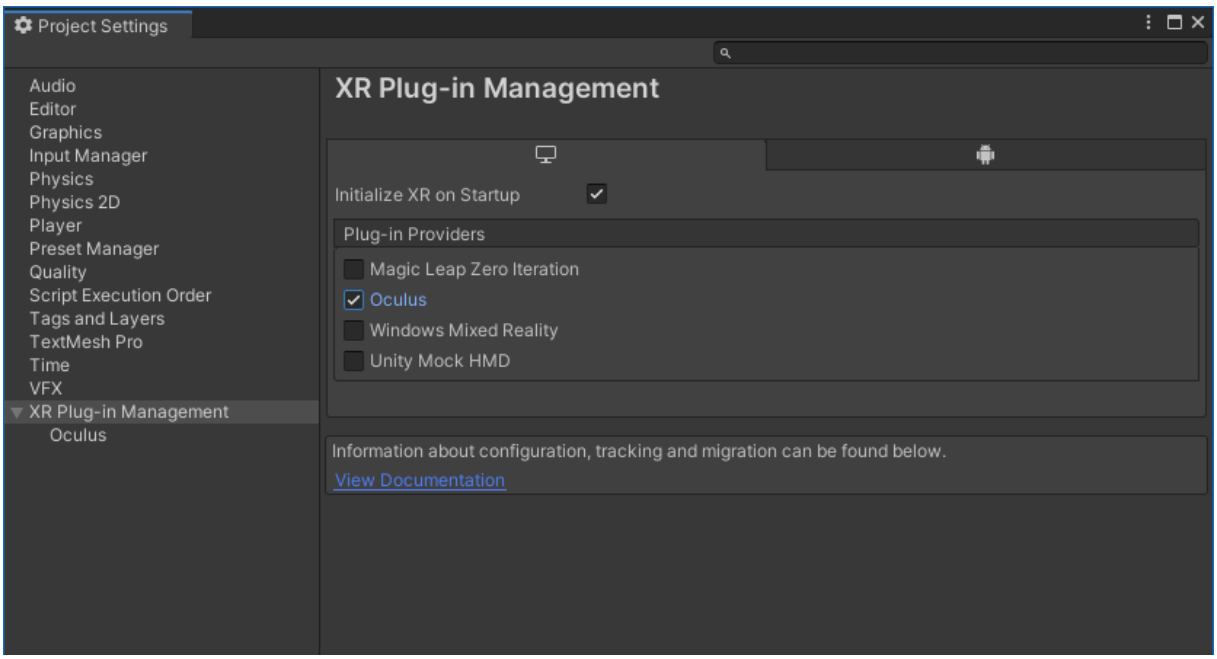
Setting Up Ubik for VR

We are using Unity's new XR Plug-in Management system. This allows you to enable a VR plugin matching your device and to interact with it through a unified interface. There is a quick guide below for each supported platform.

Oculus or Windows Mixed Reality (desktop)

1. In Unity, open the project settings window (Edit/Project Settings...) and go to the XR Plug-in Management menu.

2. Enable the plug-in here - just tick either the Oculus or Windows Mixed Reality box. Also tick the box for “Initialize XR on Startup”.



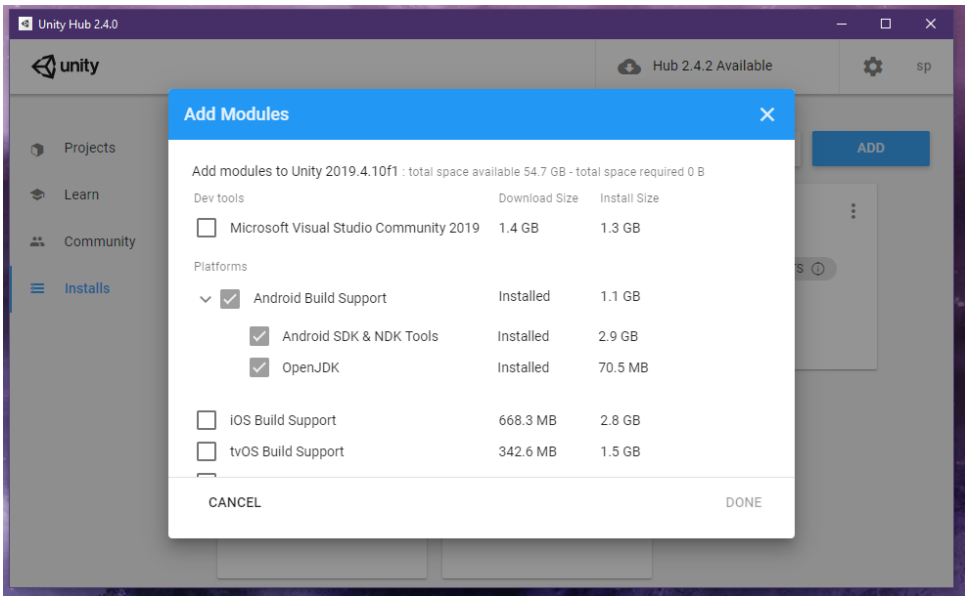
OpenVR (desktop)

1. Note that while rendering and tracking works well, this subsystem is currently missing input from the hand controllers. Unfortunately, this is a limitation with the plugin and a fix does not seem to be on the horizon.

2. Follow the instructions to download the OpenVR Unity XR plug-in here: https://github.com/ValveSoftware/unity-xr-plugin/releases/tag/installer

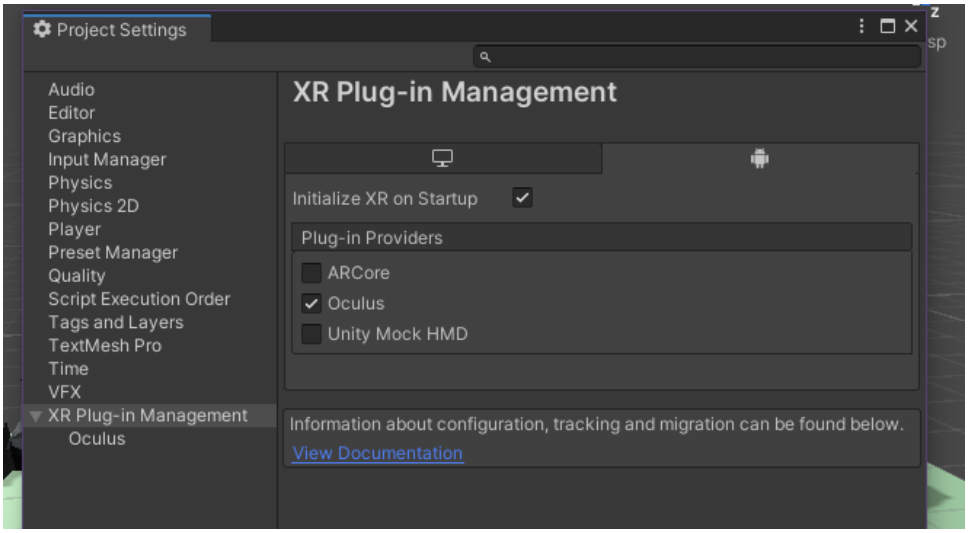
Oculus Quest

1. Install android build tools. In Unity Hub, click 'Installs' on the left-hand menu. Click the three dots in the top right corner of the box for your Unity 2019.4.x installation and select 'Add modules' from the dropdown. Select 'Android Build Support' and both subsequent options ('Android SDK & NDK Tools' and 'OpenJDK'). Wait for installation to complete, then re-open your project.



2. In Unity, open the project settings window (Edit/Project Settings...) and go to the XR Plug-in Management menu.

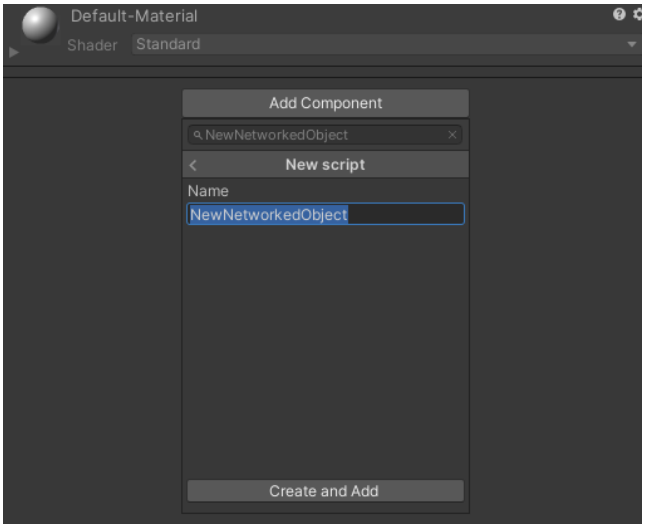
3. Click the Android tab and check the boxes for Oculus and “Initialize XR on Startup”.



4. (Optional) To create your first build for the Quest, follow the Oculus 'Enable Device for Development and Testing' guide: https://developer.oculus.com/documentation/unity/unity-enable-device/

Building a Basic Networked Object

1. Create a new Unity Script and add it to your GameObject that you want to have networked. You can do this via the inspector by clicking on “Add Component” and typing the new name.



2. Include Ubik.Messaging

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

public class NewNetworkedObject : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

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

    }
}
```

3. Inherit from INetworkObject and INetworkComponent

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4   using Ubik.Messaging;
5
6   public class NewNetworkedObject : MonoBehaviour, INetworkObject, INetworkComponent
7   {
8       // Start is called before the first frame update
9       void Start()
10      {
11
12      }
13
14      // Update is called once per frame
15      void Update()
16      {
17
18      }
19  }
```

4. Implement their interfaces
   In Visual Studio this can be done through the context menu.
   Right Click -> Quick Actions and Refactoring -> Implement interface
   Note: This will only give you the stubs. You will need to fill them in yourself in the next steps

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

public class NewNetworkedObject : MonoBehaviour, INetworkObject, INetworkComponent
{
    public NetworkId Id => throw new System.NotImplementedException();

    public void ProcessMessage(ReferenceCountedSceneGraphMessage message)
    {
        throw new System.NotImplementedException();
    }

    // Start is called before the first frame update
    void Start()
    {

    }

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

    }
}
```

5. Implement Network ID creation

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

public class NewNetworkedObject : MonoBehaviour, INetworkObject, INetworkComponent
{
    public NetworkId Id { get; } = new NetworkId();

    public void ProcessMessage(ReferenceCountedSceneGraphMessage message)
    {
```

6. Register your networked object with the network Scene
   This should be done at the start of the objects life i.e. in the Start() function.
   If you want to send messages as well, you also need to save the context object that is returned.

```
    NetworkContext ctx;

    // Start is called before the first frame update
    void Start()
    {
        ctx = NetworkScene.Register(this);
    }
```

7. Define how your message will look like.
   This is best done as a struct in the class. It being defined in the class prevents naming conflict.
   In the message, write the variables that you want to send. A good start is TransformMessage that is built to store the transform and is useful if you want your object's location and orientation to be synchronised.
   Do not forget the constructor! It allows to create the message in one line.

```
public class NewNetworkedObject : MonoBehaviour, INetworkObject, INetworkComponent
{
    public NetworkId Id { get; } = new NetworkId();

    // An example message carrying information about location, orientation, and colour
    public struct Message
    {
        public TransformMessage transform;

        public Color colour;

        public Message(Transform transform, Color colour)
        {
            this.transform = new TransformMessage(transform);
            this.colour = colour;
        }
    }
```

8. Receiving Messages
   Messages are received automatically. However, you will have to define how they are processed.
   For that, fill in ProcessMessage(…) The first step is usually “decoding” the message. Usually it will be sent as a JSON, but if you send it in a different format, you need to decode it differently as well.

```
    public void ProcessMessage(ReferenceCountedSceneGraphMessage message)
    {
        // "Decoding"
        var msg = message.FromJson<Message>();


        // Process content of the message
        // This is an example. Replace below with what you need
        transform.localPosition = msg.transform.position;
        transform.localRotation = msg.transform.rotation;

        ObjectMaterial.color = msg.colour;
    }
```

9. Sending Messages
   You can send messages at any time and anywhere in the code through using your context object. However, most of the time you will probably want that the objects move in sync, so it makes sense to send an update each frame. For this, put your sending in Update(…).

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

        ctx.SendJson(new Message(transform, ObjectMaterial.color));

    }
```

Additional Object and Networking Functionalities

Ownership

With this networked object, you will notice that you cannot move them in the editor or change their colour. The reason for this is that both objects are sending each other their position and any change you make gets overwritten by the incoming messages before it can go out. This situation where two changes race against each other is called a "race condition". One easy way to prevent this, is to give one of the network objects a flag that signals whether the object is owned by the local instance or controlled by a remote one. This is not only useful for interaction between the player and the object, but also for when the object is controlled by the physics system!

1. Add a bool that signals whether the object is locally owned and should send or whether it is owned by a remote instance and should only receive.
2. Add a if-clause that prevents the object from sending when not owned locally

After that, it should look like this:

```
7   public class NewNetworkedObject : MonoBehaviour, INetworkObject, INetworkComponent
8   {
9       public NetworkId Id { get; } = new NetworkId();
10
11      // An example message carrying information about location, orientation, and colour
12      public struct Message
13      {
14          public TransformMessage transform;
15
16          public Color colour;
17
18          public Message(Transform transform, Color colour)
19          {
20              this.transform = new TransformMessage(transform);
21              this.colour = colour;
22          }
23      }
24
25      NetworkContext ctx;
26
27      public bool owner = true;
28
29      private Material objectMaterial;
30      public Material ObjectMaterial { get { return objectMaterial != null ? objectMaterial : GetComponent<Renderer>().material; } }
31
32      // Start is called before the first frame update
33      void Start()
34      {
35          ctx = NetworkScene.Register(this);
36      }
37
38      // Update is called once per frame
39      void Update()
40      {
41          if (owner)
42          {
43              ctx.SendJson(new Message(transform, ObjectMaterial.color));
44          }
45      }
```

Now, however, it must be determined somehow if the object was spawned locally or remote to set the owner flag accordingly. For objects spawned by the NetworkSpawner (see section “Spawning Objects” below), this is easy and described here. For objects that are part of the scenery and already present at the start of the game, the objects will somehow need to negotiate an owner or find another way to prevent a race condition. We leave that to your ingenuity ;-)

1. Include Ubik.Samples and inherit ISpawnable

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Ubik.Messaging;
using Ubik.Samples;

public class NewNetworkedObject : MonoBehaviour, INetworkObject, INetworkComponent, ISpawnable
```

2. Implement the Interface. It only consist of one function: OnSpawned(bool local). The parameter "local" is either true if the object is owned locally or false if it is a remote object. You can directly assign that to "owner".

```
public void OnSpawned(bool local)
{
    owner = local;
}
```

Spawning Objects

To spawn an object in the local and all remote clients at the same time, there is a NetworkSpawner.

There are two ways to spawn objects:

1. NetworkSpawner.Spawn(…)
   Use this for objects that are not meant to be persistent. Objects spawned through this function can only be seen by players that are already connected. A new player that joins later will not see them.
2. NetworkSpawner.SpawnPersistent(…)
   Use this for objects that are meant to be persistent. Objects spawned through this function will be stored by the server and the client of any new player receives a copy of this list to make sure the new player sees all the persistent objects already in the world.

Note:
The new object prefab needs to be known to the environment! For that, add it to the PrefabCatalogue of the SceneManager. If the scene manager does not have a catalogue yet, you can create one in the project window by right-clicking->create->Prefab catalogue
You then have to drag it into the scene manager to use it.

Making an Object Graspable

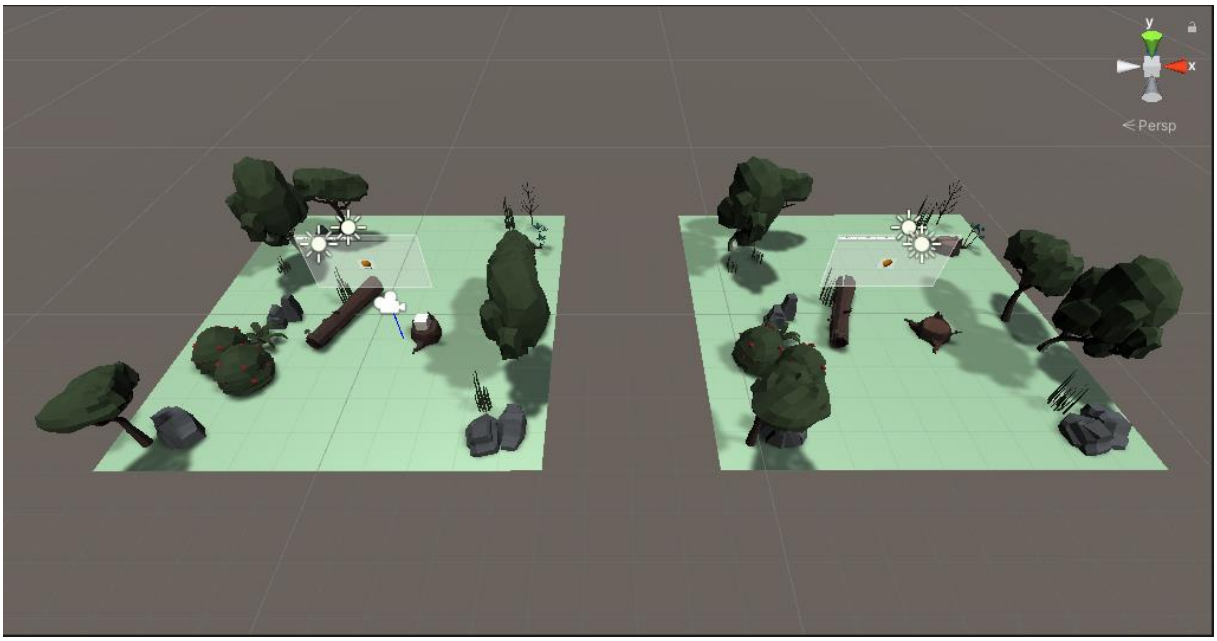
If you want to have your object being grabbable with the user's hands, you will have to inherit from IGraspable and implement Grasp(...) and Release(...)

Making an Object Usable

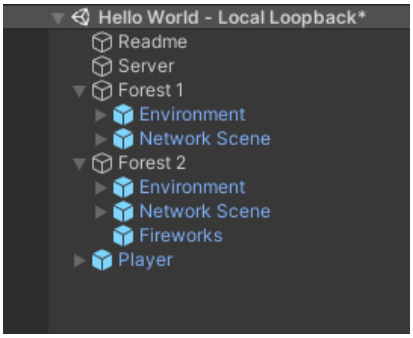
If you want your object to do something when the user presses the trigger button while holding it, you need to inherit from IUsable and implement Use(...) and UnUse(...)

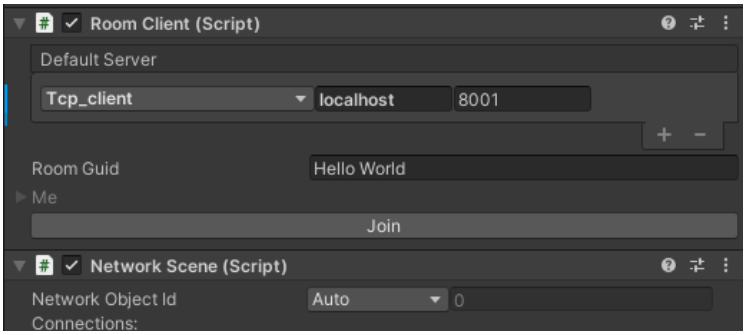
Testing Networked Objects

To enable you to test your networked objects without the help of a friend, the “Hello World – Local Loopback” scene is included in the Ubik samples. It has the scenery of the “Hello World” scene, but twice. Another addition is a local server that is automatically started when you play the scene and the scene is configured to connect both instances of the environment to it. This way, you create you have two connected clients in one active Unity scene and can test your networked objects without involving the UCL server or any other person.



To use it, you have to import the Ubik Samples and open the scene in the samples\introduction folder. To start the loopback, simply click the play button. The local server is now started and you will see an avatar in the left forest, but not the right, because they have not joined the same room yet. For that, open the nodes for both forests and find the nodes with the name “NetworkScene” within (see left image below). In the inspector window you will find a button with the title “join” (see right image below). If you click it for both nodes, you will notice that the forest on the right now also has an avatar in it that moves in sync with the left one. The two clients are now connected and you can test your networked objects.





Servers and Rooms

When your application starts, RoomClient automatically connects to the server provided by its Default Server property. However, even though you are connected to the server, you will not be able to see other users or networked objects. This is because you are not yet in the same “room”. One server can manage multiple rooms. You can join a room through the scripting interface or through the “Join” button shown on the inspector panel of the component in the Unity editor. You can also join a room by selecting its icon on the “Rooms” screen on the in-game UI. In the local loopback scene, make sure you click join on both RoomClients to test your networked objects.