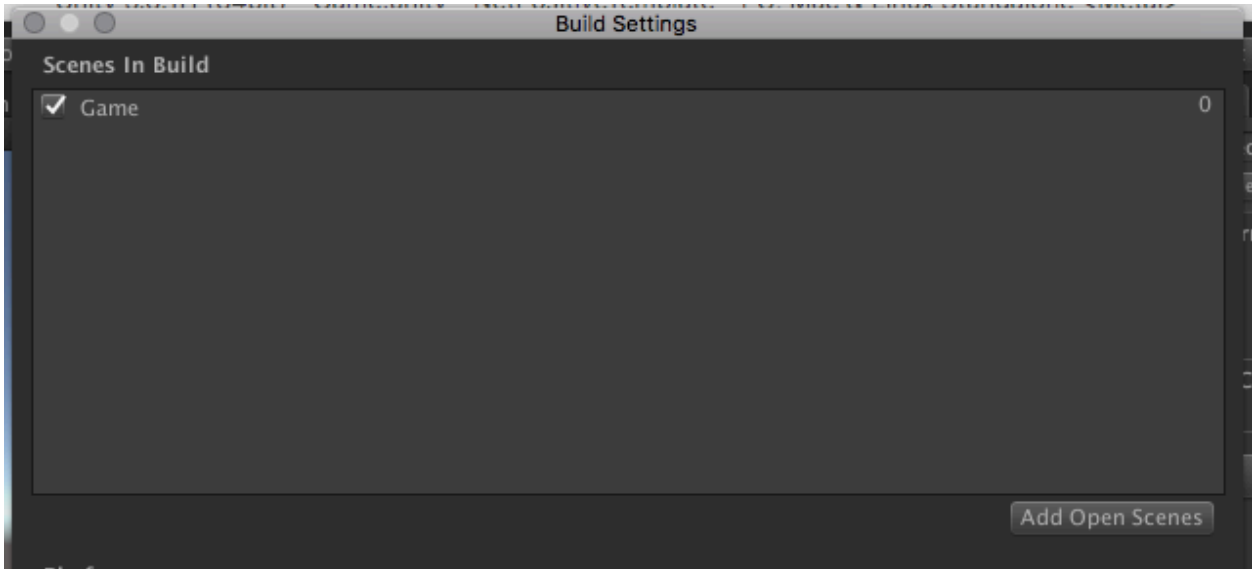
(!) Open **File->Build Settings…**

(!) Select and delete all the scenes in build



Select and delete these

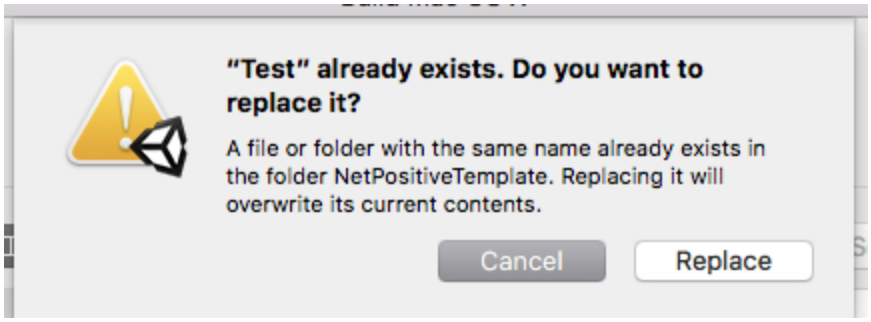
(!) Click the ‘Add Open Scenes’ button to add “Game” scene to the build.



You should now just have 1 scene in the build

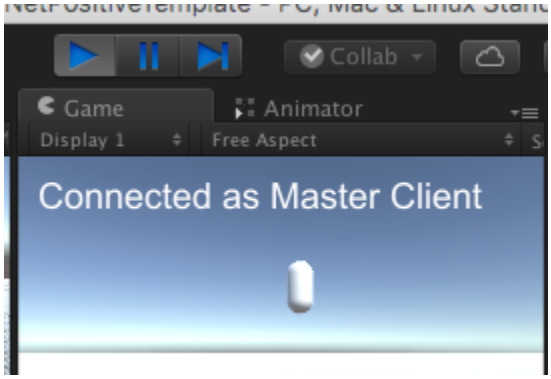
(!) Build your game. again

Don’t change the name or location in the file dialog! This way, you can just use your taskbar/dock shortcut again.



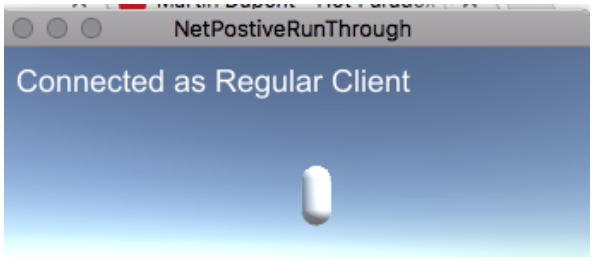
Just replace the old build! It will save you time

(!) **First** Run the game in the **EDITOR**.



Build should say, connected as Master Client

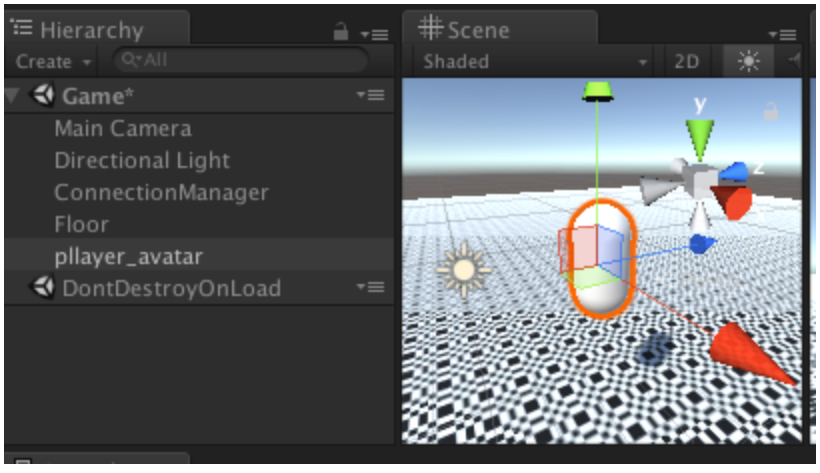
(!) **Second,** run the build you just made.



Build should say, connected as Regular Client

(!)Try moving *"player_avatar"* in the scene view of the editor with the move tool.

You should see it move on the build window too when you do!



*Try moving **'player_avatar'** in the scene view*

Now the other way around

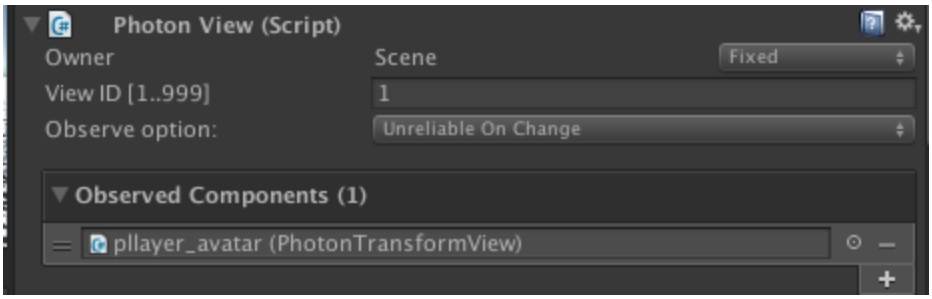
(!) Try running the build first, THEN the editor. & try moving the player in the Editor this way.

…….It won't work!

Build should be master, editor should be regular

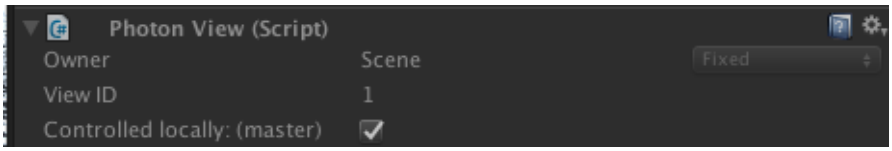
About the Photon View

A ***Photon View component*** designates a game object whose state needs be synchronized over the network. Unfortunately, it doesn't just magically sync everything about an object, synced information must be contained in special scripts. ***PhotonTransformView*** is one such script that does a good job syncing position/rotation/scale over the network. Later we will write our own script to sync our own custom player state over the network.

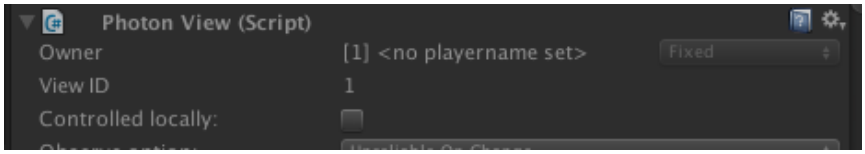


Why could we sometimes move the object, and other times not?
Remember our pong discussion? In networked games, game objects, (like this player) have an ***owner***. I.E. which of the player owns, and is responsible for an object. The player that *owns* the object is the authority on its various state (like, position, rotation, etc…) and all other players are updating their *local* copy of the object to match the owner's version of the object.

Objects in the scene, are as you would expect are given to the first player who joins the room. This first player is called the ***master client***. In our "broken" case, the build-game ***owns*** the player-capsule, and the editor-game is constantly updating the capsules position to reflect the build's version (The Photon Transform View is the culprit!, helped out by the Photon View). Our attempts to move this object that doesn't belong to us is being quickly overwritten by this background process that makes its position match the own. If you pay attention to the ***Photon View*** component in both situations, you'll see that a checkbox will appear *"Controlled Locally"* signifying whether or not the editor is the owner of a particular object.

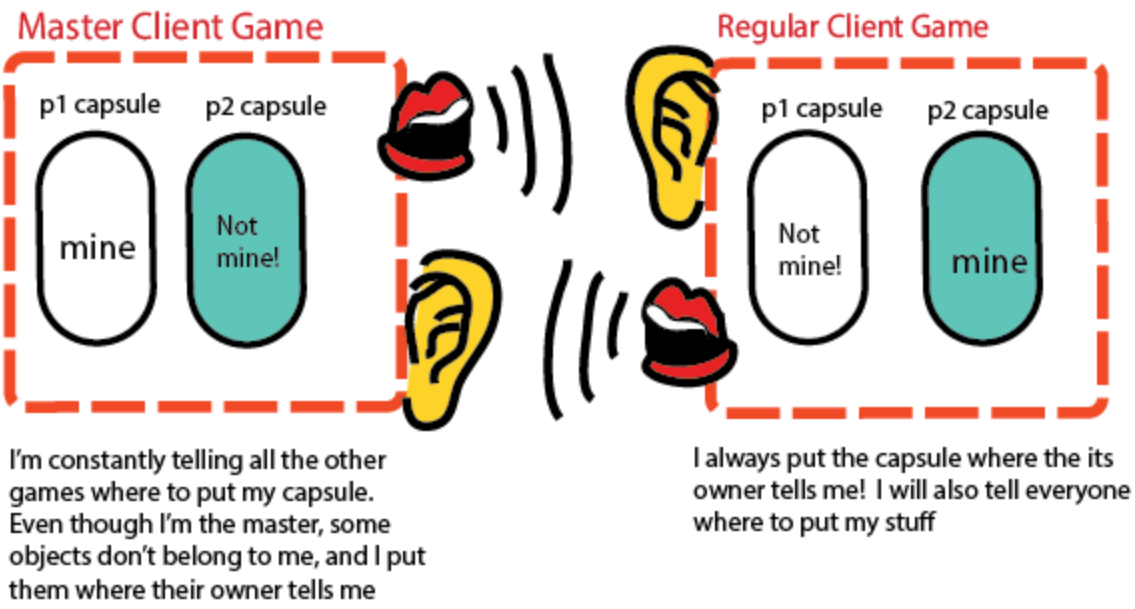


If the editor owns the object…



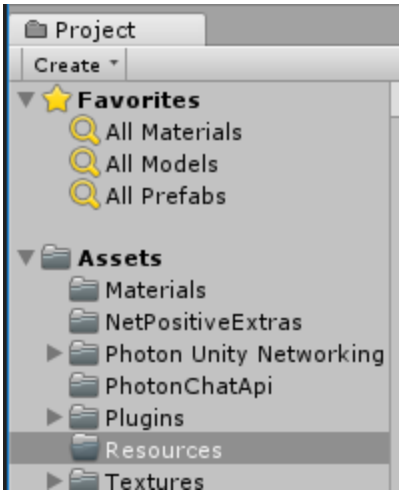
If the editor DOESN'T own the object…

What we really want, is create an avatar for *each* connected player, and have each player be the owner of their avatar.



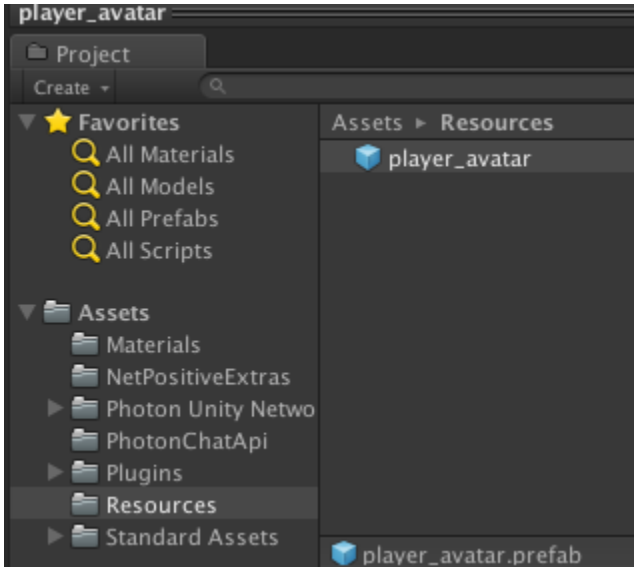
Creating our own 'Player Game Object'

(!) Create a new Folder called 'Resources'



As many of you may know already, the contents of folder named 'Resources' are available to dynamic loading with 'Resources.Load()'

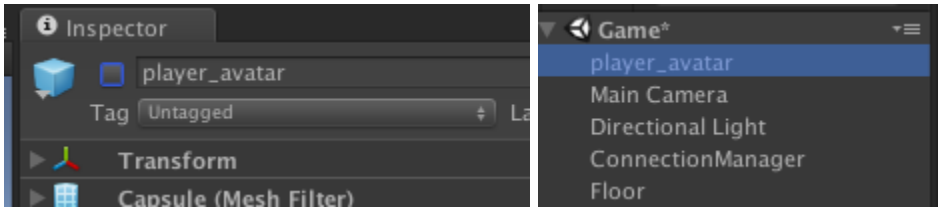
(!) Make the *player_avatar* a prefab by dragging it from the hierarchy into the ***Resources*** folder in the *Project* tab.



*It **MUST** go in the **'Resources'** folder*

(!) Disable the *player_avatar* object in the hierarchy

We will be modifying this prefab, so it will be helpful to keep it around in the scene.



(!)In PhotonConnectionManagerIncomplete script, we're going to add the following function

```
void OnJoinedRoom()
{
GameObject localPlayer = PhotonNetwork.Instantiate("player_avatar", Vector3.zero,
Quaternion.identity, 0);
localPlayer.name = "local avatar";
}
```

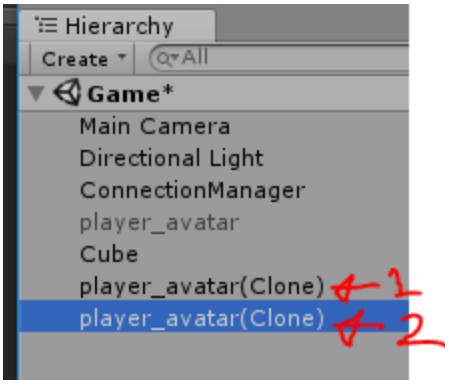
Explanation:

`OnJoinedRoom()` is automatically called when we join a room. Now when a player first joins a room, they'll instantiate their own avatar object. When a player creates an object with PhotonNetwork.Instantiate, that player owns the object.

(!) Build your game again

(!) Run the build, and the Editor..

you'll see two new objects in the hierarchy, and you'll be able to correctly move the one labeled "local avatar (Clone)".



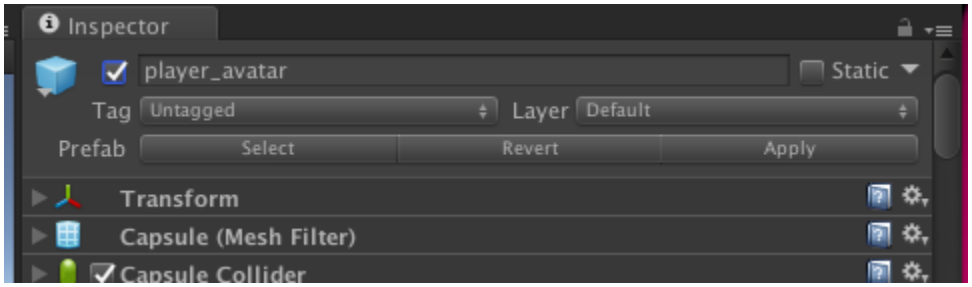
Experiment: What will happen if you try to move the these in the editor??

Make your player keyboard controlled

Warning: Make sure you stop the Editor, and your build before moving on to this step! Otherwise you'll lose your changes and have to do it all over again!

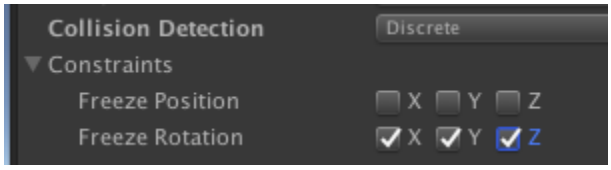
Moving the objects in the editor is alright for testing… but let's make it so we can control each object with the keyboard. First, we need to update our 'player_avatar' prefab.

(!) unhide the player_avatar object



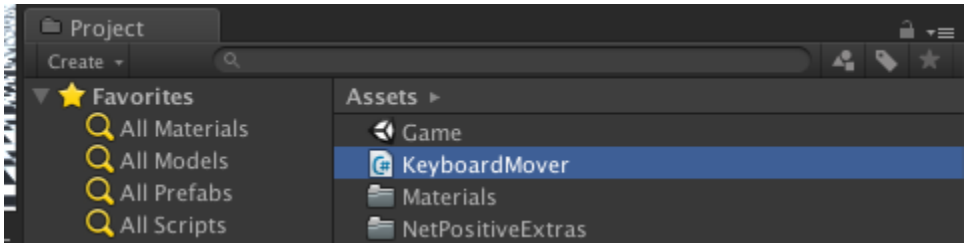
(!) Add a rigid body to the player_avatar object

(!) on the rigidbody,in ***Constraints***, Check ***Freeze Rotation*** for X,Y & Z



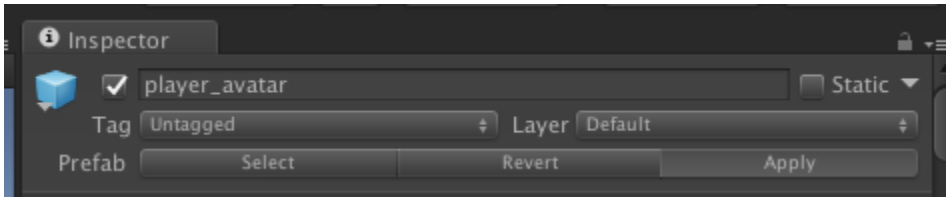
Freeze rotation for x,y, and z

(!) Add a ***KeyboardMover*** script to the prefab.



This is a very simple premade script I'm providing.

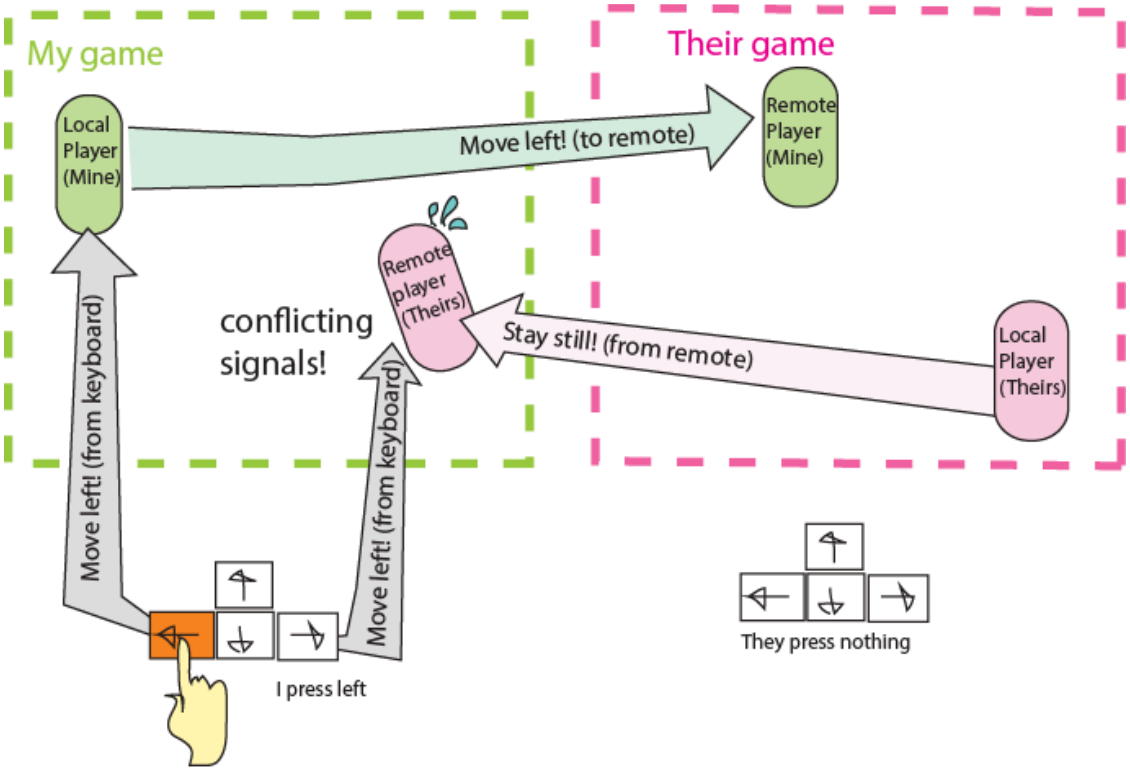
(!) Hit ***'Apply'*** at the top of the inspector apply these changes to the ***player_avatar*** prefab

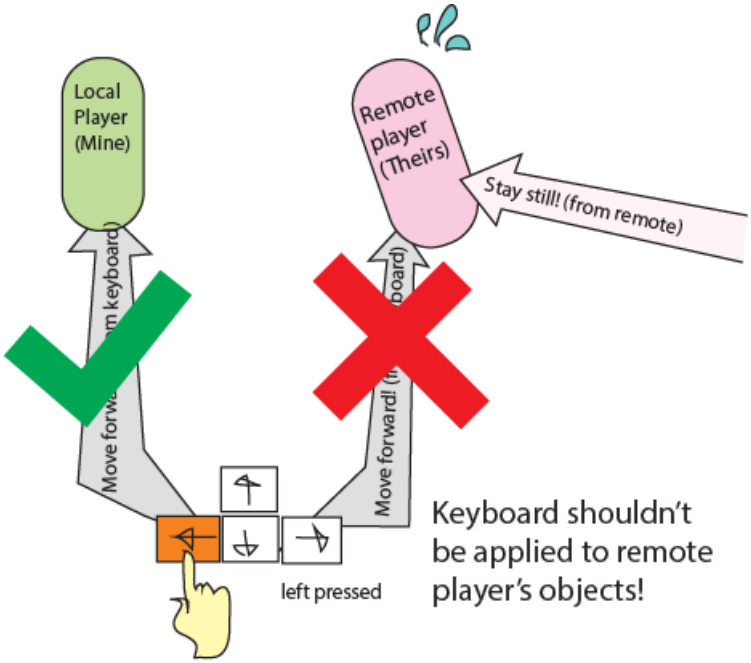


(!) hide the **player_avatar** object again

Try it out, and you can see that it kind of works, but that both players move when you hit the keys.

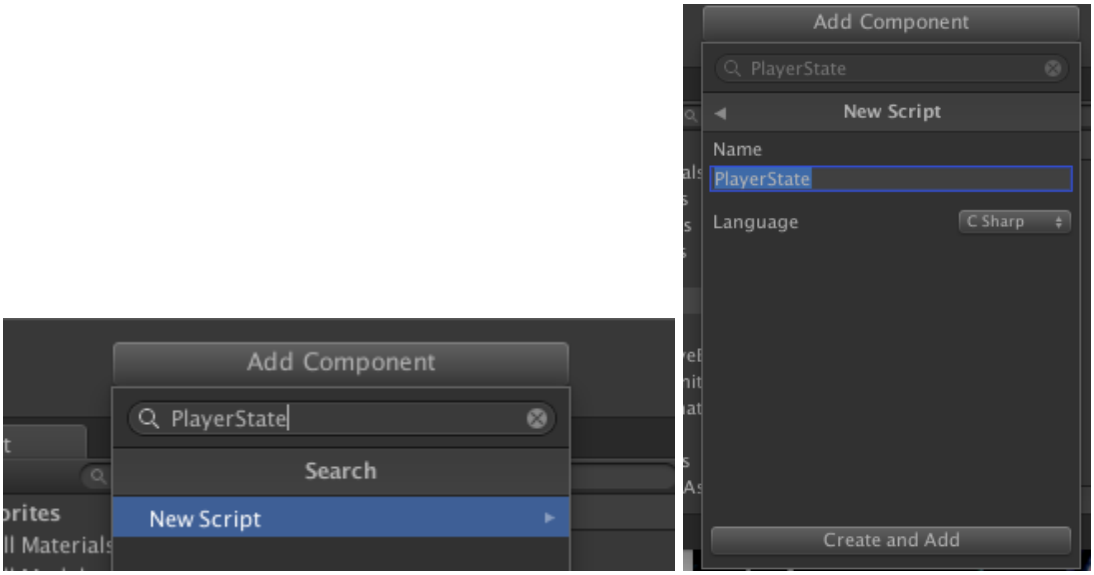
What's wrong?????
Both our local player, and the remote player have Keyboard.cs attached. My keyboard movements are applied to both my own player object, and the remote player object. On the remote player object shouldn't be affected by my keystrokes, and these competing with the remote player's "true" position in the build.





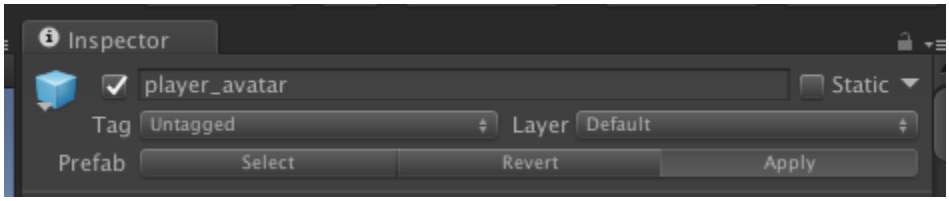
Turn off KeyboardMover.cs for any player we don't own

(!) Add a new script "PlayerState" to the *player_avatar* prefab.



You can add a new script by using the ***Add Component Button***, typing the name you want to use (PlayerState in our case), and then hitting **"New Script**", then hitting **'Create and Add'**

(!) Hit *'Apply'* at the top of the inspector apply these changes to the *player_avatar* prefab



<Come up with a better name than PlayerState? NetworkedPlayer??>

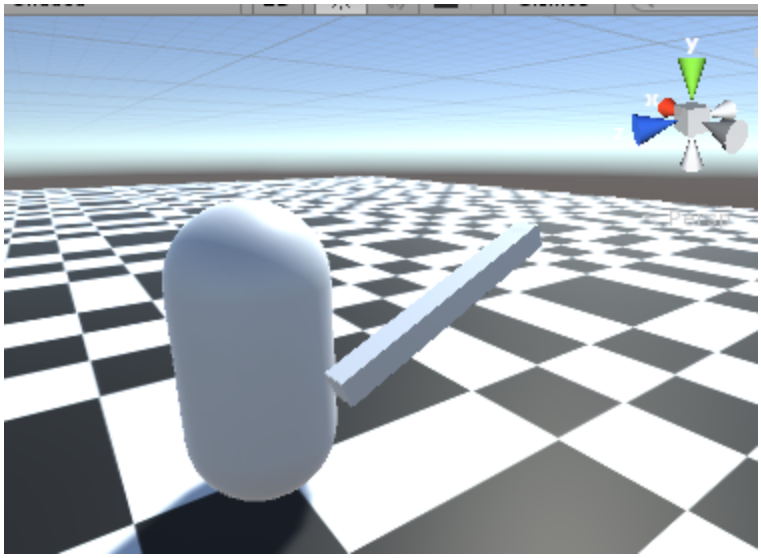
(!) Open PlayerState, and change its Start() function to the below:

```
void Start ()
{
    if (!this.GetComponent<PhotonView>().isMine)
       {
       this.GetComponent<KeyboardMover>().enabled = false;
       this.GetComponent<Rigidbody>().isKinematic = true;
       }
}
```

The line: `this.GetComponent<PhotonView>().isMine` is ***true*** if we own this object, and our changes will propagate over the network. If it's ***false*** it's owned by another player on another computer. We only want to control move our own local object with keyboard, so we should turn off the *KeyboardMover* for any player object that isn't ours. Similarly, we should also disable the rigid body (done by setting isKinematic=true). Remember, the owner of an object is the authority on its state (in this case its position/orientation). Simulating physics on an object changes its position/orientation! If we run physics on an object we don't own, our local physics system will set a position for the object in competition with the "true" position sent from the remote player.

Making a Weapon

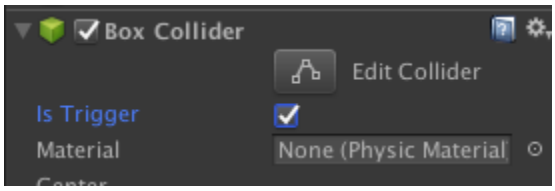
Let's make a "sword" our players can swing by hitting space.



(!) Make a cube and stretch into a long stick/dagger/club shape.

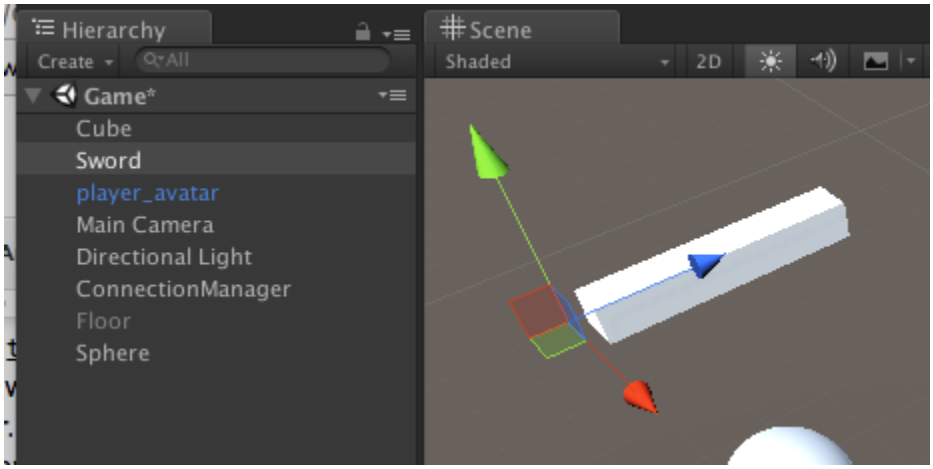
I recommend a scale of (0.1, 0.1, 2)

(!) Check “isTrigger” on the stick’s collider



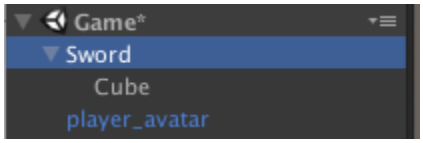
(!) Make an empty game object, and line it up with one end of your stick (a little bit past is good)

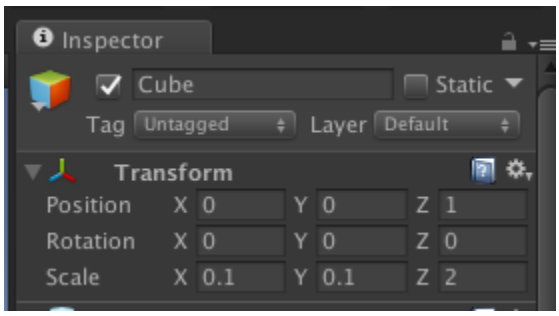
(!) Rename the empty game object *“sword”*



The cube-stick be lined up with the blue z-handle of the empty object.

(!) Child the stick to this empty *“sword”* object



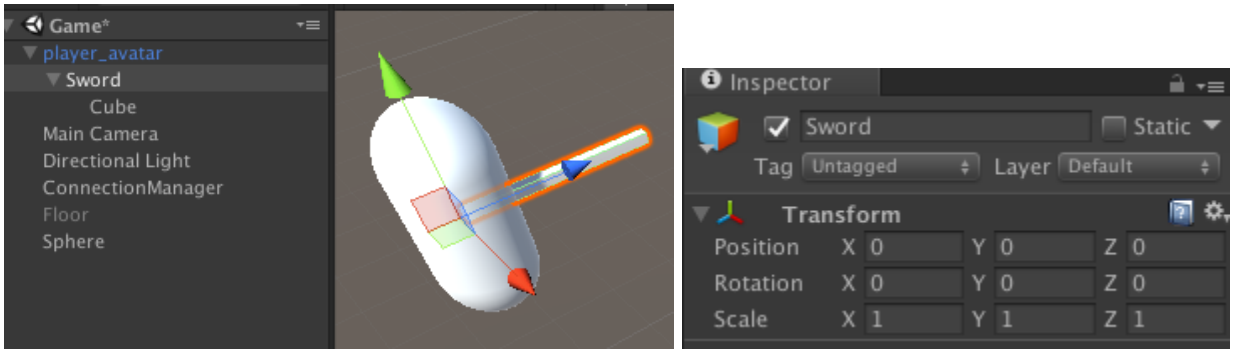


(A good position for the stick (0,0,1) if you also used my recommended scale)

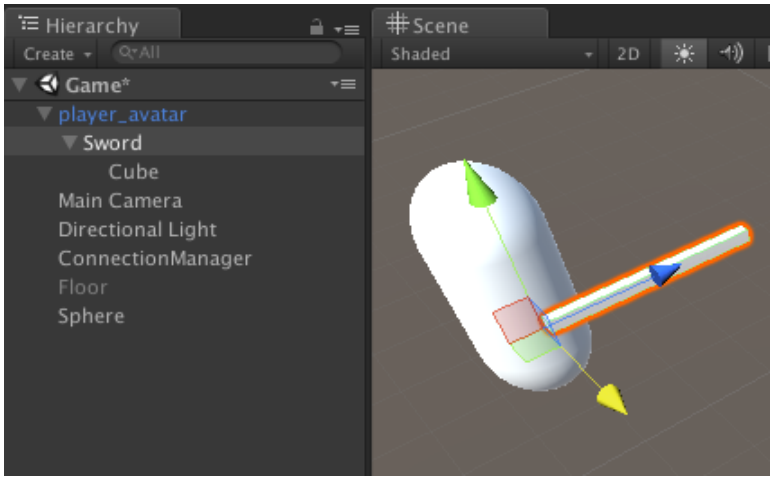
(!) Unhide *player_avatar* and child your *sword* to *player_avatar*

(!) Select the *“sword”* in the hierarchy

(!) Zero out the “sword” object’s position, in the inspector and optionally move it just to the right of the player.

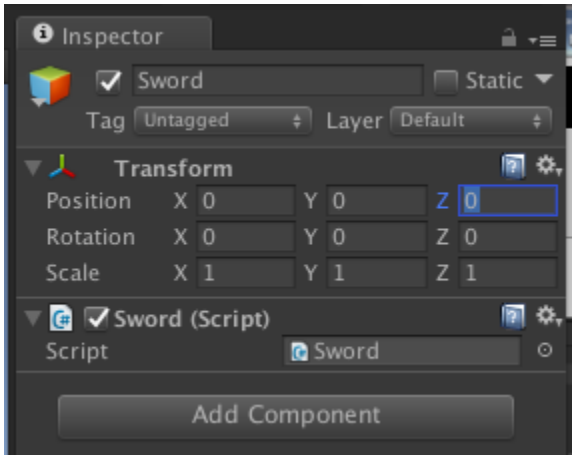


Sword at (0,0,0) position



Pull the red move handle, so the sword is at the player’s side

(!) Add a new Script to your sword object, calling the script ‘Sword’

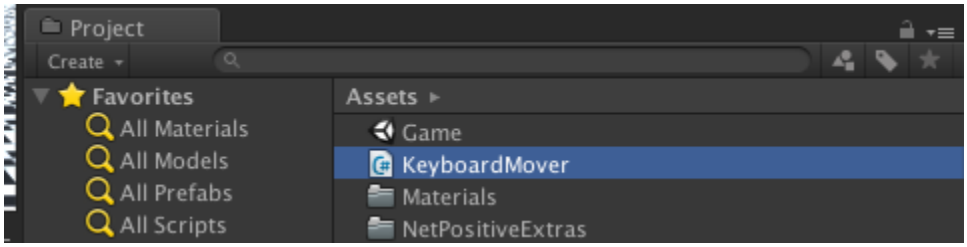


(!) Add the following function to ‘Sword’ script (it can be called to swing the sword)

```
public void swingSword()
{
	//make the sword appear
	this.gameObject.SetActive(true);

	//This function, varyWithT() performs a 1-time animation specified in code
	//It takes 2 arguments:
	//1st: an animation function
	//2nd: the duration of the animation
```

```
        this.varyWithT(

        //Animation function, called over repeatedly the 'animation duration'
        (float t) =>
        {
                //t is is the 'normalized animation time'
                // t = '0' at beginning of animation
                // t = '.5' halfway through
                // t = '1' at end of animation

                //Move the Sword from pointing up, rotation = (-90,0,0) ...
                // ... to pointing forward, rotation = (0,0,0)
                this.transform.localEulerAngles =
                        Vector3.Lerp(new Vector3(-90, 0, 0), new Vector3(0, 0, 0), t);

                //at the end of the animation, make the sword disappear
                if (t == 1)
                {
                        this.gameObject.SetActive(false);
                }
        },

        //Animation duration
        .2f
        );
}
```

(!) Select *player_avatar* and apply the changes to the prefab.



(!) Disable player_avatar in the hierarchy again.

Actually swing the sword by calling this function

(!) In *PlayerState*, add the following function

```
void swingWeapon()
{
	this.GetComponentInChildren<Sword>(true).swingSword();
	//the 'true' above has GetComponentInChildren also check in inactive children
	//which our sword often is!
}
```

(!) Also, in *PlayerState*, replace the empty Update() with the following:

```
void Update()
{
	if (Input.GetKeyDown(KeyCode.Space))
	{
		swingWeapon();
	}
}
```

Test it out: can you swing your sword?
If you try it out now, when you hit space, your player's sword is working OK, but you can't see the other players swinging their swords! The function *swingWeapon()* is only being called *locally,* on our single instance of the game. We want it to be called on everyone's else game too!

<Fixing the first problem… similar to disabling the keyboard for remote players… need to ignore keyinput, except for a player we own>
/*&& this.GetComponent<PhotonView>().isMine*/

RPC (Remote Procedure Call)

To call a function over the network, we use something called a *Remote Procedure Call* (RPC)
https://doc.photonengine.com/en-us/pun/current/manuals-and-demos/rpcsandraiseevent

First, we must designate the function as an *RPC*

(!) In *PlayerState* add the line [PunRPC] just above swingWeapon(), like so:

```
[PunRPC] //marks a function so that it can called over the network
void swingWeapon()
{
	this.GetComponentInChildren<Sword>(true).swingSword();
}
```

Next, we use the method *RPC()* of our object's *PhotonView* component to call that object:

(!) replace the swingWeapon() line in Update to match the following:

```
...
if (Input.GetKeyDown(KeyCode.Space))
{
	this.GetComponent<PhotonView>().RPC("swingWeapon", PhotonTargets.All);
}
...
```

Try it out, and you should see the other player swinging their sword.

Breaking PRPC line down...

`this.GetComponent<PhotonView>().RPC("swingWeapon", PhotonTargets.All);`

PhotonView.RPC takes 2 arguments, first, **name of the function** you want to call, and 2nd, the **target players**, on the network you want receive the function. Here, we're calling it on every connected player, including ourselves. It's also possible to call a function on a specific connected player, or subset of players, but we won't get into any of those cases today <still true?>

More info about *PhotonTargets*:
https://doc-api.photonengine.com/en/pun/current/group__public_api.html#gab84b274b6aa3b3a3d7810361da16170f

<Think about it…. Why don't we also sync the swords transformation info, like we did for the player?>

Hitting Things

Let's make it so things respond to being hit with sword.

First, let's just print to the console when our sword hits something.

(!) Open Sword script, and add the following

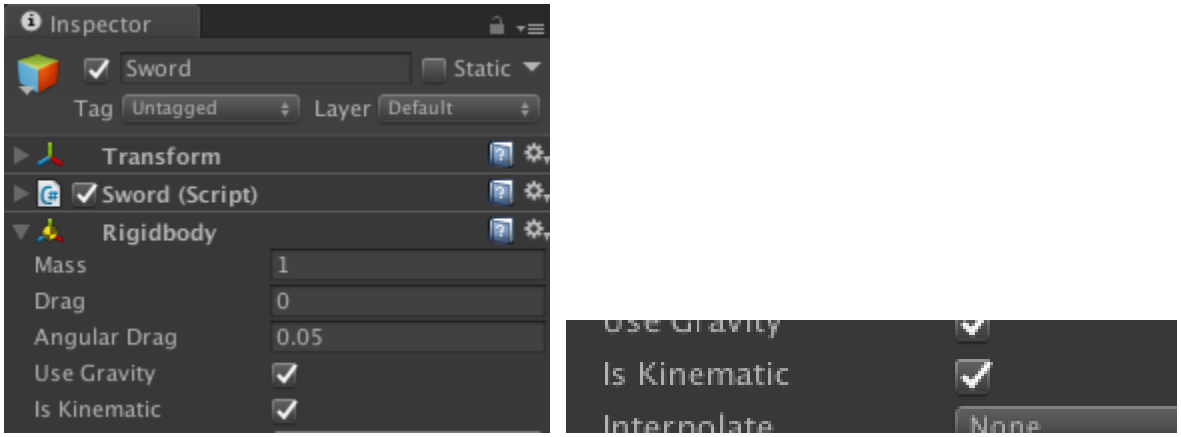
```
public void OnTriggerEnter(Collider other)
{
	Debug.Log("Sword hit " + other.name, other.gameObject);
}
```

(!) Run the game, (just the editor is fine), and check the console as you hit space to swing the sword to see what your sword is hitting

We might be hitting our own player! The google doc Sword will be updated accordingly to fix this if necessary.

(!) We should also add rigidBody to the sword, and check is kinematic, so our sword will recognize objects without rigidbodies.

(FYI, at least one object in trigger collision needs a rigid body)



*Select the sword, add rigid body, check **'Is Kinematic'***

Making ‘Hittable’ things

What we really want, is for the sword to act on a wide range of “hittable” objects, which will respond to being hit in their own way. Enter, the *interface*.

(!) Make a new script, Hittable

(!) Open, delete everything and paste in below

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public interface Hittable
{
	void takeHit(int damage);
}
```

This script won’t be attached to anything directly. Any other script can *implement* this interface by including a public function called `takeHit(int damage)` and be recognized as a *Hittable* typed object.

Read more about interfaces here
http://csharp-station.com/Tutorial/CSharp/Lesson13

(!) Add the below to Sword

```
public void OnTriggerEnter(Collider other)
{
	Debug.Log("Sword hit " + other.name, other.gameObject);

	Hittable otherThingHittableScript = other.GetComponent<Hittable>();

	if (otherThingHittableScript != null)
	{
		otherThingHittableScript.takeHit(1);
	}
}
```

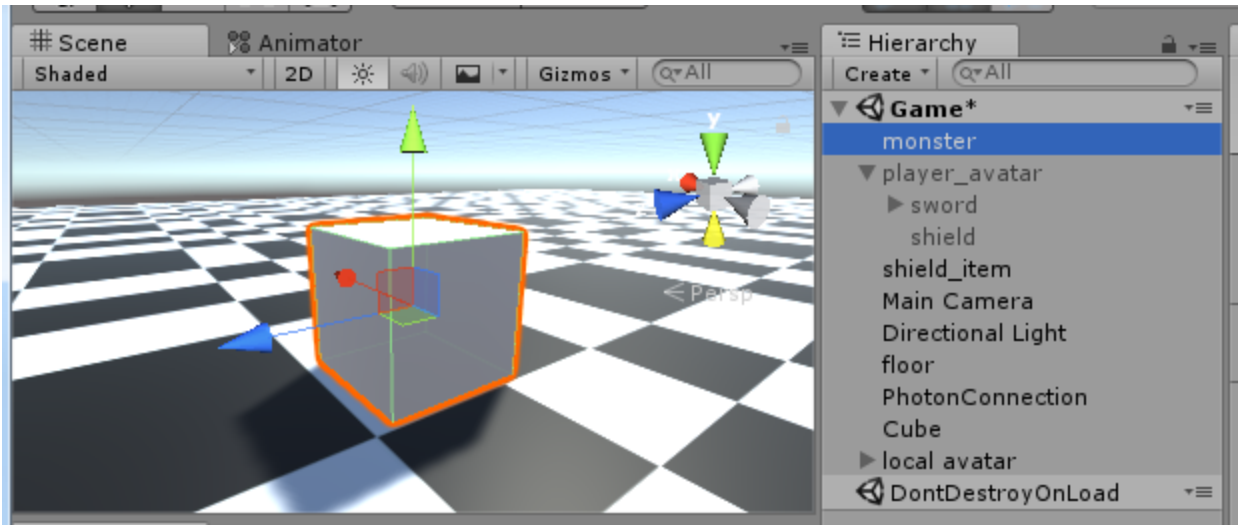
In this new code, the sword checks if the the thing colliding *implements* our Hittable interface, in which case we can call its *takeHit()* function.

Make a reusable destroyable/hittable behavior

Let's make a script that make an object disappear if hit by a sword, a specified number of times. First we'll use it on a killable monster/destroyable wall.

(!) Make a cube

(!) Name it monster



(!) Add a **RigidBody** component

(!) Make and attach new Script, HittableAndDie

(!) Alter the following ", Hittable" after "MonoBehavior" in HittableAndDie

---

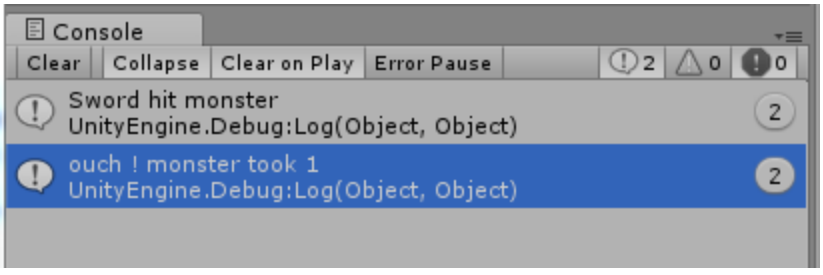
```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class HittableAndDie : MonoBehaviour, Hittable
{
...
```

This designates HittableAndDie as implementing 'Hittable'. But we're getting a compiler error!

(!) Add the function takeHit(int Damage) to HittableAndDie to finish implementing the 'Hittable' interface.

```
public void takeHit(int damage)
{
	print("ouch! " + this.name + " took " + damage + " damage");
}
```

(!) Run the game, (just in the editor is fine), a check that hitting the monster generates an 'ouch' message in the console.



(!) Let's add some more code, and make use of PhotonNetwork.Destroy(), to make the object disappear if hit 3 times. Complete code will be in the google doc version of HittableAndDie

---

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class HittableAndDie : MonoBehaviour, Hittable
{
  public int hp = 3;

  public void takeHit(int damage)
  {
       this.GetComponent<PhotonView>().RPC("takeDamage", PhotonTargets.All, damage);
  }

  [PunRPC]
  public void takeDamage(int damage)
  {
       Debug.Log(this.name + " took " + damage, this.gameObject);

       //only on the owner's instance actually modify hp,
       //and destroy the object if no HP left.
       if (this.GetComponent<PhotonView>().isMine)
       {
         hp -= damage;

         if (hp <= 0)
         {
              PhotonNetwork.Destroy(this.gameObject);
         }
       }
  }


}
```

(!) You'll have to add a **PhotonView** component as well to so the RPC will work

<Hold on!!! Is this really the right way??? What if I hit the monster a bunch… and another player signs in after???>

(Optional) Make the monster wander around

(!) Add **RandomNPCMover** script

(!) Add a **PhotonTransformView** component,

(!) Drag the **PhotonTransformView** onto *Observed Component* slot of the PhotonView Component

(!) Check *'Synchronize Position'* &*'Synchronize Rotation'* on the ***PhotonTransformView***

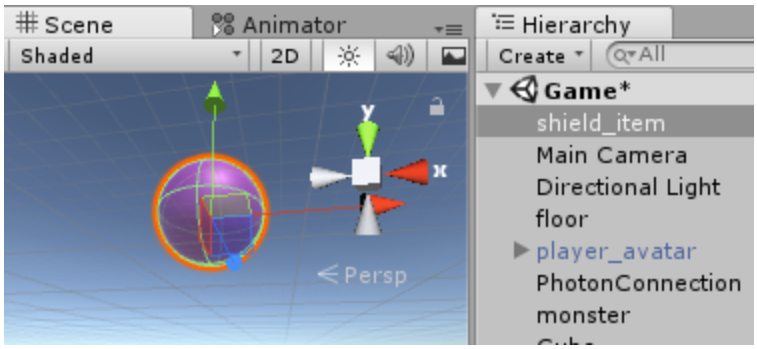
(Optional) Make a hittable door.

Add this script, HittableDoor to an object and it will lift up when hit with a sword.
You'll need to add Photon view, and a PhotonTransformView.

Items, Syncing variables over the network.

Let's make a collectible item, that gives the player a temporary magical barrier.

Make the collectible shield item



(!) Create a sphere (this will be the item)

(!) Check ‘isTrigger’ on its collider

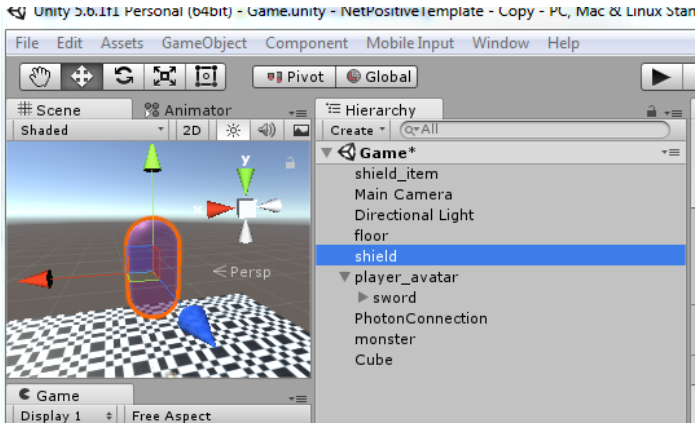
(!) name it “shield_item”

(!) Add a PhotonView

(!) size, and apply a material to make it look however you want.

Adding a shield to the player

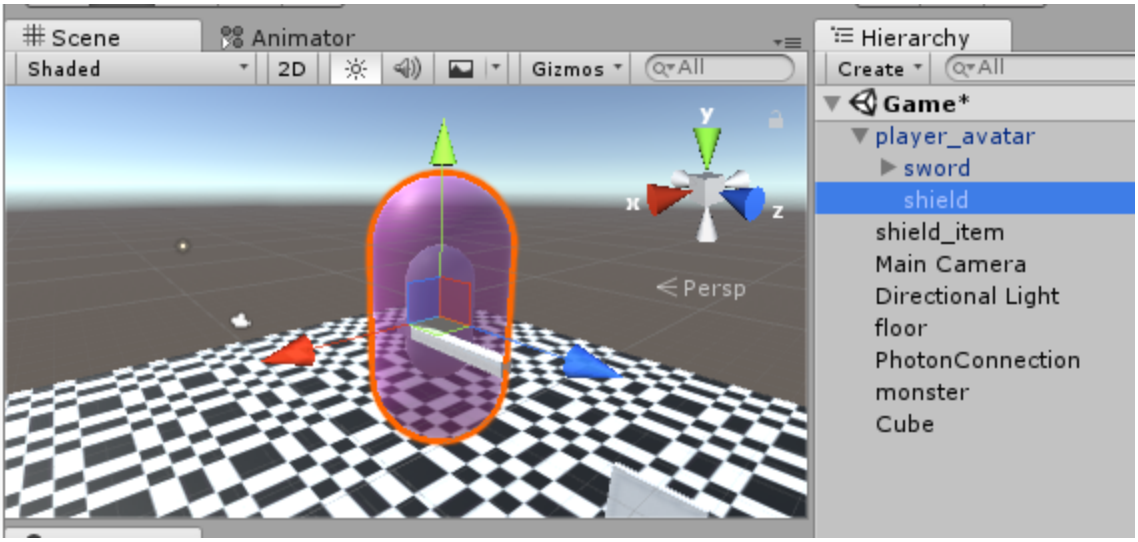
(!) create a capsule, and apply a translucent material to it. (this will be the magic shield)



(!) Name the capsule ‘shield’

(!) remove the collider from the capsule

(!) Child this object to your player prefab, and zero out its position



Making the shield turn on, upon collecting the item

<This section could be refactored a bit… probably… do something more object-y? Check for a a shield object, and have that take the hit ?>

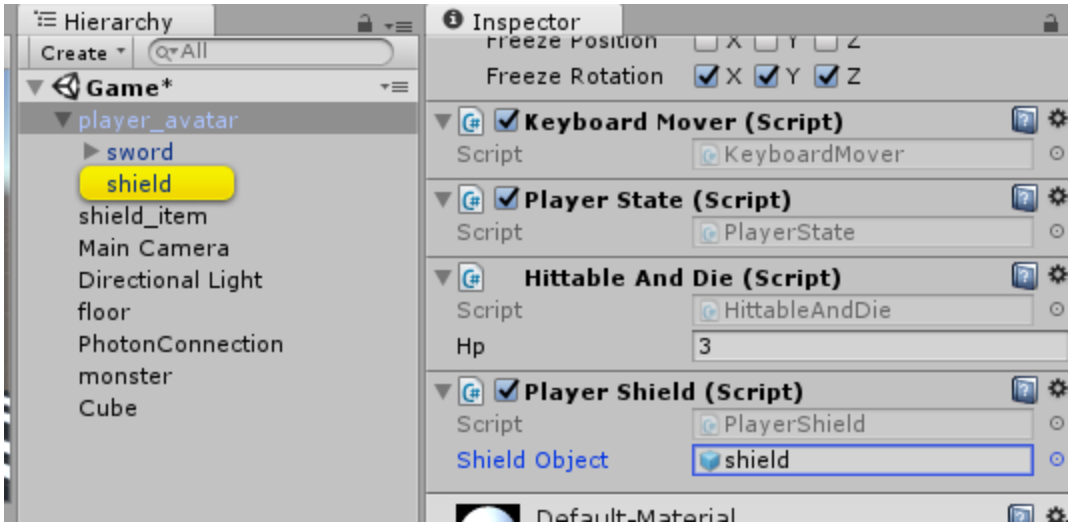
(!) Add a new script PlayerShield to the *player_avatar* prefab

---

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class PlayerShield : MonoBehaviour {

       public GameObject shieldObject;

       bool shieldIsOn = false;

      void Update ()
       {
         shieldObject.SetActive(shieldIsOn);
       }

       private void OnTriggerEnter(Collider other)
       {
        if (other.name == "shield_item" && this.GetComponent<PhotonView>().isMine)
         {
              //Turn on the shield!
              shieldIsOn = true;

              //Collect the item
              PhotonNetwork.Destroy(other.gameObject);
         }
       }
}
```

<!!! There's actually… something strange that can go wrong here, (race condition), you can read about it here: <where?> >

(!) In inspector, drag the 'shield' capsule object onto the 'shieldObject' slot on the 'Player Shield' script.



(!) If you make a build and test now, you'll see shield appear locally, but it won't appear for other players on the network.

(!) Hit apply on 'player_avatar', then hide again

Syncing 'shieldIsOn' variable across the network

Similar to making function calls over the internet, it's also possible to sync variables over the network. It's a little more complicated than an RPC. A script has to have a special function called `OnPhotonSerializeView`, which sends and receives variables over the network. This script must be attached to an object with a PhotonView, and the script must finally added to the PhotonView's observed component area (just like the *PhotonTransformView* we use for the player).

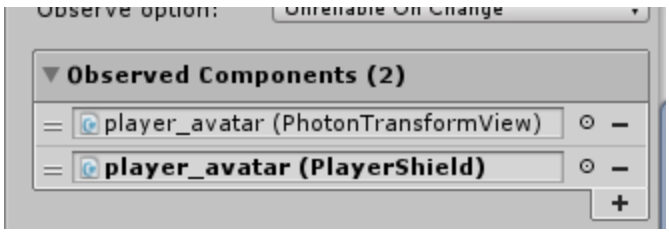
(!) Add the following function to PlayerShield

```
public void OnPhotonSerializeView(PhotonStream stream, PhotonMessageInfo info)
{
	if (stream.isWriting)
	{
	// We own this player: send the others the variable
	stream.SendNext(this.shieldIsOn);

	}
	else
	{
	// we don't own this player, receive the new value for the variable
	this.shieldIsOn = (bool) stream.ReceiveNext();

	}
}
```

All `OnPhotonSerializeView` functions will be of this form: In the 'stream.isWritingBranch' 'stream.SendNext()' call with each variable we want to receive, and in the 'else' branch, reading those values out in the same order. The '(bool)' part is called a cast. `stream.ReceiveNext()` returns the variables as a generic 'object', and this cast is used to convert it back to its proper specifc type (this case, a bool).

More info on *OnPhotonSerializeView()*
https://doc.photonengine.com/en-us/pun/current/getting-started/feature-overview#_observembpun

(!) **CRITICAL** Add the *PlayerShield component* onto the *Observed Components* slot on the *Photon View.*



(!) Apply the changes to the player_avatar prefab, then hide it

The End (~ish)

This is all I have time to write down, but there are several natural extensions/continuations that I can show you or help you try on your own.

- Try Creating multiple kinds of sword swings

- Try making a projectile weapon, that calls also uses Hittable and Take hit
- Make the enemies attack players if Nearby
- Make other kinds of items
- Display HP over the network
- Make the HittableOnDie ignore a hit if there's a PlayerShield active.

Bonus: Here's a link to a fancier version of this project