EPFC
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

1 Introduction

1.1 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 créés : la ***Tentacule Pourpre***. Cette dernière serait devenue folle, à la suite de l'absorption d'un produit chimique, et aurait pour 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'entrer 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 série 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 utiliser 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'utilisation.

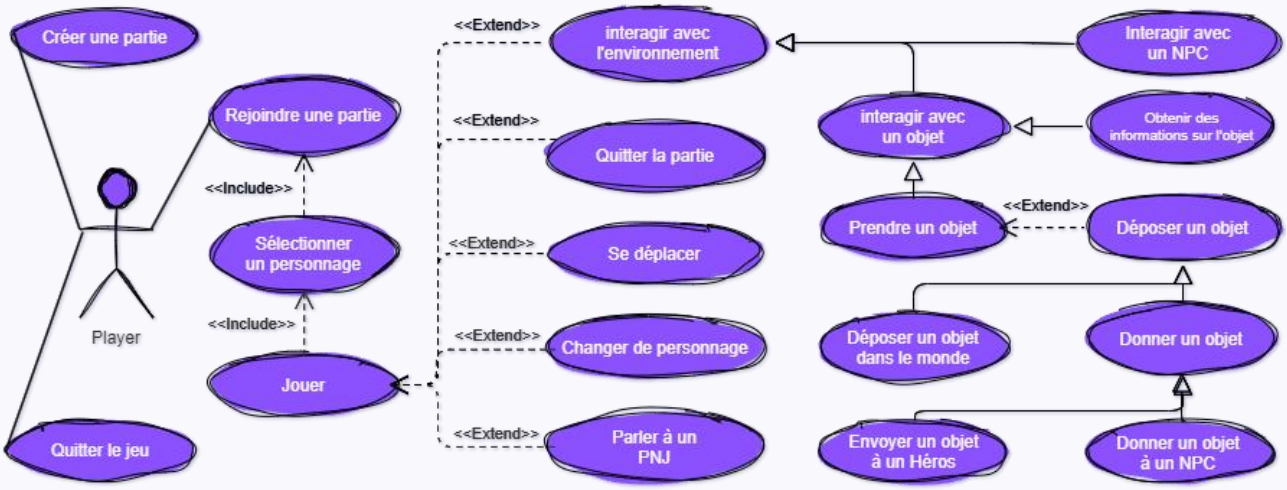
1.2 Fonctionnalités

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

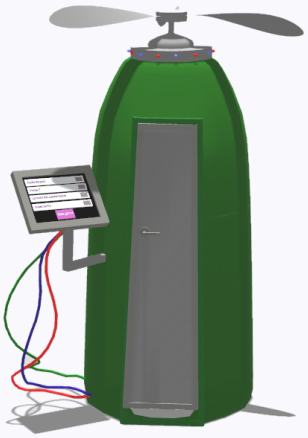
Lorsqu'un premier joueur entre dans une partie, il peut incarner l'un des trois personnages disponibles : **Bernard**, **Huagui** ou **Laverne**. Il a la possibilité de passer d'un personnage à l'autre à tout moment en revenant au menu principal. Si un deuxième joueur rejoint la partie, il peut choisir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



Quand il lance le jeu, le joueur arrive dans le menu principal. Ce dernier est représenté par un Chrono-WC et son panneau de contrôle. Le joueur peut à cet endroit : *Quitter le 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 un bouton du panneau de contrôle du Chrono-WC afin de quitter le jeu. Cette action met fin au programme.

✓ Créer une partie

Sur le panneau de contrôle s'affiche une liste de toutes les parties disponibles sur le serveur. Un bouton permet de générer une nouvelle partie. S'il clique sur ce dernier, une nouvelle partie sera créée sur le serveur avec un nom auto-généré (voir page 8) 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 Personnage encore disponibles il souhaite incarner.

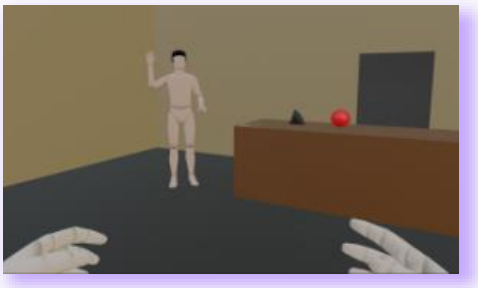
✓ Sélectionner un Personnage

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 époques de la partie. À tout moment, il est susceptible de voir si un joueur rejoint la partie, via l'affichage d'un pop-up 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. Le décor change en fonction de l'époque choisie.



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

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

✓ 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 changements qu'il a opéré dans cette époque.
Comme nous sommes dans le cadre d'un jeu multijoueur, il aurait été cohérent de penser aux sauvegardes automatiques. Toutefois, d'une part, je voulais laisser au joueur la possibilité de ne pas sauver sa progression s'il s'apercevait qu'un de ses choix n'était pas judicieux et d'autre part, le nombre de demandes de sauvegardes envoyées au serveur aurait été si important, qu'il aurait pu nuire gravement aux performances du jeu.

Ex: Si un joueur déplace un objet et qu'il désire 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 à jour de la base de données pour la position et la rotation de chaque objet. Cette méthode se déclenchant un grand nombre de fois par seconde, le serveur recevrait un important nombre de requêtes à exécuter via le **websocket**. Chacune d'entre elle déclencherait par ailleurs 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 sauvegardes via une méthode qui ne serait appeler que dans certaines circonstances (quand l'objet touche une surface, par exemple) mais cela compliquerait grandement l'implémentation et la gestion des sauvegardes, entraînerait quoi qu'il en soit une multiplication des requêtes et n'apporterait pas de réel bénéfice au jeu.

✓ 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.

✓ Se déplacer

Le joueur peut faire bouger le personnage soit en maniant le *Stick Analogique* de la manette de gauche, soit en se déplaçant lui-même dans la réalité.

✓ Changer de personnage

En quittant la partie en cours, le joueur se retrouve à nouveau dans le menu principal où il a donc l'occasion de retourner dans sa partie en incarnant un autre personnage.

✓ 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'interagir avec lui.

- **Interagir avec un NPC**

  Lorsqu'un NPC est en surbrillance, 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, sous forme d'une fenêtre de dialogue.

- **Interagir avec un objet**

  Les interactions possibles avec un objet 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 informations 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 Personnage

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.

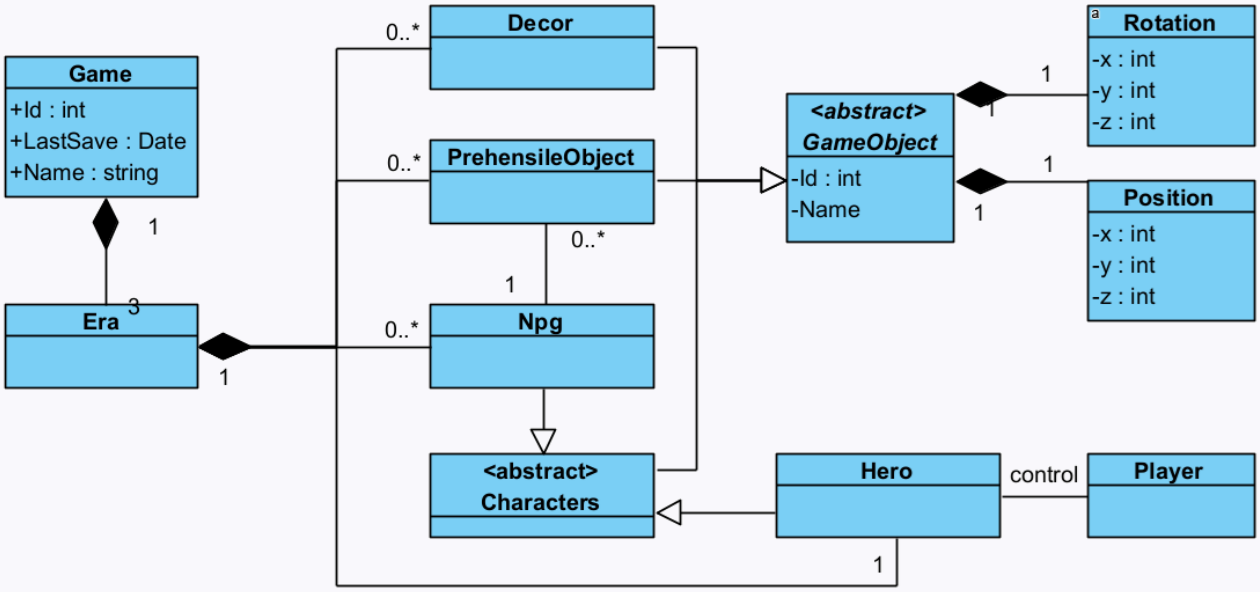
Une fois cette sélection effectuée par simple pression avec sa main virtuelle, le joueur peut jeter l'objet qu'il souhaite envoyer dans la cuvette du WC et l'objet est envoyé.

Le cas échéant, le joueur contrôlant le héros de destination pourra alors voire instantanément l'objet envoyé jaillir du ***Chrono-WC***.

✓ Donner un objet à un NPC

Quand un joueur tient un objet en main et l'approche d'un NPC, 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 NPC accepte l'objet, ce dernier disparaît de la main et est donné au NPC. Si la réaction du NPC indique un refus, l'objet retourne alors dans l'inventaire du joueur.

Diagramme de classe théorique



✓ 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 le 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'actions devront être réaliser, comme le message de bienvenue, un didacticiel sur la façon de jouer, etc.

- name
Ce String 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'une 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 entraîne, en cascade, la destruction de ces trois époques. La collection d'époques a toujours une longueur de 3 et est initialisée à l'instanciation de chaque partie. On ne peut ni ajouter, ni retirer d'époque à 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 présente à la base dans chaque époque mais qui peut passer de l'une à l'autre.

✓ GameObject

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

- **Name**
- **Rotation**
  Composé de trois Integer : x, y et z, compris entre 0 et 360. Ils présentent le nombre de degrés de la rotation qu'a subi 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 statiques du jeu tels que les murs, les portes ou les meubles. Ils ne peuvent pas être déplacé par le joueur.

✓ GrapableObject

À l'inverse, ces derniers peuvent être déplacé et tenu par les personnages.

✓ Npc

Ils sont les personnages non-joueurs (Non Player Characters) 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 ses objets aux personnages 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 Npc.

✓ Personnage

Les Personnage sont semblables aux NPC mais ils sont liés aux joueurs qui les contrôlent.

1.3 Architecture

Le choix du multijoueur a été fait afin de pouvoir mettre en place une architecture backend - frontend. Le tout utilisera les sockets pour toutes les communications en temps réel durant une session de jeu. La sauvegarde des données entre les sessions nécessitera l'utilisation d'une Db.

Choix des technologies

✓ Frontend : le jeu

Ce jeu sera développé sur **Unity** qui utilise le language C# pour les écritures de scripts.

L'utilisation de Unity est certainement ce qui nécessitera le plus d'apprentissage. En effet, comme nous le verrons plus loin (voire page 14), **Unity** a sa propre achitecture, un fonctionnement bien à lui et sort donc du contexte de la programmation traditionnelle telle qu'apprise à l'école.

Unity est un moteur 3D de création de jeux / d'applications. Il n'est pas un modélisateur d'objets 3D (comme Blender), mais fonctionne avec une scène 3D gérant lumières, shaders et caméras.

Il permet l'insertion de code en C#. Des plug-ins d'aide à la création sont intégrés, comme un générateur de terrain, de végétation, ou encore des packs de Préfabs facilitant le contrôle du joueur, des caméras etc...

Le principe est d'inclure des éléments graphiques (nommés GameObject) auxquels on applique des Scripts ou Components qui vont régir les comportements du jeu.

Nous reviendrons plus en détail sur le fonctionnement de Unity dans le chapitre « *Apprentissage de la structure et de la programmation dans Unity* ».

✓ 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.

Je vais vous présenter ici un aperçu des technologies utilisées pour le Backend et reviendrais plus en détail sur leur utilisation en cours d'implémentation.

- **NodeJs**

  Avant, JavaScript était utilisé principalement pour les scripts côté client. Node.js, est un environnement d'exécution comprenant tout ce qui est nécessaire pour exécuter un programme écrit en JavaScript du coté Backend.

  Il fonctionne avec le moteur d'exécution JavaScript V8 et utilise une architecture d'E / S non bloquante et pilotée par les événements, ce qui le rend efficace et adapté aux applications en temps réel.

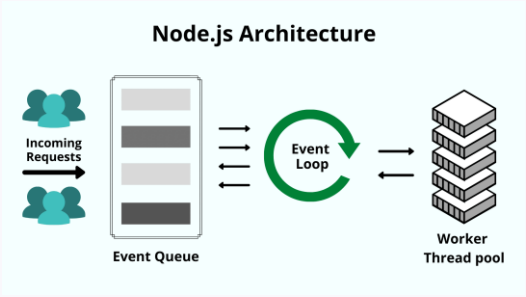


Image extraite de l'article "Architecture de Node.js" sur kinsta.com

L'efficacité de node.js provient de son architecture. Contrairement aux autres plates-formes web traditionnelles telles que *.Net* qui utilisent en général un système de **multi-threading**, Node utilise principalement **un seul thread**.

Pour l'expliquer de manière simplifiée, à chaque fois qu'une requête arrive elle entre dans un **EventLoop** qui n'utilise qu'un seul des threads disponibles.

Si la requête est **non-bloquante** (ne nécessitant qu'un traitement simple) elle est traitée immédiatement.

Si elle est **bloquante** (nécessitant un appel à une API externe, une consultation de base de données, etc.) elle est envoyée vers le **Worker Thread Pool** qui dédie alors un de ces Thread au traitement de la requête avant de la renvoyer dans le **Event Loop**.

C'est pour son efficacité et sa rapidité dans le traitement des requêtes que j'ai choisi **Node.js** pour implémenter le serveur de jeu. En effet, l'utilisation va nécessiter un très grand nombre de requêtes, les Sockets échangeant en permanence des données entre Backend et Frontend.

- **Les webSocket**

  Websocket est une technologie qui permet d'ouvrir de façon **bidirectionnelle** un canal de communication entre un client et un serveur. Cette technologie repose sur le protocole **TCP** qui s'occupe du transport des données et est destinée à des transport de messages courts entre le client et le server.

  Les requêtes Html sont basées sur un système se requête/réponse et sont très verbeuses et plus lourdes notamment à cause de leurs en-têtes qui contiennent un certain nombre de données.

  Les WebSockets pour leur part sont des canaux ouverts en permanence et sont plus courts et plus performants pour une communication continue entre serveur et client. Ils requièrent moins de bande passante, ont une latence réduite et permettent une communication en quasi-temps réel.

- **SocketIO**

  SocketIO est basé sur les web sockets. C'est une bibliothèque qui permet une communication à faible latence, **bidirectionnelle** et basée sur les **événements** entre un client et un serveur. Il est construit sur le protocole WebSocket mais offre des fonctionnalités et garanties supplémentaires telles que la reconnexion automatique.

Le site officiel nous précise d'ailleurs que :

« *Bien que Socket.IO utilise effectivement WebSocket pour le transport lorsque cela est possible, il ajoute des métadonnées supplémentaires à chaque paquet. C'est pourquoi un client WebSocket ne pourra pas se connecter avec succès à un serveur Socket.IO, et un client Socket.IO ne pourra pas non plus se connecter à un serveur WebSocket ordinaire.* » (socket.io, 2022)

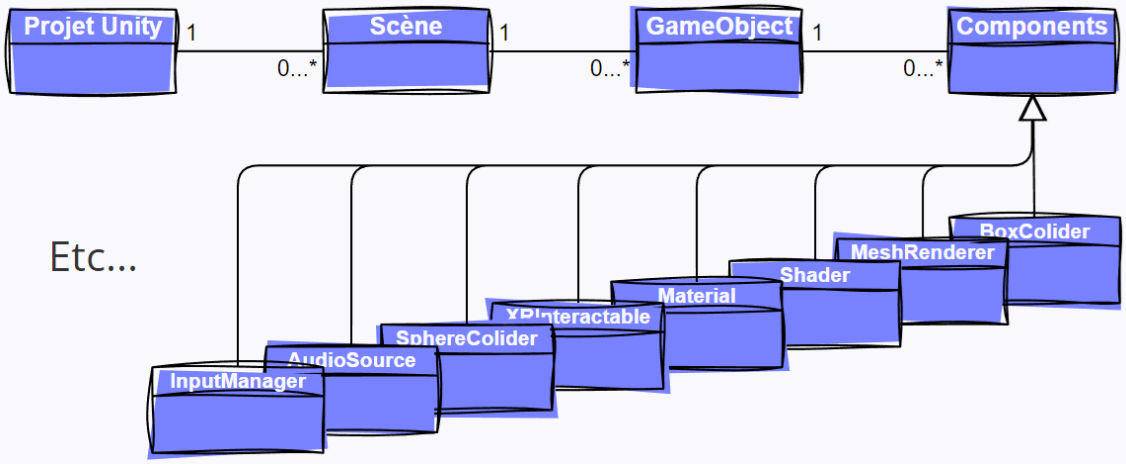
✓ Base de données

Pour la base de données, je choisis une technologie enseignée à l'école et sur laquelle il me semble inutile de m'étendre tant elle est connue et commune : le SQL de **MariaDB.**

Comme nous le faisions en cours, j'ai choisi d'utiliser un server **Xampp** comme serveur de base de données de développement.

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 fenêtre 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) .

Chaque scène est composée d'un nombre illimité de *GameObjects*.

✓ Les GameObjects

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

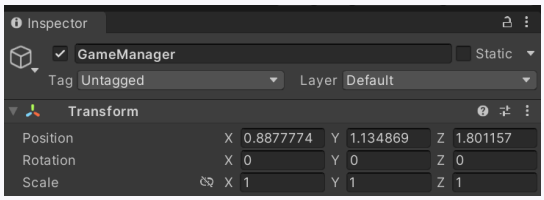
Partant de ce principe, on pourrait comparer un *GameObject* à une Vue dans le model MVC. Ce n'est toutefois pas rigoureusement exact. Car si un *GameObject* peut avoir un rendu visuel (un personnage, une maison, un monstre, un objet sur la table, etc.) ce n'est pas nécessairement le cas. Certains d'entre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

- **Le GameManager**

  Le meilleur exemple est sans aucun doute le GameManager, que 9 développeurs Unity sur 10 utilisent comme contrôleur général du jeu et qui contient, en résumé, un script reprenant toutes les méthodes générales 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 en page 16) mais n'est absolument pas visuel. Si malgré tout, on voudrait continuer la comparaison avec le model MVC, ce *GameObject* devrait plutôt être associé à un contrôleur. Ce serait aussi le cas d'autres *GameObjects* comme le NetWorkManager qui s'occupe de toute la communication réseau.

- **Les tags**



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

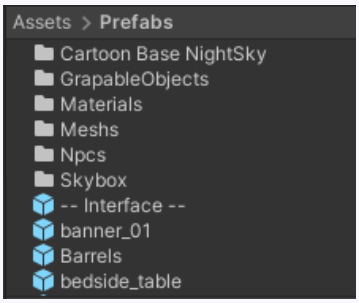
  Dans ces propriétés, on retrouve un combo-box « **Tag** ». Les tags permettent de distinguer les GameObjects entre eux et de faire des sortes de catégories. Ces catégories nous permettent d'appliquer certains effets sur des GameObjects spécifiques comme nous le verrons dans le chapitre suivant.

- **Active**

  La check-box présente dans l'inspecteur à côté du nom du GameObject permet d'activer ou de désactiver le GameObject. La désactivation d'un GameObject désactive chaque composant, y-compris les rendus, les collisionneurs, les corps rigides et les scripts attachés.

- **Les Préfabs**

  Comme nous l'avions vu plus haut, les **GameObjects** sont les éléments ajoutés à une scène et qui sont donc instanciés dès le chargement de cette dernière. Nous aimerions toutefois avoir des « models » de ces GameObjects afin de pouvoir les instancier quand nous le souhaitons en cours de route.



  C'est là que les Préfabs interviennent. Elles ne seront pas stockées directement dans une scène mais dans les dossiers du projet.

  Ces espèces de « models » peuvent être drag and drop dans la représentation visuelle de la scène afin de générer un GameObject basé sur son modèle ou être instanciées en ligne de code au moment de notre choix, comme nous le verrons en cours de développement.

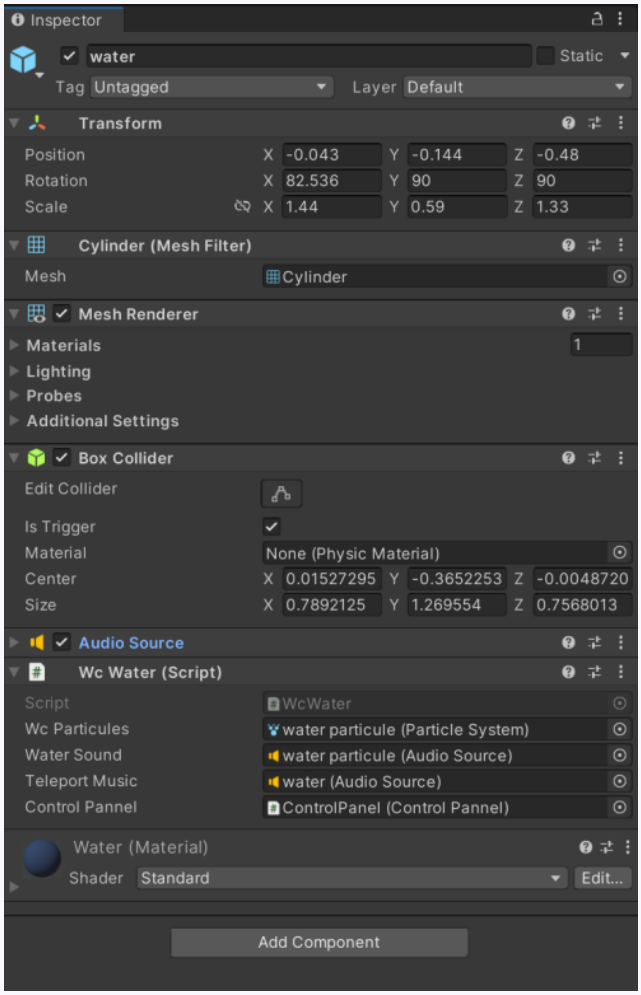
✓ Les Components

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 tous 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 nécessaires en son sein. Ce cube est une espèce de préfabriqué (nommé ***préfab***) proposé par Unity qui comporte déjà des éléments nécessaires à son affichage. Mais abstraction faite de l'utilisation de ce type de ***préfab***, c'est en règle générale 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 faire vivre 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 components suivants :

- le *Transform*, commun à tout GameObject ;
- des meshs pour le rendu visuel ;
- un box collider qui sert à détecter la collision de l'eau avec un GameObject ;
- une source audio qui permet d'y associer un 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 tous les comportements appropriés quand le Collider lui signale la détection d'un objet grapable.

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

- Les scripts en détails

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

```
public class WcWater : MonoBehaviour
{
    [Tooltip("Effet de particule à lancer lors de la collision avec un objet grapable")]
    public ParticleSystem WcParticules;

    [Tooltip("Premier effet sonore à lanser lors de collision avec un objet grapable")]
    public AudioSource waterSound;

    [Tooltip("Second effet sonore à lanser lors de collision avec un objet grapable")]
    public AudioSource teleportMusic;

    [Tooltip("Panneau de controle gérant l'envoie lors de collision avec un objet grapable ")]
    public ControlPannel controlPannel;


    // Décleché lorsque le trigger du Collider est déclenché.
    private void OnTriggerEnter(Collider other)
    {
        // n'agit que si l'objet déclencheur est un object grapable.
        if(other.tag == "GrapableObject")
        {
            PlaySounds();
            WcParticules.Play();
            controlPannel.SendObject(other.gameObject );
        }

    }

    private void PlaySounds()
    {
        waterSound.Play();
        teleportMusic.Play();
    }
}
```

Tous 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 évidemment, il est toujours possible de créer des méthodes personnalisées, qu'elles soient privées ou publiques, comme *Playsound()* dans l'exemple ci-dessus.

L'héritage de *MonoBehaviour* nous donne toutefois accès à toute une série de méthodes 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.

Parmi 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 qui 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 pas surcharger le programme, cette méthode se déclenchant entre 30 et 60 fois par seconde.

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 toutes classes C#, on peut créer à loisir des propriétés qui représenteront ce dont nous aurons besoin dans notre script. La particularité de Unity est que les propriétés publiques seront visibles et assignables dans Unity.

On remarque dans l'exemple de « *WcWater.cs* » que les propriétés publiques apparaissent dans l'inspecteur et que ces dernières sont assignables 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 commentaires 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 survol du nom de la propriété avec la souris.

Cela permet d'une part à une personne qui n'aurait pas conçu le code de comprendre quelle propriété il doit mettre à quel endroit et d'autre part, 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 structure particulière. Dans une implémentation classique d'applications telle que nous les avons vu en cours, on peut structurer le travail en faisant un diagramme de classes d'implémentations en se basant sur les classes des objets et leurs héritages. On sait qu'un objet de type « Ecoles » comprend des objets de type « Classe », qui contiennent eux-mêmes des « Elèves », etc.

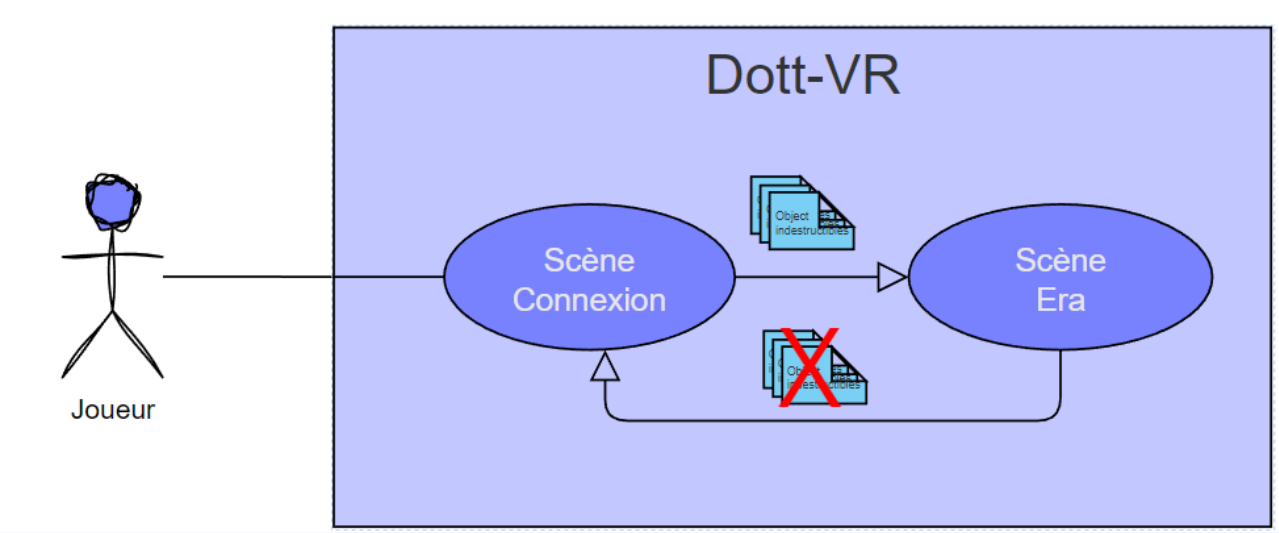
La difficulté ici est que toutes les entités du jeu sont des ***GameObjects*** et qu'il n'est pas possible de les différencier au sein d'un diagramme en se fiant à une classe qui les différencie. Chaque GameObject reçoit des scripts qui ont chacun une classe mais l'entité en elle-même reste une instance 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. D'ailleurs, on constate sur le site de Microsoft que ce problème est signalé et que ni les diagrammes d'analyse, ni la modélisation classique ne sont envisageables.
(*https://learn.microsoft.com/fr-fr/visualstudio/gamedev/unity/application-lifecycle-management-alm-with-unity-apps*)

Il m'a 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 » (Voire page 20) en m'appuyant sur la logique d'un diagramme de classe traditionnel.

Cycle de vie du jeu

Dans ce chapitre, j'explique ce qui se déroule au niveau de l'architecture durant une session de jeu.



Lorsque le joueur arrive dans le jeu, il arrive dans la scène de connexion ou il doit choisir une partie. Quand il l'a fait et qu'il a choisi lequel des trois personnage il désire incarner il est envoyé dans l'une des trois époques et la scène Era correspondante à son choix est donc lancée.

Ce qu'il est important de comprendre ici c'est que quand Unity passe d'une scène à l'autre, il détruit toute la scène que l'on vient de quitter ainsi que tous les **GameObjects** qu'il contient.

Nous verrons en cours de développement que cela peut nous poser des problèmes car nous souhaiterons que certains GameObjects doivent être conserver au passage de la scène **Connexion** vers l'une des **Era**.

Un problème risque toutefois de se présenter au moment où nous prévoyons la possibilité de revenir à tout moment à la scène **Connexion**. En effet, lorsque nous sommes dans la scène **Era,** nous avons des GameObjects venant de **Connexion** et comme nous les avons programmés pour être indestructibles au passage d'une scène à l'autre, si nous ne faisons rien, Unity va les conserver et les renvoyer dans la scène **Connexion**. Comme ils font partis des objets de base de la scène Connexion, elle va les réinstancier au chargement.

Il faudra donc prendre soin de détruire manuellement chacun des composants « Indestructibles » quand on passe d'une **Era** vers la scène de **Connexion,** sous peine de se retrouver avec deux joueurs par exemple.

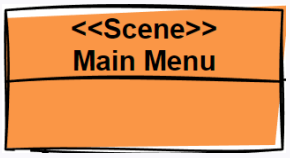
2 Développement

2.1 Diagramme de structure d'implémentation

Comme nous l'avons vu plus haut, 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

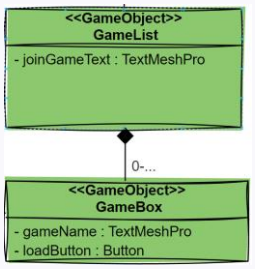


Elles sont en orange et portent la balise << Scene >>. Le nom de chaque « class » du diagramme correspond au type auquel elle appartient. Quand on parle de type ici, ce n'est pas en rapport avec les class mais plutôt avec la sorte de scène dont il s'agit. Ci-dessous nous parlerons de ***Type*** en se référant non pas à une classe mais au Tag attribué à un groupe de scènes comparable 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 représentées 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 toutes scènes 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 cases « GameObject » du diagramme correspondra à un Tag. À 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.

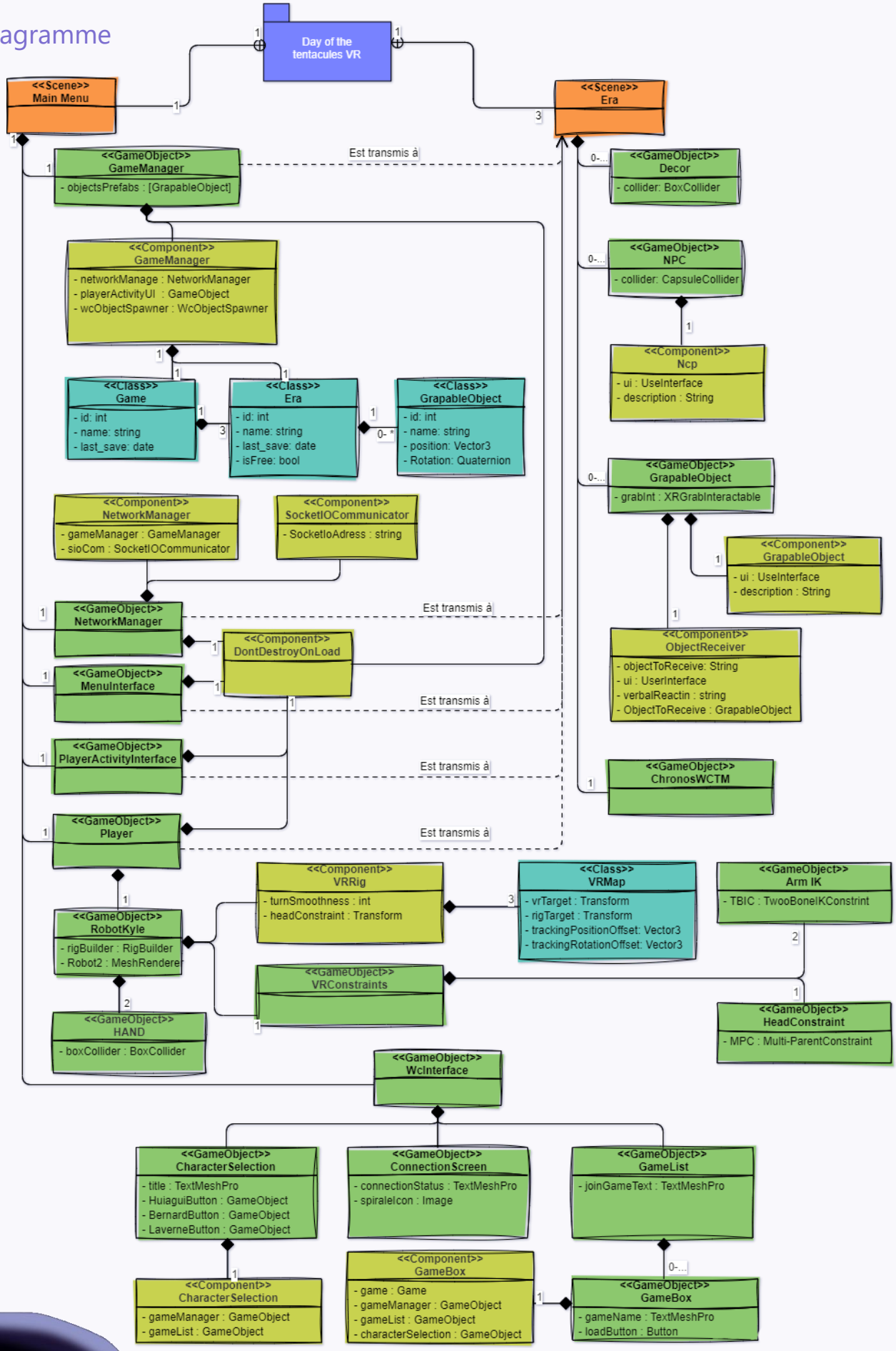
✓ Les Components

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

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

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

Le diagramme



2.2 Le processus d'implémentation

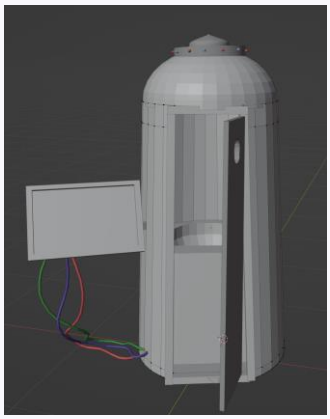
Dans ce chapitre je décrirai pas à pas les étapes du processus d'implémentation. J'ai suivi un processus de développement basé sur les **Use Cases**, en les implémentant les uns après les autres. Je reprendrais donc dans l'ordre chronologique les **Use Cases** que j'ai développé et expliquerai tout ce 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 obstacles rencontrés, les bugs qui se sont présentés et ceux qui posent encore des problèmes.

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 titanesque qui est généralement celui de toute une équipe de développeurs 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 étape, abordons les choix d'ordre généraux effectués.

Le graphisme

Comme nous en avons discuté avec monsieur Verhaegen, il est évident que le graphisme n'est pas la priorité de ce travail. Il était toutefois aisé de donner un design élégant 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 majeure partie des choses figurant dans ce travail.



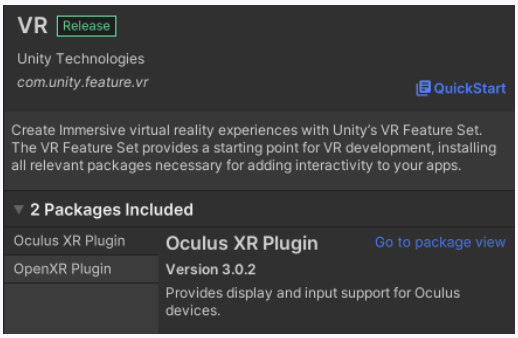
J'ai néanmois décidé de m'intéresser quelque peu à la construction d'éléments graphiques. En effet, il existe une relation étroite entre la partie graphique d'un objet et la programmation de ce dernier. Apprendre sommairement de quoi se compose une préfab m'a permis de comprendre comment on pouvait lui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

Par ailleurs, certains des éléments dont j'avais besoin afin de rester cohérent avec l'histoire n'était pas disponibles 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

  Il nous permet de rendre le projet compatible avec le casque Meta Quest 2 (anciennement Oculus Quest 2) et de fournir à Unity toutes les instructions concernant les contrôleurs.

- OpenXRPlugin

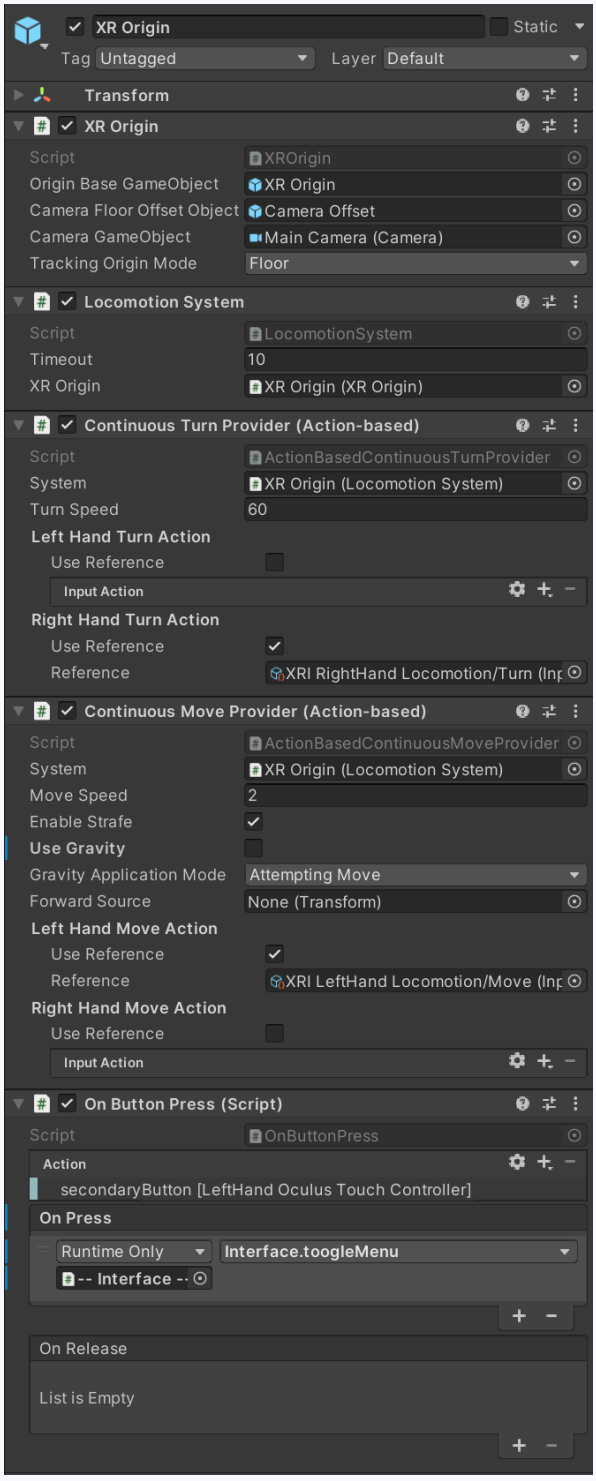
  OpenXR est une norme ouverte et libre de droits développée par Khronos qui vise à simplifier le développement AR/VR en permettant aux développeurs de cibler de manière transparente une large gamme d'appareils AR/VR.

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, les manettes et à les relier à Unity. Dès 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 manettes. Maintenant, regardons de 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 images qu'il capture pour les renvoyer au casque.
  - Un paramètre qui permet de déterminer sur base de quoi 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. Avoir plusieurs instances d'un **Locomotion System** conduisant un seul **XR Origin** n'est pas recommandé car il y aurait un grand risque de créer des conflits.

- Un component **Continuous Turn Provider**

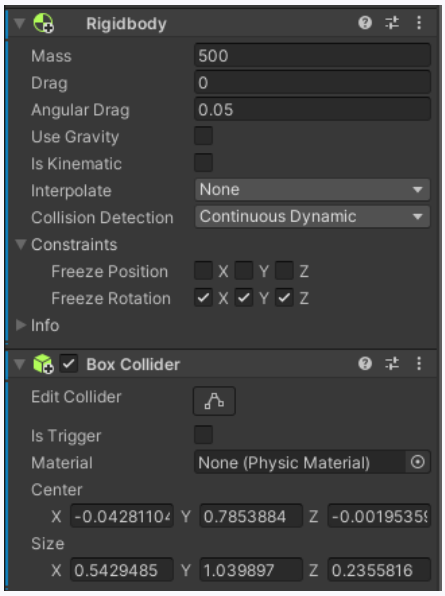
  Permet à l'utilisateur de faire pivoter en douceur le **XR Origin** en continu au fil du temps à l'aide d'une action d'entrée spécifiée.

- 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 à 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 à 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 vraie 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 grâce à l'ajout de cet élément que nous pouvons éviter de passer à travers les murs.

Problème rencontré

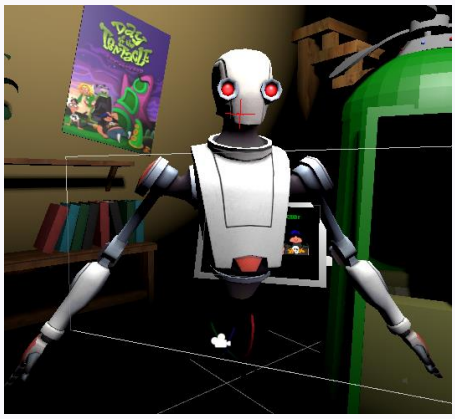
Quand j'ai mis en place le **Box Collider** la première fois, tout un tas de problèmes apparurent car je n'avais pas compris de prime abord que ce qui empêchait le personnage de passer à travers les murs était le Rigid Body.

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 initialisée par rapport au sol, le **Box collider** se retrouvait à 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 était comme un carcan et il ne pouvait donc plus avancer. La solution fut de donner au **Collider** une grandeur de 1m. De cette manière il couvrait 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 murs repoussent notre personnage, ce dernier se mettaient a tournoyer constamment. Pour éviter ceci, j'ai dû interdire la rotation sur tous les axes au **Box Collider**.

L' **XR Origin** étant instancié, 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 en synchronisation avec le personnage.

✓ Robot Kyle



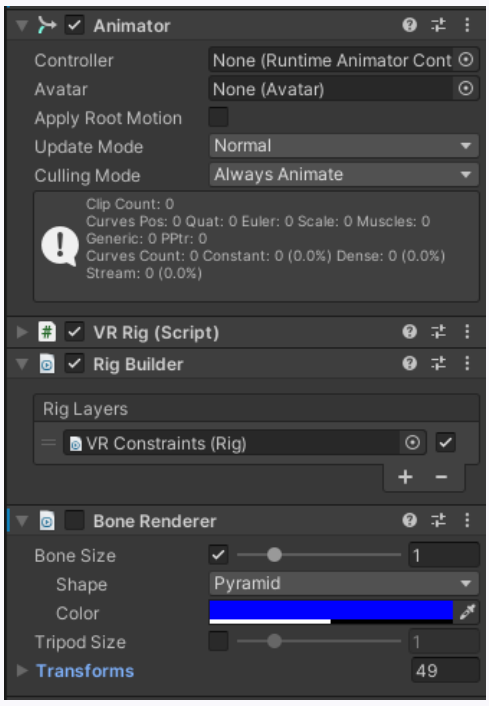
Comme précisé plus haut, nous allons utiliser un personnage préfabriqué à animer. Pour qu'il le soit, il doit posséder un **Rig,** c'est-à-dire un squelette. Son rôle est de rendre les mouvements du **mesh** de rendu harmonieux dès que le personnage bouge.

L'asset gratuit que j'ai pu trouver et qui répondait à ces conditions est **Robot Kyle**. Je me servirais donc de lui pour représenter tous mes personnages joueurs.

Pour animer notre personnage, nous allons utiliser une technique qu'on appelle l'**Inverse Kinematic**. C'est une technique qui consiste à faire bouger l'ensemble du corps en se basant sur les mouvements d'une seule de ses parties.

Si on observe Robot Kyle on voit que son corps est composé d'un tas de parties différentes. 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 les 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.

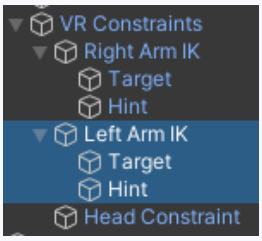
Afin de faciliter la sélection des différents os avec lesquels je vais travailler, je vais ajouter un Rig Builder qui va permettre d'afficher les différents os que possède **Robot Kyle**. Une fois que le composant est ajouté, je drag and drop tous les os de RobotKyle Dans la liste « Transform » qui est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

Nous remarquons aussi un component **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, le point d'entrée principal pour toutes les contraintes dans un Rig de contrôle donné. J'ai ajouté une référence vers mon nouveau gameObject au **Rig Builder** qui sait désormais où 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** ».

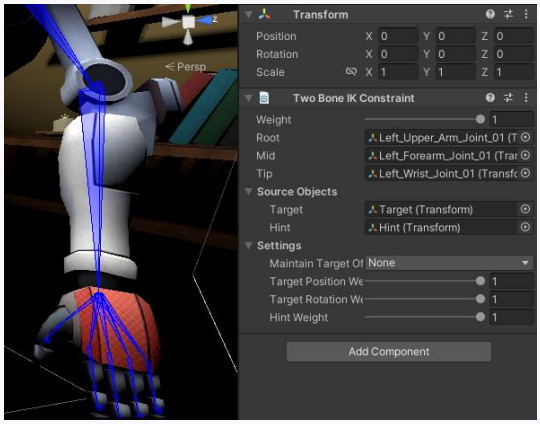
- **Les Arm IKs de Robot Kyle**



Aux contraintes des deux bras (**Right Arm IK** et **Left Arm IK**), nous allons ajouter un component qui se nomme **Two Bone IK Contstraint**. Au moment où on ajoute se composent, on voit que deux GameObjects 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 à l'endroit de la main et **Hint** un peu derrière le coude pour indiquer le sens de la pliure.

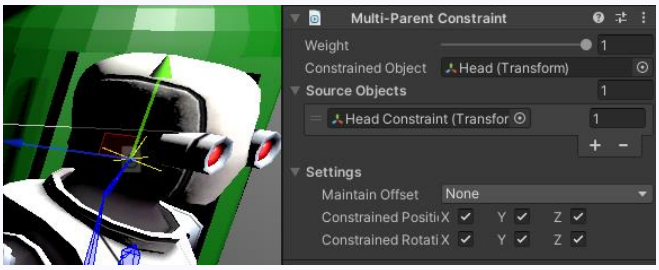


Désormais lorsque j'entre en Playmode et que je déplace le GameObject cible « **Target** », le bras part dans la bonne direction et se plie au bon endroit.

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

- **Le HeadConstraint de Robot Kyle**

Occupons-nous maintenant de la tête. Pour ce faire, je vais m'occuper du **Head Constraint** et faire en sorte que comme ses 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**.

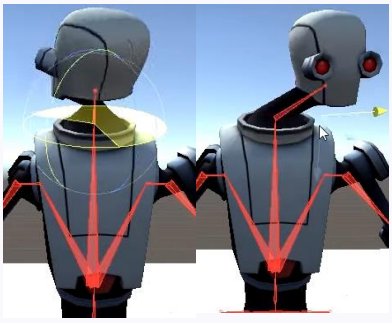


Comme pour les mains nous allons ajouter un component au **Head Constraint** mais ce dernier sera un peu différent.

Ici, il nous suffit d'ajouter un **Multi-Parent Constraint** qui va simplement déplacer et faire 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 corps 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 rédigé ci-dessous et qui va gérer les mouvements du personnage en les synchronisant avec le casque VR : le component **VRRig**.

- **Le VRRig de Robot Kyle**

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

Dans ce script je commence par créer une classe nommée **VRMap** qui va contenir toutes les données dont j'ai besoin pour chacune des parties du corps que je veux contrôler.

```
[System.Serializable]
public class VRMap
{
    public Transform vrTarget;
    public Transform rigTarget;
    public Vector3 trackingPositionOffset;
    public Vector3 trackingRotationOffset;



    public void Map()
    {
        rigTarget.position = vrTarget.TransformPoint(trackingPositionOffset);
        rigTarget.rotation = vrTarget.rotation * Quaternion.Euler(trackingRotationOffset);
    }
}
```

C'est-à-dire :

- **vrTarget**
  Le component **Transform** du casque VR, 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 où est l'os de la tête de Robot Kyle nous serions à l'intérieur de sa tête et la vision serait gênée 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 du **VRRig** depuis l'éditeur de code (à gauche) et depuis l'inspecteur Unity (à droite).

```
public class VRRig : MonoBehaviour
{
    public int turnSmoothness;

    public VRMap head;
    public VRMap leftHand;
    public VRMap rightHand;

    public Transform headConstraint;
    private Vector3 headBodyOffset;

    // Start is called before the first frame update
    void Start()
    {
        headBodyOffset = transform.position - headConstraint.position;
    }

    // Update is called once per frame
    void Update()
    {
        transform.position = headConstraint.position + headBodyOffset;
        transform.forward = Vector3.Lerp( transform.forward,
                                          Vector3.ProjectOnPlane(headConstraint.up, Vector3.up).normalized,
                                          Time.deltaTime * turnSmoothness );

        head.Map();
        leftHand.Map();
        rightHand.Map();
    }
}
```

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 du **Gameobject** dans lequel se trouve **Robot Kyle**.

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

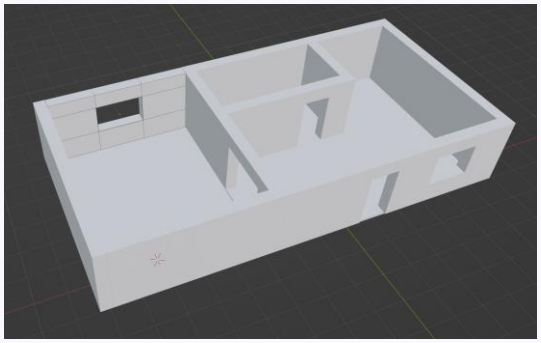
En suite , on lui demande de mettre la tête elle-même à la bonne position et on fait de même pour les mains, via la fonction map de chaque composant. On remarque dans le component vu par l'inspecteur Unity que l'on a drag and drop tous les éléments demandés par le script de façon à ce qu'il ait toutes 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 ont tout de même la particularité de posséder chacun un BoxCollider de manière à pouvoir être détecté et générer une collision en cas de besoin.

Le premier élément de décor 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** (composants qui gèrent la couleur et le rendu général d'un GameObjects) que l'on va appliquer à chaque mur afin que le décor varie selon les époques.

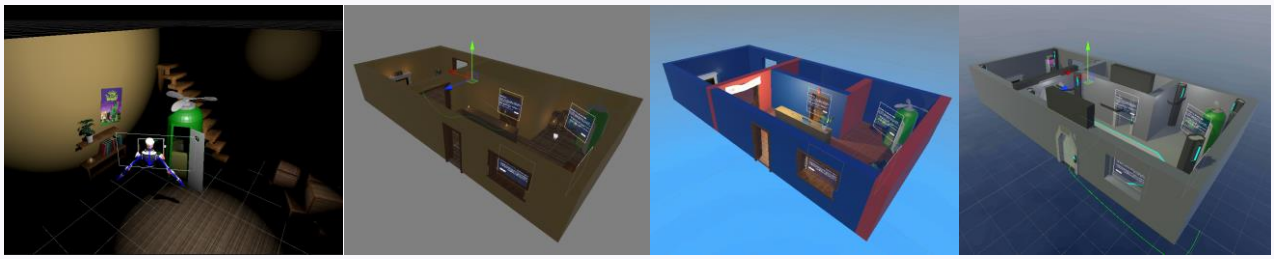
Les autres objets viennent de l'Asset-store de Unity, excepté bien entendu le **Chrono-WC** dont nous parlerons plus loin.

Voilà qui clôture ce chapitre sur les déplacements 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 murs 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.



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 que des éléments sont en commun, mais que les couleurs et les décorations varient selon les époques.

Prendre et déposer un objet

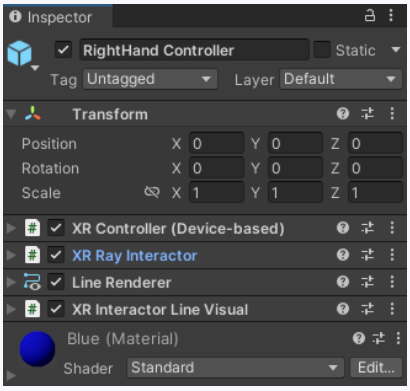
Afin de pouvoir prendre des objets en main il est nécessaire de faire quelques petits réglages sur le **XR Origin** et sur les GameObjects **LeftHandController** et **RightHandController** qu'il contient.

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 fournit à cet objet les références vers les boutons des contrôleurs.

✓ 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, en plus de référencer toute leurs touches.

- **XR Ray Interactor**



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

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

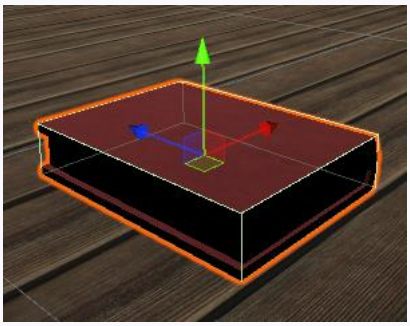
Le **Ray Interactor** contient toute une série de paramètres qui nous permettent de régler le comportement des interactions mais qui n'ont pas beaucoup d'intérêt à ce stade.

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 une épaisseur assez fine.

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

✓ GrapableObject

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



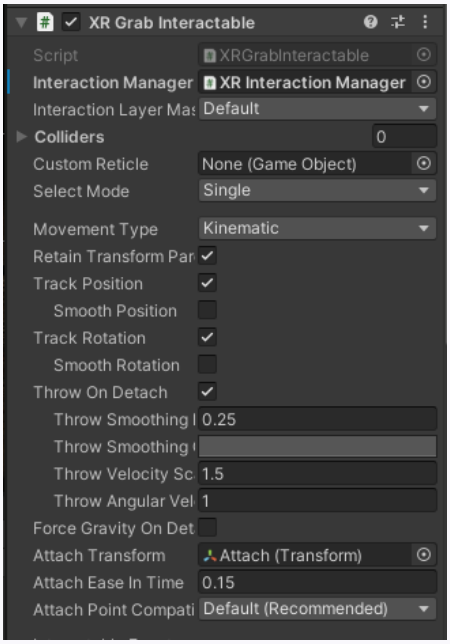
Prenons l'exemple d'un livre que nous voulons ajouter à la liste des objets que nous pourrons saisir.

Si nous nous contentons de l'ajouter et que nous entrons en **playMode**, le livre flottera dans les airs à l'endroit où nous l'avons placé 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 va 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 où il passerait à travers elle.

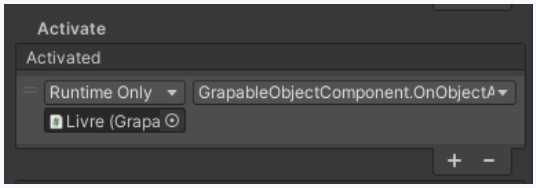
Pour éviter ce problème, nous allons donc créer pour chaque **GrappableObject** un GameObject enfant que nous allons appeler « **Attach** » et a qui nous allons donner une position et une rotation adéquates.

Dès lors, nous pouvons le fournir en paramètre au component **XR Grab Interactable.**

À ce stade, nous avons donc un rayon lumineux émis par chaque main et qui nous permet d'interagir avec les différents éléments du monde. En effet, les mains du Personnage sont des **Interactor** qui nous permettent d'utiliser tout objet **Interactacle** dans le monde. En plus de prendre les objets grâce à un **Interactable** spécifique comme le **XR Grab Interctable**, nous verrons dans les points suivants qu'il existe aussi d'autres interactions possibles qui une fois programmées nous permettront 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 component **XR Grab Interactable,** on voit qu'il est possible qu'une fonction soit lancée une fois que l'objet est activé (quand le joueur pointe l'objet ou le tient en main et qu'il appuie sur la gâchette située près de son index). N'entrons pas dans trop de détails 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.

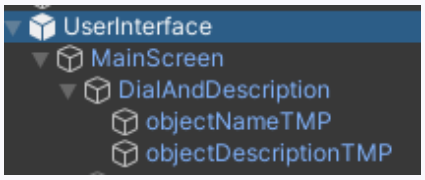
Pour ce faire, j'ai donc créé un component **GrappableObjectComponent** qui renferme les données de l'objet et la fonction dont j'ai besoin pour afficher un message à l'utilisateur quand j'active un objet.

Les étapes à suivre seront donc :

- Créer un 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 l' **UserInterface**

✓ GamebObjcet UserInterface

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



Je 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.

C'est à l'intérieur que nous allons placer le premier élément dont nous aurons besoin : Le GameObject **DialAndDescription**. Ce dernier sera une boîte 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 NPCs 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 destinées à être appelées afin de réaliser telle ou telle tâche. Voici à quoi ressemble le script :

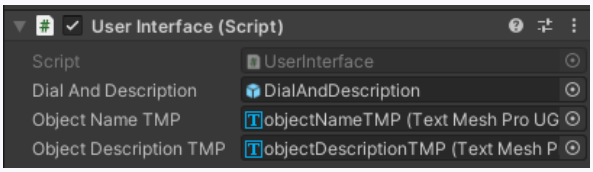
```
public class UserInterface : MonoBehaviour
{
    public GameObject dialAndDescription;
    public TextMeshProUGUI objectNameTMP;
    public TextMeshProUGUI objectDescriptionTMP;

    public void Start()
    {
        GameObject.DontDestroyOnLoad(gameObject);
    }

    public void displayDescription(string objectName, string objectdescription)
    {
        StartCoroutine(StartDisplayDescription(objectName, objectdescription));
    }

    private IEnumerator StartDisplayDescription(string objectName, string objectDescription)
    {
        objectNameTMP.text = objectName;
        objectDescriptionTMP.text = objectDescription;
        dialAndDescription.SetActive(true);
        yield return new WaitForSeconds(10);
        dialAndDescription.SetActive(false);
    }
```

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 NPC.



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 événement 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.

✓ GrappableObjectComponent

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

```
public class GrapableObjectComponent : MonoBehaviour
{
    public UserInterface userInterface;
    public string description;

    void Start()
    {
        userInterface = GameObject.Find("UserInterface").GetComponent<UserInterface>();

    }

    public void OnObjectActive()
    {
        userInterface.DisplayDescription(gameObject.name, description);

    }
}
```

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

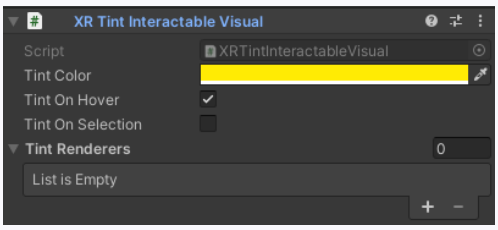
Nous retrouvons ici la fonction **OnObjectActive()** que nous avons associé tout à l'heure à l'**Activate** de **notre XR Grab Interactable**.

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

- ✓ Quand le joueur clique 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

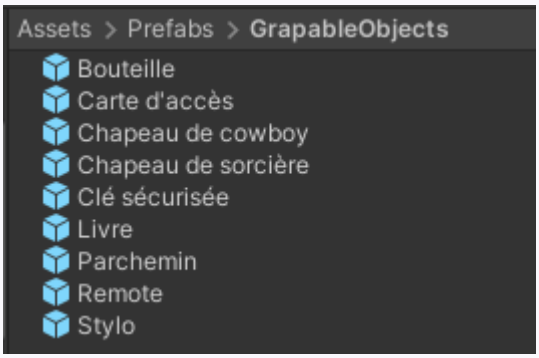
Il s'agit ici d'un simple petit ajustement cosmétique qui va consister à faciliter l'utilisation des objets. Afin que le joueur sache que l'objet est sélectionné, on aimerait que ce dernier s'illumine au survol. D'une part cela indique au joueur que des interactions sont possibles avec ce dernier et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



Pour y parvenir, nous allons 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 saisis et qui fournissent une description quand on interagit avec eux, nous pouvons en faire des préfabs.



Pour rappel, les préfabs sont des espèces de « modèles » d'un objet. Ils permettent d'une part, en cas de modifications, de répercuter les changements sur tous les **GameObjects** basés sur la préfab du jeu, et d'autre part de pouvoir instancier un **GameObject** basé sur cette préfab à tout moment.

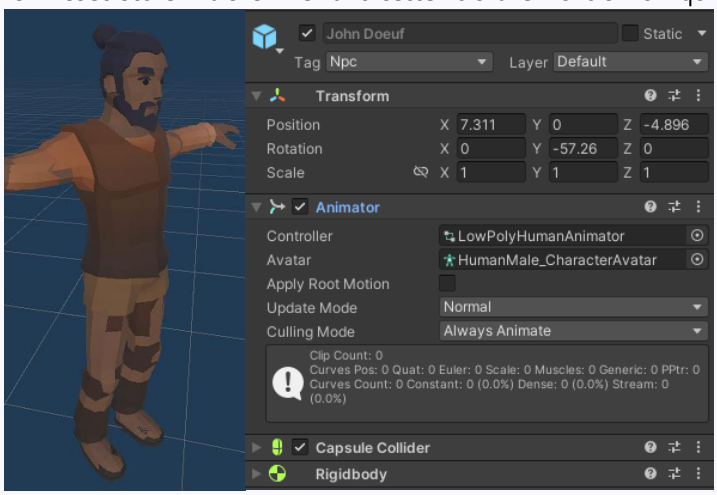
Nous verrons qu'avoir des préfabs de nos **GrapableObject** sera très utile pour la suite. Nous allons simplement drag and drop chaque **GrapableObject** que nous avons créés dans un dossier dédié de notre projet afin d'avoir notre collection de préfabs.

Obtenir des informations sur un NPC

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 NPC.

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

Comme pour **Robot Kyle**, nous le récupérons dans l'Asset store mais en veillant cette fois à en choisir un qui contienne des animations.



En effet, nous aimerions 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.

Nous lui ajouterons un component **Animator** qui servira à lancer des animations, mais nous verrons ça plus loin.

Dans un premier temps, ce que nous allons faire 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'interagir avec lui. Avec les grapableObjects, nous avions fait en sorte de pouvoir le faire en passant par le component XR GrabInteractable, qui servait avant tout à prendre les objets et possédait une option pour interagir de manières différentes. Dans le cas d'un NPC, 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 component qui offre des 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 nous créons un script pour chaque NPC qui contiendra des informations sur ce dernier, en plus de la fonction permettant d'appeler le UserInferface, exactement comme nous l'avions fait pour les GrapableObjects.

Voici donc le script Npc.cs :

```
public class Pnj : MonoBehaviour
{
    public UserInterface userInterface;
    public string description;

    // Start is called before the first frame update
    void Start(){
        userInterface = GameObject.Find("UserInterface").GetComponent<UserInterface>();
    }

    public void OnObjectActive(){
        userInterface.DisplayDescription(gameObject.name, description);
    }
}
```

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

Sauvegarder une partie

Maintenant que nous avons nos **grapableObjets** et nos **NPC**, 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'éléments. Afin de m'y retrouver, je vais commencer par faire le scénario de ce qui devra se passer lors d'une action de sauvegarde telle que je compte l'implémenter:

- Le joueur appuie sur la touche Y pour faire apparaître le menu.
- Le menu apparait sur son écran par l'intermédiaire de l'interface créée plus tôt, lui donnant le choix de sauvegarder ou quitter la partie.
- Le joueur clique sur sauvegarder.
- Une fonction SaveGame(), présente dans un script nommé GameManager qui renferme toutes les fonctions d'ordre général 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 toutes les communications avec le serveur du 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 et 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 objets Unity. Nous allons donc créer des classes C# correspondantes aux tables de la base de données en restant fidèle au diagramme de classe de l'analyse de base (Page 8).

Nous créerons ici les Classes nécessaires à la sauvegarde d'une partie : **Game**, **Era**, **GrapableObject** et **Npc**. Le diagramme de classe figurant déjà en page 8, 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.

```
namespace Models
{
    [Serializable]
    public class Era
    {
        public int id;
        public string name;
        public List<GrapableObject> grapableObjects;
        public List<Npc> npcs;
        public bool isFree = true;

        public Era(int id, string name, List<GrapableObject> grapableObjects, List<Npc> npcs)
        {
            this.id = id;
            this.name = name;
            this.grapableObjects = grapableObjects;
            this.npcs = npcs;
        }
    }
}
```

J'ai choisi volontairement la classe Era car elle contient une petite particularité. Contrairement à ses 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 connecter à cette **Era** en arrivant dans le jeu.

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

Nous remarquons également que dans celle classe Model comme dans toutes les autres, j'ai placé une balise **[Serializable].** Cette dernière sert à indiquer au compilateur que cette classe peut être « sérialisée », 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 sur 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 31) et lui ajouter un nouveau canevas qui s'appellera **Menu**.

Nous 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 côté pour le moment. Nous allons maintenant demander au bouton Save de pointer vers une fonction **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 19), 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 général.

- **Les propriétés**

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

```
public class GameManager : MonoBehaviour
{
    public Game activeGame = null;
    public Era activeEra = null;

    public List<GameObject> objectsPrefabs;
    public List<GameObject> npcsPrefabs;

    public NetworkManager networkManager;
```

Nous voudrions qu'il sache dans quelle partie (Game) on se trouve, quelles Eras (époques) sont contenues dans cette partie et quelle est l'époque active (passé, présent ou futur). Nous ajoutons donc les variables activeGame et activeEra.

Pour y parvenir, il nous faut 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 **GrapableObjects** et quels **NPCs** sont dans la scène. Nous lui ajoutons donc deux listes contenant ces informations qui sont elles aussi des listes 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 à communiquer avec le **NetworkManager.**

- **DontDestroyOnLoad**

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

Pour rappel, quand Unity charge une nouvelle scène, il détruit tous les objets qu'il a créé dans la scène actuelle. Dans notre cas, il nous faut éviter ceci car nous souhaitons garder les données 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.

```
void Start(){
    DontDestroyOnLoad(this.gameObject);
}
```

Nous allons donc dire à Unity qu'à l'initialisation de l'objet (dans la méthode **Start()**), 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().**

```
//création de la liste des GrapableObjects
var gosGapObjs = GameObject.FindGameObjectsWithTag("GrapableObject");
var grapableObjects = new List<GrapableObject>();
```

Avant tout, commençons par récupérer dans la scène active tous les GameObjects qui ont le tag « GrapableObject » et stockons-les dans une variable **gosGrabObj** (GameObjects Grapable Objects).

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

```
foreach (var go in gosGapObjs)
{
    var grapableObject = new GrapableObject();
    grapableObject.name = go.name;
    grapableObject.position = go.transform.position;
    grapableObject.rotation = go.transform.rotation;
    grapableObjects.Add(grapableObject);
}
```

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 NPCs.

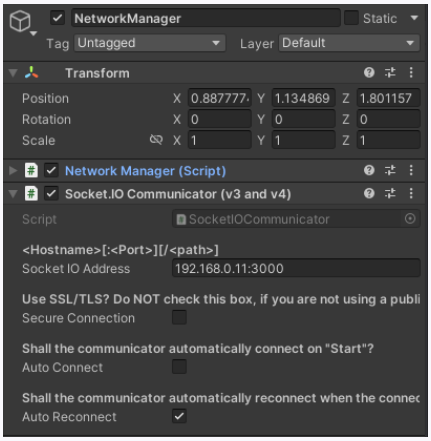
Je répète donc rigoureusement la même opération pour les **GameObject** pourtant le tag « Npc » et me voilà avec les deux listes de données que je dois sauver en base de données.

```
this.activeEra.grapableObjects = grapableObjects;
this.activeEra.npcs = npcs;
networkManager.SaveEra(activeEra);
```

Je stocke maintenant ces deux listes dans **l'Era** active et transmets cette dernière au **NetworkManager** afin qu'il la transmette au Backend pour les sauvegarder en DB.

✓ Création du NetworkManager

Le rôle du network manager sera de gérer le socket du côté Frontend, d'envoyer & de recevoir les requêtes et de transmettre les données reçues au GameManager pour traitement.



La première chose à faire est de créer un GameObject **NetworkManager** à la scène « Connexion ». Comme pour le **GameManager,** nous veillerons à ce qu'il ne soit pas détruit lors du chargement d'une autre scène et à ce qu'il survive le temps d'une partie.

Nous allons maintenant ajouter le package **SocketIO** à Unity et ajouter un component SocketIO à notre NetworkManger. Ce component reçoit l'adresse du server SocketIO que nous allons créer en Backend et s'y connecter dés l'instanciation de notre NetworkManager, c'est-à-dire au lancement de la scène « Connexion » .

Enfin, nous allons ajouter un component **NetworkManager** au GameObject du même nom et nous allons maintenant le passer en revue.

- Les propriétés

```
public GameManager gameManager;
private SocketIOCommunicator sioCom;

public GameObject connectionScreen;
public GameObject connectionStatus;
```

A ce stade, les deux seules choses dont nous ayons en priorité besoin sont :

- **GameManager** afin de communiquer avec lui ;
- **SocketIOCommunicator** pour gérer le socket.

Nous allons toutefois ajouter deux propriétés qui nous serons très utiles afin d'afficher le statut de la connexion et renseigner le joueur.

En effet, dans la scène « Connexion », le joueur interagit avec le jeu par le biais d'une interface présente sur l'écran des Chronos-WC. Nous allons donc faire un lien avec cette interface de manière à pouvoir afficher diverses choses, comme l'état de la connexion par exemple.

- La fonction Start()

Dans le start, nous ajouterons un **DontDestroyOnLoad** comme nous l'avions fait pour le GameManager et assignerons la valeur de **sioCom** (qui gère le socket).

C'est également ici que nous allons configurer l'entièreté du socket en lui disant comment il doit réagir à tout événement. Par exemple, dans l'événement suivant nous allons lui demander de nous signaler tout établissement de connections par un nouveau socket :

```
sioCom.Instance.On("connect", (payload) =>{
    Debug.Log("Connected! Socket ID: " + sioCom.Instance.SocketID);
    sioCom.Instance.Emit("getGames");
});
```

Une fois que nous avons donné toute les instructions à notre SioCom, nous lui demandons de connecter le socket via la Coroutine **ConnectSocket()** qui va se charger d'afficher l'écran de connexion sur l'écran des Chronos-WC, et d'y changer le texte pour le faire correspondre au statut de connexion.

Une fois que la connexion est établie, il affiche la liste des parties que nous allons créer dans le chapitre suivant (Chapitre *Voir la liste des parties* - Page 42).

```
IEnumerator ConnectSocket(){
    connectionScreen.SetActive(true);
    sioCom.Instance.Connect();

    yield return new WaitUntil(() => sioCom.Instance.IsConnected());

    TextMeshProUGUI Texte =
        connectionStatus.GetComponent(typeof(TextMeshProUGUI)) as TextMeshProUGUI;
    Texte.text = "Connected!";

    yield return new WaitForSeconds(2);

    gameLists.SetActive(true);
    connectionScreen.SetActive(false);
}
```

- La fonction SaveEra()

À présent, voici venu le moment d'ajouter la fonction à laquelle nous avons fait appel dans le GameManager afin de sauver une Era.

Elle reçoit l'Era que le **GameManager** lui a envoyé, la transforme en JSON et l'envoie au backend via la méthode **Emit()**.

```
public void SaveEra(Era era){
    string eraJson = JsonUtility.ToJson(era);
    sioCom.Instance.Emit("saveEra", eraJson,false);
}
```

✓ Création du Backend

Le backend que nous allons créer est une application Node. Après avoir créé l'application et installé les modules **Express, mysql** et **SocketIO** je vais simplement configurer le tout dans **index.js** qui est le point d'entrée de l'application.

```
var app = require('express')();
var server = require('http').createServer(app);
var io = require('socket.io')(server);
var mysql = require('mysql');
var sqlCon = mysql.createConnection( config: {
    host: "localhost",
    user: "root",
    password: "",
    database: "dott-db"
});
server.listen( port: 3000);
```

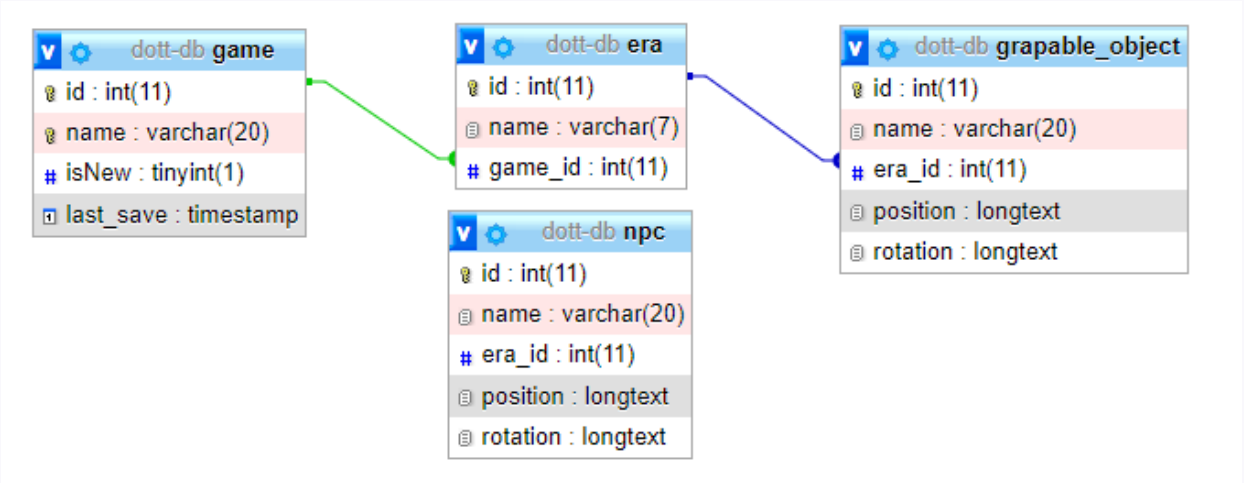
Nous avons maintenant un **server http,** un server **socketIO** et une connexion à la DB via **mysql.**

C'est dans la déclaration de msql que nous allons donner les paramètres du server MySql qui est un simple server Xampp et qui contient la base de données dont nous allons parler juste après.

On demande au server d'écouter sur le **port 3000** pour qu'il envoie et reçoive les requêtes faites par le socket. Le port 3000 est en effet recommandé pour tout ce qui est transport de données par Socket.

- La base de donnée

La base de données sql tourne sur **Xamps** qui utilise MariaDb. Sans grande surprise, la structure correspond à celle qui avait été annoncée dans le diagramme de classe de l'analyse avec juste un booléen isNew en plus pour le la table Game qui indique qu'aucune sauvegarde n'a encore été faite sur cette partie.



- Comportement du socket à la connexion

Nous allons maintenant dire au socket comment il doit se comporter durant la connexion d'un nouveau socket.

```
io.on('connection', function(socket) {

    var roomName = null;
    var activeEra = null;

    console.log("new connection Socket: " + socket.id);
    console.log("");

    socket.on('saveEra', function (data) {...});
```

Via la fonction On() nous allons pouvoir dire à socketIO que quand il reçoit une nouvelle connexion il doit instancier deux nouvelles variables (que nous utiliserons un peu plus loin) **roomName** et **activeEra**.

C'est ici-même, à l'intérieur de la connexion au Socket que nous allons définir toutes les réactions de notre socket.

Nous allons donc commencer ici par celle qui nous intéresse pour le moment : saveEra qui correspond à la requête que notre jeu a envoyé tout à l'heure.

- saveEra

Nous allons dire au socket comment il doit se comporter quand il reçoit la requête « saveEra ».

```
socket.on('saveEra', function (data) {
    let eraId = data.id;

    //Retire de la base de données tout les objets qui ne sont pas dans la liste d'obets reçus
    removeUnnecessaryObjects(data);

    //Mise à jour de tous les objets reçus en base de donnée
    data.grapableObjects.forEach(g => {
        updateGrapableObject(eraId, g);
    });

    //Mise à jour de tous les npcs reçus en base de donnée
    data.npcs.forEach(npc => {
        updateNpc(eraId, npc);
    });
});
```

Il va commencer par faire appel à une fonction **removeUnnecessaryObjects** qui prend l'Era que nous avons envoyée depuis le front et qui va retirer de la base de données tous les objets qui y sont mais ne figurant pas dans la liste des objets que nous souhaitons sauver.

En effet, il est possible qu'un objet ait été envoyé dans une autre scène ou donné à un npc et qu'il ne soit plus présent dans l'Era actuel alors qu'il y était lors de la dernière sauvegarde.

Ensuite, nous passons en revue tous les **grapableObjects** présents dans l'**Era** et les mettons à jour en base de données via la méthode **updateGrapableObject** qui va ajouter l'objet en DB s'il n'existe pas, ou le mettre à jour s'il est déjà présent.

Nous faisons de même pour les NPC.

Afin que ce rapport ne devienne pas trop « indigeste » par sa longueur et son niveau de détails, je me contenterai d'expliquer ce que font les fonctions **removeUnnecessaryObjects**, **updateGrapableObject** et **updateNpc.**

En effet, je pars du principe qu'un passage en revue total du code n'est pas nécessaire, mes professeurs ayant reçu le code en parallèle de ce rapport.

Voilà qui clôture ce long chapitre sur la **sauvegarde des parties**. Ce chapitre étant celui où j'ai créé tous les éléments dont nous allions avoir besoin par la suite, il s'est avéré particulièrement long et fastidieux.

Nous avons toutefois à présent un **Backend**, notre **GameManager** et notre **NetworkManager** qui vont certainement être les scripts les plus utilisés pour le reste de ce travail.

Voir la liste des parties

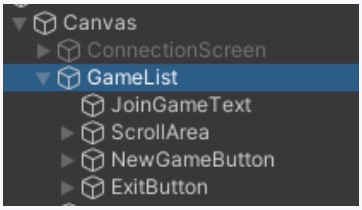
Comme nous l'avons vu plus haut, lorsqu'un joueur lance le jeu, il arrive dans la scène « Connexion » où il se retrouve devant les Chronos-WC et leur panneau de contrôle avec lequel il peut interagir. Dés que la scène se lance elle affiche un écran qui lui indique que le jeu est entrain de se connecter au serveur et indique le succès de la connexion.

Nous voudrions à présent qu'une fois que la connexion est établie, il voit la liste des parties disponibles sur le serveur.

Nous avions déjà prévu plus haut l'affichage de la liste des parties, dans le paragraphe consacré au **NetworkManager** et à la création de sa fonction **start()** (*Page 39*). Nous avions alors fait en sorte que dès qu'une connexion était établie, le message « Connected ! » soit affiché sur l'écran actuel et que ce dernier soit rendu inactif pour laisser place à un GameObject nommé « **GameList** ».

✓ Le GameObject GameList

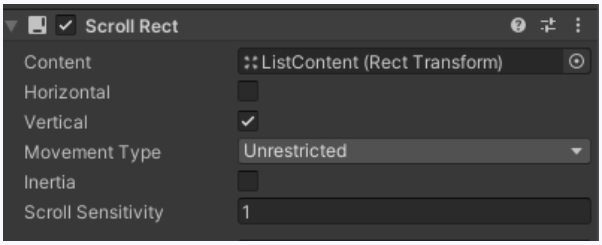
Dans le canevas que nous avons créé précédemment pour l'affichage de l'écran du Chrono-WC, nous allons donc ajouter le GameObject **GameList** et lui ajouter plusieurs **GameObject** enfants que nous allons passer en revue.



- **JoinGameText** est le titre de cet écran, seulement composé d'un component TextMeshPro qui est un simple champ texte.
- **NewgameButton** et **ExitButton** sont les deux boutons qui apparaitront en bas de la liste et qui ne pointent encore sur rien à ce stade.
- **ScrollArea** va contenir une liste de **GameObjects** à afficher. C'est l'élément qui nous intéresse ici et que nous allons passer en revue dans ce chapitre.

- GameObject ScrollArea

Il s'agit d'un élément UI proposé par Unity et qui permet de gérer facilement une liste de **GameObjects** qui vont automatiquement s'afficher les uns après les autres.



ScrollArea contient un component **ScrollRect** qui permet de le paramétrer et ainsi décider du comportement qu'il va adopter.

Nous allons notamment pouvoir préciser que la liste est verticale et définir le type de mouvement désiré pour faire défiler la liste.

Problème rencontré et non-résolu

Nous remarquons que le paramètre **MovementType** a été mis sur « Unrestricted » ce qui permet de faire défiler la liste indéfiniment et même de la faire totalement disparaitre en cas de scroll abusif vers le haut ou le bas. Ce choix a été fait délibérément afin de contourner un problème rencontré et choisir la solution « du moindre mal ».

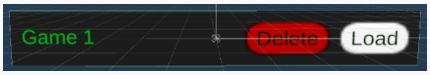
En effet, l'idéal aurait été de laisser le paramètre de base qui était « Elastic » et permettait de faire revenir la liste à une position correcte si l'on remontait trop haut ou trop bas lors d'un scroll. Toutefois, pour une raison que je ne comprends pas, ce paramètre m'empêchait d'afficher correctement les éléments les plus bas dans la liste. Lors d'un scroll vers le bas, nous apercevions un bref instant les éléments non-visibles jusqu'alors. Mais la liste revenait de façon « élastique » au début dès que l'on relâchait le scroll.

La solution précédente était bloquante car elle empêchait totalement l'accès à certaines parties. J'ai donc opté pour cette option qui ne pose un problème que dans le cas où l'on scroll trop.

Nous voyons également un paramètre Content qui pointe vers un autre GameObject qui contiendra l'ensemble des éléments à afficher et que nous créons et nommons « **ListContent** ».

- GameObject Gamebox

Notre but maintenant et de créer une Préfab qui pourra être instanciée un nombre illimité de fois à l'intérieur de ce **ListContent** et que nous appellerons **GameBox**.



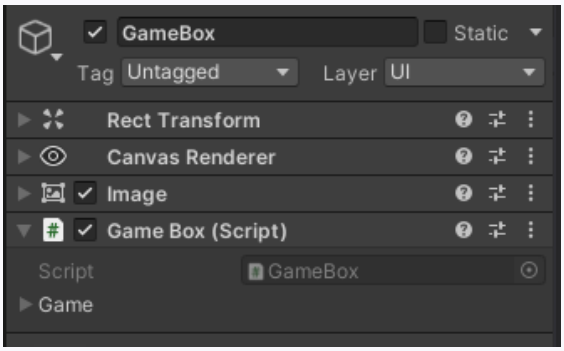
Le **GameBox** va donc être un **GameObject** parent qui va contenir différents éléments enfants.

Il contiendra en effet :

- Une GameObject contenant le texte du titre
- Un bouton Load pour charger la partie
- Un bouton Delete pour effacer la partie

Le Gamebox lui-même aura un component du même nom qui va nous permettre de contrôler le tout.

- Component GameBox



Le Script que j'écris ici à pour but principal de représenter une des **Game** présente sur le serveur et de donner au joueur la possibilité de l'effacer ou de la lancer.

On voit donc ci-contre que l'élément principal de ce script va être un objet Game (l'un de nos Models) que l'on souhaite pouvoir manipuler à souhait.

Il contiendra aussi deux fonctions : **`DeleteGame`** (que nous aborderons dans le chapitre suivant) et **`loadGame.`**

Dans ce script nous allons avoir besoin des variables suivantes :

- Un objet **Game** que nous viendrons initialiser à l'instanciation de chaque **GameBox.**
- Le **GameManager** pour lui confier certaines tâches en fonction du bouton pressé.

```
public class GameBox : MonoBehaviour
{
    public Game game;
    private GameObject gameManager;
    private GameObject gameList;
    private GameObject characterSelection;
```

- Le GameList que nous avons créé plus haut et qui est l'écran qui contient tous les **GameBox**.
- Et un GameObject **CharacterSelection** que nous n'avons pas encore créé mais qui sera l'écran que notre joueur devra voir s'il appuie sur « Load ».

On remarque que seul le Game est public car il sera assigné depuis l'extérieur. Les autres variables sont en privés car elle seront initialisées dans la fonction **Start().**

```
void Start()
{
    gameManager = GameObject.Find("GameManager");
    gameList = GameObject.Find("GameList");
    characterSelection = GameObject.Find("CharacterSelection");
```

Problème rencontré

Au début, j'ai traité la préfab comme si j'avais à faire à un GameObject simple. Je l'ai mis dans la scène, avec les propriétés en public et un drag and drop des éléments dont j'avais besoin dans l'éditeur Unity. Enfin, j'ai fait une préfab du GameObject.

Le problème est qu'avant d'être instanciée, une préfab ne connait pas la scène et ne peut donc pas recevoir les objets en faisant partie. Il m'a donc fallu mettre toutes les propriétés en privées et aller les chercher via la fonction Find() dans le Start du script.

La dernière étape est maintenant de créer la fonction **LoadGame()** qui sera directement reliée au bouton.

```
public void loadGame()
{
    var txt = gameObject.GetComponentInChildren<TextMeshProUGUI>().text;
    gameManager.GetComponent<GameManager>().activeGame = game;
    gameManager.GetComponent<GameManager>().GetAreasOfTheActiveGame();

    gameList.SetActive(false);
    characterSelection.SetActive(true);
    deleteScreen.SetActive(false);
}
```

Cette dernière renseigne le **GameManager** sur la partie qui a été sélectionnée par le joueur en assignant le Game à la variable **activeGame** et envoyer une requête en Backend afin de récupérer les Eras de l'activeGame.

Elle va ensuite masquer le GameObject **gameList** et afficher le **characterSelection**.

✓ Récupération de la liste des parties

À ce stade nous sommes donc prêt à recevoir la liste des parties et à instancier un **GameBox** par partie, en n'oubliant pas d'assigner un objet **Game** à chacun d'entre eux. Comme nous le savons déjà, dès l'instanciation du NetworkManager ce dernier lance une requête « **getGames** » au Backend (voir page *39*).

Du coté Backend, il nous faut maintenant faire en sorte que le Socket réagisse correctement à cette demande, c'est-à-dire qu'il aille chercher toutes les parties disponibles sur le serveur et les renvoie au jeu.

Il exécute donc une simple requête SQL et stocke chacun des résultats dans une collection. Il renvoie cette dernière en émettant une requête « gamesListReceived » qui contient un objet JSON {games :games}.

```
socket.on('getGames', function () {
    console.log("Games list required by socket " + socket.id);
    var games = [];
    sqlCon.query("SELECT * FROM game", function (err, result, fields) {
        if (err) throw err;

        result.forEach(row => {
            games.push(row);
        });

        socket.emit('gamesListReceived', {games: games});
```

Il s'agit maintenant de récupérer cette liste de Game du coté Frontend.

Problème rencontré

On remarque que le Json qui nous est envoyé depuis le Backend est un JSON contenant une variable « games » qui est une liste de Games. J'ai donc essayé au début de convertir chaque élément de cette liste en **Game** de diverses manières, mais malheureusement, il n'est ni possible de faire un foreach sur un JSON et ni possible de convertir directement une liste d'objets via l'outil JsonUtility que j'utilise.

J'ai donc dû me résoudre à créer un Model **GameList**, comme je l'ai vu faire sur internet dans un tutoriel Youtube (https://www.youtube.com/watch?v=tn3cWcSYmHE&list=PLH83kaNoEWK5wkiNhGlhfSI_nGDcR1DX8).

En effet, de nombreuses sources m'on confirmé que c'était là une limite de l'outil qui ne permet pas d'écrire quelque chose comme : JsonUtility.FromJson(payload.games, typeof(Game[]))en espérant qu'il renverra une liste d'élément Game.

- **Création du model GameList**

```
[Serializable]
public class GameList
{
    public List<Game> games;

    public GameList(){}
}
```

Afin d'avoir un Model conforme à l'objet Json reçu pour pouvoir le convertir en objet, nous allons créer GameList qui a la même configuration.

Comme les autres Models, il doit être sérialisable et contenir une liste de Games.

À présent, nous allons commencer par ajouter une variable GameList privée au **NetworkManager** et ajouter une réaction du socket à la requête que le Backend a envoyé.

```
sioCom.Instance.On("gamesListReceived", (payload) =>
{
    gameList = JsonUtility.FromJson(payload, typeof(GameList)) as GameList;

    gameList!.games.ForEach((game) =>
    {
        GameObject box = GameObject.Instantiate(gameBox, listContent);

        //Debug.Log(box);
        var GBScript = box.GetComponent("GameBox") as GameBox;
        //Debug.Log("script Game: " + GBScript.game);
        GBScript.game = game;

        var tmp = box.GetComponentInChildren(typeof(TextMeshProUGUI)) as TextMeshProUGUI;
        tmp.text = game.name;
    });

});
```

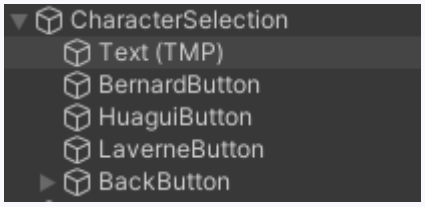
Pour chaque Game, il instancie un **GameBox** dans l'objet **listContent** qui contient l'ensemble des **GameBox** affichés. Il place le Game dans le component **GameBox** et change son titre afin que chaque box affiche le nom de la partie.

Charger une partie

Maintenant que nous avons une liste de parties, nous devons nous assurer que lorsqu'on appuie sur le bouton « Load » d'un GameBox, nous entrions dans cette partie et arrivions sur la sélection d'un personnage. Une fois que l'on va sélectionner un personnage, nous allons entrer dans l'Era lui correspondant.

Comme nous l'avons vu dans le chapitre des Gamebox (Page 43), quand un joueur appuie sur load, l'écran de sélection de personnage s'affiche. Nous allons donc créer ce GameObject.

✓ GameObject CharacterSelection



Ce GameObject va contenir 5 enfants :

- Un texte qui correspond au titre de l'écran
- 3 boutons correspondant chacun à un personnage
- Un bouton « Back » qui permet de revenir à l'écran de sélection d'une partie.

Chacun des boutons renvoie à une fonction du script **CharacterSelection** attaché au GameObject du même nom et à un petit script qui change sa couleur en rouge si le boolean « IsFree » de l'Era est false.

✓ Component CharacterSelection

Il contient une référence vers le GameManager, son script et la liste des parties.

```
public void LoadHuagui()
{
    SceneManager.LoadScene("PastScene");
    var gameManagerScript = gameManager.GetComponent<GameManager>();
    var disponiblesEras = gameManagerScript.activeGame.eras;
    gameManagerScript.EnterInEra(disponiblesEras.Find(e => e.name == "Past"));

}
```

Lorsque le joueur clique sur un personnage (Huagui dans cet exemple), la fonction correspondante signale au **GameManager** que dans les **Era**s disponibles dont il dispose pour la partie active, il doit sélectionner celle correspondant à l'époque que le joueur a choisie. Il charge la scène correspondante et lance la fonction **EnterInEra()** qui reçoit la bonne Era comme paramètre.

✓ La fonction EnterInEra()

```
public void EnterInEra(Era era)
{
    this.activeEra = era;
    networkManager.ConnectToEra(era);
}
```

Cette fonction installe l'Era fournie par l'interface comme activeEra et demande signale au Backend que cette Era a été sélectionnée.

Le NetworkManager envoie la requête « connectToEra » au Backend.

Du coté Backend :

```
socket.on('connectToEra', function (data){

    console.log("Player connection required to era " + data.id + "with socket " + socket.id);
    console.log("  └→ " + disponibleEras.length + " disponibles Eras" );

    activeEra = disponibleEras.find(e => e.id == data.id);
    console.log("  └→ Era n°" + activeEra.id + " active: " + activeEra.name );

    //  retrait de l'Era de la liste des Eras disponibles
        var index = disponibleEras.findIndex(e => e.id == data.id);
        disponibleEras.splice(index,  deleteCount: 1);

    console.log("  └→ " + disponibleEras.length + " disponibles Eras left" );
    console.log("");

    activeEra.isFree =false;
    io.to(roomName).emit("PlayerJoin", activeEra);
});
```

On va retirer l'Era de la liste des Eras Disponibles qu'on avait constitué au lancement de serveur et signaler à tous les joueurs connectés à la même partie qu'un joueur a rejoint la partie.

```
public void PlayerJoin(Era era)
{
    if(activeGame != null)
    {
        activeGame.eras.ForEach(e => {
            if(e.id == era.id)
            {
                e.isFree = false;
            }
        });

        StartCoroutine("DisplayJoinTheGame", era);
    };
}
```

Une fois que le GameManager est mis au courant de l'arrivée sur son activeGame d'un nouveau joueur, il va faire deux choses :

Il parcourt les **Era**s de son **activeGame** et change le booléen « isFree » de celle qui vient d'être occupée.

En suite, il lance une coroutine qui va afficher un petit message sur l'interface utilisateur pour signaler qu'un joueur a rejoint la partie.

✓ GrapableObjects et Ncp Spawner

À ce stade, nous sommes bien arrivés dans la partie à l'époque que nous avons choisie. Au moment où la scène se charge, notre GameManager (qui a survécu au chargement de la scène grâce au DontDestroyOnLoad) connait la partie active ainsi que la scène active.

Nous allons profiter de cette situation pour créer deux GameObjects qui, à leur instanciation vont récupérer respectivement les GrapableObjects et les NPCs de la scène. Ces deux GameObjects sont **NpcSpawner** et **GrapableObjectsSpawner**.

Ils ont tous les deux strictement le même fonctionnement et la même structure. Dans la méthode Start du script qui les compose, ils vont demander au GameManager de récupérer les **GrapableObject/Npc** de la scène et de les instancier.

```
objList.objects.ForEach(obj => {
    Debug.Log("instanciation de l'objet " + obj.name);
    var prefab = objectsPrefabs.Find(o => o.name == obj.name);
    var newObj = GameObject.Instantiate(prefab, obj.position, obj.rotation);
    newObj.name = obj.name;
});
```

Comme ce dernier connait l'Era active, il est parfaitement capable d'envoyer une requête au Backend pour récupérer tous les **GrapableObject/Npc** de la scène en base de données et une fois reçue, il va instancier chacun d'entre eux à la place à laquelle ils ont été sauvegardé lors de la dernière partie.

Supprimer une partie

Le processus pour supprimer une partie est très simple. Lorsque nous avions créées les GameBox (page 43) nous avions placé un bouton « Delete ». Contrairement à son homologue, le bouton load(), il ne va pas pointer directement sur une fonction du GameManager.

Il va d'abord ouvrir une demande de confirmation de suppression et en cas de validation va demander au **NetworkManager** d'envoyer une requête au backend et supprimer le **GameBox** correspondant de la liste des parties (**GameList**).

De son coté le Backend va simplement exécuter une requête SQL qui va supprimer la partie de la base de données.

Envoyer un objet à un personnage

Nous allons jouer ici avec les triggers de Unity. Pour rappel, quand on met un collider sur un GameObject on peut cocher l'un de ces paramètres pour en faire un Trigger. On peut alors se servir de la fonction OnTriggerEnter de tout component pour déclencher une réaction à ce Trigger.

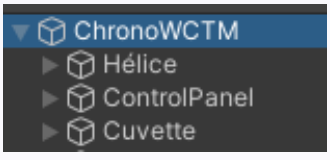
Nous voulons donc envoyer un objet d'un Personnage à un autre par l'intermédiaire des Chronos-WC. Le mécanisme général est le suivant :

- En appuyant sur l'un des deux boutons le joueur installe une variable dans le **ControlPanel** qui renseigne ce dernier sur l'Era de cette Game recevant l'objet.
- Quand il jette un **GrapableObject** dans l'eau de la cuvette un trigger se déclenche, demande au **ControlPannel** quelle est l'**Era** de destination et passe le tout au GameManager pour qu'il envoie l'objet à la bonne époque et qu'il le détruise.
- Le GameManager de son côté passe toute les infos au NetworkManager qui transmet au Backend.
- Le Backend reçoit l'objet à envoyer et le change d'Era en base de données avec des coordonnées correspondant à l'endroit situé juste devant les Chronos-WC. À ce stade, si un joueur rejoint l'Era de destination, il trouvera le GrapableObject devant les Chronos-WC. Il retire également l'objet de l'Era de départ de manière à ce que l'objet ne se retrouve pas dans les deux Eras.
- Le Backend va tout de même envoyer une notification à tous les sockets connectés à cette partie afin de leur faire part de l'envoie. En effet, si l'Era de destination est actuellement occupée par un joueur, dans l'état actuel des choses, il devrait relancer son jeu pour voir l'objet au chargement de l'Era. Pire : s'il fait une sauvegarde, il écrasera l'enregistrement qui vient d'avoir lieu en base de données.
- Tous les jeux connectés à cette partie reçoivent une notification du serveur et si leur **ActiveEra** est celle qui devait recevoir l'objet, les Chronos-WC font apparaitre l'objet.

✓ GameObject ChronoWCTM

Nous allons commencer par ajouter les Chrono-WC dans chacune des scènes représentant une époque. Nous l'appelons **ChronosWCTM** (pour Time Machine) afin différencier sa préfab de celles des Chronos-WC de l'écran de connexion.

J'ai créé ce Chrono-WC sur Blender de telle façon qu'il soit décomposés en plusieurs GameObjects parmi lesquels deux qui nous intéressent tout particulièrement ici : **Water** et **ControlPanel**. **ControlPanel** va servir à la sélection du personnage de destination et **Water** va nous servir à l'envoi proprement dit.



✓ Le Gameobject ControlPannel

Ce GameObject contient un script du même nom qui a les attributs suivants :

```
public string selectedEra;

public WcButton button1;
public WcButton button2;

private GameManager gameManager;
```

Il retient en permanence l'Era sélectionnée afin de pouvoir la transmettre au GameManager en cas de besoin.

Il a une référence vers ces deux boutons afin de pouvoir les contrôler en cas de besoin, comme nous allons le voir ci-dessous.

Et naturellement, il connait le GameManager afin de pouvoir lui demander d'envoyer un objet.

- **La fonction Start()**

Nous allons nous arranger pour que les boutons soient dans la bonne position à l'instanciation de la scène et que la bonne valeur soit stockée dans **selectedEra**.

```
void Start()
{
    button1.SetUnPressed();
    selectedEra = button2.eraName;
    gameManager = GameObject.Find("GameManager").GetComponent<GameManager>();
}
```

Nous allons appeler la fonction qui met le bouton 1 dans l'état « non-pressé » et que nous découvrirons juste après. La **selectedEra** prend donc la valeur du bouton pressé c'est-à-dire le numéro 2.

Nous en profitons pour donner une valeur au GameManager. En effet, une fois de plus ce dernier n'était pas présent dans la scène avant l'instanciation de cette dernière, puisqu'il est initialisé dans la scène de connexion. Impossible donc de le drag and drop depuis l'interface avant de lancer le jeu et donc de lui donner une valeur initiale.

Les boutons en revanche font partie de la scène et peuvent donc être initialisés.

- **Les autres fonctions nécessaires**

Nous avons maintenant besoin d'une fonction qui inverse l'état des boutons. En effet, on souhaite que quand un bouton est appuyé, l'autre se relève.

```
internal void ToogleButtons()
{
    button1.inverseState();
    button2.inverseState();
}
```

Cette fonction sera appelée juste après par les boutons.

Enfin, naturellement nous avons besoin d'une fonction qui envoie l'objet dans l'Era sélectionnée.

```
public void SendObject(GameObject obj)
{
    Era targetEra = gameManager.activeGame.eras.Find(e=> e.name == selectedEra);
    gameManager.SendObjectTo(obj, targetEra);
}
```

✓ WCButton

Ce GameObject contient un script du même nom qui a les attributs suivants :

```
public bool isPressed;
public GameObject controlPannel;
public string eraName;
```

Un booléen qui indique si le bouton est pressé.

Une référence vers le **controlPannel** pour qu'il puisse communiquer avec lui.

eraName qui est un string que nous remplirons dans l'inspecteur pour chaque bouton et qui correspondra au nom de l'Era auquel on veut envoyer un objet.

- **OnTriggerEnter()**

Chaque bouton a un trigger qui détecte la main du joueur quand il presse le bouton.

```
private void OnTriggerEnter(Collider other){
    if (!isPressed) {
        var cp = controlPannel.GetComponent<ControlPannel>();
        cp.ToogleButtons();
        cp.selectedEra = eraName;
    }
}
```

Si le bouton n'est pas déjà pressé, on demande au ControlPannel d'inverser les boutons. Pour ce faire, il va Appeler la méthode **SetPressed()** sur un bouton et **setUnpressed()** sur l'autre.

```
public void SetPressed(){
    transform.localPosition = new Vector3(0,0.23f,0);
    GetComponent<Renderer>().material.color = Color.green;
    GetComponent<AudioSource>().Play();
    isPressed = true;
}

public void SetUnPressed() {
    transform.localPosition = new Vector3(0, 0.8f, 0);
    GetComponent<Renderer>().material.color = Color.red;
    isPressed = false;
}
```

✓ WcWater

Le GameObject Water contient un component WcWater qui est là pour détecter un **GrapableObject** entrant comme nous l'avons vu plus haut. Il en profite pour lancer de petits effets sonores et un effet de particules pour simuler l'éclaboussure de l'eau.

```
// Décleché lorsque le trigger du Collider est déclenché.
private void OnTriggerEnter(Collider other){
    // n'agit que si l'objet déclencheur est un object grapable.
    if(other.tag == "GrapableObject")
    {
        PlaySounds();
        WcParticules.Play();
        controlPannel.SendObject(other.gameObject );
    }
}
```

✓ Fonction SendObjectTo du GameManager

Cette fonction créer un model **GrappableObject** à partir du GameObject reçu afin de le passer au **NetworkManager** puis détruit le GameObject.

```
internal void SendObjectTo(GameObject obj, Era targetEra)
{
    var grapObj = new GrapableObject();

    grapObj.name = obj.name;
    grapObj.era_id = targetEra.id;
    grapObj.position = new Vector3(-4.3f, 1, -4.3f);
    grapObj.rotation = new Quaternion(0, 0, 0,0.9f);

    networkManager.sendObjectTo(grapObj);
    GameObject.Destroy(obj);
}
```

On remarque que lors de la création du model on va déjà lui donner ses nouvelles coordonnées (c'est-à-dire devant les Chrono-WC) et qu'on va en profiter pour déjà lui assigner le era_id de l'Era de destination.

Cela nous permet d'envoyer un seul objet GrapableObject au Backend sans devoir lui fournir l'Era.

La fonction **SenObjectTo** du **NetworkManager** se contente de transmettre au Backend l'objet converti en JSON.

Le Backend fait de son coté les modifications mentionnées plus haut en base de données et renvoie une requête "GrapableObjectReceived" à tous les sockets connectés sur cette partie en y joignant l'objet mis à jour.

```
socket.on('objectToSend', function (data){
    console.log("Object to send received for era " + data.era_id);

    sqlCon.query("DELETE FROM grapable_object WHERE `era_id` = " + activeEra.id +" " +
        "AND `name` = '" + data.name + "'", function (err, result, fields) {
        if (err) throw err;
    });
    updateGrapableObject(data.era_id, data);
    io.to(roomName).emit("GrapableObjectReceived", data);
});
```

Lorsque le NetworkManager reçoit cette requête, il fait appel à la méthode **SpawnObjectFromWC** du GameManager

✓ Fonction SpawnObjectFromWC du GameManager

Elle s'occupe de faire apparaitre l'objet au pied de Chronos-WC si un joueur est connecté à l'Era correspondante.

```
public void SpawnObjectFromWC(GrapableObject obj)
{
    if(activeEra.id == obj.era_id)
    {
        Debug.Log("Object recu par le WC Spawner avec le nom: " + obj.name);

        var prefab = objectsPrefabs.Find(o => o.name.Equals(obj.name));
        wcObjectSpawner = GameObject.Find("WcObjectSpawner").GetComponent<WcObjectSpawner>();
        wcObjectSpawner.SpawnObject(prefab);
    }
}
```

Le Spawner des chronos-WC instancie alors l'objet en y ajoutant quelques petites animations.

Donner un objet à un personnage non-joueur (NPC)

Il manque encore quelque chose au jeu pour qu'on puisse en faire une aventure. Il faut que lorsqu'on donne un objet au personnage, qu'il prenne l'objet, qu'il nous en donne un autre au besoin, qu'il réponde quelque chose et qu'il se déplace.

✓ Component PathFollower

Le **PathFollower** est un outil que l'on peut trouver dans l'asset store et qui permet de tracer un chemin qu'un **GameObject** va suivre. Nous allons ajouter un **PathFollower** à chaque NPC et lui fournir en paramètre la référence vers un GameObject qui est le dessin d'un chemin que nous avons placé dans l'éditeur et qui se nomme un **PathCreator**.

Nous allons désactiver le PathFollower via le check-box qui est à coté du nom de manière qu'il ne se déclenche pas directement à l'instanciation mais uniquement quand nous le désirons.

✓ Component ObjectReceiver

C'est un script que j'ai créé et qui permettra de donner tout un tas d'instructions au personnage via l'éditeur Unity.

ObjectToReceive est le nom du **GrapableObject** que le NPC accepte de recevoir.

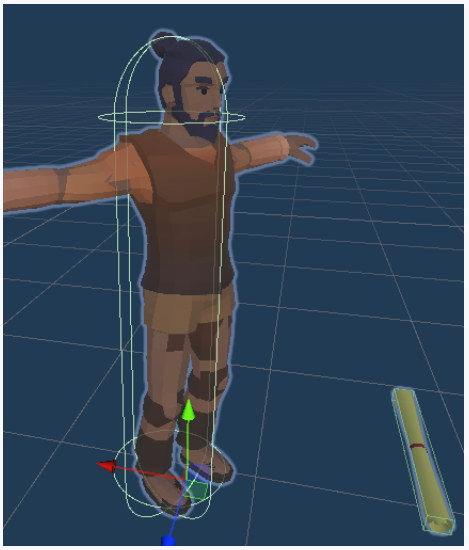
VerbalReaction est la phrase qu'il dira s'il reçoit le bon objet.

NegativeVerbalRéaction est ce qu'il dira si on lui donne le mauvais objet.

Si l'on référence une préfab dans le champ **ObjectToInstanciate,** cette dernière sera instanciée là où le NPC se tient.

Il se peut aussi qu'on ait prévue que l'objet que le NPC va nous donner était visible mais non-Grapable avant qu'il ne reçoive ce qu'il désirait. Dans ce cas nous allons glisser le GameObject qu'il faut masquer en cas de succès dans **ObjectToHide**.

- Fonctionnement du script



Comme on peut le voir, notre Npc a un **CapsuleCollider** qui est un Trigger et va donc déclencher une méthode **OnTriggerEnter** dans notre script.

Le script vérifie d'abord si le nom de l'objet qui a déclenché le trigger est le même que le paramètre **ObjectToReceive.**

Si tel est le cas il détruit ce GameObject et fait appel à l'interface via la méthode **DisplayDescription** que nous avions créée plus haut (voir page 32) en lui passant le nom du personnage et la réaction verbal qu'on a installé en paramètre **VerbalReaction**.

S'il y a un GrapableObject à détruire (le parchemin dans notre exemple), il le fait et instancie le nouvel Object a sa place. S'il n'y en avait pas il va instancier le parchemin à l'endroit où se tient le NPC.

S'il y a un chemin à suivre, le script Active le component **PathFollower** pour que le NPC se déplace vers l'endroit souhaité.

Créer une partie

Dans le chapitre « Voir la liste des parties » (voire page 42) nous avons créé l'écran sur lequel apparait la liste des parties ainsi qu'un bouton permettant de créer une nouvelle partie.

Ce bouton ne déclenche rien de très compliqué au niveau du Frontend. Il pointe vers une fonction **CreateNewGame()** du **GameManager** qui elle-même se contente de passer l'instruction au **NetworkManager** via sa méthode **CreateNewGame()**.

Le NetworkManager envoie alors une requête **createGameRequired** auquel le backend va réagir de la façon suivante.

```
socket.on('createGameRequired', function() {
    console.log("Game creation required");
    console.log("");

    var query1 = "INSERT INTO `game` (`name`,`isNew`) VALUES ('game', true )";

    sqlCon.query(query1, function (err1, result1, fields) {
        if (err1) throw err1;

        sqlCon.query("SELECT * FROM `game` WHERE `name` = 'game'", function (err2, result2, fields) {
            if (err2) throw err2;

            console.log("Partie trouvée : '" + JSON.stringify(result2) +"'")
            var newGame = result2[0];
            newGame.name = "Partie " + newGame.id;

            sqlCon.query("UPDATE `game` SET `name`='"+ newGame.name +"' WHERE `id` =" + newGame.id ,
                function (err3, result3, fields ) {

                if (err3) throw err3;
                CreateErasForGame(newGame.id);
            });

            socket.emit("gameCreated", newGame);
        });

    });
```

Le serveur va commencer par ajouter une entrée dans la table **Game** de la base de donnée en lui attribuant un nom provisoire. En effet, le nom d'une partie devant correspondre à « Partie » + l'ID, il n'est pas encore possible de lui donner un nom définitif puisque l'ID que la base de donnée auto-incrémentera nous est encore inconnu.

Nous allons en suite récupérer la partie nouvellement créée et ajouter en mofidier le nom pour le faire correspondre au schéma précité.

Il ne reste plus qu'à lancer la fonction **CreateErasForGame** en lui passant l'id de la partie pour que les trois Eras de la partie soient créées. Chacune d'entre elles sera ajoutée à la liste des Eras disponibles afin que les joueurs puissent s'y connecter.

Une fois ce travail terminé, le backend renvoie un message « **gameCreated** » avec la nouvelle partie pour fournir au jeu les données de la nouvelle partie créée.

Quand il reçoit le message, le front se contente d'ajouter un **GameBox** à la liste des partie en se servant de l'object Game qu'il vient de recevoir.

2.3 Petites touches finales

À ce stade, nous avons tous les éléments dont nous avons besoin, le jeu fonctionne et tout ses composants aussi. Toutefois il est encore un peu prématuré de parler de jeu. En effet, il n'a pas encore de but, ni même de trame.

Le scénario

Nous allons donc maintenant nous servir de tous les outils qui ont été créés pour réaliser un petit scénario. Comme vous l'aurez bien entendu compris, la lecture de ce paragraphe est rigoureusement interdite à tout professeur qui désirerait tester le jeu sans tricher lors de ma présentation orale.

✓ Le chemin vers la victoire

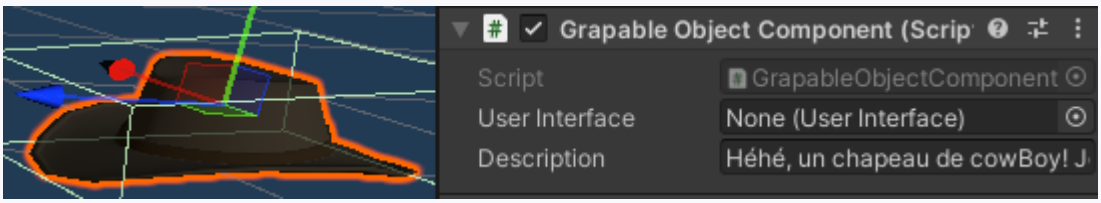
La trame de l'histoire sera la suivante

- ✓ Chez **Bernard,** le joueur récupère le **livre de cuisine** qu'il envoie chez Houagui.
- ✓ **Houagui** récupère le **livre** et le donne à **John Dœuf** qui cherche désespérément une idée de petit plat à cuisiner pour **George Washington** alors qu'il surveille la précieuse **déclaration d'indépendance**. Séduit par la recette des escargots à la mayonnaise ce dernier quittera la table à laquelle il était assis pour aller en cuisine, laissant sans surveillance la **déclaration d'indépendance**.
- ✓ **Houagui** prend la **déclaration d'indépendance** et l'envoie à Laverne dans le futur.
- ✓ **Laverne** récupère la **déclaration d'indépendance** et la confie à Hervé Concombre qui rêve de posséder un objet précieux à revendre pour devenir riche et quitter son boulot de gardien de la porte de la Tentacule Pourpre. Quand ce dernier reçoit la déclaration d'indépendance, il quitte son poste laissant derrière lui le **badge d'accès** au bureau.
- ✓ **Laverne** récupère le **badge d'accès** et s'en sert en le passant devant le terminal de la porte qui s'ouvre et lui donne accès au lugubre bureau de la Tentacule Pourpre.
- ✓ **Laverne** récupère la clé du bureau du docteur Fred que la Tentacule Pourpre gardait en souvenir de son vieil ennemi.
- ✓ **Laverne** envoie la clé à **Bernard** qui s'en sert pour ouvrir la porte du Docteur Fred et gagner le jeu.

Un fois que Bernard a mis la clé dans la serrure, nous aimerions qu'une petite musique retentisse et qu'une scène de générique soit lancée.

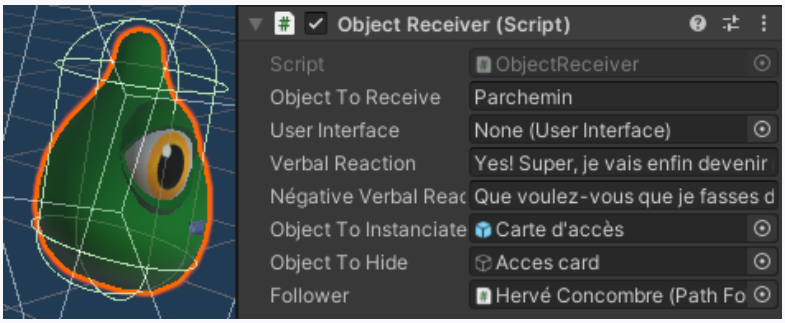
✓ Remplissage des données des GrapableObjects

Nous allons ouvrir chaque préfab de **GrapableObjects** et remplir le champs « Description » de leur GrapableObjectComponent.



✓ Remplissage des données des Npc

Nous allons faire exactement la même chose pour les NPCs.



De plus, nous allons compléter le component **ObjectReceiver** en prenant garde de bien *drag and drop* les GameObjects qui doivent éventuellement être supprimés et les Préfabs qui doivent être instanciées.

Nous lui écrivons également des réactions appropriées à l'histoire et un chemin à suivre.

✓ Ouverture de la porte du Bureau de la Tentacule Pourpre

Il nous faut maintenant faire en sorte que la porte s'ouvre quand la carte d'accès passe devant le terminal. Nous allons alors écrire un dernier petit script pour le GameObject **TentaculeOfficeGate**.

- **TentaculeOfficeGate component**

```
private void OnTriggerEnter(Collider other) {
    if(other.name == "Carte d'accès")
    {
        accessController.GetComponent<AudioSource>().Play();
        if (leftDoor.open == false)
        {
            OpenDoors();
        }
        else
            CloseDoors();
    }
}
```

Ce component connait les deux portes coulissantes et se contente, quand le bon GameObject déclenche le trigger qu'on a placé sur le terminal, de déclencher l'ouverture.

L'ouverture ou la fermeture n'est en fait qu'un simple déplacement des GameObjects **Door** à chaque frame jusqu'à ce qu'ils atteignent certaines coordonnées.

✓ Ouverture de la porte du Bureau du Docteur Fred

Une fois que la clé du bureau est en possession de Bernard, il faut désormais qu'il puisse l'introduire dans la serrure de la porte. Pour se faire nous allons utiliser un outil de Unity appelé les Sockets (à ne pas confondre avec les WebSockets).

- **Le socket de la serrure**

Le but ici est de faire en sorte que quand le joueur approche la clé à quelques centimètres de la serrure, cette dernière fasse apparaitre une image fantôme de la clé en place.

Si le joueur lâche la clé quand cette image fantôme apparait, la clé prend la position que le fantôme avait prédit et une méthode déclenche l'ouverture de la porte et la scène de FIN.

Voilà qui clôture la partie implémentation de Day Of The Tentacule VR, nous avons désormais un jeu fonctionnel, qui suit un déroulement logique et est stable.

3 Conclusions

Réaliser ce projet était vraiment une expérience formidable pour moi et son résultat symbolise parfaitement le but que je m'étais fixé en suivant mes études à l'EPFC.

J'ai toujours vu les développeurs davantage comme des créateurs, voire des artistes, que comme des techniciens. Il leur est souvent demandé de suivre une ligne directrice, voire de suivre les instructions de spécialistes en design et leur mission est certes de concevoir avant tout quelque chose de fonctionnel. Mais de cette nécessité de concevoir une application pratique, ergonomique et agréable découlera toujours un minimum de créativité et d'un certain sens de l'esthétique dans la conception.

En réalisant Dott VR, j'ai eu le plaisir de pouvoir travailler sur un projet où j'ai utilisé les outils que l'on m'a transmis en les mettant au service d'une création alliant esthétique, logique, technique et esprit ludique. J'ai découvert un domaine où l'appel à la créativité s'est montré plus présent et où j'ai pu allier le côté artistique à l'aspect technique bien plus que ne me l'auraient permis d'autres types de projet.

C'est par cet aspect que Dott VR clôture parfaitement ce Bachelier car il répond à la question que je m'étais posée en le commençant : « Vers quel domaine m'orienter après ? ». Par la réponse qu'il a apporté à mon envie sempiternelle d'allier créativité, technique et logique, ce projet m'a fait entrer dans un domaine du développement informatique dont j'imagine mal me lasser un jour tant il est stimulant intellectuellement et artistiquement.

La stimulation de ma curiosité et de mes envies de création fut par ailleurs source d'un arrière-gout amer qui me reste en bouche en cette fin de projet. En effet, le domaine est tellement vaste, englobe un tel panel d'améliorations possibles qu'il est frustrant de devoir être raisonnable en limitant le développement conformément aux contraintes de temps. Bien des améliorations pourraient encore être apportées à ce jeu :

- Pouvoir nommer les parties qu'on sauvegarde.
- Avoir des conversations à double sens avec les Pnj.
- Leur donner des voix.
- Avoir un design personnalisé pour chaque personnage.
- …

Autant d'objectifs que je garde en tête et que je me ferai sûrement un plaisir d'implémenter dans le futur dans une version non-basée sur la franchise **Day Of The Tentacule** et que vous retrouverez peut-être un jour, je l'espère, sur vos plateformes de jeux favorites.

Merci à tous ceux qui m'ont soutenus durant ce projet mais également pendant toute mes études : Julien PETRE, Pierre HALLOY, Anthony CAPELUTO et bien entendu et surtout mon père, Gérald DIELMAN.

Je lui dédie ce travail et ce cycle d'études que je lui dois pour m'avoir soutenu de toutes les manières qu'il est possible d'imaginer et m'avoir aidé à clôturer ce cycle d'études mais surtout à commencer une superbe nouvelle aventure professionnelle.

4 Bibliographie

Barnett, J. P. (s.d.). *Cinématique inverse en réalité virtuelle : comment utiliser Unity pour contrôler vos bras et vos mains*. Récupéré sur youtube: Cinématique inverse en réalité virtuelle : comment utiliser Unity pour contrôler vos bras et vos mains

Carnagey, A. (s.d.). Récupéré sur https://www.youtube.com/watch?v=tn3cWcSYmHE&list=PLH83kaN0EWK5wkiNhGlhfSI_nGDcR1DX8

nodeJs Fondation. (s.d.). Récupéré sur https://nodejs.org/en/

Robichaud, J. (s.d.). Récupéré sur https://docplayer.fr/14434777-Presentation-de-unity-3d.html

socket.io. (2022, 9 19). *Introduction*. Récupéré sur socket.io: https://socket.io/docs/v4/

Unity Technologie. (s.d.). Récupéré sur Unity Learn: https://learn.unity.com/

Unity Technologies. (2020, 09 30). *Coroutines*. Récupéré sur Unity Documentation: https://docs.unity3d.com/Manual/Coroutines.html