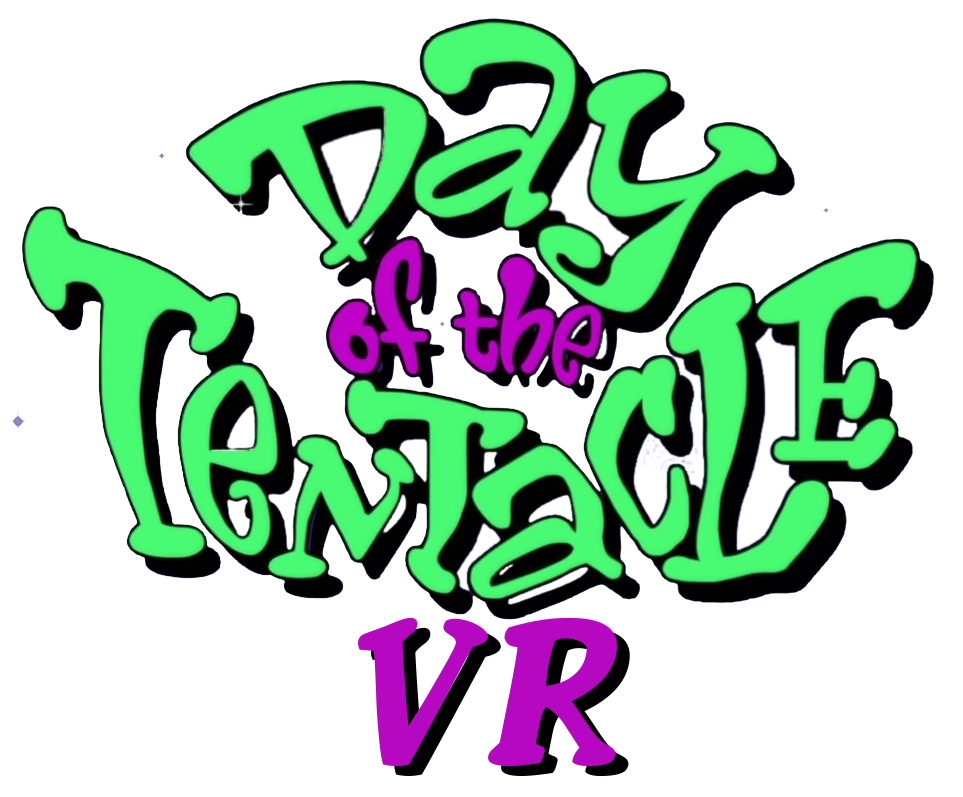
****

ENSEIGNEMENT DE PROMOTION ET DE FORMATION CONTINUE DE L’UNIVERSITE LIBRE DE BRUXELLES ET DE LA CHAMBRE DE COMMERCE ET D'INDUSTRIE DE BRUXELLES

**Rapport d’épreuve intégrée**



Année académique 2021-2022

Travail présenté par **Geoffrey DIELMAN**

en vue de l'obtention d'un bachelier en Informatique de gestion

Supervisé par : **Boris VERHAEGEN**

Table des matières

[Introduction 2](#_Toc116148716)

[Présentation du projet 2](#_Toc116148717)

[Un petit bout d’histoire 2](#_Toc116148718)

[But du jeu 2](#_Toc116148719)

[Mécanique de Jeu 2](#_Toc116148720)

[Fonctionnalités 3](#_Toc116148721)

[Use Cases 3](#_Toc116148722)

[Diagramme de classe 7](#_Toc116148723)

[Architecture 9](#_Toc116148724)

[Choix des technologies 9](#_Toc116148725)

[Apprentissage de la structure et de la programmation dans Unity. 10](#_Toc116148726)

[Développement 16](#_Toc116148727)

[Diagramme de structure d’implémentation 16](#_Toc116148728)

[La structure du diagramme 16](#_Toc116148729)

[Le diagramme 17](#_Toc116148730)

[Le processus d’implémentation 18](#_Toc116148731)

[Le graphisme 18](#_Toc116148732)

[Moove character 18](#_Toc116148733)

[Prendre un objet 26](#_Toc116148734)

[Bibliographie 30](#_Toc116148735)

# Introduction

## Présentation du projet

Le but de ce projet est de reproduire en VR une démo de base du jeu, si célèbre en son temps : **Day Of The Tentacule**. Basé naguère sur le système du ***point and click****,* le gameplay proposé tirerait énormément de bénéfice des progrès technologiques de notre époque et particulièrement de ceux réalisés dans le domaine de la réalité virtuelle.

### Un petit bout d’histoire

Alors que ***Bernard***, ***Huagi*** et ***Laverne***, les trois protagonistes de l’histoire, sont chez eux et vaquent à des occupations aussi absurdes qu’inutiles, un télégramme arrive de la part d’un vieil ami de Bernard: le ***Docteur Fred***. Ce dernier sollicite leur aide afin d’échapper à une catastrophe provoquée par l’un des animaux de compagnie originaux qu’il a génétiquement créé: la ***Tentacule Pourpre***. Cette dernière serait devenue folle suite à l’absorption d’un produit chimique et aurait le projet de dominer le monde.

Une fois arrivés au manoir du Docteur Fred, Bernard, Huagui et Laverne le trouvent ligoté. Ce dernier leur apprend qu’il est désormais trop tard pour empêcher la Tentacule Pourpre de mettre en œuvre son terrible projet. Si on veut l’arrêter, il faudra donc intervenir HIER!

Notre jeu commence au moment où nos trois Héros se réveillent après un accident provoqué par les ***Chrono-WC***, une machine à voyager dans le temps inventée par le savant fou qui les a envoyés chacun dans une époque différente.

### But du jeu

Dans une version complète du jeu, le but serait bien évidemment de trouver le moyen d’empêcher la Tentacule Pourpre de dominer le monde. Dans le cadre de cette démo, il sera simplement demandé au joueur de parcourir les trois premières pièces du manoir auxquelles il a accès pour le moment et d’entrée dans le bureau du Docteur Fred afin d’en savoir plus sur la manière de réparer les ***Chrono-WC***.

### Mécanique de Jeu

En explorant le manoir, le joueur trouvera une quantité d’objets qui lui seront utiles durant sa quête. Il devra se servir judicieusement de chacun d’entre eux afin d’évoluer dans le jeu pour atteindre son objectif. Trouver la clé correspondant à la porte close, utiliser les allumettes sur la cheminée afin de réchauffer George Washington ou encore utilise un poisson mort pour attirer le chat dans une pièce adjacente: autant d’actions qui promettent d’être aussi intéressantes à implémenter qu’à vivre en Réalité Virtuelle.

En effet, chacune de ces actions pourra être réalisée de façon plus interactive que dans l’ancien système via les implémentations d’interactions que je décrirai plus loin dans le cadre de l’analyse des cas d’utilisations.

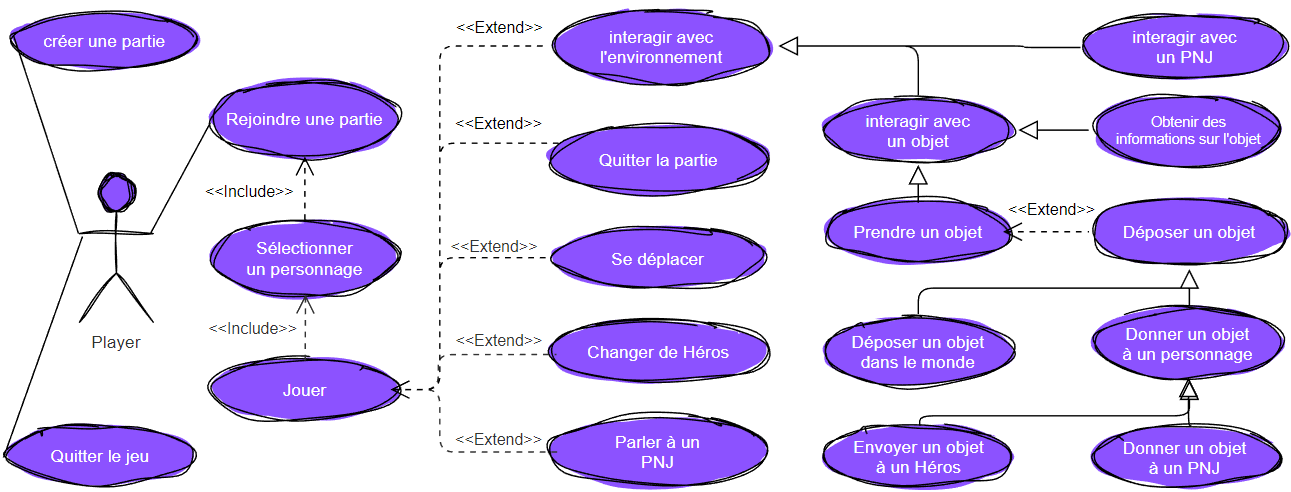
## drFredFonctionnalités

Contrairement au jeu de base, cette version est solo et multijoueur. En arrivant, les joueurs a la possibilité de créer une partie ou d’en rejoindre une existante.

Lorsqu’un premier joueur entre dans une partie, il peut incarner l’un des trois Héros disponibles: Bernard, Huagui ou Laverne. Il a la possibilité de passer d’un Héros à l’autre à tout moment en revenant au menu principal. Si un deuxième joueur rejoint la partie, il peut choisir l’un des deux Héros encore disponibles lors de sa connexion et ainsi de suite. Les Héros vivront simultanément l’aventure dans leur époque respective mais peuvent néanmoins s’entraider en s’envoyant des objets utiles à travers le temps via les Chrono-WC.

### Use Cases

Voici une analyse détaillée des actions possibles au cours d’une partie de ***Dott-VR***.

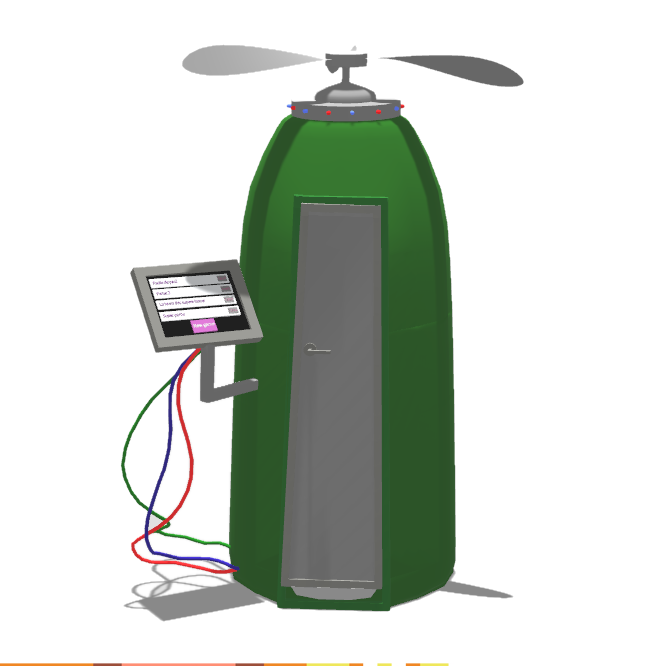


Lorsqu’il lance le jeu, le joueur arrive dans le menu principal. Ce dernier est représenté par un Chrono-WC et son tableau de commande. Le joueur peut à cette endroit *Quitter les jeu*, *créer une partie*, *rejoindre une partie* ou *sélectionner un personnage*.

#### Quitter le jeu

Le premier choix qui s’offre au joueur est de cliquer sur la porte du ***Chrono-WC*** afin de quitter le jeu. Cette action met fin au programme.

#### Créer une partie

 Sur l’écran de commande s’affiche une liste de toutes les parties disponibles sur le serveur et un bouton permettant de générer une nouvelle partie. S’il clic sur ce dernier, une nouvelle partie sera créée sur le serveur avec un nom auto-généré et apparaîtra dans la liste.

#### Rejoindre une partie

Si le joueur sélectionne une partie dans la liste et appuie sur le bouton ***Load***, il lance cette partie. Une fenêtre apparaît pour lui demander lequel des Héros encore disponibles il souhaite incarner.

#### Sélectionner un personnage

Une fois qu’il est dans une partie, le joueur voit sur l’écran les trois personnages disponibles. Pour rappel chacun d’eux correspond à une époque dans laquelle il évolue et le choix de l’un d’entre eux en cliquant sur son image, fera que le joueur rejoindra la partie dans l’époque correspondante.

A ce stade, le joueur est dans la partie, incarnant un des trois personnages dans une des trois époque de la partie. A tout moment il est susceptible de voir si un joueur rejoint la partie pas l’affichage d’un popup en bas de l’écran lui indiquant quel joueur quitte ou rejoint la partie.

#### Jouer

Une fois dans la partie, le joueur se retrouve dans une pièce du manoir dont le décor variera selon l’époque dans laquelle il se trouve.

Il y aura dans chaque époque divers personnages non-joueurs (PNJ) et des objets avec lesquels il pourra interagir, comme nous le verrons plus loin.

Quand il est en jeu, le joueur peut accéder à tout moment à un menu à l’aide de la touche ***Menu*** de la manette de gauche. Il aura alors le choix entre deux options: *Quitter la partie* ou *Sauver la partie*.

#### Quitter une partie

S’il sélectionne quitter la partie dans le menu accessible depuis la manette de gauche quand il est en jeu, le joueur revient au menu principal et est signalé comme ayant quitté la partie auprès des autres joueurs.

#### Sauver la partie

Cette option du menu sauve l’état exact de la partie à ce stade du jeu pour l’époque dans laquelle le joueur se trouve. Un joueur a donc l’occasion de quitter la partie sans sauver les changement qu’il a opéré dans cette époque.

Comme nous sommes dans le cadre d’un jeu multijoueur, il aurait été cohérent de penser à des sauvegarde automatique. Toutefois, je voulais d’un part laisser au joueur la possibilité de ne pas sauver sa progression s’il s’apercevait qu’un de ce choix n’était pas judicieux, et d’autre part le nombre de demande de sauvegarde envoyée au serveur aurait été très importante et aurait pu nuire gravement aux performances.

**Ex:** Si un joueur déplace un objet et que l’on veut garder en temps réel la position de l’objet, il faudrait que la fonction Update() de l’objet en question (dont nous parlerons plus loin) envoie une demande de mise a jours de la base de donnée pour la position et la rotation de chaque objet. Cette méthode se déclenchant un grand nombre de fois par seconde, le serveur recevrait un grand nombre de requêtes à exécuter via le websocket. Chacun d’entre elle déclencherait une requête sql au serveur de base de données, qui serait lui aussi vite surchargé.

Il serait possible de contourner ce problème en gérant manuellement les sauvegarde via un méthode qui ne serait appeler que dans certaine circonstance (quand l’objet touche une surface par exemple) mais cela compliquerait grandement l’implémentation et la gestion des sauvegarde, entraînerait quoi qu’il en soit une multiplication importante des requêtes et n’apporterait pas de réel bénéfice au jeu.

#### Se déplacer

Le joueur peut se déplacer en utilisant le *Stick Analogique* de la manette de gauche ou en se déplaçant physiquement.

#### Changer de Héros

En quittant la partie en cours le joueur se retrouve à nouveau dans le menu principal ou il a donc l’occasion de retourner dans sa partie en incarnant un autre héros.

#### Interagir avec l’environnement

Quand le joueur pointe un élément avec sa manette, l’élément se met alors en surbrillance et il a la possibilité d’intéragir avec lui.

##### Interagir avec un pnj

Lorsqu’un PNJ est en rurbrillance, le joueur peut appuyer sur la touche action de son contrôleur (bouton de gâchette situé près de l’index) afin de lui parler. Ce dernier lui dit alors ce qu’il attend de lui ou lui donne un indice sur son rôle dans l’histoire.

##### Interagir avec un objet

Les interaction possibles avec un objets sont décrites dans les use cases suivants: ***Obtenir des informations sur un objet*** ou ***prendre un objet***.

#### Obtenir des informations sur un objet

Alors que le joueur survole un objet et qu’il est en surbrillance, ce dernier peut utiliser la touche action de son contrôleur pour obtenir des information sur l’objet. Une info-bulle apparaîtra alors afin de lui donner une description de l’objet ainsi, éventuellement, qu’une appréciation sur ce dernier ou sur son utilité.

#### Prendre un objet

Quand un objet est préhensile dans l’environnement, il est mis en surbrillance quand le joueur tend la main vers lui. S’il appuie sur le bouton de Grip de la main qu’il tend, le joueur prend alors l’objet en main. Le bouton de préhension sera à contact fugitif, c’est-à-dire activé uniquement tant que l’on appuie dessus.

#### Déposer un objet dans le monde

Le bouton de préhension étant à contact fugitif, le joueur tient l’objet tant que ce bouton est enfoncé. Il peut donc le déposer dans le monde en lâchant simplement le bouton de *Grip.*

#### Envoyer un objet à un Héros

Un Chrono-WC est présent dans chaque époque et le joueur y a accès tout le long d’une partie.

Sur le panneau de contrôle de ce dernier, le joueur verra deux boutons qui lui permettent de sélectionner le personnage auquel il souhaite envoyer un objet.

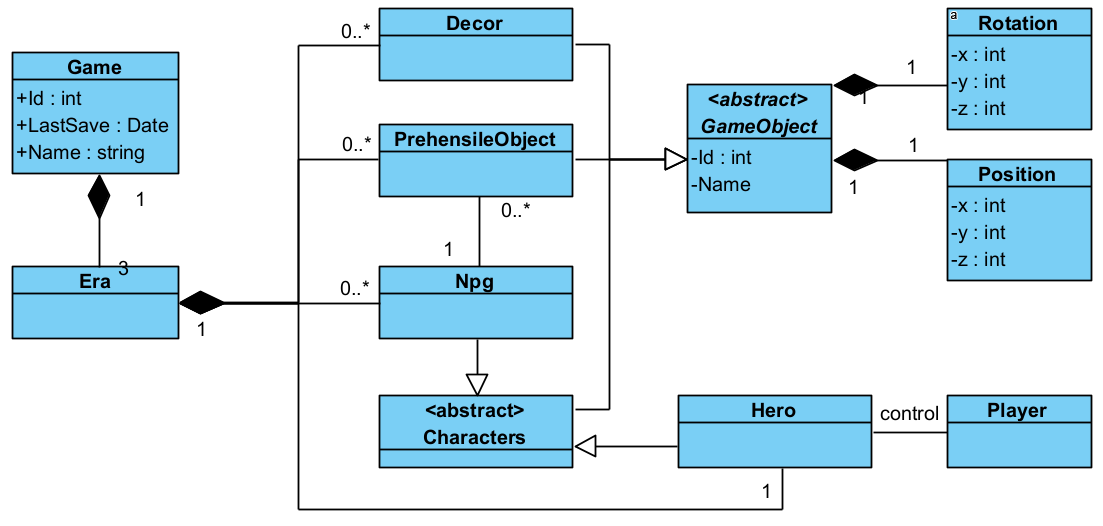
Une fois cette sélection effectuée par simple pression avec sa main virtuelle, le joueur peut jeter l’objet qu’il souhaite envoyer dans la cuvette du WC et l’objet est envoyé.

Le cas échéant, le joueur controllant le personnage de destination pourra alors voir instantanément l’objet envoyé jaillir du ***Chrono-WC***.

#### Donner un objet à un PNJ

Quand un joueur tient un objet en main et l’approche d’un PNJ, ce dernier se met en surbrillance. Cela indique au joueur qu’il peut le lui donner. S’il lâche le bouton de *Grip*et que le PNJ accepte l’objet, ce dernier disparaît de la main et est donné au PNJ. Si la réaction du PNJ indique un refus, l’objet retourne alors dans l’inventaire du joueur.

### Diagramme de classe



#### Game

Il est instancié lorsqu’un joueur charge l’une des sauvegardes ou quand il crée une nouvelle partie. Les attributs d’une partie (game) sont:

**- Id**

Il est un Integer unique et sert à identifier la partie. Il est auto-incrémenté par la système.

**- isNew**

Est un boolean qui permet de savoir si la partie est ouverte pour la première fois. Si tel est le cas une série d’action devront être réaliser, comme le message de bienvenu, un didacticiel sur la façon de jouer, etc.

**- name**

un String qui est uniquement accessible en get() et est une concaténation du mot “partie “ et de l’Id de la partie. Cela permet de s’assurer qu’un partie ait toujours un nom unique.

**- lastSave**

De Type date, il indique le moment de la dernière sauvegarde.

**- eras**

Une partie contient toujours trois époques (Era): Le présent, le passé et le futur. Ils sont liés par un lien de constitution, la suppression d’une partie entraînant, en cascade, la destruction de ces trois époques. La collection d’époque a toujours une longueur de 3 et est initialisée à l’instanciation de chaque partie. On ne peut ni ajouter, ni retirer d’époque d’une partie.

#### Era

Les époques (era) sont des parties du jeu qui sont des mondes dans lesquels les trois joueurs évoluent en parallèle.

Chaque époque contient:

**- Decors**

Collection d’objets Decor. Chaque époque a des décors différents et leur nombre est variable.

- **GrapableObjects**

Collection d’objets préhensiles qui sont présent à la base dans chaque époque mais peuvent passer de l’une à l’autre.

#### Gameobject

Représentent tout les objets pouvant être présent dans le monde. Il s’agit d’une abstraction, qui possède les arguments suivants:

**- Id**

**- Name**

**- Rotation**

Composé de trois integer: x, y et z qui sont compris entre 0 et 360. Ils présentent, en effet le nombre de degrés de la rotation qu’a subit l’objet sur chacun des trois axes.

**- Position**

Composé de trois integer: x, y et z qui servent à localiser un objet dans un environnement tridimensionnel.

#### Decor

Il s’agit des éléments statique du jeu tels que les murs, les portes ou les meubles. Ils ne peuvent être déplacés par le joueur.

#### GrapableObject

A l’inverse ces derniers peuvent être déplace et tenu par le joueur.

#### Npc

Ils sont les personnage non-joueurs (Non Player Charecter) qui sont dans chaque époque.

Npc hérite de la classe *GameObject* et a les attributs suivants:

**- Inventary**

liste de GrapableObject en possession du NPC. Il peut donner l’un de ces objets au joueur ou en recevoir un.

**- AcceptedObjects**

liste de GrapableObject que le NPC peut accepter dans son inventaire.

**- Dialogues**

Listes de strings qui sont les phrases que peut dire le Pnj.

Héros

Les Héros sont aussi des personnages mais ils sont liés à un joueur qui les contrôle.

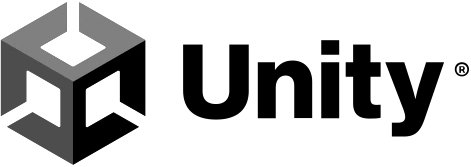
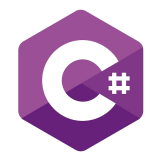
## Architecture

Le choix du multijoueur a été fait afin de pouvoir mettre ne place une architecture backend - frontend. Le sauvetage des données nécessitera l’utilisation d’une Db.

### Choix des technologies

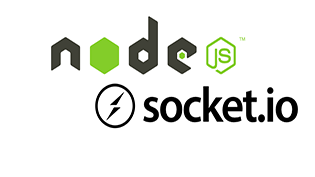
#### Frontend : le jeu

Ce jeux sera développé sur Unity qui utilise le language C# pour ces écritures de script.

#### Backend: le serveur de jeu

Je développerai le backend en Node Js, cette technologie étant utilisée par Socket IO qui me servira à assurer une communication constante entre les backend et le frontend en cours de jeu.

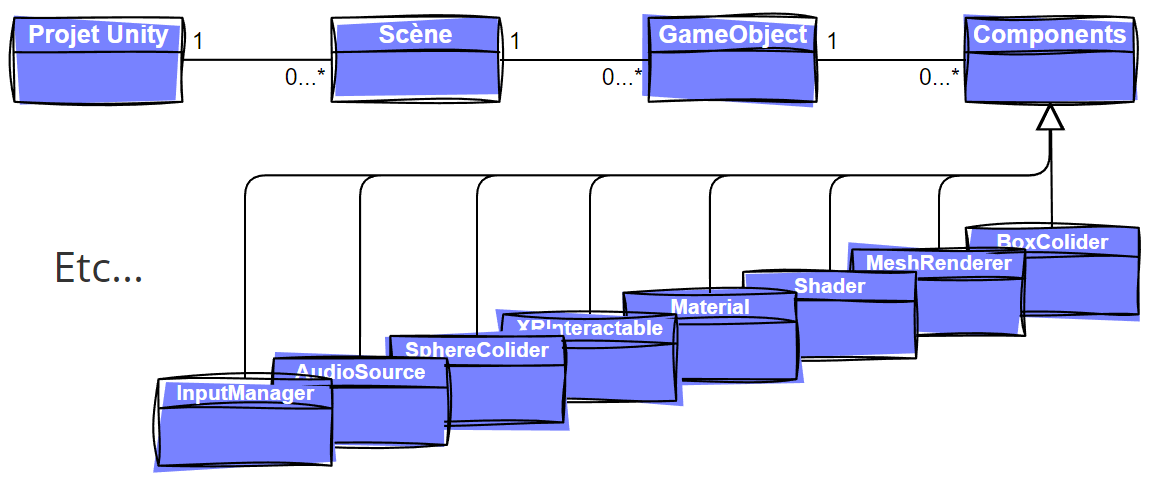


#### Base de donnée



### Apprentissage de la structure et de la programmation dans Unity.

L’architecture de Unity est vraiment particulière et il est vraiment très difficile de la classer dans les architectures connues telles que MVC ou MVVM. Je schématiserais un projet Unity de cette façon:



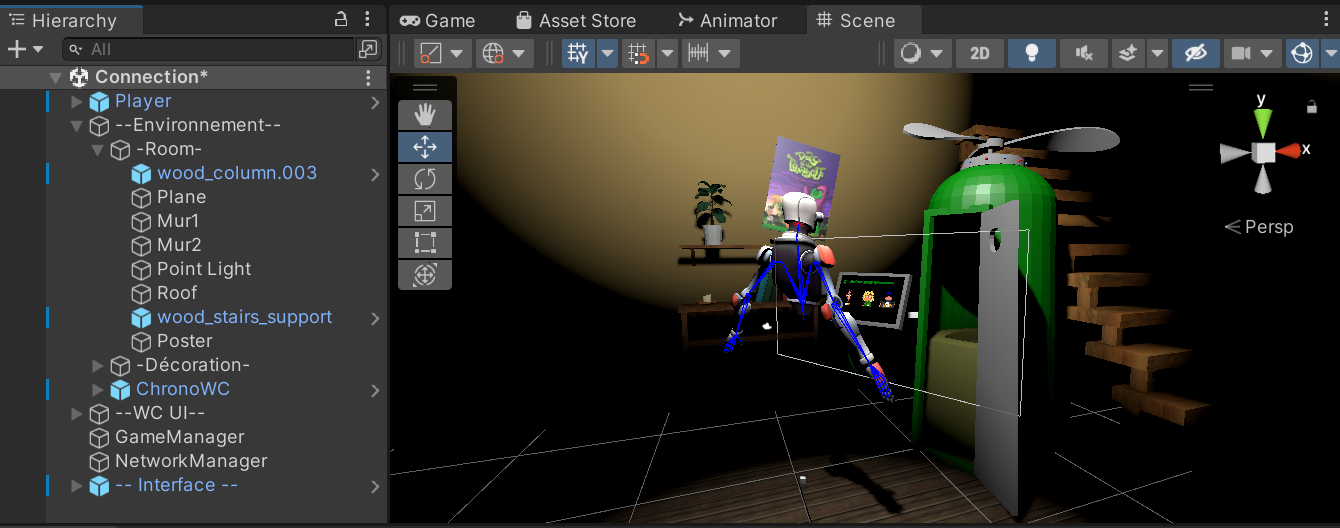
#### Les scènes

Un programme Unity est composé de scènes. Ces dernières sont classées par ordre croissant dans la fenètre de paramétrage du Build afin de leur donner un ordre de priorité et permettre au compilateur de savoir quelle scène il doit définir comme point d’entrée du programme.

Une fois qu’une scène est lancée, elle reste active jusqu’à ce qu’on donne l’instruction au SceneManager de charger la suivante. Une fois la nouvelle scène chargée, l’ancienne est automatiquement détruite avec tout ce qu’elle contenait (sauf exception en utilisant DontDestroyOnLoad() que nous verrons plus loin, **page XXX** ) .

Chaque scène est composée d’un nombre illimité de *GameObject*s.

#### Les GameObjects



Chaque *GameObject* est un élément ajouté à votre scène et apparaît dans la vue 3D de Unity. Tous ont une positon et une rotation destiné à les placer judicieusement dans la scène.

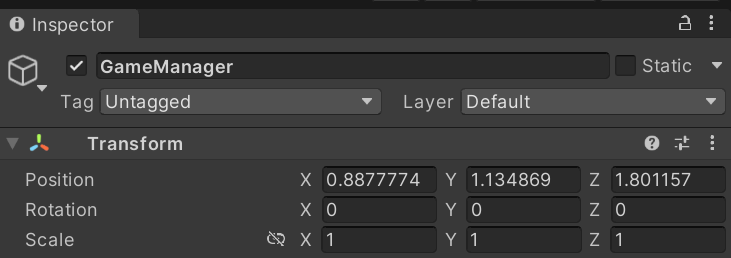
Partant de ce principe, on pourrait comparer un *GameObject* à une Vue dans le model MVC. Ce n’est toutefois pas rigoureusement exact. Car si un *GameObject* peut avoir un rendu visuel (un personnage, une maison, un monstre, un objet sur la table, etc...) ce n’est pas nécessairement le cas. Certain d’entre eux seront seulement des *GameObject* vides destinés a contenir des script ou à créer une hiérarchie.

##### Le GameManager

Le meilleurs exemple de ceci est sans aucun doute le GameManager, que 9 développeurs Unity sur 10 utilisent comme contrôleur général du jeu et qui contient, en gros, un script reprenant toutes les méthodes générale du jeu (sauver la partie, quitter, afficher un message, etc.).

Cet élément est un *GameObject* nommé GameManager contenant un script du même nom (qui est en fait un ***Component*** que nous définirons plus loin) mais n’est absolument pas visuel. Si malgré tout on voulait continuer la comparaison avec le model MVC, ce *GameObject* devrait plutôt être associé à un contrôleur. Ce sera aussi le cas d’autre *GameObject*s comme le NetWorkManager qui s’occupe de toute la communication réseau.

##### Les tags

Quand un GameObject est selectionné, on peut voir dassn l’inspecteur toutes ces propriétés ainsi que ces composants (***Components***) que nous aborderons dans le chapitre suivant.

Dans ces propriétés, on retrouve un combo-box “**Tag**”. Les tags permettent de distinguer les GameObjects entre eux et de faire des sortes de catégories. Ces catégories nous permettrons d’appliquer certains effets sur des GameObjects spécifique comme nous le verrons dans le chapitre suivant.

##### Active

La check-box présente dans l’inspecteur à coté du nom du GameObject permet d’activer ou désactiver le GameObject. La désactivation d'un [GameObject](https://docs.unity3d.com/ScriptReference/GameObject.html) désactive chaque composant, y compris les rendus, les collisionneurs, les corps rigides et les scripts attachés.

##### Les préfabs

<<Parler des préfabs ici>>

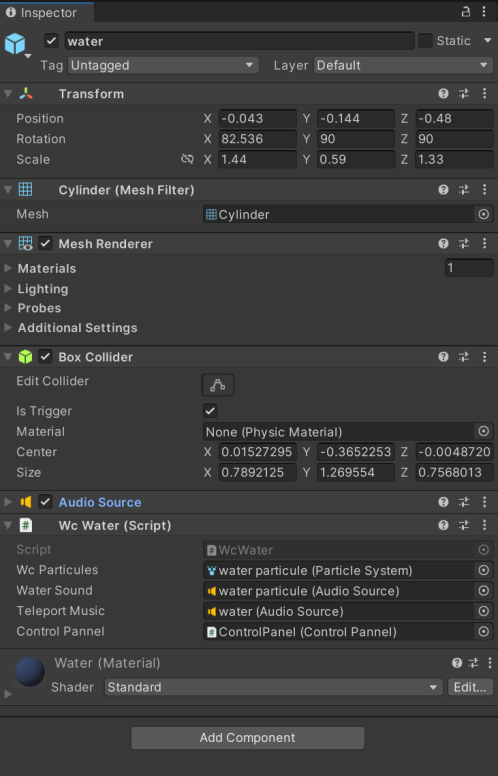
#### Les Components

Comme leur nom l’indique, ils composent le GameObject et se composent en réalité de scripts en C#. Chaque component est en fait une classe. C’est donc ici que le coté programmation prend sa place dans Unity.

Le composant commun à tout les Gameobjects est le ***Transform*** qui contient la position et la rotation de chaque objet et qui sera automatiquement présent.

L’ajout de certains GameObjects, comme par exemple un cube, entraîne automatiquement l’apparition de certains components nécessaire en son sein. Ce cube est une espèce de préfabriqué (nommé ***préfab***) proposé par Unity qui comporte déjà des éléments nécessaires a son affichage mais abstraction faite de l’utilisation de ce type de ***préfab*** c’est en règle général au développeur d’ajouter lui-même ceux dont il a besoin.

Beaucoup de composants utilisés dans un projet sont des standards de Unity néanmoins l’élaboration de scripts reste indispensable afin de personnaliser l’expérience que l’on souhaite donner au joueur.

Dans l’élément exemple ci-contre, on voit que l’élément “water”, qui représente l’eau présente au fond de la cuvette du chrono-Wc, a les compnents suivants:

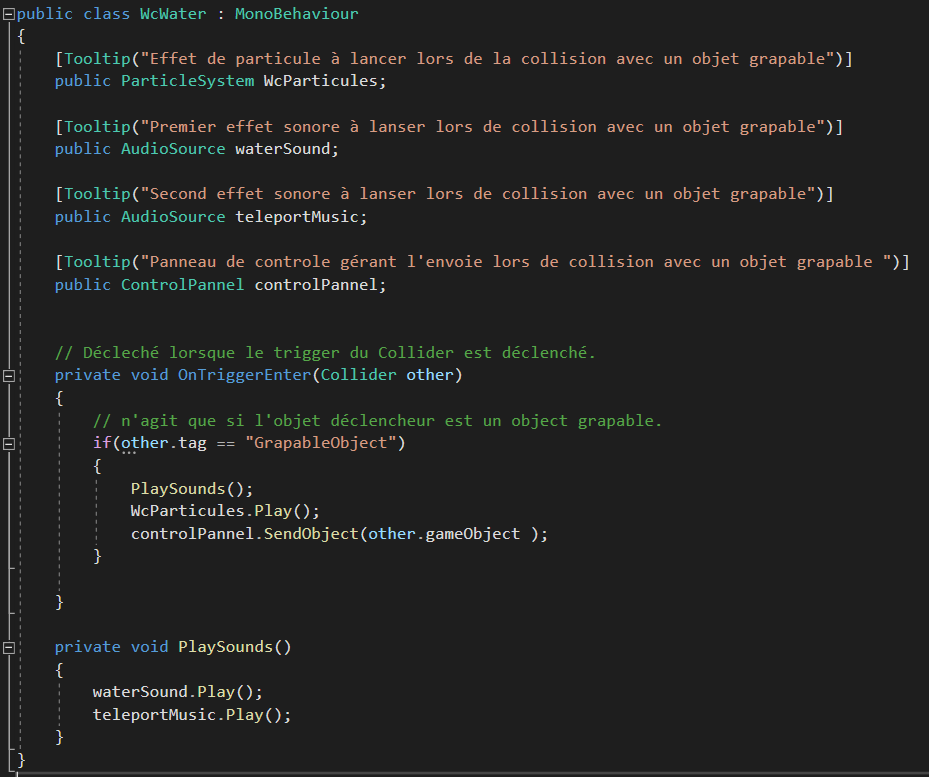
* Le *Transform* commun à tout GameObject
* des meshs pour le rendu visuel,
* un box collider qui sert à détecter la colision de l’eau avec un GameObject,
* une source audio qui permet de lancer un petit effet audio
* un script personnalisé “*WcWater.cs*” qui gère le tout.

Nous reviendrons sur la composition détaillée du script “WcWater.cs” plus tard mais pour bien comprendre son rôle, disons seulement qu’il sert à déclencher tout les comportements appropriés lorsque le Collider lui signale la détection d’un objet grappable.

Il lance alors des effets sonores, un l’effet de particule dont on lui a donné la référence et amorce le processus d’envoi de l’objet via le ControlPannel.

##### Les scripts vue de plus près

Reprenons comme exemple le Component “*WcWater.cs”* que nous avons observé ci-dessus.



Tout les scripts que nous créons pour Unity héritent de la classe MonoBehaviour ce qui leur donne accès à certaines méthodes et variables.

###### Les méthodes

Bien évidement, il est toujours possible de créer des méthodes personnalisée, quelles soient privées ou public, comme *Playsound()* dans l’exemple ci-dessus.

L’héritage de *MonoBehaviour* nous donne toutefois accès à toute une série de méthode indispensables afin de pouvoir manier Unity comme nous le désirons.

Nous voyons dans l’exemple ci-dessus la méthode ***OnTriggerEnter()*** qui se déclenche lorsqu’un GameObject entre en collision avec n’importe quel collider présent dans le GameObject auquel le script est attaché. Dans cette méthode, je vérifie si l’objet qui a déclenché la méthode porte le tag “Grapable”. On retrouve donc ici l’utilisation des ***tags*** qui, en occurrence, sont bien pratiques afin de n’appliquer les effets de cette méthode que sur les objets que le joueur aura lancé dans les toilettes et non, par exemple sur le bras que le joueur aurait pu y introduire.

Parmis les méthodes les plus utilisées mais qui ne sont pas illustrées dans notre exemple, nous retrouvons:

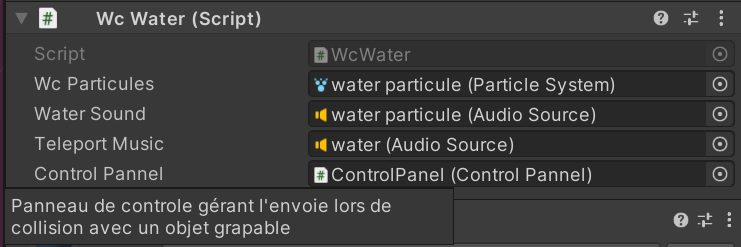
* ***Start()*** est appelé lorsqu'un script est activé juste avant que l'une des méthodes Update ne soit appelée pour la première fois.

Comme la fonction Awake, Start est appelée exactement une fois dans la durée de vie du script. Cependant, Awake est appelé lorsque l'objet script est initialisé, que le script soit activé ou non. Start ne peut pas être appelé sur la même trame que Awake si le script n'est pas activé au moment de l'initialisation.

* ***Awake()*** est appelée sur tous les objets de la scène avant l'appel de la fonction Start d'un objet. Ce fait est utile dans les cas où le code d'initialisation de l'objet A doit s'appuyer sur l'objet B déjà initialisé ; L'initialisation de B doit être effectuée en Awake tandis que celle de A doit être effectuée en Start.
* ***Update()*** qui permet d'exécuter des actions à chaque frame. Une seconde de jeu est composée de plusieurs frames mais ce nombre de frames est dépendant de la vitesse de l'ordinateur. Un ordinateur peut très bien afficher 30 frames par seconde, un autre 60 frames par secondes. Il faut donc user de cette méthode judicieusement de manière à ne pass surcharger le programme, cette méthode se déclenchant entre 30 et 60 fois par secondes.

Il existe encore bien des méthodes héritées de MonoBehaviour qui peuvent être utiles mais elles ne seront pas utilisées dans le cadre de ce travail. On peut toute les retrouver dans la documentation Unity (https://docs.unity3d.com/ScriptReference/MonoBehaviour.html).

###### Les propriétés

Comme dans toute classes C#, on peut créer à loisir des propriété qui représenterons ce dont nous aurons besoin dans notre script. La particularité dans unity est que les propriété public seront visibles et assignable dans Unity.

On remarque dans l’exemple de “*WcWater.cs”* que les propriété publiques apparaissent dans l’inspecteur et que ces dernières sont assignable simplement en faisant un *drag and drop* de l’objet désiré dans la case de propriété.

L’usage des Tooltip dans le code, permet par ailleurs l’ajout de commentaire qui non seulement sont utiles au développeur dans le code (comme tout commentaire classique) mais également dans l’interface de Unity qui affichera ce commentaire en infobulle lors du survole du nom de la propriété avec la sourie.

Cela permet d’une part à une personne qui n’aurait pas conçue le code de comprendre quelle propriété il doit mettre à quel endroit mais également de rappeler cette information au développeur quand il utilise le ***component*** longtemps après avoir écrit son script.

#### Conclusion de l’analyse de la structure de Unity

Comme nous pouvons le constater, Unity à une structre particulière. Dans une implémentation classique d’application telle que nous les avons vu en cours, il peut structurer le travail en faisant un diagramme de classe d’implémentation en se fiant aux classe des objets et en se basant sur leurs héritages. On sait qu’un objets de type “écoles” comprend des ojets de type “classe”, qui contiennent eux-même des élèves, etc.

La difficulté ici est que toute les entités du jeux sont des ***GameObjects*** et qu’il n’est pas possible de les différencier au sein d’un diagramme en se fiant à une classe qui les différencie. Chaque GameObject reçoit des scripts qui ont chacun une classe mais l’entité en elle même reste une instance de la classe ***Gameobject.***

Malgré mes recherches je n’ai pas trouvé de façon reconnue d’implémenter un diagramme de classe avec le système Unity tel qu’il existe. On constate d’ailleurs sur le site de microsoft que ce problème est signalé et qu’il est reconnu que des diagrammes d’analyse et une modélisation classiques ne sont pas envisageables.

(*https://learn.microsoft.com/fr-fr/visualstudio/gamedev/unity/application-lifecycle-management-alm-with-unity-apps*)

Il m’a donc fallu adapter légèrement les choses afin de pouvoir implémenter un diagramme logique. Je ne parlerai donc pas de “Diagramme de classe” mais plutôt de “Diagramme de structure” en m’appuyant sur la logique d’un diagramme de classe classique.

### Cycle de vie du jeu

**<< A écrire>>**

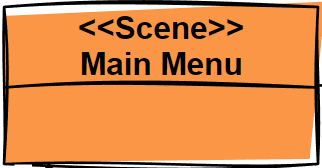
# Développement

## Diagramme de structure d’implémentation

Comme nous l’avons vu plus hauts, certains aménagements sont indispensables afin de pouvoir implémenter un diagramme de classe. J’ai commencé par annoter et coloriser les différentes cases du diagramme en fonction de ce qu’elles sont.

### La structure du diagramme

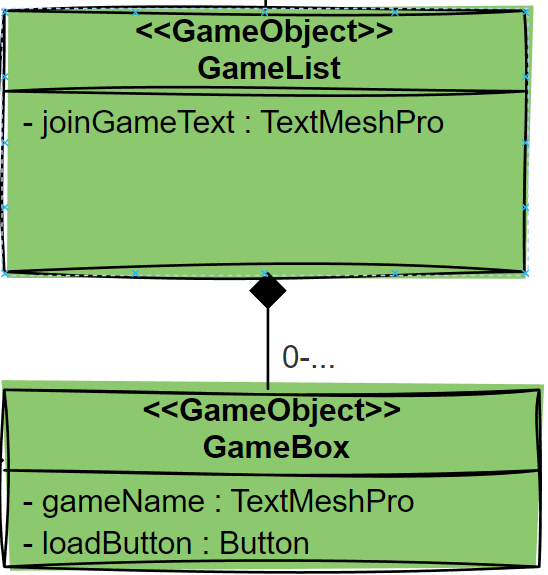
#### Les scènes

Elles sont en orange et portent la balise <<Scene>>. Le nom de chaque “class” du diagramme correspond au type auquel elle appartient. Quand on parle de type ici, ce n’est pas en rapport avec les class mais plutot avec la sorte de scène dont il s’agit. Ci-dessous nous parlerons de ***Type*** en se référent non-pas à une classe mais au Tag attribué à un groupe de scène comparables à ceux que nous pouvons appliquer aux GameObjects.

En effet, dans un diagramme classique on représente des class qui sont en fait des espèces de modèles qui servent à instancier des objets y correspondant. C’est ce raisonnement de base que je tente ici d’appliquer avec les “Tags”. Ainsi donc, toutes les scènes ayant le même ***Tag*** sont semblables, construites sur le même modèle et donc représenté par un ***Type*** comme s’il s’agissait d’une classe.

Dans l’exemple, ci-contre, on voit la scène de Type “MainMenu”. Cette case de diagramme représente donc toute scène ayant le Type MainMenu qui auront toutes la même composition.

#### Les GameObjects

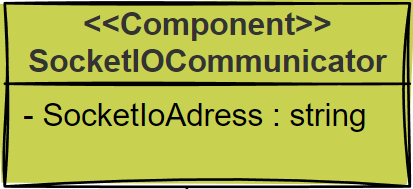
Ils sont en vert et portent la balise <<GameObject>>. Comme pour les scènes le nom que je donne à chacune des case “GameObject” du diagramme corespondra à un Tag. A la différence des scènes, les Tag ne sont plus seulement théoriques car ils sont bel et bien présents dans Unity, comme nous l’avons vu plus haut.

Comme on peut le voir dans l’exemple ci-contre tout ***GameObject*** ayant le type “*GameList*” a une collection de ***GameObject*** de type “*GameBox”*.

Chacun d’eux a aussi des ***Components*** mais nous verrons ça dans le chapitre suivant

#### Les Components

Ici peuvent atténuer les adaptations puisque les Component, eux sont des classes C# et peuvent donc être représentées de manière classique au sein de ce diagramme. La seule petite adaptation que nous ferons encore est de réprésentrer les Components différemment des Classes traditionnelles en leur attribuant une couleur.

En effet, les ***Components*** sont des classe particulières qui héritent de *MonoBehaviour* et qui jouent un rôle bien spécifique dans Unity comme nous l’avons vu plus haut.

Ainsi donc, le composent SocketIOCommunicator est une classe C# qui a hérité de MonoBehaviour et a une propriété SocketIoAdress de type string.

### classdiagramm (3)Le diagramme

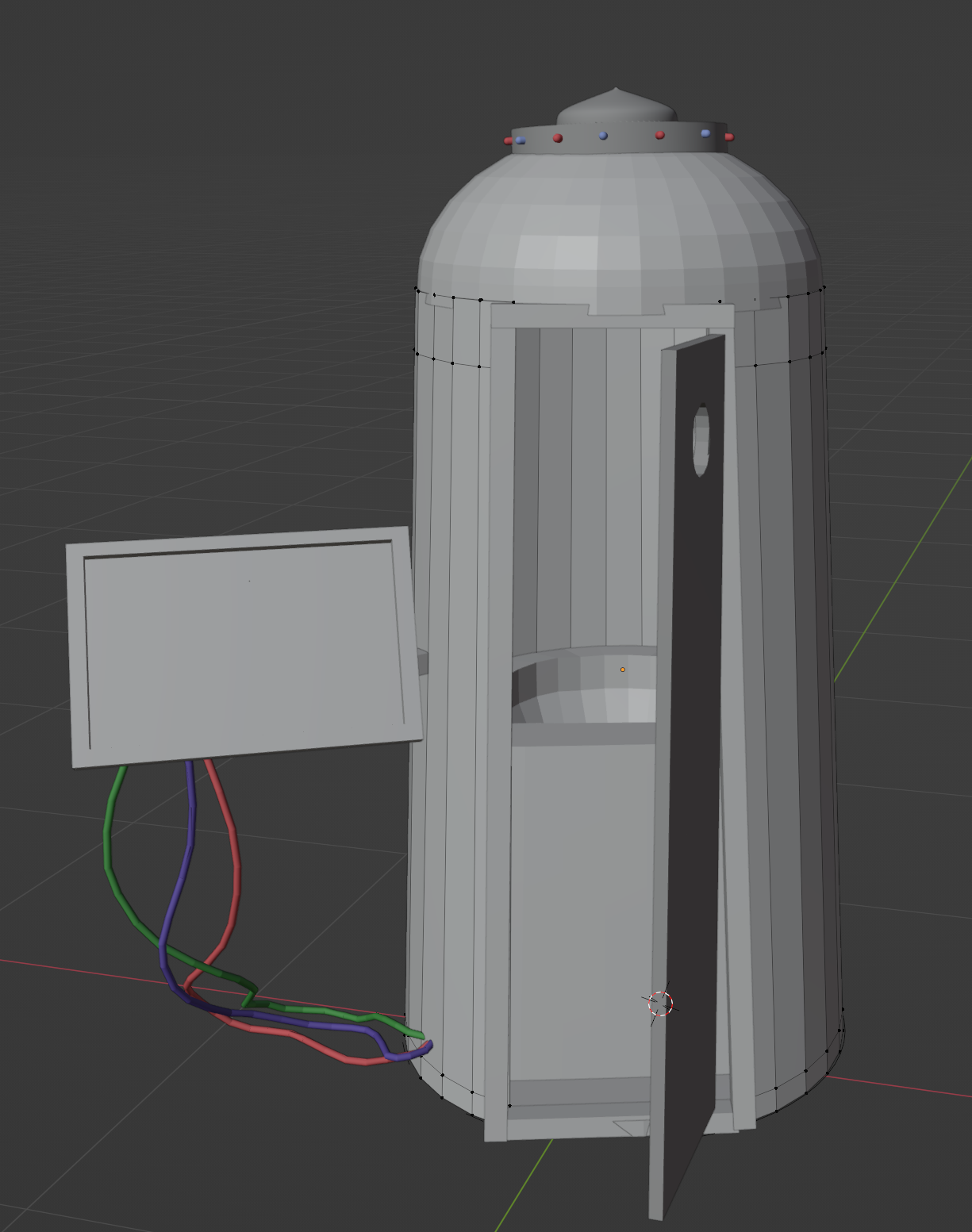
## Le processus d’implémentation

Dans ce chapitre je décrirai pas à pas les étapes du processus d’implémentation. J’ai suivi un processus de développement basé sur les **Use Cases**, en les implémentant les uns après les autres. Je reprendrais donc dans l’ordre chronologique les **Use Cases** que j’ai développé et expliquerai tout ce qui a été implémenté en me référant au diagramme de classe ci-dessus. J’aborderai pour chaque use case l’aspect technique mais aborderai également les obstacle rencontré, les bugs qui se sont présentés et ceux qui restent encore un problème.

Nous verrons également dans ce chapitre les choix qui ont dû être fait au niveau de l’implémentation et les améliorations à apporter éventuellement dans le futur. En effet, il a été nécessaire de faire preuve de réalisme dans ce travail. Développer un jeu complet est un travail énorme qui est généralement celui de toute une équipe de développeur pendant plusieurs années. Il m’a donc fallu faire toute une série de choix de manière à obtenir quelque chose de modeste mais qui peut tout de même être considéré comme une démo de jeu complète.

Avant de passer en revue chaque étapes, abordons les choix d’ordre généraux faits.

### Le graphisme

Comme nous en avons discuté avec monsieur Verhaegen, il est évident que le graphisme n’est pas la priorité de se travail. Il était toutefois aisé de donner un design élégent à ce jeu en ayant recours aux assets. Les assets sont des préfabs que l’on peut télécharger sur l’asset store de Unity (*https://assetstore.unity.com/*). On peut aisément y trouver des chaises, des tabourets, des objets divers, des personnages, des animations, etc. J’ai donc décidé de faire appel aux assets pour la majore partie des choses figurant dans ce travail.

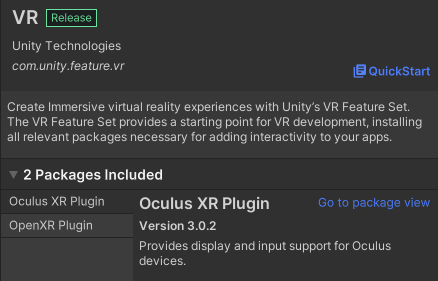
J’ai toutefois décidé de m’intéresser quelque peu à la construction d’éléments graphiques. En effet, il existe une relation étroite entre la partie graphique d’un objet et la programmation de ce dernier. Apprendre sommairement de quoi se compose une préfab m’a permis de comprendre comment l’on pouvait lui programmer des comportements, des animations, etc. J’aborderai ceci plus en détail dans les chapitres qui suivent lorsque nous rencontrerons l’un de ces éléments.

Par ailleurs, certain des éléments dont j’avais besoin afin de rester cohérent avec l’histoire n’était pas disponible dans le store. Ainsi donc, les deux grands éléments qui sont une création personnelle sont le manoir en lui même et les ***Chronos-WC***.

### Déplacer le personnage

La première étape fut de construire un personnage qui soit capable de bouger en suivant les mouvements traqués par le casque VR et les manettes.

Pour se faire, la première étape est d’importer dans le projet les packages nécessaires :

* Occulus XR Plugin

Nous permettra de rendre le projet compatible avec le casque Meta Quest 2 (anciennement Oculus Quest 2) et de fournir a Unity toutes les instructions concernant les contrôleurs.

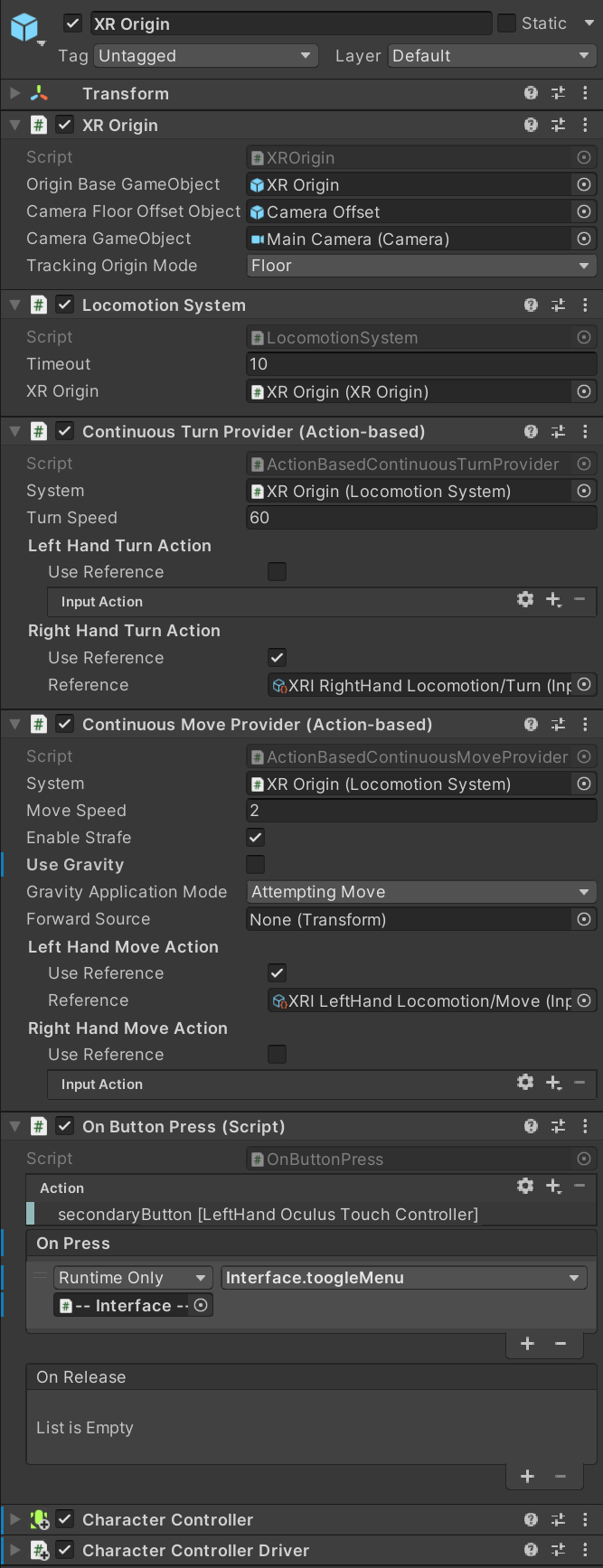
* OpenXRPlugin

OpenXR est une norme ouverte et libre de droits développée par Khronos qui vise à simplifier le développement AR/VR en permettant aux développeurs de cibler de manière transparente une large gamme d'appareils AR/VR.

Une fois que c’est fait on a la possibilité d’ajouter un GameObject rendu disponible: le ***XROrigin***. Ce dernier sert à implémenter le casque et les manettes et à les relier à Unity. Dés que l’on a ajouté le ***XROrigin*** et qu’on l’a correctement configuré on peut déjà mettre le casque relié au PC, entrer en ***Play Mode*** et voire le monde dans le casque ainsi que les manette. Regardons d’un peut plus près le ***GameObject XROrigin***.

#### XROrigin

Voici la composition du GameObject XROrigin :

* Un component **XR Origin**

Il a comme paramètres:

- Son gameObject parent

- Le cameraOffset qui est le Gameobject qui représente le décalage qui se fait quand un joueur a le casque et qu’il marche dans la pièce sans pousser sur le joystick de la manette.

- La MainCaméra qui est le Gameobject qui renvoie les image qu’il capture pour les renvoyer au casque.

- Un paramètre nous permettant de déterminer à partir de où l’on veut traquer la hauteur à laquelle sera placé la caméra. En indiquant “**Floor**” on signifie que la position de la caméra sera calculée en fonction de l’espace qui sépare le casque du sol.

* Un component **Locomotion System**

Il est utilisé pour contrôler l'accès à **XR Origin**. Le système impose qu'un seul **Locomotion System** à la fois puisse déplacer le **XR Origin** .

C'est le seul endroit où l'accès à un **XR Origin** est contrôlé car avoir plusieurs instances d'un [**Locomotion System**](https://docs.unity3d.com/Packages/com.unity.xr.interaction.toolkit@2.0/api/UnityEngine.XR.Interaction.Toolkit.LocomotionSystem.html) conduisant un seul **XR Origin** n'est pas recommandé car risque de créer des conflis.

* Un component **Continuous Turn Provider**

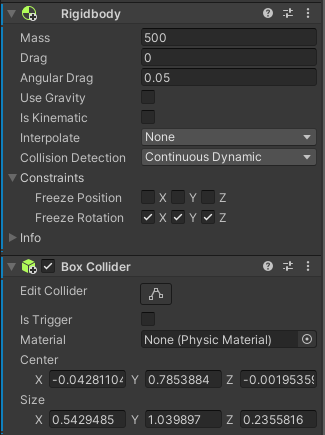
Permet à l'utilisateur de faire pivoter en douceur le **XR Origin** en continu au fil du temps à l'aide d'une action d'entrée spécifiée.

* Un component **Continuous Move Provider**

Permet à l'utilisateur de déplacer en douceur **son XR Origin** en continu au fil du temps à l'aide d'une entrée d'axe 2D spécifiée.

* Un component **Onbutton press**

Il est relié a l’interface et lui demande d’ouvrir le menu à tout moment dans le jeu sauf si la scène active est le menu principal. En effet, comme nous le verrons plus loin, le menu ne sert qu’à sauver et a quitter la partie en cours, ce qui n’est pas utile si l’on est déjà dans le menu principal et donc hors d’une partie.

*  Un component **Box Collider**

Unity gère les collisions entre les **GameObjects** avec des **Collider**, qui s'y attachent et définissent leurs formes afin de gérer les collisions physiques. Un collisionneur est invisible et n'a pas besoin d'avoir exactement la même forme que le GameObject.

On décide ici d’utiliser un collider en forme de Box (ou de cube) qui permettra de détecter toute collision avec les murs mais pas encore à éviter que notre joueur ne passe à travers les murs.

* Un component **Rigid Body**

Le rigidBody permet de donner des caractéristiques physiques à un objet. Ce dernier aura alors une vrai consistance, son Collider deviendra rigide et il pourra être soumis à la gravité. Comme dans le cadre de *Day of the tentacule* la gravité n’a aucune importance, nous laisserons cette option décochée.

C’est grasse à l’ajout de cet élément que nous pouvons éviter de passer à travers les murs.

Problème rencontré

Quand j’ai mis en place le **Box Collider** la première fois, tout un tas de problèmes car je n’avais pas compris de prima bord que ce qui empêchait le personnage de passer a travers les murs était le Rigid Body.

Une fois que ce dernier a été implémenté, j’ai eu envie de donner au personnage une taille moyenne humaine de 1m70. Toutefois, si le joueur commence la partie assis, comme la position de la caméra est initialisé par rapport au sol, le **Box collider** il est é moitié enterré dans le sol. Comme il est prévu que le personnage ne puisse avancer quand il percute quelque chose, le sol autour de lui sert de carcan et il ne peut plus avancer. La solution fut de donner au **Collider** une grandeur de 1m. De cette manière il couvre une hauteur suffisante pour percuter tous les obstacles mais pas suffisante pour être enterré dans le sol au début du jeu.

Le problème suivant était que comme les mur repoussent notre personnage, ce dernier se mettaient a tournoyer incessamment. Pour éviter ceci, j’ai du interdire la rotation sur tout les axes au **Box Collider**.

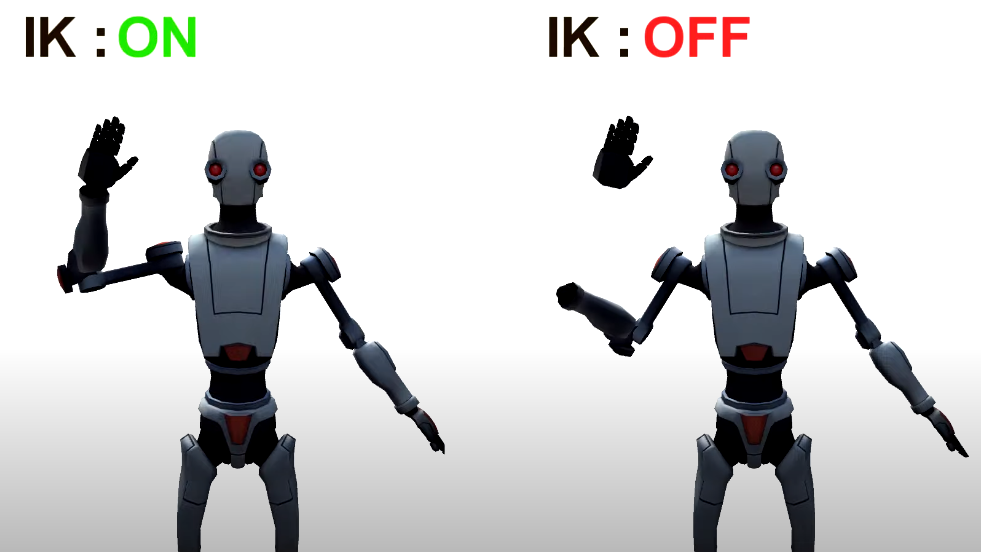
L’étape du **XR Origin** est instancié et nous pouvons désormais bouger sans passer à travers les murs. Toutefois, il est maintenant nécessaire d’afficher un personnage et de faire en sorte que ce dernier bouge avec en synchronisation avec le personnage.

#### Robot Kyle

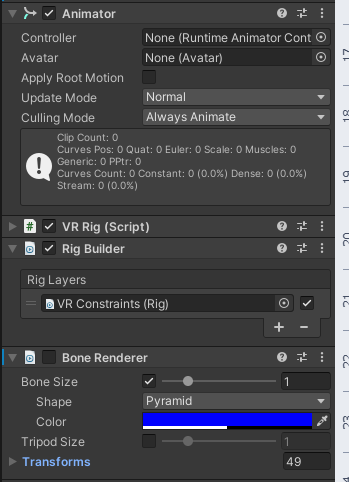
Comme précisé plus haut nous allons utiliser un personnage préfabriqué qui est animable. Pour qu’il le soit, il doit avoir un **Rig** c’est a dire un squelette qui, lorsqu’il bouge déforme de **mesh** de rendu de manière à ce que les mouvements soient harmonieux.

L’asset gratuit que j’ai pu trouver et qui répondait à ces conditions est **Robot Kyle**. C’est donc de lui dont je me servirais pour représenter tout mes personnages joueurs.

Afin de faire en sorte d’animer notre personnage, nous allons utiliser une technique qu’on appel l’**Inverse Kinematic** . c’est une technique qui consiste a faire bouger l’ensemble du corp en se basant sur les mouvement d’une seule de ces partie.

Si on observe Robot Kyle on voit en effet qu’il est composé d’un tas de partie du corp différente. Dans l’exemple ci-contre, on peut voir que si je bouge sa main sans appliquer l’**Inverse Kinematic**, la main se détache du corps et le bras ne la suit pas.

Pour mettre ceci en application, nous allons nous servir du package **Animation Rigging** de Unity qui va nous permettre de maintenir ensemble des parties du corps.

Nous allons commencer par ajouter à Robot Kyle le component **Rig Builder**. Ce dernier apparaîtra systématiquement avec un composant **Animator** dont il dépend pour assurer son bon fonctionnement.

Afin de faciliter la selection des différents os avec lesquels je vais travailler, je vais ajouter un Rig Builder qui va permettre d’afficher les différents os que possède **Robot Kyle**. Une fois que le composant est ajouté, je drag and drop tout les os de RobotKyle Dans la liste “Transform” qui est tout en bas et je peux désormais voir une représentation graphique de chaque os dans l’éditeur de Unity.

Nous remarquons aussi un composent **VR Rig** qui est un script que j’ai écrit et sur lequel je reviendrais un peu plus bas.

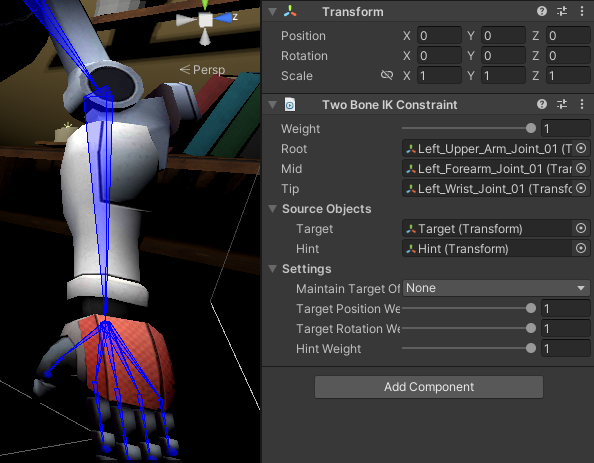
J’ai maintenant créé un Gameobject que j’ai appelé **VR Constraints** et qui comporte seulement un élément Rig qui est le point d'entrée principal pour toutes les contraintes dans un Rig de contrôle donné. J’ai donné une référence vers mon nouveau gameObject au **Rig Builder** qui sait désormais ou trouver les contraintes qu’il va devoir appliquer à **Robot Kyle**.

Maintenant que nous avons le point d’entrée de nos contraintes, nous allons y ajouter trois GameObjects enfants qui correspondent aux trois contraintes qui nous intéressent: “**Right Arm IK”**, “**Left Arm IK”** et “**Head Constraint”**.

##### Les Arm IKs de Robot Kyle

Aux contraintes des deux bras(**Right Arm IK** et **Left Arm IK”)**, nous allons ajouter un component qui se nome **Two Bone IK Contstraint**. Au moment ou on ajoute se composent, on voit que deux GameObjects lui étant utiles ont été créés.

La contrainte **Two Bone IK** permet d'inverser le contrôle d'une hiérarchie simple de deux GameObjects, afin que le bout d'un membre puisse atteindre une position cible (l’objet Target). Un GameObject Hint supplémentaire permet de spécifier la direction dans laquelle le membre doit être orienté lorsqu'il se plie.

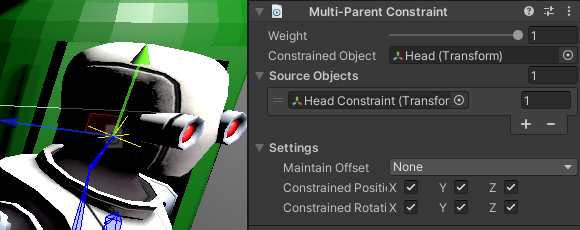
Il ne nous reste plus qu’à renseigner **Two Bone Ik Constraint** sur les os qu’il va devoir contrôler et placer **Target** pile a l’endroit de la main et **Hint** un peu derrière le coude pour indiquer le sens de la pliure.

Désormais lorsque j’entre en Playmode et que je déplace le GameObject cible “**Target”**, le bras part dans la bonne direction et se plie au bon endroit.

Il reste maintenant à faire en sorte que ce dernier suive les déplacements de la manette VR ce que nous ferons un peu plus loin.

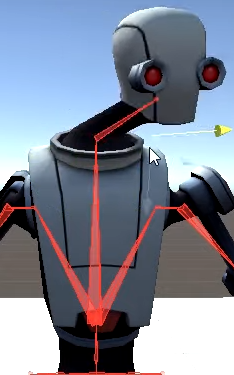
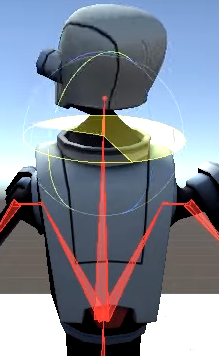
##### Le HeadConstraint de Robot Kyle

Occupons-nous maintenant de la tête. Pour ce faire, je vais maintenant m’occuper du **Head Constraint** et faire en sorte que comme ces compères il me permette d’avoir un **Target** qui va me servir de “poignée” que je pourrais bouger et faire tourner pour contrôler la tête. Mon but est bien entendu in-fine de synchroniser ce Target avec le casque vr de manière à faire reproduire parfaitement les mouvements de ma tête à celle de **Robot Kyle**.

Comme pour les mains nous allons ajouter un composent au **Head Constraint** mais ce dernier sera un peu différent.

Ici il nous suffit d’ajouter un **Multi-Parent Constraint** qui va simplement déplacer et fait pivoter l’os de la tête comme s'il était l'enfant du Target dans la fenêtre Hiérarchie.

Si nous regardons le résultat de ce que nous avons fait, nous voyons qu’il reste deux petits problèmes.

D’une part, si l’on fait une rotation du GameObject Head Constraint, la tête est synchronisée comme nous le voulions mais pas le reste du corp qui ne tourne pas.

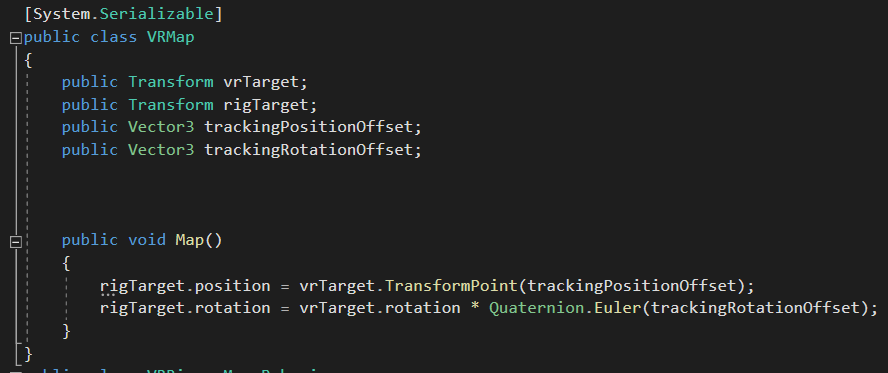
D’autre part, on voudrait que lorsque le casque VR bouge, tout le corps le suive, ce qui n’est, pour le moment, pas le cas.

Je m’occupe de ce problème dans le script que j’ai écris ci-dessous et que va gérer les mouvements du personnage en les synchronisant avec le casque VR: le composent **VR Rig**

##### Le VRRig de Robot Kyle

Il s’agit d’un script que j’ai écrit et qui va nous permettre de gérer les déplacement du personnage en synchronisation avec le casque VR.

Dans ce script je commence par créer une classe nommée **VRMap** qui va contenir toute les données dont j’ai besoins pour chacune des partie du corp que je veux contrôler.



C’est à dire:

* **vrTarget**

Le component **Transform** du casque VR, c’est a dire sa position et sa rotation

* **rigTarget**

Le composent Transform du Target que j’ai créé plus haut et qui, pour rappel, est l’espèce de “poignée” qui me permet de bouger chacun des trois membres de Robot Kyle.

* **trackingPositionOffset**

Il s’agit du décalage que l’on veut appliquer entre la position de l’élément VR (Manette ou casque) et le **Target** auquel on le relie.

Cela nous sera très utile pour le casque car si l’on positionnait la caméra principale à l’endroit exact ou est l’os de la tête de Robot Kyle nous serions à l’interieur de sa tête et la vision serait gainée par le mesh qui la compose.

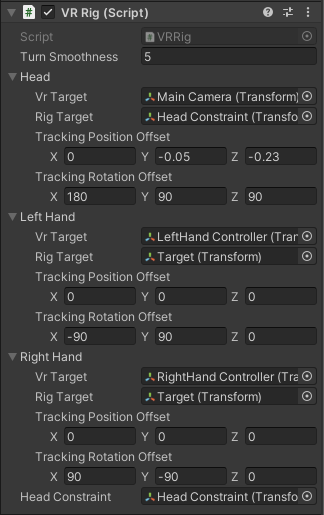
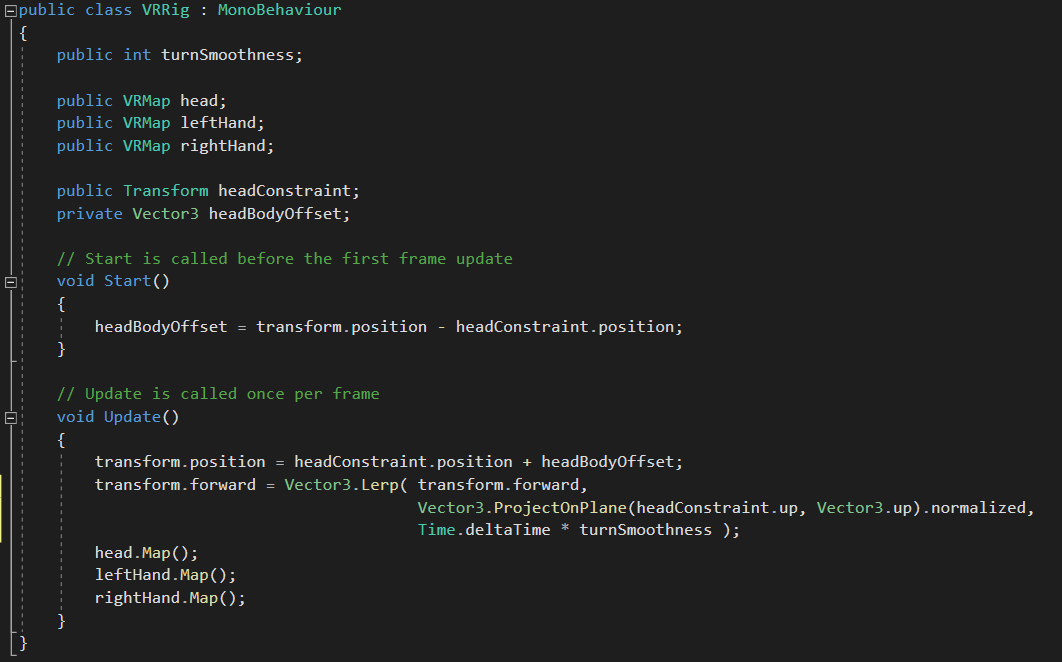
* **trackingRotationOffset**

Il s’agit du décalage de rotation que l’on veut appliquer entre la position de l’élément VR (Manette ou casque) et le **Target** auquel on le relie. Cela sera utile pour bien aligner les mains à la position des manettes.

L’application de la fonction Map() appliquera la position du **Target** de l’élément au périphérique VR correspondant.

Le **VRRig** en lui-même contiendra une variable **VRMap** pour chaque élément que l’on souhaite contrôler.

Voici une vue de VRRig depuis l’éditeur de code (à gauche) et depuis l’inspecteur Unity (à droite).



Afin de régler le petit soucis que nous avons vu plus haut, J’ai également besoin d’avoir accès aux coordonnées du **headConstraint** et calculer la distance qui sépare le **Target** de la tête et le **Gameobject** dans lequel on se trouve: **Robot Kyle**.

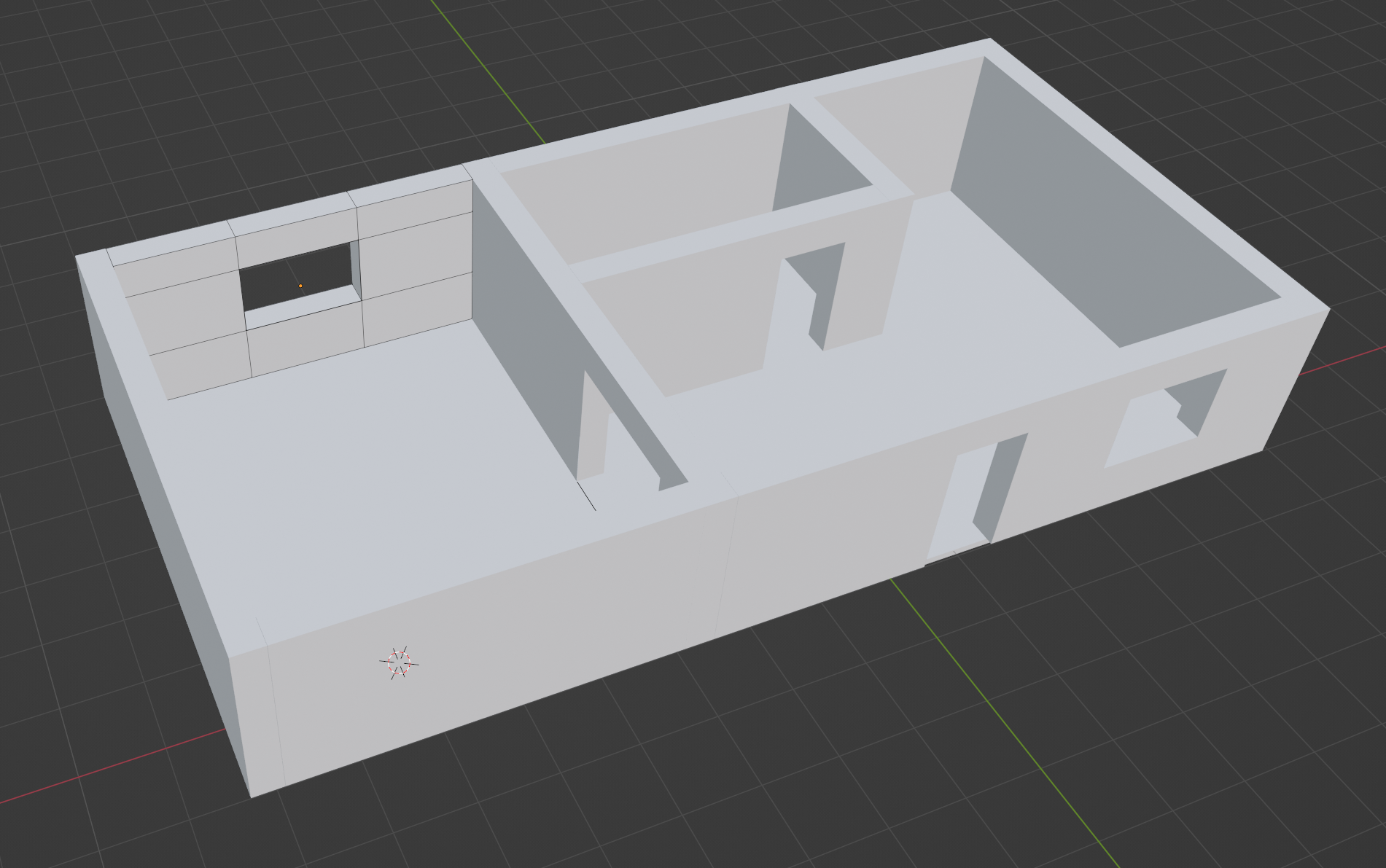
En effet, via la méthode **Update()**, on demande a Unity qu’à chaque frame il mette a jour la position du corps de Robot Kyle par rapport à la position du **Target** de la tête.

En suite on lui demande de mettre la tête elle-même à la bonne position et on fait de même pour les mains, via la fonction map de chaque composant. On remarque dans le component vue par l’inspecteur Unity que l’on a drag and drop tous les éléments demandé par le script de façon a ce qu’il ait toutes les références dont il a besoin.

#### Decor

Avant de clôturer le chapitre sur les mouvements, comme nous avons parlé de ce qu’il se passe quand on se cogne à un objet ou à un mur, je vais aborder ici brièvement les décors.

J’ai décidé de placer le **tag** “Decor” sur tous les GameObjects qui ne sont là que pour l’esthétique mais qui on tout de même la particularité de procéder chacun un BoxCollider de manière a pouvoir être détecter et générer une collision en cas de besoin.

Le premier élément de décors à avoir été ajouté fut le manoir. Je l’ai créé sur le programme **Blender** qui est l’outil de création utilisé pour générer des objets et personnages pour Unity. Ce manoir est le seul élément de décor à être présent dans les trois époques et à être placé précisément au même endroit dans chacune d’entre elle.

La seule différence réside dans les **Materials** (composant qui gérent la couleur et le rendu général d’un GameObjects) que l’on va appliquer à chaque mur afin que le décors varie selon les époques.

Les autres objets viennent de l’Asset-store de Unity, excepté bien entendu le **Chrono-Wc** dont nous parlerons plus loins.

Voila qui clôture ce chapitre sur la possibilité de bouger dans le jeu. Nous pouvons maintenant bouger dans le jeu et le corps de **Robot Kyle** reproduit exactement les mouvements de notre matériel VR. L’implémentation des mur et des décors, fonctionne car ils arrêtent la course du joueur.

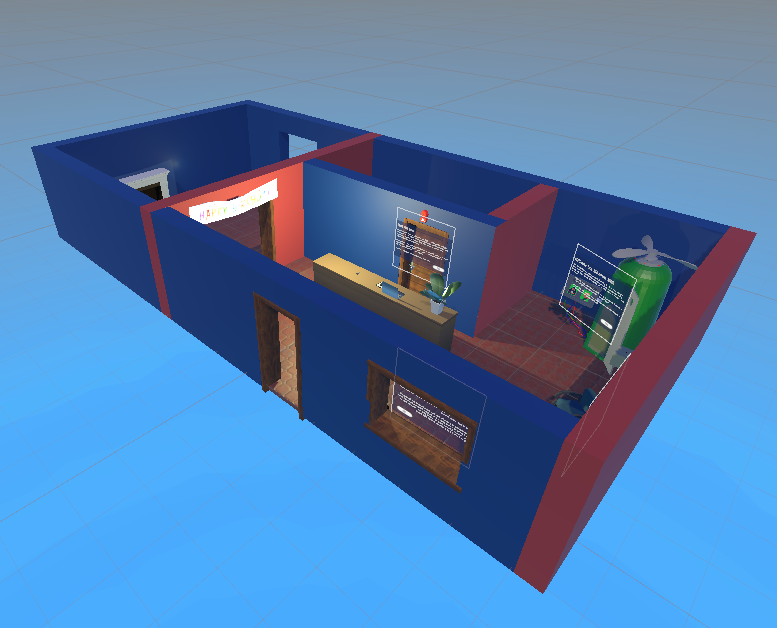
### Création des scènes

Avant d’aller plus loin, il est déjà possible de créer les différentes scènes dont nous allons avoir besoin. Nous savons que nous aurons besoin de quatre scènes :

* une pour l’écran de connexion où le joueur peut sélectionner la partie dans laquelle il désire entrer et le personnage qu’il veut incarner.
* une pour le passé
* une pour le présent

Et une pour le futur

Scène Connection
Une image contenant texte, équipement électronique

Description générée automatiquementUne image contenant texte, intérieur, plancher

Description générée automatiquement

La scène de connexion a son décor spécifique avec l’écran du chrono-WC qui sert d’interface. Pour les trois époques, on remarque qu’elles ont le même bâtiment de base et quelques éléments en commun mais que leurs couleurs et décorations varient selon les époques.

### Prendre et déposer un objet

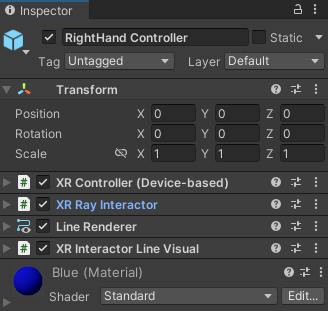
Afin de pouvoir prendre des objets en main il est nécessaire de faire quelques petits réglages sur le **XR Origin** et sur les GameObjects **LeftHandController** et **RightHandController** qu’il contient.

Nous allons ajouter au XR Origin un GameObject appelé **XR Interaction Manager** qui sera le script de base gérant toutes les interactions. Il contient un component du même nom et un **Input Action** qui fait le lien entre cet objets et les références vers le boutons des controllers.

#### HandsControllers

Passons maintenant à **LeftHandController** et **RightHandController.** Chacun d’entre eux contient déjà un component de base **XR Controller**. Qui sert à traquer la position et la rotation de chaque contrôleur et à référencer toute leur touches.

##### XR Ray Interactor

Nous allons appliquer sur chacun de deux GameObjects un nouveau component appelé XR **Ray Interactor** qui permettra aux manettes d’interagir avec tout les GameObject ayant un component de type **Interactable**. On peut y appliquer tout un tat de paramètre qui permettent de personnaliser l’expérience que l’on souhaite donner au joeur en faisant varier les effets, en appliquand des sons lors des interactions, etc..

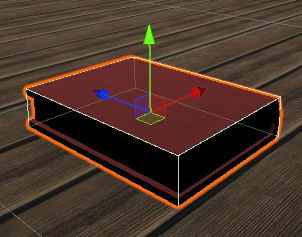
Les interactions se feront grâce à ce composent par l’intermédiaire de rayons lasers qui nous permettrons de pointer des objets, d’interagir avec eux, de les prendre, mais aussi d’interagir avec l’interface utilisateur.

Le **Ray Interactor** contient toute une série de paramètres qui nous permettent de régler le comportement des interactions mais qui n’ont pas beaucoup d’intérêt à ce stade.

Le **Line Renderer** et le **XR Interactor Line Visual** permettent de régler le design des rayons de lumière. Dans notre cas, nous commencerons par les mettre dans une couleur voyante et de les laisser assez fin.

A des fins esthétiques, nous les rendrons en suite transparents et plus épais et aiderons le joueur à savoir ce qui est pointé en appliquant une aura à tout objet sélectionnés.

#### GrapableObject

Nous pouvons maintenant ajouter aux scènes les objets que le joueur pourra prendre en main. Nous allons commencer par glisse un objet dans la scène en allant le chercher parmi les assets présents dans l’Asset Store.

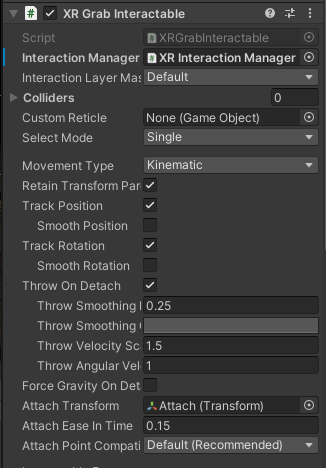
Prenons l’exemple d’un livre que nous voulons ajouter à la liste des objets que nous pourrons saisir.

Si nous nous contentons de l’ajouter et que nous entrons en **playMode**, le livre flottera dans les airs a l’endroit où nous l’avons mis et le joueur passera à travers.

Comme pour les murs et les éléments de décors, il nous faudra donc ajouter un **BoxCollider** au livre pour détecter les collisions et un **RigidBody** afin que l’objet soit solide, qu’il soit arrêté par tout **collider** qu’il rencontrera et qu’il soit soumis à la gravité.

Nous allons maintenant nous occuper du component qui vont nous permettre d’interagir avec les objets.

##### Le XR Grab interactable

Le premier d’entre eux est le **XR Grab Interactable** car il va permettre à tout **Interactor** de saisir un objet.

Nous allons devoir lui fournir pour cela une référence vers le XR Interaction Manager qui gère toutes les interactions.

Via les check-box Nous allons indiquer au component qu’il faudra traquer la position et la rotation de l’objet quand il sera dans la main du joueur.

On remarque aussi le paramètre **AttachTransform** qui est un paramètre qui va donner une position précise quand il sera dans la main du joueur. En effet, le livre pourrait apparaître dans la main du joueur dans une position ou il passerait à travers elle.

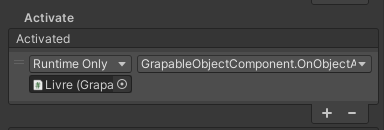
Pour éviter ça, nous allons donc créer pour chaque **GrappableObject** un GameObject enfant que nous allons appeler « **Attach »** et a qui nous allons donner une position et une rotation adéquat.

Ceci fait nous pouvons le fournir en paramètre au component **XR Grab Interactable.**

A ce stade nous avons donc maintenant un rayon lumineux émis par chaque main et qui nous permet d’interagir avec les différents éléments du monde. Nos mains sont en effet désormais des **Interactor** et nous permettent d’utiliser tout objet **Interactacle** dans le monde. En plus de prendre les objet grâce à un **Interactable** spécifique comme le **XR Grab Interctable**, nous verrons dans les chapitre suivant qu’il existe aussi d’autre interactions possible qui agrémentées d’un peu de programmation nous permettra de réaliser des actions très utiles.

### Obtenir des informations sur un objet

Maintenant que nous avons des **Interactors**, et des objets **Interactables**, nous pouvons faire en sorte que les objets aient d’autres fonctionnalités. J’ai voulu ici que les objets **Interactables** (Via le **XR Grab Interactable**) affichent une description de l’objet pointé lorsque l’utilisateur appuie sur la gâchette de la manette.

Si l’on va plus bas dans le composant **XR Grab Interactable** on voit qu’il est possible de faire en sorte qu’une fonction soit lancée lorsque l’objet est activé(Quand le joueur pointe l’objet ou le tient en main et qu’il appuie sur la gâchette situé près de son index). N’entrons pas dans trop de détail de fonctionnement mais notons tout de même que la fonction **Activate()** est définie comme telle dans le composant **XR Interactor** : la fonction **Activate()** de tout **Interactable** est déclenchée par un bouton défini.

Pour ce faire, j’ai donc créé un component **GrappableObjectComponent** qui renferme les données de l’objet et la fonction dont j’ai besoin pour afficher un message à l’utilisateur quand j’active un objet.

Les étapes à suivre seront donc :

* Créer une Gameobject **UserInterface** qui affiche les messages à l’utilisateur
* Lui ajouter un script **UserInterface** qui lance l’affichage de façon variable selon l’objet concerné.
* Créer le script que **GrappableObjectComponent** qui sera ajouté à chaque **GrapableObject** et qui contiendra toutes les données de l’objet ainsi que la fonction qui pointe vers le **UserInterface**

#### GamebObjcet UserInterface

Je prévois que dans l’avenir, j’aurais besoin de plusieurs types d’interfaces utilisateur.

Une image contenant texte

Description générée automatiquementJe crée donc un GameObject vide que je nomme **UserInterface** et qui contiendra les différentes interfaces dont j’aurais besoin. Je lui ajoute un script (donc un component) **UserInterface** qui va gérer l’ensemble et sur lequel nous allons revenir plus bas.

Dans Unity, pour créer une interface utilisateur, il faut ajouter un **Canevas** qui sera en gros le « calque sur lequel on va écrire ». Nous l’appellerons ici **MainScreen** puisqu’il sera configuré de manière à couvrir et suivre en permanence l’entièreté du champ de vision du joueur.

Une image contenant texte

Description générée automatiquementC’est à l’intérieur que nous allons placer le premier élément dont nous aurons besoin : Le gameObject **DialAndDescription**. Ce dernier sera une boite de dialogue fixée en bas au centre de l’écran et contiendra le nom de l’objet en gras et sa description.

Notons que cet objet s’appelle Dial**AND**description puisque nous utiliserons ce même pop-up pour les dialogues avec les PNJs et les descriptions d’objets.

#### Le component UserInterface

Comme annoncé ci-dessus ce script va contenir plusieurs références vers les objets qu’il doit gérer et des méthodes destinée à être appelé afin de réaliser telle ou telle tâche. Voici a quoi ressemble le script :

Une image contenant texte

Description générée automatiquement

Nous avons besoin dans ce script d’une référence vers le GameObject **dialAndDescription** afin de pouvoir le faire apparaitre et disparaitre à notre guise et d’une référence vers chaque élément texte de cette interface afin de pouvoir les adapter à chaque objet ou PNJ.

Une image contenant texte

Description générée automatiquement Pour rappel, comme tous ces paramètres sont en « public », ils sont assignables dans Unity via un drag and drop dans l’inspecteur.

La méthode **displayDescription** reçoit deux strings représentant respectivement le titre et la description à afficher et se charger de les passer à une Coroutine.



« *Une coroutine permet de répartir les tâches sur plusieurs frames. Dans Unity, une coroutine est une méthode qui peut interrompre l'exécution et rendre le contrôle à Unity, mais reprendre ensuite là où elle s'est arrêtée sur l'image suivante.* » *(Unity Technologies, 2020)*

Concrètement, nous l’utilisons ici pour déclencher un évènement durant un certain nombre de secondes. Unity va exécuter les trois premières lignes de la coroutine, rendre le contrôle à la méthode appelante durant 10 secondes et le reprendre pour exécuter le reste.

La Coroutine change le texte du titre et le texte de la description puis affiche le gameObject **DialAndDescription.** Elle attend 10 secondes puis masque cet élément.

**<<Parler du do’nt destroy on load>>**

#### GrappableObjectComponent

Ce component va être ajouté à tous les **GrapableObject** afin de leur donner des propriétés et des fonctions spécifiques à tous ces objets.

Une image contenant texte

Description générée automatiquement

Il contient une référence vers le script **UserInterface** que nous avons créé avant ainsi qu’une description de l’object. Une variable Name n’est pas utile, nous nous servirons du nom du **GameObject** lui-même.

Nous retrouvons ici la fonction **OnObjectActive()** que nous avons associé tout à l’heure à l’**Activate** de **notre XR Grab Interactable**.

Petit résumé des évènements une fois que tous ces éléments sont implémentés :

* Quand le joueur clic sur un **GrapableObject**, la paramètre **Activate** de cet objet active la fonction **OnObjectActive()** de son **grappableObjectComponent** qui demande à l’interface utilisateur d’afficher les données de l’objet qu’on lui passe, c’est à dire le nom du **GrapableObject** et une description de celui-ci.
* Quand l’interface reçoit ces données, elle affiche une petite fenêtre avec le nom de l’objet et sa description pendant 10 secondes.

### Mettre un GrapableObject en surbrillance au survol

Une image contenant texte

Description générée automatiquementIl s’agit ici d’un simple petit ajustement cosmétique qui va consister à faciliter l’utilisation des objets. Afin que le joueur sache que l’objet est sélectionné, on aimerait que ce dernier s’illumine au survol. D’une part cela indique au joueur que des interactions sont possibles avec ce dernier et d’autre part, si l’on veut rendre, pour des questions d’esthétique, les rayons de pointages invisibles cela rendra le pointage d’objet plus facile.

Pour ce faire nous allons simplement ajouter à chaque **GrapableObject** un component **XRTintInteractableVisual** qui fera automatiquement ce que nous désirons ici.

### Création des préfabs des GrapableObjects

A présent que nous avons des **GrapableObjects** complets qui peuvent être saisie et qui fournissent une description quand on interagit avec eux, nous pouvons en faire des préfabs.

Une image contenant texte

Description générée automatiquementPour rappel, les préfabs sont des espèces de « modèle » d’un objet. L’idée est que d’une part que si l’on modifie ce modèle, on peut répercuter les changements sur tout les **GameObjects** basé sur la préfab qui sont dans le jeu ; et d’autre part de pouvoir à tout moment instancier un **GameObject** basé sur cette préfab à tout moment.

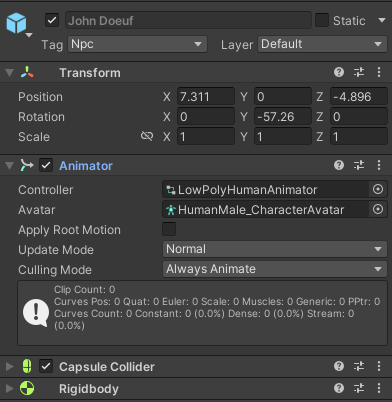
Nous verrons qu’avoir des préfabs de nos **GrapableObject** va être très utile pour la suite. Nous allons simplement drag and drop chaque **GrapableObject** que nous avons créés dans un dossier dédié de notre projet Afin d’avoir notre collection de préfabs.

### Obtenir des informations sur un PNJ

L’opération va être extrêmement semblable à ce que nous avons fait ci-dessus. En effet, nous allons réutiliser le système que nous avons créé plus haut pour les GrapableObjects et l’appliquer aux PNJ.

Nous allons donc commencer par ajouter un PNJ à notre jeu.

Une image contenant bleu

Description générée automatiquementComme pour **Robot Kyle**, nous allons aller le chercher dans l’Asset store mais nous veillerons cette fois à en prendre un qui contienne des animations.

En effet, nous aimerions pouvoir faire en sorte qu’il ait des réactions quand on lui donnera un objet, qu’il soit capable de bouger ou tout simplement qu’il ait l’air plus réel.

Cela se fera par un composent **Animator** qu’il contient et qui servira à lancer des animations mais nous verrons ça plus loin.

Ce que nous allons faire en un premier temps c’est de lui ajouter un **CapsuleCollider** de manière à pouvoir interagir avec lui et détecter les collisions ainsi qu’un RigidBody de manière à ce qu’il ait des caractéristiques physiques et soit soumis à la gravité.

#### XR SimpleInteractable component

Maintenant il nous faut la possibilité d’intéragir avec lui. Avec les grapableObjects, nous avions fait en sorte de pouvoir le faire en passant pas le composant XR GrabInteractable, qui servait avant tou à prendre les objets et possédait une option pour interagir d’autres manières. Dans le cas d’un PNJ il est évident que ne pouvons pas faire la même chose étant donné que nous ne désirons évidemment pas le prendre en main! À la place, nous allons lui ajouter un autre composant qui en gros n’offre que les possibilités d’interaction simples le **XRSimpleInterctable**.

Comme dans son grand frère que nous avons utilisé pour les **GrapableObjects**, nous retrouvons le **Activate** qui nous permet de déclencher une réaction à un clic du joueur avec la gâchette de son contrôleur.

Ainsi allons-nous donc créer un script pour chaque PNJ qui contiendra des informations sur ce dernier ainsi que la fonction permettant d’appeler le UserInferface, exactement comme nous l’avions fait pour les GrapableObjects.

Voici donc le script Pnj.cs :

Une image contenant texte, capture d’écran, écran, argent

Description générée automatiquement

Comme pour les objets, nous avons une description et une fonction **OnObjectActive()** déclenchée au clic du joueur. La procédure suivit est donc exactement la même : l’interface utilisateur reçoit le nom de l’objet et sa description et l’affiche.

### Sauvegarder une partie

Maintenant que nous avons nos **grapableObjets** et nos **PNJ**, il est temps de mettre au point la sauvegarde d’une partie. C’est une grosse partie du travail qui va nécessiter l’implémentation de beaucoup d’élément. Afin m’y retrouver, je vais commencer par essayer de faire le scénario de ce qui devra se passer lors d’une action de sauvegarde telle que je vais l’implémenter:

* Le joueur appuie sur une touche menu
* Le menu apparait sur son écran par l’intermédiaire de l’interface créée plus tot, lui donnant le choix de sauvegarder ou quitter la partie
* Le joueur clic sur sauvegarder
* Une fonctions SaveGame() présente dans un script nommé GameManager, qui renfèrme toutes les fonctions d’ordre générale du jeu, est lancée.
* La fonction récolte toute les GrappableObjects de la scène actuelle et déclenche la fonction SaveEra() du NetworkManager (qui gère pour sa part toute les communications avec le serveur de jeu), en lui envoyant les données des objets à sauver.
* Le NetworkManager se sert d’un socket ouvert en début de partie afin d’envoyer la requête au serveur.
* Le serveur reçoit la requête, sauve chaque objet en base de données.

#### Création des Models

Avant de commencer quoi que ce soit, nous aurons besoin de Models. Ces derniers serviront à faire le lien entre ce qui sera sauvé en base de données et les objects Unity. Nous allons donc créer des clases C# correspondant aux tables de la basse de données en restant fidèle au diagramme de classe de l’analyse de base (Page 6).

Nous créerons ici les Classes nécessaires à la sauvegarde d’une partie : **Game**, **Era**, **GrapapbleObject** et **Pnj**. Le diagramme de classe figurant déjà en page 6, nous ne reviendrons pas sur les propriétés et la structure de chaque classe. Mais attardons-nous tout de même sur l’une d’entre elles afin d’observer la manière dont elles sont créées.

Une image contenant texte

Description générée automatiquement

J’ai choisi volontairement la classe Era car elle contient une petite particularité. Contrairement à ces semblables, elle contient un paramètre qui n’est pas en base de donnée : **isFree**.

En effet, cette donnée représente la situation en temps réel de la disponibilité de l’**Era**. Il faut que le jeu soit tenu au courant en temps réel du fait qu’un joueur est connecté ou non à l’**Era** à laquelle on souhaite se connecter.

**Ex:** Si le joueur 1 est connecté au « présent », le joueur 2 ne doit pas pouvoir se connecté à cette **Era** en arrivant sur le jeu.

Nous verrons plus loin comment cette variable est mise à jour en temps réel.

Nous remarquons également que dans celle classe Model quand dans toute les autres, j’ai placé une balise **[Serializable].** Cette dernière sert à indiquer au compilateur que cette classe peut être « serialisée », et pourra donc être instanciée par le convertisseur que nous emploierons plus loin pour traiter les objets **JSON** reçus depuis le Backend.

Nous avons maintenant nos Models et pouvons revenir su la création proprement dite de la sauvegarde des données.

#### Création du Menu

Nous allons ici nous servir du GameObject **UserInterface** que nous avons créé plus haut (page 26) et lui ajouter un nouveau canevas qui s’appellera **Menu**.

Une image contenant texte

Description générée automatiquementNous le configurons pour qu’il suive la caméra et prenne tout l’écran comme nous l’avions fait pour le **MainScreen** auparavant. Ce canevas contiendra un background, un titre et deux boutons : « Sauvegarder » et « quitter ».

Nous allons nous occuper du bouton Sauvegarder et laisserons le bouton quitter de coté pour le moment. Nous allons maintenant demander au bouton Save de pointer vers une fonctions **SaveGame()** qui sera présente dans le **GameManager**.

#### Création du GameManager

Comme nous l’avons vu plus haut, le **GameManager** est un contrôleur général du jeu et qui contient un script reprenant toutes les méthodes générales du jeu (sauver la partie, quitter, etc.).

Nous allons créer ce **GameManager** dans la scène connexion. En effet, comme nous l’avons vu plus haut dans le chapitre « Cycle de vie du jeu » (Page 15), cette scène est celle par laquelle le joueur entre dans le jeu et nous aimerions qu’a chaque nouvelle partie, un nouveau GameManager soit instancié et conservé tout au long d’une partie. Comme nous l’avons fait pour l’interface.

Nous allons donc créer un **Gameobject** dans cette scène appelé **GameManager** et qui va contenir un script du même nom qui va s’enrichir au fur et à mesure de l’implémentation de nouveaux paramètres indispensables et de nouvelles fonctions d’ordre générale.

##### Les propriétés

Voici à quoi va ressembler en un premier temps le script **GameManager**.

Une image contenant texte

Description générée automatiquementNous voudrions qu’il sache dans quelle partie (Game) on se trouve, quelles Eras (époques) contient cette partie et quelle est l’époque active (Passé, présent ou futur). Nous ajoutons donc les variables activeGame et activeEra.

Pour se faire nous aurons besoin des Models que nous avons créé juste avant et qui contiendront toutes les données que nous irons chercher en DB.

Nous souhaitons aussi que le **GameManager** retienne à tout moment quels **grapableObjets** et quels **PNJs** sont dans la scène. Nous lui ajoutons donc deux listes contenant ces informations qui sont elles aussi des liste d’objets que nous créerons juste après.

Enfin, il faut que notre **GameManager** ait accès au **NetworkManager** afin de pouvoir y faire appel quand on le désire. En effet, à des fins de bonne structure, le **GameManager** sera le seul a communiquer avec le **NetworkManage.**

##### DontDestroyOnLoad

A présent, nous allons faire en sorte que le GameObject auquel ce script est rataché ne soit pas détruit lors du passage d’une scène à l’autre.

Pour rappel, quand Unity charger une nouvelles scène, il détruit tout les objets qu’il a créé dans la scène actuelle. Dans notre cas, il nous faut éviter ceci car nous souhaitons garder les donnée recueillies par l’écran de connections (à savoir **activeGame** et **activeEra**) au passage de la scène connexion à la scène correspondant à l’époque choisie.

Une image contenant texte

Description générée automatiquementNous allons donc dire à Unity à l’initialisation de l’objet (dans la méthode **Start()** )que nous souhaitons que l’objet contenant ce script ne soit pas détruit à cette occasion.

##### Fonction SaveGame()

Maintenant que le **GameManager** est en place il ne nous manque plus qu’à créer la méthode qui sera appelée par l’interface pour sauver un jeu : **SaveGame().**

Une image contenant texte, horloge, périphérique

Description générée automatiquement

Commençons avant tout par récupérer dans la scène active tous les GameObjects qui ont le tag « grapableObject » et stockons-les dans une variable **gosGrabObj** (GameObject Grapable Objetcts).

Je crée ensuite une liste vide Qu’il me faudra remplir avec les models correspondants aux **GameObjects** que je viens de récupérer.

Une image contenant texte

Description générée automatiquement

Je passe en revue chaque **GameObject** et stocke ses propriétés dans son équivalent « Model » que j’ajoute à la liste des **GrapableObjects.**

Nous avons maintenant une liste de **GrapableObject**, il ne nous manque plus que la liste des PJNs.

Je répète donc rigoureusement la même opération pour les **GameObject** pourtant le tag « Npc » et me voilà avec les deux listes de donnée que je dois sauver en base de données.

Une image contenant texte, orange

Description générée automatiquementJe stocke maintenant ces deux listes dans l’Era active et transmet cette dernière au NetworkManager afin qu’il la transmette au Backend pour sauvegarde en DB.

### Donner un objet à un personnage

#### Envoyer un objet à un héros

#### Donner un objet à un personnage non-joueur(PNJ)

### Donner un objet à un pnj

#### Réactions des pnj

### Charger une partie

### Créer une partie

### Supprimer une partie

**A placer quelque part**

Quand on envoie un objet par les chronos-Wc, il est automatiquement effacé de l’époque actuelle en base de donnée et est ajouté à l’époque de destination, même si le joueur n’a pas fait de sauvegarde. Ce “sauvetage de force” est en effet inévitable afin d’éviter tout doublon ou toute disparition d’objets.

Déroulement du projet.

- apprentissage de Unity

A duré longtemps

Suivi des cours de Unity learn: [*https://learn.unity.com/*](https://learn.unity.com/) deux mois.

Problèmes rencontrés

- les sockets

- les murs

- le build

- lenteur de l’ordi

- soucis de visualisation du fonctionnement général avant implémentation des GOs

- Nécessité d’apprendre blender

- construction de Bernard abandonnée

-> GravitySketch problème de rendu dans Blender

- synchronisation du mouvement de robot-kyle avec les mouvements.

->apprentissage des rigs

- Problème destruction du Network Manager

-> problème pour Sauver/Quitter, plantage occasionnel

- Télécommande qui ne fonctionne plus, problème de référencement

- Plantage serveur au démarrage alors que le Front est démarré

- Problème de persistance des données du Game Manager

-> Don’t destroy onload

-

Parler de la fonction Update d’un gameobjet().

Ajouter Isfree pour les Era et IsNex pour les games

Bibliographie

Unity Learn

Bibli unity