Document Explicatif

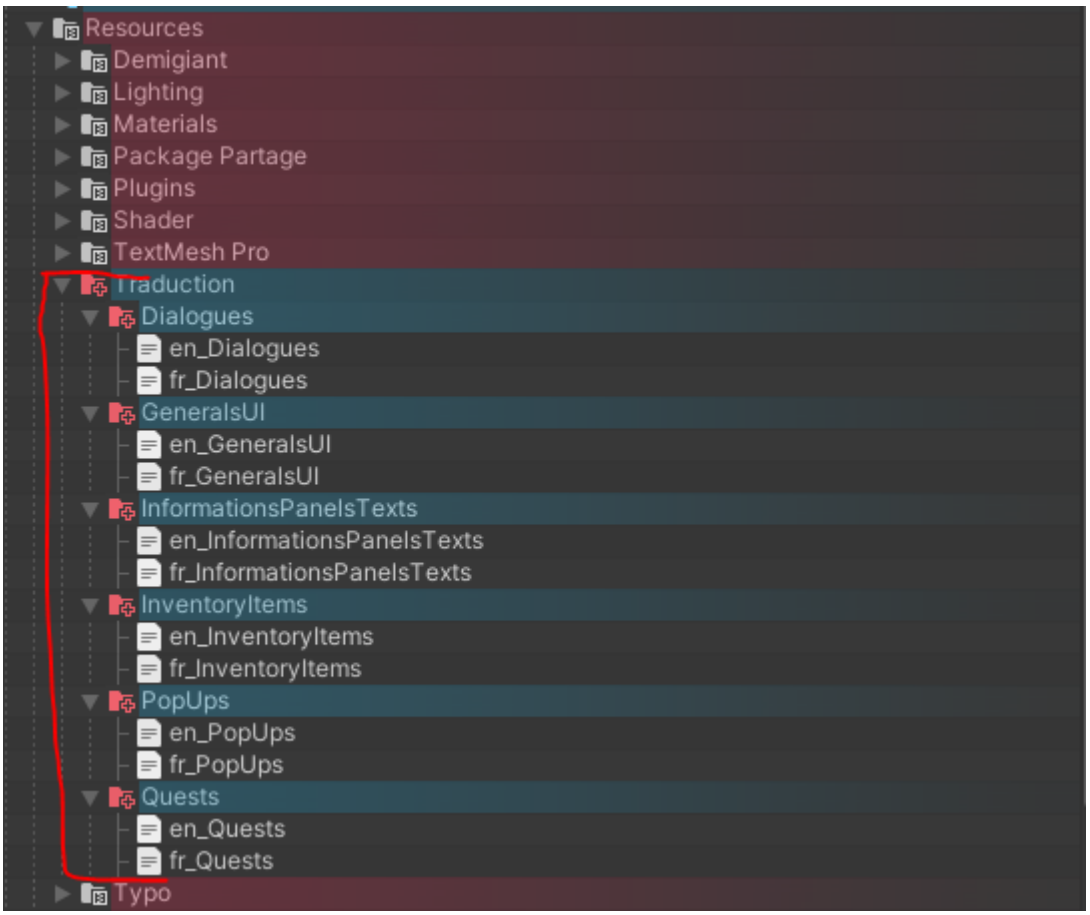
TRADUCTION

Sommaire :

1) Utilisation et création des dictionnaires JSON de traduction

Le système de traduction dans le projet fonctionne à l'aide de dictionnaires en JSON qui sont chargés par le jeu pour afficher les textes utilisant le système dans la langue active.

Ces dictionnaires sont regroupés dans différentes catégories pour des soucis d'organisation et d'optimisation (**dialogues, UI générales, informations des panneaux d'affichages, Objets d'inventaire, pop ups de succès, quêtes**). On a donc un fichier par langue pour chaque catégorie, qui sont trouvables dans le projet Unity au chemin **Ressources > Traduction** :



Chacun de ces dictionnaires se présentent comme ceci une fois ouverts :

```
{
    "dictionnaire": [
        {
            "key": "Dialogue_Baptiste_Martin_JePeuxTAider",
            "value": "Je peux t'aider ?"
        },
        {
            "key": "Dialogue_Baptiste_Martin_QuiEtesVous",
            "value": "Qui êtes-vous ?"
        },
        {
            "key": "Dialogue_Baptiste_Martin_JeDoisYAller",
            "value": "Je dois y aller."
        },
        {
            "key": "Dialogue_Baptiste_Martin_PourCertainsJeSuisUneProjection",
            "value": "Pour certains, je suis une projection de Jérôme, pour d'autres je suis 2P de 1P2P.."
        },
        {
            "key": "Dialogue_Baptiste_Martin_PourJeromeJeSuisMoche",
            "value": "Pour Jérôme, je suis moche. Enchanté."
        },
        {
            "key": "Dialogue_Baptiste_Martin_FaisCommeChezToi",
            "value": "Fais comme chez toi !"
        },


        {
            "key": "Dialogue_Jérôme_Fait_CoucouBienvenueChezNous",
            "value": "Coucou, bienvenue chez nous, fais comme chez toi !"
        },
        {
            "key": "Dialogue_Jérôme_Fait_AQuoiMeneLaPorte",
            "value": "A quoi mène la porte condamnée en haut de l'escalier ?"
        },
        {
            "key": "Dialogue_Jérôme_Fait_JeDoisYAller",
            "value": "Je dois y aller."
        },
        {
            "key": "Dialogue_Jérôme_Fait_CEstNotreAncienBalcon",
            "value": "C'est notre ancien balcon, il est en rénovation pour le moment donc inaccessible."
        },
        {
            "key": "Dialogue_Jérôme_Fait_APlusTard",
            "value": "À plus tard !"
        },
```


La première ligne “**key**” doit être la même dans chaque langue, c’est la variable qui sera indiquée en jeu dans les textes à traduire pour que le système sache quelle traduction appliquée à l'exécution.
La seconde ligne “**value**”, comme son nom l’indique, est la valeur du texte à traduire, soit sa traduction dans la langue du dictionnaire, c’est ici que le traducteur doit saisir la traduction du texte qui sera lue par le jeu.

Attention à ne pas modifier les keys sur un dictionnaire et pas sur les autres ainsi que sur le texte à traduire en jeu, la traduction ne fonctionnera plus.
Attention également à bien respecter la syntaxe, une virgule en moins ou en trop, une accolade en moins ou en trop, et le fichier JSON ne fonctionnera plus.
Attention enfin à ne pas mettre de virgule sur l’accolade de la dernière traduction du fichier JSON, sinon le fichier ne pourra pas non plus être lu en jeu et affichera une erreur.

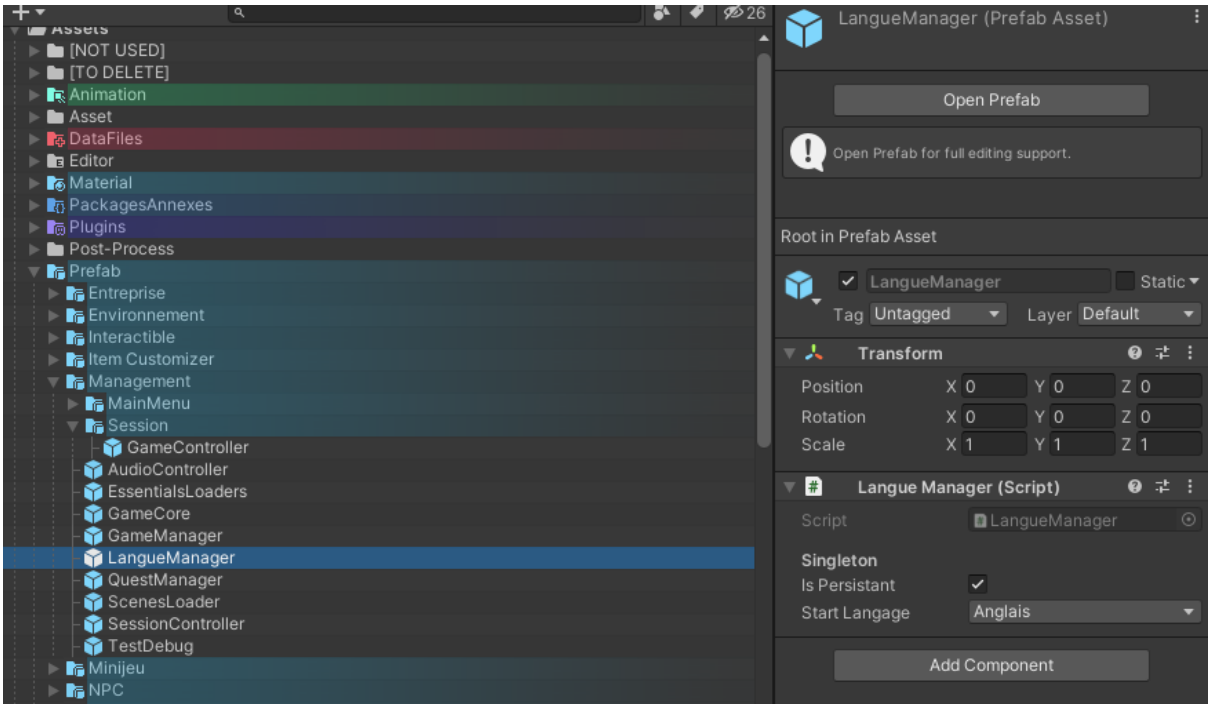
Si jamais en jeu les traductions ne gèrent pas les accents ou les caractères spéciaux, cela signifie que le fichier JSON a été mal encodé à sa création, celui-ci doit être encodé en UTF8, ce qui est normalement le cas nativement, mais Visual Studio ainsi que certains autres éditeurs le gèrent mal. Dans ce cas, il est souhaitable de recréer le fichier sur Notepad++ en l’encodant dans le format souhaité (si des difficultés il est assez simple de trouver comment faire sur le net). Sinon, on peut aussi tout simplement dupliquer l’un des précédents dictionnaires présents dans le projet et correctement encodés, puis le renommer et remplacer son contenu.

2) Gestion des langues

Une fois les dictionnaires de traduction créés et agencés comme il faut, on a besoin de pouvoir les utiliser en jeu là où c'est nécessaire, pour cela on a quelques scripts et objets principaux à configurer et utiliser.

Tout d'abord, dans le prefab **Essentials Loaders** qui est censé être présent sur toutes les scènes et impérativement sur les scènes de départ (Loading Screen / Menu), on a un prefab **Langue Manager** qui est instancié à l'initialisation du jeu et qui reste présent tout du long.

Cet objet possédant le **script singleton LangueManager.cs**, comme son nom l'indique, sert à gérer les langues en jeu et la traduction des textes selon la langue active, c'est par ce script que passe toutes les actions sur les langues et les traductions.
Le principal élément qu'on peut modifier dans l'inspecteur concernant cet objet est la variable **StartLangage** dans laquelle on définit quelle langue on souhaite par défaut au lancement du jeu.



Note : Le prefab LangueManager est trouvable dans le projet Unity au chemin **Prefab > Management**

Si besoin, à travers ce script, on peut changer la langue active via la méthode **ChangeLangage** :

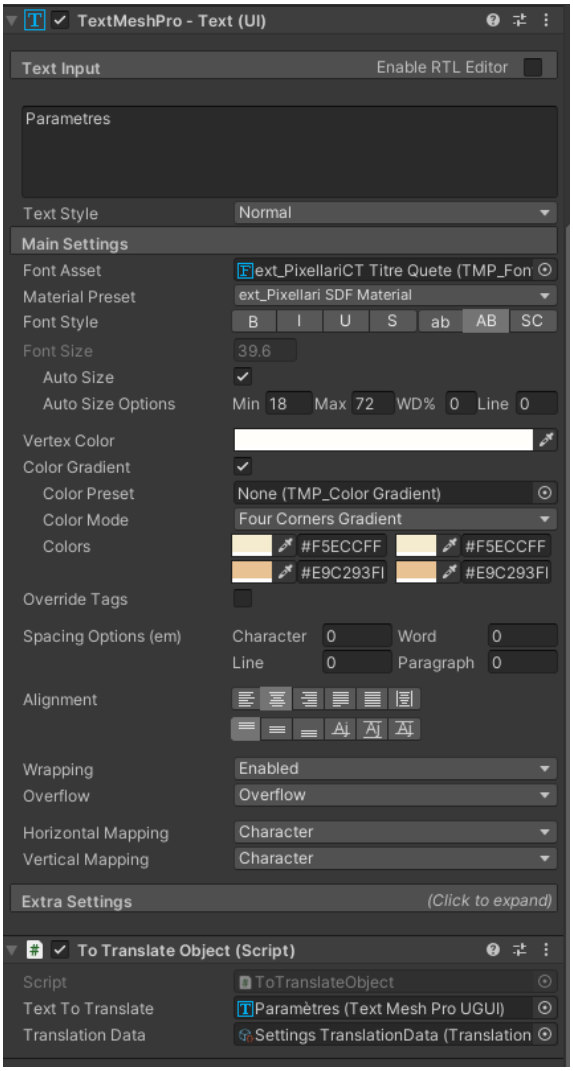
```
4 références
public void ChangeLangage(LocalisationManager.Langage newLangage)
{
    LocalisationManager.SetCurrentLangage(newLangage);
    Debug.Log(LocalisationManager.currentLangage);
    if (onLangageUpdated != null)
        onLangageUpdated();
}
```

Exemple d'utilisation :

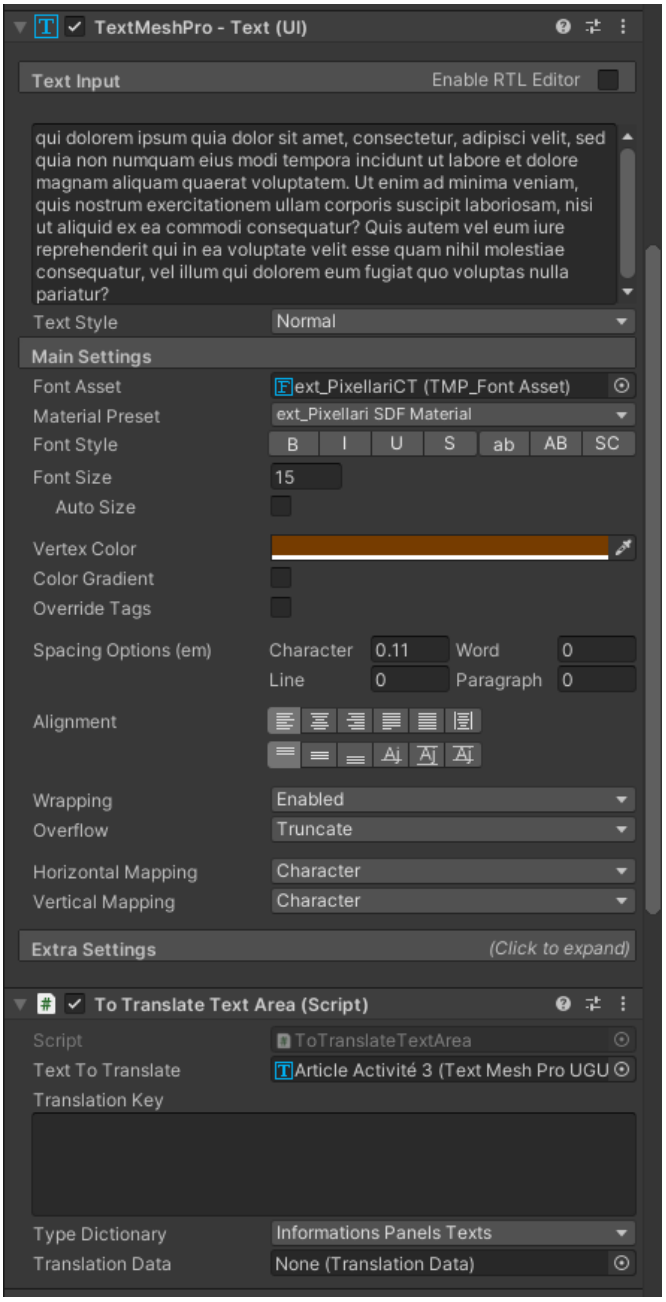
```
LangueManager.Instance.ChangeLangage(LocalisationManager.Langage.Francais);
```

3) Traduction des textes de jeu

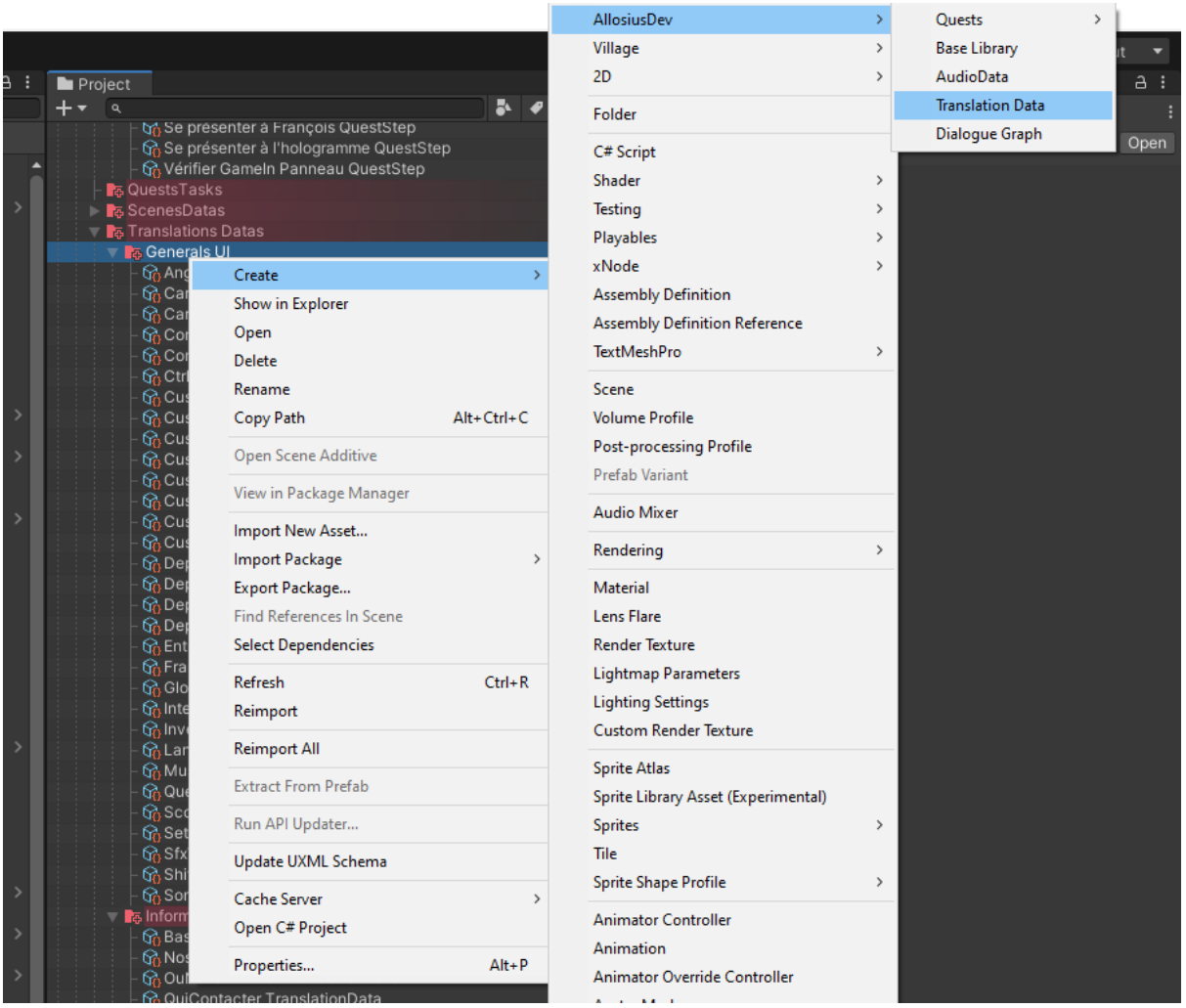
Pour traduire les textes simples statiques (présents sur la scène de base sans être modifié pendant le déroulement de jeu, exemple : textes de menus, d'options, etc…), il est nécessaire de placer sur celui-ci le script **ToTranslateObject.cs**. Pour qu'il fonctionne, il faut affecter le texte à traduire ainsi que le fichier data de traduction associé dans lequel on va définir la clé de traduction du texte à rechercher dans le dictionnaire associé.



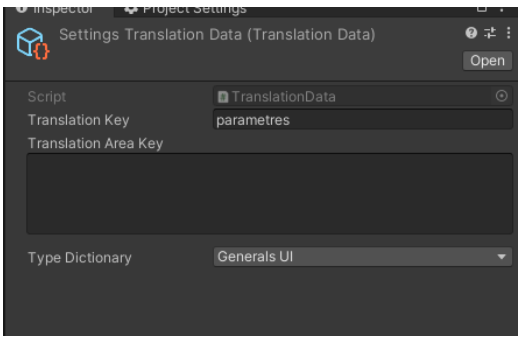
Si jamais le texte est un texte composé avec des sauts de lignes pour la mise en page par exemple ou composé de plusieurs textes différents mis bout à bout, il est nécessaire d'utiliser un script **ToTranslateTextArea.cs** à la place du **ToTranslateObject.cs**, celui-ci nécessite ensuite les mêmes configurations que précédemment :

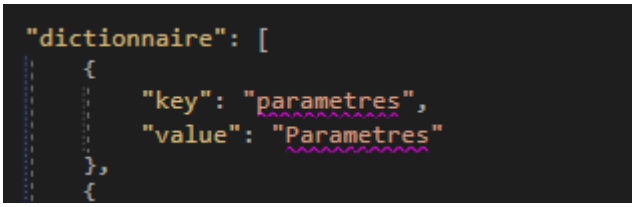


Le **Translation Data** est créable à partir d'un scriptable object dans les assets du projet via le menu **Create > AllosiusDev > Translation Data** :



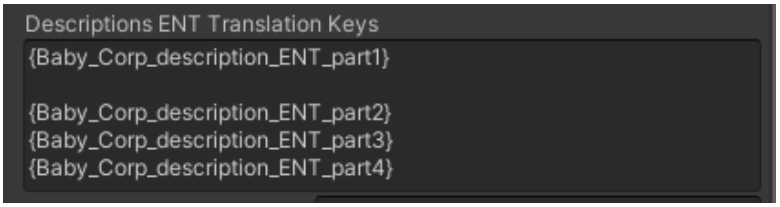
Dans ce fichier data, on a 3 variables à configurer pour que la traduction fonctionne :

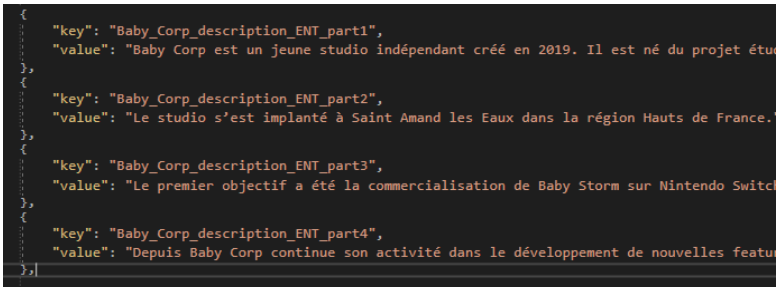




On utilise la **translation key** pour traduire les textes simples, sa valeur doit correspondre à la “**key**” du dictionnaire JSON associé. (variable utilisée par le script **ToTranslateObject.cs**)

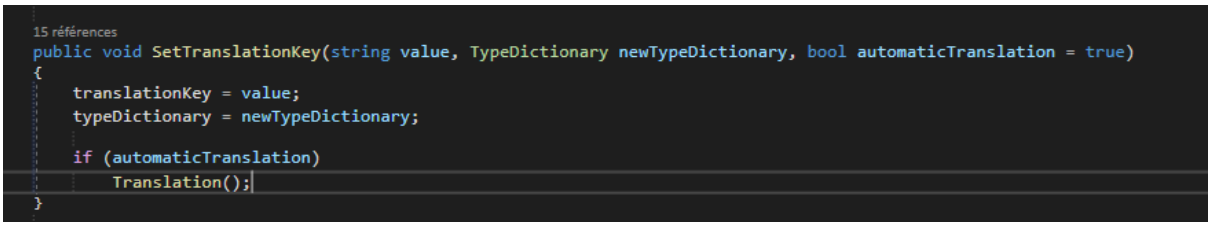
La **translation area key** fonctionne de façon similaire mais n’est utile que pour les textes complexes avec des sauts de ligne pour la mise en page. Les keys doivent être écrites entre accolades pour être lues correctement. (variable utilisée par le script **ToTranslateTextArea.cs**)



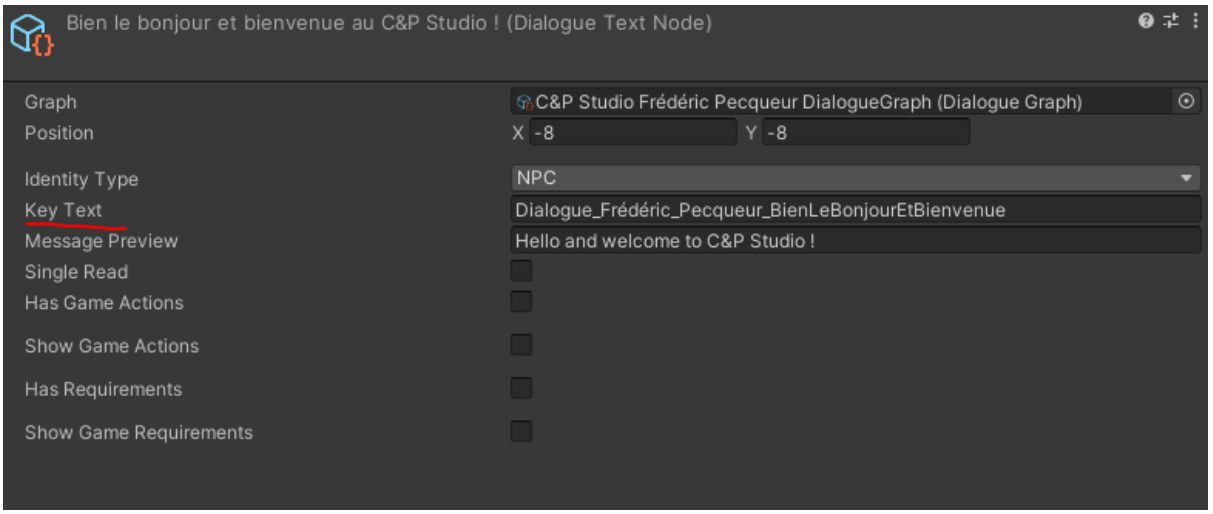


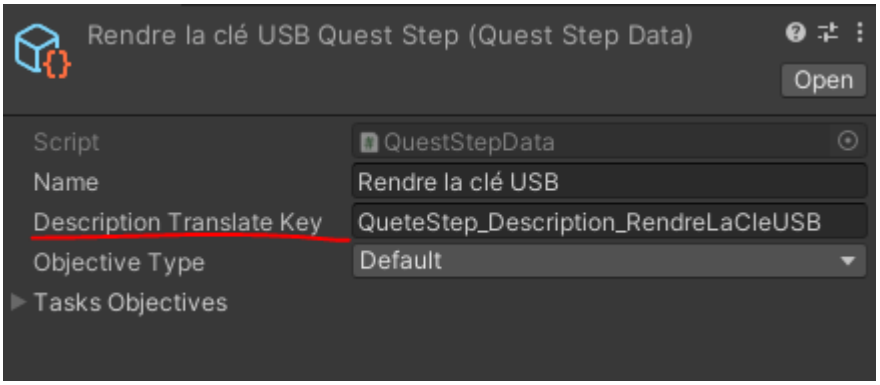
Enfin le **type dictionary** correspond au type de dictionnaire auquel appartient le texte à traduire parmi les différentes catégories de dictionnaires définies (UI générales, Panneaux d'affichage, dialogues, quêtes, etc…)

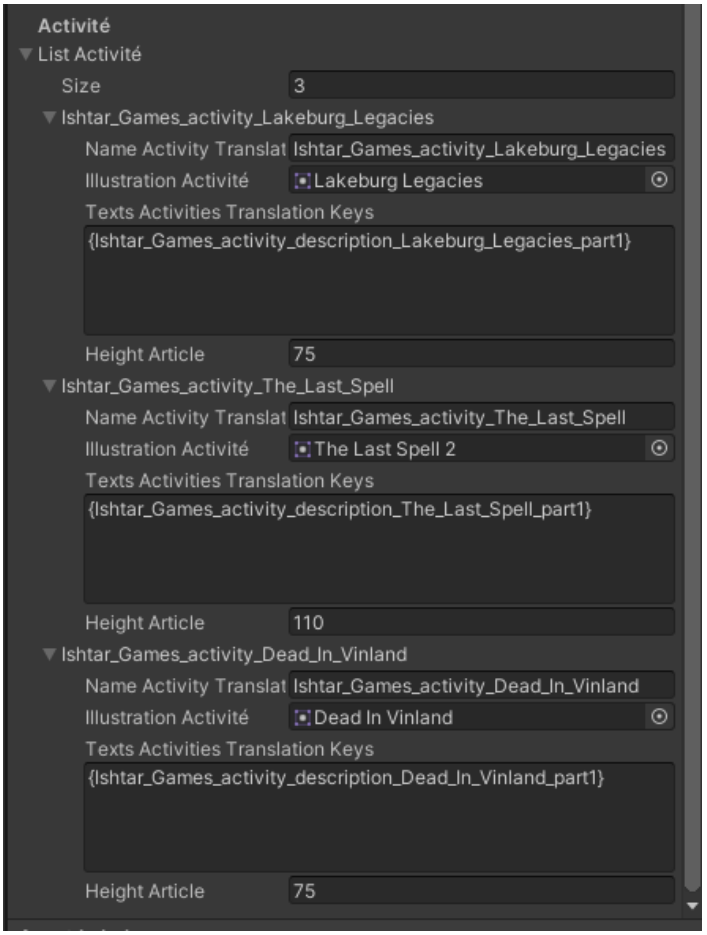
Attention, les datas ne sont utiles que dans le cas où les textes à traduire sont fixes et définis une fois pour toutes . Pour les textes dynamiques, instanciés pendant le jeu, dont la valeur va être amenée à changer durant l'exécution (exemple les dialogues, l'ui de suivi de la quête en cours), on va être amené à affecter les keys des textes à traduire ainsi que le typeDictionary associé directement dans le code ou via d'autres scripts ou datas selon les objets, en utilisant la méthode **SetTranslationKey** des scripts **ToTranslateObject.cs** ou **ToTranslateTextArea.cs** :



ex : Dans l'éditeur de dialogues, dans les quêtes, dans les datas de panneaux d'affichage :







4) Fonctionnement technique des scripts de lecture et d'application des traductions

Attention, il est nécessaire de bien respecter la nomenclature des fichiers JSON ainsi que leur localisation, pour toute modification il faut également corriger le script **JSONLoader.cs**, car il utilise le nom et la localisation précise des dictionnaires pour les charger.

Exemple :

```
if (typeDictionary == TypeDictionary.Dialogues)
{
    codeLangue += "_Dialogues";
    //path = Application.streamingAssetsPath + "/Traduction/Dialogues/" + codeLangue + ".json";
    //json = File.ReadAllText(path, Encoding.UTF8);
    json = Resources.Load<TextAsset>("Traduction/Dialogues/" + codeLangue).text;
    //json = Resources.Load<TextAsset>(string.Format("Traduction/Dialogues/{0}", codeLangue)).text;
}
```

En effet dans ce script, on retrouve une méthode **LoadJson** qui va charger les dictionnaires correspondant au type de textes qu'on souhaite traduire (dialogues, UI, quêtes, etc…). Tel qu'il fonctionne, il va chercher le fichier dont le nom commence par le "**code langue**" voulu à savoir **fr** pour français, **en** pour anglais (ces "codes" sont définis dans le script **LocalisationManager.cs**), auquel il va ajouter **"_NOMCATEGORIE"**, soit dans l'exemple ci dessus, si on souhaite traduire un dialogue en anglais, il va rechercher le dictionnaire : **en_Dialogues**.
Il va alors récupérer ce fichier dans le chemin indiqué par la méthode **json = Resources.Load<TextAsset>("CHEMINVOULU")**. Si jamais on souhaite changer l'endroit où sont stockés les dictionnaires il faudra aussi changer cette méthode pour qu'elle aille les chercher au bon endroit, ici le script cherchera le dictionnaire dans **Ressources > Traduction > Dialogues**. C'est donc dans ce répertoire qu'il faut placer les dictionnaires de chaque langue de traduction des dialogues.

Si l'on souhaite ajouter d'autres langues au projet (actuellement traduisible uniquement en français et en anglais), il faudra se rendre dans le script **LocalisationManager.cs** qui gère les différentes langues

```
24 références
public class LocalisationManager
{
    /// <summary>
    /// Liste des langues gérées
    /// </summary>
    17 références
    public enum Langage
    {
        Francais,
        Anglais,
    }

    public static Langage currentLangage = Langage.Anglais;
```

Celui-ci possède une **enum Langage** qui regroupe la liste des langues possibles, à laquelle on peut ajouter d'autres langues si nécessaire.

```
6 références
private static void LoadJSONDictionary(TypeDictionary typeDictionary)
{
    JSONLoader json;

    switch (currentLangage)
    {
        case Langage.Francais:
            json = JSONLoader.LoadJSON("fr", typeDictionary);
            break;
        case Langage.Anglais:
            json = JSONLoader.LoadJSON("en", typeDictionary);
            break;
        default:
            json = JSONLoader.LoadJSON("en", typeDictionary);
            break;
    }

    localDico = json.GetDictionaryValues(localDico);
}
```

Ensuite, dans la méthode **LoadJSONDictionary** du même script, on doit gérer le cas de chaque langue pour qu'elle soit lue par le système, par exemple si on voulait ajouter de l'espagnol il faudrait glisser après le **case Langage.Anglais** :

case Langage.Espagnol:
json = JSONLoader.LoadJSON("es", typeDictionary);
break;

Le **"es"** à l'intérieur du **JSONLoader.LoadJSON**, correspondant au code langue souhaité pour les dictionnaires de cette langue, ils devront alors tous avoir leur nom commencer par "es" comme les dictionnaires français qui commencent par "fr" et les dictionnaires anglais par "en".
Enfin, la variable **typeDictionary** passée en paramètre de la fonction correspondant à la catégorie de dictionnaires qu'on cherche à traduire (dialogues, UI, quêtes, etc…).

Pour ajouter des nouvelles catégories de dictionnaires si besoin, on a une autre enum tout en bas du script, **TypeDictionary** :

```
52 références
public enum TypeDictionary
{
    Default,
    GeneralsUI,
    InformationsPanelsTexts,
    PopUps,
    InventoryItems,
    Dialogues,
    Quests,
}
```

On peut ajouter à la suite de cette enum de nouvelles catégories, il faudra néanmoins appeler leur traduction dans la méthode **Init** du script :

```
2 références
public static void Init(TypeDictionary typeDictionary)
{
    localDico = new Dictionary<string, string>();

    //LoadJSONDictionary(typeDictionary);

    //LoadJSONDictionary(TypeDictionary.Default);

    LoadJSONDictionary(TypeDictionary.Dialogues);
    LoadJSONDictionary(TypeDictionary.GeneralsUI);
    LoadJSONDictionary(TypeDictionary.InformationsPanelsTexts);
    LoadJSONDictionary(TypeDictionary.InventoryItems);
    LoadJSONDictionary(TypeDictionary.PopUps);
    LoadJSONDictionary(TypeDictionary.Quests);

    isInit = true;
}
```

Sans oublier, dans la méthode **LoadJSON** du script **JSONLoader.cs** évoqué précédemment d'ajouter les conditions de recherches du fichier JSON, pour que celui-ci puisse être récupéré et lu par le système :

```
else if (typeDictionary == TypeDictionary.PopUps)
{
    codeLangue += "_PopUps";
    //path = Application.streamingAssetsPath + "/Traduction/PopUps/" + codeLangue + ".json";
    //json = File.ReadAllText(path, Encoding.UTF8);
    json = Resources.Load<TextAsset>("Traduction/PopUps/" + codeLangue).text;
    //json = Resources.Load<TextAsset>(string.Format("Traduction/PopUps/{0}", codeLangue)).text;
}
else if (typeDictionary == TypeDictionary.Quests)
{
    codeLangue += "_Quests";
    //path = Application.streamingAssetsPath + "/Traduction/Quests/" + codeLangue + ".json";
    //json = File.ReadAllText(path, Encoding.UTF8);
    json = Resources.Load<TextAsset>("Traduction/Quests/" + codeLangue).text;
    //json = Resources.Load<TextAsset>(string.Format("Traduction/Quests/{0}", codeLangue)).text;
}
```

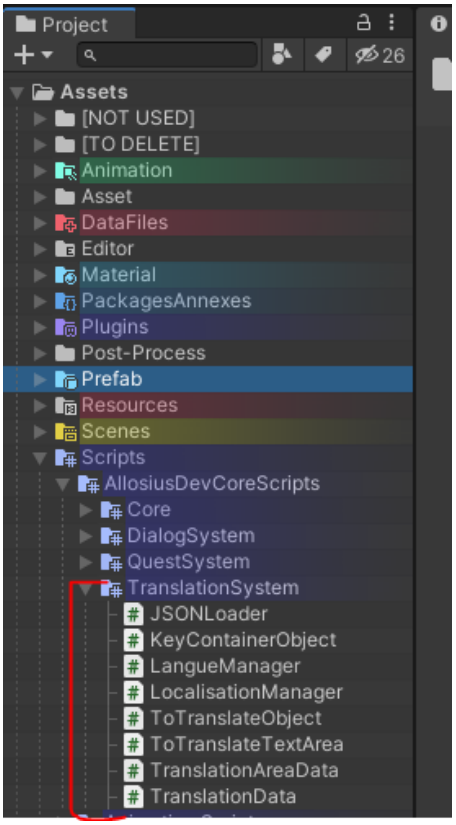
5) Codes spéciaux

Il existe certains codes spéciaux pour l'écriture des textes, notamment quand on souhaite récupérer des valeurs de jeu dans un texte de dialogue :

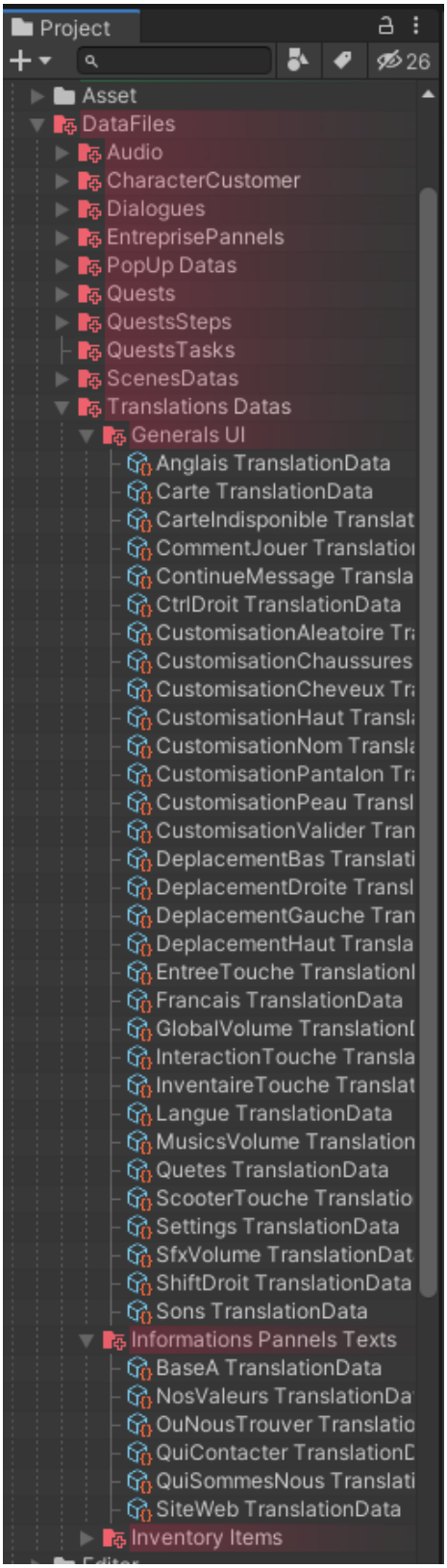
- **Nom du joueur** : si on veut afficher le nom du joueur dans un texte, il faut le saisir de la façon suivant : [PLAYER] exemple : "Hello [PLAYER], bienvenue chez les Crafteurs !". En jeu le [PLAYER] sera remplacé par le nom du joueur qu'il a défini au lancement de sa partie.

6) Répertoires des scripts et datas

Vous pouvez retrouver dans le projet l'ensemble des scripts concernant le système de traduction au chemin : **Scripts > AllosiusDevCoreScripts > TranslationSystem** :



Vous pouvez retrouver dans le projet les fichiers datas de **TranslationDatas** au chemin : **DataFiles > Translations Datas** :


Project
26
Asset
DataFiles
Audio
CharacterCustomer
Dialogues
EntreprisePannels
PopUp Datas
Quests
QuestsSteps
QuestsTasks
ScenesDatas
Translations Datas
Generals UI
Anglais TranslationData
Carte TranslationData
CarteIndisponible Translat
CommentJouer Translatio
ContinueMessage Transla
CtrlDroit TranslationData
CustomisationAleatoire Tr
CustomisationChaussures
CustomisationCheveux Tr
CustomisationHaut Transl
CustomisationNom Transl
CustomisationPantalon Tr
CustomisationPeau Transl
CustomisationValider Tran
DeplacementBas Translati
DeplacementDroite Transl
DeplacementGauche Tran
DeplacementHaut Transla
EntreeTouche TranslationI
Francais TranslationData
GlobalVolume TranslationI
InteractionTouche Transla
InventaireTouche Translat
Langue TranslationData
MusicsVolume Translation
Quetes TranslationData
ScooterTouche Translatio
Settings TranslationData
SfxVolume TranslationDat
ShiftDroit TranslationData
Sons TranslationData
Informations Pannels Texts
BaseA TranslationData
NosValeurs TranslationDa
OuNousTrouver Translatio
QuiContacter TranslationD
QuiSommesNous Translati
SiteWeb TranslationData
Inventory Items