Contents

Overview

The first part of this tutorial is a walkthrough to one of the Photon Unity Networking (short: PUN) demos. You will get the PUN package into Unity, set it up to use the Photon Cloud and give it a test run.

The second part teaches you to develop multiplayer features with the Photon Unity Networking package and the Photon Cloud service.

What you need

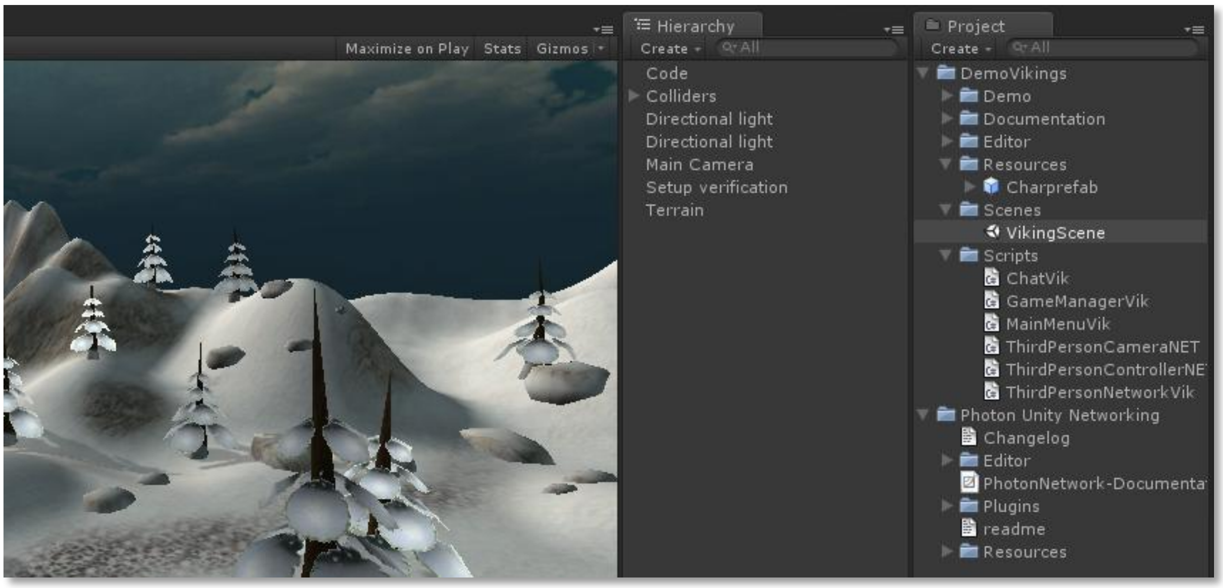
This tutorial assumes you know the basics of using the Unity Editor and programming. The sample code is written in C# but works similarly in Unity Script.

Walking Vikings

Let's first try the "Viking Demo", one of the Photon Unity Networking samples from the Asset Store.

Everything needed, is in the package, so we create a new, empty project in Unity. In the Asset Store, search for "Photon Viking Demo" (or click the link). Download and import the package.

This adds two folders to our project: "DemoVikings" and "Photon Unity Networking". The PUN directory wraps up all the networking code you would need in a project. The "Vikings" folder contains the sample.



```
Hierarchy
Code
Colliders
Directional light
Directional light
Main Camera
Setup verification
Terrain

Project
DemoVikings
  Demo
  Documentation
  Editor
  Resources
    Charprefab
  Scenes
    VikingScene
  Scripts
    ChatVik
    GameManagerVik
    MainMenuVik
    ThirdPersonCameraNET
    ThirdPersonControllerNE
    ThirdPersonNetworkVik
Photon Unity Networking
  Changelog
  Editor
  PhotonNetwork-Documenta
  Plugins
  readme
  Resources
```

Importing the package also opens a "PUN Setup Wizard" window in the Editor.

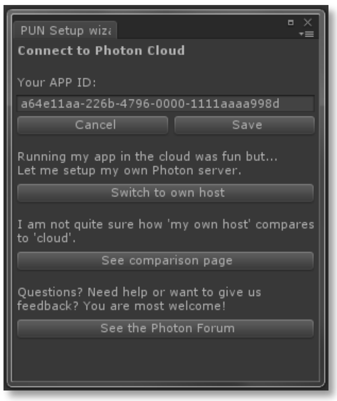
Wizard

The PUN Setup Wizard is there to help with the network settings and offers a very convenient way to get started with our multiplayer game: The Photon Cloud!

Cloud? Yes, Cloud. It's a bunch of Photon servers which we can use for our game. We will explain in a bit.

Cloud registration

Using the Cloud for a trial is free and without obligation, so for now, we just enter our mail address and the Wizard does its magic:


PUN Setup wiz
Connect to Photon Cloud
Your APP ID:
a64e11aa-226b-4796-0000-1111aaaa998d
Cancel
Save
Running my app in the cloud was fun but...
Let me setup my own Photon server.
Switch to own host
I am not quite sure how 'my own host' compares to 'cloud'.
See comparison page
Questions? Need help or want to give us feedback? You are most welcome!
See the Photon Forum

If the mail address is unknown to the Cloud, we will get an "app ID" right away. And mail.

If the address is known, we registered it before and the wizard leads us to an account page. We login to fetch our "app ID". In the Wizard, click "Setup" and copy & paste the "app ID".

Press "save" and close the Wizard.

Walking the Viking

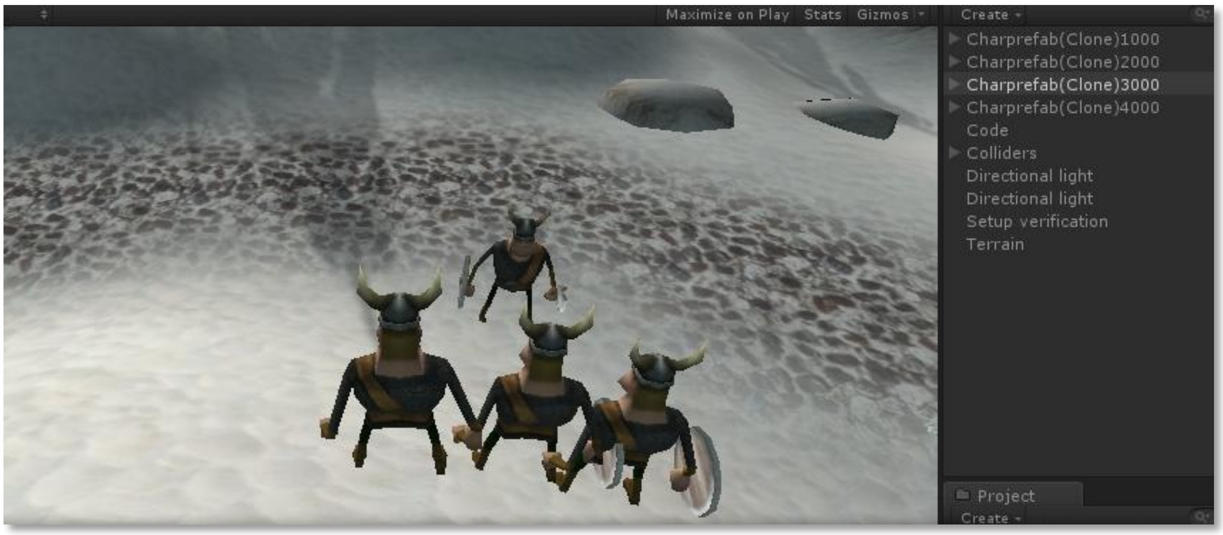
After all that work, it's time for some action. Multiplayer action!

Open the scene DemoVikings\Scenes\VikingScene and build and run the demo as webplayer or standalone.

A "main menu" pops up and we can enter a player name and room names and there is a (empty) "Room Listing". When we "Create" a room the scene switches and finally a Viking shows up. It's only one but that's a start.

Start the demo in the editor as well and the "Room Listing" contains a single entry. We "join" that room and finally there are Vikings (as in plural).

We can start a few more, running around with each. The camera always follows our "own" Viking per client.



This already concludes the first part of this tutorial. You learned how to get the PUN package, import and set it up. Not surprising but also important: You learned that you will need to run multiple clients to test your game.

Photon Cloud

So, what exactly does this “Photon Cloud” do?!

Basically, it’s a bunch of PCs with the Photon Server running on them. This “cloud” of servers is maintained by Exit Games and offered as hassle-free service for your multiplayer games. Servers are added on demand, so any number of players can be dealt with.

Even though Photon Cloud is not completely free, the costs are low, especially compared to regular hosting. Read more about the pricing here.

Photon Unity Networking will handle the Photon Cloud for you but this is what’s going on internally in a nutshell:

A single Master Server knows all the other servers and existing games. This is where all clients connect to. Each time one creates or joins a game (room), they are forwarded to one of the other machines (called “Game Server”).

The setup in PUN is ridiculously simple and you don’t have to care about hosting costs, performance or maintenance. Not once.

Rooms

The Photon Cloud is built with “room-based games” in mind, meaning there is a limited number of players (let’s say: less than 10) per match, separated from anyone else. In a room (usually) everyone receives whatever the others send. Outside of a room, players are not able to communicate, so we always want to get them into rooms asap.

In Photon, rooms have names to identify them. There is a list of current rooms but players could also use random matchmaking to join any existing room. The list is provided in the lobby.

Lobby

The lobby for your application exists on the master server to list rooms for your game. PUN will automatically join this lobby and get the room list. That’s not strictly a must-have: you could join rooms directly if you know their name or join a random game.

Application IDs & Game Version

If everyone connects to the same servers, there must be a way to separate your players from everyone else’s.

Each game (as in application) gets its own “app ID” in the Cloud. Players will always only meet other players with the same “app ID” in their client.

There is also a “game version” you can use to separate players with older clients from those with newer ones.

Starting From Scratch

Let’s start something new. For the sake of a simple tutorial, we will not create the next MMO but a Marco Polo game – with Monsters. This shows the basics and you should be able to build something on top.

Start over by creating a new, empty project. This time, during import, we uncheck the “DemoVikings” folder. Enter your app ID in the wizard and save.

Reception: Getting a Room

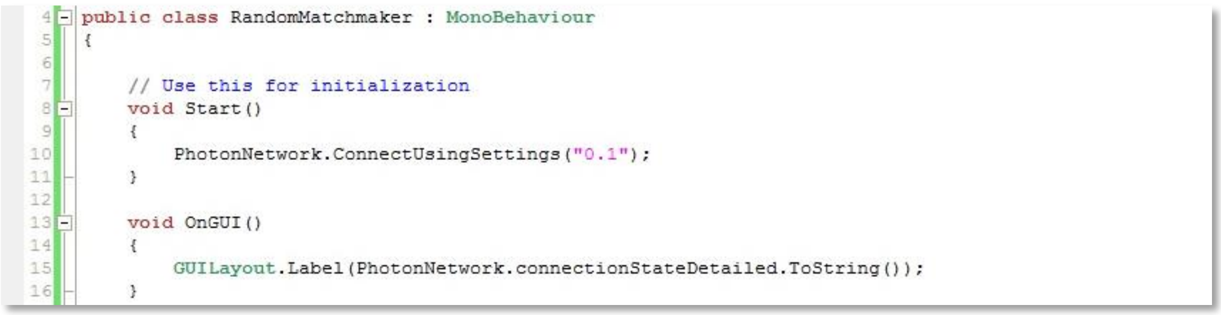
Before we do anything else, we need to get our players into a room. In a room, we can move around and let others watch it.

Create a folder “Marco Polo” and a new C# script: “RandomMatchmaker”.

The script contains Unity’s Start() method - the perfect place to immediately connect to the Photon Cloud. The most important class in the PUN package is called PhotonNetwork. It’s similar to Unity’s Network class and contains almost all methods we’re going to use.

We did the setup with the Wizard, so we can use PhotonNetwork.ConnectUsingSettings() and pass “0.1” as gameVersion. This will read our settings and use them. The gameVersion should be any short string as identification for this client.

If started this would get us connected and into the lobby. However, we would not notice any of that. Let’s show some state! With a minimum of gui, the code looks like this:



```
public class RandomMatchmaker : MonoBehaviour
{

    // Use this for initialization
    void Start()
    {
        PhotonNetwork.ConnectUsingSettings("0.1");
    }

    void OnGUI()
    {
        GUILayout.Label(PhotonNetwork.connectionStateDetailed.ToString());
    }
```

The script is not yet in the scene. We create a new, empty GameObject and name it “Scripts”. This makes it easier to find the scripts later on and we don’t rely on the camera being active all the time.

PUN is calling

Like Unity, PUN will call certain methods in our code when something interesting is happening. Currently, we are interested in something like “arrived in the lobby” or “found a room” and “didn’t find a room”. The list of methods PUN will call can be found in the documentation.

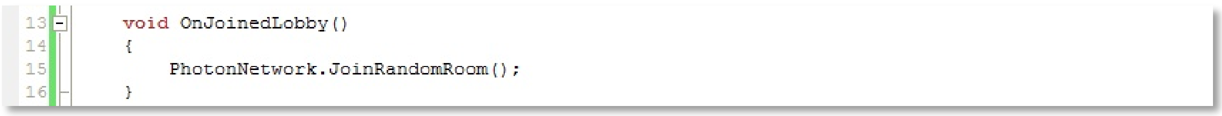
There is also an enum with all those method names in code. So autocompletition in MonoDevelop can give us a quick hint which methods our script might have. Type PhotonNetworkingMessage and the dot and a dropdown will list the names.

The methods can be implemented with void as return type and no parameter.

OnJoinedLobby

One of the callback methods in PUN is OnJoinedLobby. It's called when PUN got you into a lobby.

For the time being, we ignore the list of Rooms we now got and instead try to get into a room quickly. The PhotonNetwork class has a JoinRandomRoom() method. This tries to get us into any room. Let's try.



```
void OnJoinedLobby()
{
    PhotonNetwork.JoinRandomRoom();
}
```

Obviously, JoinRandomRoom() does not really work for some reason. The current code stays in the Lobby.

Handling errors

In case of errors, PUN makes use the Log and the Console. Now is a good time to open it in the Editor (Ctrl+Shift+C). There is a message:

"joinrandom failed, client stays on masterserver: OperationResponse 225: ReturnCode: 32760 (No match found)."

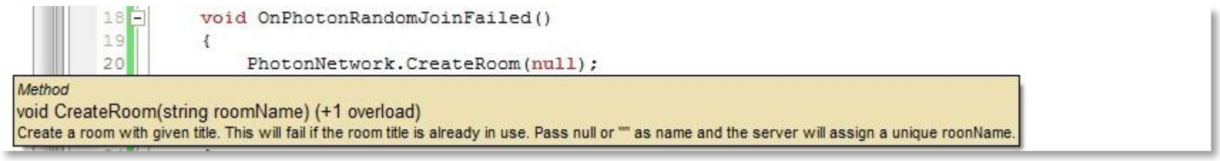
Ok, so there is no match. This can happen. And we can fix this.

A check in the PhotonNetworkingMessage enum shows there is a callback for exactly this situation: OnPhotonRandomJoinFailed. Don't mix it up with the similar OnPhotonJoinRoomFailed.

Creating a room

Joining a random room fails if no one else is playing or if all rooms are maxed out with players. We could show this to our player (GUI work left as exercise) and let her retry in a while. We skip this and simply create a room.

We implement OnPhotonRandomJoinFailed() and lookout for a "create room" method in PhotonNetwork. There is a perfect match: CreateRoom(). The tooltip explains that there are two overloads and if we pass null as room name it will become a GUID. As we don't show room names yet, we don't care and pass null.



```
void OnPhotonRandomJoinFailed()
{
    PhotonNetwork.CreateRoom(null);
```

```
Method
void CreateRoom(string roomName) (+1 overload)
Create a room with given title. This will fail if the room title is already in use. Pass null or "" as name and the server will assign a unique roonName.
```

Try out this code by running it. The detailed state is changing more often than before and ends on "Joined". We're in some room!

Marco Polo: Syncing Positions

We've seen the Vikings running around, so let's try to do the same. First, get a "Monster" character from the Asset Store.

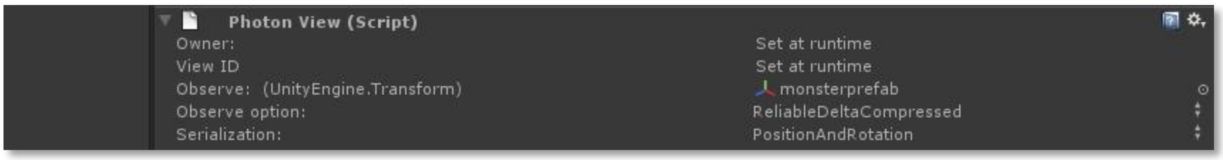
After import, we will do some preparation for later use: Rename the folder “character1” to “Resources”. We will instantiate this prefab by name and this means it must be in a Resources folder. Add a PhotonView component to the “monsterprefab”. They are found in the Components menu under “Miscellaneous”.

PhotonView

If you know Unity’s Networking: A PhotonView is PUN’s equivalent of a NetworkView. PUN needs a PhotonView per instantiated prefab to keep the networking reference (known as PhotonViewId), the owner of the object and a reference to an “observed” component.

PUN keeps track of the PhotonViews it instantiated locally and at runtime regular checks of observed components will send updates to the other clients. More observed objects mean more work and network traffic.

To setup the new PhotonView to observe the translation of its “monsterprefab”, drag & drop the translation to the “observed” field. We don’t need to change the other settings of the PhotonView.



Photon View (Script)

Owner:	Set at runtime
View ID	Set at runtime
Observe: (UnityEngine.Transform)	monsterprefab
Observe option:	ReliableDeltaCompressed
Serialization:	PositionAndRotation

Adding Monsters

To instantiate a monster use PhotonNetwork.Instantiate(). Don’t mix it up with Unity’s Instantiate() or Network.Instantiate(). This makes sure the view gets instantiated on the other clients, too.

In OnJoinedRoom(), we call Instantiate() like so:

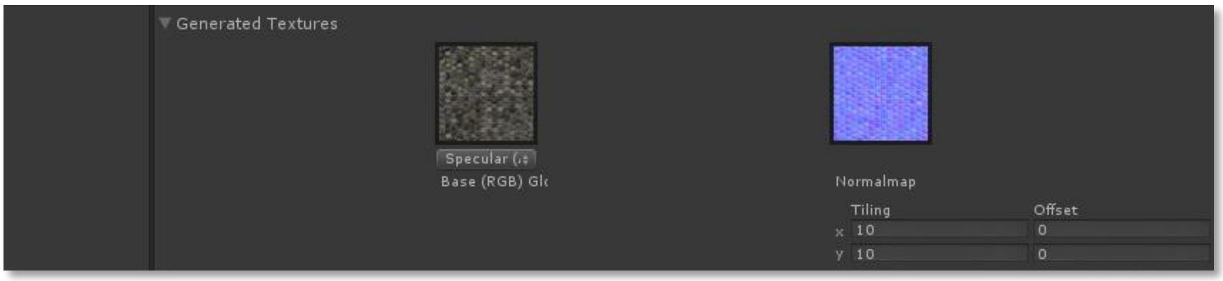
```
GameObject monster = PhotonNetwork.Instantiate("monsterprefab", Vector3.zero, Quaternion.identity, 0);
```

Running this will show a monster pop up and start falling. Poor monster. Let’s create some ground.

Grounded

The next steps are common Unity “work”: Create a directional light in the scene and rotate it to point down in some angle - imitating the sun.

Add a plane, scale it to 10, 1, 10 and move it to 0, 0, 0. Let’s try some substances from the Asset Store. Download the “Eighteen Free Substances” package and import it. The “Pavement_01” looks nice, so we open its folder and apply the material to the plane. In the inspector, change the texture tiling from 1 to 10 for x and y.



Generated Textures

Specular (
Base (RGB) Gl

Normalmap

	Tiling	Offset
x	10	0
y	10	0

Fight for Control

Checking the progress with two clients, we notice that all monsters are moved by our key input. We cloned the monster including the enabled myThirdPersonController component! There are countless ways to do this.

One simple way is to disable the script in the “monsterprefab” (!) and enable it only when we instantiate a monster for “our” client. Instantiate() returns the GameObject it created, so this is no big deal. Aside from one handicap: The myThirdPersonController is a UnityScript script.

To make any script of one language available to the scripts of another language, you can move it to a “Plugins” folder in your project. Create this folder and move the script over (drag & drop in the editor).

After instantiating our monster, we can grab its myThirdPersonController component and activate it.

```
void OnJoinedRoom()
{
    GameObject monster = PhotonNetwork.Instantiate("monsterprefab", Vector3.zero, Quaternion.identity, 0);
    myThirdPersonController controller = monster.GetComponent<myThirdPersonController>();
    controller.enabled = true;
    myMonsterPv = monster.GetComponent<PhotonView>();
}
```

Smooth Moves

So far, monsters of other players are moved but don’t walk. The animation is missing. Also, the position updates are not smooth. This can be fixed by a bit of code.

We need another script. Create a “NetworkCharacter” C# script in the Marco Polo folder. Add it to the “monsterprefab” and make it the observed component of the PhotonView (drag & drop).

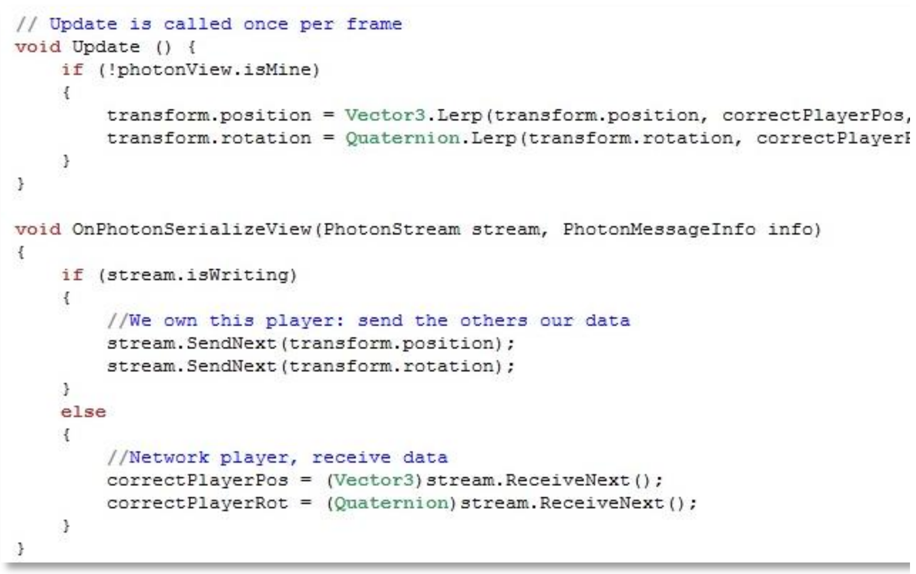
While a script is observed, the PhotonView regularly calls the method OnPhotonSerializeView(). The task of this is to create the info we want to pass to others and to handle such incoming info – depending on who created the PhotonView.

A PhotonStream is passed to OnPhotonSerializeView() and the value of isWriting tells us if we need to write or read remote data from it. First, we send and receive Position and Rotation. This has the same effect as the PhotonView without script:

```
if (stream.isWriting)
{
    //We own this player: send the others our data
    stream.SendNext(transform.position);
    stream.SendNext(transform.rotation);
}
else
{
    //Network player, receive data
    transform.position = (Vector3)stream.ReceiveNext();
    transform.rotation = (Quaternion)stream.ReceiveNext();
```

A simple way to smoothly “push” a monster to its correct position is to Lerp it there over time.

We add Vector3 correctPlayerPos and Quaternion correctPlayerRot to the NetworkCharacter. In OnPhotonSerializeView we store the new values and apply them (bit by bit) in Update().



```
// Update is called once per frame
void Update () {
    if (!photonView.isMine)
    {
        transform.position = Vector3.Lerp(transform.position, correctPlayerPos,
        transform.rotation = Quaternion.Lerp(transform.rotation, correctPlayer
    }
}

void OnPhotonSerializeView(PhotonStream stream, PhotonMessageInfo info)
{
    if (stream.isWriting)
    {
        //We own this player: send the others our data
        stream.SendNext(transform.position);
        stream.SendNext(transform.rotation);
    }
    else
    {
        //Network player, receive data
        correctPlayerPos = (Vector3)stream.ReceiveNext();
        correctPlayerRot = (Quaternion)stream.ReceiveNext();
    }
}
```

Monsters of other players won't be using the exact same speed this way but they will reach the same point in a short time. For some games it might be more important to have the characters at the correct position than how exactly they get there.

Adding Animation

We can't grab the animation state easily – there are multiple ones, blended, etc - so we have to find another way to sync this.

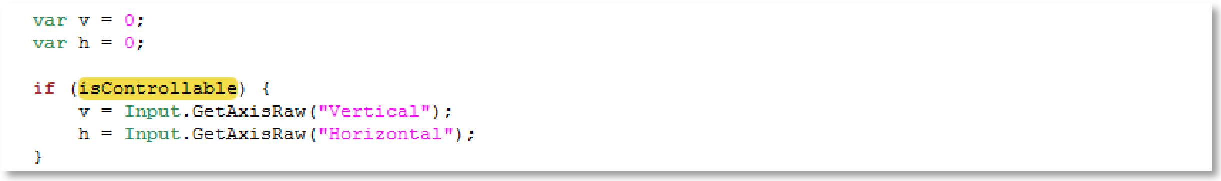
The myThirdPersonController class (short: controller) uses a _characterState to trigger almost all animations. The state is derived from the CharacterController.velocity of our monster, which is great locally but it won't work remotely as we just re-position other monsters. We could pass the state of "our" monster though and apply that on the other clients. This should work nicely, if we didn't disable the controller in the first place.

Obviously, we need another way to ignore our input for scripts on "remote monsters". The variable isControllable looks good: It's never set and not always checked but we can fix both.

First, enable the myThirdPersonController of the prefab and remove the "enabling" code in RandomMatchmaker.OnJoinedRoom().

Make isControllable and _characterState public and find all places where Input is used. Check isControllable first and ignore input if it's not true.

We read input axes only if controllable:



```
var v = 0;
var h = 0;

if (isControllable) {
    v = Input.GetAxisRaw("Vertical");
    h = Input.GetAxisRaw("Horizontal");
}
```

We suppress “firing” unless isControllable:

```
    if (isControllable && Input.GetButton("Fire1"))
    {
        animation[attackPoseAnimation.name].AddMixingTransform(buik);
        animation.CrossFade(attackPoseAnimation.name, 0.2);
        animation.CrossFadeQueued(idleAnimation.name, 1.0);
    }
```

And no one is jumping, unless controlled:

```
function Update() {

    if (isControllable && Input.GetButtonDown ("Jump"))
    {
        lastJumpButtonTime = Time.time;
    }
```

Note: The original script was resetting Input at the beginning of Update(). That’s gone. We have multiple scripts running and we don’t mind which one is executed first. Don’t reset Input.

Now, take a look how animations are applied. Except for “idle”, the animation is set by the current state and faded in.

The condition for the “idle” animation should also check isControllable. Then we detect and set the idle state which would otherwise never be synced. In the code block below, add a “if idle state” and apply the animation accordingly:

```
 if(this.isControllable && controller.velocity.sqrMagnitude < 0.5) {
     _animation.CrossFade(idleAnimation.name);
     this._characterState = CharacterState.Idle;
 }
 else
 {
     if(_characterState == CharacterState.Idle) {
         _animation.CrossFade(idleAnimation.name);
     }
     if(_characterState == CharacterState.Running) {
         _animation[runAnimation.name].speed = runMaxAnimationSpeed;
         if (this.isControllable)
         {
             _animation[runAnimation.name].speed = Mathf.Clamp(controller.velocity.magnitude, 0.0, runMaxA
         }

         _animation.CrossFade(runAnimation.name);
     }
```

While the original script modifies the animation speed by velocity, we can’t do that for other payers’ monsters. Just skip this part and apply the desired animation speed. This could be done better but looks ok for this case.

Starting this version finally syncs animations and has smoothed movement:

Notice how the Editor (left) shows the same animation as the standalone on top (right).

Shout and Respond

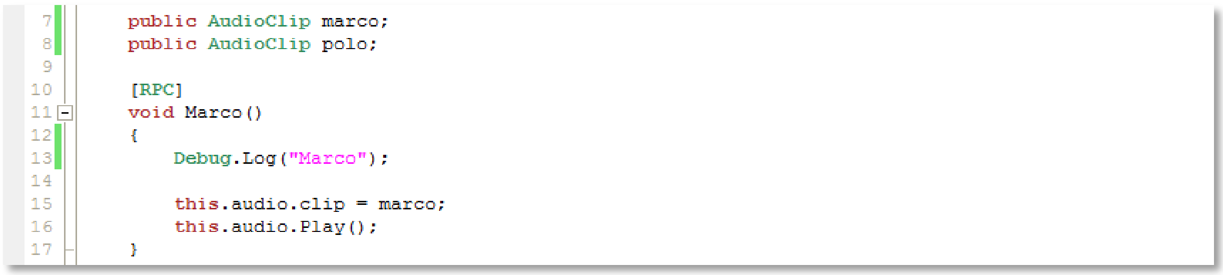
It would be cool if our monsters actually talk. We could record something or just look around the web for a text to speech synthesizer. No matter how: Get two audio clips for this (e.g. from www.fromtexttospeech.com).

A PhotonView has a method called "RPC" which is short for "Remote Procedure Call". So the point of this is to call another method? Boring!

The good thing about an RPC is: It will call the method on other clients in the same room. For convenience, it can also call the method on "this" client.

The RPC method can't just call any method but only those marked as "RPC". To make any method callable remotely, you have to add the attribute RPC to it. This is the same as in Unity's Networking (described here).

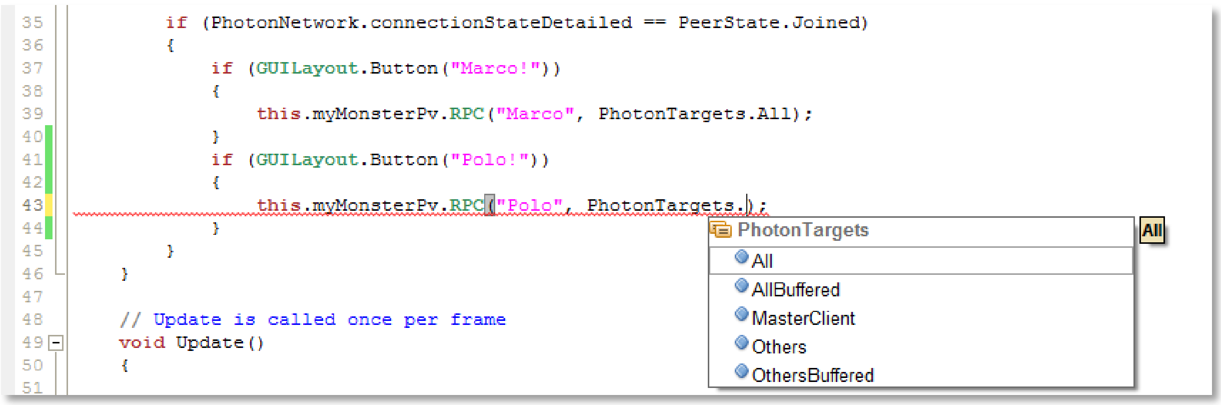
Nice. We can use RPC methods easily to make our monster shout – locally and on other player's clients. In folder "Marco Polo", create another script "AudioRpc". Add the methods "Marco" and "Polo" with the RPC attribute. Also add public AudioClip fields, so you can reference the sound files to play them.



```
    public AudioClip marco;
    public AudioClip polo;

    [RPC]
    void Marco()
    {
        Debug.Log("Marco");

        this.audio.clip = marco;
        this.audio.Play();
    }
```

Add this "AudioRpc" now to the "monsterprefab" and apply the two sound files to the references.

For convenience, let's just add two buttons to the RandomMatchmaker GUI and call the RPCs there:



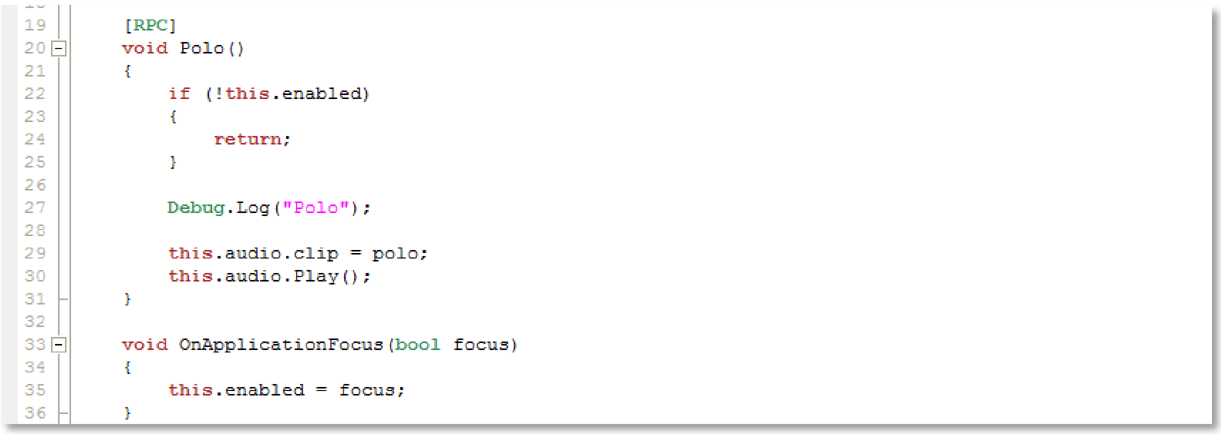
```
        if (PhotonNetwork.connectionStateDetailed == PeerState.Joined)
        {
            if (GUILayout.Button("Marco!"))
            {
                this.myMonsterPv.RPC("Marco", PhotonTargets.All);
            }
            if (GUILayout.Button("Polo!"))
            {
                this.myMonsterPv.RPC("Polo", PhotonTargets.);
            }
        }
    }

    // Update is called once per frame
    void Update()
    {
```

Note that there are different PhotonTargets options. We're using "All" now, to play the sound locally, too.

If you run two clients now and click either button, the voice will echo: Both clients will play the sound. The "remote" player will have a short delay (depending on the networking delay).

This only makes a difference while testing but let's disable the RPC audio in background. Implement OnApplicationFocus in AudioRPC and disable the script when the app loses focus. Then test again. The echo is still there.

RPCs are called on disables scripts, too. This could be annoying but most likely it's a good thing. If you want to skip RPCs on disabled scripts, check: .enabled.



```
    [RPC]
    void Polo()
    {
        if (!this.enabled)
        {
            return;
        }

        Debug.Log("Polo");

        this.audio.clip = polo;
        this.audio.Play();
    }

    void OnApplicationFocus(bool focus)
    {
        this.enabled = focus;
    }
```

Switch "It"

So far, we can call out and answer. What's missing is a basic game logic to know who is "it" and to tag others.

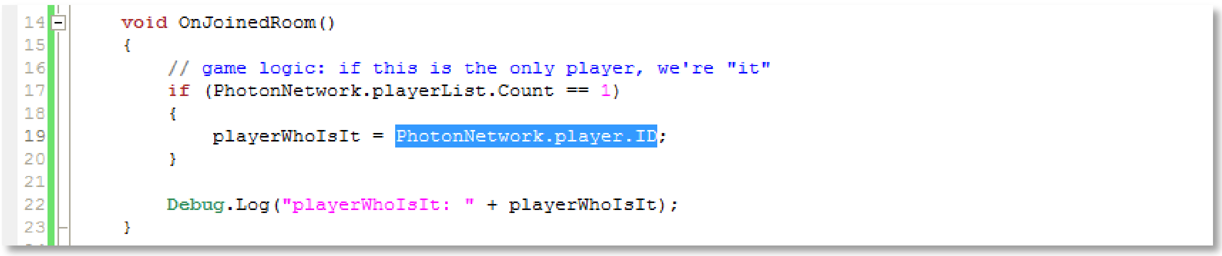
For our first piece of game logic, we create a C# script "GameLogic" in the "Marco Polo" folder and add it to the "Scripts" object.

The first player who gets into a room is "it". We know we just got into a room when the method OnJoinedRoom() is called and by checking PhotonNetwork.playerList we can easily find out if we're the first one.

Player IDs

Photon uses a player "ID" or "playerNumber" to mark each of the players in a room. It's an int, which makes it relatively small in terms of data. As it's not re-assigned ever, we can "talk" about players in a room by this ID.

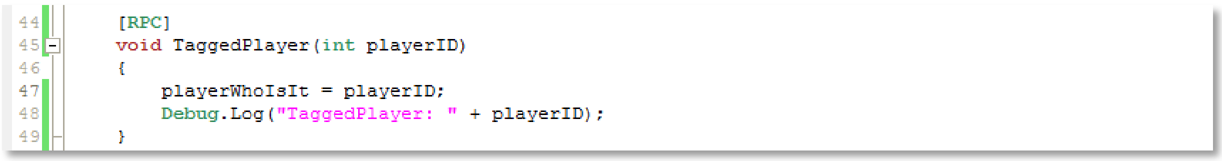
The ID of our local client is stored in PhotonNetwork.player.ID. If we're alone, we'll save this into "playerWhoIsIt", a static field which is easy to access it in different scripts.



```
    void OnJoinedRoom()
    {
        // game logic: if this is the only player, we're "it"
        if (PhotonNetwork.playerList.Count == 1)
        {
            playerWhoIsIt = PhotonNetwork.player.ID;
        }

        Debug.Log("playerWhoIsIt: " + playerWhoIsIt);
    }
```

When more players are joining, we need to let them know who is currently "it". PUN calls OnPhotonPlayerConnected() when someone new arrives , so reacting to that is easy.

We will create a method "TaggedPlayer" to set "playerWhoIsIt" and make it a RPC, so it can be called by anyone.



```
    [RPC]
    void TaggedPlayer(int playerID)
    {
        playerWhoIsIt = playerID;
        Debug.Log("TaggedPlayer: " + playerID);
    }
```

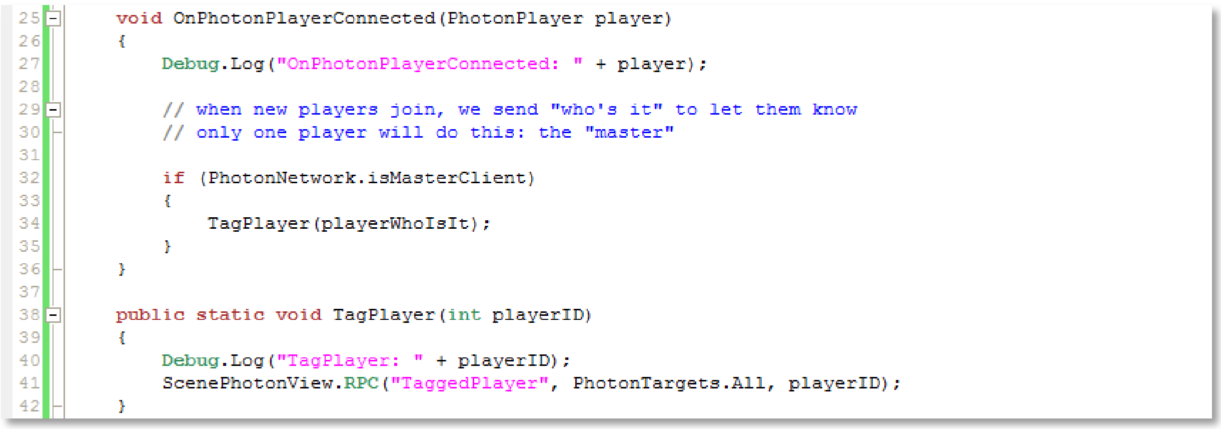
Scenic PhotonView

To call an RPC, we need a PhotonView and as it's logically not the task of our monsters to remember or set the player who is "it", we add another PhotonView.

Add a PhotonView to the "Scripts" object in the scene. It's there all the time and can be used by any player. This is often useful to sync states. Our GameLogic will find the PhotonView in Start() and make it accessible as static "ScenePhotonView".

We want just one player to update newcomers. This is a good task for PUN's "master client". This is always the player with the lowest ID in a room. Each client can check if it is currently the master by: PhotonNetwork.isMasterClient.

Combined, this looks like so:

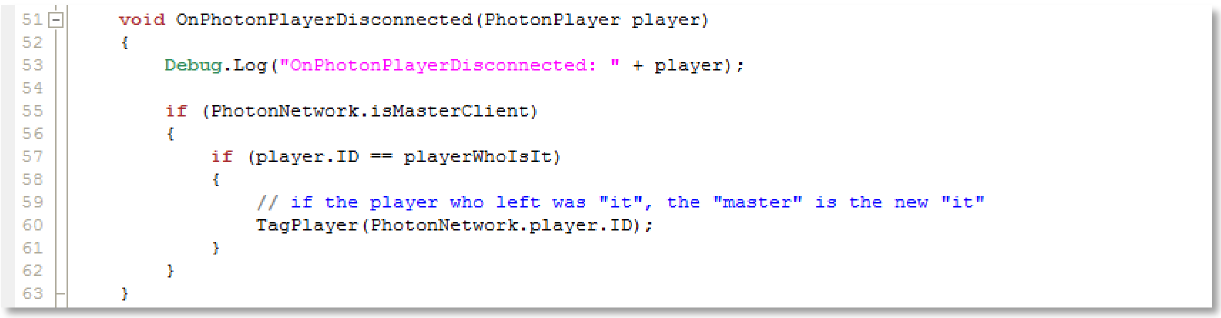


```
    void OnPhotonPlayerConnected(PhotonPlayer player)
    {
        Debug.Log("OnPhotonPlayerConnected: " + player);

        // when new players join, we send "who's it" to let them know
        // only one player will do this: the "master"

        if (PhotonNetwork.isMasterClient)
        {
            TagPlayer(playerWhoIsIt);
        }
    }

    public static void TagPlayer(int playerID)
    {
        Debug.Log("TagPlayer: " + playerID);
        ScenePhotonView.RPC("TaggedPlayer", PhotonTargets.All, playerID);
    }
```

Left behind

Players can leave anytime and maybe we lose our “seeking” player this way.

PUN will call OnPhotonPlayerDisconnected() when anyone leaves. We should find out if the player who left is “it” and assign a new one. Again, this work is only done by one player, the master.



```
    void OnPhotonPlayerDisconnected(PhotonPlayer player)
    {
        Debug.Log("OnPhotonPlayerDisconnected: " + player);

        if (PhotonNetwork.isMasterClient)
        {
            if (player.ID == playerWhoIsIt)
            {
                // if the player who left was "it", the "master" is the new "it"
                TagPlayer(PhotonNetwork.player.ID);
            }
        }
    }
```

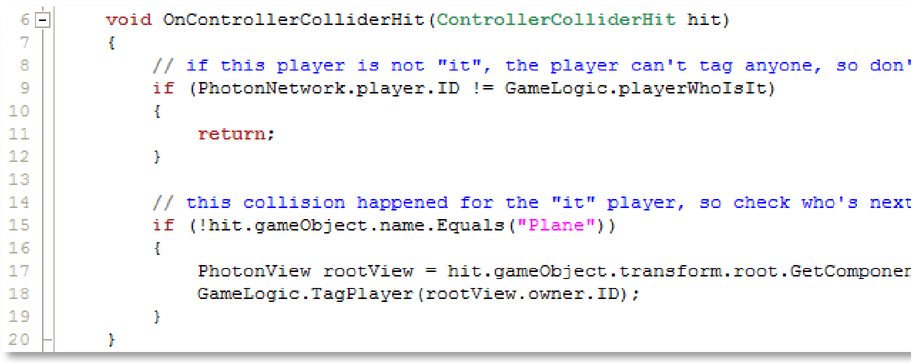
Being Hit

We still need a way to tag others. This is a job for collision detection on the characters we have. Looking at the monster, we find two spheres at the hands. Perfect.

We create a new C# script “CollisionDetector” to the “Marco Polo” folder and drag it to the “monsterprefab”. To get the collision, we implement OnControllerColliderHit().

As a first step, we log all passed “hit” objects and find that the fingers also collide with the plane! As simple workaround, we will ignore the hits with the Plane. In best case, we would setup the colliders into separate layers, so the plane does not affect our hand spheres.

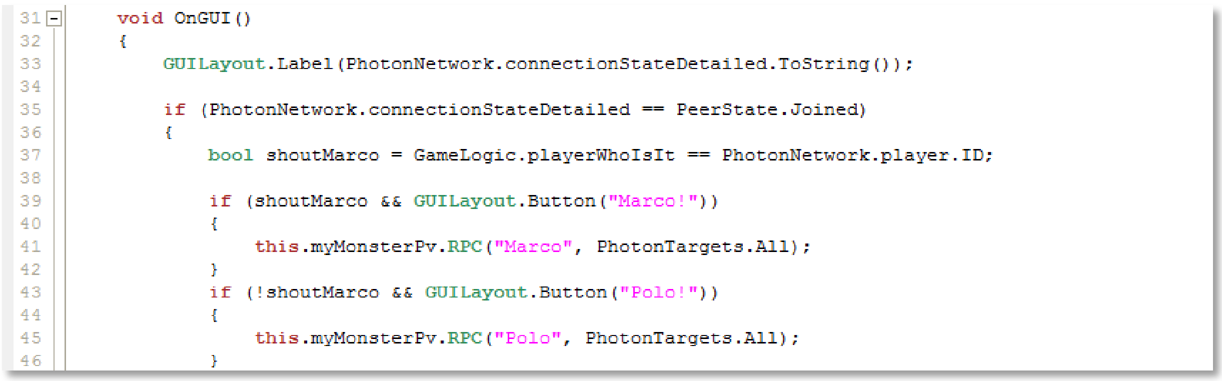
We also just care about hits if our player is currently “it”. If we hit some other monster, tag it!



```
void OnControllerColliderHit(ControllerColliderHit hit)
{
    // if this player is not "it", the player can't tag anyone, so don'
    if (PhotonNetwork.player.ID != GameLogic.playerWhoIsIt)
    {
        return;
    }

    // this collision happened for the "it" player, so check who's next
    if (!hit.gameObject.name.Equals("Plane"))
    {
        PhotonView rootView = hit.gameObject.transform.root.GetComponer
        GameLogic.TagPlayer(rootView.owner.ID);
    }
}
```

Using the static TagPlayer() method, it's very easy to tag another player.

The buttons for "Marco" and "Polo" are still both shown for any player. This can be changed easily by checking "playerWhoIsIt", too. We show "Marco" only we are "it".



```
void OnGUI()
{
    GUILayout.Label(PhotonNetwork.connectionStateDetailed.ToString());

    if (PhotonNetwork.connectionStateDetailed == PeerState.Joined)
    {
        bool shoutMarco = GameLogic.playerWhoIsIt == PhotonNetwork.player.ID;

        if (shoutMarco && GUILayout.Button("Marco!"))
        {
            this.myMonsterPv.RPC("Marco", PhotonTargets.All);
        }
        if (!shoutMarco && GUILayout.Button("Polo!"))
        {
            this.myMonsterPv.RPC("Polo", PhotonTargets.All);
        }
```

Conclusion

We now have small prototype of a simple game. A lot is missing, sure, but we definitely have the basics covered. Our game finds players in some random room, syncs positions, smoothes out movement and has a basic game logic on top of that. You learned to use player IDs, the "Master Client", implement PUN Networking Messages and a lot more.

It doesn't sound that scary anymore to hide players who are not "it", to sync the "attack"animation with a RPC or to delay the next tagging, after someone was just tagged. Also, as we figured out how to control our character, controlling the camera in a similar way couldn't be that hard anymore.

Unleash your monsters! :)