****

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**



Travail présenté par **Geoffrey DIELMAN**

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

Supervisé par : **Boris VERHAEGEN**

Table des matières

##### Table de matières numérotée

##### Introduction

* Résumé du sujet.
* Présentation du cadre du travail.

##### Analyse

* Description du problème posé (situation existante, situation finale désirée…).
* Etude préalable (discussion avec "l'utilisateur").
* Analyse (MCD, cas d’utilisation, diagrammes de classes, de séquence, d’activités…, présentation des différentes méthodes pour obtenir le résultat, contraintes, avantages et inconvénients des choix faits, …).
* Architecture de l’application (modèle en couches, MVC, …)

##### Réalisation

* Structures de données mises en jeu et/ou structure et organisation des fichiers.
* Structure des traitements : description des algorithmes essentiels.
* Description du fonctionnement du programme.

##### Appréciation personnelle

* Rapport avec le(s) cours.
* Difficultés rencontrées.
* Suggestions pour l'avenir.
* Utilité pour la vie professionnelle.

##### Conclusion

* Bilan des choix effectués et du résultat obtenu.

##### Bibliographie

* Référencée dans le texte et au format APA (<https://www.scribbr.fr/normes-apa/bibliographie-aux-normes-apa/>)

##### Annexes (si nécessaire)

* Listing (avec commentaires et spécifications). Privilégiez un support électronique (CD…) plutôt que papier
* Divers.

Introduction

Présentation du projet

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

Un petit bout d’histoire

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

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

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

But du jeu

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

Mécanique de Jeu

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

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

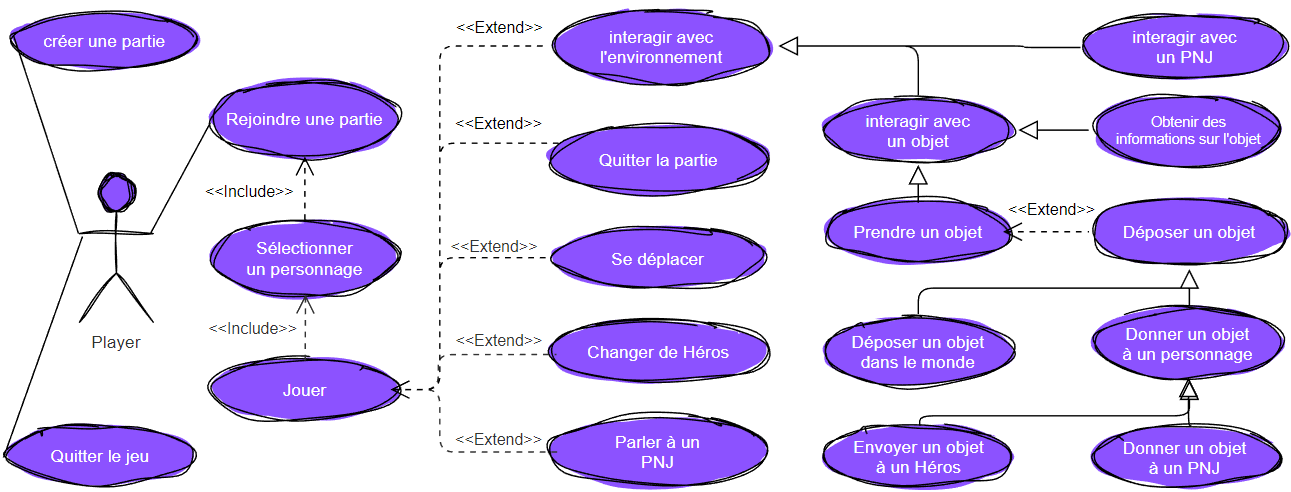
Fonctionnalités

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

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

Use Cases

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

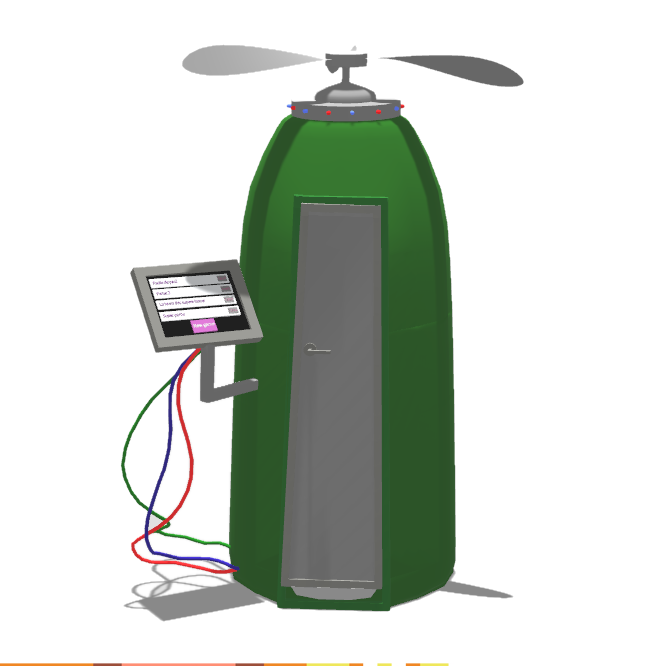


Lorsqu’il lance le jeu, le joueur arrive dans le menu principal. Ce dernier est représenté par un Chrono-WC et son tableau de commande. Le joueur peut à cette endroit *Quitter les jeu*, *créer une partie*, *rejoindre une partie* ou *sélectionner un personnage*.

* Quitter le jeu

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

* Créer une partie

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

* Rejoindre une partie

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

* Sélectionner un personnage

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

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

* Jouer

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

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

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

* Quitter une partie

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

* Sauver la partie

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

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

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

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

* Se déplacer

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

* Changer de Héros

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

* Interagir avec l’environnement

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

**- Interagir avec un pnj**

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

**- Interagir avec un objet**

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

* Obtenir des informations sur un objet

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

* Prendre un objet

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

* Déposer un objet dans le monde

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

* Envoyer un objet à un Héros

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

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

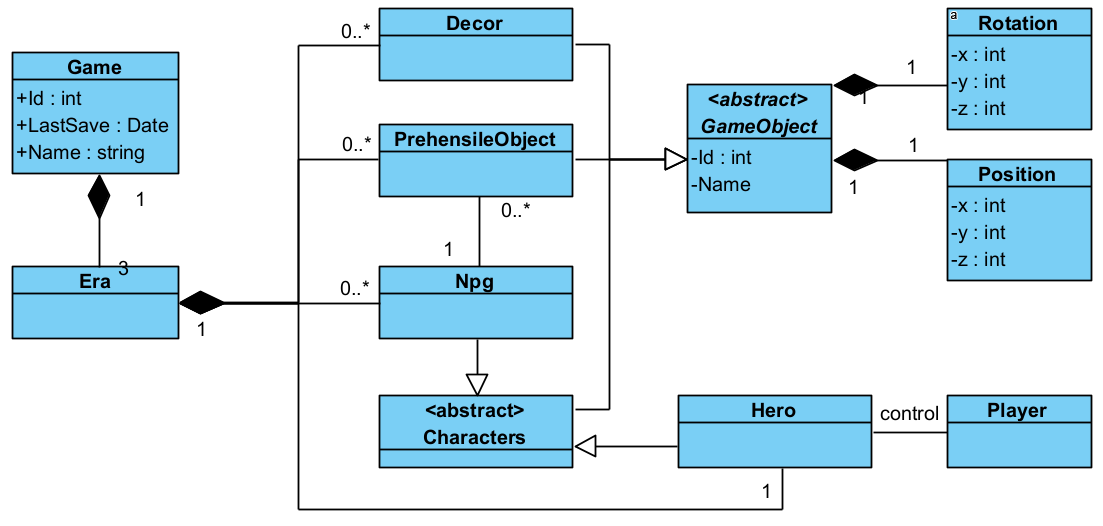
Une fois cette sélection effectuée par simple pression avec sa main virtuelle, le joueur peut jeter l’objet qu’il souhaite envoyer dans la cuvette du WC et l’objet est envoyé.

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

* Donner un objet à un PNJ

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

Diagramme de classe



* Game

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

**- Id**

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

**- isNew**

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

**- name**

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

**- lastSave**

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

**- eras**

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

* Era

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

Chaque époque contient:

**- Decors**

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

- **GrapableObjects**

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

* Gameobject

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

**- Id**

**- Name**

**- Rotation**

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

**- Position**

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

* Decor

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

* GrapableObject

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

* Npc

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

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

**- Inventary**

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

**- AcceptedObjects**

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

**- Dialogues**

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

Héros

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

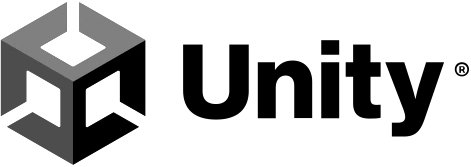
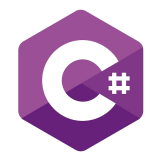
Architecture

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

Choix des technologies

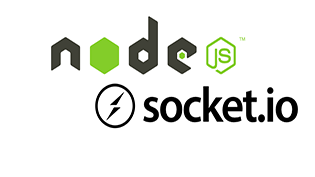
* Frontend : le jeu

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

* Backend: le serveur de jeu

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

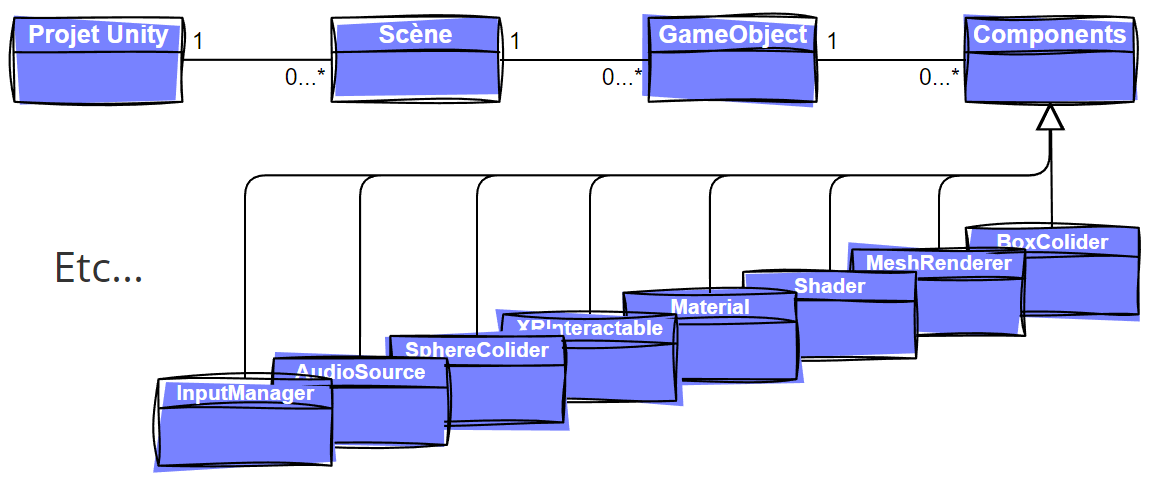


* Base de donnée



Apprentissage de la structure et de la programmation dans Unity.

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



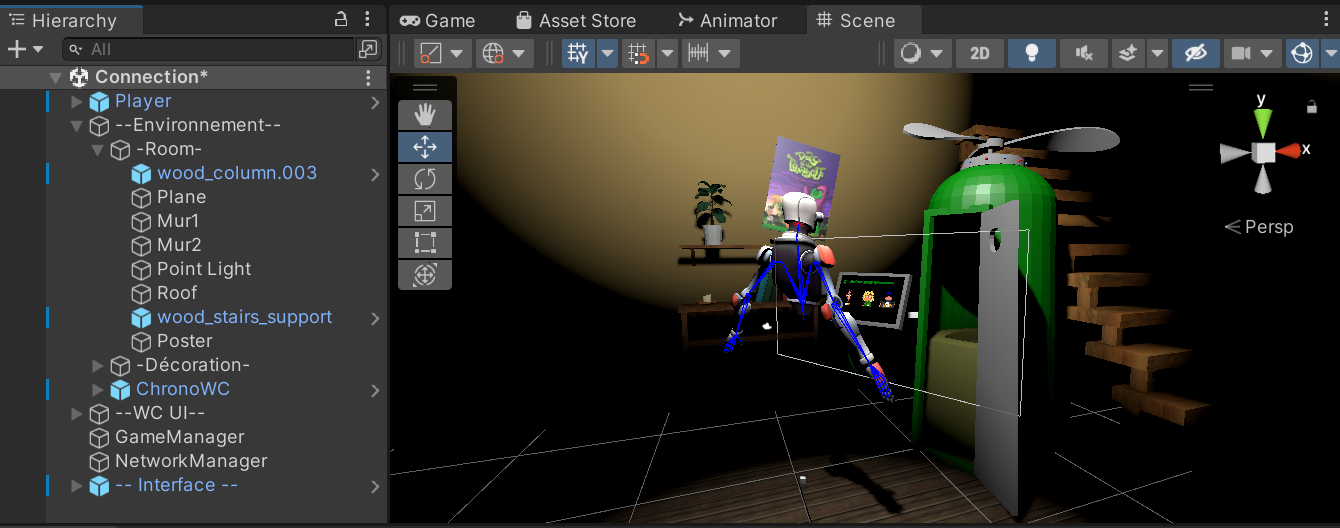
* Les scènes

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

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

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

* Les GameObjects



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

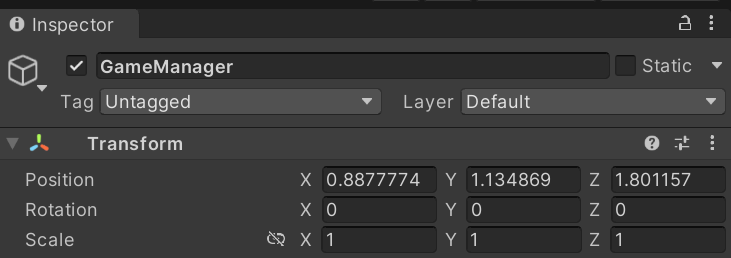
Partant de ce principe, on pourrait comparer un *GameObject* à une Vue dans le model MVC. Ce n’est toutefois pas rigoureusement exact. Car si un *GameObject* peut avoir un rendu visuel (un personnage, une maison, un monstre, un objet sur la table, etc...) ce n’est pas nécessairement le cas. Certain d’entre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

**Le GameManager**

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

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

**Les tags**

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

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

**Active**

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

**Les préfabs**

<<Parler des préfabs ici>>

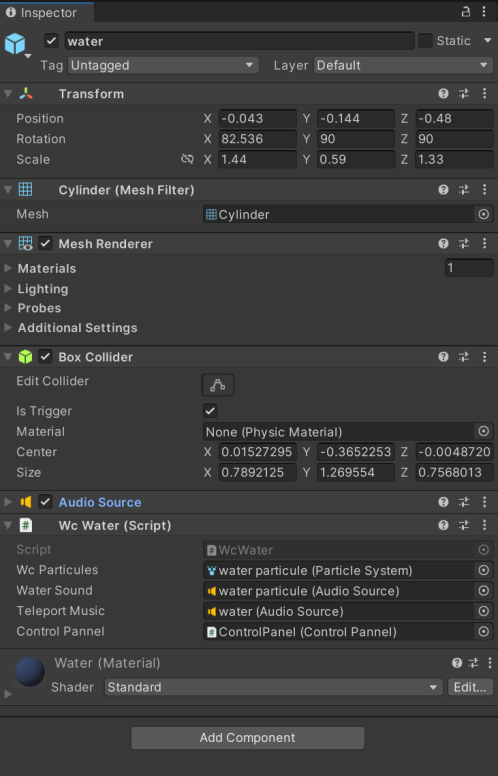
* Les Components

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

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

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

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

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

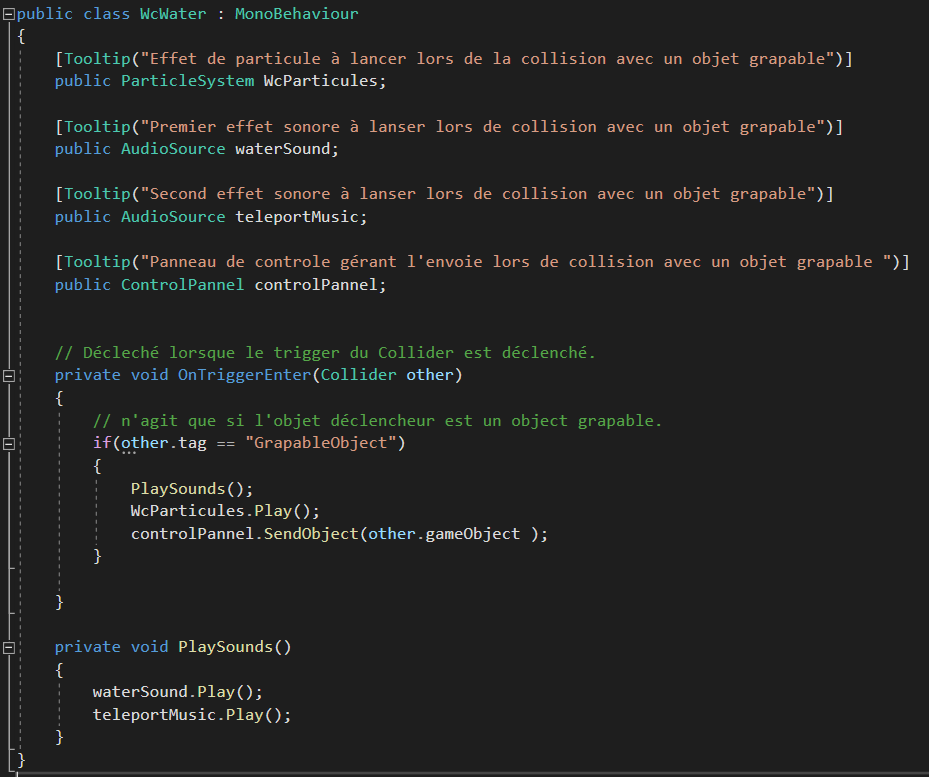
* Le *Transform* commun à tout GameObject
* des meshs pour le rendu visuel,
* un box collider qui sert à détecter la colision de l’eau avec un GameObject,
* une source audio qui permet de lancer un petit effet audio
* un script personnalisé “*WcWater.cs*” qui gère le tout.

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

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

**- Les scripts vue de plus près**

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



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

**Les méthodes**

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

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

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

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

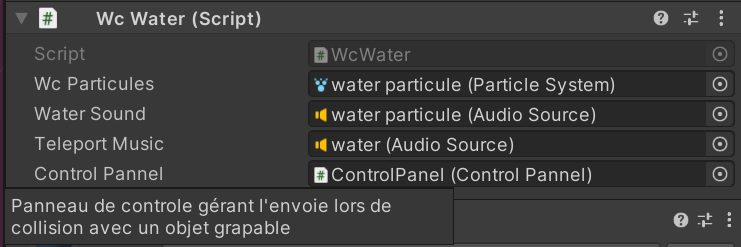
* ***Start()*** est appelé lorsqu'un script est activé juste avant que l'une des méthodes Update ne soit appelée pour la première fois.

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

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

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

**Les propriétés**

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

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

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

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

* Conclusion de l’analyse de la structure de Unity

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

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

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

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

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

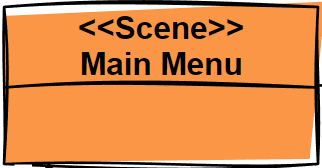
**Développement**

Diagramme de structure d’implémentation

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

La structure du diagramme

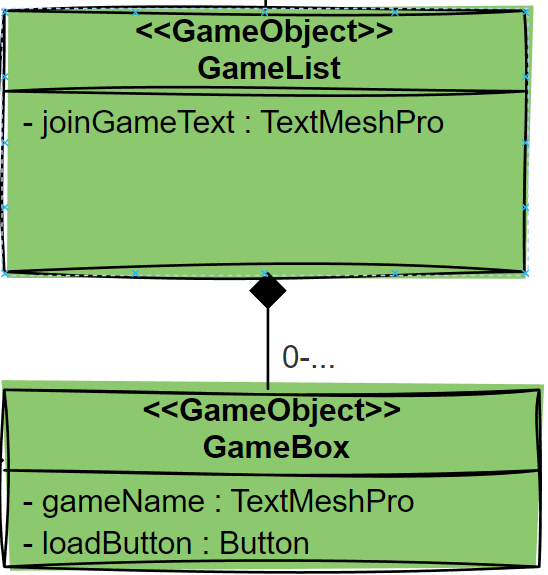
* Les scènes

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

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

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

* Les GameObjects

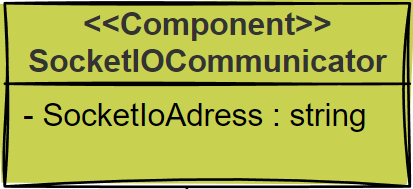
Ils sont en vert et portent la balise <<GameObject>>. Comme pour les scènes le nom que je donne à chacune des case “GameObject” du diagramme corespondra à un Tag. A la différence des scènes, les Tag ne sont plus seulement théoriques car ils sont bel et bien présents dans Unity, comme nous l’avons vu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

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

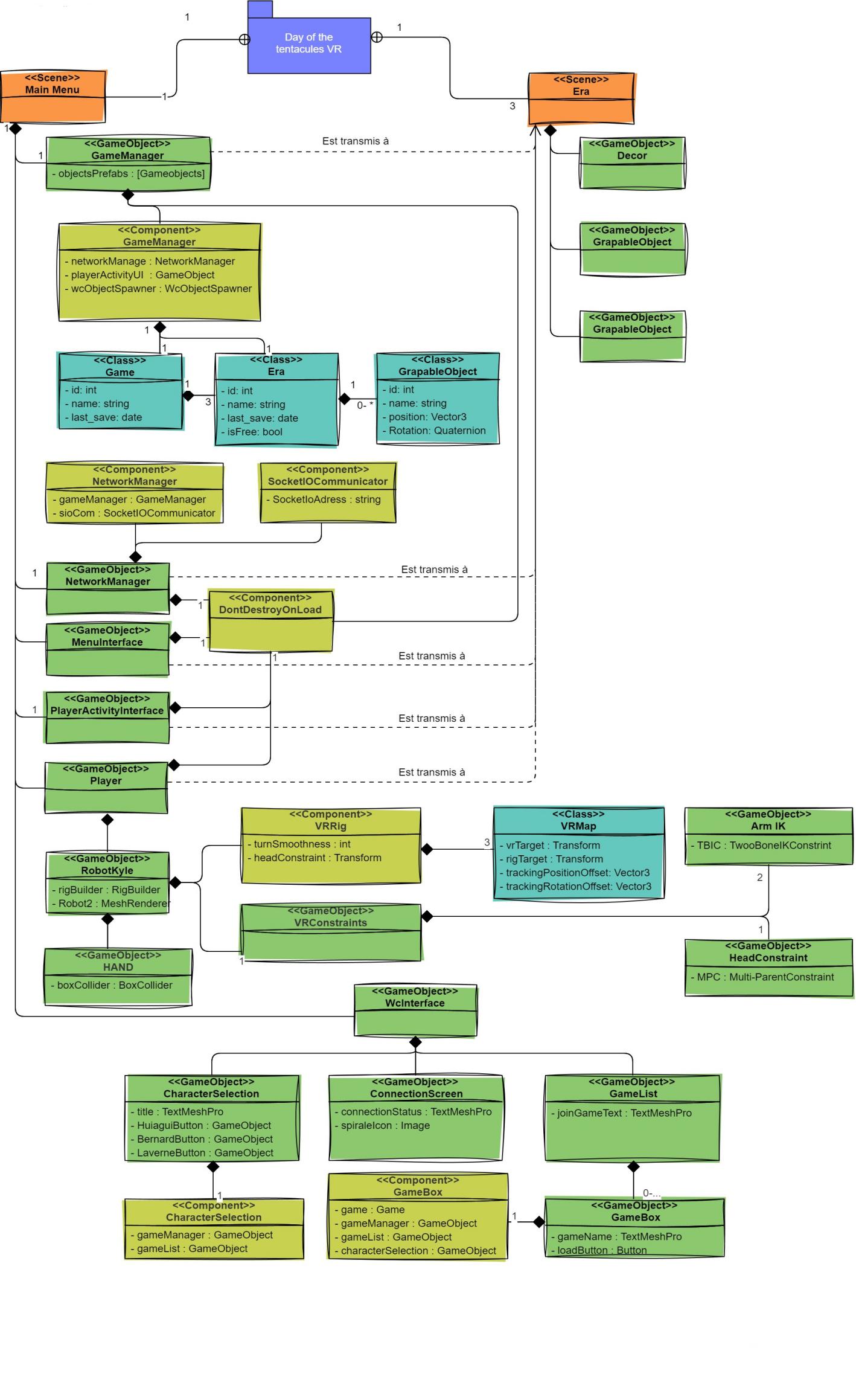
Chacun d’eux a aussi des ***Components*** mais nous verrons ça dans le chapitre suivant

* Les Components

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

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

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

Le diagramme

Le processus d’implémentation

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

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

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

Le graphisme

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

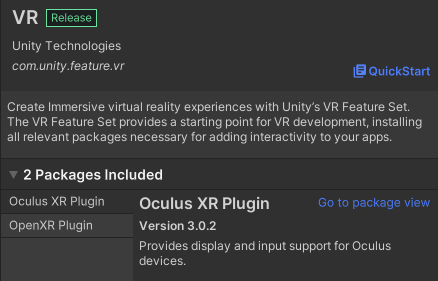
J’ai toutefois décidé de m’intéresser quelque peu à la construction d’éléments graphiques. En effet, il existe une relation étroite entre la partie graphique d’un objet et la programmation de ce dernier. Apprendre sommairement de quoi se compose une préfab m’a permis de comprendre comment l’on pouvait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

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

Moove character

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

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

* Occulus XR Plugin

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

* OpenXRPlugin

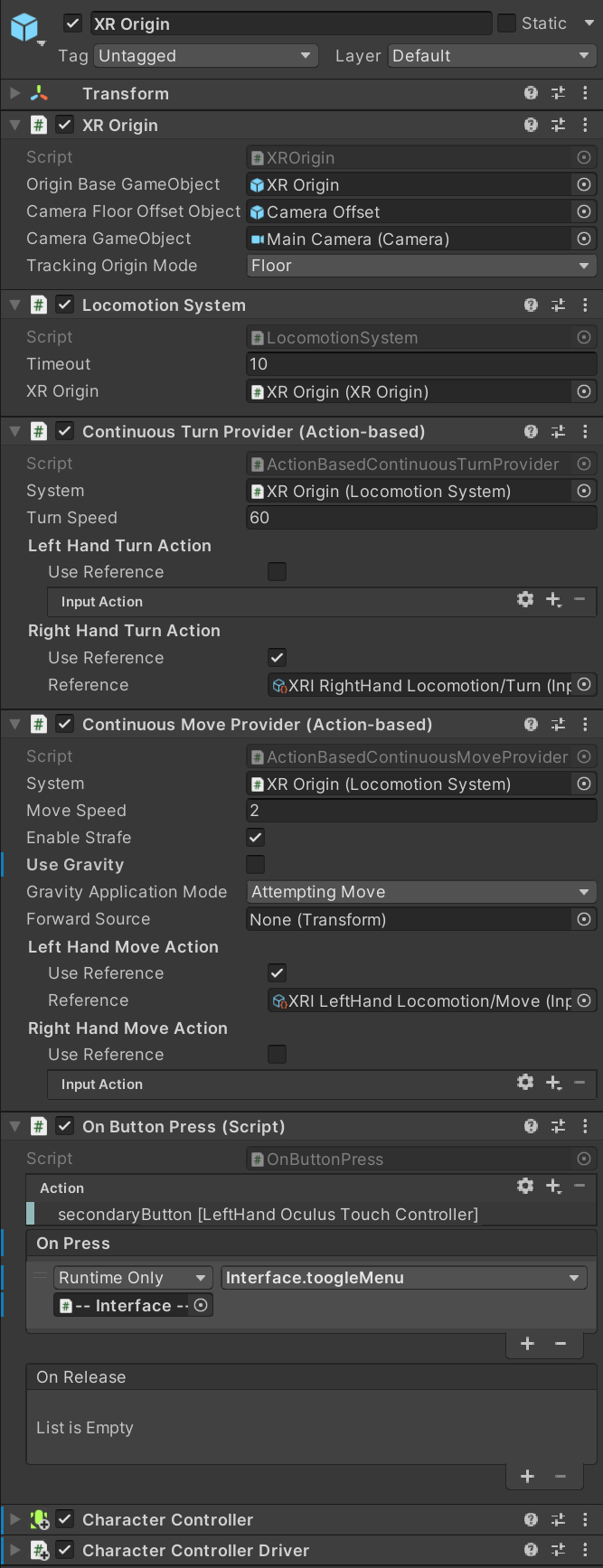
OpenXR est une norme ouverte et libre de droits développée par Khronos qui vise à simplifier le développement AR/VR en permettant aux développeurs de cibler de manière transparente une large gamme d'appareils AR/VR.

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

Regardons d’un peut plus près le ***GameObject XROrigin***.

* XROrigin

Voici la composition du GameObject XROrigin :



* Un component **XR Origin**

Il a comme paramètres:

- Son gameObject parent

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

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

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

* Un component **Locomotion System**

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

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

* Un component **Continuous Turn Provider**

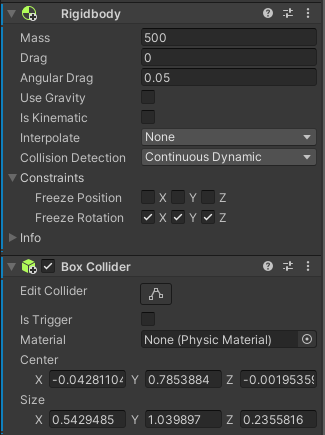
Permet à l'utilisateur de faire pivoter en douceur le **XR Origin** en continu au fil du temps à l'aide d'une action d'entrée spécifiée.

* Un component **Continuous Move Provider**

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

* Un component **Onbutton press**

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

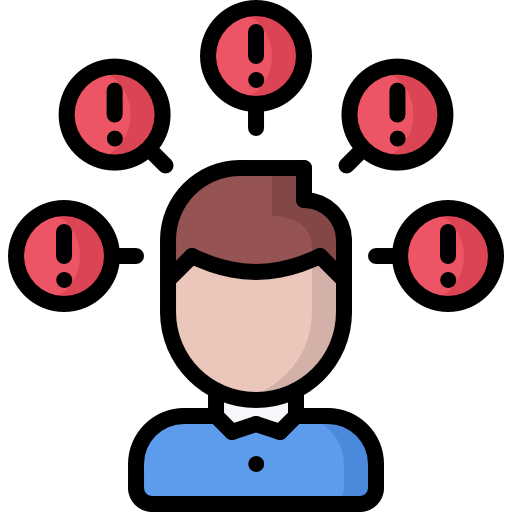
*  Un component **Box Collider**

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

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

* Un component **Rigid Body**

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

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

Problème rencontré

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

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

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

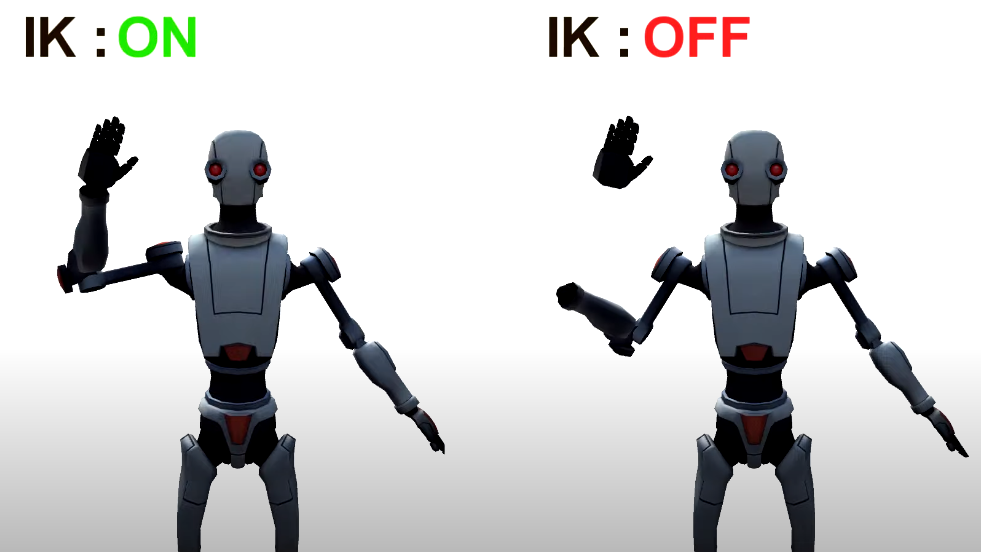
L’étape du **XR Origin** est instancié et nous pouvons désormais bouger sans passer à travers les murs. Toutefois, il est maintenant nécessaire d’afficher un personnage et de faire en sorte que ce dernier bouge avec en synchronisation avec le personnage.

* Robot Kyle

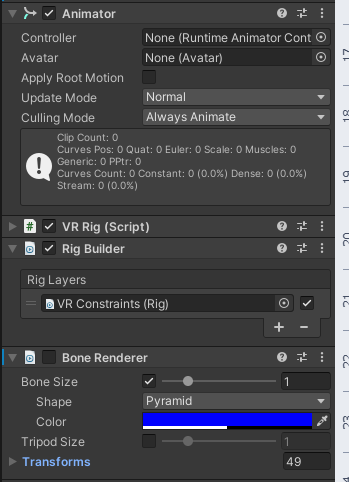
Comme précisé plus haut nous allons utiliser un personnage préfabriqué qui est animable. Pour qu’il le soit, il doit avoir un **Rig** c’est a dire un squelette qui, lorsqu’il bouge déforme de **mesh** de rendu de manière à ce que les mouvements soient harmonieux.

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

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

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

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

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

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

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

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

Maintenant que nous avons le point d’entrée de nos contraintes , nous allons ajouter trois GameObjects qui correspondent aux trois contraintes qui nous intéressent.



**A placer quelque part**

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

Déroulement du projet.

- apprentissage de Unity

A duré longtemps

Suivi des cours de Unity learn: *<https://learn.unity.com/>* deux mois.

Les étapes suivies

Choix faits et structure générale de l’application

Backend - frontEnd

Problèmes rencontrés

- les sockets

- les murs

- le build

- lenteur de l’ordi

- soucis de visualisation du fonctionnement général avant implémentation des GOs

- Nécessité d’apprendre blender

- construction de Bernard abandonnée

-> GravitySketch problème de rendu dans Blender

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

->apprentissage des rigs

- Problème destruction du Network Manager

-> problème pour Sauver/Quitter, plantage occasionnel

- Télécommande qui ne fonctionne plus, problème de référencement

- Plantage serveur au démarrage alors que le Front est démarré

- Problème de persistance des données du Game Manager

-> Don’t destroy onload

-

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

Bibliographie

Unity Learn

Bibli unity