



Vostopia Avatars and Photon Cloud

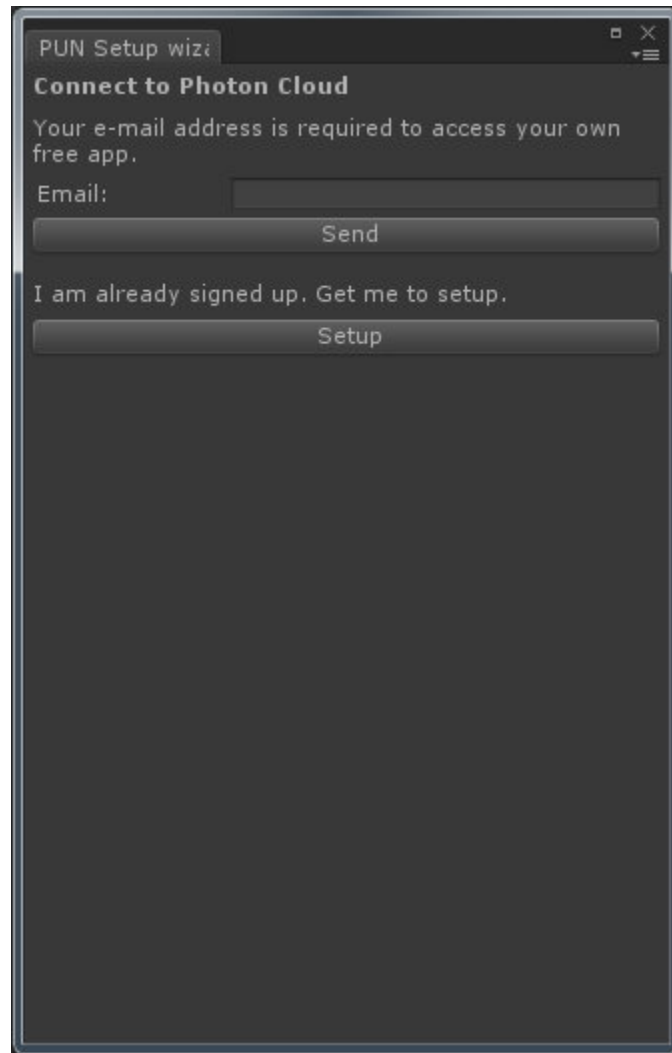
Introduction

This document describes Vostopia Avatars + Photon Cloud Demo, and describes the differences between a single player scene that uses the vostopia.com avatars and a multiplayer game that uses exitgames.com Photon Cloud to synchronise avatars between multiple clients.

It is assumed that You are already familiar with the contents of the “Getting Started with Vostopia Avatars” documentation and have managed to set up a scene containing a VostopiaClientInitializer and an avatar so that the user can log in to vostopia.com, have their avatar load and wander around.

Getting Started

When you first open the project you may be prompted to set up your Photon cloud app ID:

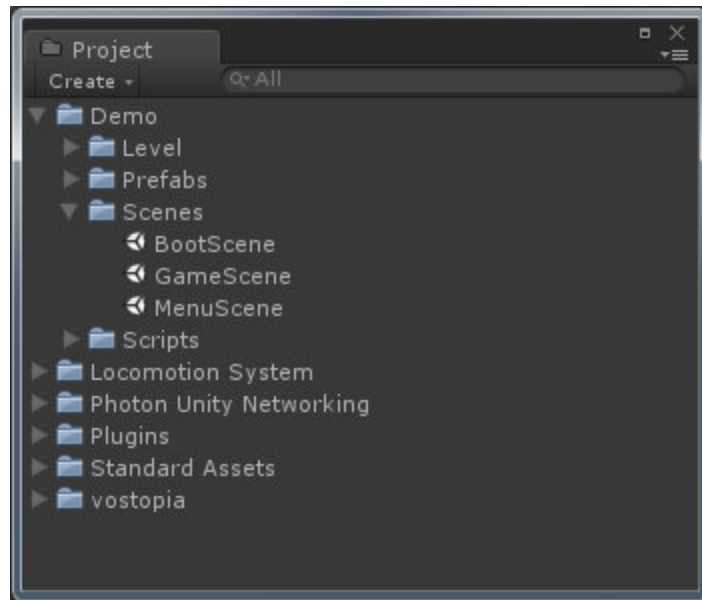


The image shows a dark-themed dialog box titled "PUN Setup wizard". It has a close button (X) in the top right corner. The main heading is "Connect to Photon Cloud". Below this, it says "Your e-mail address is required to access your own free app." There is an "Email:" label followed by a text input field. Below the input field is a "Send" button. Further down, it says "I am already signed up. Get me to setup." followed by a "Setup" button. The bottom half of the dialog box is empty.

Either enter your email and click "Send" to be sent an Application ID, or if you have already signed up manually on the [photon cloud website](#) click "Setup" and enter your Application ID.

Demo Structure

All of the demo assets are located in subfolders. Everything relevant to this document is located in the “Demo” folder. The other root folders were imported as-is from their respective sources.



The demo is that it is split up into three separate scenes which live inside the “Scenes” folder: “BootScene”, “MenuScene” and “GameScene”. The reason for this split is that we need at least a rudimentary GUI for displaying and joining available Photon Rooms (This GUI exists in MenuScene) and then the actual level (or potentially multiple levels) that are loaded when you join a room.

BootScene

BootScene is the scene that is loaded first and is very simple. All it does is handle the one-time setup of the VostopiaClientInitializer and then loads the MenuScene via the BootLoader script:

```
void Start ()
{
    if (VostopiaClientInitializer != null)
    {
        VostopiaClientInitializer.Connect();

        if (MenuSceneName != null && MenuSceneName != "")
        {
            Application.LoadLevel(MenuSceneName);
        }
    }
}
```

The VostopiaClientInitializer script uses DontDestroyOnLoad(this) which keeps the object loaded even if we load another scene. It does this to maintain a persistent connection to the vostopia.com servers even as other scenes are loaded and unloaded. It is because of this persistent behaviour that we keep it in the BootScene which is only loaded once at startup. If there were VostopiaClientInitializers in the GameScene or MenuScene then we would accumulate multiple persistent versions of it as we loaded and unloaded those scene files.

The Demo calls VostopiaClientInitializer.Connect() right at the start, but there is no reason you couldn't delay this until the appropriate point in your front end flow so as to control when the vostopia.com login appears.

MenuScene

MenuScene is also relatively simple. It contains a single script that displays the front end GUI. The Menu script contains a simple state machine that waits for the network connections to vostopia.com and the photon cloud to connect, and then uses the PhotonNetwork API to display a list of available rooms. The script also has an attribute that is set up to contain a list of available levels that can be used to create new rooms. The details of the specific PhotonNetwork calls are beyond the scope of this document but are fully described in the documentation and demos on the Exit Games website. For the demo the most significant part is the OnJoinedRoom() callback:

```
private void OnJoinedRoom()
{
    if (PhotonNetwork.room != null)
    {
        PhotonNetwork.isMessageQueueRunning = false;
        Application.LoadLevel(mSelectedLevelName);
        mMenuState = MenuState.Loading;
    }
}
```

This callback fires immediately after the client has first joined a room, either by creating a new room, or joining an existing one. Immediately after joining a room Photon sends lots of messages detailing the existence and state of objects (such as other players) that already exist in the shared room, however at this point we are still in our menu scene and don't want to create those objects until we have loaded the actual game scene. Disabling the message queue as we join the room and then re-enabling it after we have loaded the game scene delays the creation of these objects until we have the correct scene loaded.

GameScene

The Game scene is where everything happens. The LevelLogic Script (in the associated GameObject) renables the Photon message queue (which was disabled by the Menu script when we first joined the room) so that the initial Photon messages detailing the existence of other entities in the room are processed:

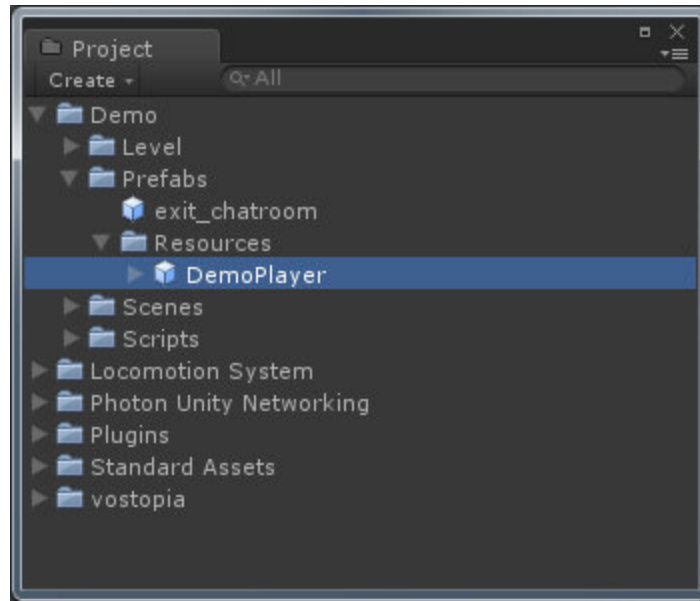
```
void Awake ()
{
    PhotonNetwork.isMessageQueueRunning = true;
}
```

The first major departure from a standalone scene with an avatar in it is that the avatar isn't actually present in the scene at load time, instead it exists as a prefab that is instantiated at run-time. This is because we don't want a single avatar that is shared by all of the players in the room, rather we want each player to create a new unique avatar when they join the room. The LevelLogic script is also responsible for creating the local player avatar using the referenced "PlayerPrefab":

```
void Start()
{
    if (PlayerPrefab != null)
    {
        GameObject player = (GameObject)PhotonNetwork.Instantiate(
            PlayerPrefab,
            this.transform.position,
            this.transform.rotation,
            0
        );

        if (player != null)
        {
            FollowCam camera = Camera.mainCamera.GetComponent<FollowCam>();
            if (camera != null)
            {
                camera.Target = player;
            }
        }
    }
}
```

It is worth noting that the referenced PlayerPrefab needs to be located in a folder called "Resources":

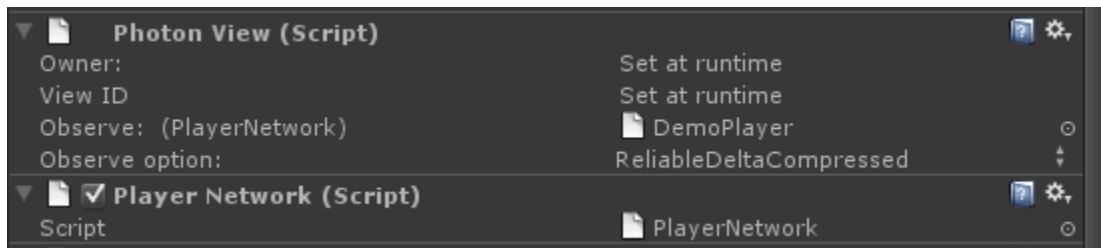


This is because when a remote client instantiates a new GameObject that is to be shared across the network that information is broadcast to the other clients which in turn use the Unity function `Resources.Load()` to instantiate a local copy of the appropriate GameObject on the local client. `Resources.Load()` takes a single string which corresponds to the name of the prefab to create, and it requires these resources to exist in a folder called Resources as described in the Unity documentation [here](#).

Since the player is being created dynamically (and since there are potentially going to be multiple instances of the player prefab as other remote clients join the room) the camera system can't know in advance which player to follow. This is why when the LevelLogic script creates the local player it also tells the camera to start following that player.

Player Prefab

The player prefab is mostly the same as a single player vostopia avatar but with a few additions/alterations. The first addition is the PlayerNetwork Script (and an associated PhotonView):



This script does three things, firstly it enables or disables the controller based on whether or not the avatar is associated with the local player or a remote client:

```
void Start()
```

```

{
    if (photonView.isMine)
    {
        controllerScript.enabled = true;
    }
    else
    {
        controllerScript.enabled = false;
    }
    gameObject.name = gameObject.name + photonView.viewID.ID;
}

```

The second thing it does is to implement `OnPhotonSerializeView()` to read and write the position and orientation of the avatar whenever the attached `PhotonView` updates:

```

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();
    }
}

```

Note, when receiving we don't directly write to `transform.position`, rather we cache the location in a member variable, which brings us to the the third and final thing the `PlayerNetwork` script does, which is interpolate the location and orientation to smooth out the motion of remote players:

```

void Update()
{
    if (!photonView.isMine)
    {
        transform.position = Vector3.Lerp(
            transform.position,

```



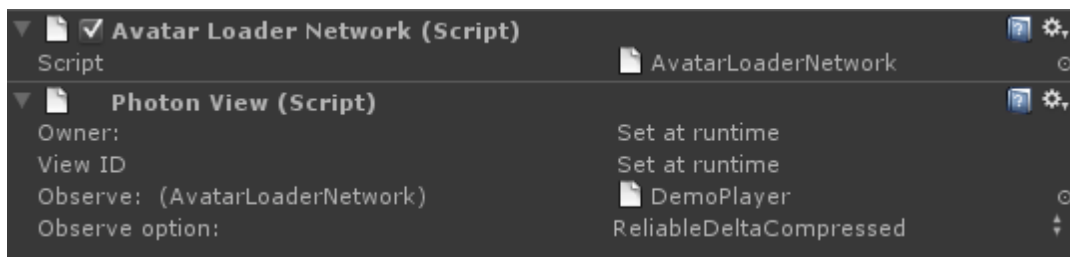
```

        correctPlayerPos,
        Time.deltaTime * 5
    );
    transform.rotation = Quaternion.Lerp(
        transform.rotation,
        correctPlayerRot,
        Time.deltaTime * 5
    );
}
}

```

There are multiple approaches to interpolating/predicting position based on unreliable network packets. The method used here is intended to be one of the simplest possible in order to demonstrate the principle in the clearest way possible. Depending on the style and pacing of your game you will probably want to play around with the interpolation.

The other change to the Player prefab relates to how the avatar is loaded. The default AvatarLoaderUserId script, when attached to an avatar loads the avatar associated with the currently logged in user. However in a multiplayer network environment we want the avatars of remote clients to load the appearance of each remote user. In order to facilitate this we remove the AvatarLoaderUserId script from our player prefab and replace it with the AvatarLoaderNetwork script. We also add a PhotonView that is set to observe the AvatarLoaderNetwork so that the avatars can communicate their state across the network.



The first thing the AvatarLoaderNetwork does is check if the Avatar is local, and if so load the default outfit for the currently logged in user.

```

void Start()
{
    if (photonView.isMine)
    {
        mLoadingController.SetDefaultOutfit("");
    }
}

```

The AvatarLoaderNetwork also implements OnPhotonSerializeView to send/receive the userID associated with this avatar.

```

void OnPhotonSerializeView(
    PhotonStream stream,
    PhotonMessageInfo info
)
{
    if (stream.isWriting)
    {
        string userId = VostopiaClient.Authentication.ActiveUser.Id;
        //We own this player: send the others our data
        stream.SendNext(userId);
    }
    else
    {
        //Network player, receive data
        string userId = (string)stream.ReceiveNext();
        mLoadingController.SetDefaultOutfit(userId);
    }
}

```

When the avatar is owned by the local client the currently logged in userID is written to the stream, however because it is not (in theory) changing and the associated PhotonView has it's synchronization type set to ReliableDeltaCompressed it will only be sent over the network once. When the avatar is representing a remote client then it will pull the userId from the stream and then tell the loading controller to load the default avatar outfit for that userId. Again, this message should only be received once when the avatar is first created due to the client which owns the avatar only sending out updates when the userId changes (which shouldn't be happening).

In theory this functionality could be incorporated in the PlayerNetwork script, however then the UserId would be broadcast along with the position and orientation whenever those changed. This would be slightly wasteful of bandwidth. Splitting the parts up and having two PhotonViews allows Photon to optimise the traffic.

Conclusion

Both the Photon Unity plugin and the Vostopia.com avatar system are written to be modular and as easy as possible to integrate into your game. This document has demonstrated that it requires relatively little work to get them working together to create a shared virtual environment to which multiple players can bring their customisable avatars and where they can interact.