

Projet **Domus VR**

Documentation

Dans le cadre projet objet connecté EI-SE 5
22/02/2021

Utilisation d'un casque VR Oculus Quest 2 programmé en Unity pour visualiser et commander depuis le casque une solution domotique.

Encadrant

PECHEUX François

Étudiants

JEBALI	Afif
PAILLIER	Théo
FRAYSSE	Julie
MUDRY	Mervis

Sommaire

[Description du projet](#)

[Le projet mère : Maison Intelligente](#)

[Une solution domotique privé](#)

[Choix des technologies](#)

[Schéma d'architecture](#)

[Notre projet : Domus VR](#)

[Interconnecter le Virtuel et le Réel](#)

[Raccordement avec le projet Maison Intelligente](#)

[Schéma d'architecture](#)

[Matériel & Installation](#)

[Installation](#)

[Post-installation](#)

[Unity : la base créer un environnement](#)

[Text_Pro_script](#)

[Script Mqtt](#)

[Dependencies](#)

[Main Functions](#)

[Configuration du broker](#)

[Utiliser les boutons du contrôleur](#)

[Récupérer un objet dans un script](#)

[Afficher ou cacher un objet](#)

[Suite](#)

[Sources](#)

Description du projet

Le projet mère : Maison Intelligente

Une solution domotique privé

Pour bien comprendre dans quelle démarche notre projet s'intègre, il faut parler d'un autre projet encadré également par M. Pêcheux nommé *Maison Intelligente*. Notre projet est le prolongement de celui-ci dont l'idée principale est de pouvoir commander des capteurs/actionneurs d'une maison sans qu'aucune donnée ne sorte de celle-ci. On trouve sur le marché de l'objet connecté grand public des solutions pour connecter et commander des appareils (exemple : passerelles vendues par *Ikea*, *Phillips* ou *Xiaomi*). Le problème est que celles-ci font fuir les données collectées par les appareils présents dans la maison vers leurs entreprises. Cela pose des questions sur la protection des données et leur utilisation à des fins commerciales. L'idée est donc de créer une solution alternative donnant accès aux mêmes fonctionnalités mais avec des données qui restent dans le périmètre de la maison. La fonctionnalité principale étant d'avoir une interface utilisateur personnalisable sur laquelle on peut lire les données capteurs et modifier l'état des actionneurs.

Choix des technologies

- Interface utilisateur

La plateforme *Jeedom* a été choisie pour implémenter cette interface car elle s'exécute sur un serveur local et open source. Ces caractéristiques permettent de garder les données privées (pas de transferts avec des serveurs extérieurs) et d'avoir une transparence complète sur leur traitement. Cette plateforme est également compatible avec de nombreux protocoles utilisés par les acteurs de l'IoT.



Figure 1 - Exemple interface *Jeedom*

- Protocole de communication

La solution envisagée utilise des capteurs/actionneurs communiquant avec le protocole *Zigbee*. Ce protocole est privilégié pour ses qualités d'interopérabilité et de facilité d'utilisation. Il permet la communication entre les appareils de la maison (capteurs et actionneurs) et l'interface utilisateur.

Schéma d'architecture

Une *Raspberry Pi 4* fera donc tourner un serveur *Jeedom* local et un dongle *Zigate* sera branché sur celle-ci pour récupérer les valeurs communiquées par les capteurs avec le protocole *Zigbee*.

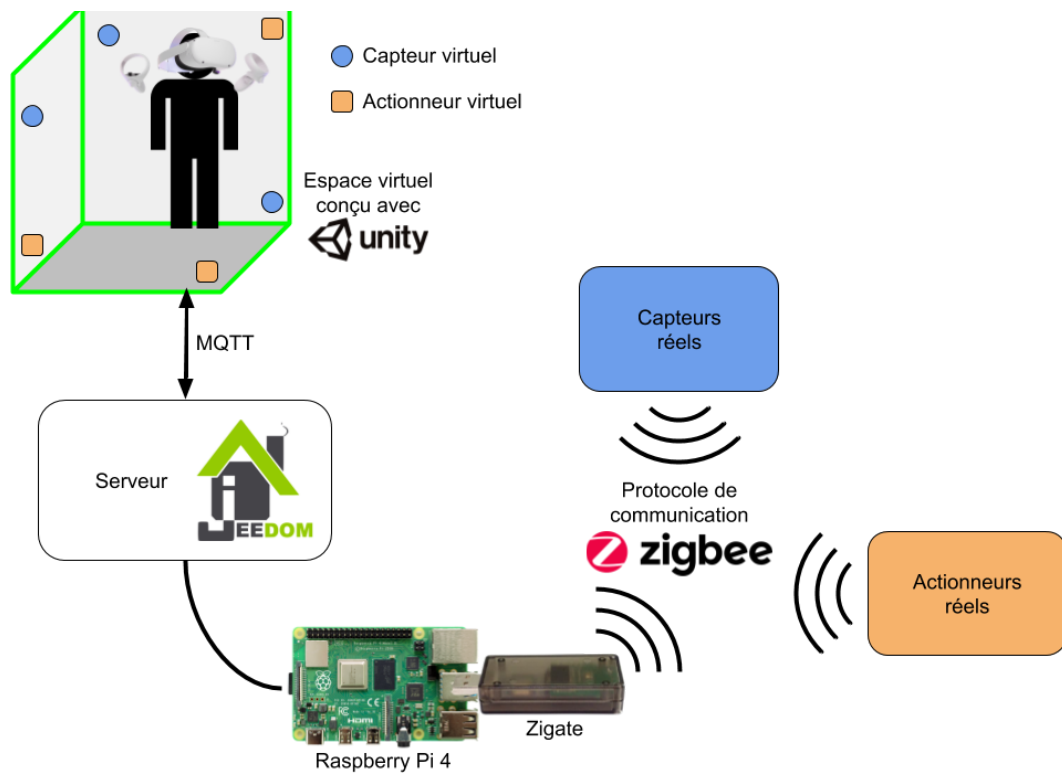


Figure 2 - Schéma architectural du projet *Maison Intelligente*

Notre projet : Domus VR

Interconnecter le Virtuel et le Réel

Ce projet est né de la curiosité quant à l'interconnexion entre réalité et virtuel. Le but à long terme serait de faire de la réalité augmentée avec des lunettes HoloLens. Notre projet *Domus VR* constitue une étape intermédiaire dans cette entreprise. Le but étant de faire un premier pas vers une solution qui interconnecte réalité et virtuel.

Globalement cela permet d'améliorer l'expérience utilisateur. Notre projet s'inscrit dans l'exploration des manières d'interconnecter réel et virtuel.

Raccordement avec le projet *Maison Intelligente*

Le prolongement de ce projet consiste alors à fournir une nouvelle interface permettant de visualiser et de commander les actionneurs/capteurs, initialement sur *Jeedom* au profit d'une représentation virtuelle. Ce monde virtuel serait construit en utilisant *Unity*. Ces deux projets se recouvrent donc pendant le processus de récupération des données. Au lieu d'être récupéré par un serveur *Jeedom local*, le but est de récupérer/poster les données depuis le casque sur un broker MQTT extrait de ce *Jeedom local*.

MQTT les

Capteurs & Topics MQTT

Liste des capteurs : xxx

- Les topics capteurs

`/capteurs` `/temperature`
 `/lumiere`
 `/prise`
 `/vibration`
 `/mouvement`

- Les topics actionneurs :

`/actionneurs` `/temperature`
 `/lumiere`
 `/prise`

Schéma d'architecture

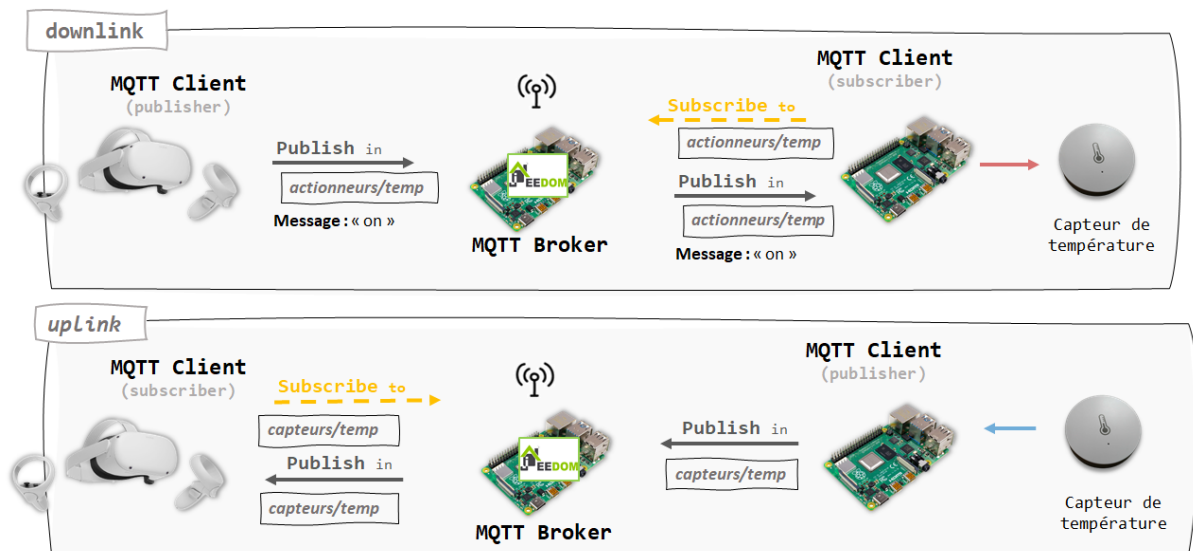


Figure 3 - Schéma de la circulation des données capteurs/actionneurs dans le projet *Domus VR*

Matériel & Installation

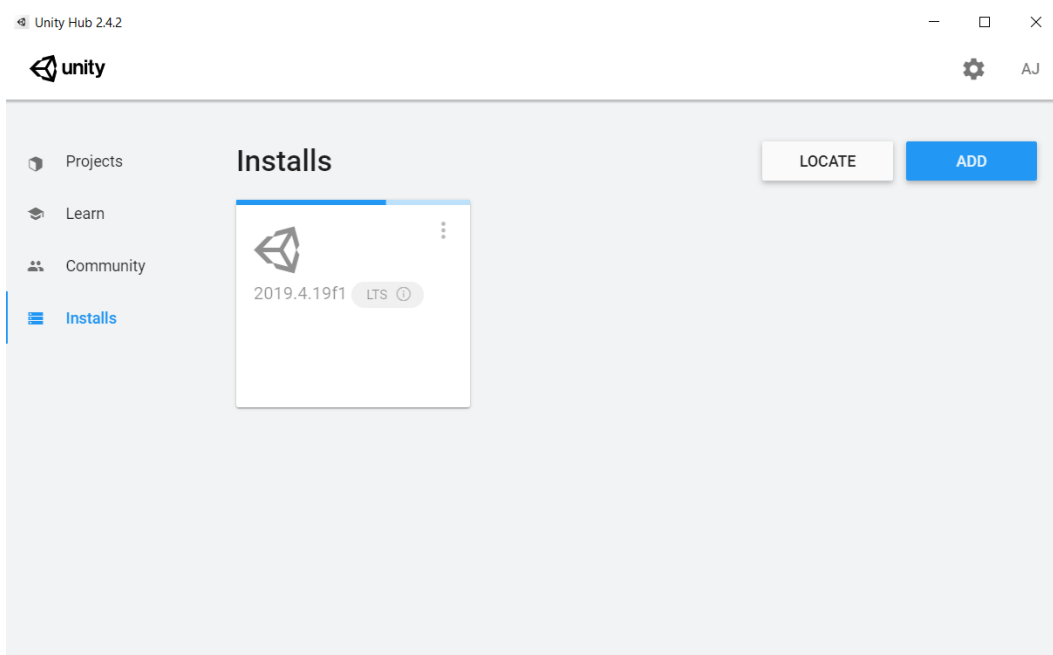
Pour la suite du projet nous allons avoir besoin du matériel suivant :

- Un casque **Oculus Quest 2**, ses deux manettes Oculus Touch et son câble USB-C.
- Le logiciel **Oculus Link**. Il permet d'appairer le casque Oculus Quest 2 à votre machine. Cela permettra par la suite de téléverser notre application sur le casque.
Il est téléchargeable depuis le site Oculus officiel :
<https://www.oculus.com/setup/>
- Le logiciel **Unity**. C'est le logiciel de développement.

Installation

Download et Installer Unity Hub : <https://unity3d.com/fr/get-unity/download>

Après avoir installer Unity Hub, il faut installer une version de unity :



Appuyer sur add et prendre la version recommandée :

Attention seulement la version 2019.3-4 supporte le plugin XR management

Add Unity Version
✕

1 Select a version of Unity
2 Add modules to your install

Can't find the version you're looking for? Visit our [download archive](#) for access to [long-term support](#) and [patch releases](#), or join our [Open Beta program](#) releases.

Recommended Release

☐ Unity 2019.4.19f1 (LTS)

Official Releases

☐ Unity 2020.2.3f1

☐ Unity 2020.1.17f1

☒ Unity 2018.4.31f1 (LTS)

Pre-Releases

CANCEL
BACK
NEXT

Puis sélectionner Android Build support, pour l'oculus quest 2:

Add Unity Version
✕

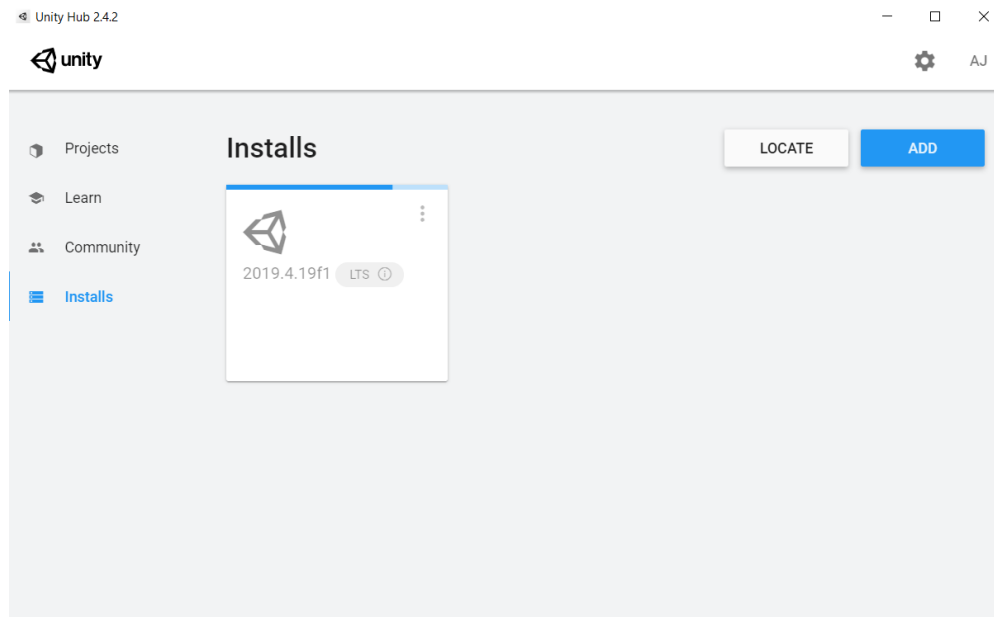
☒ Select a version of Unity
2 Add modules to your install

Add modules to Unity 2020.2.3f1 : total space available 20.3 GB - total space required 14.8 GB

Dev tools	Download Size	Install Size
<input type="checkbox"/> Microsoft Visual Studio Community 2019	1.4 GB	1.3 GB
Platforms		
<input checked="" type="checkbox"/> <input checked="" type="checkbox"/> Android Build Support	252.1 MB	1.1 GB
<input checked="" type="checkbox"/> Android SDK & NDK Tools	1.0 GB	3.0 GB
<input checked="" type="checkbox"/> OpenJDK	153.0 MB	70.5 MB
<input type="checkbox"/> iOS Build Support	368.2 MB	1.6 GB

CANCEL
BACK
NEXT

Et maintenant votre version s'installe :



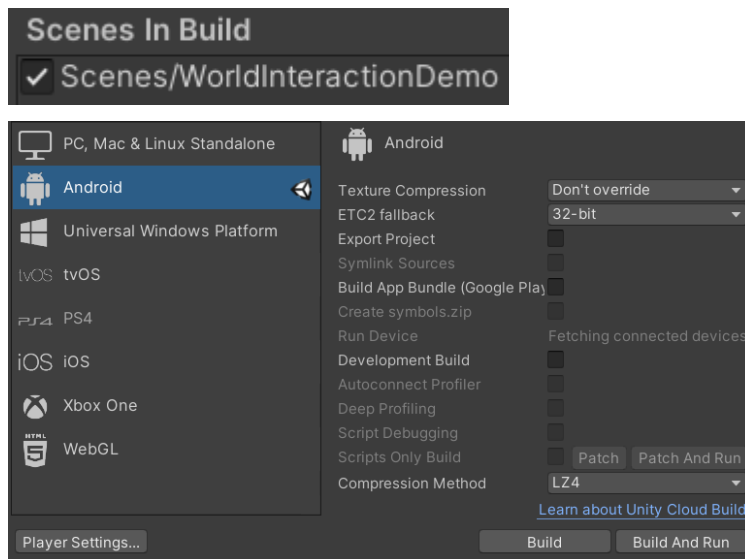
Post-installation

Setup pour le VR :

- 1) Windows -> Package Manager : Install XR Interaction Toolkit
- 2) Edit -> Project Setting -> Player:
 - Dans XR setting : cliquer dans “deprecated Setting”, sur “Virtual reality Support”
 - puis ajouter le sdk pour le openVR et Oculus, en déroulant le menu au niveau du “+”
 - Faire pareil pour l’onglet android !

Exporter notre application sur l'Oculus Quest

- Aller dans File -> Build Setting
- Puis choisir la scène développée
- Sélectionner le module Android
- Faire "Build and run" si le casque est branché. Sinon faire "Build".



- On obtient un fichier .apk qui comprend tous les fichiers nécessaires à l'installation de l'application sur l'oculus.



XR_affichage.apk

- Enfin nous pouvons observer notre application se lancer sur l'oculus.

Unity : la base créer un environnement

Raccourci :

X : Modifier la taille d'un objet

W : Modifier la position dans l'espace d'un objet

- Créer une scène :
 - Dans la fenêtre "Project", onglet "Scène", clique droit où il y a les scènes et sélectionner "Create".
- Créer un "plan" :
 - Dans notre scène, clique droit, dans le menu déroulant, 3D object, selectionner plan

Il est possible d'ajouter un script à un objet pour le rendre plus dynamique et lui donner une fonction/tâches à effectuer :

Script en C# :

```
Classe: {  
    void Start  
    {  
        Initialisation avant le début de l'expérience virtuelle  
    }  
    Void Update  
    {  
        Tâches à effectuer à chaque update de l'image.  
    }  
}
```

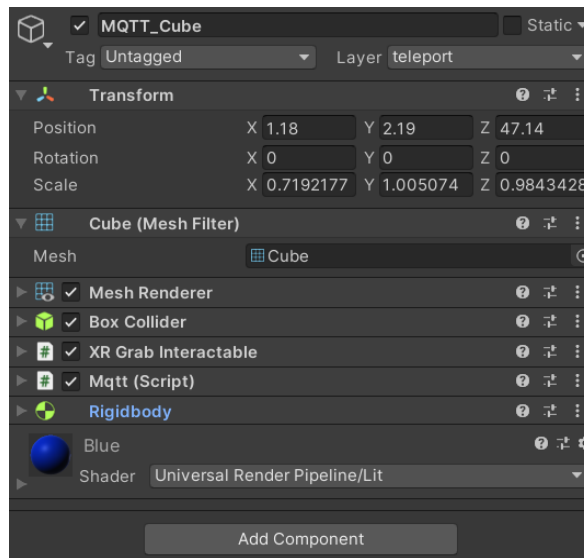
Exemple : Création d'un objet capteur/actionneur

Dans notre cas, il nous fallait créer différent module représentant les capteur/actionneurs du réels

Pour un capteur de température (affichage de la température reçu par mqtt) :



Il est nécessaire d'ajouter différent composant permettant de "customiser" notre GameObject :

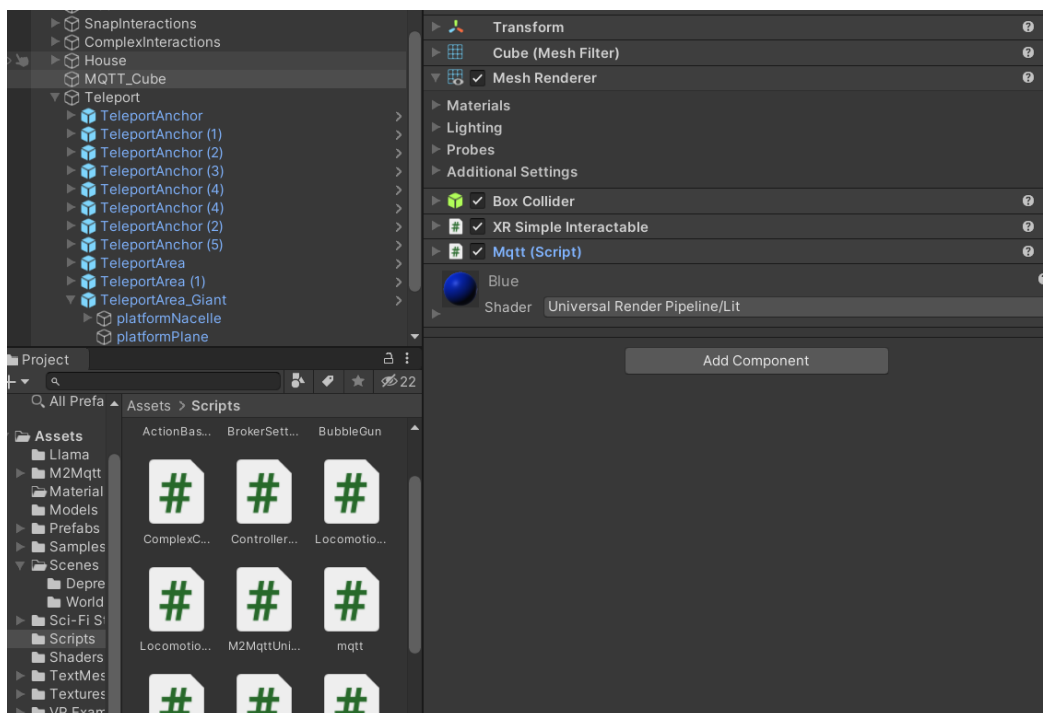


Etapes :

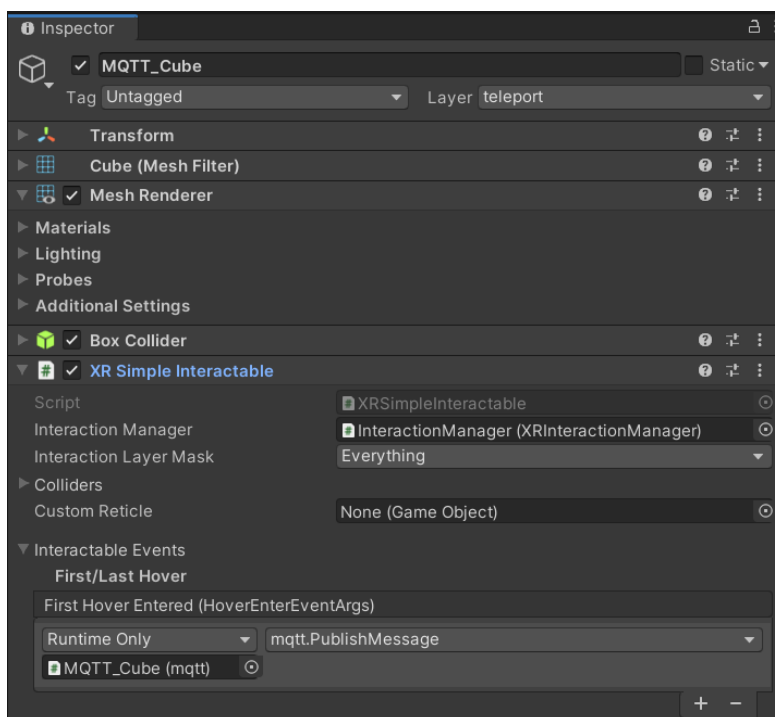
- Box Collider :
 - Permet de repérer la collision en donnant un "corps" à l'objet
- Solid Body :
 - Permet d'appliquer des lois physiques sur l'objet
- XR Interactable :
 - *Permet les interactions avec l'utilisateur*
- Script :
 - *Text_Pro_Script* : Script pour manipuler un text 3D
 - *Mqtt (publish)* : Script pour publier une donnée

En ajoutant ces composants, nous pouvons interagir avec le capteur virtuel et afficher la température dans la maison.

Pour lier le script à l'objet, il suffit de sélectionner l'objet, et d'y glisser le script :

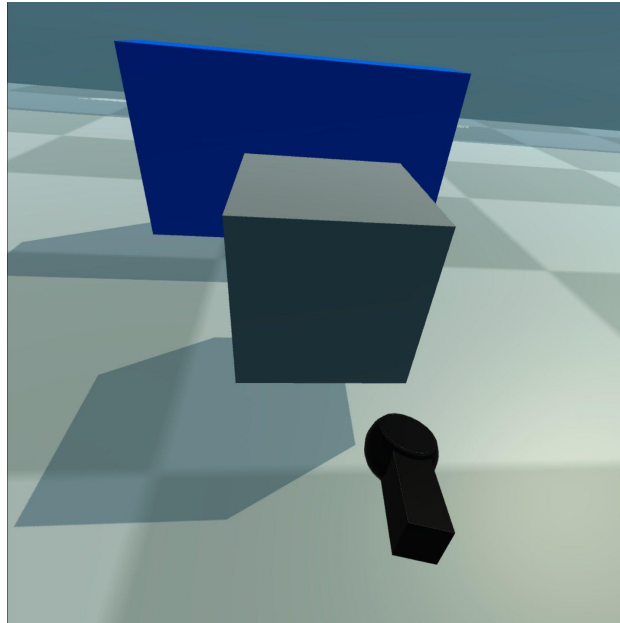


Pour attacher une fonction spécifique à une action, il faut ensuite glisser ce script dans une des actions du composant **XR Simple Interactable**. Il faut qu'elle soit **publique**.



Ici, en survolant le cube **MQTT_Cube** avec le pointeur de notre contrôleur, nous allons déclencher la fonction **PublishMessage()**.

Le cube se présentant de cette manière.



Text_Pro_script

Pour manipuler un texte et le mettre à jour au sein d'un script, nous utilisons des TextMeshProUGUI (GameObject > UI > Text - TextMeshPro).

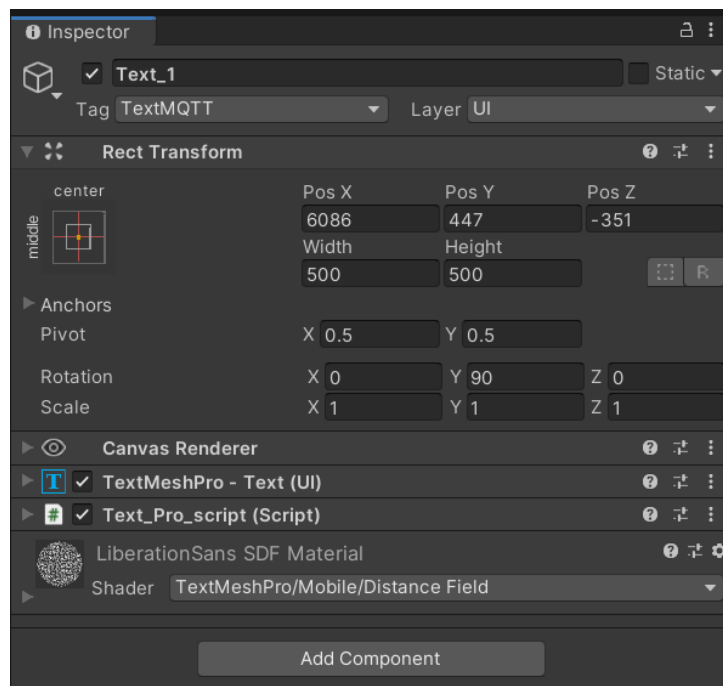
Nous avons créé une classe permettant de facilement mettre à jour le texte. Nous allons donc régulièrement utiliser la méthode **TextUpdate(string s)**.

```
public class Text_Pro_script : MonoBehaviour
{
    public TextMeshProUGUI textMesh;

    Message Unity | 0 références
    public void Start()
    {
        textMesh = GetComponent<TextMeshProUGUI>();
        textMesh.text = "In Text Pro Script's Start";
    }

    2 références
    public void TextUpdate(string s)
    {
        textMesh.text = s;
    }
}
```

Pour utiliser ces textes dans un script, il est nécessaire de leur associer le script précédent.



Nous récupérons ensuite l'objet via son nom, et nous pouvons appeler la fonction précédemment définie.

```
Text_Pro_script Text_1;
Text_Pro_script Text_2;

Text_1 = GameObject.Find("Text_1").GetComponent<Text_Pro_script>();
Text_2 = GameObject.Find("Text_2").GetComponent<Text_Pro_script>();

//Text.textMesh = "";
Text_1.TextUpdate("In MQTT's Start");
Text_2.TextUpdate("In MQTT's Start");
```

Script Mqtt

Le code peut être décomposé en deux parties :

- un script "**bibliothèque**" M2MqttUnityClient.cs

Cette bibliothèque fournit une classe contenant des fonctions pour faire du Mqtt.

Elle permet de gérer la connexion au broker et surtout les files d'attente de message.

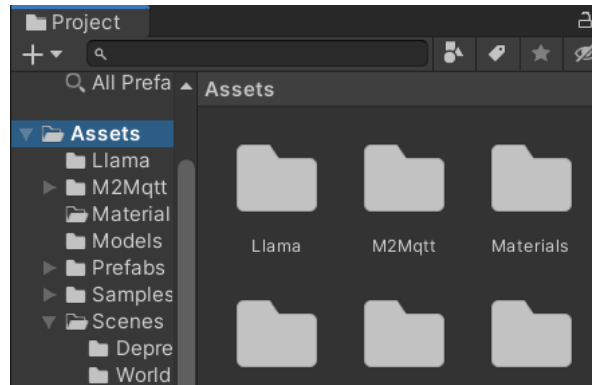
- un **script** mqtt.cs à lier à l'objet virtuel

La classe *mqtt* correspondant à ce script hérite de la classe M2MqttUnityClient et donc de toutes ses fonctions.

Le script est composé de 2 fonctions principales :

- Une fonction *start()* appelée avant que la première image soit actualisée.
Dans cette fonction on se connecte au broker et on souscrit aux topics précisés dans la fonction *SubscribeTopics()*
- Une fonction *update()* qui est appelée à chaque actualisation de l'image.
Avec les fonctions de la bibliothèque, on récupère les messages dans une file d'attente de message. On les stocke dans un buffer de message déclaré dans *mqtt.cs*. Les messages dans ce buffer sont régulièrement parcourus, traités puis effacés.

Pour faire fonctionner ce script il faut que le dossier **M2Mqtt** soit présent dans **Assets**.



Dependencies

// bibliothèque utilisé par la classe *M2MqttUnityClient*

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using uPLibrary.Networking.M2Mqtt;
using uPLibrary.Networking.M2Mqtt.Messages;
```

Main Functions

```
public class mqtt : M2MqttUnityClient // hérite de la classe M2MqttUnityClient
{
    private List<string> eventMessages = new List<string>();

    // Start is called before the first frame update
    void Start()
    {
        Debug.Log("mqtt Start()");
        // Connection au broker et souscription aux topics
        base.Start(); // → Connect() → DoConnect() → OnConnected() → SubscribeTopics()

        Debug.Log("base.Start() done");
        base.OnConnecting(); // simple print de debug
    }
}
```

```

    }

    // Update is called once per frame
    protected override void Update()
    {
        /* récupération des messages mqtt et stockage dans eventMessages */
        base.Update(); // call ProcessMqttEvents()

        if (eventMessages.Count > 0) // si des msg sont ajoutés dans le buffer
        {
            // On traite chaque message à la suite un par un
            foreach (string msg in eventMessages)
            {
                ProcessMessage(msg);
            }
            // On supprime les messages du buffer de message
            eventMessages.Clear();
        }
    }
}

```

```

protected override void SubscribeTopics()
{
    /* Précise le topic auquel on souhaite souscrire */
    client.Subscribe(new string[] { "test1" },
        new byte[] { MqttMsgBase.QOS_LEVEL_EXACTLY_ONCE });
}

```

Dans la fonction **ProcessMessage(string msg)**, nous faisons bouger un objet identifié par son nom "MQTT_Cube" (une forme de débbugage).

```

private void ProcessMessage(string msg)
{
    Debug.Log("FPX Received: " + msg);
    GameObject go = GameObject.Find("MQTT_Cube");
    Debug.Log("go=" + go);
    Vector3 v = new Vector3(1.0f, 0.0f, 0.0f);
    go.transform.Translate(v, Space.World);
}

```

```

protected override void Publish()
{
    /* Précise le topic et le message à publier (topic = oculus, msg = "Hello from oculus!")

```

```
String txtPublish = "Hello from oculus!\n";
client.Publish(new string[] { "oculus" }, Encoding.UTF8.GetBytes(txtPublish.Text),
MqttMsgBase.QOS_LEVEL_EXACTLY_ONCE, true);
}

protected virtual void Update()
{
    ProcessMqttEvents();
}

protected virtual void ProcessMqttEvents()
{
    // process messages in the main queue
    SwapMqttMessageQueues();
    ProcessMqttMessageBackgroundQueue();
    // process messages income in the meanwhile
    SwapMqttMessageQueues();
    ProcessMqttMessageBackgroundQueue();

    if (mqttClientConnectionClosed)
    {
        mqttClientConnectionClosed = false;
        OnConnectionLost();
    }
}

// Swap the message queues to continue receiving messages when processing a queue.
private void SwapMqttMessageQueues()
{
    frontMessageQueue = frontMessageQueue == messageQueue1 ? messageQueue2
    : messageQueue1;
    backMessageQueue = backMessageQueue == messageQueue1 ? messageQueue2
    : messageQueue1;
}

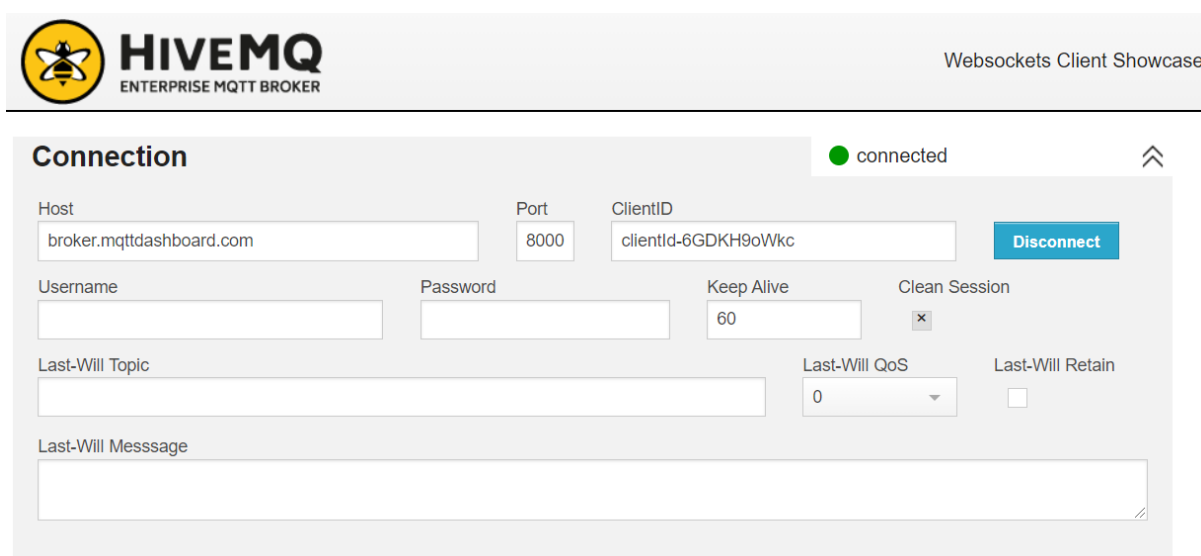
private void ProcessMqttMessageBackgroundQueue()
{
    foreach (MqttMsgPublishEventArgs msg in backMessageQueue)
    {
        DecodeMessage(msg.Topic, msg.Message);
    }
    backMessageQueue.Clear();
}
```

Configuration du broker

Il faut choisir le broker à utiliser pour communiquer avec Mqtt. Dans un premier temps, il est préférable de choisir un broker public comme [HiveMQ](#).

Il sera alors possible de directement envoyer un message sur un topic ou lire un message sur un topic.

Sur le client, voici la configuration (configuration par défaut) :



HIVEMQ ENTERPRISE MQTT BROKER Websockets Client Showcase

Connection ● connected

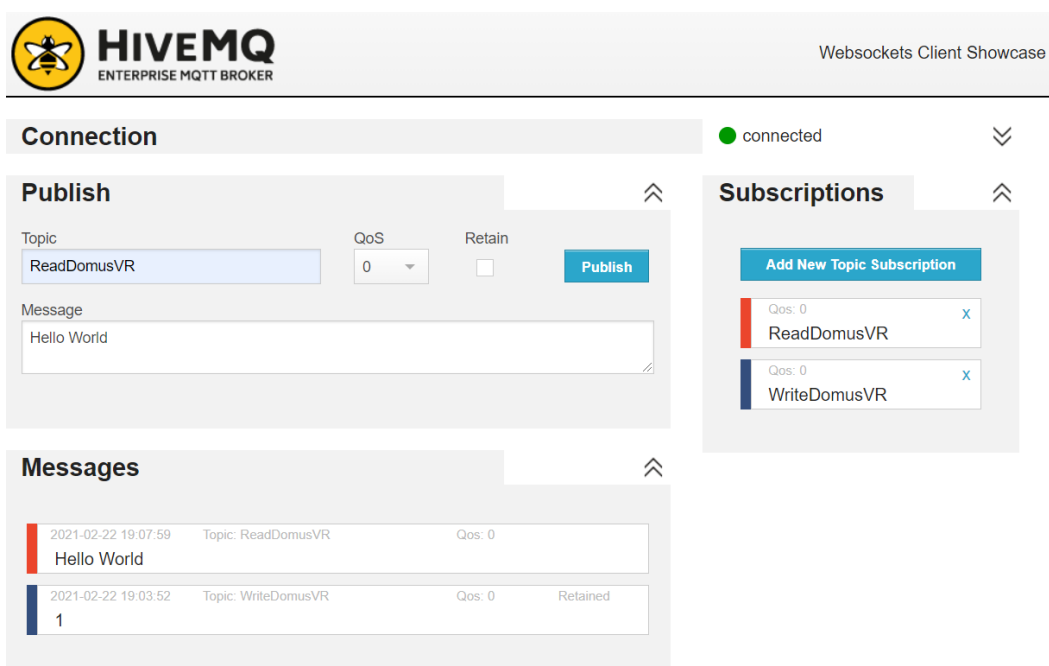
Host: Port: ClientID: **Disconnect**

Username: Password: Keep Alive: Clean Session: ☒

Last-Will Topic: Last-Will QoS: Last-Will Retain: ☐

Last-Will Message:

On peut ensuite subscribe à un nouveau topic et envoyer des messages :



HIVEMQ ENTERPRISE MQTT BROKER Websockets Client Showcase

Connection ● connected

Publish

Topic: QoS: Retain: ☐ **Publish**

Message:

Subscriptions

Add New Topic Subscription

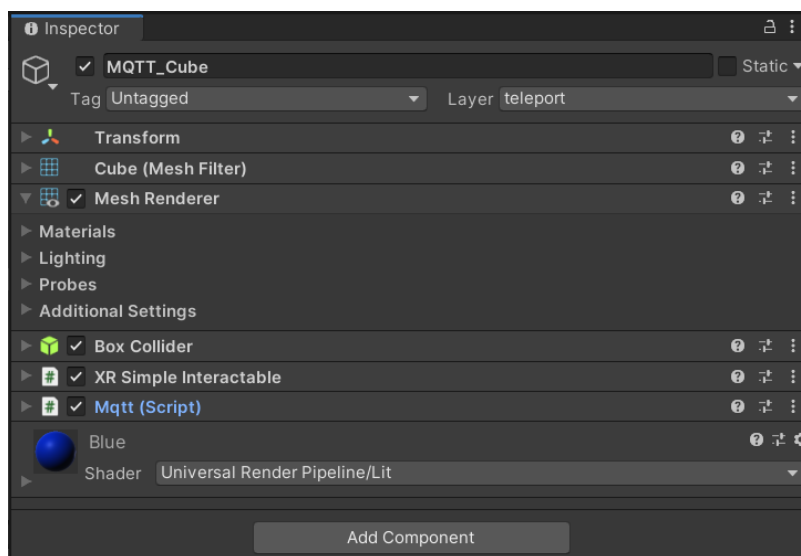
- ☒ QoS: 0 x
- ☒ QoS: 0 x

Messages

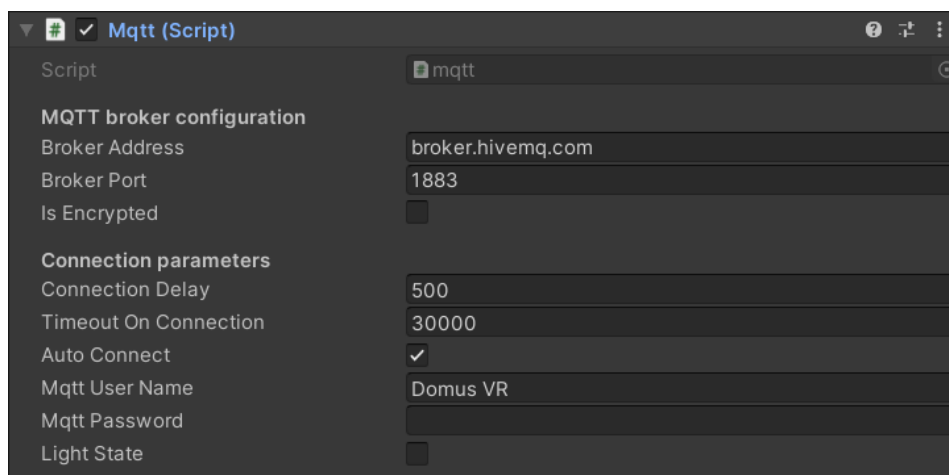
2021-02-22 19:07:59	Topic: ReadDomusVR	Qos: 0	Hello World
2021-02-22 19:03:52	Topic: WriteDomusVR	Qos: 0	Retained

Dans Unity, il faut également spécifier le broker utilisé.

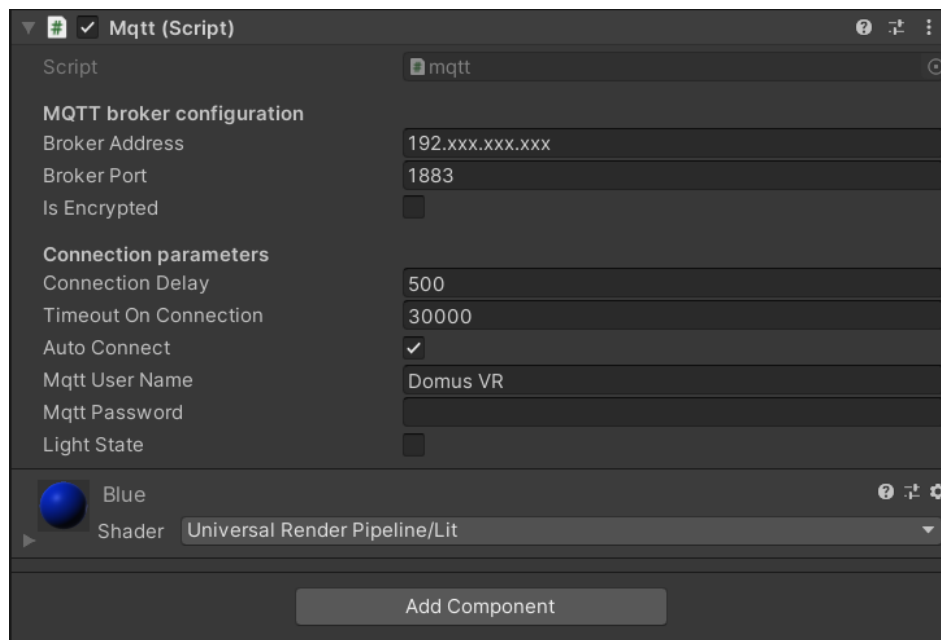
On sélectionne l'objet rattaché au script MQTT.



Puis on entre l'adresse du broker.



Si l'on passe à un serveur privé (sur une raspberry par exemple), c'est ici, dans le champ **Broker Address** que l'on rentrera l'adresse IP en question.



Attention à ce que le casque et le serveur soient sur le même réseau.

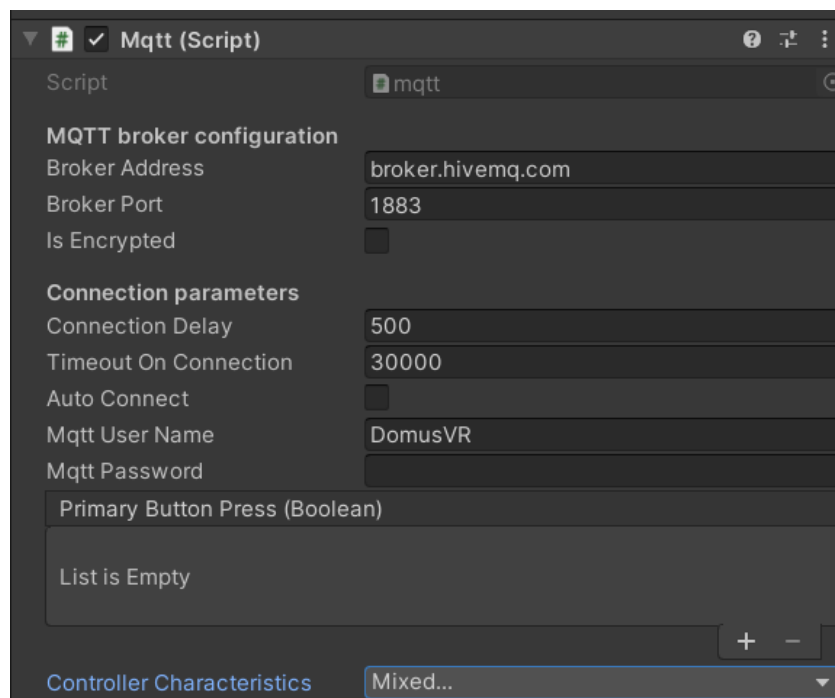
Utiliser les boutons du contrôleur

Il est possible d'utiliser les boutons du contrôleur pour effectuer différentes actions. Pour cela, il faut importer la librairie **UnityEngine.XR.Interaction.Toolkit**.

Il faut également déclarer la classe correspondante :

```
[System.Serializable]
public class PrimaryButtonEvent : UnityEvent<bool> {}
```

Il faut ensuite spécifier le contrôleur à utiliser dans Unity.



Pour utiliser le contrôleur droit par exemple, on sélectionne “Controller” et “Right” dans le menu déroulant **Controller Characteristics**.

Nous déclarons ensuite plusieurs attributs.

```
List<InputDevice> devices = new List<InputDevice>();
InputDevices.GetDevicesWithCharacteristics(controllerCharacteristics, devices);

RightHand = devices[0];

RightHand.TryGetFeatureValue(CommonUsages.primaryButton, out bool PrimaryButtonValue);
RightHand.TryGetFeatureValue(CommonUsages.secondaryButton, out bool SecondaryButtonValue);
```

devices correspond à la liste des devices récupérés. Nous précisons ensuite que nous choisissons les **devices** en fonction des caractéristiques définies plus haut.

Ayant seulement choisi le contrôleur droit, il est à la position 0 dans la liste.

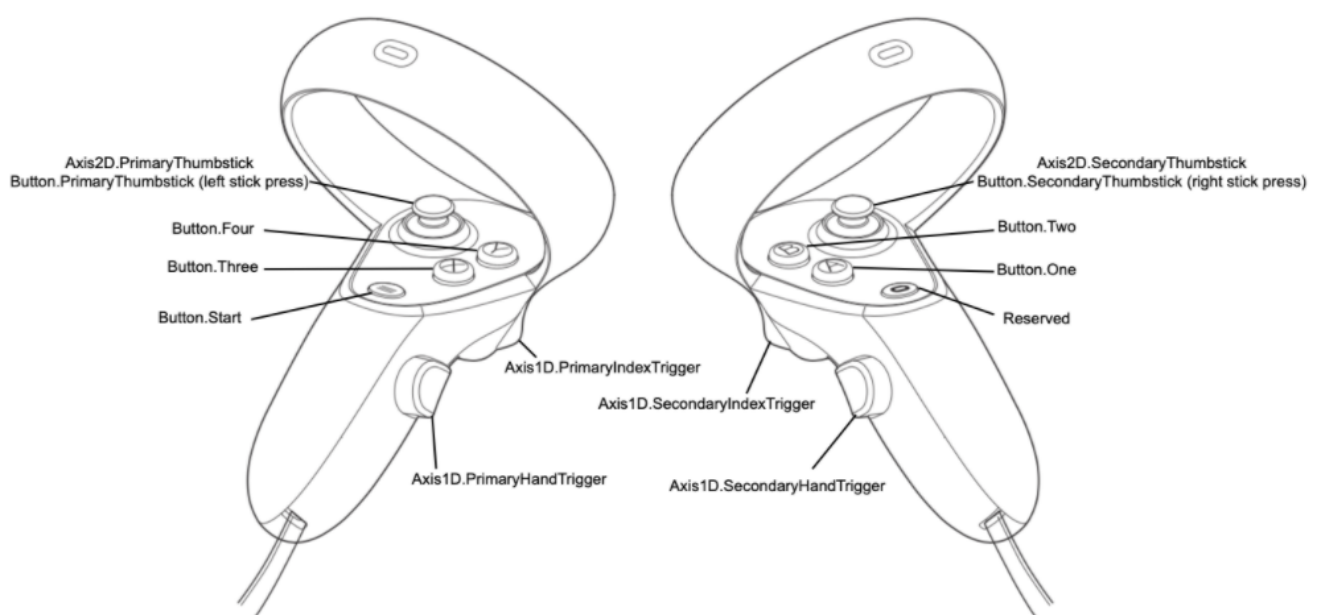
Nous récupérerons ensuite le booléen de chaque bouton que nous affectons à deux variables.

Nous pouvons par exemple afficher l'état des boutons dans un texte :

```
Text_1.TextUpdate(PrimaryButtonValue.ToString());
Text_2.TextUpdate(SecondaryButtonValue.ToString());
```



Les boutons étant définis de cette manière :



Plus de détails dans la documentation d'Oculus :

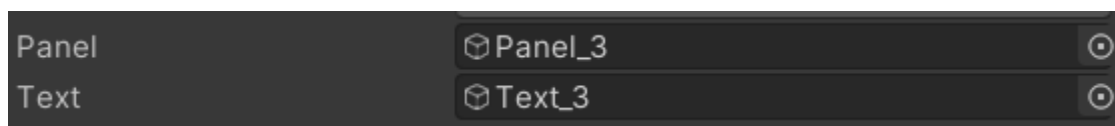
<https://developer.oculus.com/documentation/unity/unity-ovrinput/>

Récupérer un objet dans un script

Nous pouvons récupérer un objet via son nom, mais une manière plus “sûre” et permettant plus de manipulations est de le faire en créant un attribut dans la classe concernée.

```
public GameObject Panel;  
public GameObject Text;
```

On peut ensuite directement glisser l'objet concerné dans le script en question.



Afficher ou cacher un objet

On peut ensuite par exemple afficher ou cacher un objet. **Ne fonctionne à priori qu'en déclarant un objet comme réalisé ci dessus.**

```
Panel.SetActive(!Panel.activeSelf);  
Text.SetActive(!Text.activeSelf);
```

Ici, à chaque fois que la fonction est appelée, nous inversons l'état des objets. Nous les rendons donc visibles ou invisibles.

Sources

Chaîne Youtube de “Valem” :

<https://www.youtube.com/channel/UCPJlesN59MzHPPCp0Lg8sLw>

Chaîne Youtube de “VR with Andrew” :

<https://www.youtube.com/channel/UCG8bDPqp3jykCGbx-CiL7VQ>

Documentation Oculus :

https://docs.unity3d.com/Manual/xr_input.html