



RÉPUBLIQUE
FRANÇAISE

*Liberté
Égalité
Fraternité*



Navmer3D - NavmerInput

Usage des tableaux de commandes Navmer & Intégration UE

CEREMA Dtec Risques, eau et mer

30/11/2022

Table des matières

1 Avant-propos	4
2 Les tableaux de commandes	5
2.1 La Ship Console	6
2.2 L'Azimuth Console	9
2.3 Dans Unreal Engine	10
3 Analyse de l'existant	11
3.1 Les plugins existants	12
3.1.1 Controlysis	13
3.1.2 WM Input Manager	15
3.1.3 SDL2InputDevice	16
3.1.4 JoystickPlugin	17
3.1.5 Conclusion : Les plugins existants	21
3.1.6 RawInput Plugin	24
4 Le plugin NavmerInput	25
4.1 Directives d'emploi	28
4.1.1 Installer le plugin	29
4.1.2 Configurer les tableaux de commandes	31
4.1.3 Installer le menu de configuration	38
4.1.4 Utiliser le menu de configuration	42
4.2 Directives de maintenance	43
4.2.1 Les périphériques HID	44
4.2.2 Code : Général	47
4.2.3 Code : Structures de données principales	49
4.2.4 Code : Classes principales et fonctionnalités	51
4.2.5 Code : La classe UJoystickSubsystem	52
4.2.6 Code : La classe FJoystickInputDevice	54
4.2.7 Code : Echange entre les classes principales	55
4.2.8 Code : La classe UJoystickFunctionLibrary	56
4.2.9 Code : Paramètres du plugin	57
4.2.10 Code : Widgets UMG	59
4.2.11 Blueprints : Général	66
4.2.12 Blueprints : Widget Menu de configuration	67
4.2.13 Blueprints : Widget Messages	76
4.2.14 Blueprints : Widget MessagesListEntry	80
4.2.15 Blueprints : Animation du Widget MessagesListEntry	84

4.2.16 Blueprints : Les widgets dans l'HUD	86
4.3 Améliorations	90
5 Sources utiles	92

1 Avant-propos

Présentement, l'intégration des tableaux de commandes Navmer sous Unreal Engine n'est pas triviale.

Elle implique d'abord la connaissance des **Périphériques d'Interface Humaine (HID)**, puis des outils de développement adéquats, notamment dans le cadre d'une intégration sur Windows. Elle nécessite également d'avoir un certain niveau d'expertise dans le développement modulaire sous Unreal Engine.

Les tableaux de commandes Navmer sont des périphériques USB dits "génériques" qui **présentent certaines spécificités** (l'on verra par la suite qu'Unreal Engine ne gère pas naturellement ce genre d'équipements sous Windows).

Au bonheur des non initiés, Unreal est un moteur qui encourage la modularité : l'utilisation de "**Plugins**"¹ communautaires comme point de départ permet de gagner du temps dans le développement en évitant un apprentissage profond des technologies complexes sous-jacentes.

Ce document technique, à la croisée entre une synthèse de la recherche initiatique menée et un manuel opératoire, regroupe tous les éléments nécessaires à la compréhension du sujet et de l'outil d'intégration proposé comme solution.

Il est présenté en trois grandes parties. La **première partie** introduit les tableaux de commandes, leurs caractéristiques et les besoins dans le cadre de l'intégration. La **deuxième** fait état des solutions pré-existantes.

Enfin la **dernière partie** concerne **la mise en place et l'utilisation de l'outil d'intégration**, puis **la maintenance de l'outil**. Pour l'un qui souhaite installer l'outil sur un projet Unreal ou simplement s'en servir : il lui sera juste nécessaire d'explorer la sous-section **Directives d'emploi**. Pour l'autre qui souhaite reprendre le code source : il est fortement recommandé de lire au préalable la section **Les tableaux de commandes** pour en comprendre les propriétés, les nécessités et décisions prises dans la construction de l'outil avant d'aborder la section **Le plugin NavmerInput**.

ATTENTION !

Ce document a été conçu pour une utilisation d'Unreal Engine sous Windows ! Il n'y a donc aucune garantie sur le portage des différentes technologies, modules et opérations introduits sur d'autres systèmes d'exploitations.

D'autre part, certains termes techniques liés à l'informatique en général qui sont cités dans ce document ne seront évidemment pas expliqués. C'est donc à la charge du lecteur de se renseigner à leur propos.

1. Un Plugin est un module d'extension à Unreal Engine. Dans le cas présent, le Plugin propose des fonctionnalités permettant de traiter des contrôleurs de jeux spéciaux.

2 Les tableaux de commandes

Les tableaux de commandes utilisés sur le simulateur Navmer sont des périphériques d'entrée USB dédiés à la manœuvre de différents bateaux. Ils sont définis matériellement et détectés par le système d'exploitation comme des contrôleurs de jeux génériques de type "Joystick".

Ils sont fournis par les distributeurs *VRinsight*² et *Wilco Publishing*³.

2. *VRinsight* : <http://www.vrinsightshop.com>

3. *Wilco Publishing* : <https://www.wilcopub.com>

2.1 La Ship Console

La Ship Console est un des tableaux de commandes à disposition.

Elle est dotée de 22 boutons (valeurs booléennes) et de 4 axes (valeurs analogiques). Les axes sont répartis sur les manettes de gouvernail, moteur gauche, moteur droit, mais également sur 4 boutons pour les “axes de tête” qu’on peut trouver parfois sur les joysticks.

La disposition de la Ship Console en fait un contrôleur orienté commande de bateaux à **propulsion par hélices**.

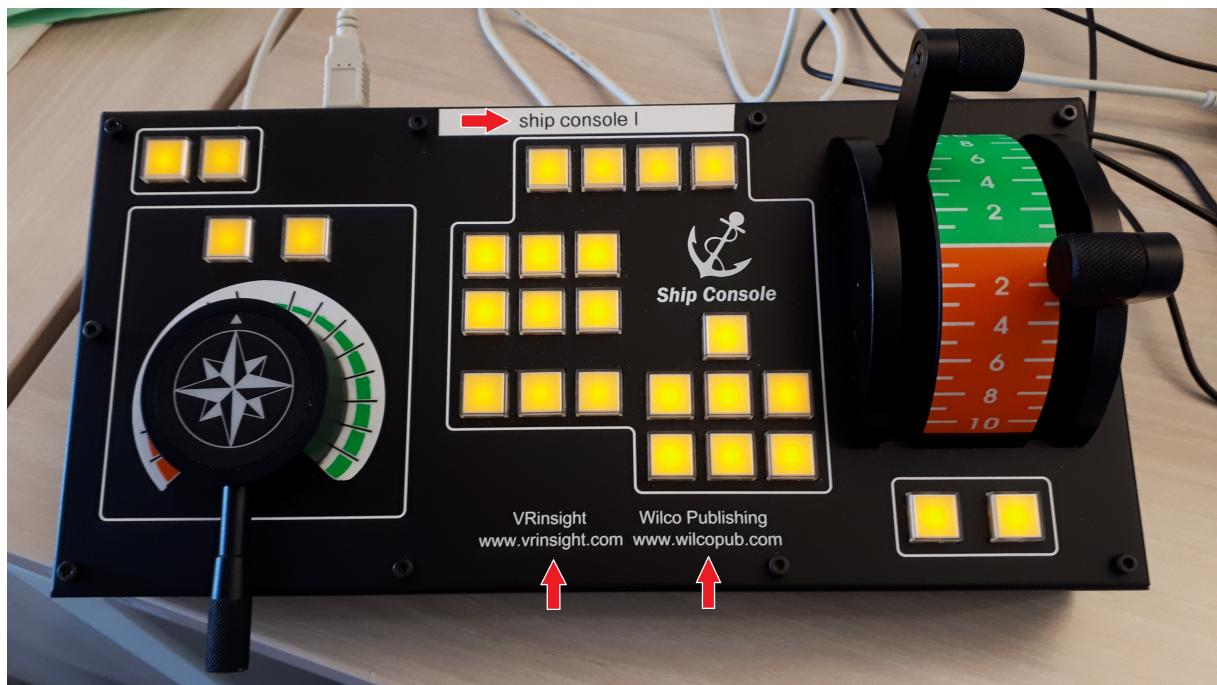


FIGURE 1 – Photo d’une Ship Console⁴. La manette de gouvernail se trouve à gauche et les manettes de propulsion à droite.

Différents tests ont permis d’identifier la présence d’erreurs sur les valeurs perçues générées par les composants matériels aux commandes analogiques (manettes). Typiquement, on peut noter des **erreurs de plages de valeurs** et de **décentrage** sur les axes.

4. Ship Console de chez VRinsight : <http://www.vrinsightshop.com/shop/step1.php?number=24>

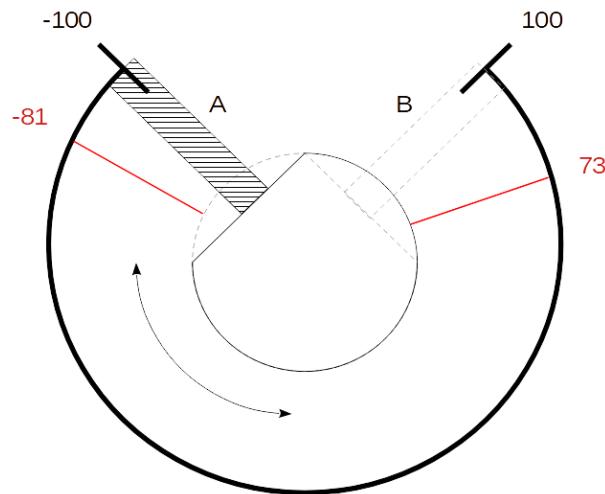


FIGURE 2 – Exemple d'une plage de valeur asymétrique avec l'axe du gouvernail. Bien que les butées soient atteintes, les valeurs analogiques n'atteignent pas les extrêmes.

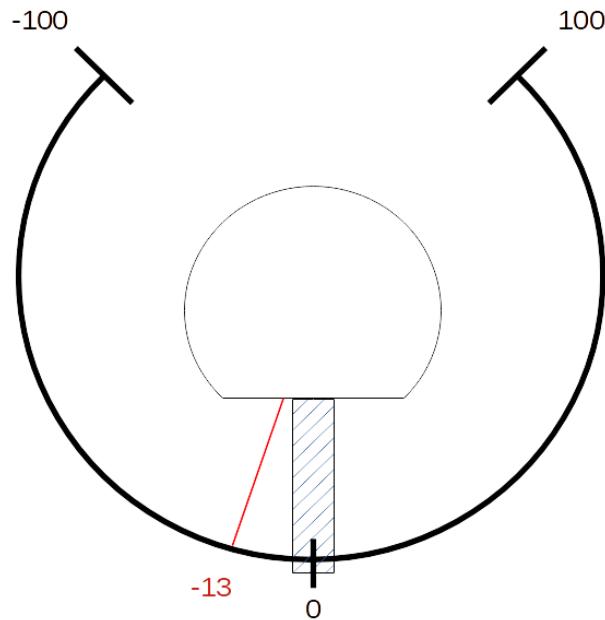


FIGURE 3 – Exemple du décentrage avec l'axe du gouvernail. Bien que la manette soit centrée, la valeur analogique obtenue n'est pas nulle.

On notera également que les valeurs faussées restent persistantes, même après plusieurs branchage/-débranchage (ie. les valeurs perçues ne changent pas).

Une solution à ce type de problème serait d'effectuer **un étalonnage** du contrôleur qui prendrait en compte la valeur au centre de la manette perçue.

Windows propose pour exemple un wizard⁵ permettant d'effectuer un étalonnage simple des équipements de jeu.

Cependant il est à noter qu'il ne prend pas en compte la valeur au repos de l'axe : après réassiguation des plages de valeurs, le décentrage est toujours présent.

À ce jour, deux Ship Consoles sont à disposition : la Ship Console étiquetée **Ship Console 1** provenant de chez *VRinsight et Wilco Publishing*, et la Ship Console étiquetée **Ship Console 2** provenant de chez *VRinsight* seulement.

Les deux font présence des mêmes problèmes; les valeurs perçues aux extremums et au centres des axes diffèrent cependant.

Aussi, étant **deux produits identiques qui proviennent des mêmes fabricants**, il se peut que certains outils de traitement des entrées (on entend par "entrée" les boutons, les axes, etc.) puissent confondre les valeurs perçues : forçant une intervention au niveau logiciel.

Il existe d'autres différences notables entre les deux Ship Consoles qui seront mieux détaillées dans la sous-section **Directives de maintenance**.

NOTE

Malheureusement, il se trouve que la Ship Console 2 montre des sauts non-uniformes de valeurs sur chacune des manettes. Il n'existe à ce jour aucune solution à ce problème.

5. Étalonnage via le wizard de Windows : <https://www.tenforums.com/tutorials/103910-calibrate-game-controller-windows-10-a.html>

2.2 L'Azimuth Console

L'Azimuth Console est un des tableaux de commandes à disposition.

Elle est composée de 10 boutons et de 4 axes. Les axes sont entièrement répartis sur les manettes, avec 2 manettes à propulsion pivotables sur 360°.

La disposition de l'Azimuth Console en fait un contrôleur orienté commande de bateaux à **propulsion par pods azimutaux**.



FIGURE 4 – Photo d'une Azimuth Console⁶. Les deux manettes à propulsion peuvent effectuer des rotations sur 360°.

Les manettes à propulsion sont similaires à celles de la Ship Console, de même que les problèmes de plages de valeurs et de décentrage se reportent également sur les axes de rotation des manettes.

6. Azimuth Console de chez VRinsight : <http://www.vrinsightshop.com/shop/step1.php?number=90>

2.3 Dans Unreal Engine

Par défaut sur Windows, Unreal Engine utilise l'**API** native *XInput*⁷ pour traiter les contrôleurs de jeu.

Il y a plusieurs conséquences à cela : le type de contrôleur pris en charge qui est limité aux périphériques XInput conformes (les contrôleurs récents comme p.ex. les manettes Xbox de Microsoft), et l'impossibilité de gérer plusieurs contrôleurs en simultané.

NOTE

On suppose que les contrôleurs de jeu non génériques de constructeurs mainstream (*Sony*, *Nintendo*, etc.) sont traités différemment et fournissent des pilotes spécifiques.

Heureusement, il existe des méthodes, voire des solutions directes pour outrepasser ces limites afin d'accéder à la prise en charge des tableaux de commandes Navmer dans Unreal Engine.

Il subsiste cependant les problèmes particuliers de plages de valeurs et de décentrage des axes qui, faute d'être présents au niveau matériel, devront être traités spécialement au niveau logiciel.

7. API *XInput* de Microsoft pour prendre en charge les contrôleurs actuels : <https://fr.wikipedia.org/wiki/XInput>

3 Analyse de l'existant

Unreal Engine est un moteur de jeu vidéo hautement modulaire, qui fait intervenir des “plugins”, ie. des modules de codes qui s’intègrent au moteur et/ou à un projet, et qui apportent des fonctionnalités ou du contenu supplémentaire.

Dans le cas présent de recherche, le plugin serait capable de traiter des contrôleurs de jeu génériques.

Il est bien sûr possible via un plugin Unreal d’interfacer un outil de développement externe au moteur (API, bibliothèque, etc.) pour en extraire les fonctionnalités.

Cette analyse va donc faire état **des plugins existants** qui pourraient potentiellement répondre aux exigences de l’intégration des tableaux de commandes Navmer sur Unreal Engine (cf. section **Les tableaux de commandes**).

3.1 Les plugins existants

Du point de vue de l'utilisateur simple qui se tourne vers l'existant, le **Marketplace** d'*Epic Games* propose divers plugins capables de prendre en charge des contrôleurs de jeu génériques.

Le Marketplace étant relativement riche en contenu, cette partie de l'analyse va se concentrer sur une partie de ce qui est proposé : précisément sur les plugins retenus comme étant les plus attrayants par rapport aux besoins.

Les plugins présentés, de toutes origines (web ou Marketplace), ne feront pas tous l'affaire d'une analyse très exhaustive car malheureusement, étant non libres et issues de la communauté, à moins de tous les acheter et de les essayer (non concevable) il n'existe que très peu d'informations publiques à leur sujet.

Cadrée sur les besoins et les spécificités en terme de fonctionnalités, cette analyse présentera donc une sélection de plugins qui semblent avoir un potentiel à répondre aux exigences des tableaux de commandes, et conclura sur une comparaison avec les données à disposition.

On rappelle les nécessités des tableaux de commandes Navmer : ce sont des contrôleurs génériques qui ont des complications matérielles qui obligent une configuration particulière des manettes.

3.1.1 Controlysis

Auteur : TriClover

Version(s) UE supportée(s) : 4.x, 5.0

Documentation : Oui, à l'achat

Lien site : <https://www.unrealengine.com/marketplace/en-US/product/controlysis>

Ce plugin n'ayant que trop peu de précisions données par son créateur, il est difficile d'analyser correctement celui-ci.

La majeure partie des informations proposées viennent de sa courte description et sont vérifiées à partir des reviews et réponses apportées aux utilisateurs sur le Marketplace : le lecteur est donc averti sur l'exactitude des propos.

Controlysis est un plugin vraisemblablement implémenté avec la bibliothèque *SDL*⁸ qui permet d'utiliser tout type de contrôleur de jeu sur Unreal Engine.

Le plugin est orienté “code”⁹ mais il peut également fonctionner sur des projets “non-code”.

Il est en capacité de gérer plusieurs contrôleurs de jeu en simultané, et par joueur.

Des noeuds **Blueprint**¹⁰ de capture sont fournis pour extraire l'état d'un contrôleur de jeu (connecté/-déconnecté) et ses données d'entrées (ie. les valeurs logiques fournies par l'appui des boutons, des axes, etc.).

De manière optionnelle, il propose une **UI** (Interface Utilisateur) préfaite développée via **UMG**¹¹ qui permet la visualisation des données de capture.

La découverte des contrôleurs de jeu se fait au moyen des identifiants matériels¹², ce qui permet à contrario d'un numéro de série de conserver toute configuration logicielle lorsqu'un contrôleur change d'état.

Comme précisé plus en haut, le manque d'informations permet de se poser certaines questions.

Il n'y a aucun moyen de savoir s'il existe un système permettant d'effectuer un étalonnage dans le cas où les valeurs reçues d'axes sont décalées ou erronées (au même titre que les ajustements de type **sensibilité**, **deadzone** ou **inversion**).

8. Bibliothèque logicielle libre *SDL2* : https://en.wikipedia.org/wiki/Simple_DirectMedia_Layer

9. Les projets sur Unreal Engine sont identifiés selon deux types : “code” (ou C++) pour un projet complexe qui fait intervenir du code C++, et “non-code” (ou Blueprint only) pour un développement plus accessible au moyen du Visual Scripting.

10. Blueprint : brique de programmation réalisée avec le **Visual Scripting** d'Unreal Engine

11. *Unreal Motion Graphics* : le système de création d'UI, à partir d'éléments graphiques appelés “Widgets”

12. Les identifiants matériels inscrits par un vendeur de périphérique USB reconnu comme HID correspondent aux numéros uniques du vendeur et du produit

Le plugin ne semble pas s'intégrer à l'interface d'Unreal.

Rien ne montre que les entrées capturées sont facilement réutilisables ailleurs que dans les Blueprints où sont appelés les fonctions d'écoute.

Il semble nécessaire avec Controlysis d'affecter les valeurs sondées à des variables accessibles uniquement dans le Blueprint en question pour pouvoir les traiter en aval.

NOTE

Unreal Engine, dans son traitement des entrées joueur, permet d'assigner ces entrées à des **clés d'actions** nommables.

C'est une fonctionnalité du moteur très pratique puisqu'elle octroie flexibilité et réutilisabilité en permettant de programmer une action quelconque suite à un évènement produit par le joueur, de manière à s'accorder avec les différents éléments de programmation d'Unreal, et ce peu importe les entrées associées.

3.1.2 WM Input Manager

Auteur : Lemontmoon

Version(s) UE supportée(s) : 4.x, 5.0

Documentation : Non

Lien site : <https://www.unrealengine.com/marketplace/en-US/product/wm-input-manager>

WM Input Manager est un plugin du Marketplace qui semble avoir été implémenté en combinant les APIs *Raw Input*¹³ et *XInput* de Windows.

Il est donc en capacité de prendre en compte n'importe quel type de contrôleur de jeu.

Il est aussi capable d'assigner plusieurs contrôleurs à un seul même joueur.

La détection des contrôleurs et de toutes les entrées possibles de chacun se fait automatiquement.

Le type d'un axe¹⁴ peut également être détecté.

Au total, un nombre de 128 boutons, axes, curseurs et DPads sont supportés.

Les projets qui proposent du multijoueur en local sont supportés.

À la manière de *Controlysis*, il propose une interface développée via UMG avec une visualisation complète du système ainsi qu'une configuration des contrôleurs de jeu via notification d'évènements widgets (un clic souris p.ex.).

Une configuration riche des périphériques qui comprend réassignation des plages de valeurs d'entrées, ajustement sensibilité, deadzone et affectation directe **en jeu** (par opposition à un paramétrage dans l'éditeur d'Unreal Engine) d'un contrôleur à un joueur.

Si l'UI proposée ne convient pas au projet, il est possible de passer par les Blueprints pour réaliser ses propres **templates** visuels.

L'extraction des données d'état des contrôleurs est aussi possible via les fonctions Blueprint fournies.

Hormis le manque de documentation et un surplus imposant de fonctionnalités, l'on espère juste qu'un étalonnage des entrées puisse être possible dans le cas où les valeurs en entrée sont décalées ou erronées.

De manière optionnelle, le plugin n'utilisant que des APIs propres au système Windows, rien n'assure donc qu'il puisse être porté sur d'autres plateformes.

13. API *Raw Input* de Microsoft pour traiter des entrées brutes : <https://learn.microsoft.com/en-us/windows/win32/inputdev/about-raw-input>

14. Le type d'un axe dépend de la plage de valeurs sortante : p.ex. un joystick trouve ses valeurs en entrées dans $[-1, 1]$ alors qu'une manette à propulsion dans $[0, 1]$ pour certains contrôleurs de simulation

3.1.3 **SDL2InputDevice**

Auteur : WisE Digital Reality Lab

Version(s) UE supportée(s) : 4.x

Documentation : Oui

Lien site : <https://www.unrealengine.com/marketplace/en-US/product/sdl2input-device-plugin>

SDL2InputDevice est un plugin du Marketplace qui a été implémenté avec la bibliothèque *SDL*.

Il propose le minimum en permettant de gérer plusieurs contrôleurs de jeu en simultané et une interface code et Blueprint au travers de fonctions code et Blueprint, avec parmi les fonctions fournies la possibilité d'extraire des données via la notification d'évènements sur l'état des contrôleurs connectés ou sur les entrées.

À l'instar de *Controlysis*, il ne semble exister aucun moyen de configuration des contrôleurs (du moins il ne semble pas y avoir d'abstraction à ce niveau).

Cependant, comme il semble fonctionner en accord avec le pipeline de gestion des entrées d'Unreal Engine en lui fournissant les entrées du contrôleur de jeu (associables à des clés d'actions), il est sûrement possible d'invoquer un plugin externe de configuration, comme *Enhanced Input*¹⁵.

Malheureusement, il est à noter un défaut majeur : il semble se baser sur une liste de contrôleurs connus¹⁶. Une base de données certes bien remplie car la majorité des contrôleurs de jeu sur le marché actuel sont supportés, mais aucune certitude sur la prise en charge de vieux contrôleurs comme les tableaux de commandes Navmer.

Une autre particularité est que les valeurs en entrée des contrôleurs sont réceptionnées deux fois à cause d'une supposée concurrence entre la *SDL* et l'API *XInput* utilisée par défaut sur Unreal Engine.

Le plugin impose donc **un identifiant unique de série** pour chaque contrôleur, qui est régit par un paramètre modifiable uniquement dans le code source du plugin.

Dernier point négatif relevé : ce plugin n'a aucun support sur la version 5.0 d'Unreal Engine.

15. *Enhanced Input* : <https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/Input/EnhancedInput/>

16. Liste des contrôleurs de jeu connus (par numéros vendeur et produit) : https://gist.github.com/nondebug/aec93dff7f0f1969f4cc2291b24a3171#file-known_gamepads-txt

3.1.4 JoystickPlugin

Auteurs : JaydenMaalouf, tsky1971, Ikarus76, SamPersson

Version(s) UE supportée(s) : 4.x, 5.0

Documentation : Oui

Lien site : <https://github.com/JaydenMaalouf/JoystickPlugin>

JoystickPlugin est un plugin implémenté à l'aide de la bibliothèque *SDL* et qui a été testé avec les tableaux de commandes Navmer.

C'est un plugin très simple comme *SDL2InputDevice Plugin* qui permet aussi d'utiliser plusieurs contrôleurs de jeu en simultané.

Il est cependant capable de gérer tout type de contrôleur de jeu car il ne dépend pas d'une liste de périphériques connus.

À noter que les contrôleurs de type **retour de force** sont également pris en charge, même si cela a peu d'intérêt dans le cas présent de recherche.

Il n'y a pas vraiment d'interface Blueprint à disposition.

Cependant, la mise en place d'un processus particulier d'Unreal Engine appelé **sous-système**¹⁷ permet de faire appel dans un Blueprint à des fonctions code pour y traiter les connexions d'appareils et données d'entrées.

Un autre point à noter est la possibilité de réassigner les plages de valeurs des axes des contrôleurs connectés dans l'éditeur d'Unreal Engine.

Le plugin s'accorde également au pipeline de gestion des entrées d'Unreal Engine à la manière de *SDL2InputDevice Plugin*.

Les contrôleurs sont différenciés par leurs identifiants matériels.

Afin d'assurer leur unicité dans le cas où plusieurs instances du même produit soient connectés, les contrôleurs se voient assigner **d'un identifiant unique de série** : un numéro qui s'incrémente à chaque connexion.

C'est un problème car lors d'un changement d'état du contrôleur (p.ex. déconnexion/connexion), les configurations des contrôleurs qui s'appliquent au travers des associations clé – entrée (clés qui sont nommées selon l'identifiant unique de série du contrôleur), il n'est pas possible de conserver une configuration pour un contrôleur.

Or si l'on supprime l'identifiant unique de série, on perd l'unicité dans le cas où les différentes instances de contrôleurs se voient dotées des mêmes identifiants matériels (spoiler : c'est le cas pour les deux Ship Consoles).

17. Grossièrement, un procédé permettant d'exposer de manière externe des fonctionnalités "code" : <https://docs.unrealengine.com/4.26/en-US/ProgrammingAndScripting/Subsystems/>

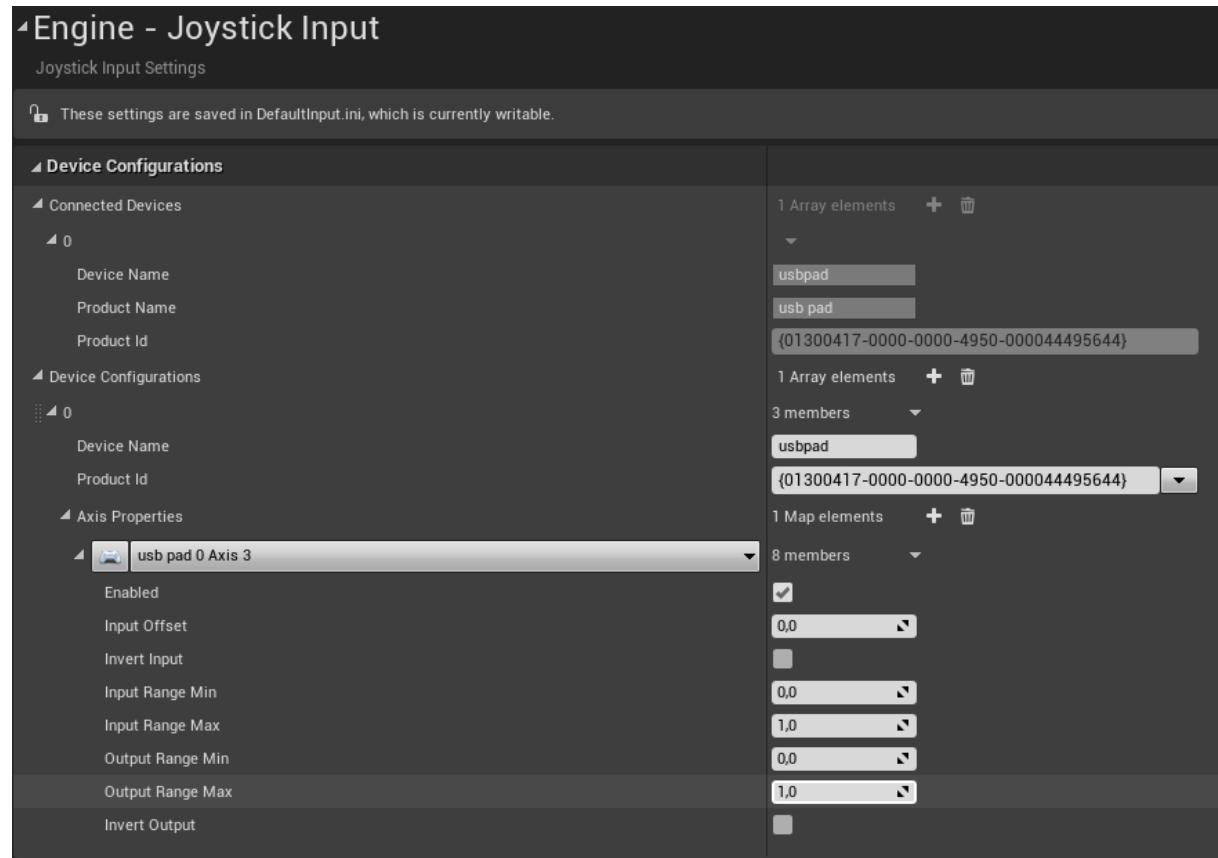


FIGURE 5 – La possibilité de configurer un appareil et les entrées de type “axe”.

```
▶ Click for Mouse Control

Showing Debug for ThirdPersonCharacter_2, Press [PageUp] and [PageDown] to cycle between targets.
P-03024-318551D14B7A
Location: V(0) Rotation: R(0)
Instigator: None Owner: PlayerController_0
P-03024-318551D14B7A
Location: V(X=-932.87, Y=258.36, Z=228.42) Rotation: R(Y=61.44)
Instigator: ThirdPersonCharacter_2 Owner: PlayerController_0
CONTROLLER PlayerController_0 Pawn ThirdPersonCharacter_2
STATE Playing
<<< INPUT STACK >>>
PlayerController_0.GameplayDebug_Input
PlayerController_0.PC_InputComponent0
ThirdPersonExampleMap_C_0.InputComponent_0
ThirdPersonCharacter_2.PawnInputComponent0
INPUT PlayerInput_0
Left Mouse Button: 0.00 (raw 0.00)
usb pad 0 Axis 0: 0.00 (raw 0.00)
usb pad 0 Axis 1: 0.00 (raw 0.00)
usb pad 0 Axis 2: 0.00 (raw 0.00)
usb pad 0 Axis 3: 0.00 (raw 0.00)
usb pad 0 Axis 4: 0.00 (raw 0.00)
usb pad 0 Axis 5: 0.00 (raw 0.00)
usb pad 0 Hat 0 X: 0.00 (raw 0.00)
usb pad 0 Hat 1 Y: 0.00 (raw 0.00)
Mouse X: 0.00 (raw 0.00)
Mouse XY 2D-Axis: 0.00 (raw 0.00)
Mouse Y: 0.00 (raw 0.00)
Z: 0.00 (raw 0.00)
Q: 0.00 (raw 0.00)
S: 0.00 (raw 0.00)
D: 0.00 (raw 0.00)
W: 0.00 (raw 0.00)
Left: 0.00 (raw 0.00)
Right: 0.00 (raw 0.00)
Down: 0.00 (raw 0.00)
Up: 0.00 (raw 0.00)
Space Bar: 0.00 (raw 0.00)
Enter: 0.00 (raw 0.00)
Left Cmd: 0.00 (raw 0.00)
usb pad 0 Button 11: 0.00 (raw 0.00)
usb pad 0 Button 12: 0.00 (raw 0.00)
usb pad 0 Button 13: 0.00 (raw 0.00)
usb pad 0 Button 14: 0.00 (raw 0.00)
usb pad 0 Button 18: 0.00 (raw 0.00)
usb pad 0 Button 17: 0.00 (raw 0.00)
usb pad 0 Button 16: 0.00 (raw 0.00)
> -
```

FIGURE 6 – Il dispose d'un affichage de debug en accord avec Unreal Engine. On peut remarquer l'appareil “usb pad” qui est actuellement traité par JoystickPlugin et ses valeurs d'axes, numérotés de 0 à 5.

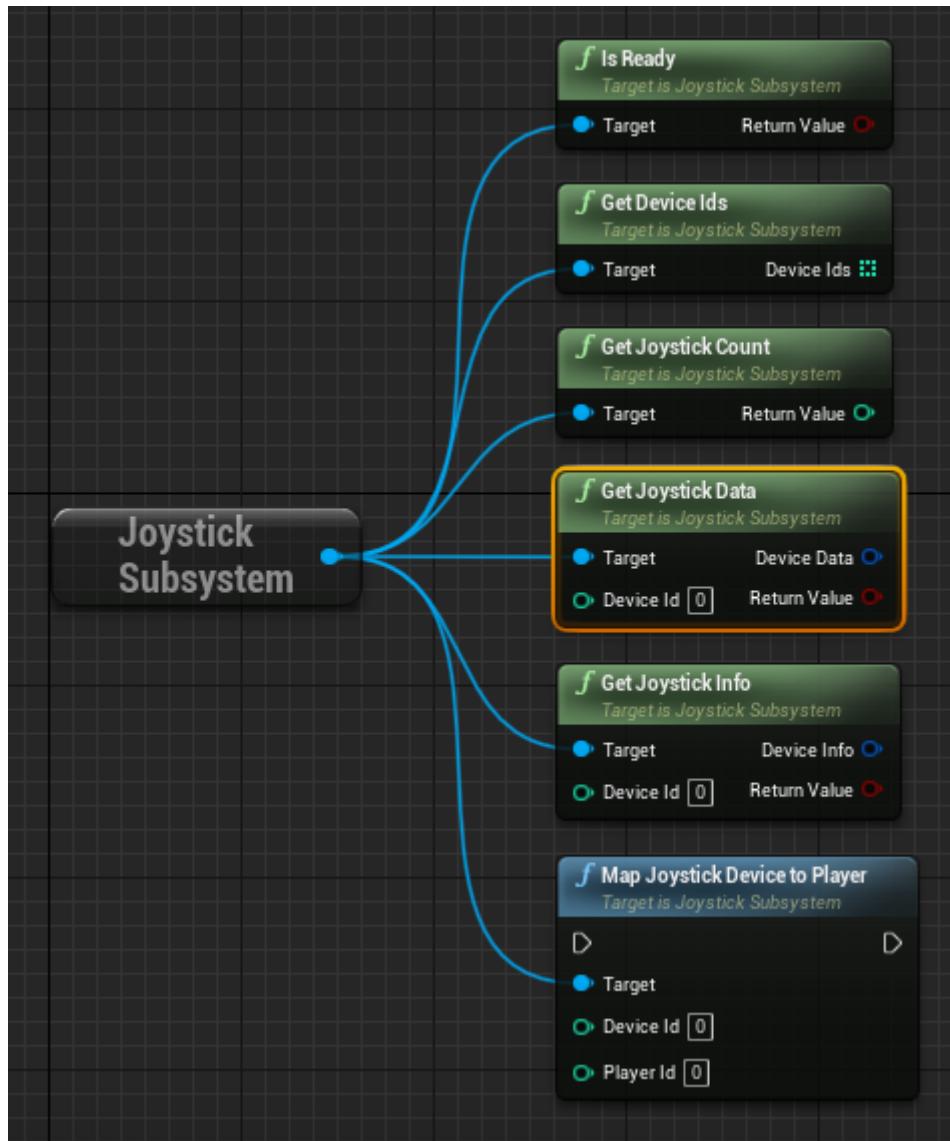


FIGURE 7 – Quelques fonctionnalités du sous-système de JoystickPlugin, utilisables dans un Blueprint.

3.1.5 Conclusion : Les plugins existants

Il existe plusieurs points à prendre en compte dans le choix d'une solution pré-existante.

Les tableaux de commandes ont le défaut d'avoir non seulement une plage de valeur biaisée pour les axes, mais aussi une valeur de centre erronée.

Afin de parvenir au bon fonctionnement de ces périphériques, il est requis dans le plugin d'opérer sur les entrées de manière à pouvoir effectuer un étalonnage précis.

Dans une moindre mesure, l'usage d'une interface de configuration disponible en jeu / en simulation n'est pas de refus, notamment en vue de donner la possibilité de configurer les appareils en dehors de l'éditeur d'Unreal.

NOTE

Dans la version "distribuée" d'un projet, l'éditeur d'Unreal n'est évidemment pas disponible.

Dans un souci de maintenabilité, il est intéressant de favoriser une API de gestion des entrées efficace et facile à apprivoiser, de même qu'il est intéressant d'avoir des fonctionnalités qui s'accordent avec les principes et les services offerts par Unreal.

Pour permettre à terme une meilleure disposition du "poste de pilotage", le plugin doit être capable de traiter plusieurs contrôleurs en même temps.

Optionnellement, l'outil d'intégration serait compatible avec plusieurs plateformes (Windows, Linux, etc.) : sur ce point, le choix de l'API exploitée est déterminant.

Les quelques plugins présentés ont un fort potentiel à prendre en charge les tableaux de commandes Navmer.

On citera leurs points notables dans un classement de satisfaction (i.e. du plus à même à répondre aux exigences au plus éloigné) :

1	<i>WMIInputManager</i>	<ul style="list-style-type: none">— Une utilisation combinée des APIs <i>Raw Input</i> et <i>XInput</i> qui en fait un plugin assez puissant capable de détecter n'importe quel périphérique d'entrées— La capacité à prendre en charge plusieurs contrôleurs en simultané— Une configuration riche des entrées des contrôleurs— La présence de multiples abstractions pour le traitement des entrées, et d'une interface de visualisation et de configuration en jeu très développée— La compatibilité avec un projet multijoueur en local— Plugin supporté sur Unreal Engine 5.0
2	<i>JoystickPlugin</i>	<ul style="list-style-type: none">— Utilisation de la bibliothèque multiplateforme SDL— Capacité à détecter n'importe quel périphérique d'entrées et à prendre en charge plusieurs contrôleurs en simultané, et même des équipements identiques— Open source— Fonctionnalités en accord avec le pipeline d'Unreal et utilisables dans des blueprints— Prise en charge des appareils à retour de force— Plugin supporté sur Unreal Engine 5.0
3	<i>Controlysis</i>	<ul style="list-style-type: none">— Utilisation de la bibliothèque multiplateforme SDL— Capacité à détecter n'importe quel périphérique d'entrées et à prendre en charge plusieurs contrôleurs en simultané, et même des équipements identiques— Fonctionnalités Blueprint pour traiter les données d'entrées— Présence d'une interface de visualisation des contrôleurs connectés pour l'utilisateur— Plugin supporté sur Unreal Engine 5.0
4	<i>SDL2InputDevice</i>	<ul style="list-style-type: none">— Utilisation de la bibliothèque multiplateforme SDL— Fonctionnalités Blueprint pour traiter les données d'entrées— Fonctionnalités en accord avec le pipeline d'Unreal et utilisables dans des blueprints

En conclusion, il n'y a potentiellement que les trois premiers plugins du classement qui sont à même de pouvoir répondre aux exigences des tableaux de commande Navmer.

On retiendra les plugins *WMIinputManager* et *JoystickPlugin* qui se trouvent être, parmi la sélection, les plus complets et les plus généreux quant aux détails sur leurs fonctionnalités.

Quant au plugin *SDL2InputDevice* : l'utilisation d'une liste de contrôleurs connus le place en consé-

quence hors jeu.

Cependant, il est nécessaire de rappeler les problèmes propres aux Ship Consoles (cf. la sous-section **La Ship Console**).

Les Ship Consoles ont la particularité d'avoir des données d'identification identiques.

Ce qui peut poser problème aux APIs sous-jacentes de traitement des entrées, du fait qu'il est impossible de différentier l'une de l'autre (cf. la section **Le plugin NavmerInput** pour plus de précision à ce propos).

En conséquence, promettre une configuration dissociée des Ship Consoles qui persiste après la déconnexion de ces appareils n'est à première vue pas envisageable de manière classique.

On comprend alors que ces complications, au coeur de la problématique d'intégration des tableaux de commande, sont bien trop particulières pour que les solutions pré-existantes les prennent en compte.

En conséquence, il est nécessaire d'implémenter un plugin personnel ou de reprendre la structure d'un plugin existant si faisable.

3.1.6 RawInput Plugin

Auteur : Epic Games, Inc.

Lien site : <https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/Input/RawInput/>

NOTE

Ce plugin est présenté de manière annexe et brève car il a été conclu suite à des tests complets qu'il n'est pas en capacité d'intégrer les tableaux de commandes.

Bien qu'il ne soit pas viable pour l'intégration des tableaux de commandes Navmer, il est purement présenté à titre indicatif car c'est le premier plugin sur lequel cette recherche s'est orientée, de par le fait qu'il est fourni avec le moteur.

RawInput Plugin est un plugin officiel gratuit et natif (déjà fourni avec le moteur Unreal) implémenté avec l'API *Raw Input* de Windows qui permet l'utilisation d'un contrôleur générique sur Unreal Engine.

Il se décrit comme "expérimental", ie. qu'il ne convient pas à une utilisation sur des projets complexes : il souffre de plusieurs manques cruciaux et de défauts de conception vraisemblablement dûs à un développement hâtif.

Aux premiers abords, il semble intéressant de par son accès et sa simplicité d'utilisation.

Le contrôleur est identifié au moyen de ses identifiants matériels, chaque contrôleur connecté est configurable (dans une certaine mesure) suivant ses identifiants matériels, et les entrées sont assignables à des clés d'actions.

Cependant, lorsque l'on est amené à augmenter les nécessités, on arrive rapidement à des défauts élémentaires.

Dû à sa conception, le plugin est par exemple dans l'incapacité de gérer plusieurs contrôleurs en même temps.

Si l'on souhaite tout de même brancher plusieurs contrôleurs à la même station, les valeurs sondées aux entrées se retrouvent mélangées. Cela s'explique plus précisément par le fait que le plugin, dans la construction de son **interface**, gère un seul contrôleur, mais les appels à l'API sous-jacente permettent de sonder les entrées de plusieurs contrôleurs.

Également, la configuration d'un contrôleur se résume seulement à l'assignation des entrées à des clés actions, le changement du type d'un axe et la possibilité d'appliquer un décalage dans la valeur d'un axe. Il n'y a aucun moyen d'effectuer un étalonnage sur les axes, et donc il n'est pas possible de régler les problèmes spécifiques aux tableaux de commandes Navmer.

4 Le plugin NavmerInput

Le plugin *NavmerInput* est issue d'un retravail du plugin *JoystickPlugin* (cf. sous-section **JoystickPlugin**).

Intégrer NavmerInput à un projet permet le bon fonctionnement des tableaux de commandes Navmer, là où Unreal Engine n'en est pas capable autrement.

Il utilise en tant que module **third-party** la bibliothèque *SDL2* et son sous-système *Input Events/Joystick* pour écouter les évènements contrôleurs.

Le retravail implique **une réassigntion des plages de valeurs des données d'entrées** sur deux temps afin de pallier au décentrage des axes des tableaux de commandes (cf. section **La Ship Console**).

De manière optionnelle, il y est également compris une suite de Blueprints pour intégrer un menu utilisable pendant la simulation permettant de changer à souhait les configurations d'appareils pré-définies dans l'éditeur d'Unreal Engine.

Le principe du plugin NavmerInput repose sur un système d'association de configurations avec les appareils.

On associe un ou plusieurs appareil(s) de même instance à une **configuration d'appareils**, qui est également associée à un ensemble de paramètres définies dans une **page de configuration**.

Ces configurations d'appareils (ou “Device Configurations” dans le jargon du plugin) sont une manière d'enregistrer une certaine configuration sur un appareil connecté, même après la déconnexion de celui-ci ou la fermeture du projet.

Quant aux pages de configurations (ou “Configuration Pages” dans le jargon du plugin), elles sont une manière d'obtenir une certaine unicité des appareils et de permettre un traitement intermédiaire des valeurs en entrée des axes (notamment une réassigntion des plages de valeurs pour les axes des consoles), entre la *SDL* et Unreal Engine.

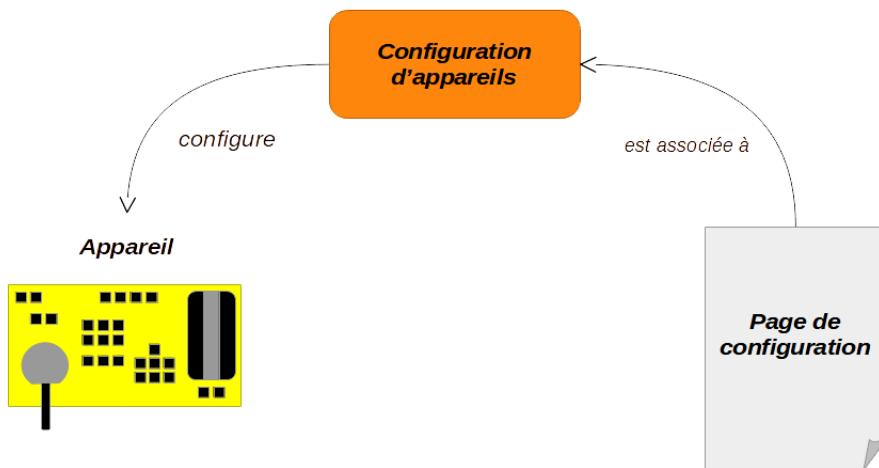


FIGURE 8 – Schéma des liens d'associations pour le système employé par le plugin NavmerInput.

Normalement, un appareil se verrait simplement associé à une configuration.

Un appareil est reconnu par le système selon ses identifiants matériels définis par le constructeur : le **VendorID** et le **ProductID**.

Dans le cas où deux appareils ont des identifiants matériels identiques (p.ex. les Ship Consoles), ils sont considérés comme étant un seul et même appareil.

Généralement pour différentier ces **instances**, on leur assigne un identifiant logiciel, selon leur ordre de connexion par exemple.

Malheureusement, dans le cas particulier des Ship Consoles, celles-ci souffrent de problèmes similaires au niveau matériel (cf. **La Ship Console**), mais qui diffèrent sur certains points (p.ex. les bornes au valeurs d'axes ne sont pas les mêmes).

En partant du principe que les appareils sont dissociés par un identifiant logiciel défini sur leur ordre de connexion : l'ordre de connexion des appareils peut changer, ce qui implique qu'il n'est pas possible d'établir une configuration persistante en alignement avec ce principe.

Les identifiants matériels et les valeurs d'entrées ne sont pas les seules données échangées entre un appareil et le système.

Une solution plutôt évidente serait de différentier les appareils sur certaines données échangées qui présentent des différences notables.

Par exemple, dans le cas des Ship Consoles, les plages de valeurs des axes et **les pages d'usages** (cf. la sous-section **Les périphériques HID**) sont des données exploitables car elles présentes des différences entre les deux consoles.

Cependant cette méthode met en péril la robustesse à terme, dans le cas où une n-ième Ship Console

ne satisferait plus la condition d'unicité de ces quelques données : il y a trop peu de données significativement différentes.

Le système actuel du plugin NavmerInput, plus manuel, utilise une configuration d'appareils comme un intermédiaire sur lequel va être associé une page de configuration, définie pour une seule instance d'appareil.

Il demande à l'utilisateur de lier la bonne page de configuration pour le bon appareil lorsqu'il connecte un appareil qui présente plusieurs instances.

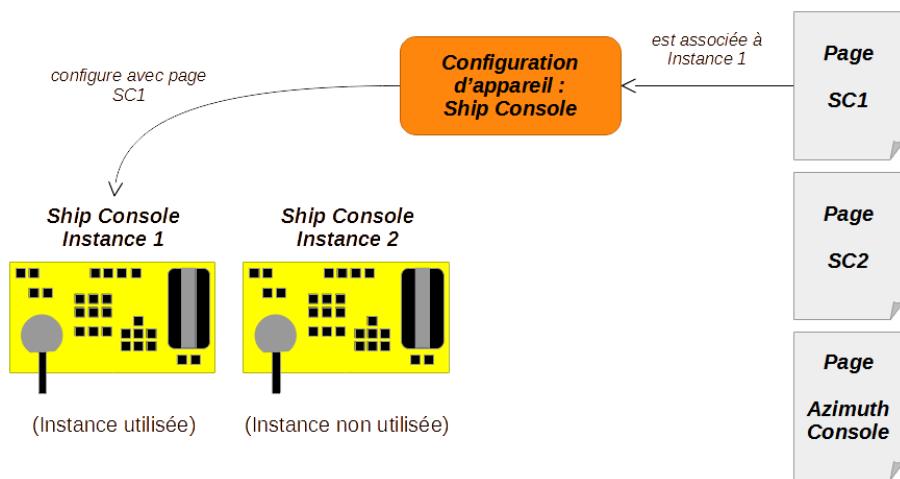


FIGURE 9 – Schéma final des liens d'associations. Plusieurs instances d'un même appareil peuvent se voir configurée sur des pages différentes. C'est l'utilisateur qui choisit quelle page doit être appliquée à l'appareil (en fonction de l'instance utilisée).

La figure suivante montre l'utilisation classique du plugin avec le simulateur Navmer3D :

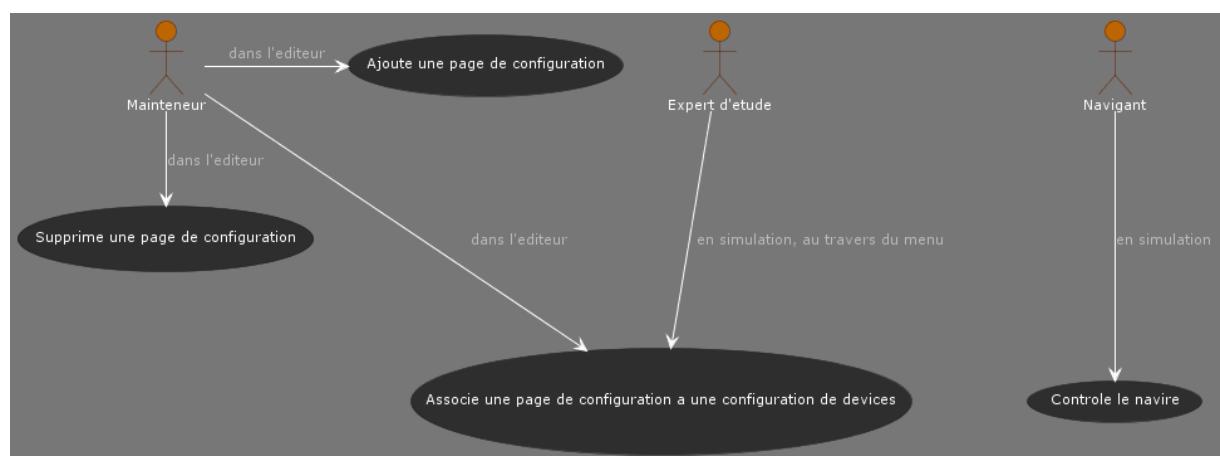


FIGURE 10 – Cas d'utilisation simple du plugin.

4.1 Directives d'emploi

Le plugin NavmerInput est un plugin standalone : il exige simplement d'être installé dans un projet.

Il est cependant nécessaire de bien configurer les tableaux de commandes dans l'éditeur après son installation.

Si l'on souhaite utiliser la suite de blueprints à disposition, une certaine mise en place est également nécessaire.

L'utilisation du plugin en jeu se fait au travers du menu intégré, tandis que dans l'éditeur elle se fait dans la page de paramètres dédiée au plugin.

NOTE

Pour le simulateur Navmer3D, les parties **Installer le plugin**, **Configurer les tableaux de commandes** et **Installer le menu de configuration** ne nécessitent pas d'être suivies car le simulateur contient déjà le plugin installé. De plus, comme certains fichiers du plugin sont inclus dans le code du projet Navmer3D, désinstaller NavmerInput pourrait en conséquence mettre en échec sa compilation. Seule la partie **Utiliser le menu de configuration** reste utile.

4.1.1 Installer le plugin

Insérez tout d'abord le répertoire **NavmerInput** dans le répertoire **Plugins** situé à la racine du projet.

Si le répertoire **Plugins** est inexistant, créez-le à la racine du projet en respectant la casse et le “s” final dans le nom.

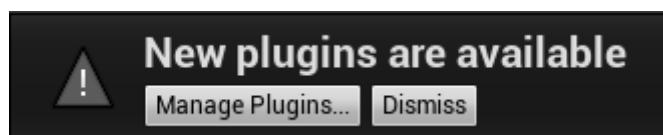
Ceci fait, partant de la racine du projet, ouvrez le fichier “*Source/NOM_DU_PROJET/NOM_DU_PROJET.Build.cs*” et insérez les lignes suivantes (ou rajoutez ce qu'il manque si les lignes existent déjà) :

```
1 PublicDependencyModuleNames.AddRange(new string[] { "Core", "CoreUObject", "Engine", "InputCore", "JoystickPlugin" });
2 PublicIncludePaths.AddRange(new string[] {"JoystickPlugin/Public", "JoystickPlugin/Classes"});
```

Copiez maintenant le contenu du fichier **DefaultInput.ini** situé à la racine du plugin dans le fichier “*Config/DefaultInput.ini*” du projet (partant de la racine).

Ouvrez le projet.

Un message apparaît vous avertissant qu'un ou plusieurs plugins ont été associés au projet :



Cliquez directement sur le bouton “Manage Plugins...” pour qu'il vous ouvre la fenêtre des plugins.

Cette fenêtre est originellement accessible via *Edit > Plugins* dans la barre menu de l'éditeur.

Cherchez “NavmerInput”.

Si la check box “Enabled” est cochée, vous en avez terminé. Autrement, cochez là pour activer le plugin.

Un message sur fond jaune apparaît alors vous proposant de redémarrer Unreal.

Procédez ainsi.

Si Unreal redémarre correctement, alors le plugin est normalement installé.

Enfin, pour vérifier la bonne installation, il existe deux façons :

1. Dans l'éditeur, allez dans *Edit > Project Settings*, et cherchez sous la catégorie *Engine* la page de paramètres **Joystick Input** (page de paramètres du plugin).
2. Dans l'éditeur, allez dans *Window > Developer Tools*, cochez l'outil **Output Log**. Recherchez par filtre dans la fenêtre *Output Log* le mot “LogJoystickPlugin”. Vous devriez voir apparaître les messages suivants :

```
1 LogJoystickPlugin: DeviceSDL Starting
2 LogJoystickPlugin: DeviceSDL::InitSDL() SDL init 0
3 LogJoystickPlugin: DeviceSDL::InitSDL() SDL init subsystem joystick
4 LogJoystickPlugin: DeviceSDL::InitSDL() SDL init subsystem joystick
```

4.1.2 Configurer les tableaux de commandes

⚠ ATTENTION !

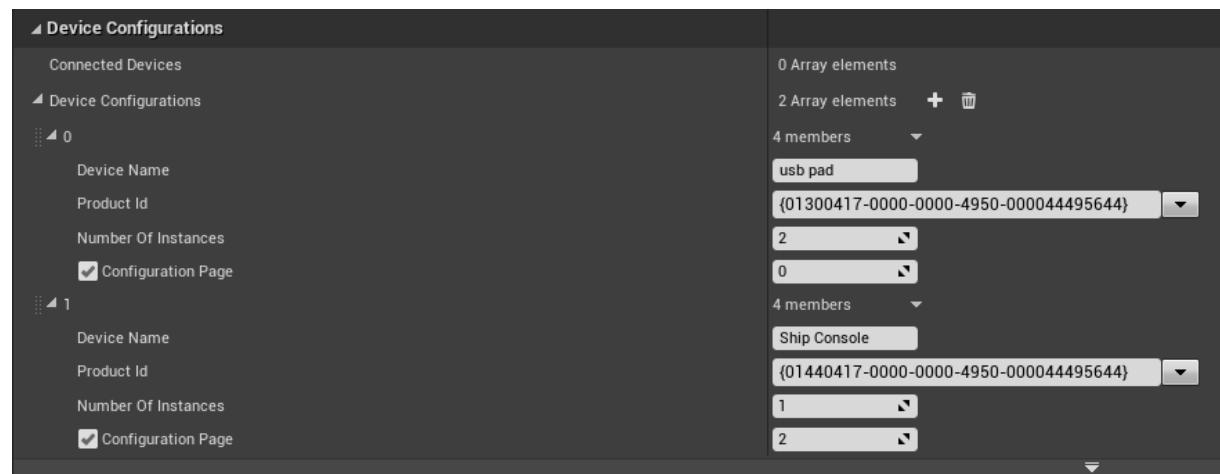
Dans cette partie, l'exemple ne figure que le cas d'une seule console.

Cependant les manipulations qui suivent restent similaires pour toutes les consoles : il vous faudra réitérer les démarches pour chacune d'entre elles.

Ouvrez tout d'abord la fenêtre des paramètres de projet via *Edit > Project Settings* dans la barre menu de l'éditeur.

Rendez vous dans la page de paramètres Joystick Input sous la catégorie Engine.

Pour avoir inséré le bout de code dans le fichier **DefaultInput.ini**, vous devriez voir en résultat sous la rubrique **Device Configurations** les éléments “usb pad” et “Ship Console”.



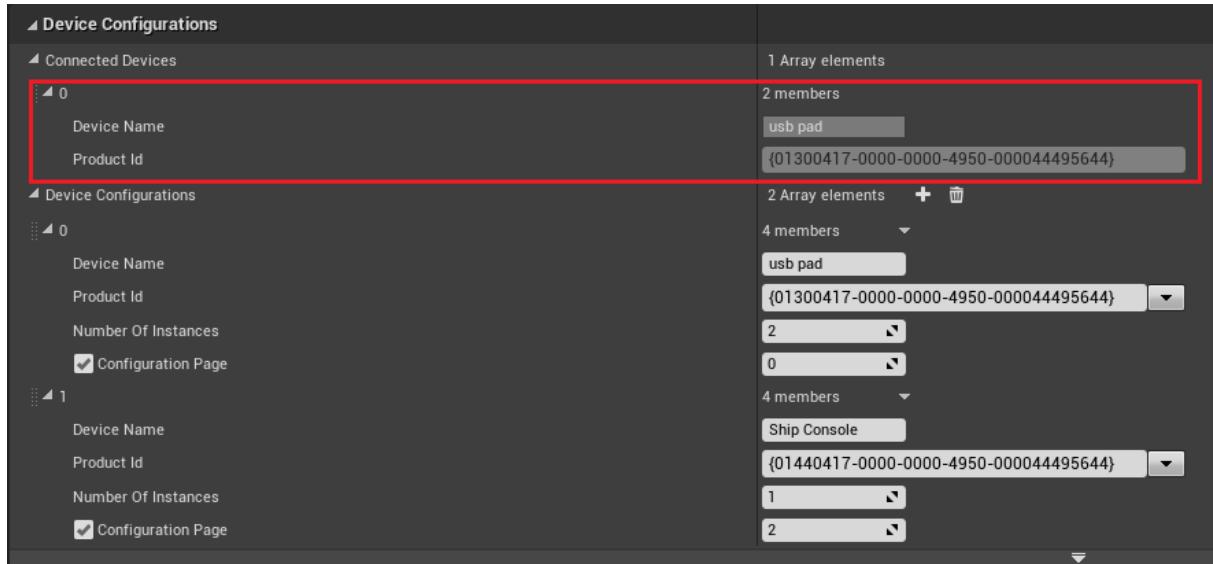
Ces éléments correspondent respectivement aux **configurations d'appareils** pour les Ship Consoles (usb pad) et pour l'Azimuth Console (Ship Console).

📝 NOTE

Aussi étrange que cela soit, les tableaux de commandes sont reconnus par le système d'exploitation comme tel.

Connectez maintenant la Ship Console I.

Vous pouvez alors apercevoir dans la rubrique **Connected Devices** un appareil sous le nom de “usb pad”, avec un Product Id.



L'intérêt du fichier DefaultInput.ini est de conserver les paramètres saisis dans la page de paramètres Joystick Input, même après la fermeture du projet.

L'échange entre le fichier et l'application se fait dans les deux sens : il est possible de modifier le contenu du fichier en dehors de l'éditeur pour que celui-ci récupère les informations à l'ouverture du projet.

Bien que ces fichiers soient propres à la version éditeur, ils sont également présents dans la version packagé (déploiement du projet sous forme d'executable).

Dans le cas où un nouvel appareil nécessite d'être configuré :

1. Ajoutez un élément à la rubrique Device Configurations (bouton au symbole “+” au même niveau que le nom de la rubrique).
2. Connectez ledit appareil pour qu'il soit visible dans la rubrique Connected Devices.
3. Copiez les champs **Device Name**¹⁸ et **Product Id**¹⁹ visibles dans les champs identiques du nouvel élément.
4. Laissez les autres champs libres pour le moment.

En dessous de la rubrique Device Configurations se trouve une barre avec au centre une flèche pointant vers le bas.

Cliquez sur la barre pour découvrir les paramètres avancés.

Vous pouvez alors apercevoir la rubrique **Configuration Pages**.

18. Le nom d'un appareil connecté donné par le système.

19. Dans le plugin, cela correspond à un nombre unique composé des identifiants matériels d'un appareil connecté (VendorID et ProductID).

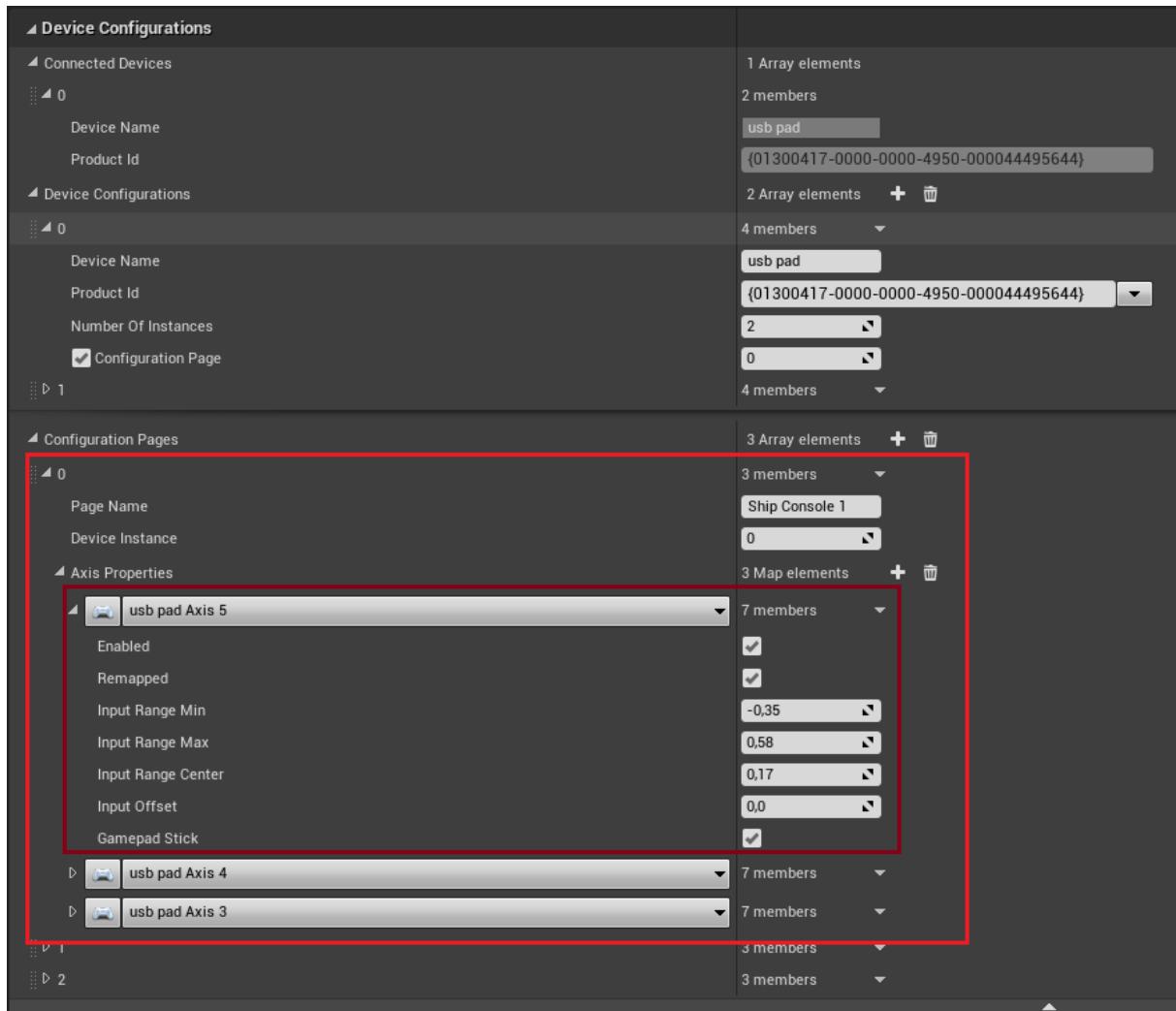


FIGURE 11 – Encadré en rouge : la page de configuration d'indice 0. Encadré en rouge foncé : les paramètres de l'axe d'indice 5 pour la Ship Console 1.

Chaque page de configuration porte un nom, qui diffère de celui de la configuration d'appareils.

Pour chaque page de configuration, il est possible de paramétriser les axes pour une configuration d'appareils au moyen de :

- La clé²⁰ de l'axe sur lequel les paramètres vont s'appliquer
- Le **remapping** de l'axe (i.e. la réassignation de la plage de valeurs) en indiquant les champs **Input Range Min**, **Input Range Max** et **Input Range Center** correspondants aux bornes et au milieu de la plage de valeurs brute de l'axe
- L'**offset** sur les valeurs de l'axe en indiquant le champ **Input Offset** (laisser à zéro si non utilisé)

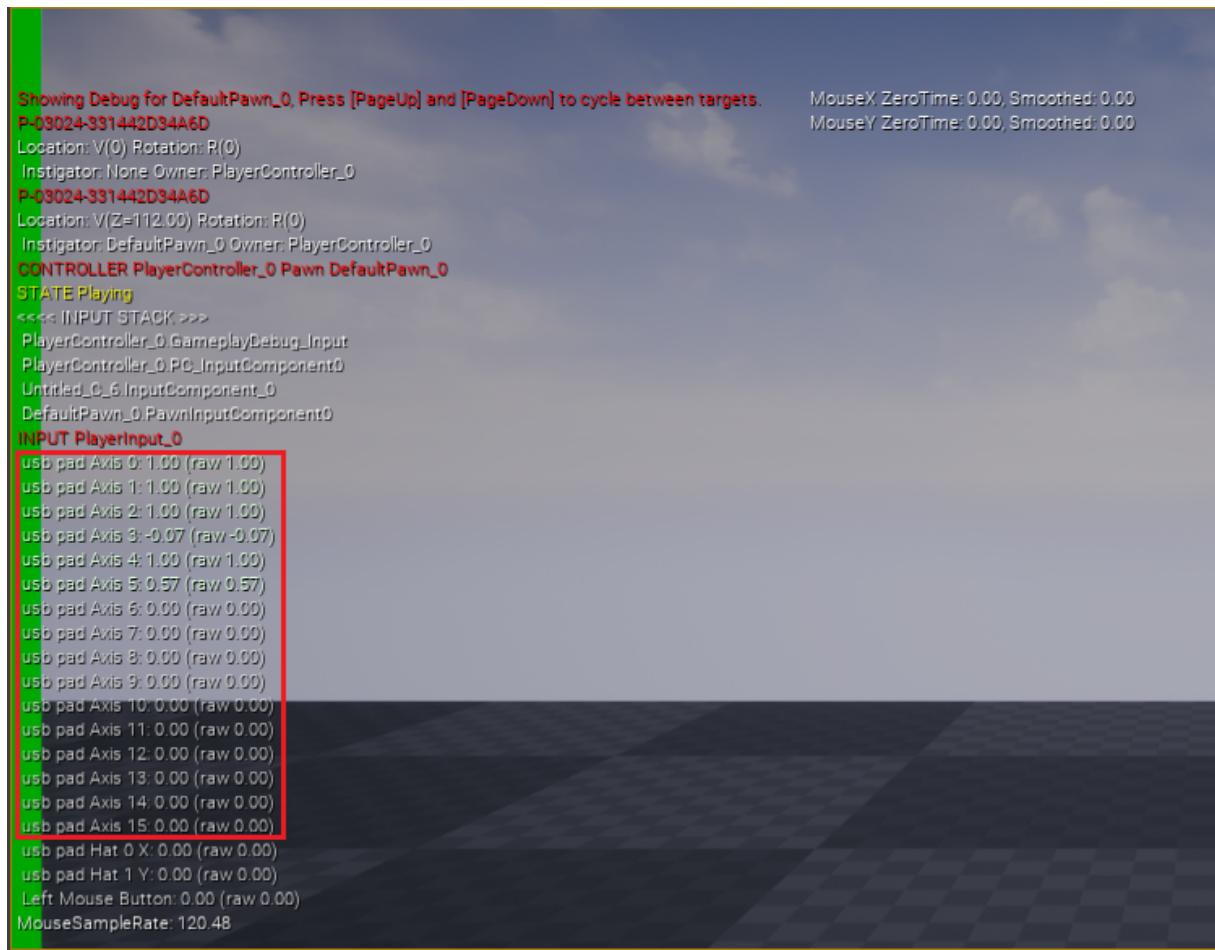
20. Une association clé – valeur d'entrée propre à Unreal Engine pour associer de manière efficace des actions/comportements à un appui de bouton ou au déplacement d'un axe produit par l'utilisateur.

- Le type de plage de valeurs que l'axe doit prendre ($[0, 1]$ ou $[-1, 1]$) en cochant le champ **Gamepad Stick** ($[0, 1]$ par défaut)

Pour connaître les valeurs à rentrer dans les champs **Input Range Min**, **Input Range Max**, **Input Range Center** et/ou **Input Offset**, il faut pour cela :

1. Lancer le projet dans la fenêtre principale de l'éditeur en appuyant sur le bouton **Play**
2. Appuyer sur la touche ² pour ouvrir la console de debug (non disponible dans la version packagé)
3. Taper “ShowDebug INPUT” et presser la touche Entrée

Un affichage comportant différentes informations apparaît alors.



```
Showing Debug for DefaultPawn_0. Press [PageUp] and [PageDown] to cycle between targets.
P-03024-331442D34A6D
Location: V(0) Rotation: R(0)
Instigator: None Owner: PlayerController_0
P-03024-331442D34A6D
Location: V(Z=112.00) Rotation: R(0)
Instigator: DefaultPawn_0 Owner: PlayerController_0
CONTROLLER PlayerController_0 Pawn DefaultPawn_0
STATE Playing
<<< INPUT STACK >>>
PlayerController_0 GameplayDebug_Input
PlayerController_0.PG_InputComponent0
Untitled_C_6.InputComponent_0
DefaultPawn_0.PawnInputComponent0
INPUT PlayerInput_0
usb pad Axis 0: 1.00 (raw 1.00)
usb pad Axis 1: 1.00 (raw 1.00)
usb pad Axis 2: 1.00 (raw 1.00)
usb pad Axis 3: -0.07 (raw -0.07)
usb pad Axis 4: 1.00 (raw 1.00)
usb pad Axis 5: 0.57 (raw 0.57)
usb pad Axis 6: 0.00 (raw 0.00)
usb pad Axis 7: 0.00 (raw 0.00)
usb pad Axis 8: 0.00 (raw 0.00)
usb pad Axis 9: 0.00 (raw 0.00)
usb pad Axis 10: 0.00 (raw 0.00)
usb pad Axis 11: 0.00 (raw 0.00)
usb pad Axis 12: 0.00 (raw 0.00)
usb pad Axis 13: 0.00 (raw 0.00)
usb pad Axis 14: 0.00 (raw 0.00)
usb pad Axis 15: 0.00 (raw 0.00)
usb pad Hat 0 X: 0.00 (raw 0.00)
usb pad Hat 1 Y: 0.00 (raw 0.00)
Left Mouse Button: 0.00 (raw 0.00)
MouseSampleRate: 120.48
```

Si votre appareil est connecté et que vous pressez les différents boutons ou bougez les axes : ses données d'entrées devraient être affichées.

Dans le cas des périphériques traités par le plugin NavmerInput, les informations sont affichées comme suit (de gauche à droite) :

- Le nom de l'appareil (Device Name)

- Le nom de la clé pour l'entrée (p.ex. *Axis 0 à 7* pour les axes)
- La valeur de l'entrée (post-traitement par le plugin, sauf si non configuré)
- La valeur d'origine de l'entrée (pré-traitement par le plugin)

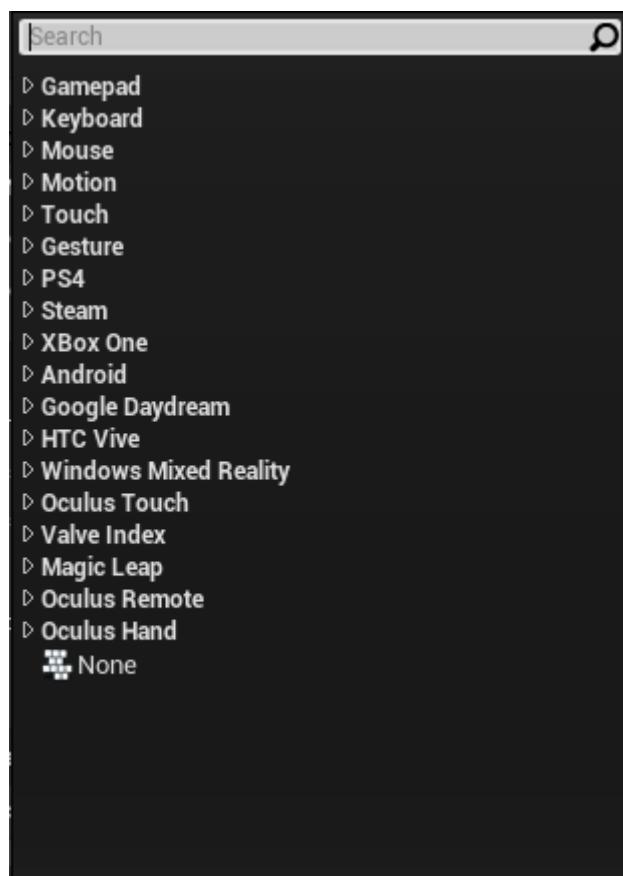
Il vous suffit donc à partir des valeurs affichées de tester, noter et renseigner dans votre page de configuration les clés d'axes qui fournissent une réponse aux mouvements ainsi que les valeurs pour les différentes positions de vos axes (valeur minimale, valeur maximale et valeur centrale).

Pour enlever l'affichage de debug : réouvez la console, tapez simplement “ShowDebug” et pressez la touche Entrée.

Dans le cas où un axe doit être rajouté à une page de configuration : cliquez simplement sur le bouton au symbole “+” au même niveau que la liste d'axes **Axis Properties**.

Pour chaque appareil identifiable, il est possible de paramétriser un nombre de **8 clés d'axes** par défaut, **30 clés de boutons** et de **2 clés d'axes de tête**.

Lorsque vous modifiez une clé en cliquant dessus en vu de paramétriser un axe, une liste de catégories apparaît alors.



Il est nécessaire de connecter un appareil pour que les clés soient créées dans Unreal si elles n'existent pas.

Les entrées des tableaux de commandes se trouvent dans la catégorie **Gamepad**. Autrement vous pouvez les rechercher en tapant le Device Name dans la barre de recherche.

Une page de configuration est précisément associée à une configuration d'appareils.

Il est donc intéressant de définir une page de configuration pour chaque appareil à connecter, et de paramétriser les axes associés à ces appareils.

Cependant une configuration d'appareils peut englober plusieurs appareils, car comme précédemment mentionné : il est possible d'avoir plusieurs appareils qui sont indiqués comme étant des produits identiques (i.e. qu'ils portent le même Device Name et le même Product Id).

Cela à pour conséquence que les appareils reconnus identiques partageront la même configuration, et donc la même plage de clés d'axes. Les clés d'axes utilisées peuvent également différer selon l'instance d'appareil, de même que les entrées affectées.

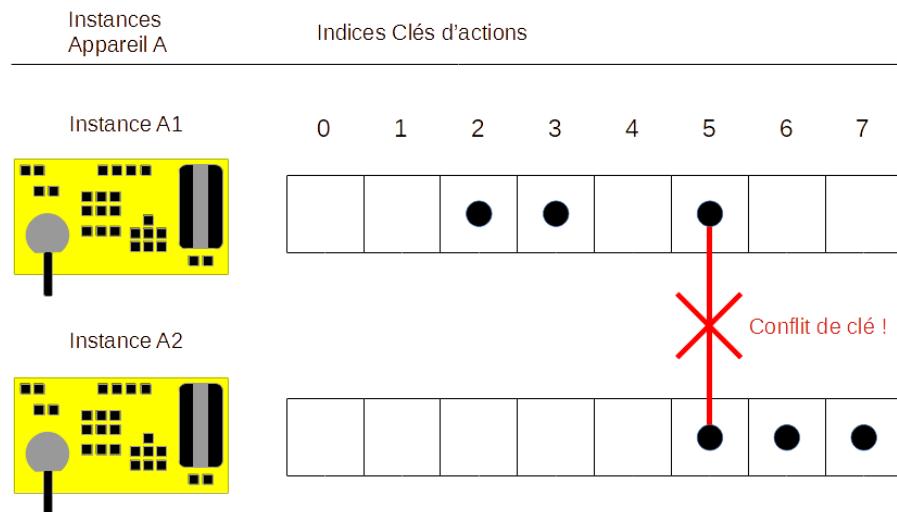


FIGURE 12 – Exemple de plages de clés d'actions des axes entre deux instances d'un même appareil. Ici, l'axe d'indice 5 est utilisé sur les deux instances, cependant l'instance A1 utilise le gouvernail, alors que l'instance A2 utilise une manette de gaz pour l'axe 5.

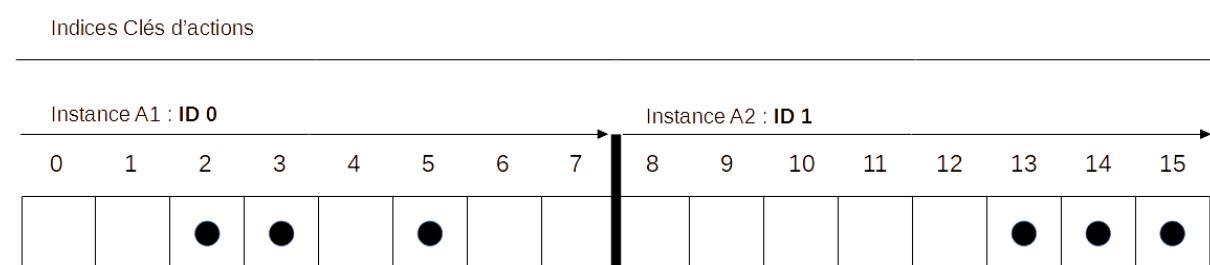


FIGURE 13 – Une solution consiste à augmenter le nombre de clés d'axes possibles et d'utiliser une plage de clés différente selon l'instance d'appareil.

Plus généralement, l'instance d'indice i enverra ses données d'axes sur la plage de clés d'axes $[8i, 8(i + 1) - 1]$.

NOTE

Il est tout de même important de noter que deux appareils reconnus identiques pourront fonctionner en simultané, mais ne pourront pas avoir une configuration unique, car une page de configuration est uniquement associée à une configuration d'appareils.

Le champ **Device Instance** dans une page de configuration permet de renseigner à quelle instance d'une configuration d'appareils doit s'appliquer la page.

Il faut renseigner pour cela l'indice de l'instance (p.ex. pour 2 instances d'une configuration d'appareils l'instance 1 porte l'indice 0, et plus généralement l'indice vaut $NUMERO_INSTANCE - 1$).

Il faut également changer dans la rubrique **Device Configurations** le nombre d'instances d'un appareil via le champ **Number Of Instances**.

Enfin, le champ **Configuration Page** dans une configuration d'appareils permet d'indiquer quelle page est associée à la configuration.

Il suffit pour cela de renseigner l'indice de la page de configuration souhaitée (les indices sont marqués sur chaque page de configuration dans la rubrique Configuration Pages).

Si aucune paramétrisation n'est souhaitée, il suffit de décocher la case à gauche du champ (celle-ci est automatiquement décochée s'il n'existe aucune page de configuration).

À priori, après avoir rajouté le contenu du fichier DefaultInput.ini du plugin, il n'est pas nécessaire de configurer les tableaux de commandes existants car tous les paramètres sont déjà renseignés.

Cependant, si un nouvel appareil différent des autres (i.e. Device Name et/ou Product Id différents) doit être ajouté, il sera alors nécessaire de paramétrier ses axes dans une nouvelle page de configuration, et d'associer la page de configuration à une nouvelle configuration d'appareils.

Une page de configuration peut être ajoutée en cliquant sur le bouton au symbole "+" au même niveau que la rubrique Configuration Pages.

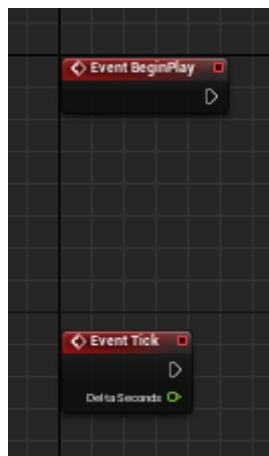
4.1.3 Installer le menu de configuration

Si une configuration doit se faire en jeu, il est nécessaire de passer par le menu intégré.

Ce menu sous forme de widget UMG²¹ nécessite cependant d'être appelé et lié avec d'autres composants au préalable.

Pour ce faire, rendez vous d'abord dans le blueprint d'un niveau (celui de votre choix, où vous souhaitez faire apparaître le menu en soi) en cliquant sur *Blueprints > Open Level Blueprints* dans la fenêtre principale de l'éditeur.

Sur un blueprint vierge, vous devriez n'avoir que les deux noeuds suivants : **Event Begin Play** et **Event Tick**.



Pour commencer, il faut faire appel au widget du menu à l'initialisation du niveau, i.e. à partir du noeud Event Begin Play.

Créez un noeud **Create Widget** à partir du menu des noeuds blueprints accessible en faisant un clique droit souris dans le viewport du graphe.

Dans ce noeud, associez l'attribut **Class** au blueprint *WBP_NavmerInputMenu* en cliquant sur la listbox à côté.

Associez l'attribut **Owning Player** au noeud **Get Player Controller**, avec l'attribut **Player Index** à 0.

Fixez ensuite le noeud Event Begin Play au nouveau noeud **Create WBP_NamverInputMenu Widget** qui remplace Create Widget.

Tirez un lien depuis la valeur de retour **Return Value** pour affecter une nouvelle variable en choisissant "Promote to variable" dans le menu d'ajout de noeuds.

Nommez la "NavmerInputMenu".

21. *Unreal Motion Graphics* : le système de création d'UI, à partir d'éléments graphiques appelés "Widgets"

Liez le noeud **Create WBP_NamverInputMenu Widget** au noeud **Set**.

Un autre widget est livré avec le menu de configuration.

Cet autre widget permet d'afficher en jeu des **messages d'informations** sur les évènements produits par la connexion d'un appareil, ou par un changement produit sur les configurations d'appareils à parti du menu de configuration.

Pour inclure ce widget, répétez les actions précédentes : créez un noeud Create Widget avec l'attribut Class associé au blueprint *WBP_NamverInputMessages*, et appelez votre variable "NavmerInputMessages".

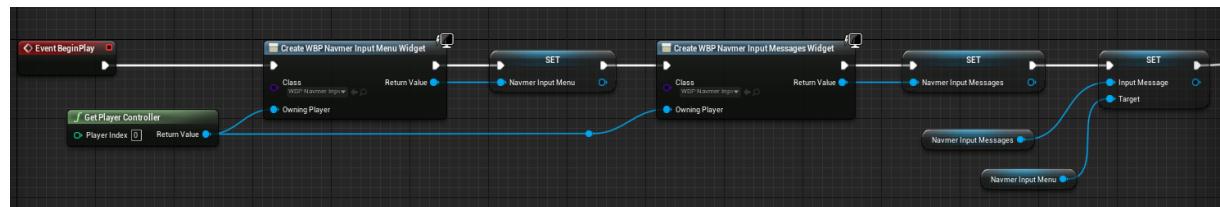
Ceci-fait liez le premier noeud Set au noeud **Create WBP_NamverInputMessages Widget**.

Vous devrez ensuite associer le menu au widget de messages en donnant au menu une référence vers le widget.

Créez les noeuds **Get NavmerInputMenu** et **Get NavmerInputMessages**.

À partir du noeud **NavmerInputMenu**, créez le noeud **Set Input Message** et associez y l'attribut **Input Message** avec le noeud **NavmerInputMessages**.

À ce niveau, si tout est établi correctement, vous devriez vous retrouver avec le graphe suivant :



Il ne reste plus qu'à attacher les deux widgets au viewport présent en jeu.

Créez un noeud **Add to Viewport**.

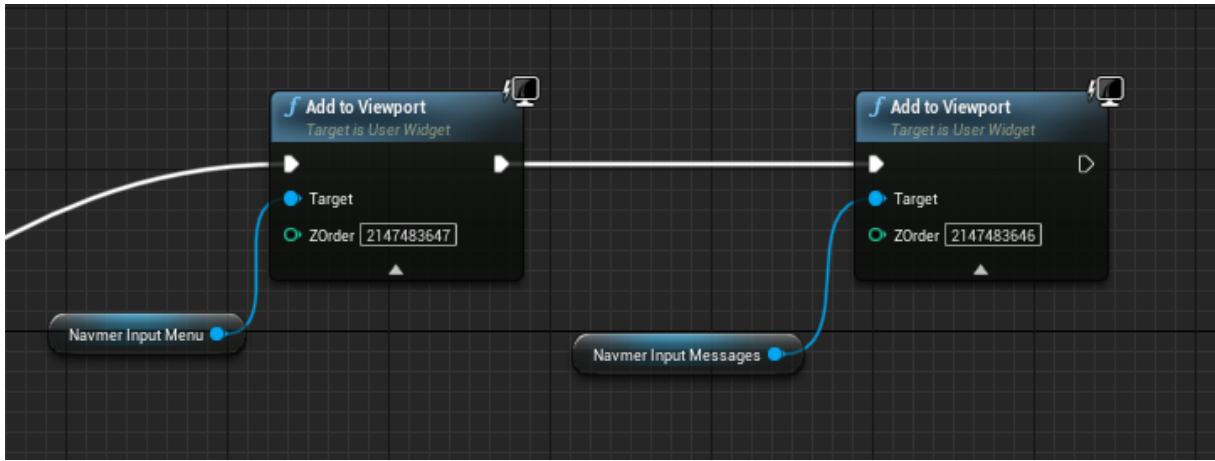
Associez à l'attribut **Target** le noeud **NavmerInputMenu**.

Définissez le **ZOrder** avec le nombre le plus élevé possible (cela implique que, par effet de priorité, le widget survolera tous les autres).

Créez à nouveau un noeud Add to Viewport avec comme Target le noeud **NavmerInputMessages** et comme ZOrder le ZOrder pour **NavmerInputMenu-1** (il sera affiché au dessus des autres widgets, et juste en dessous du menu).

Liez les deux noeuds Add to Viewport entre eux, et liez le premier noeud Add to Viewport avec le dernier noeud Set.

Maintenant, vous devriez avoir pour cette partie le résultat suivant :



Pour finir, le menu est caché par défaut.

Pour l'ouvrir en jeu, il serait intéressant de programmer une touche clavier pour l'afficher/le cacher à souhait.

En dessous du noeud Event Begin Play, créez un noeud **O** (pour la touche du clavier “o”) en recherchant dans le menu de création de noeuds “Event O”.

Ce noeud activera toute la branche d'exécution qui le suit lorsque la touche du clavier “o” sera pressée.

Créez un noeud **Branch** qui correspond à un simple “if-else” dans un langage de programmation.

Depuis le noeud NavmerInputMenu, créez le noeud **Get Menu Displayed**.

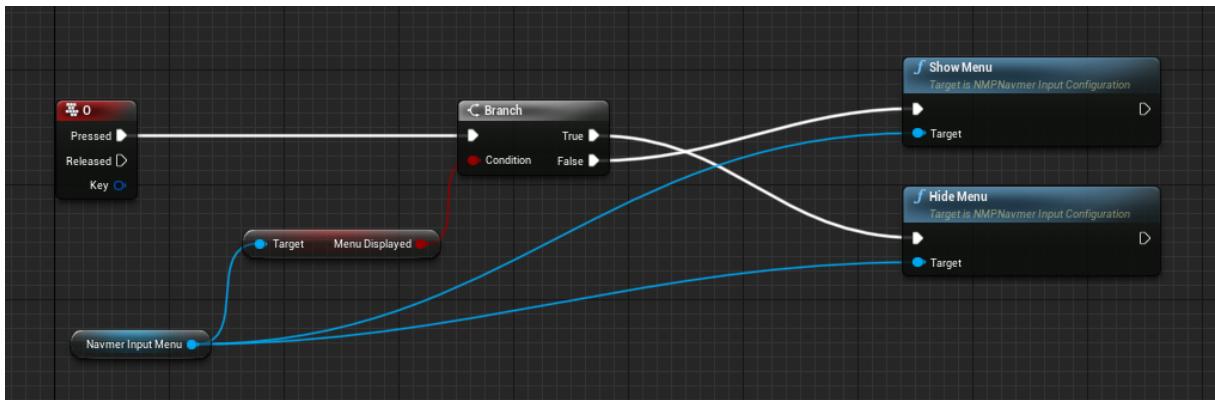
Associez le noeud **Menu Displayed** à la condition du noeud Branch.

Enfin, depuis le noeud NavmerInputMenu, créez les noeuds **Show Menu** et **Hide Menu**.

Ces deux noeuds permettent respectivement d'afficher et de cacher le menu.

Associez le **True** au noeud Hide Menu et **False** au noeud Show Menu.

Somme toute, vous devriez avoir pour cette partie le résultat suivant :



Vous en avez fini avec la mise en place du menu.

Vous pouvez alors essayer en jeu d'ouvrir/fermer le menu avec la touche “o” du clavier pour vérifier que tout fonctionne.



FIGURE 14 – Menu de configuration en jeu qui se place en haut à gauche de l'écran/fenêtre/viewport.

4.1.4 Utiliser le menu de configuration

Si vous comprenez le système de configurations d'appareils et de pages de configuration derrière le plugin NavmerInput (cf. la section **Le plugin NavmerInput**), l'utilisation du menu reste simple.

L'interaction avec le menu se fait par la souris.

Sur le menu figure deux onglets : un onglet **Association Contrôleur**, et un onglet au symbole “x” pour fermer le menu au moyen de la souris.

L'onglet Association Contrôleur (affiché par défaut) permet d'associer en jeu une configuration d'appareils existante avec une page de configuration existante (préalablement définies dans l'éditeur, dans la page de paramètres du plugin “JoystickInput”, ou dans le fichier DefaultInput.ini du projet).

Pour effectuer une nouvelle association :

1. Choisissez dans la listbox sous “Configuration du contrôleur” la configuration d'appareils associée à la console connectée.
2. Si vous venez de connecter la console, un message temporaire est affiché en haut à droite de votre écran/viewports/fenêtre vous indiquant le nom de l'appareil (et donc de la configuration d'appareils associée) avec la page de configuration associée actuellement.
3. Choisissez dans la listbox sous “Plage de configuration” la page de configuration adéquate ²².
4. Cliquez sur le bouton **Associer**.

À cet effet, vous devriez avoir un message qui apparaît vous indiquant le changement effectué.

Autrement l'état actuel d'une configuration d'appareils choisie dans la première listbox est indiqué sous le bouton Associer.

Pour désassocier une page de configuration sur une configuration d'appareils :

1. Choisissez dans la listbox sous “Configuration du contrôleur” la configuration d'appareils associée à la console connectée.
2. Cliquez sur le bouton **Dissocier**.

En plus d'un message de retour, vous devriez voir que l'état de la configuration d'appareils indique “DEVICE_NAME [Vide]”.

“Vide” signifie donc que la configuration d'appareils n'est associée à aucune page de configuration.

À noter que ces changements sont persistants pour toute la durée de l'application (soit, tant qu'elle n'est pas fermée).

22. Parmi les pages de configurations prédefinies qui sont : *Ship Console 1*, *Ship Console 2* et *Azimuth Console*.

4.2 Directives de maintenance

Cette partie détaille l'ensemble de l'implémentation du plugin.

Sauf si vous effectuez la maintenance sur le PC fixe du stage, il vous faut tout d'abord mettre en place l'environnement de développement.

NOTE

Pour le simulateur Navmer3D, il n'est pas nécessaire de cloner le plugin car il est déjà présent dans le projet. Il sera peut être simplement nécessaire de vérifier le suivi de la branche remote *stage2022* pour être sûr que la branche locale soit bien à jour.

Vous pouvez retrouver le repository du projet sur le Gitlab du CEREMA à ce lien : <https://gitlab.cerema.fr/navmer3d/navmerinput>.

La branche de développement du plugin est “*stage2022*” (c'est aussi la branche stable).

Le code est maintenu avec l'IDE **Visual Studio 2022**, configuré pour Unreal Engine.

Voici un lien vers la documentation pour procéder à la mise en place de l'outil : <https://docs.unrealengine.com/4.26/en-US/ProductionPipelines/DevelopmentSetup/VisualStudioSetup/>

Une fois votre Visual Studio installé correctement, le dossier du plugin doit être cloné ou placé dans le dossier **Plugins** situé à la racine de votre projet Unreal.

Pour modifier le code du plugin, vous devrez le faire depuis la solution VS de votre projet.

Puisque vous venez d'intégrer nouvellement les fichiers du plugin à votre arborescence, il vous faudra d'abord re-générer la solution comme suit :

- Faites un clique droit souris sur le fichier *MON_PROJET.uproject* localisé à la racine du projet
- Cliquez sur l'option **Generate Visual Studio project files**

Pour compiler le plugin avec Visual Studio : utilisez la configuration de solution “Development Editor” pour l'environnement “Win 64”.

Notez cependant que compiler un plugin nécessite de redémarrer Unreal après chaque compilation (à la différence du code source d'un projet qui peut être compilé “à chaud” grâce à la fonctionnalité **hot-reload** d'Unreal).

La modification des blueprints se fera dans l'éditeur du projet Unreal.

Les blueprints sont situés dans le dossier *Content/Blueprints/* qui se trouve à la racine du plugin.

4.2.1 Les périphériques HID

NOTE

Cette section permet uniquement d'introduire la terminologie **HID** (“*Human Interface Device*”) en vu d'apporter une connaissance supplémentaire étroitement liée aux tableaux de commandes Navmer.

Le terme *HID* désigne une norme et une classe de périphériques utilisés par l'humain pour contrôler des systèmes informatiques.

On y range donc l'ensemble des périphériques comme le clavier, les dispositifs de pointage (souris), les contrôleurs de jeu, et même les dispositifs de simulation (pédales, volants, etc.).

Les types d'architectures de communication pris en charge par la classe HID dépend de l'OS.

Dans le cas des tableaux de commandes Navmer qui sont des périphériques HID conformes, le type de transport de données utilisé est l'USB.

Le principe du HID repose sur la description du type de donnée transmise et non sur la description spécifique de celle-ci, permettant aux concepteurs d'employer leurs propres protocoles de transfert des données en vu de traiter n'importe quel appareil.

Elle fonctionne selon deux concepts : **les rapports** et **les descripteurs de rapports** (“*Report Descriptor*”).

Un rapport représente simplement un échange entre un appareil et une application (p.ex. une connexion/déconnexion, un état produit par l'interaction de l'utilisateur avec les entrées comme l'appui d'un bouton, etc.).

Le rapport est catégorisé selon 3 types, en partant du point de vue d'une application :

- Réception de données : “*Input*”
- Envoi de données : “*Output*”
- Réception et envoi de données : “*Feature*”

Exemple : Le mouvement d'une souris est représenté sur deux axes.

Le rapport envoyé lorsque la souris effectue un mouvement contient une donnée décrivant de combien la souris s'est déplacée sur les deux axes.

De même, lorsque le clic gauche de la souris est pressé, le rapport envoyé décrit quel est le bouton pressé et dans quel état il se trouve.

Le descripteur de rapport définit la métadonnée autour d'un rapport, i.e. le format d'un rapport et la signification des données échangées.

Le descripteur de rapport permet donc à l'application de reconnaître les différents champs de données utilisés dans un rapport.

Ces champs sont désignés comme des **objets** (“*Items*”).

```

1  0x05, 0x01,          // USAGE_PAGE (Generic Desktop)
2  0x09, 0x02,          // USAGE (Mouse)
3  0xa1, 0x01,          // COLLECTION (Application)
4  0x09, 0x01,          //   USAGE (Pointer)
5  0xa1, 0x00,          //   COLLECTION (Physical)
6  0x05, 0x09,          //     USAGE_PAGE (Button)
7  0x19, 0x01,          //     USAGE_MINIMUM (Button 1)
8  0x29, 0x03,          //     USAGE_MAXIMUM (Button 3)
9  0x15, 0x00,          //     LOGICAL_MINIMUM (0)
10 0x25, 0x01,          //     LOGICAL_MAXIMUM (1)
11 0x95, 0x03,          //     REPORT_COUNT (3)
12 0x75, 0x01,          //     REPORT_SIZE (1)
13 0x81, 0x02,          //     INPUT (Data,Var,Abs)
14 0x95, 0x01,          //     REPORT_COUNT (1)
15 0x75, 0x05,          //     REPORT_SIZE (5)
16 0x81, 0x03,          //     INPUT (Cnst,Var,Abs)
17 0x05, 0x01,          //     USAGE_PAGE (Generic Desktop)
18 0x09, 0x30,          //     USAGE (X)
19 0x09, 0x31,          //     USAGE (Y)
20 0x15, 0x81,          //     LOGICAL_MINIMUM (-127)
21 0x25, 0x7f,          //     LOGICAL_MAXIMUM (127)
22 0x75, 0x08,          //     REPORT_SIZE (8)
23 0x95, 0x02,          //     REPORT_COUNT (2)
24 0x81, 0x06,          //     INPUT (Data,Var,Rel)
25 0xc0,                //     END_COLLECTION
26 0xc0                //   END_COLLECTION

```

FIGURE 15 – Exemple de descripteur de rapport HID. Chaque champ est renseigné par des entiers non signés (en marge gauche). Il s’agit en l’occurrence ici d’un descripteur de rapport pour une souris.

Dans le lot d’objets présentés par le descripteur de rapport, **l’usage**²³ définit une information supplémentaire sur l’utilisation attendue de l’appareil et des divers entrées/contrôles (i.e. les boutons, les axes, etc.) par le concepteur.

Chaque appareil et chaque contrôle est assigné à un usage.

Tous les usages sont regroupés dans des **pages d’usages**²⁴.

Les pages d’usages permettent d’identifier par catégorie les usages selon les utilisations primaires attendues des contrôles (p.ex. des contrôles destinés à une utilisation sur ordinateur, des contrôles destinés à la simulation, des LEDs, des instruments médicaux, etc.).

23. Usages HID : <https://learn.microsoft.com/fr-fr/windows-hardware/drivers/hid/hid-usages>

24. Table des pages d’usages USB : https://www.usb.org/sites/default/files/documents/hut1_12v2.pdf

Exemple : Dans le cas des tableaux de commandes Navmer, si on se réfère à la table des usages USB : se sont des contrôleurs de jeux, qui sont donc destinés à être utilisés sur ordinateur.

Conclusion : La page ciblée est la page d'ID 0x01 (“Generic Desktop Controls”); les tableaux de commandes sont probablement assignés aux usages d'ID 0x04 (“Joystick”) ou 0x05 (“Game Pad”).

La classe HID, via un parser, permet également une **organisation des descripteurs**²⁵ en regroupant plusieurs rapports qui ont un lien.

Enfin, il existe différentes APIs²⁶ selon l'OS permettant aux concepteurs d'appareils et développeurs de logiciels de créer ou traiter des paquets HID.

Un driver générique proposé par l'OS intervient alors dans le formatage des données du paquet reçu par une application cible.

NOTE

En dehors du descripteur de rapport HID, il existe selon l'architecture de communication employée d'autres descripteurs importants permettant d'apporter de l'information sur un appareil.

Par exemple pour l'architecture USB, **le descripteur de périphérique** (“Device Descriptor”) donne des renseignements de manufacture comme l'ID du vendeur ou l'ID du produit.

25. Les *Collections HID* : <https://learn.microsoft.com/fr-fr/windows-hardware/drivers/hid/hid-collections>

26. Documentation de l'API HID de Windows: <https://learn.microsoft.com/fr-fr/windows-hardware/drivers/hid/introduction-to-hid-concepts>

4.2.2 Code : Général

Présentement, le plugin s'appelle “NavmerInput” mais l'outil de réflexion réfère le plugin/module avec le nom “JoystickPlugin”.

Ceci est tout à fait normal puisque le code de NavmerInput s'appuie sur la base du plugin **JoystickPlugin** écrit par JaydenMaalouf (cf. **JoystickPlugin**).

Il est sûrement possible de changer cela, au même titre qu'il est possible de changer le nom d'un projet Unreal ou le nom d'une classe.

Cependant ce type d'opération n'est pas trivial à cause des outils de **construction de projet**²⁷ et **d'analyse du code d'Unreal**²⁸, et des potentielles multiples références au plugin dans le cas où le plugin est déjà inclus dans un projet.

Le code se situe dans le dossier *Source/JoystickPlugin* à la racine du plugin.

Il est scindé en deux sous-dossiers : **Private** et **Public**.

Le dossier Public contient l'interface public du plugin (i.e. les fichiers entêtes .h dont les structures de données), tandis que le dossier Private contient le code source.

Le code essaie de respecter une convention lexicale sur certains points, notamment pour être en accord avec les best-practices d'Unreal :

- L'écriture des noms de variables, de fonctions et de classe suit la méthode “CamelCase”
- Les noms de variables, de fonctions et de classes sont majoritairement écrits sans abréviations pour des questions de compréhension, et entièrement en anglais
- Les classes propres à Navmer3D ont toutes le préfixe “NM” pour les classes du projet, et “NMP” combiné au nom du plugin pour les classes plugins
- Les préfixes Unreal sont utilisés (“A”, “U”, “I”, “F”, etc.)²⁹

27. *Unreal Build Tool* : <https://docs.unrealengine.com/4.27/en-US/ProductionPipelines/BuildTools/UnrealBuildTool/>

28. *Unreal Header Tool* : <https://docs.unrealengine.com/4.27/en-US/ProductionPipelines/BuildTools/UnrealHeaderTool/>

29. Convention de nommage d'Unreal Engine : <https://docs.unrealengine.com/4.26/en-US/ProductionPipelines/DevelopmentSetup/CodingStandard/>

 **NOTE****Légende des diagrammes :**

- Les flèches creuses indiquent un héritage entre les classes (avec en tête de flèche la classe parente)
- Les flèches creuses en traitillé indiquent les implémentations d'interfaces (avec en tête de flèche l'interface)
- Les liens en losange creux indiquent une communication par aggrégation
- Les membres sont étiquetés par un carré, un cercle et un losange, signifiant respectivement que les membres sont soit privés, publics ou protégés
- Les fonctions membres soulignées sont des fonctions statiques

Toutes les fonctions du plugin ne seront évidemment pas expliquées dans ce document.

Il y sera principalement détaillé celles qui le nécessite, dans un but de compréhension global.

C'est donc à la charge du développeur de s'intéresser au contenu des autres.

4.2.3 Code : Structures de données principales



FIGURE 16 – Les principales structures de données du plugin.

Les principales structures de données du plugin sont situées dans le dossier *Public/Data*.

La structure **FAxisData** englobe les données d'un axe.

L'attribut **Value** correspond à la valeur de l'axe.

Par défaut elle est désignée dans une plage de valeur [0, 1].

Cependant, suivant les disfonctionnements sur l'axe, il se peut que la plage réelle soit différente.

Les attributs **InputRangeMin**, **InputRangeMax** correspondent donc aux bornes de la plage réelle de l'axe, avec **InputRangeCenter** la valeur de centre réelle.

Ce sont ces attributs qui vont permettre d'établir un remapping précis des valeurs de l'axe (i.e. la réassignation de la plage de valeurs de l'axe suivant la plage de valeur réelle et le centre réel).

L'attribut **bGamepadStick** permet de préciser si, après le remapping, l'axe doit prendre des valeurs négatives (i.e. des valeurs dans la plage $[-1, 1]$, ou dans la plage $[0, 1]$ par défaut).

L'attribut **Key** correspond à la clé d'action associée à l'axe.

En plus de la structure FAxisData, il existe d'autres structures pour les boutons, les axes de tête, etc.

La structure **FDeviceInfoSDL** regroupe les données d'un “**device SDL**”.

Dans un objectif d'encapsulation : un “device SDL”, à la différence d'un “**device plugin**” (ou simplement “**device**”), s'inscrit dans une couche inférieure de traitement des contrôleurs du plugin, i.e. là où les échanges se font directement avec la SDL.

Les traitements comme le remapping ou la configuration via l'éditeur ou le menu intégré se font dans une couche supérieure du plugin en affectant les “devices plugin”.

NOTE

Il faut bien différencier les termes “device SDL” et “device plugin”.

Par la suite, il est possible que le terme “device plugin” soit abrégé en “device”, là où le terme “device SDL” restera comme tel afin d'éviter toute ambiguïté.

On notera les attributs **DeviceIndex**, **DeviceId** et **InstanceId**.

L'attribut DeviceIndex correspond à l'indice du device SDL dans l'ensemble des appareils connectés au système. Il est défini par une accession depuis une structure SDL et permet par un appel à la SDL d'ouvrir un échange avec un appareil connecté.

L'attribut InstanceId correspond à un identifiant de série de device SDL, incrémenté à chaque fois qu'il y a une connexion. Son accession se fait par un appel à la SDL.

L'attribut DeviceId correspond à un identifiant de device, en suivant l'InstanceId d'un device SDL. Il permet d'identifier un device sur un ensemble de devices enregistrés depuis l'exécution du programme. Son existence prend plus de sens dans la couche supérieure du plugin. Dans la structure FDeviceInfoSDL, il permet de retrouver un device suivant le device SDL correspondant.

La structure **FJoystickInfo** regroupe les données d'un device. On y retrouve le Device Name, l'identifiant produit du device ou encore l'indice de la page de configuration affectée. On dispose également du DeviceId pour le device instancié.

Enfin, la structure **FJoystickDeviceData** centralise les données de toutes les entrées d'un device.

4.2.4 Code : Classes principales et fonctionnalités

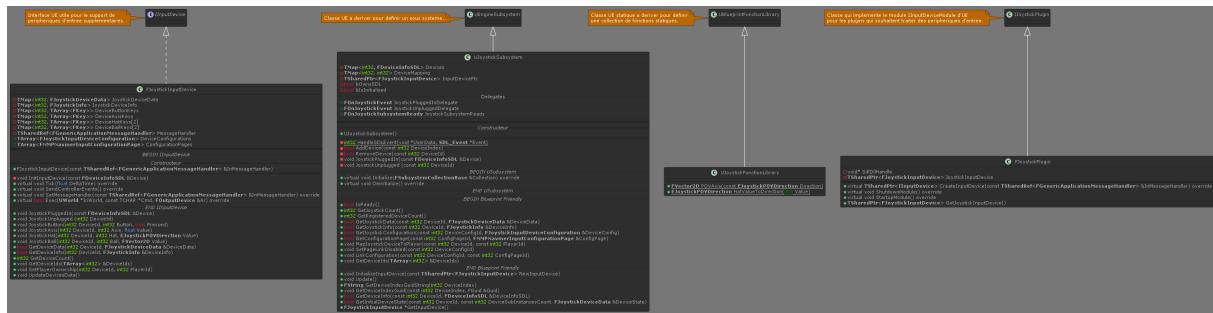


FIGURE 17 – Les classes principales et fonctionnalités du plugin.

Les classes principales du plugin, **UJoystickSubsystem** et **FJoystickInputDevice**, se trouvent directement sous le dossier **Public**.

Dans l'objectif d'effectuer des traitements sur les contrôleurs avec Unreal, le plugin dispose d'une interface et encapsule la plupart des données échangées et fonctionnalités de la SDL. Dans un premier temps, l'utilisateur connecte un appareil au système.

L'appareil reconnu commence alors à échanger des données avec le système (au travers du bus USB dans le cas des tableaux de commandes) lorsque l'utilisateur interagit avec l'appareil.

Le plugin fait alors appel dans sa couche d'abstraction inférieure aux fonctionnalités de la SDL (précisément, aux fonctionnalités d'écoute des APIs sous-jacentes) pour réceptionner les données d'entrée échangées.

Le plugin remonte enfin dans sa couche d'abstraction supérieure l'information en vue d'effectuer divers traitements.

La reconnaissance des appareils se fait au travers de la page de paramètres du plugin dans l'éditeur d'Unreal, où sont indiqués les identifiants matériels de l'appareil sollicité.

4.2.5 Code : La classe UJoystickSubsystem

La classe **UJoystickSubsystem** contient les données échangées avec la SDL et permet de gérer l'ensemble des devices SDL : elle correspond à la couche inférieure.

Dans Unreal Engine, un sous-système³⁰ fonctionne comme **un singleton** qui est automatiquement instancié par Unreal et qui a une durée de vie définie.

Dans le cas de UJoystickSubsystem qui hérite de la classe **UEngineSubsystem**, sa durée de vie est maintenue jusqu'à la fin de l'exécution de l'éditeur ou du jeu.

Les méthodes **Initialize()** et **Deinitialize()** permettent à Unreal de respectivement initialiser et détruire le singleton, avant que celui-ci ne passe par la case "garbage-collector" si sa référence n'est plus utilisée alors.

Le principal avantage du sous-système est qu'il est accessible de manière globale après son initialisation.

Il est accessible depuis le code, et depuis un blueprint.

De ce fait, il est intéressant dans le cas du plugin de le considérer comme un manager de devices.

Tout d'abord, lorsque le singleton est initialisé via Initialize(), le sous-système SDL utilisé pour l'écoute et la gestion de périphériques (*SDL_INIT_JOYSTICK*) l'est également.

Le poll des évènements SDL se fait via la fonction **Update()**.

La méthode **HandleSDLEvent()**, via la réception d'un évènement SDL de type "JOYSTICK" (soit un contrôleur de jeu), va faire appel aux différentes méthodes de la classe pour traiter l'ajout d'un device SDL, l'appui d'un bouton ou encore le mouvement d'un axe.

Deux méthodes appellées par HandleSDLEvent() sont à noter :

- La première, **AddDevice()**, permet d'initialiser un device SDL (couche inférieure) lorsqu'une connexion d'appareil s'effectue, puis d'enregistrer un device (couche supérieure) par un appel de la fonction **JoystickPluggedIn()**. Le contenu de la fonction JoystickPluggedIn() sera expliqué un peu plus bas dans la sous-section **Code : La classe FJoystickInputDevice**.
- La deuxième, **JoystickAxis()**, permet lors d'un évènement *SDL_JOYSTICKMOTION* (i.e. lorsqu'un mouvement d'axe est capté) de traiter le mouvement d'un axe. Cette méthode appartient à la classe **FJoystickInputDevice** et sera détaillée plus bas dans la sous-section **Code : La classe FJoystickInputDevice**.

La méthode **GetInitialDeviceState()** sert à initialiser le nombre d'entrées d'un device SDL pour un DeviceId donné.

Les devices ont un nombre fixé à 8 axes de base.

30. Article sur l'utilisation du sous-système d'Unreal Engine : <https://benui.ca/unreal/subsystem-singleton/>

Selon s'il existe plusieurs instances du même device (nombre défini dans l'argument **DeviceSubInstancesCount**), le nombre d'axe peut varier (i.e. 16 axes s'il y a 2 instances, 24 s'il y a 3 instances, etc.). Le lecteur est incité à revoir la partie sur **les conflits de clés d'axes** dans la sous-section **Configurer les tableaux de commandes** pour comprendre ce choix.

Pour rappel, le nombre d'instances d'un appareil connecté peut être configuré via la page de paramètres du plugin dans l'éditeur en modifiant le champ **Number Of Instances**.

De même qu'une page de configuration peut être dédiée à une instance précise en donnant l'indice de l'instance dans le champ **DeviceInstance** d'une page de configuration dans la page de paramètres du plugin.

Pour conclure sur la classe UJoystickSubsystem, le singleton possède un pointeur vers un objet **FJoystickInputDevice**, ce qui lui permet de communiquer avec la couche supérieure.

D'autres fonctions sont à noter, comme **LinkConfiguration()** ou encore **SetPageLinkDisabled()** qui permettent de traiter les changements faits sur les devices via le widget du menu de configuration (détails plus bas dans la sous-section **Code : Widgets UMG**).

4.2.6 Code : La classe FJoystickInputDevice

La classe **FJoystickInputDevice** correspond à la couche supérieure de traitement des contrôleurs de jeu génériques manipulés par le plugin.

Elle permet de faire remonter les évènements contrôleurs de la SDL à Unreal Engine en communiquant avec le singleton UJoystickSubsystem.

On notera tout d'abord le constructeur qui effectue une copie des configurations de devices et des pages de configurations présentent dans la page de paramètres du plugin, de manière à ne pas modifier les données directement dans la page de paramètres lorsqu'on se trouve en simulation / en jeu.

La méthode **SendControllerEvents()** permet d'envoyer au système de traitement des entrées contrôleurs d'Unreal Engine les données d'entrées parsées dans la couche inférieure, via l'objet **Message-Handler** et en associant une clé d'action (type *FKey*) à chaque entrée.

SendControllerEvents() appelle également en fin de corps la méthode Update() du singleton UJoystick-Subsystem.

La méthode **InitInputDevice()** permet d'initialiser un device plugin tout d'abord en lui donnant un état initial au travers de la méthode **GetInitialDeviceState()** du singleton UJoystickSubsystem.

Elle initialise par la suite les clés d'actions qui seront associées aux entrées du device dans la méthode SendControllerEvents().

Enfin, elle appelle la méthode **UpdateDevicesData()** pour initialiser les données des entrées.

La méthode UpdateDevicesData() est appelée dans plusieurs fonctions du plugin pour mettre à jour les données d'entrées lorsqu'un changement est effectué, notamment via le widget menu ou la page de paramètres.

On en revient maintenant à la méthode **JoystickPluggedIn()**, appelée par la méthode AddDevice() de la classe UJoystickSubsystem, qui fait justement un appel à la méthode InitInputDevice() lorsqu'un device est enregistré.

Quant à la méthode **JoystickAxis()**, appelée par la méthode HandleSDLEvent() appartenant à la classe UJoystickSubsystem, elle permet pour un device donné de définir la valeur de l'axe parsée par la SDL.

L'emplacement de la valeur est corrigé selon le nombre d'instances de l'appareil.

Chaque type d'entrée possède une fonction équivalente.

Dans le cas des axes, c'est la méthode **GetValue()** de la structure FAxisData qui s'occupe de corriger les valeurs des axes pour être conforme aux paramètres de l'utilisateur.

Enfin, un seul objet FJoystickInputDevice est instancié.

Le pointeur associé est attaché au singleton UJoystickSubsystem (attribut) lors de l'initialisation du plugin, via InitialiseInputDevice() qui est appelée par **CreateInputDevice()** (classe FJoystickPlugin).

4.2.7 Code : Echange entre les classes principales

La classe **FJoystickPlugin** sert de point d'entrée du module pour Unreal. Elle hérite de la classe **IJoystickPlugin** qui implémente l'interface **IInputDeviceModule**.

Cette interface proposée par Unreal offre la possibilité de traiter avec des périphériques d'entrée, voire d'ajouter de nouveaux périphériques en passant une référence vers un objet **FGenericApplication-MessageHandler** pour permettre une programmation évènementielle.

La méthode **CreateInputDevice()** instancie un objet **FJoystickInputDevice** et fait appel à la méthode **InitialiseInputDevice()** de la classe UJoystickSubsystem (le singleton du sous-système a été initialisé par Unreal au préalable).

La fonction InitialiseInputDevice() permet :

- De rattacher la couche inférieure avec la supérieure via le passage d'un pointeur sur un objet FJoystickInputDevice
- D'initialiser toutes les devices SDL actuellement connectées
- De passer une référence à la fonction HandleSDLEvent() au sous-système de la SDL afin de traiter un évènement contrôleur lors d'un poll.

4.2.8 Code : La classe UJoystickFunctionLibrary

La classe **UJoystickFunctionLibrary** est une classe statique. Elle ne contient que des fonctions de type “helper”.

L'intérêt d'une telle classe est de pouvoir faire appel n'importe où à des fonctionnalités utilitaires, dans le code comme dans un Blueprint. Ceci-dit, elle ne dispose pas d'un grand intérêt pour le moment : l'on va préférer se servir du sous système UJoystickSubsystem qui est également accessible partout.

4.2.9 Code : Paramètres du plugin

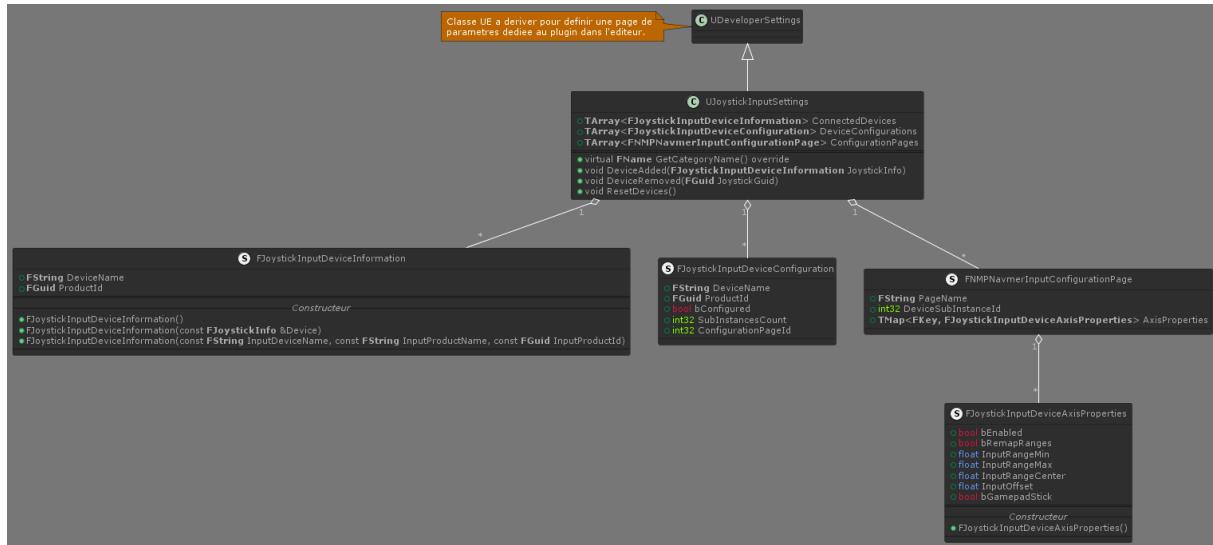


FIGURE 18 – Les structures de données relatives aux paramètres du plugin et la classe **UJoystickInputSettings**.

Les structures de données relatives aux paramètres du plugin se trouvent dans le dossier *Public/Settings*.

La classe **UJoystickInputSettings** se trouve directement sous le dossier **Public**.

La structure **FJoystickInputDeviceConfiguration** regroupe les données d'une configuration de devices, et la structure **FNMPNavmerInputConfigurationPage** regroupe les données d'une page de configuration.

Le contenu de la classe **UJoystickInputSettings** est reflété dans l'éditeur par l'ajout d'une section dans les paramètres du projet portant le nom du plugin (i.e. "JoystickPlugin").

Elle hérite de la classe de base **UDeveloperSettings** qui permet d'étendre l'éditeur en ajoutant à la fenêtre des paramètres de projet une page de configuration dédiée à un module.

On peut y retrouver les rubriques **ConnectedDevices**, **DeviceConfigurations** et **ConfigurationPages** (cf. partie **Configurer les tableaux de commandes**), qui sont en fait des listes de structures automatiquement interprétées et affichées dans l'éditeur.

La classe précise via un décorateur que ses variables peuvent être lues-depuis/écris-dans un fichier externe³¹.

31. Les fichiers de configuration d'Unreal : <https://docs.unrealengine.com/4.27/en-US/ProductionPipelines/ConfigurationFiles/>

Les paramètres du plugin sont donc sauvegardés dans le fichier de configuration **DefaultInput.ini** situé dans le dossier **Config** à la racine du projet (cf. le fichier **DefaultInput.ini**, partie **Installer le plugin**).

Elle a ses propres fonctions membres lui permettant d'ajouter ou supprimer de la page de paramètre du plugin les appareils qui respectivement se connectent et se déconnectent (cf. rubrique **Connected Devices**, partie **Configurer les tableaux de commande**).

Tout changement temps-réel depuis l'éditeur sur la page de paramètre du plugin est traité dans cette classe, au travers de la fonction membre héritée **PostEditChangeChainProperty**.

Enfin, elle permet via l'éditeur (ou le fichier de configuration DefaultInput.ini) la manipulation de l'ensemble des configuration de devices et de l'ensemble des pages de configurations.

4.2.10 Code : Widgets UMG

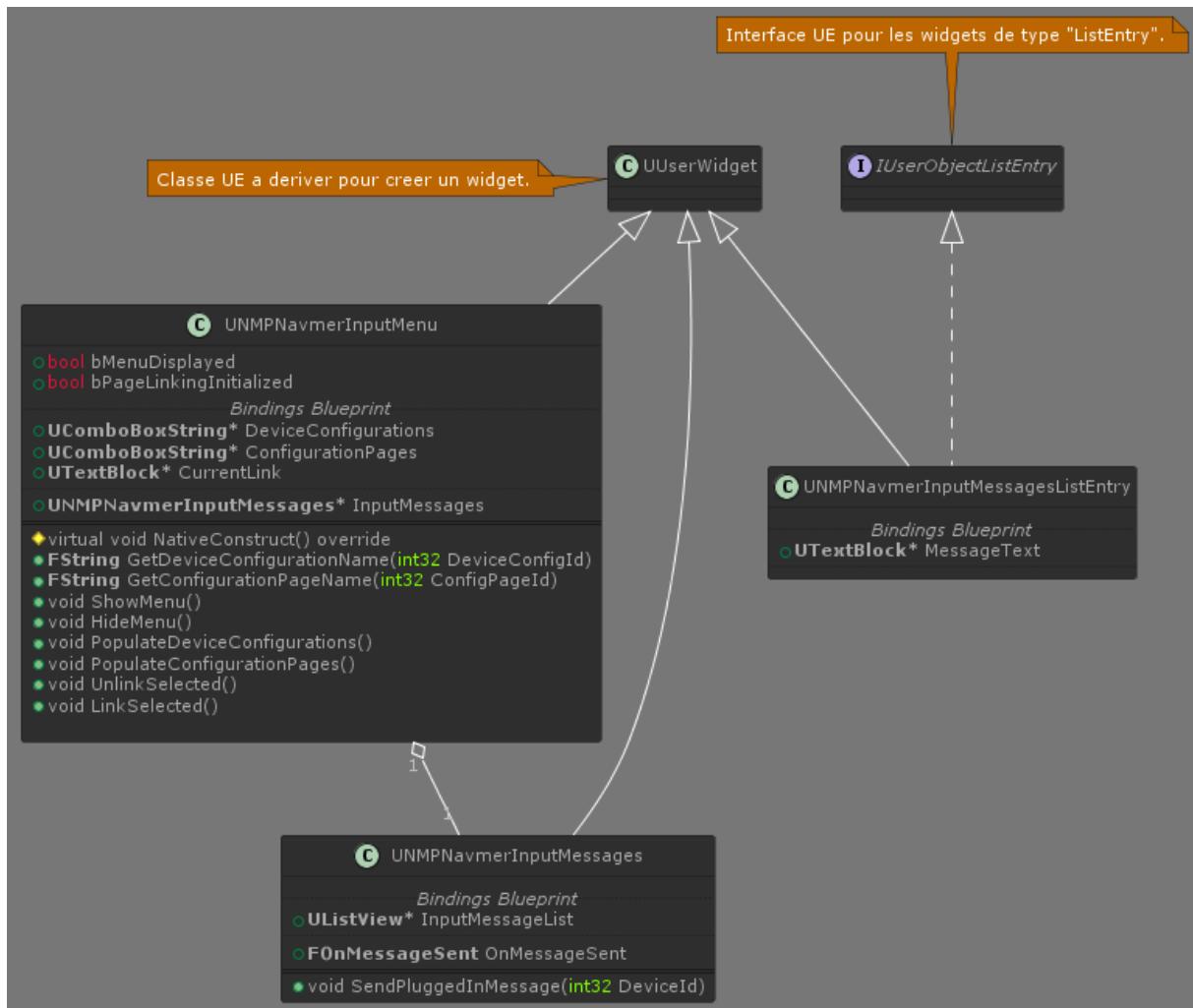


FIGURE 19 – Les classes des widgets du plugin.

Les classes des widgets **menu** et **messages** se situent dans le dossier **Widgets**.

Un objet de type **Widget** est un élément visuel du designer **UMG**.

Un widget complexe est créé à partir d'autres widgets (p.ex. un menu utilise des widgets boutons, listes, texte, etc.).

L'outil UMG permet au travers d'un blueprint particulier (un "*Widget Blueprint*") la création d'un **HUD**, d'un menu, et plus généralement de toute interface graphique. Il offre la possibilité, à partir d'un widget blueprint, de **binder** les widgets à une variable dans une classe pour y définir la logique ou effectuer des modifications sur les widgets (p.ex. remplir une combobox, modifier un textblock, changer le style d'un widget, etc.).

Un binding UMG s'effectue de la manière suivante :

- Dans un widget blueprint, il faut cocher la case **Is Variable** dans le panel **Details** d'un widget duquel l'on souhaite définir la logique dans une classe
- Dans la classe qui est dérivée par le widget blueprint, il faut déclarer comme membre un pointeur sur une variable du type du widget à binder
- Il faut également que l'identifiant de la variable et du widget dans le blueprint soient exactement identiques.
- Enfin, il faut insérer au dessus de la déclaration de la variable une propriété Unreal en précisant la métapropriété **BindWidget**

Exemple : En prenant le menu en exemple :

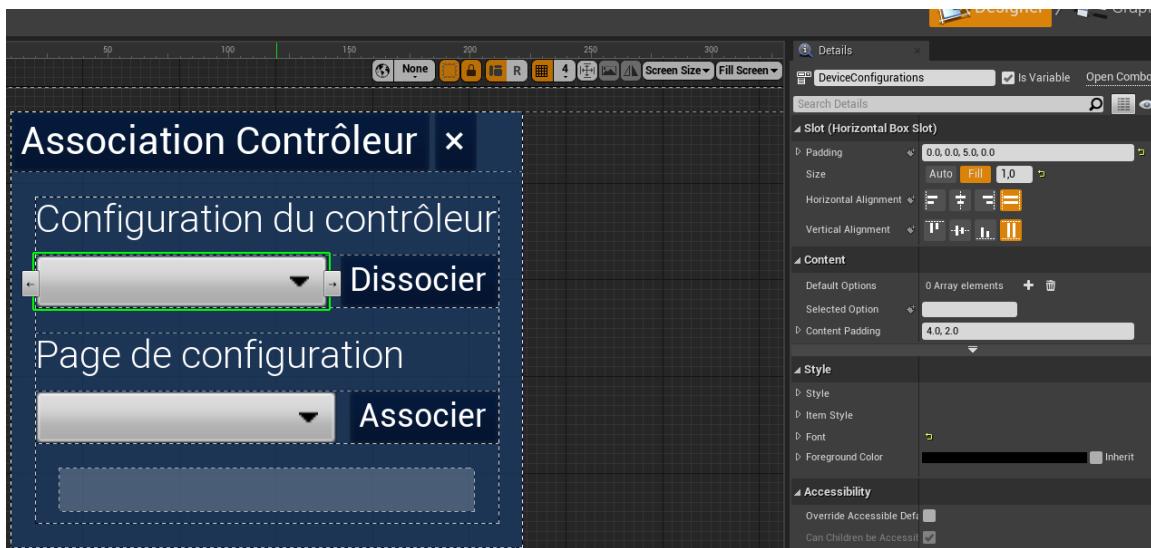


FIGURE 20 – Dans le blueprint **WBP_NavmerInputMenu**, on peut voir que pour la combobox sélectionnée “DeviceConfigurations” (encadrée en vert), la case **Is Variable** est cochée dans le panel **Details**, présent par défaut sur la droite lorsque l’on se trouve dans l’onglet **Designer** de l’édition de blueprint.

```
/**  
 * @brief Combobox qui regroupe les configurations de devices existantes.  
 */  
UPROPERTY(BlueprintReadOnly, Category="Inputs Configuration | Page Linking", meta=(BindWidget))  
class UComboBoxString *DeviceConfigurations;  
...
```

FIGURE 21 – Dans la classe **NMPNavmerInputMenu**, on précise au dessus de la variable “DeviceConfigurations” un UPROPERTY avec la métapropriété **BindWidget**.

Les différents comportements du widget **menu** sont définis par la classe **UNMPNamverInputMenu**.

Le menu est un widget, et comporte un ensemble de sous-widgets prédéfinis dans Unreal.

Les membres **DeviceConfigurations**, **ConfigurationPages** et **CurentLink** sont des bindings sur les sous-widgets du menu.

DeviceConfigurations correspond à la combobox supérieure de l'onglet **Association Contrôleur**, et ConfigurationPages à la combobox inférieure.

CurentLink correspond au textblock en bas de l'onglet qui indique le statut d'association actuel.

La variable **InputMessages** est une référence vers un objet de type **UNMPNavmerInputMessages**. Elle permet via une communication de type **DAC**³² avec le widget **messages** l'affichage des résultats d'interactions utilisateur avec le menu.

La fonction membre **NativeConstruct()** fonctionne comme un constructeur pour les widgets, et doit être utilisée à cet effet. Elle est héritée de la classe **UUserWidget**.

Les fonctions **ShowMenu()** et **HideMenu()** sont assez explicites de par leur nom. Cependant leur existence est redondante avec certaines fonctions d'une autre classe.

La cause de cette redondance et les autres fonctions seront détaillées plus loin, avec la classe **ANM-HUD**.

Le widget **messages** est défini par la classe **UNMPNavmerInputMessages**. Le comportement du widget est défini en quasi-totalité dans le blueprint.

Le widget comporte un sous-widget du type **ListView**. Elle est bindée à une variable dans la classe UNMPNavmerInputMessages, cependant le binding n'est actuellement pas utilisé.

Une listview définie simplement une liste de widgets. Ces widgets sont affichés verticalement dans un ordre FIFO.

Le membre **OnMessageSent** est une **delegate** de type **multicast dynamique**.

Les *Delegates*³³ sont un type de communication par évènement inter-classes (à la différence de la communication DAC qui est une communication par référence directe) qui fonctionne sur la base de l'envoi/la réception de messages. Un message envoyé par un appel à une delegate est traité par une **fonction handler** qui est bindé à ladite delegate.

Une delegate dite "multicast dynamique" implique que l'on peut binder plusieurs handlers à cette delegate si l'on souhaite effectuer différens traitements, et qu'elle est implantable dans un blueprint.

Un envoi au travers de cette delegate revient à "broadcaster" le message sur toutes les handlers.

32. Direct Actor Communication : <https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/ActorCommunication/DirectCommunicationQuickStart/>

33. Les Delegates Unreal : <https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/ProgrammingWithCPP/UnrealArchitecture/Delegates/>

La fonction **SendPluggedInMessage** s'execute dans le blueprint lorsqu'un device vient de se connecter. Cette fonction envoie un message contenant le nom du device qui apparaît.

OnMessageSent étant bindé dans le blueprint de la classe, le message est alors traité dans le blueprint.

Pour conclure sur les classes widgets, le widget **entrée de messages** est défini par la classe **UNMP-NavmerInputMessagesListEntry**.

Cette classe n'a actuellement pas réellement d'intérêt si ce n'est que pour l'évolutivité du plugin, dans le cas où une implémentation est nécessaire.

Le comportement de ce widget est défini dans son blueprint.

Les classes widget héritent toutes de la même classe de base : **UUserWidget**. Elles fonctionnent également de paire avec les Blueprints qui les dérives, au travers des bindings widgets.

La mise en place des widgets s'effectue comme indiqué dans la partie **Installer le menu de configuration**.

Concernant le simulateur Navmer3D, c'est différent.

Dans le cas du simulateur Navmer3D, la mise en place des widgets s'effectue au travers du HUD.

Le HUD se comporte comme un manager de l'ensemble des widgets qui sont affichés en jeu. Il est associé à un **Player Controller**.

L'accès d'un HUD se fait donc à partir d'un Player Controller.

Le HUD de Navmer3D est défini par la classe **ANMHUD** qui hérite de la classe de base **AHUD**.

Cette classe se trouve dans le dossier *Source/Navmer3D/HUD* en partant de la racine du projet.

NOTE

Dans ce document, la classe ANMHUD de Navmer3D ne sera pas détaillée dans son ensemble.

Pour le widget menu comme pour le widget messages, chacun a deux fonctions dédiées dans ANMHUD : **ShowNavmerInputMenu()/ShowNavmerInputMessages()** et **HideNavmerInputMenu()/HideNavmerInputMessages()**.

Ces fonctions permettent respectivement d'instancier et attacher les widgets au viewport in-game, et de détacher et détruire les widgets.

Elles portent finalement les deux sens d'"afficher" et de "cacher", ce qui rappelle les fonctions ShowMenu() et HideMenu() du widget menu, remettant alors en question leur existence.

Alors que tous les autres widgets du projet sont affichés et cachés uniquement par des fonctions dédiées présentes dans la classe ANMHUD, l'existence des fonctions ShowMenu() et HideMenu() s'explique par un **problème d'initialisation du contenu** des widgets combobox.

Dans une simulation en mode “standalone”, lors de la phase d’instanciation du widget menu : il ne semble pas possible d’interagir avec le module NavmerInput autrement qu’avec le sous système UJoystickSubsystem.

Pour remplir le contenu des combobox, les fonctions **PopulateDeviceConfigurations()** et **PopulateConfigurationPages()** font un fetch de l’ensemble des configurations de devices et des pages de configuration existantes depuis la référence **InputDevice** de l’instance UJoystickSubsystem.

Cette référence, initialisé au préalable lorsque le plugin est chargé (chargement qui normalement devrait s’opérer dans le processus de chargement d’Unreal bien avant celui des widgets), n’est pas accessible à la construction du widget (et donc inaccessible dans les fonctions NativeConstruct(), PreNativeConstruct() ou Initialize() du widget menu).

De même, les données de l’instance de UJoystickInputSettings (page de paramètres du plugin) ne sont pas accessibles.

Dès lors, le seul moyen trouvé pour populer les combobox est de cacher à sa création (i.e. via ShowNavmerInputMenu()) le menu avec la fonction HideMenu().

C’est ensuite par l’appel à la fonction ShowMenu() que le contenu est initialisé **une seule fois** en contrôlant un état d’initialisation via une variable booléenne.

Ce problème n’est présent que dans le cas où la simulation s’effectue en “standalone”.

Si la simulation s’effectue dans le viewport de l’éditeur ou dans une nouvelle fenêtre de l’éditeur (PIE), la problématique ne se pose pas.

Ce choix d’implémentation se justifie par le fait que l’exécution du simulateur en mode “standalone” fonctionne sur le même principe que sur une version “packagé” (i.e. un exécutable livrable et des ressources système dédiées).

NOTE

Note de l’auteur : “Malheureusement, le manque d’expérience sur UE ne m’a pas permis de comprendre entièrement la problématique et de trouver une solution correcte.

Du coup c’est assez compliqué à expliquer. En espérant que le gros pavé sur la problématique au dessus était un maximum clair.”

L’appel des fonctions **ShowNavmerInputMenu()/ShowNavmerInputMessages()** et **HideNavmerInputMenu()/HideNavmerInputMessages()** se fait dans le blueprint d’un niveau, par l’accession du HUD (cf. exemple avec le niveau *Lyon* dans [Blueprints : Les widgets dans l’HUD](#)).

Enfin, de manière à appeler les bonnes fonctions avec les bonnes touches, il est nécessaire d’associer les clés d’entrées (FKey) à des actions.

Pour cela, il faut se rendre dans la rubrique **Inputs**³⁴ des paramètres du projet Unreal, via *Edit > Project*

34. L’objet Input d’Unreal Engine : <https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/Input/>

Settings dans la barre principale de l'éditeur.

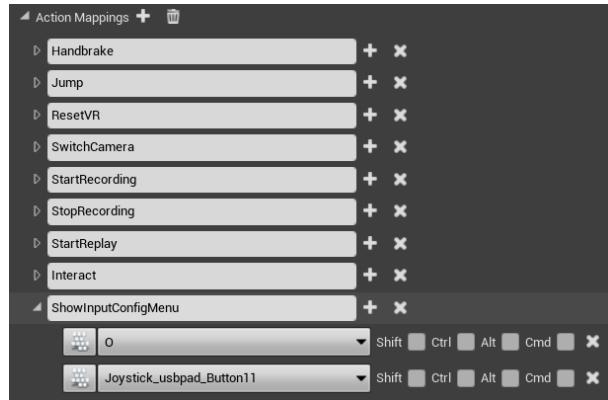


FIGURE 22 – Action Mappings : Association d'un bouton de la Ship Console pour ouvrir le menu via l'action **ShowInputConfigMenu**.

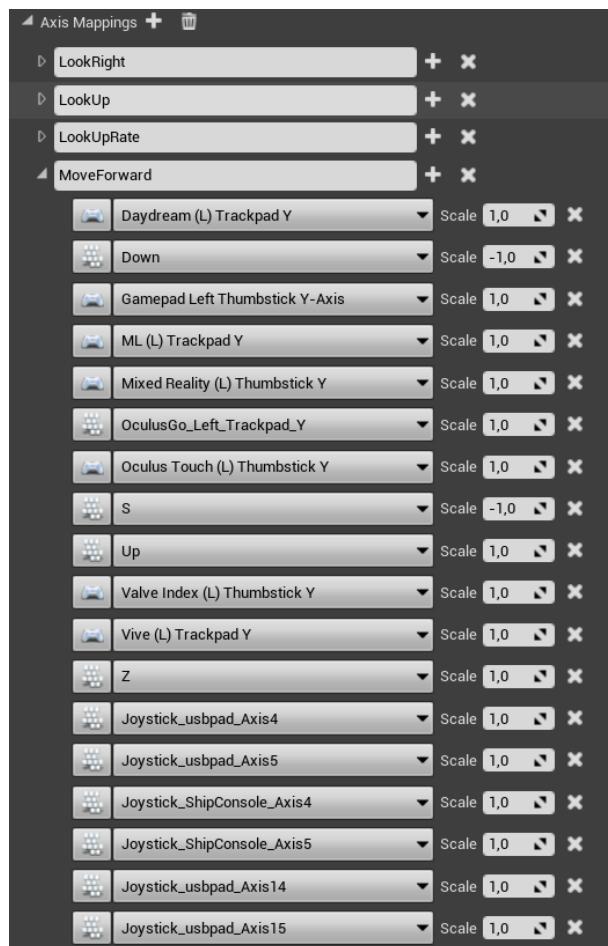


FIGURE 23 – Axes Mappings : Association de plusieurs axes de différentes Ship Consoles pour faire avancer le navire via l'action **MoveForward**.

Chaque entrée de chaque contrôleur peut être associée grâce à leur clé à des actions, en donnant à ces actions un nom dans la rubrique ***Inputs**.

Plusieurs entrées peuvent être associées à une action.

Les entrées traitées par le plugin NavmerInput sont nommées comme tel : *Joystick_NOM-DU-DEVICE_NOM-DE-L-ENTREE*.

Les actions sont programmables, au choix, dans un blueprint ou bien dans le code.

4.2.11 Blueprints : Général

Comme pour le code, l'appellation des blueprints respecte une certaine convention :

- Les simples blueprints sont préfixés par “BP_”
- Les widget blueprints sont préfixés par “WBP_”
- Le nom des blueprints suit la méthode *CamelCase*
- De manière générale, tous les assets d’Unreal portent un préfixe et sont nommés selon la méthode *CamelCase*

Les blueprints liés aux classes widgets se trouvent dans le dossier *Content/Blueprints*.

Il y a au total 3 widget blueprints :

- **WBP_NavmerInputMenu** pour le widget **menu**
- **WBP_NavmerInputMessages** pour le widget **messages**
- Et **WBP_NavmerInputMessagesListEntry** pour le widget **entrée de messages**

NOTE

Bien que les démarches soient assez détaillées, il est supposé que le développeur possède déjà des notions de base sur l’utilisation d’un blueprint Unreal (p.ex. créer un noeud, créer une variable, relier des noeuds entre eux, créer un widget ou modifier le style d’un widget).

Également, ses notions sur l’utilisation de l’éditeur du moteur sont sollicitées par défaut (p.ex. accéder aux ressources du plugin et du projet, aux blueprints, aux fenêtres de l’éditeur, etc.).

Cette partie sur les Blueprints va de paire avec la partie sur le Code.

4.2.12 Blueprints : Widget Menu de configuration

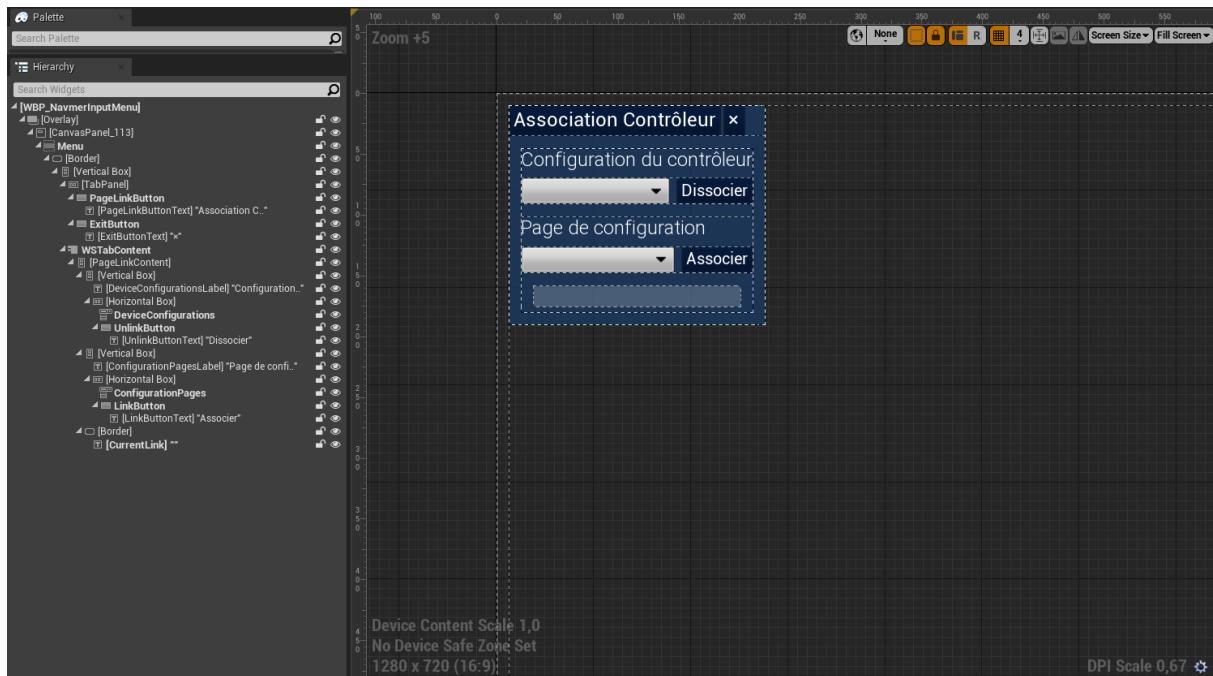


FIGURE 24 – Designer du widget blueprint menu, avec la hiérarchie des sous-widgets utilisés.

Il y a finalement très peu à comprendre sur le **designer** du widget menu.

Dans l'arbre de hiérarchie, les sous-widgets qui ont leur nom en gras sont déclarés comme des variables (i.e. que la case **Is Variable** est cochée).

Les autres widgets sont pour la plupart des widgets qui servent soit à la disposition graphique des widgets, soit à étiquetter les widgets.

La définition d'écran est 1280x720 par défaut.

Si l'on souhaite designer un widget avec une définition adaptée à un moniteur 21.5-24" (1920x1080), il faut cliquer sur le bouton **Screen Size** en haut à droite du viewport du designer et choisir l'option adéquate.

Unreal est cependant capable d'adapter automatiquement la disposition des widgets ainsi que la taille de police selon la définition employée. Il faut donc en général faire plus attention à la disposition des éléments, p.ex. pour ne pas retrouver un widget qui en survole un autre lorsque la définition est inférieure à celle utilisée pour designer.

Pour modifier l'apparence des widgets, il suffit de cliquer dessus et d'intéragir avec les différentes options du panel **Details** (qui apparaît après le clique souris).

Comme le menu est disposé de sorte à se situer dans le coin supérieur gauche de l'écran, il n'y aura à priori aucun problème de survole (du moins, pas entre les widgets du plugin NavmerInput).

L'interaction avec le menu se fait uniquement au moyen de la souris.

Les sous-widgets importants du widget menu sont les suivants :

- Le widget switcher **WSTabContent**
- Les boutons d'onglets **PageLinkButton** et **ExitButton**
- Les boutons **LinkButton** et **UnlinkButton**
- Les combobox **DeviceConfigurations** et **ConfigurationPages**
- Le textblock **CurrentLink**

WSTabContent permet de changer les onglets du menu.

Pour le moment, seul un onglet est défini : l'onglet *Association Contrôleur*.

L'onglet *Association Contrôleur* prend la forme d'un sous-widget du type *VerticalBox* qui porte le nom **PageLinkContent**.

Tous les widgets qui sont placés directement sous WSTabContent dans l'arbre sont considérés comme des onglets.

Le bouton au symbole “X” sert à fermer le menu à l'aide de la souris. Bien qu'il semble se faire passer pour un onglet, ce n'en est pas un.

Le changement d'onglet est contrôlé par les boutons d'onglets.

Changer un onglet revient à rendre invisible l'ancien onglet et à rendre visible l'onglet actuel.

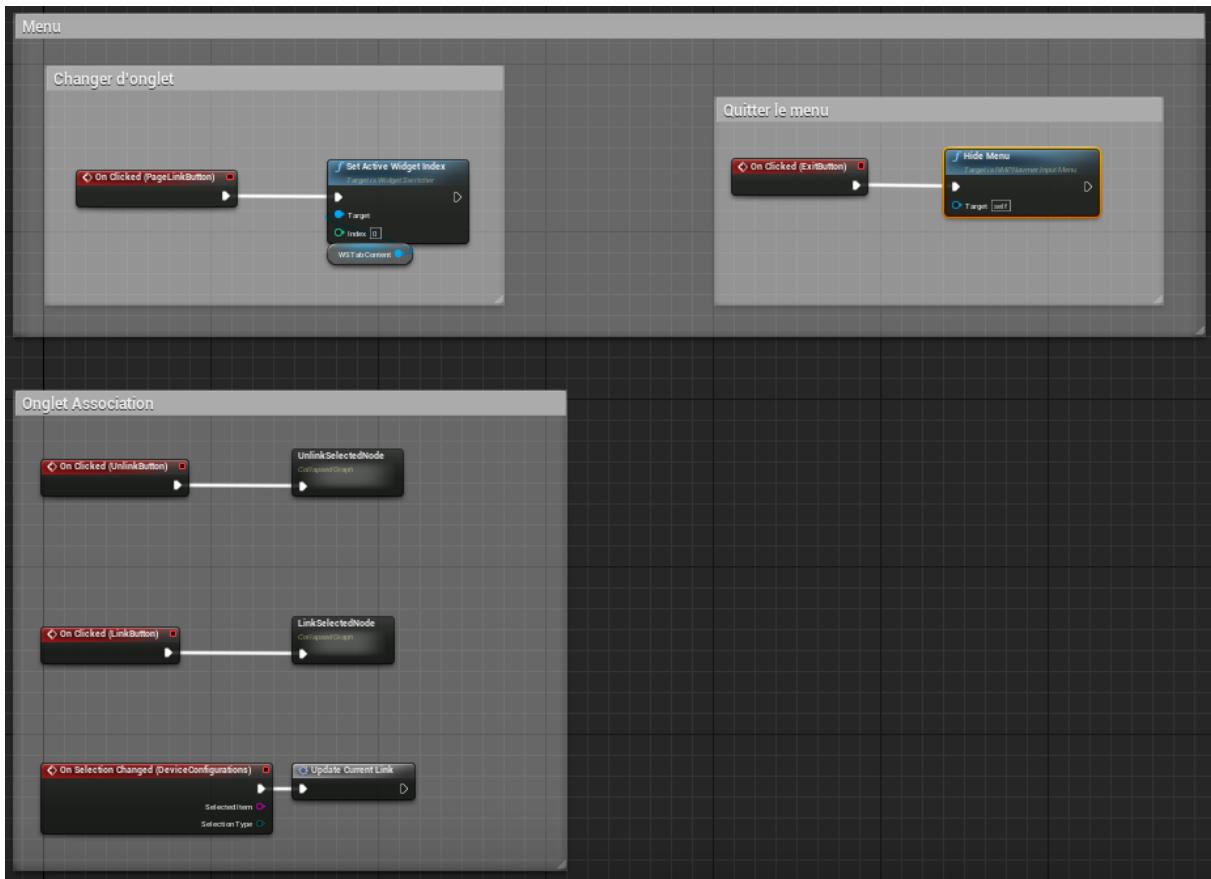


FIGURE 25 – Graphe du widget blueprint menu.

Le **graph** du widget blueprint menu contient un certain nombre d'appels fonctions et de **collapsed graph** (ce type de noeud blueprint imbrique d'autres noeuds blueprints, permettant d'organiser proprement son blueprint).

Tout d'abord, le changement d'onglet consiste à faire appel au noeud **Set Active Widget Index** en précisant l'index de l'onglet en question.

L'attribut **Target** est relié au widget switcher **WSTabContent**, accessible depuis la rubrique **Variables** sous le panel **My Blueprint** (par défaut à gauche du viewport de graphe).

L'accès à l'élément **WSTabContent** dans le graphe est uniquement possible car la case **Is Variable** a été cochée dans le designer pour ce widget.

Le noeud évènement **On Clicked (PageLinkButton)** permet d'effectuer une suite de traitement lorsque le bouton **PageLinkButton** est pressé.

Ce noeud est créé à partir du designer, en ajoutant un **Event** "On Clicked", situé sous la rubrique **Events** du panel **Details** du bouton.

Par défaut, l'onglet affiché est l'onglet *Association Contrôleur* (cf. rubrique **Switcher**, panel **Details** de

WSTabContent).

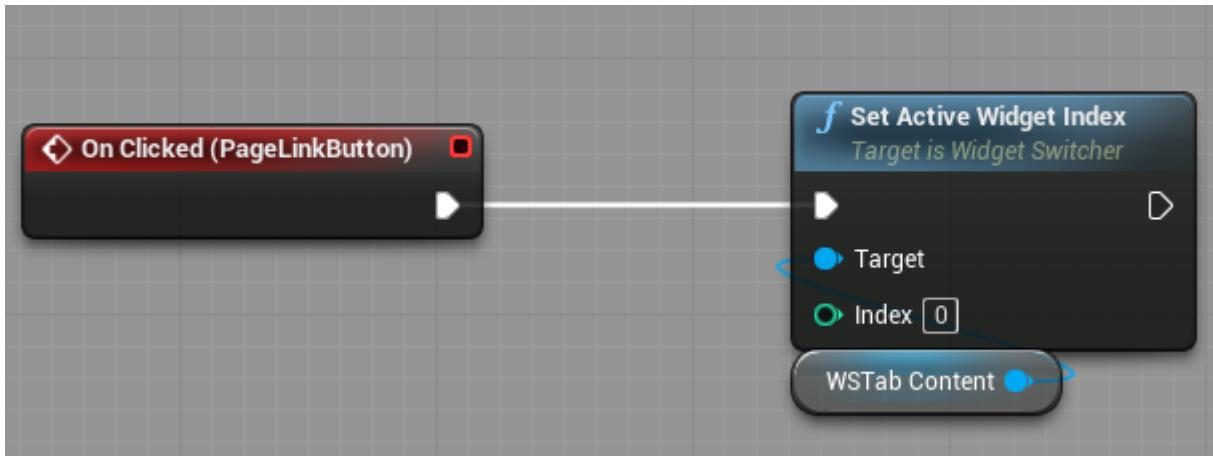


FIGURE 26 – Le noeud **Set Active Widget Index** est attaché au noeud évènement **On Clicked (PageLinkButton)**; le noeud **Set Active Widget Index** est relié au widget switcher **WSTabContent**.

Ensuite, nous avons depuis le noeud **On Clicked (ExitButton)** l'appel au noeud **HideMenu** qui correspond à la fonction HideMenu() de la classe UNMPNavmerInputMenu.

Cela est possible car la propriété Unreal **BlueprintCallable** (définie au travers de la macro *UFUNCTION()* au dessus de la déclaration de la fonction) a été précisée dans le code pour cette fonction : elle est donc appellable depuis un blueprint qui hérite de cette classe (et ailleurs depuis une instance de ce blueprint selon le specifier d'accès).

Comme WBP_NavmerInputMenu hérite de la classe UNavmerInputMenu, lorsque le widget menu est instancié au travers du blueprint : il devient également une instance de la classe UNavmerInputMenu.

C'est pourquoi l'accès à la fonction HideMenu est possible, et que l'attribut Target a comme valeur "self" (i.e. que l'appel à la fonction se fait depuis lui-même).

L'héritage d'une classe en blueprint se fait à la création du blueprint, ou en modifiant le "lien de parenté" (**Parent Class**) depuis l'outil *Class Settings* du blueprint.

Concernant le noeud **On Clicked**, c'est un noeud évènement (au même titre que On Clicked (PageLinkButton)) qui permet d'effectuer un traitement lorsque l'on clique avec le souris sur le widget.

Cliquer sur le bouton au symbole "X" revient donc à fermer le menu, ou plutôt le cacher en modifiant sa visibilité (pour rappel, la fonction HideMenu() ne détruit pas le widget menu, c'est la fonction HideNavmerInputMenu() du HUD qui s'en occupe).

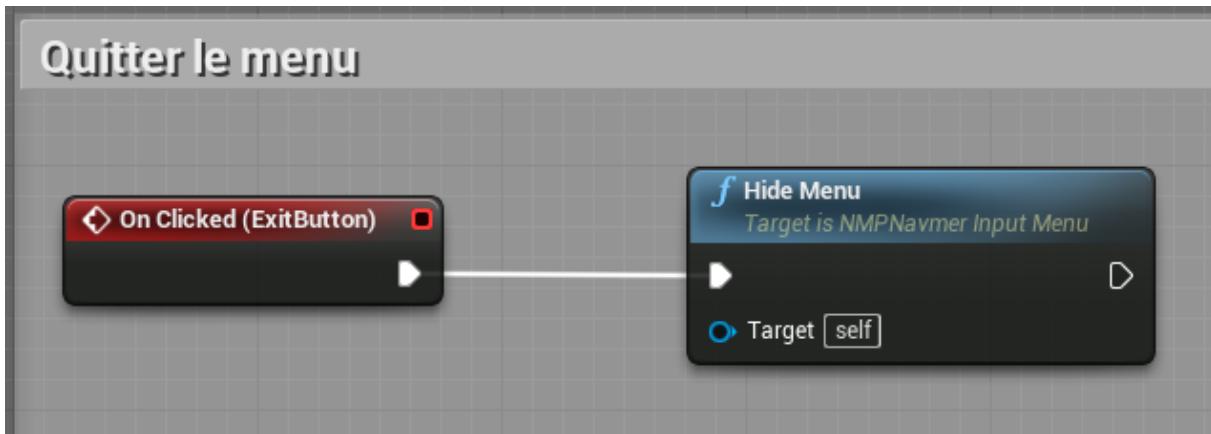


FIGURE 27 – Le noeud **Hide Menu** hérité de la classe dérivée UNMPNavmerInputMenu est attaché au noeud évènement **On Clicked (ExitButton)**. Le noeud public HideMenu est appelé depuis l’instance du blueprint.

Viens maintenant les noeuds de comportement de l’onglet Association.

Premièrement, on a le noeud **UnlinkSelectedNode** qui est rattaché au noeud évènement **On Clicked (UnlinkButton)**.

Le bouton **UnlinkButton** correspond au bouton étiqueté “Dissocier”.

Ce lien permet d’effectuer une dissociation de page de configuration pour une configuration de devices sélectionnée lorsqu’on clique sur le bouton “Dissocier” du menu.

Si l’on double-clique sur le noeud UnlinkSelectedNode (ou qu’on y accède depuis la rubrique *Event-Graph/Graphs* du panel *My Blueprint*), on arrive sur le sous-graphe de ce noeud.

Pour revenir au graphe global, il suffit de double-cliquer sur la rubrique **EventGraph**.

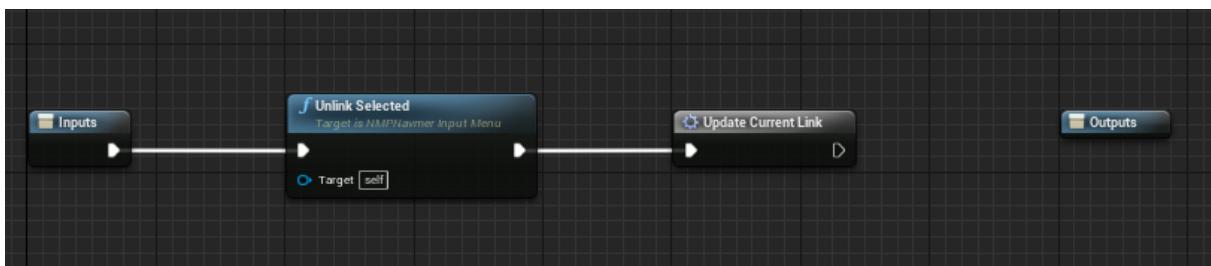


FIGURE 28 – Sous-graphe **UnlinkSelectedNode**.

On remarque les noeuds **Inputs** et **Outputs**.

Ces noeuds sont des noeuds d’entrée et de sortie d’execution, à la manière d’une fonction.

Il est possible d’y passer un lien d’execution, comme des arguments (cf. panel Details du noeud **Inputs** ou du noeud **Outputs**).

Le noeud **UnlinkSelected**, hérité de la classe UNMPNavmerInputMenu, est relié au noeud input. Sa fonction (définie dans la classe UNMPNavmerInputMenu) est de dissocier une page de configuration selon la configuration de devices sélectionnée dans la combobox DeviceConfigurations.

Le noeud UnlinkSelected est ensuite attaché à la **macro blueprint UpdateCurrentLink**.

Le principe d'une macro blueprint est identique à un collapsed graph : des noeuds paramètres, des noeuds de retour et un corps de macro composé de plusieurs noeuds.

La différence entre un collapsed graph et une macro blueprint est que cette dernière est réutilisable dans le graphe, alors que l'utilité d'un collapsed graph est purement orientée organisation du graphe.

NOTE

Il existe également la **fonction blueprint** qui est une fonction déclarée dans un blueprint.

La fonction blueprint diffère de la macro blueprint par le fait qu'elle est appellable depuis une instance de ce blueprint (i.e. qu'elle est utilisable en dehors du blueprint).

La fonction blueprint prend cependant en entrée et en sortie un seul lien d'exécution.

Si l'on accède à l'intérieur du noeud **UpdateCurrentLink** (i.e. de la même manière que pour un collapsed graph), on remarque plusieurs choses.

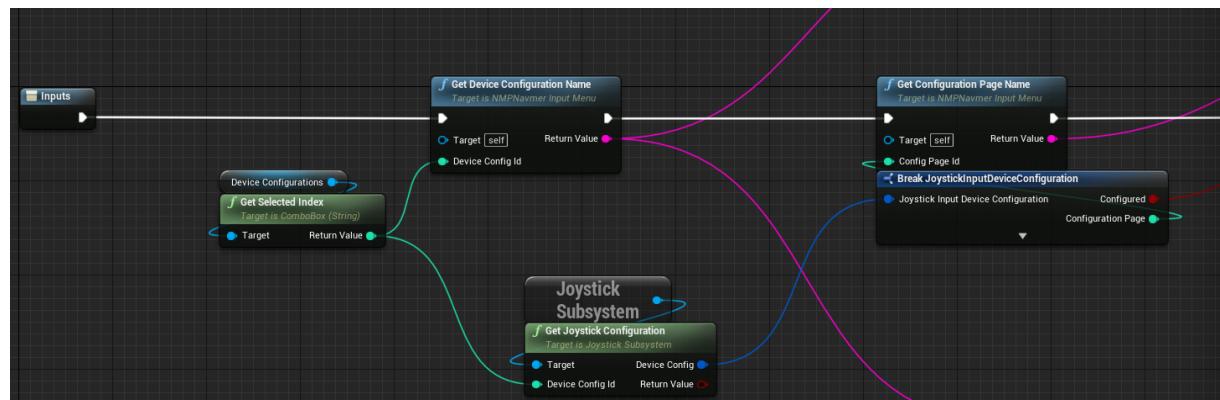


FIGURE 29 – Première partie de la macro blueprint **UpdateCurrentLink** : Accésion de la configuration de device sélectionnée et des données de la page de configuration attachée.

Le noeud **GetDeviceConfigurationName** hérité de la classe UNMPNavmerInputMenu est attaché au noeud Inputs.

Ce noeud permet d'obtenir le Device Name d'une configuration de devices suivant son index (i.e. un index dans l'ensemble des configurations de devices définies dans les paramètres du plugin). L'index est donné au noeud GetDeviceConfigurationName par l'appel du noeud **GetSelectedIndex** depuis la combobox DeviceConfigurations.

GetSelectedIndex donne l'index de l'objet sélectionné dans la combobox (i.e. la configuration de devices choisie dans la liste à l'aide de la souris).

Le noeud **GetConfigurationPageName** hérité de la classe UNMNavmerInputMenu suit l'exécution du noeud GetDeviceConfigurationName.

À l'instar de GetDeviceConfigurationName, il permet d'obtenir le nom d'une page de configuration suivant un index de page (i.e. un index dans l'ensemble des pages de configurations définies dans les paramètres du plugin).

L'index est obtenu par le biais de l'attribut **ConfigurationPage** d'une instance de FJoystickInputDeviceConfiguration.

Le noeud **Break** est utilisé pour “casser” une structure de donnée de manière à avoir accès dans le blueprint à chacun de ses membres publics.

Cette méthode n'est disponible par défaut que pour les structures de données déclarées dans un blueprint.

Pour une structure déclarée dans le code, il faut impérativement que la propriété Unreal **BlueprintType** soit précisée (i.e. à l'intérieur de la macro *USTRUCT()* au dessus de la déclaration de la structure).

Également, les attributs doivent être dotés des propriétés Unreal **BlueprintReadOnly** ou **BlueprintReadWrite**.

L'instance FJoystickInputDeviceConfiguration de la configuration de devices sélectionnée est obtenue par le noeud **GetJoystickConfiguration**, défini dans la classe du sous-système UJoystickInputSubsystem.

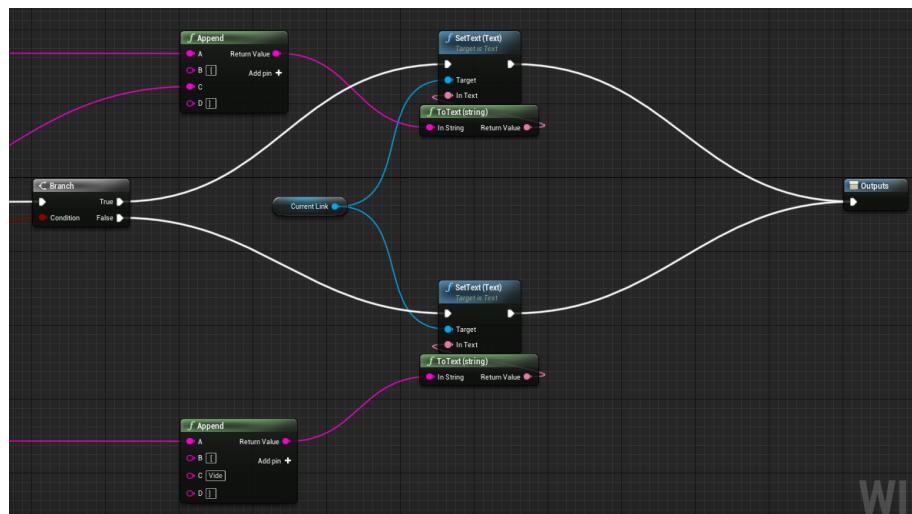


FIGURE 30 – Deuxième partie de la macro blueprint **UpdateCurrentLink : Définir le texte du textblock **CurrentLink**.**

Le noeud **Branch** suit l'exécution du noeud GetConfigurationPageName.

Son fonctionnement est identique à une expression conditionnelle “if-else”.

La condition branchée au noeud est déterminée par le booléen membre de l’instance FJoystickInput-DeviceConfiguration : **bConfigured**.

Le noeud supérieur **SetText** est attaché à la branche **True** du noeud Branch.

L’appel de ce noeud se fait depuis le widget textblock CurrentLink.

Son attribut **In Text** est branché à un noeud **Append**.

Le noeud Append correspond à une concaténation de chaîne de caractères (plus précisément, la concaténation de plusieurs objets FString).

Par défaut, il n’y a que les **pins A et B** qui sont présentes.

Les pins d’un noeud sont ses différents embranchements/ports.

Le pin **A** est relié à la valeur de retour du noeud GetDeviceConfigurationName, soit au Device Name de la configuration de devices sélectionnée.

Le pin **C** est relié à la valeur de retour du noeud GetConfigurationPageName, soit au nom de la page de configuration associée à la configuration de devices sélectionnée.

Le noeud inférieur SetText est attaché au pin **False** du noeud Branch.

Les liens sont quasiment similaires au noeud supérieur, à la différence que seul le pin **A** du noeud Append est relié : les autres ont des objets FString instanciés par le noeud Append, suivant les chaînes de caractères rentrées dans les pins.

On notera la présence du noeud **ToText(string)** qui effectue une conversion d’un objet de type FString en FText.

Les noeuds SetText sont ensuite branchés au noeud Outputs, afin que l’exécution du blueprint puisse suivre son court en dehors de la macro.

En conclusion de cette macro, si l’instance de configuration de devices sélectionnée est configurée (i.e. est associée à une page de configuration) : le textblock CurrentLink, qui a pour fonction d’afficher le status d’association de la configuration de devices sélectionnée, va afficher dans le menu le texte suivant : “DEVICE_NAME [PAGE_ASSOCIEE]”.

Autrement, il affichera le texte suivant : “DEVICE_NAME [Vide]” qui correspond à une configuration de devices affectée par aucune page de configuration.

On en revient alors au collapsed graph UnlinkSelectedNode, qui effectue donc une dissociation sur la configuration de devices sélectionnée, puis met à jour l’information sur le textblock CurrentLink.

Deuxièmement, dans le graphe global du widget blueprint du menu, on a le noeud **On Clicked (Link-Button)** qui est relié au collapsed graph **LinkSelectedNode**.

Le contenu de ce noeud est exécuter lorsque le bouton “Associer” du menu est pressé.

À l'intérieur de LinkSelectedNode, on a un contenu presque similaire à celui du collapsed graph UnlinkSelectedNode.

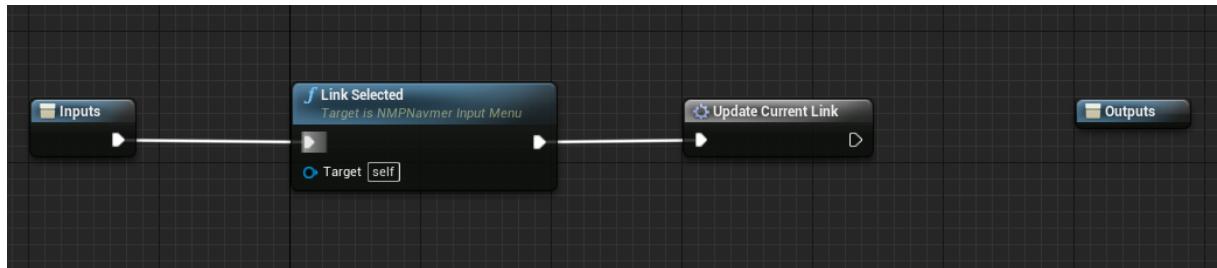


FIGURE 31 – Sous-graphe LinkSelectedNode.

La différence se situe sur l'appel du noeud **LinkSelect**, hérité de la classe UNMPNavmerInputMenu.

Ce noeud permet d'associer la page de configuration sélectionnée dans la combobox Configuration-Pages à la configuration de devices sélectionnée.

Enfin, le noeud **On Selection Changed (DeviceConfigurations)** est branché à la macro UpdateCurrentLink.

Cette suite de noeuds permet lorsque l'on change la sélection de la combobox DeviceConfigurations par un clique souris de mettre à jour le textblock CurrentLink.

4.2.13 Blueprints : Widget Messages

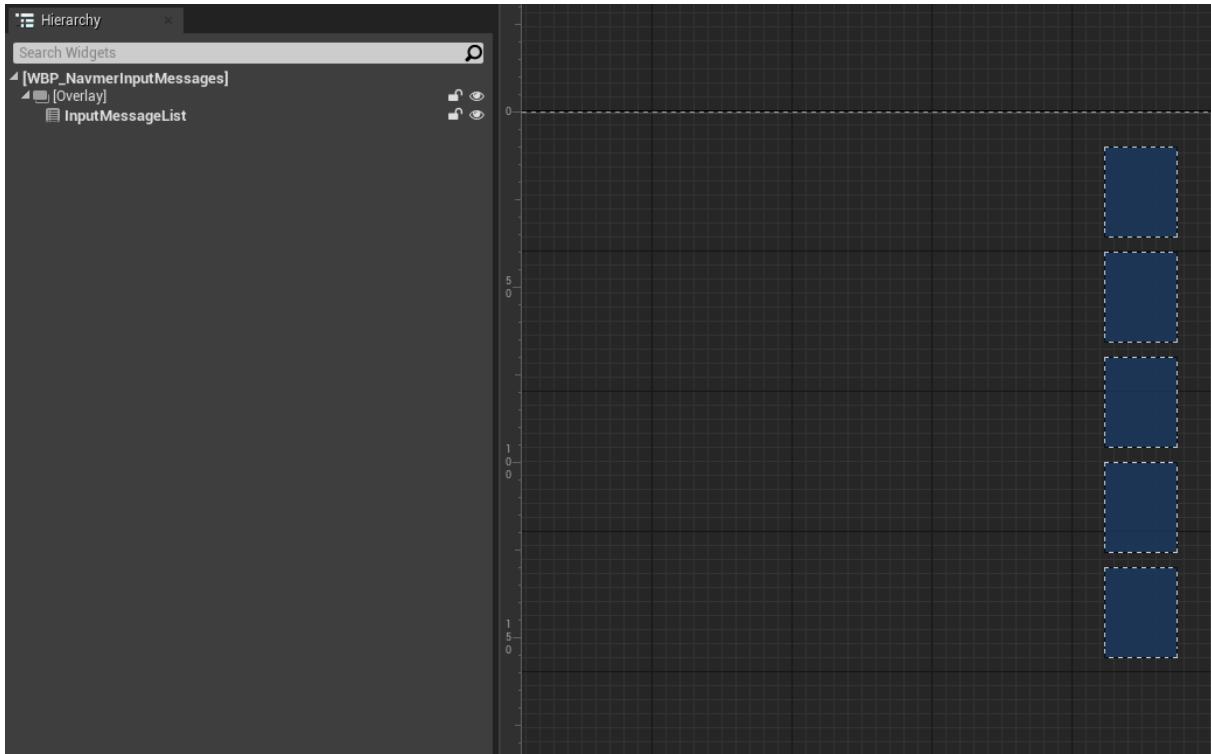


FIGURE 32 – Designer du widget blueprint messages, avec la hiérarchie des sous-widgets utilisés.

Le widget blueprint messages n'est seulement composé que d'une listview nommée **InputMessages-List** (le widget **Overlay** est juste un widget de disposition graphique).

Le visuel de cette listview est visible par la présence de plusieurs carrés dans le viewport du designer.

Ce widget n'est visible en simulation que si un élément de la listview est généré.

Les options de la listview sont configurables depuis la rubrique **List View** du panel Details de cette listview.

Dans la rubrique **List Entries** du panel Details de la listview, le champ **Entry Widget Class** permet d'assigner la classe de génération des entrées de la liste.

Dans l'éditeur d'Unreal, assigner une classe peut se faire de deux manières : en assignant une classe définie dans le code, ou en assignant un blueprint.

Assigner un blueprint permet d'avoir accès aux fonctionnalités décrites dans son graphe.

Généralement dans le projet Navmer3D, ce sont des blueprints qui héritent de classes définies dans le code qui sont définis.

Ce genre de champs/attributs prennent un type particulier d'Unreal : le type **TSubclassOf**.

Le champ Entry Widget Class doit prendre en argument une classe qui hérite de la classe UUserWidget.

Ici, il prend en argument le blueprint WBP_NavmerInputMessagesListEntry, ce qui permettra à la listview de générer des entrées ayant le type WBP_NavmerInputMessagesListEntry (et par transmission ayant le type UNMPNavmerInputMessagesListEntry).

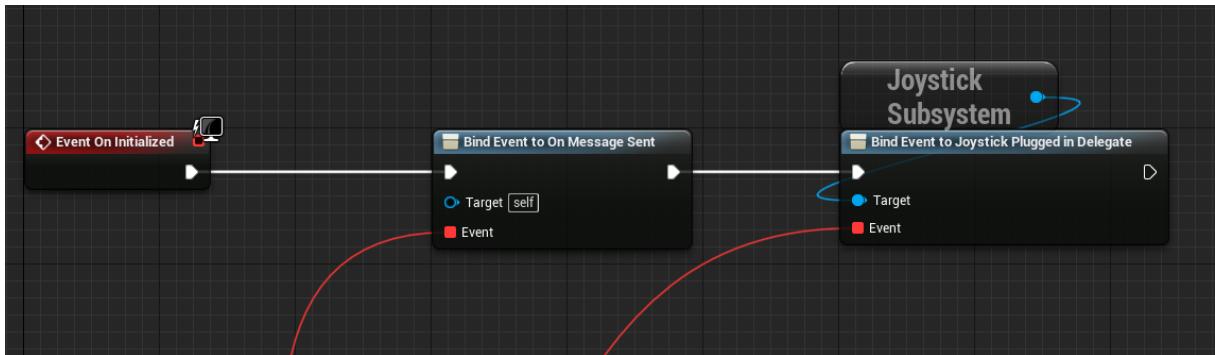


FIGURE 33 – Partie supérieure du graphe du widget blueprint messages.

La partie supérieure du graphe s'occupe d'effectuer des bindings.

Le noeud évènement **Event On Initialized** applique l'exécution qui le suit lorsque le widget doit être initialisé, au même titre que la fonction NaviteConstruct() qui n'est disponible que dans le code.

Il est obtenu en surchargeant la fonction **Initialized()** en cliquant sur la liste **Override** cachée derrière le bouton "+" au même niveau que la rubrique **Functions** dans le panel *My Blueprint* du graphe.

La fonction **Initialized()** est une fonction héritée de la classe **UUserWidget** qui porte la propriété Unreal **BlueprintImplementableEvent**.

Cette propriété permet de préciser qu'une fonction membre d'une classe doit uniquement être implémentée dans un blueprint qui hériterait de cette classe. Elle reste cependant appellable dans le code.

Le noeud **Bind Event to On Message Sent** qui suit est un noeud blueprint qui permet d'associer un noeud évènement implémentable à une delegate définie dans le code.

Pour cela, après avoir créé le noeud dans le graphe, il faut tirer un lien depuis l'attribut de sortie **Event** pour ajouter un **Custom Event** qui sera bindé à la delegate.

Ici, c'est la delegate **OnMessageSent** de la classe **UNMPNavmerInputMessages** qui est bindée dans le blueprint au noeud évènement **On Message Sent**.

Le noeud **Bind Event to Joystick Plugged in Delegate** fait le binding du noeud évènement **On Joystick Plugged In**. Il est rattaché au précédent noeud de binding et est appelé depuis le noeud du sous-système **UJoystickInputSubsystem**.

Ce nouveau noeud évènement va permettre d'effectuer un traitement dans le blueprint lorsqu'un appareil qui vient de se connecter est reconnu par le plugin NavmerInput.

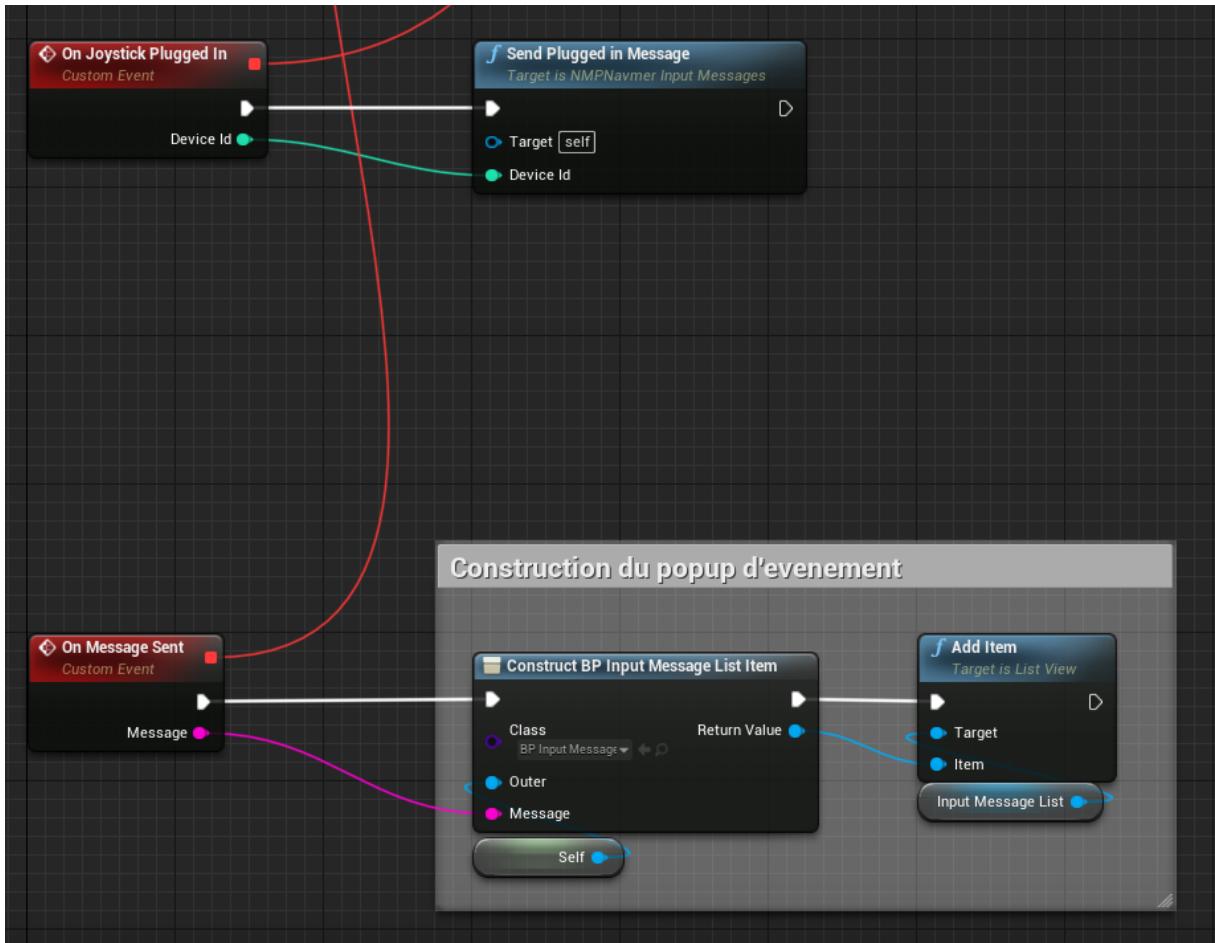


FIGURE 34 – Partie centrale et inférieure du widget blueprint messages.

La partie centrale fait appel au noeud **SendPluggedInMessage** hérité de la classe UNMNavmerInput-Messages.

Sa fonction est d'envoyer un message informant l'utilisateur qu'un device vient d'être connecté, en donnant la configuration de devices associée ainsi que sa page de configuration.

La valeur de retour **DeviceId** passée depuis le noeud évènement *On Joystick Plugged In* est envoyée dans le pin **DeviceId** du noeud *Send Plugged in Message*, afin d'avoir accès aux données du device connecté.

Enfin, la partie inférieure est liée à la construction d'une entrée de la listview.

Lorsqu'un message est envoyé au travers de la delegate *OnMessageSent*, le noeud évènement *On Message Sent* qui lui est bindé fait suivre le traitement du message, en passant par la même occasion le contenu dudit message.

Le noeud **Construct** permet d'instancier un objet dédié à un autre objet suivant une classe déterminée.

Ici, c'est une instance de BP_InputMessagesListItem qui est créée, avec l'objet **Message** en paramètre, et avec une référence vers son owner (soit une instance vers le blueprint actuel).

Il ne faut pas confondre les termes “**entrée de listview**” et “**objet de listview**”.

Une entrée de listview correspond à un widget qui est généré par la listview, alors qu'un objet de listview correspond au contenu d'une entrée de listview.

Le blueprint BP_InputMessagesListItem défini donc le contenu d'une entrée de listview (UNMPNavmerInputMessagesListEntry) : ce n'est pas un widget blueprint.

Il se comporte comme une structure de donnée pour définir un objet de la listview de la classe UNMNavmerInputMessages.

Son existence est cependant remise en question par le fait qu'il ne contient qu'une seule variable de type FString.

Sa présence permet de respecter une certaine modularité, dans le cas où, à l'avenir, la listview doit générer des entrées plus complexes : ce qui explique l'utilisation d'une structure de donnée plutôt qu'une simple chaîne de caractères que l'on aurait pu définir dans la classe UNMPNavmerInputMessagesListEntry, ou de manière similaire dans le blueprint WBP_NavmerInputMessagesListEntry.

NOTE

De manière générale, le choix de créer des variables dans un blueprint plutôt que dans la classe qu'il hérite appartient au développeur.

Cependant, il est intéressant de faire travailler les deux systèmes, de manière à séparer sur plusieurs niveaux la maintenabilité d'un projet et de la rendre plus attrayante.

NOTE

Note de l'auteur : “J'aurai pu créer une structure de donnée dans un fichier .h à la place du blueprint BP_InputMessagesListItem.

Mais bon... je trouvais pas ça terrible, surtout que c'est juste pour une seule variable.

Du coup, à l'avenir, ce blueprint peut être supprimé et remplacé par une vraie structure de donnée si les besoins sur la listview doivent être augmentés.”

L'instance de BP_InputMessagesListItem créée est retournée et passée au noeud **Add Item**.

Le noeud Add Item permet de générer une nouvelle entrée dans la listview **InputMessagesList** avec comme objet la nouvelle instance de BP_InputMessagesListItem.

4.2.14 Blueprints : Widget MessagesListEntry

Ce qu'il se passe lorsqu'une entrée de la listview est générée se présente dans le blueprint de la classe UNMPNavmerInputMessagesListEntry.

Le widget blueprint WBP_NavmerInputMessagesListEntry qui hérite de la classe UNMPNavmerInputMessagesListEntry permet de gérer l'affichage d'une entrée de la listview InputMessageList suite à sa génération.

Concernant le designer du widget blueprint WBP_NavmerInputMessagesListEntry, il n'y pas grand chose à dire autre que la présence d'un textblock **MessageText** qui va servir à afficher le message contenu dans l'objet de listview affecté.

On notera cependant que les limites du **canvas** dans le viewport du designer sont réduites à la taille du widget.

Cet effet est utile lorsque l'on souhaite créer des widgets personnalisés qui sont inclus dans d'autres widgets (l'utilisation d'un canvas aux dimensions de l'écran n'a pas de sens ici).

Pour reproduire cela, il suffit de cliquer sur la dernière liste sous la règle horizontale du viewport designer, et choisir l'option **Desired** à la place de **Fil Screen**.



FIGURE 35 – La liste **Desired** pointée par la flèche rouge, située par défaut sous la règle horizontale du viewport designer.



FIGURE 36 – Graphe du widget blueprint WBP_NavmerInputMessagesListEntry.

Le graphe global de WBP_NavmerInputMessagesListEntry est très simple.

Le noeud évènement **Event On List Item Object Set** est une implémentation blueprint de la fonction **OnListItemObjectSet()** qui est héritée de la classe UNMPNavmerInputMessagesListEntry qui implémente l'interface **IUserObjectListEntry**.

Ce noeud fait suivre une execution lorsque le widget est généré dans une listview, en passant l'objet de listview qui doit être assigné à ce widget (cf. noeud Add Item, blueprint WBP_NavmerInputMessages).

À noter que l'objet passé est générique (i.e. qu'il prend le type de base **UObject**) : il faudra le **caster** en BP_InputMessagesListItem pour avoir accès à son contenu.

NOTE

De la même manière qu'un blueprint est considéré comme une classe par Unreal, il est aussi considéré comme un type.

De ce fait, lorsqu'une instance de WBP_NavmerInputMessagesListEntry est générée par la listview InputMessageList, le contenu du collapsed graph **List Entry Initialization** va être executé.

Le noeud évènement *Event On List Item Object Set* peut être trouvé sous la rubrique **Interfaces** dans le panel *My Blueprint*.

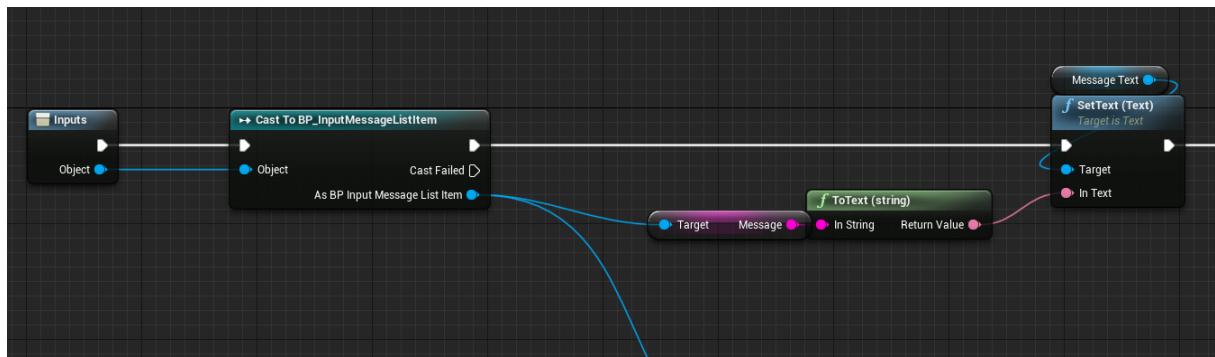


FIGURE 37 – Première partie du collapsed graph *List Entry Initialization*.

Le collapsed graph *List Entry Initialization* s'occupe du comportement que doit prendre l'entrée de listview lorsqu'elle est générée par la même listview.

L'objectif ici est d'afficher dans le textblock MessageText le message de l'objet de listview, de jouer une animation de disparition de l'entrée, puis d'exécuter sa suppression.

Tout d'abord, après avoir casté l'objet passé en BP_InputMessagesListItem à l'aide du noeud **Cast**, on accède à l'attribut **Message** par l'intermédiaire de l'instance BP_InputMessagesListItem.

On modifie alors le contenu de MessageText par la valeur de l'attribut Message.

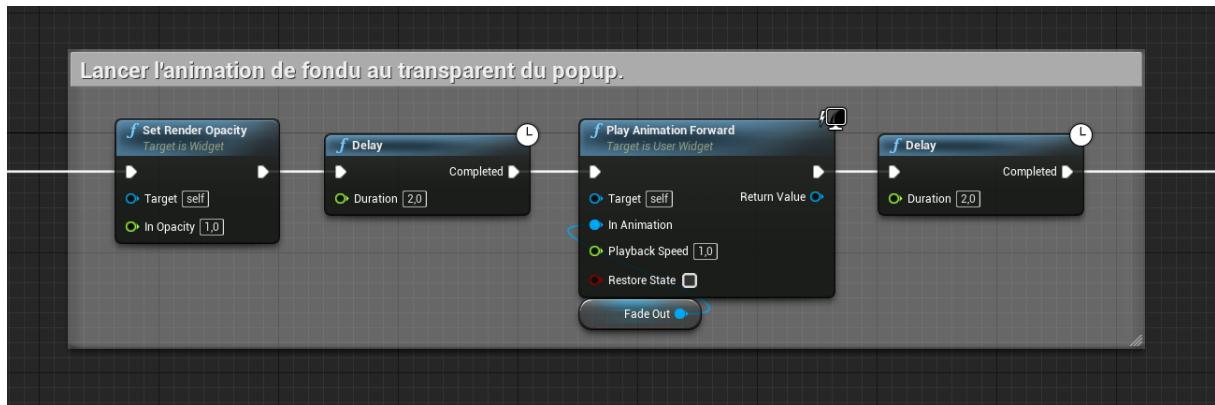


FIGURE 38 – Deuxième partie du collapsed graph *List Entry Initialization*.

Par la suite, on doit jouer une animation de disparition du widget.

Une telle animation est optionnelle, mais permet de rajouter une touche visuelle agréable à la disparition du widget.

Pour comprendre le processus de création de l'animation employée, vous pouvez consulter la partie **Blueprints : Animation du Widget MessagesListEntry**.

Le noeud **SetRenderOpacity** sert à initialisé l'opacité du widget à 1.0 (soit complètement opaque).

Avant de jouer l'animation **FadeOut**, 2 secondes sont laissées pour que l'utilisateur puisse lire le message affiché à l'écran, grâce au noeud **Delay**.

Delay est un noeud bloquant à la manière d'un "**sleep**" présent dans la plupart des langages de programmation.

L'animation de disparition du widget entrée de listview est alors jouée.

Enfin, avant d'exécuter la destruction du widget : un délai de 2 secondes est rajouté pour laisser à l'animation de disparition le temps de se conclure.

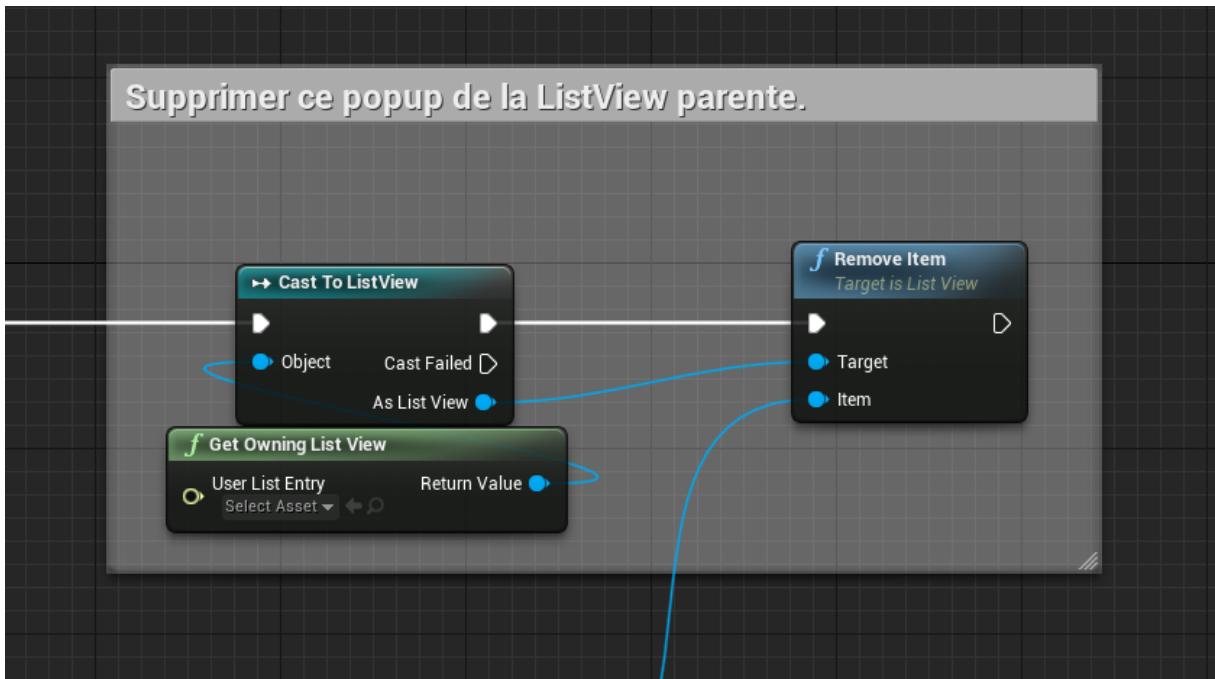


FIGURE 39 – Dernière partie du collapsed graph *List Entry Initialization*.

Pour terminer sur ce widget, il est nécessaire de détruire l’instance du widget entrée de listview pour éviter d’encombrer la listview.

Pour cela, il faut utiliser le noeud **RemoveItem** de la listview owner (soit InputMessageList de la classe UNMPNavmerInputMessages) pour supprimer le widget.

Afin d'accéder à ce noeud, il faut se servir du noeud **GetOwningListView** qui retourne la listview owner sous la forme d'une listview générique (i.e. un objet du type **ListViewBase**).

Après avoir casté l'objet ListViewBase en ListView, le noeud RemoveItem devient accessible.

On affectera au pin **Item** du noeud RemoveItem le BP_InputMessageListItem casté au départ.

4.2.15 Blueprints : Animation du Widget MessagesListEntry

 **NOTE :**

Cette partie ne concerne que la création d'une animation dans un blueprint.

La création d'une animation dans le blueprint se fait à partir des panels **Animations** et **Timeline** dans le designer du widget blueprint.

Si ces panels ne sont pas visibles par défaut, il peuvent être ajoutés en cliquant sur le bouton **Window** de la barre menu de la fenêtre du blueprint.

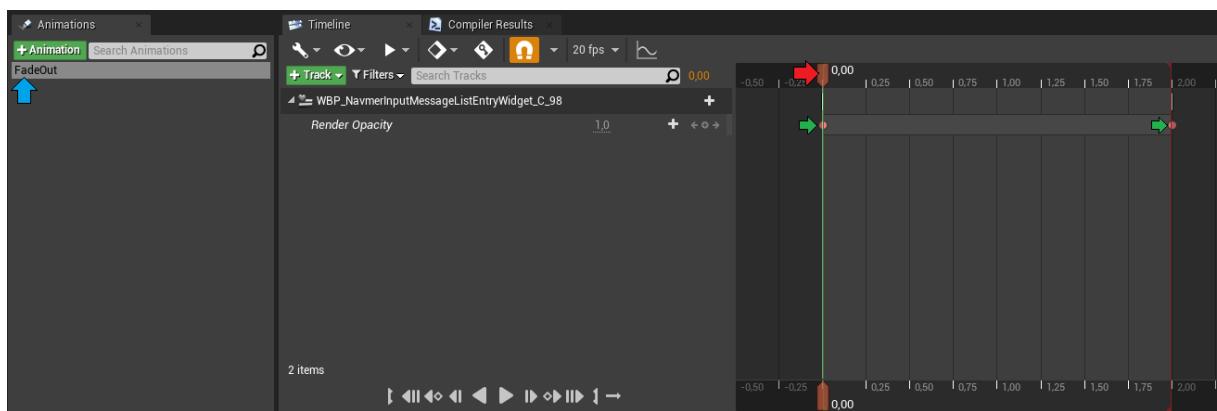


FIGURE 40 – Panels Animations et Timeline. La flèche bleue pointe sur le nom de l'animation, la flèche rouge pointe sur le curseur de la timeline, et enfin les flèches vertes pointent sur les deux points clés de l'animation.

L'animation déjà créée pour le blueprint WBP_NavmerInputMessagesListEntry s'appelle **FadeOut** : il n'est donc pas nécessaire d'en faire une.

Cependant la démarche va être expliquée :

- Pour créer une nouvelle animation il faut cliquer sur le bouton vert “+ Animation”.
- Après avoir nommé cette animation, il faut cliquer sur son nom pour avoir accès à sa timeline.
- Pour réaliser une animation de disparition, deux **clés d'animation** sont à définir : une première avec une opacité maximale, et une deuxième espacée dans le temps avec une opacité nulle.-
- Pour créer une clé, il faut d'abord placer le **curseur** de la timeline à un temps où l'on souhaite que la clé soit jouée.
- Il faut par la suite cliquer sur le bouton vert “+ Track” du panel Timeline est choisir sous l'option *All Named Widgets* la sous-option *[[This]]* : cela va permettre d'associer l'animation à toute la hiérarchie du widget WBP_NavmerInputMessagesListEntry.
- WBP_NavmerInputMessagesListEntry_C_X apparaît alors dans la liste des widgets suivis par l'animation.

- Au même niveau que WBP_NavmerInputMessagesListEntry_C_X dans la liste se trouve un bouton au symbole “+” qui permet alors, en choisissant un paramètre de WBP_NavmerInputMessagesListEntry à animer, d’ajouter une clé au niveau du curseur de timeline.
- Vous choisisrez donc le paramètre de style **RenderOpacity**.
- Si vous avez laisser le curseur au temps 0,00 (temps en secondes), la clé sera placé à ce temps-ci.
- La deuxième clé peut être placée p.ex. à 2,00 en répétant la démarche et en choisissant le paramètre RenderOpacity.
- Sous WBP_NavmerInputMessagesListEntry_C_X peut être modifiée la valeur de RenderOpacity, qui est à 1.0 par défaut (valeur dans l’ensemble [0, 1]).
- Laissez 1.0 pour la première clé et déplacez le curseur de timeline sur la deuxième clé (i.e. à 2,00).
- Modifiez alors la valeur de RenderOpacity en cliquant sur sa valeur pour la mettre à 0.0.
- Vous pouvez maintenant vérifier l’animation en remettant le curseur de timeline à 0,00 et en lançant l’animation à l’aide du **Play** en bas du panel Timeline.
- L’animation nécessite enfin d’être appelée dans un graphe.

4.2.16 Blueprints : Les widgets dans l'HUD

La mise en place des widgets du plugin se fait en accord avec le HUD, par l'intermédiaire du blueprint **BP_HUD** qui hérite de la classe ANMHUD.

Le blueprint BP_HUD se situe dans le dossier *Content/NavmerContent/Blueprints/HUD* à la racine du projet Navmer3D.

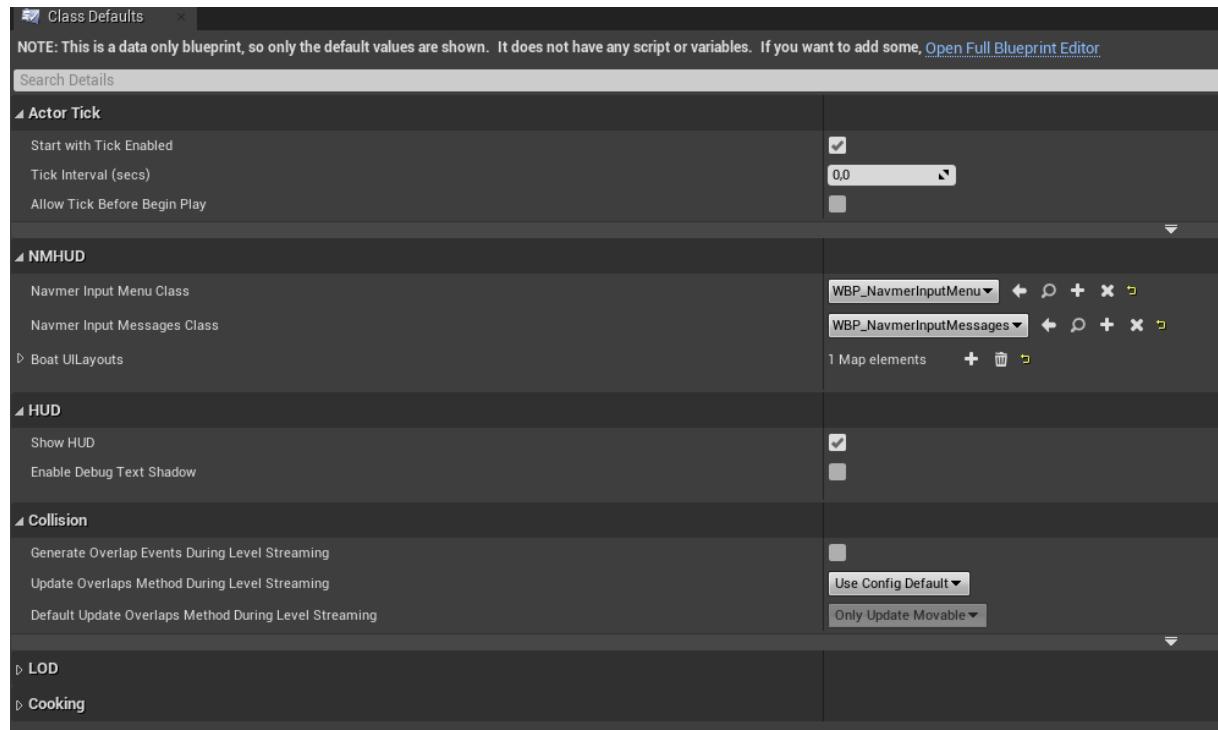


FIGURE 41 – Le blueprint BP_HUD de l'HUD de Navmer3D.

BP_HUD ne contient pas de graphe.

Afin que l'HUD génère les widgets menu et messages au travers des fonctions `ShowNavmerInputMenu()`/`ShowNavmerInputMessages()`, il faut préciser à partir de quelle classe ces widgets doivent être instanciés.

Sous la rubrique **NMHUD**, les champs *Navmer Input Menu Class* et *Navmer Input Messages Class* sont assignés par les blueprints respectifs des widgets.

Cette méthode fait partie du caractère “flexible” du code, notamment dans le cas où de nouveaux blueprints sont définis pour les widgets menu et messages.

Afin d'être sûr que le HUD est visible en simulation, il faut cocher la case **Show HUD** sous la rubrique **HUD**.

Il faut par la suite affecter le HUD au **GameMode** utilisé, afin qu'il puisse l'instancier.

Dans Unreal, le gamemode sert d'une part à gérer les données de jeu, et d'autre part à définir les règles principales qui seront appliquées en jeu (p.ex. le nombre de joueurs, les **spawns**, la pause en jeu, les transitions entre les niveaux, etc.).

Le blueprint gamemode principal du projet **BP_GameMode_V1** se situe dans le dossier *Content/Menu-System/GameModePlayercontroller*.

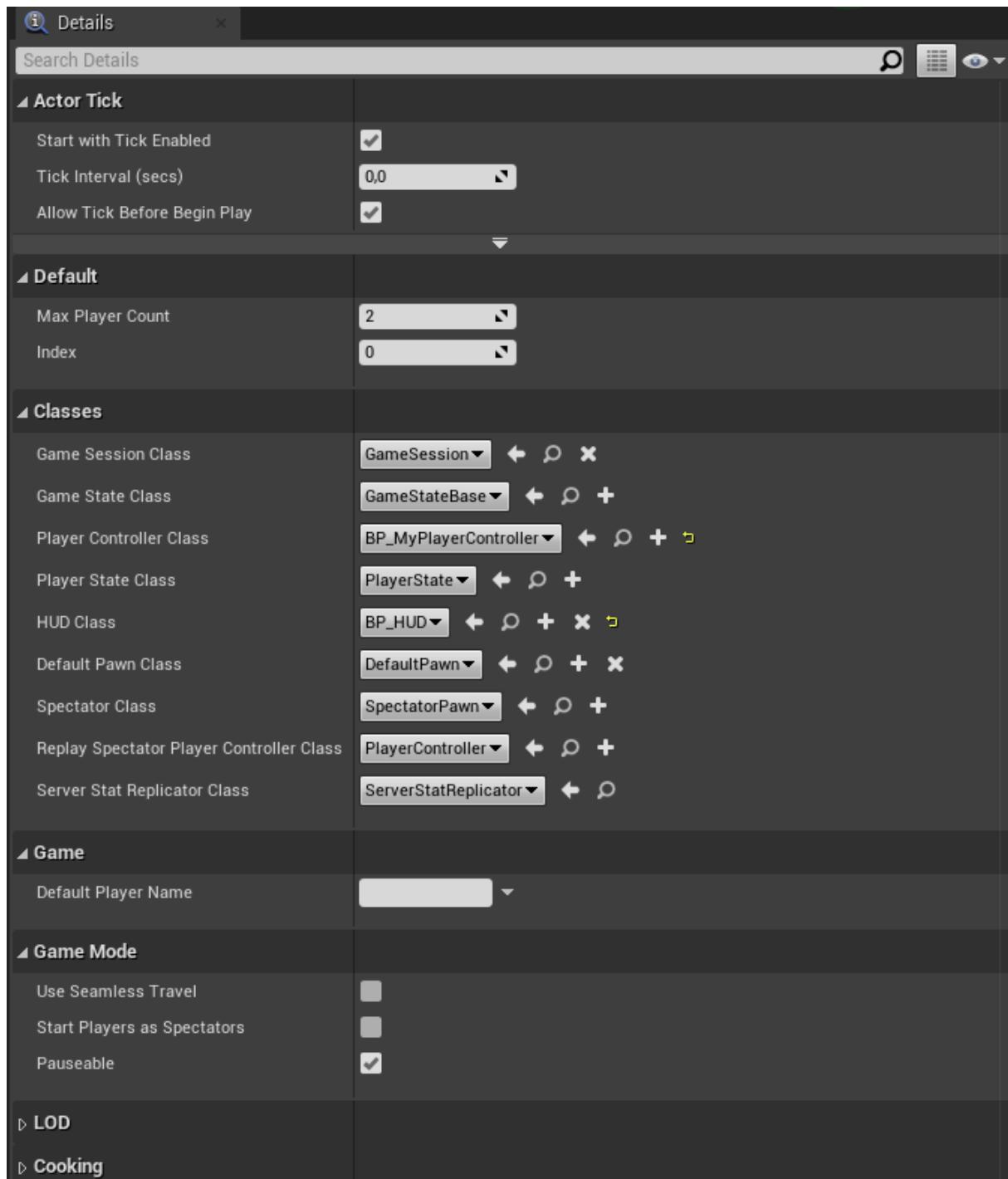


FIGURE 42 – Le panel Details du blueprint BP_GameMode_V1.

On souhaite accéder au panel Details qui est accessible en cliquant sur le bouton **Class Defaults** de la barre d'outils du blueprint.

La rubrique **Classes** permet d'assigner les classes principales du projet.

C'est dans le champ **HUD Class** que l'on doit affecter le blueprint BP_HUD.

Il faut évidemment s'assurer que le projet utilise le bon gamemode.

Pour cela, il faut se rendre dans la fenêtre/onglet **Project Settings** au travers du bouton **Edit** de la barre menu de la fenêtre principale de l'éditeur.

Dans la page **Maps & Modes**, il faut affecter au champ **Default GameMode** situé sous la rubrique **Default Modes** le blueprint BP_GameMode_V1.

Pour conclure sur le gamemode, il faut vérifier que le gamemode est affecté à chaque niveau.

En prenant le niveau **Lyon** pour exemple, il faut se rendre dans l'onglet **World Settings** à partir de la fenêtre du niveau (soit la fenêtre principale de l'éditeur, avec le viewport de la scène).

Si l'onglet n'est pas présent par défaut, il faut activer l'option **World Settings** située dans la barre menu de l'éditeur sous le bouton *Window*.

Le blueprint BP_GameMode_V1 doit alors être affecté au champ **GameMode Override** situé sous la rubrique **Game Mode** dans l'onglet *World Settings*.

Même à terme de ces opérations, l'HUD ne garantie évidemment pas la création automatique des widgets lors de la simulation.

Il faut utiliser les fonctions `ShowNavmerInputMenu()`/`ShowNavmerInputMessages()` dans le blueprint d'un niveau pour préciser l'utilisation des widgets.

Le blueprint du niveau **Lyon** possède déjà des connexions pour la mise en place de la partie HUD du plugin.

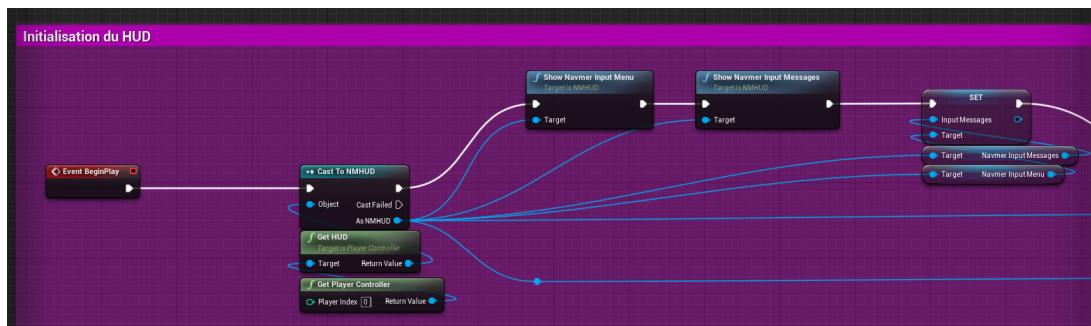


FIGURE 43 – Le blueprint niveau de la carte **Lyon** : Le noeud **BeginPlay** permet au développeur d'initialiser le niveau.

Il faut dans un premier temps faire appel au HUD du **PlayerController** actuel (**PlayerIndex** 0 pour un unique PlayerController), et de le convertir en objet ANMHUD. De là, il suffit de faire appel aux fonctions de l'objet ANMHUD pour instancier les widgets du menu et des messages.

Une référence du widget messages doit être passée au widget menu après son instantiation pour permettre une communication entre les deux.

4.3 Améliorations

Cette sous-section s'intéresse aux améliorations possibles du plugin.

Un première amélioration pourrait consister en l'ajout d'un nouvel onglet dans le menu de configuration.

Cet onglet permettrait en simulation de pouvoir modifier les paramètres des pages de configuration dans un premier temps, et dans un deuxième temps de pouvoir effectuer un étalonnage automatique des axes pour une configuration de devices donnée, en prenant en compte les bornes et le centre.

Optionnellement, le nom du plugin JoystickPlugin devrait être définitivement changé en NavmerInput (ou par un nom de votre choix) de manière à ce que Unreal puisse le reconnaître comme tel.

À l'avenir, même si le plugin est fonctionnel, il serait de bon ton de revoir son architecture, de manière à pouvoir éviter dans le code les copies de structures depuis la page de paramètres du plugin, ou encore de proposer une meilleure implémentation de l'interface Plugin - API.

Enfin, l'usage de la SDL2 peut être conservé.

Cependant, il faut noter que c'est une bibliothèque assez importante qui comprend de multiples fonctionnalités autres que le traitement des périphériques d'entrée.

De manière à optimiser les dépendances et le poids du plugin, il pourrait être intéressant d'avoir directement recours aux APIs des systèmes d'exploitation cibles (i.e. les systèmes sur lesquels seront déployé le simulateur Navmer3D).

NOTE

Si l'API Raw Input de Windows est sollicitée, il est important de vous notifier que la Ship Console 1, de par son origine double (Wilco Publishing et VRinsight), fonctionne sur des pages d'usages HID différentes de la Ship Console 2.

Enfin, plus généralement, les APIs ne traiteront pas forcément les données d'entrée de la même manière (sauf si les données réceptionnées sont brutes).

Les plages de valeurs des axes peuvent alors différer selon les APIs.

ATTENTION !

Un problème dans le code implique que, pour une configuration de devices qui peut avoir plusieurs instances, certaines clés d'axes qui ne rentrent pas dans la plage de clés de l'instance courante (ces clés d'axes ne sont donc pas utilisables par l'appareil connecté) ne sont pas remises à zéro.

Cela peut avoir des conséquences lorsque ces clés d'axes sont bindées à des fonctionnalités dans le projet, et que ces fonctionnalités s'attendent à avoir une valeur de repos de l'axe à zéro.

La cause de ceci s'explique sûrement par le fait que, lorsque la page de configuration est changée suivant l'instance de device souhaitée, le remappage des axes n'est pas réinitialisé après le

changement (dans leur page de configuration, ces axes sont remappés sur une plage de valeurs $[-1, 1]$).

Une solution temporaire est de paramétrer les axes non utilisés dans la page de configuration de l'instance de device utilisée, de manière à uniquement activer le remappage des axes et décocher la case GamepadStick.

5 Sources utiles

Les plugins existants

- Détails juridiques sur l'usage des plugins du Marketplace d'Epic Games (1) : https://marketplacehelp.epicgames.com/s/article/Can-I-share-products-with-my-team?language=en_US
- Détails juridiques sur l'usage des plugins du Marketplace d'Epic Games (2) : https://marketplacehelp.epicgames.com/s/article/What-can-a-customer-do-with-my-product?language=en_US
- Charte d'utilisation des plugins du Marketplace d'Epic Games : <https://www.unrealengine.com/en-US/marketplace-guidelines>

Péphériques HID

- Cours assez complet sur la classe HID (un peu sale ceci-dit) : <https://emrecmic.files.wordpress.com/2016/05/hid-1.pdf>
- Article sur les descripteurs de rapports HID (en anglais) : <https://learn.adafruit.com/custom-hid-devices-in-circuitpython/report-descriptors>
- Tutoriel sur la création de descripteurs de rapports HID (en anglais) : https://programel.ru/files/Tutorial%20about%20USB%20HID%20Report%20Descriptors%20_%20Eleccelerator.pdf
- Spécification USB sur la classe HID (en anglais) : https://www.usb.org/sites/default/files/hid1_11.pdf
- Documentation Microsoft : <https://learn.microsoft.com/fr-fr/windows-hardware/drivers/hid/>

Code et Blueprints

- Un repository Github qui regroupe des informations sur les technologies capables de traiter des périphériques HID : <https://github.com/MysteriousJ/Joystick-Input-Examples#hid>
- Un article du wiki communautaire d'Unreal Engine sur le développement d'un plugin orienté traitement de périphériques : <https://unrealcommunity.wiki/custom-input-device-plugin-guide-6g8yq3eb>
- Le wiki de la SDL2 : <https://wiki.libsdl.org/SDL2/FrontPage>
- L'Actor Communication avec Unreal Engine : <https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/ActorCommunication/>
- Un site de référence sur la création d'UI sur Unreal, et la documentation des propriétés Unreal : <https://benui.ca/>
- Un tutoriel vidéo sur le Widget Switcher : <https://www.youtube.com/watch?v=7-cHcyd3ZpQ>
- Un tutoriel vidéo sur la ListView : <https://www.youtube.com/watch?v=JyMEAx8-nbY>

Notes de pieds de pages

- VRinsight (**p.4**) : <http://www.vrinsightshop.com>
- Wilco Publishing (**p.4**) : <https://www.wilcopub.com>

- Étalonnage via le wizard de Windows (**p.6**) : <https://www.tenforums.com/tutorials/103910-calibrate-game-controller-windows-10-a.html>
- API *XInput* de Microsoft (**p.8**) : <https://fr.wikipedia.org/wiki/Xinput>
- Bibliothèque logicielle libre SDL2 (**p.11**) : https://en.wikipedia.org/wiki/Simple_DirectMedia_Layer
- API *Raw Input* de Microsoft (**p.13**) : <https://learn.microsoft.com/en-us/windows/win32/inputdev/about-raw-input>
- Le plugin *Enhanced Input* (**p.14**) : <https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/Input/EnhancedInput/>
- Les sous-systèmes Unreal Engine (**p.15**) : <https://docs.unrealengine.com/4.26/en-US/ProgrammingAndScripting/Subsystems>
- Usages HID (**p.42**) : <https://learn.microsoft.com/fr-fr/windows-hardware/drivers/hid/hid-usages>
- Table des pages d'usages USB (**p.42**) : https://www.usb.org/sites/default/files/documents/hut1_12v2.pdf
- Les Collections HID (**p.43**) : <https://learn.microsoft.com/fr-fr/windows-hardware/drivers/hid/hid-collections>
- Unreal Build Tool (**p.44**) : <https://docs.unrealengine.com/4.27/en-US/ProductionPipelines/BuildTools/UnrealBuildTool>
- Unreal Header Tool (**p.44**) : <https://docs.unrealengine.com/4.27/en-US/ProductionPipelines/BuildTools/UnrealHeaderTool>
- Convention de nommage d'Unreal Engine (**p.44**) : <https://docs.unrealengine.com/4.26/en-US/ProductionPipelines/DevelopmentSetup/CodingStandard>
- Article sur l'utilisation du sous-système d'Unreal (**p.47**) : <https://benui.ca/unreal/subsystem-singleton>
- Les fichiers de configuration d'Unreal Engine (**p.48**) : <https://docs.unrealengine.com/4.27/en-US/ProductionPipelines/ConfigurationFiles>
- Les Delegates avec Unreal Engine (**p.51**) : <https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/ProgrammingWithCPP/UnrealArchitecture/Delegates>
- L'objet Input d'Unreal Engine (**p.63**) : <https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/Input>