TP3 Unity

Pour réaliser ce TP, repartez du projet créé lors du TP2.

1. Réseau

UNet, l'API réseau de Unity, est considérée « deprecated » et sera remplacée dans une future version de Unity. Néanmoins, la prochaine API est encore en cours de développement.

*Il est donc conseillé aujourd'hui d'utiliser **Mirror**, une API open source toujours maintenue, et fonctionnant de la même façon que UNet :*

https://mirror-networking.com/

Ainsi, la majeure partie de la documentation de UNet (https://docs.unity3d.com/Manual/UNet.html) reste applicable à Mirror. Pour les différences, voir le guide de migration de UNet vers Mirror (https://mirror-networking.com/docs/General/Migration.html).

- Installez **Mirror** depuis l'Asset Store.
- Ouvrez la scène principale et sélectionnez l'objet représentant le joueur. Ajoutez-lui un composant « **NetworkIdentity** » (cochez la case « Local Player Authority ») et un composant « **NetworkTransform** », puis faites-en un prefab.
- Ouvrez la scène « Menu ».
- Créez un objet vide et ajoutez-lui un composant « **NetworkManager** ». Vous pouvez renommer cet objet NetworkManager. Assurez-vous que l'option « Don't Destroy on Load » soit cochée.
- Paramétrez les scène « Online » (scène principale du jeu) et « Offline » (scène Menu).
- Dans « Spawn Info », sélectionnez comme « player prefab » le prefab créé précédemment.
- Le bouton que vous avez créé au TP précédent servira à créer une nouvelle partie sur le réseau : au lieu de changer de scène, il devra faire appel à la méthode **StartHost()** du composant NetworkManager.
- Rajoutez un bouton « Rejoindre » dans le menu principal. Placez-y à côté un champ de texte permettant de saisir une adresse IP.
- Le bouton « Rejoindre » permettra donc de rejoindre une partie déjà créée : il faut alors fournir au NetworkManager l'adresse IP renseignée dans le champ texte (si le champ est vide, choisir l'adresse par défaut 127.0.0.1) et appeler la méthode **StartClient()**.
- Testez votre application : que se passe-t-il ?

Points d'apparition

- Ouvrez la scène principale. Avant de la modifier, sauvez-la sous un autre nom (**dans ce cas, pensez ensuite à mettre à jour le lien dans le NetworkManager** !).
- Ajoutez des objets vides aux endroits où le joueur peut apparaître. Ces objets doivent porter le composant « **NetworkStartPosition** ». Vous pouvez supprimer le joueur de cette scène.
- Testez à nouveau votre application.

- Testez maintenant votre application avec deux instances : une instance crée une nouvelle partie et la deuxième la rejoint. Testez d'abord en local. Que se passe-t-il ?
- Si ce n'est pas déjà fait, ajoutez un objet visible au prefab permettant de représenter le joueur (même une forme simple, comme une capsule par exemple).

NetworkBehaviour

- Créez un script appelé NetworkPlayer, dérivant de **NetworkBehaviour**, et placez-le sur le prefab du joueur.
- Créez une méthode surchargeant la méthode **OnStartLocalPlayer**. Dans cette méthode, activez les composants suivant seulement pour le joueur local (grâce à l'attribut **isLocalPlayer**) :
    - Script(s) permettant de contrôler les mouvements du joueur (FirstPersonControler ou autre)
    - Script permettant de tirer avec le pistolet
    - Camera
    - AudioListener
- Testez à nouveau : qu'est-ce qui a changé ?

La classe NetworkBehaviour peut remplacer un MonoBehaviour, pour tous les comportements liés au réseau. Un script dérivant de cette classe peut être appliqué uniquement à un GameObject portant un composant NetworkIdentity.

Instantiation d'objets

- Lorsque vous tirez avec le pistolet, l'autre joueur voit-il les balles ?
- Pour corriger cela, vous devez enregistrer le prefab des balles auprès du NetworkManager (et donc enregistrer la balle en tant que prefab si ce n'est pas déjà fait) : ajoutez le prefab aux « Registered Spawnable Prefabs » du NetworkManager.
- Dans le script de création de la balle, faites appel à la méthode **Spawn** après avoir instantié le prefab :

```
GameObject go = Instantiate (…) ;
NetworkServer.Spawn(go, connectionToClient);
```

- La méthode Spawn doit obligatoirement être appelée par le serveur. Pour s'assurer de cela, effectuez les opérations de création de la balle dans une **commande**.

*Une **commande** est une méthode appelée par un client et exécutée sur le serveur. Pour créer une commande, créez une méthode avec l'attribut « Command ». Par convention, on ajoute généralement le préfixe « Cmd » au nom des commandes : cela permet de se rappeler qu'il s'agit d'une commande lorsqu'on appelle cette méthode.*

```
[Command]
void CmdExample()
{
}
```

Choisissez un type d'objet du décor, par exemple les objets « Decorative plant ». Ouvrez le prefab et ajoutez-lui, s'il ne les a pas déjà, un Rigidbody, un NetworkIdentity et un NetworkTransform.

- Testez l'application : lorsque vous tirez sur cet objet, les mouvements sont-ils corrects ? La position est-elle toujours identique entre le serveur et le client ?
- Modifiez le script du pistolet pour que l'application de la force lors de l'impact se fasse dans une commande (méthode avec l'attribut **[Command]**).

Points de vie

- Affectez un nombre de points de vie à votre joueur.
- Affichez un texte à l'écran permettant de voir ses points de vie (vous pouvez pour cela attacher un Canvas comme enfant du prefab de votre joueur).
- Lorsqu'un joueur tire, si la balle touche un autre joueur, décrémentez ses points de vie. Pour cela, le joueur doit implémenter un appel **ClientRpc**.

*Un appel **ClientRpc** fonctionne à l'inverse d'une commande : il s'agit d'une méthode appelée par le serveur et exécutée sur un client. Pour créer un appel ClientRpc, créez une méthode avec l'attribut « ClientRpc ». Par convention, on ajoute généralement le préfixe « Rpc » à leur nom.*

```
[ClientRpc]
void RpcExample()
{
}
```

Pour vérifier que l'objet touché par la balle est un autre joueur, vous pouvez utiliser le mécanisme de tag :

- Ouvrez le prefab du joueur et, dans l'inspecteur, affectez-lui le tag « Player ».
- Lors de la collision entre la balle et un autre objet, faites la vérification suivante :

```
if (gameObject.tag == "Player")
   // …
```

Pour aller plus loin

- Si vous déplacez des objets ayant un Rigidbody (par exemple en tirant dessus), l'autre joueur les voit-il bouger ? Comment corriger cela ?
- Lorsqu'un autre joueur déclenche l'ouverture de la porte, la voit-on également s'ouvrir ? Corrigez cela en utilisant un composant **NetworkAnimator**.
- Ajoutez une couleur aléatoire à chaque joueur.

2. Oculus Quest

Afin de réaliser un build pour Oculus Quest, vous avez besoin des modules Unity suivants :

- Android Build Support
- Android SDK & NDK Tools

- Installez **Oculus Integration** depuis l'Asset Store.
- Unity peut proposer de mettre à jour **OVRPlugin**. Acceptez.
- Même chose pour le plugin **Spatializer**.

Documentation Oculus

https://developer.oculus.com/documentation/quest/latest/concepts/book-unity-gsg/

- Pour configurer votre projet, vous pouvez également vous aider de l'article suivant :

https://medium.com/@sofaracing/how-to-develop-for-oculus-quest-on-macos-with-unity-5aa487b80d13

- Importez les prefabs Oculus dans votre scène (OVRCameraRig et OVRPlayerControler).

Cf. documentation Oculus :

https://developer.oculus.com/documentation/unity/latest/concepts/unity-utilities-overview/

- Testez le déplacement dans la scène virtuelle (via un déplacement physique ou grâce à la manette).
- Essayez d'implémenter un rayon permettant d'attraper et de lancer un objet distant (en utilisant **OVRGrabber** et **OVRGrabbable**).