Dialogue system - Notice

Le 05-06-2021 par Alexandre Venet https://twitter.com/alhomepage

1. Introduction

Ce **système de dialogue** permet la gestion rapide de dialogues apparaissant en boîte de dialogue UI, en réduisant le besoin d'écrire des scripts. Ce système est pensé pour être intégré dans chaque scène (s'il n'est pas requis dans tel niveau, alors il n'est pas à intégrer). Il est pensé pour des dialogues linéaires et n'a pas vocation à gérer des embranchements (il les permet néanmoins mais dans une certaine mesure). Enfin, un dialogue en cours d'affichage ne bloque pas le flux du jeu (pas d'effet de pause ou gel d'action).

Il requiert un ensemble de **fichiers** à installer : son **noyau de scripts** (*MonoBehaviour*, *ScriptableObject*, classes de types), les **interfaces utilisateur pour l'éditeur Unity** (*Editor*, *EditorWindow*), les **éléments d'UI pour le produit final** (*Prefabs*), une scène d'exemple.

Il dépend du package *Cinemachine* pour les caméras virtuelles et de *TextMeshPro* pour les textes d'UI et leur mise en forme.

Une fois les fichiers installés et selon les besoins, on créera autant de fichiers de **personnages**, de **dialogues** et de **progression** et autant de composants de **références de dialogues** qu'il est nécessaire.

Le principe général est le suivant. Des objets présentent le composant **DialogueRef** qui définit le **type d'interaction** et les **conditions** de **déclenchement** de **dialogue**. L'**objet jouable** ou tout autre objet teste le type d'interaction et selon ce dernier appelle **DialogueManager** en lui envoyant une référence de **DialogueRef.** Ce dernier à son tour teste les conditions de **DialogueRef** : si des conditions sont remplies, un dialogue correspondant est lancé.

Les fichiers les plus importants sont **mis en forme** par **scripts d'éditeur** pour le rendre le travail d'édition confortable. Passer le curseur sur les noms fait apparaître la plupart du temps **une bulle d'informatio**n. Les noms qui apparaissent en édition peuvent être différents des noms de variable réels, ceci à des fins de lisibilité. J'utilise dans le présent document les noms tels qu'ils apparaissent dans *Unity*.

Le **menu** d'*Unity* se voit augmenté de l'entrée suivante : **Tools > Reliquia > Dialogue**. On y trouvera des liens sélectionnant les fichiers d'aide.

La scène de test fournie a ce chemin : **Alexandre_SceneTest/DialogueTest.unity**.

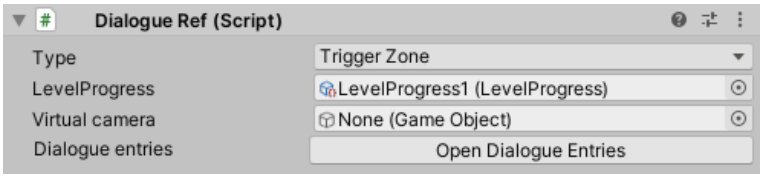
2. Objet jouable

Un dialogue fait l'objet de **déclenchements** : soit lorsque le joueur entre dans une **zone qui génère automatiquement l'événement** d'affichage, soit lorsque le joueur **décide d'interagir**.

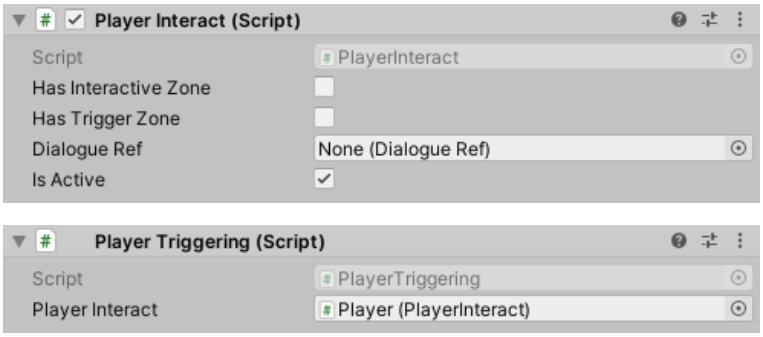
Ces deux comportements **dépendent** de **l'objet jouable**, du *gameplay* visé, etc.

Le parti-pris de développement pour la scène de test a été d'utiliser la **détection de collisions** (*Collider* de type *Trigger* et méthodes *OnTriggerEnter, OnTriggerExit*). Le principe est le suivant.

D'un côté, des **objets de déclenchement de dialogues** sont des objet 3D d'une certaine forme avec un *Collider Trigger*. On leur attache le composant **DialogueRef**. Ce dernier propose un champ **Type** qui définit soit une zone automatique (**Trigger Zone**), soit une zone interactive (**Interactive Zone**).



De l'autre côté, **l'objet jouable** présente un composant **PlayerInteract**, et contient un objet enfant nommé *Trigger* qui présente lui-même un composant *Collider* de type *Trigger* et le composant **PlayerTriggering**.

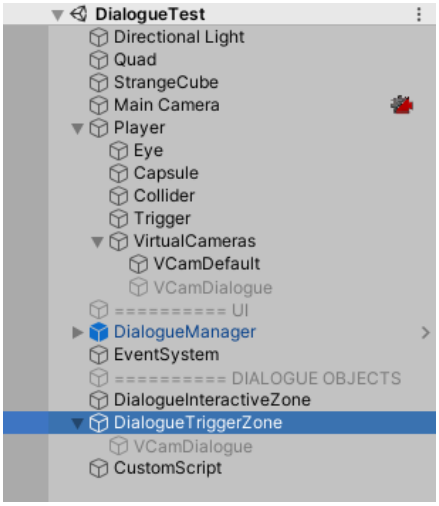


Lorsque le joueur entre dans la zone, le composant **PlayerTriggering** informe le composant **PlayerInteract**. Ce dernier évalue la valeur du champ **Type** du **DialogueRef** et réagit selon les cas : si zone automatique, alors lancement du dialogue automatiquement ; si zone interactive, alors le joueur doit presser le bouton gauche de la souris pour lancer le dialogue.

Maintenant, noter que la seule condition relative aux dialogues, à ce niveau, est d'utiliser **DialogueRef** et son champ **Type**. **PlayerInteract** et **PlayerTriggering** ne sont que des exemples mais contiennent des instructions nécessaires pour prendre en charge **DialogueRef** et l'appel à **DialogueManager**. Liberté est laissée de coder les déclenchements autrement selon le *gameplay* visé (c'est même conseillé car en l'état, les scripts d'exemple permettent de déclencher un objet en lui tournant le dos !).

Un autre aspect de l'objet jouable est la **caméra**. Le projet *Reliquia* utilise le package *Cinemachine*. Ce package permet des **transitions** entre caméras ; ainsi, lorsqu'un dialogue est lancé, il est possible de sélectionner une **caméra virtuelle** installée dans le décor de façon pertinente et qui sera le point de vue utilisé pendant le dialogue. Or, une telle caméra spécifique peut ne pas être utilisée ; par conséquent, une caméra virtuelle de dialogue **par défaut** doit être prévue et définie.

Il est laissé libre la façon de disposer des caméras. Dans la scène de test, la caméra de dialogue par défaut est intégrée à l'objet jouable, ce qui peut ne pas être suivi ; on peut utiliser une caméra ou plusieurs caméras selon les besoins. Précisons. Ouvrons la scène de test. Regardons la *Hierarchy* de la scène. la **Main Camera** comprend le composant **CinemachineBrain**, la caméra du **Player** utilisée pour le déplacement dans l'espace 3D fait l'objet d'une **caméra virtuelle** (**VCamDefault**) et la caméra par défaut pour le dialogue est aussi une **caméra virtuelle** (**VCamDialogue**). Ces caméras ont été intégrées dans le **Player**, suivant l'idée de réaliser ultérieurement un *Prefab* ayant tous les éléments utiles déjà renseignés.



3. Déclencheurs et entrées de dialogues

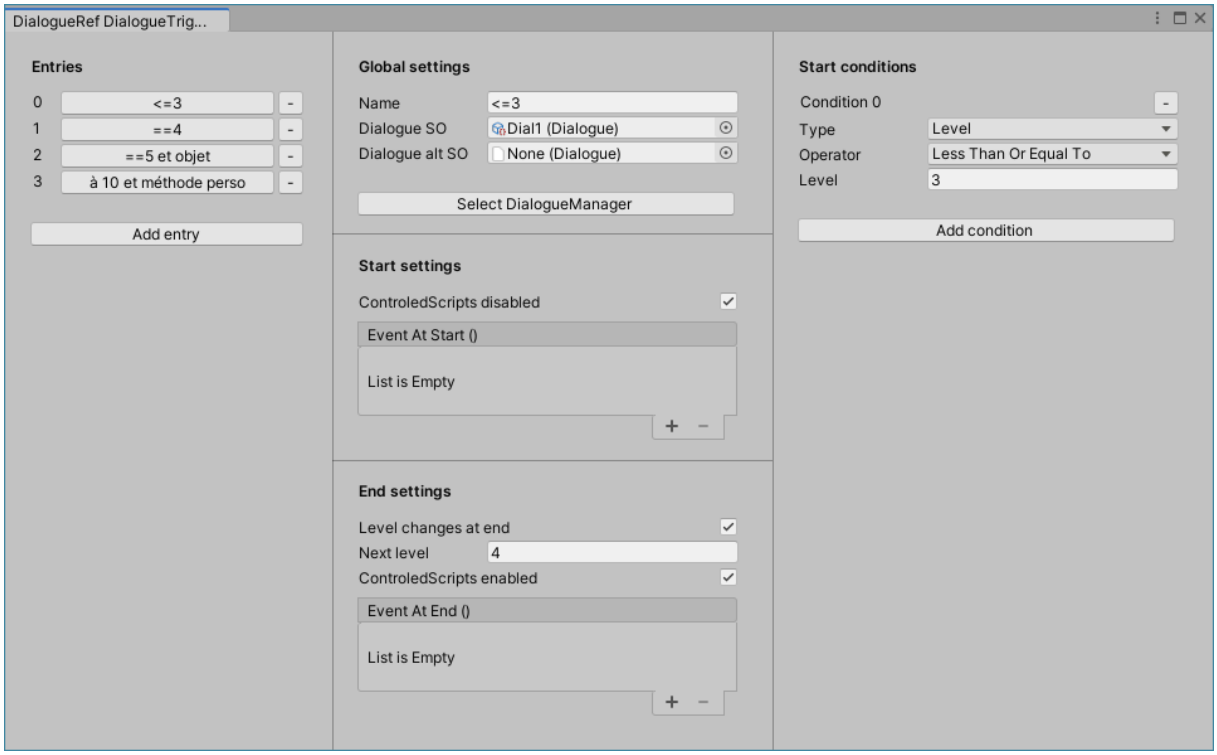
DialogueRef définit un ensemble **conditions de déclenchement** d'un dialogue. Il propose d'un côté des **paramètres généraux et locaux**, et de l'autre des **DialogueEntries** (entrées de dialogue).

Le champ **Type** définit la catégorie de **déclenchement** : zone interactive ou zone de déclenchement automatique. Ce champ est utile pour l'objet jouable qui sera programmé selon les besoins.

Le champ **LevelProgress** accueille un fichier *ScriptableObject* de **progression**. Ce fichier est nécessaire. On le crée par **clic droit dans *Project* > Create > ScriptableObjects > Dialogue > LevelProgress**. Ce fichier contient un nombre entier signifiant l'étape actuelle (**LevelStep**) de la mission, de la quête, de l'aventure, et une valeur initiale (**InitValue**) s'il s'avère utile de commencer avec une valeur autre que zéro. Ce fichier est donc corrélé à une narration ; par conséquent, on peut créer autant de fichiers de progression qu'on dispose d'axes narratifs, ce qui laisse la possibilité de penser des quêtes annexes, alternatives… Toutefois, le **LevelProgress** renseigné dans un **DialogueRef** est celui qui subira les tests de déclenchement ; je conseille de ne pas le remplacer, même dynamiquement. En effet, s'il est remplacé par un autre, alors les conditions de déclenchement ne correspondront plus à la narration souhaitée et la narration risque de présenter des valeurs inattendues. Pour gérer des axes narratifs différents, on créera donc d'autres objets **DialogueRef** que l'on gérera selon les besoins.

Le champ **Virtual camera** accueille une **caméra virtuelle** du package *Cinemachine*. Actuellement, il existe **UNE caméra** pour **UN DialogueRef**, ce qui signifie un seul point de vue pour cet objet. Néanmoins, il est permis de renseigner dynamiquement n'importe quelle caméra dans ce champ ou de programmer le comportement de cette seule caméra, pour varier les points de vue. Laisser le **champ vide** a pour conséquence d'activer la caméra de dialogue **par défaut**.

Cliquer sur le bouton « **Open Dialogue Entries** » pour lancer la fenêtre d'édition des entrées de dialogue. La fenêtre s'ouvre par défaut au milieu de l'écran. Elle peut être repositionnée et ancrée. Limitation : le mode *Play* ferme automatiquement la fenêtre (sinon il y a perte de référence et batterie de messages d'erreur ; sans dommage sur le contenu mais perturbant).



Cliquer sur le bouton « **Add entry** » pour **ajouter** des entrées de dialogue. Cliquer sur le **nom** pour afficher les informations relatives à l'entrée ; cliquer sur le bouton **supprimer** pour anéantir à jamais l'entrée. Attention, l'ordre est **fixe** : c'est l'ordre dans lequel vous voudrez que **DialogueManager** lise le contenu.

Mais qu'est-ce qu'une entrée de dialogue ? C'est un ensemble de conditions à remplir, relatifs à un dialogue particulier. On y édite non pas le dialogue mais sa **référence**, ses **conditions de déclenchement**, ses **paramètres de démarrage** et **paramètres de fin**.

Voyons d'abord les **paramètres généraux** (**Global settings**).

Le champ **Name** propose un **nom** pour l'entrée sélectionnée. Je conseille d'y inscrire les conditions de déclenchement (exemple : « <= 3 » ou « à 12 »). Rappel : les conditions de déclenchement portent sur le **LevelProgress** défini plus haut.

Le champ **Dialogue SO** accueille le fichier *ScriptableObject* de **dialogue** proprement dit ; ce champ ne peut pas rester vide.

Le champ **Dialogue alt SO** accueille aussi un *ScriptableObject* de dialogue ; ce champ peut rester vide (voir plus bas pour plus d'information sur ce champ).

Le bouton « **Select DialogueManager** » permet de lancer la sélection de l'objet idoine en *Inspector,* si présent dans la scène (et il doit l'être).

Voyons maintenant les **paramètres de démarrage** (**Start settings**). Ici, on contrôle ce qui se passe au lancement du dialogue.

Le champ **ControledScripts disabled** est de type booléen. Si la case est cochée, alors on souhaite que **DialogueManager** intervienne sur certains objets préalablement référencés et les **désactive**. Si la case est décochée, alors **DialogueManager** n'interviendra pas sur ces objets (voir le chapitre sur ce composant).

Le champ **Event At Start()** est un *UnityEvent* qui accueille des méthodes à abonner. Cela permet de déclencher des opérations, par exemple ouvrir une fenêtre au lancement du dialogue.

Voyons ensuite les **paramètres de fin** (**End settings**). Ici, on contrôle ce qui se passe lorsque le dialogue se termine.

Le champ **Level changes at end** est de type booléen. Si la case est cochée, alors on souhaite que le niveau de **LevelProgress** change. Sinon, ce niveau ne sera pas modifié.

Si la case précédente est cochée, alors il convient de renseigner une valeur pour changer le niveau de **LevelProgress**. À la fin du dialogue, le niveau de mission ou de quête sera celui renseigné. Il est permis de mettre des valeurs de progression déjà effectuées, ceci pour par exemple emprisonner le joueur dans des labyrinthes narratifs en boucle… mais, ami maître du jeu, n'oubliez pas des conditions de sortie (on modifiera les valeurs correspondantes par script car ces variables sont publiques).

Le champ **ControledScripts enabled** est de type booléen. Si la case est cochée, alors on souhaite que **DialogueManager** intervienne sur certains objets préalablement référencés et les **réactive**. Si la case est décochée, alors **DialogueManager** n'interviendra pas sur ces objets (voir le chapitre sur ce composant).

Le champ **Event At End()** est un *UnityEvent* qui accueille des méthodes à abonner. Cela permet de déclencher des opérations en fin de dialogue, par exemple fermer la fenêtre ouverte qui génère des courants d'air.

Voyons enfin les **conditions de démarrage** (**Start conditions**). Ce sont les conditions à remplir pour que le dialogue soit lancé. Comment cela fonctionne-t-il ? D'abord, il suffit de cliquer sur le bouton « **Add**

condition » pour ajouter autant de conditions qu'on le souhaite. Un bouton « - » permet l'annihilation absolue de la condition choisie.

DialogueManager lit toutes les entrées de dialogues. Pour chacune, il teste les conditions. Par défaut, une **condition non fournie** a pour conséquence **une valeur toujours vraie**. Par conséquent, le dialogue afférent sera toujours affiché ; utile par exemple pour gérer les répliques d'un perroquet. En revanche, de cette manière il est impossible de passer à d'autres dialogues car, pour cela, il faut éditer des conditions (changer la valeur de **LevelProgress** est inutile si cette valeur n'est pas testée). Enfin, les conditions pour une même entrée de dialogue sont **cumulées** sur le modèle du ET logique.

Les conditions de démarrage se présentent sous trois **types**. On choisit le type dans le menu déroulant. Voyons cela.

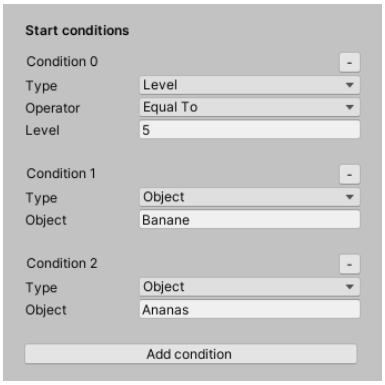
D'abord, le type **Level** est le type par défaut. Il s'agit de choisir une condition d'évaluation du niveau de **LevelProgress**. Par exemple, si la valeur doit être supérieure ou égale à 4, alors on choisit l'**opérateur de comparaison** (**Operator**) « **Greater Than Or Equal To** » et on écrit le nombre 4 dans le champ **Level**.

Noter ici que si la condition n'est pas remplie, alors il n'y a pas d'alternative et **DialogueManager** passera à une autre entrée de dialogue pour laquelle il reprendra son évaluation. S'il s'agit de créer des dialogues « d'attente » ou de « redirection vers une tâche », alors il convient d'établir une étape de narration spécifique de cette situation. Par exemple : si **LevelProgress** est à 4, alors lancer le dialogue « Je ferais mieux de vérifier mes pâtes », puis changer **LevelProgress** à 5 lorsque le personnage voit sa cuisine dévastée par la mousse, enfin tester « >=5 » ou « >4 » pour lancer le dialogue « C'est la fin du monde ! ».

Grâce au cumul des conditions, on peut complexifier le test. Pour cela, il suffit d'ajouter une autre condition de type **Level**. On peut ainsi réaliser des intervalles, par exemple : entrée 1 à >3 && <=12 et entrée 2 à >12.

Pour finir avec le type **Level**, je conseille de l'utiliser comme condition de base car c'est grâce à lui que les dialogues sont liés à une progression narrative.

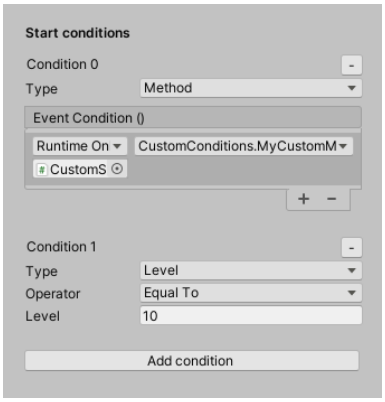
Ensuite, le type **Object** propose un champ où renseigner un objet requis pour lancer le dialogue. Comme précédemment, on ajoute autant de conditions d'objet qu'on le souhaite ; **DialogueManager** évaluera si ces objets sont dans l'inventaire ou non.



Actuellement, l'inventaire est un fichier *ScriptableObject* contenant une collection de *strings*. Pour créer cet inventaire vestimentaire, aller dans ***Project*** **puis clic droit > Create > ScriptableObjets > Dialogue > Inventory**. **Ce fichier doit être modifié selon le type d'inventaire réalisé**. Par exemple, on peut imaginer que le champ de saisie texte laisse la place à une référence d'*Asset* (*ObjectField*).

Utiliser les conditions d'objet permet d'utiliser des **dialogues alternatifs**. Le dialogue alternatif se lance si la condition d'objet n'est pas remplie. Le dialogue alternatif concerne le champ **Dialogue alt SO** vu plus haut. Il n'est pas nécessaire de le renseigner ; alors si la condition n'est pas remplie, aucun dialogue n'est lancé. Pour illustrer, voici l'exemple suivant. À l'étape 5, je découvre ma cuisine dévastée par la mousse ; je retourne dans mon garage et je parle à la porte coulissante. Si j'ai le canard dans l'inventaire, alors la condition est remplie sinon, toujours à 5, la porte attend que j'aie le canard. Donc, **LevelProgress** est à 5 mais la condition d'objet n'est pas remplie ; la porte m'informe néanmoins de quelque chose, comme toute bonne porte. Noter que **Level changes at end** ne porte pas sur le dialogue alternatif mais sur le dialogue « principal », c'est-à-dire celui pour lequel les conditions sont remplies (champ **Dialogue SO**).

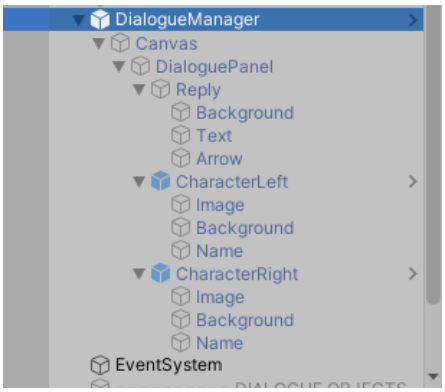
Enfin, le type **Method** propose un *UnityEvent* nommé **Event Condition** pour lequel on peut abonner une méthode (ou plusieurs) qui réalise les opérations de comparaison. Ceci peut s'avérer utile dans le cas où les options précédentes sont insuffisantes et que l'on souhaite réaliser des tests plus spécifiques (par exemple : savoir si l'invasion de mousse de pâtes s'effectue à une certaine vitesse dans le couloir tandis que la porte boude). Dans ce cas, il convient que la méthode informe **DialogueManager** du résultat ; ceci s'effectue au moyen de l'instruction suivante qui consiste à ajouter à une liste la valeur de sortie du test : **DialogueManager.Instance.m_conditionsBools.Add(valeurBooleenne);** (valeurBooleene est le résultat *true* ou *false*). Chaque méthode abonnée ajoutera donc sa valeur à la liste.



Je conseille de **ne pas envoyer plus d'une valeur par fonction abonnée** à la liste de **DialogueManager**, bien qu'il soit permis d'en envoyer plusieurs (ce qui peut s'avérer utile si la situation est complexe).

Enfin le dialogue alternatif peut être pris en compte dans les conditions personnalisées car **DialogueManager** teste **m_conditionsBools** en toute fin de procédure.

4. Gestionnaire de dialogue



Le **Dialogue Manager** est un *Prefab* à instancier dans la scène. Il est nécessaire pour que le système de dialogue fonctionne.

Il contient le composant **DialogueManager** servant à la gestion des dialogues et son arborescence présente un *Canvas* comprenant tous les objets de l'interface graphique.

Une fois dans la scène, il doit être accompagné d'un *EventSystem*. Si ce dernier n'est pas présent, penser à l'ajouter ainsi : **clic droit dans *Hierarchy* > UI > Event System**.

Le composant **DialogueManager** est un **Singleton**, c'est-à-dire que l'ajouter à la scène fait ainsi de cette instance la référence unique, ce qui convient aux besoins du système de dialogue.

Le composant présente un certain nombre de champs **déjà renseignés** et d'autres **à renseigner** (leur texte apparaît en gras et un liseré de couleur bleue accompagne la ligne sur la gauche de l'*Inspector*). Les champs déjà renseignés pointent vers des **références déjà disponibles** par le *Prefab* en tant qu'*Asset*. Les champs à renseigner pointent vers des **références de scène**, références que l'*Asset* ne peut pas connaître quand il n'est pas instancié.

Le composant présente ses champs selon leur portée **Private** ou **Public**.

Voyons les champs **Private**.

Le champ **Inventory** est une référence au *ScriptableObject* de l'inventaire. Il est nécessaire pour effectuer les conditions d'objet.

Le champ **Default camera** réfère à la caméra virtuelle *Cinemachine* par défaut.

Le champ **Canvas** réfère au *Canvas* d'UI qui contient les éléments visuels. Cet objet est désactivé par défaut, est activé dès qu'un dialogue commence et désactivé lorsque le dialogue se termine.

Le champ **Last char identifier** accueille un seul caractère. Il sert au programme pour identifier la fin du texte et n'apparaît pas à l'écran. Par conséquent, il doit être un caractère spécifique et non utilisé dans la langue dans laquelle le texte est écrit. Par défaut, ce caractère est « ^ » (un accent circonflexe n'apparaît jamais seul).

Le champ **Reply text** est une référence à l'élément d'UI du texte de réplique. C'est un élément *TextMeshPro* qui permet des mises en forme (couleur, taille de texte…).

Les champs **UI Character left** et UI **Character right** sont des objets qui comprennent des sous-champs qui réfèrent aux éléments d'UI des personnages respectivement de gauche et de droite. On y trouve une référence au **Character**, c'est-à-dire à l'objet lui-même (activé/désactivé selon le personnage qui a la parole) ; une référence à l'**Image** qui affichera le *Sprite* correspondant au personnage et à son point de vue ; une référence **Name** à l'élément d'UI où s'affiche le nom.

Voyons maintenant les champs **Public**.

Il y a deux champs publics, dont un n'apparait pas en *Inspector* dû à une limitation d'*Unity*. Les deux servent le même but : proposer une collection d'objets à désactiver au démarrage de dialogue et à réactiver à la fin du dialogue et ce simplement en utilisant une **case à cocher** dans les paramètres correspondants de l'**entrée de dialogue**. Expliquons.

Pour pouvoir traiter des objets par lot, il convient que les scripts présentent **tous la même structure**, à savoir ici une fonction avec un certain nom ; en effet, comment itérer des choses si elles n'ont pas la même nature ? Pour cela, est proposé une *Interface* **IEnableForDialogue** que l'on implémente à une classe. Or, Unity ne sait pas exposer en *Inspector* des *Interfaces*. Donc, il faut penser une solution de contournement. L'idée retenue a été de réaliser une collection de **scripts *MonoBehaviour*** librement manipulable en *Inspector* et de traiter cette liste de façon à pointer sur les scripts implémentant **IEnableForDialogue**. Pour éviter les erreurs, si dans la collection un objet n'implémente pas **IEnableForDialogue**, alors il ne sera pas considéré. **Cette solution représente néanmoins un point d'amélioration possible.**

Enfin, les autres champs sont renseignés par le programme. Ils sont non éditables et sont exposés à des fins de suivi et de *debug*.

DialogueManager fournit des **méthodes publiques** utilisables à tout moment. On appelle ces méthodes selon ce modèle : **DialogueManager.Instance.NomMethode(paramètres)**. Voyons cela.

DialogueCheckStart() est appelée pour tester un **DialogueRef** et requiert donc une référence en paramètre. C'est la méthode utilisée pour déclencher un dialogue à partir d'une situation 3D.

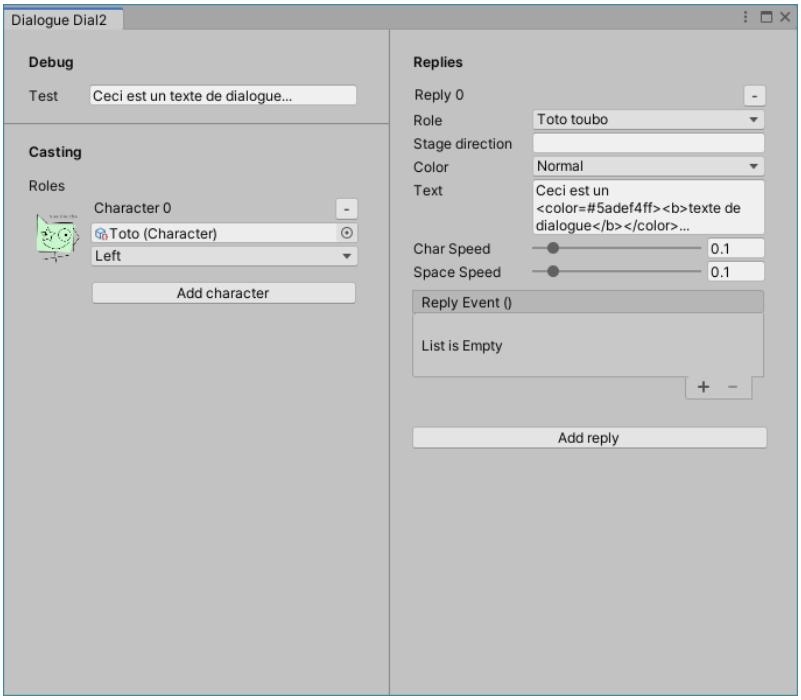
StartDialogueFromFile() permet de lancer un **fichier** de dialogue directement. Par conséquent, aucun test n'est effectué, on lit le dialogue indépendamment de la situation 3D. il requiert une référence de dialogue, une référence de caméra virtuelle, et deux booléens : ces deux variables relatives aux interfaces **IEnableForDialogue** stipulent si **DialogueManager** doit (*true*) ou non (*false*) lancer la fonction **EnableMe(bool value)**, respectivement au démarrage du dialogue et à la fin du dialogue.

5. Dialogue

Un **dialogue** est un **ensemble de données** que l'on renseigne dans une entrée de dialogue et qui est affiché dans la boîte de dialogue UI.

Un dialogue est un fichier *ScriptableObject* que l'on crée dans ***Project* avec clic droit > Create > ScriptableObjects > Dialogue > Dialogue**.

Le fichier une fois créé, en *Inspector* cliquer sur le bouton « **Open Dialogue** » pour ouvrir la fenêtre d'édition. La fenêtre s'ouvre par défaut au milieu de l'écran. Elle peut être repositionnée et ancrée. Limitation : le mode *Play* ferme automatiquement la fenêtre (sinon il y a perte de référence et batterie de messages d'erreur ; sans dommage sur le contenu mais perturbant).



Le premier champ **Test** est un champ de suivi et *debug*. Le renseigner affichera son contenu en *Console*. La nécessité de ce champ est discutable ; le supprimer ?

Le bloc **Casting** présente la **distribution des rôles** pour le dialogue. On peut en ajouter à loisir en cliquant sur le bouton « **Add character** » et les éradiquer avec le bouton « - ». Mais qu'est-ce qu'un rôle ?

Un **Character** est un personnage. **Deux images** (*Sprite)* lui sont nécessaires. En effet, par exemple, un ogre porte un ornement en os à l'oreille droite ; il convient donc que cet ornement reste à l'oreille droite quelle que soit l'orientation de l'ogre. Par conséquent, il est nécessaire de produire deux images pour les deux points de vue. Un **Character** présente aussi un **nom** (**Name**). On crée un **Character** dans ***Project* puis clic droit > Create > ScriptableObjects > Dialogue > Character**.

Maintenant, on utilise les fichiers **Character** pour définir les **Roles** du dialogue. Chaque **Role** présente un **Character** et une **orientation** ou **point de vue** pour ce dialogue, ce qui correspond à une image dont il est question précédemment. Ainsi, par exemple si l'ogre doit apparaître à droite, renseigner **Right** ; s'il doit apparaître à gauche, renseigner **Left**. Par convention, un **Character** à l'écran ne change pas d'orientation au cours de dialogue, sinon il y a un risque de désorienter le joueur.

Il est impossible d'ajouter au **Casting** un **Character** qui serait **identique** à un autre (exemple : le **même** ogre à ornement d'oreille droite)**.** On contourne néanmoins cette restriction en créant un **autre** personnage avec les mêmes données et qui remplira efficacement la fonction d'*alter-ego*. Dans le

même ordre d'idées, si un personnage se transforme en mollusque et que l'on souhaite présenter différentes étapes de sa transformation, alors on créera autant de fichiers **Character** avec des *sprites* différents et un nom qui tend à n'être qu'une collection de « m » (ceci dit, je m'avance un peu car je n'ai jamais entendu de mollusque parler).

Les rôles sont nécessaires et doivent être nécessairement renseignés, sans quoi l'édition des **répliques** est bloquée. Une fois le **Casting** complet, on peut ajouter des répliques avec le bouton « **Add reply** ».

Qu'est-ce qu'une réplique ? Une **réplique** (**Reply**) est un texte énoncé par un personnage. C'est le texte qui s'affiche dans la boîte de dialogue et que l'on passe frénétiquement ou qu'on aime lire jusqu'au bout. Voyons le détail.

Une réplique comprend un **Role**, que l'on choisit dans un menu déroulant. Le contenu de cette liste dépend du **Casting**.

La réplique comprend également des **didascalies** (**Stage Direction**). Ce champ peut rester vide. Pour le compléter, il suffit d'ajouter des mots, une phrase… Les didascalies apparaissent entre parenthèses lors du dialogue (exemple : « (Fatiguée) Je voudrais dormir. »).

Les didascalies peuvent être **mises en forme** à l'aide du champ **Color** qui propose une liste de couleurs par défaut pour *Reliquia*. Par défaut, la valeur est **Normal**.

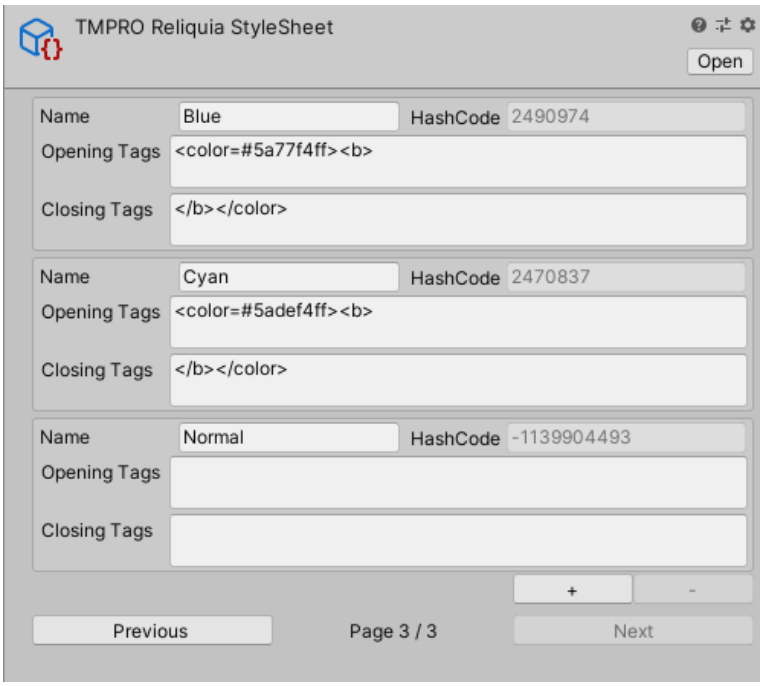
Le champ **Text** accueil le texte de la réplique. Le champ se redimensionne automatiquement selon la longueur du texte. Ce texte peut aussi être **mis en forme** mais cette fois en incluant les codes de mise en forme dans le texte ; *TextMeshPro* interprètera la chaîne de caractères pour effectuer la mise en forme à l'écran (voir le chapitre sur la feuille de styles).

Le texte de réplique s'affiche à une certaine **vitesse** pour les **lettres** et les **espaces**. Ces vitesses sont paramétrées respectivement par les champs **Char Speed** et **Space Speed**. Ces vitesses sont limitées entre 0 et 1 secondes. Par défaut, les valeurs sont 0.1 et 0.5. On modifie ces valeurs soit en entrant un nombre dans le champ de saisie, soit en déplaçant la glissière (*slider*). Côté utilisateur : un clic souris affiche tout le texte, un second clic souris passe la réplique ; si le texte s'est affiché intégralement sans interruption, le clic souris passe la réplique.

Le dernier champ est un *UnityEvent* nommé **Reply Event**. S'il a des méthodes abonnées, il les déclenchera. Noter que cet événement se produit côté *Asset*, c'est-à-dire qu'en édition, on ne peut pas y renseigner manuellement des méthodes de scripts instanciés en scène.

6. Feuille de styles

Pour éditer les **styles du texte** de **réplique**, deux fichiers sont mis à disposition : un fichier *ScriptableObject* des styles *TextMeshPro* nommé **TMPRO Reliquia StyleSheet** et un autre script nommé **DialogueColorsRef**.



Voyons d'abord **TMPRO Reliquia StyleSheet** (feuille de styles). Ce fichier *ScriptableObject* se trouve dans le *Project*. Il est une copie du fichier fourni par défaut par *TextMeshPro*. *TextMeshPro* utilisera les styles de ce fichier spécifique du projet. Pour fonctionner dans la version d'Unity utilisée pour *Reliquia* (la version 2019.3), ce fichier doit être renseigné dans **menu Edit > Project Settings… > TextMesh Pro > Settings** ; pour cette version d'Unity et de *TextMeshPro*, la référence à ce fichier s'effectue par la classe (pour d'autres versions, *TextMeshPro* autorise d'autres accès).

Ce fichier contient une liste de **styles**. Sa structure dépend de *TextMeshPro* et il convient de la conserver. Ces styles présentent un **Name** (nom), un numéro **HashCode** (identifiant unique), des **Openings Tags** (balises d'ouverture) et des **Closing Tags** (balises de fermeture). Par exemple, le champ prévu pour la couleur **Blue** présente les balises suivantes : <color=#5a77f4ff> et </color>. Le principe est celui du langage HTML. Noter que la définition du style peut présenter tout autant de balises que souhaitées (voir la documentation du *package* pour en connaître la liste).

Pour qu'un texte de réplique présente un mot en bleu, on écrira par exemple : « J'aime le <color=#5a77f4ff>bleu</color>. » *TextMeshPro* interprétera la chaîne pour afficher le texte comme il convient.

Noter que le style **Normal** ne présente pas de balisage afin de prendre les valeurs par défaut du composant *TextMeshPro* utilisé dans le *Canvas*, ceci afin de laisser la liberté de colorer un texte *via* le composant.

Pour plus d'informations sur la feuille de styles *TextMeshPro*, consulter la documentation officielle.

Voyons enfin le script **DialogueColorsRef**. J'ai classé ce fichier dans un dossier **Types** pour faire comprendre qu'il va être utilisé pour générer une instance en script C#. Pourquoi pas un *ScriptableObject* ? Car malheureusement il contient une variable d'un type qui ne peut pas être exposé en *Inspector* ; par conséquent il faut ouvrir le fichier dans l'éditeur de code (double-clic pour ouvrir).

```
using System.Collections.Generic;

namespace AlexandreDialogues
{
	[System.Serializable]
	public class DialogueColorsRef
	{
		public Dictionary<string, int> ColorsHash = new Dictionary<string,
int>()
		{
			{"Green", 85746939},
			{"Yellow", -919277916},
			{"Red", 92595},
			{"Magenta", 1269124245},
			{"Violet", -806444115},
			{"Blue", 2490974},
			{"Cyan", 2470837},
			{"Normal", -1139904493},
		};
	}
}
```

Regardons le code de ce fichier. On y lit la déclaration d'un *Dictionary* nommé **ColorHash**, c'est-à-dire une collection de paires clé-valeur. La clé est de type *string* et représente le **Name** d'un style de **TMPRO Reliquia StyleSheet** ; la valeur est de type *int* et représente le **HashCode** d'un style du même fichier. Chaque ligne est donc la **réécriture** de certaines informations d'un style.

Pourquoi ce fichier ? Le but est de permettre le choix de couleur pour les didascalies de réplique et cela par un simple menu déroulant. Ceci est un contournement de la limitation de *TextMeshPro* pour la version d'Unity avec laquelle *Reliquia* est réalisé. Des versions plus récentes de *TextMeshPro* permettent l'accès direct à un style par **Name**, ce qui facilite grandement le développement et permet de se passer d'un fichier « passerelle » dont il est question ici.

Par conséquent, l'ajout, la suppression ou la modification d'un ou plusieurs styles impose d'éditer **aussi** le fichier **DialogueColorsRef**. Ceci représente un haut risque d'erreur, il faut donc être prudent dans l'édition de **TMPRO Reliquia StyleSheet** et **DialogueColorsRef**. Mais ces données n'étant pas vouées à changer régulièrement, cette architecture me semble suffisante, surtout pour le produit final : en effet, **DialogueColorsRef** n'est utilisé que pour l'éditeur, ce qui ne cause pas de surcharge mémoire.