



RÉPUBLIQUE
FRANÇAISE

*Liberté
Égalité
Fraternité*



Navmer3D - Interface navigant avec l'outil UMG d'Unreal Engine

Notice opératoire

CEREMA Dtec Risques, eau et mer

Table des matières

1 Avant-propos	4
2 Interface navigant	5
2.1 Outil UMG	6
2.1.1 Widget Blueprint	7
2.2 Nouveaux widgets	11
2.3 Animations des widgets	12
2.3.1 Matériaux	13
2.3.2 Shaders	14
2.3.3 Utilisation des matériaux dans Unreal Engine	16
2.4 Objet HUD	23
2.5 Layouts	24
2.6 Minimap	25
2.6.1 Mapping des coordonnées	27
3 Installer le projet	28
4 Vue Contenu Navmer3D	30
4.1 Ressources et Blueprints	31
4.2 Code source	33
4.3 Convention de nommage	35
5 Maintenance Ressources	36
5.1 Ajouter une texture	36
5.2 Textures Navmer3D	37
5.3 Materials et Instanced Materials	39
5.3.1 Créer un Material et un Instanced Material	40
6 Création de layout	47
6.1 Designer un widget pour layout	48
6.1.1 Création	48
6.1.2 Héritage	50
6.1.3 Compilation et Résultats	52
6.1.4 Base du layout	54
6.1.5 Background	59
6.1.6 DynamicRangeBG	63
6.1.7 WorkValueText	66
6.1.8 Bindings de widgets	69

6.2	Designer un layout	72
6.2.1	Création et apparence	72
6.2.2	Connecter le layout au HUD	78
6.2.3	Appeler l'interface navigant	81
6.2.4	Définir BP_HUD par défaut	84
6.2.5	Tester un layout	87
7	Maintenance Widgets	90
7.1	Hiérarchie des classes	90
7.2	Communication des classes	91
7.3	Propriétés de PointerWidget	94
7.4	Graphe WBP de WindMeter et StreamMeter	96
7.5	NavMap et Carte ENC	98
7.5.1	Parser	99
7.5.2	Configuration	102
7.5.3	Mise à jour	103
8	Améliorations	106
9	Sources utiles	107
10	Annexes	109

1 Avant-propos

Ce document technique couvre l'ensemble du projet d'interface navigant de Navmer3D sur Unreal Engine en proposant des explications et tutoriels sur les fonctionnalités centrales à l'implémentation de l'interface et sur les tâches de développement.

Ce document n'est pas exhaustif dans la réalisation des nombreux composants, mais plutôt générique. Ceci est dû au fait que la grande partie du développement de l'interface navigant est répétitive et gravite autour de la production graphique.

Ecrit cependant pour être assez complet, le document vise la reproductibilité en traversant l'ensemble des points importants et en abordant suffisamment les fonctionnalités utilisées (l'outil UMG, l'éditeur de matériaux, etc.).

Dans ce document, il est supposé que le lecteur a des notions minimales sur des concepts informatiques généraux.

De plus, il suppose que le lecteur a quelques compétences avec Unreal Engine.

Par exemple, très peu d'explications seront faites sur la programmation de classes code (i.e. la programmation C++ avec Unreal Engine), et les explications autour de la programmation via les graphes d'événements des blueprints (visual scripting) n'entreront pas dans les détails.

Certaines explications sont délibérément succinctes, mais accompagnées de liens en pied de pages pour permettre au lecteur d'avoir des informations supplémentaires (p.ex. un lien dirigé vers une documentation Unreal Engine pour l'utilisation d'un méthode spécifique, ou un lien vers un article exhaustif reprenant un terme employé).

 **NOTE :**

Les liens sources sont majoritairement des liens vers des sites anglophones.

Egalement, il est supposé tout le long que l'installation Unreal Engine du lecteur soit en anglais.

Le document peut être séparé en trois parties :

1. Un point d'entrée sur les fonctionnalités principales de l'interface (cf. [Interface navigant](#))
2. La configuration du projet Navmer3D (cf. [Installer le projet](#) et [Vue Contenu Navmer3D](#))
3. Et enfin, la maintenance de l'interface (i.e. tout le reste)

2 Interface navigant

L'interface de navigation de Navmer3D permet à l'utilisateur du simulateur (en outre, le navigant) d'avoir à sa disposition sous une projection visuelle un jeu de données de navigation cohérent avec un scénario simulant un environnement. L'interface de navigation doit fournir un ensemble suffisant de données permettant une manoeuvrabilité proche d'un scénario sur terrain.

Par soucis de configuration (portable, mono-écran) et d'immersion, la méthode d'affichage choisie pour l'interface est **l'ATH**¹.



FIGURE 1 – Capture d'écran de l'ATH pour une disposition bateau à hélices.

Cette partie permet d'introduire certaines fonctionnalités d'Unreal Engine utiles à la conception de l'interface de navigation de Navmer3D. Elle introduit également au nouveau contenu développé pour l'interface de navigation.

1. "Affichage tête haute". Les informations des tableaux de bord sont superposées au champ de vision du navigateur.

2.1 Outil UMG

Dans Unreal Engine, le design d'interface utilisateur se fait au travers de l'outil visuel **UMG** (de l'anglais “Unreal Motion Graphics”).

L'outil UMG permet de construire une interface avec des objets visuels appelés **Widgets**.

Un widget personnalisé est conçu comme un agrégat de widgets. Unreal Engine met à disposition du développeur un ensemble de widgets atomiques (conteneurs, boutons, sliders, textboxs, images, etc.).

De ce fait, une interface réalisé au moyen de l'outil UMG peut donc être définie comme un widget assemblé avec d'autres widgets.

Egalement, chaque widget a une fonction bien définie. Les widgets possèdent leur propre type de blueprint, héritent d'une classe code et sont munis d'un graphe de script.

2.1.1 Widget Blueprint

Les widgets sont créés à l'aide d'un type particulier de blueprint : le **widget blueprint**.

Le widget blueprint contient deux principaux onglets : **Graph** qui est commun à tout blueprint, et **Designer** pour la création visuelle.

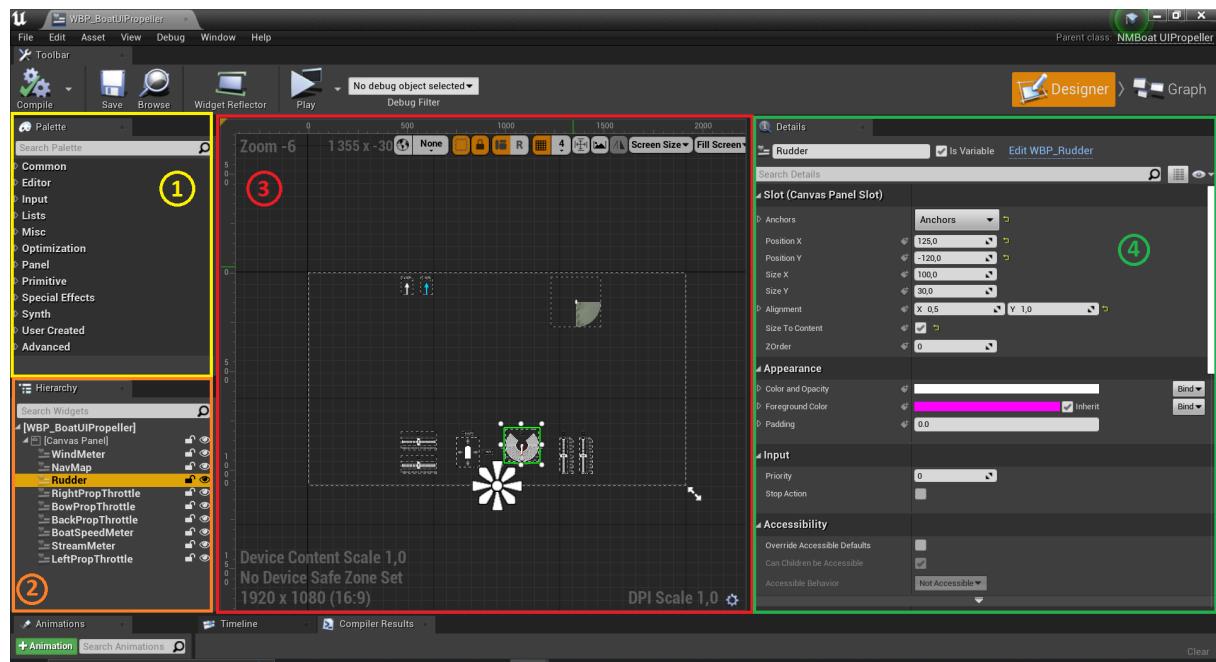


FIGURE 2 – L'onglet Designer du widget. En (1) la **Palette**, en (2) la hiérarchie, en (3) le viewport et en (4) les propriétés.

L'onglet Designer comprend tout d'abord le viewport central : un espace gradué avec une grille pour placer, sélectionner et modifier les différents widgets composants.

Dans le viewport, il est possible à l'aide de la molette de la souris de zoomer et dézoomer à souhait, ainsi que de se déplacer à l'aide du clic droit souris.

Le Designer se muni également de plusieurs panneaux utiles.

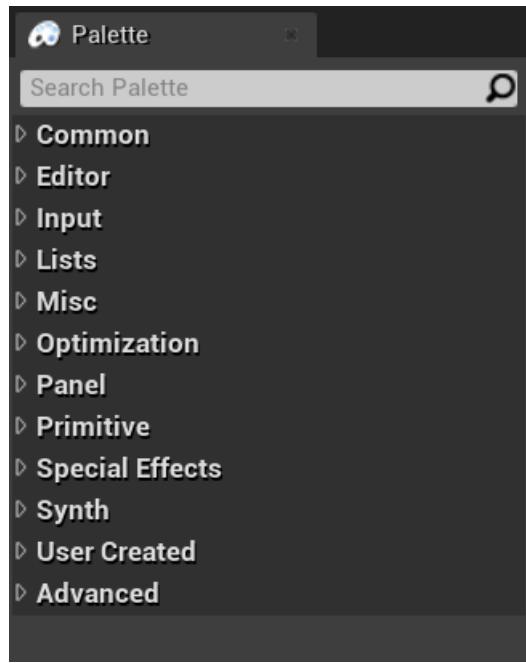


FIGURE 3 – Le panneau Palette. On peut noter les différentes rubriques où sont rangés les widgets.

La **Palette** permet d'insérer dans le viewport les widgets préexistants du moteur (p.ex. Button, Slider, Text, etc.), ainsi que d'autres widgets personnalisés (créés ou importés dans le projet) depuis la rubrique “User Created”.



FIGURE 4 – Le panneau Hiérarchie. On peut noter sous **[WBP_BoatUIPropeller]** l’ensemble des widgets utilisés pour cette interface.

Le panneau **Hiérarchie** permet d’organiser les widgets et de définir les liens de parenté entre eux.

Le lien de parenté se traduit visuellement par des coordonnées relatives dans le viewport entre les widgets, ou bien une relation conteneur-contenant.



FIGURE 5 – Le panneau des propriétés.

Le panneau **Details** (ou de “propriétés”) apparaît lorsque l’on sélectionne un widget dans le viewport ou dans la hiérarchie.

Ce panneau permet d’afficher et modifier les propriétés du widget cible : position, apparence, comportement (au moyen de paramètres), etc.

Comme n’importe quel blueprint, le widget blueprint hérite d’une classe code.

Lorsque le développeur souhaite créer un nouveau widget, le blueprint hérite généralement de la classe **UUserWidget**, ou bien d’une classe qui hérite de UUserWidget.

Dans le cas de l’interface de navigation de Navmer3D, tous les widgets créés spécialement pour le projet héritent directement ou indirectement de UUserWidget.

2.2 Nouveaux widgets

Comme le projet Navmer3D vise à être greffé à la librairie externe de modélisation physique des mouvements **libnavmer**, l'interface de navigation fait intervenir deux types de widgets :

- Les widgets d'états de simulation
- Les widgets d'états de commandes

Les **widgets d'états de simulation** sont des widgets qui transmettent visuellement au navigant des données de navigation propres au navire et à l'environnement.

Exemples :

- Orientations et forces du vent et du courant marin,
- Position géographique, vitesse longitudinale et latérale du bateau

Quelques exemples de widgets d'états de simulation :

- Indicateur de vent et de courant marin
- Indicateurs de vitesse du bateau
- Minimap

Quant aux **widgets d'états de commandes**, ils transmettent visuellement au navigant un retour d'information sur les commandes de navigation (angle en degrés, pourcentage de propulsion, etc.).

Exemples :

Consignes et valeurs réelles de commandes

La consigne de commande représente simplement la valeur perçue pour l'*input*² effectué par le navigant (soit la manoeuvre souhaitée).

NOTE :

Les inputs des Ship Consoles et de l'Azimuth Console sont gérés au travers du plugin **NavmerInput**.

À la différence, la valeur réelle de commande représente la valeur physique reproduite par le moteur sous-jacent à la commande qui augmente graduellement en fonction du régime.

Quelques exemples de widgets d'état de commande :

- Gouvernail
- Manettes de propulsion

Ces widgets travaillent par l'accès à des **structures d'états spécifiques** qui permettent l'échange des données.

2. Boutons, axes, etc. d'un dispositif de commandes (p.ex. les Ship Consoles pour les bateaux à propulsion par hélices).

2.3 Animations des widgets

L'animation des widgets (rotation, jauge, minimap, etc.) est réalisé à l'aide d'une ressource particulière proposée par Unreal Engine : le **Material** (ou "matériaux").

La manipulation des matériaux sous Unreal Engine combine la **programmation shader** et le système de **visual scripting** du moteur.

 **NOTE :**

Les explications qui suivent ne vont pas en profondeur du sujet discuté, mais permettent au lecteur d'y voir plus clair sur la création de matériaux dans Unreal Engine et sur les manipulations liées à l'animation des widgets de l'interface de navigation.

2.3.1 Matériaux

Dans le domaine de la synthèse d'image et particulièrement du rendu 3D, un **matériaux** est une collection de propriétés qui permettent de spécifier la nature de *la surface d'un objet* (un objet défini par un modèle géométrique ou analytique), en vue de calculer la couleur (et donc l'apparence) perçue à l'écran de cet objet.

En rendu moderne, cette collection de propriétés va permettre de faire du "**shading**".

Grossièrement : le shading est le procédé permettant de définir la couleur d'un objet à afficher à l'écran, en prenant en compte la lumière qui entre en contact avec la surface de l'objet (la lumière est fournie par un système d'éclairage dans la scène).

Il en résulte donc à l'écran des nuances de couleur qui permettent d'ajouter de l'information de profondeur qu'il n'y aurait pas avec des couleurs dites "plates" (sans ombrage).

NOTE :

À noter que cette définition du shading n'est peut être pas la meilleure, mais suffit à comprendre le principe.

De plus, un matériau permet *généralement* de calculer le shading.

Ce n'est pas forcément vrai puisqu'il est possible via un matériau d'opérer sur la géometrie d'un objet en utilisant certaines techniques nécessitant l'utilisation de textures (cf. "displacement mapping").

Ces propriétés d'objets sont diverses :

- sa couleur (diffuse, spéculaire, etc.)
- sa brillance, sa métalisation et sa rugosité (dans le cas de matériaux complexes)
- une ou plusieurs texture(s) pour les détails
- etc.

Aujourd'hui, ces collections de données sont traitées par la carte graphique, au travers des shaders.

2.3.2 Shaders

La carte graphique (en outre, le GPU) suit une construction d'étapes, appelée “pipeline graphique”, pour passer d'un objet 3D dans une scène à un ensemble de pixels affichés à l'écran.

Les **pipelines graphiques**³ utilisés aujourd’hui pour les applications temps-réel (p.ex. jeux-vidéos) sont dits “dynamiques”. Ils apportent plusieurs solutions pragmatiques pour le développement graphique, notamment les **shaders**.

Les shaders sont des programmes exécutés par le GPU qui donnent au développeur un certain contrôle sur le pipeline graphique. Il existe plusieurs types de shaders pour les différentes étapes du pipeline graphique.

Les principaux étant le **vertex shader** qui opère lors de la *phase de construction des primitives géométriques* (i.e. les polygones d'un objet), et le **fragment shader** qui opère après la *phase de rasterisation*⁴.

Ces programmes fonctionnels prennent en entrée les données de l'étape du pipeline qui précède celle où ils opèrent. Le shader renvoie alors une sortie pour l'étape suivante du pipeline graphique.

Exemples de données d'entrée :

- coordonnées et autres attributs de sommet (vertex shader)
- coordonnées de pixel en construction (fragment shader)

Exemples de sorties :

- coordonnées d'un sommet mappées dans un système de coordonnées différent (vertex shader)
- couleur finale d'un pixel (fragment shader)

Le développeur peut également passer au shader des paramètres supplémentaires (les variables *uniforms*).

3. Même s'il existe différents modèles de pipelines graphiques (ou de “rendu”) (p.ex. selon l'API graphique utilisée ou la méthode de rendu employée), le modèle conceptuel reste similaire : https://en.wikipedia.org/wiki/Graphics_pipeline

4. Etape du pipeline graphique temps-réel qui succède à l'*étape de projection des primitives* et qui consiste à tracer et remplir les polygones sur le *plan image* (qui représente l'écran).

Puisque le programme est executé sur une architecture hautement parallélisée⁵ (en l'occurrence le GPU), il intervient dans le cadre de la programmation parallèle.

Cela signifie que le code du shader (compilé au préalable côté CPU) est executé indifféremment pour chaque donnée concernée. Le traitement de ces données se fait presque simultanément, et de manière “aveugle” (i.e. que pour une donnée, les données voisines ne sont pas accessibles).

Pour donner un exemple, le fragment shader opère sur chaque *fragment de pixel*⁶ : les instructions vont donc s’appliquer indifféremment à tous les fragments. Egalement, puisqu’on reçoit en entrée les coordonnées **d’un** pixel en construction de l’écran : impossible d’accéder aux autres coordonnées.

Toutes ces caractéristiques impliquent donc une manière de programmer différente qu’en séquentiel et sur CPU. Il n’y a également pas de moyens conventionnels de débugger le code (p.ex. pas de moyens de consulter les valeurs numériques) : la seule manière est de constater ce qui se passe à l’écran visuellement … et de corriger lorsque ça ne va pas.

Enfin, le lien entre les matériaux d’Unreal Engine et les shaders est fort.

Dans le cadre de l’interface navigant de Navmer3D, l’objectif est l’animation d’UI (i.e d’éléments 2D) via **l’éditeur de matériaux** d’UE, similaire à la programmation de fragment shader.

5. Le GPU repose sur une architecture *SIMT* (“Single Instruction Multiple Threads”) : <https://cvw.cac.cornell.edu/GPUArch/threadcore>

6. Terme exact pour désigner la collection de données fournie par l’étape de rastérisation (on ne parle pas encore de “pixel”).

2.3.3 Utilisation des matériaux dans Unreal Engine

Prenons l'exemple de la jauge dynamique du widget de gouvernail pour montrer **l'éditeur de matériau**. La jauge dynamique du widget de gouvernail est un matériau utilisé dans Navmer3D pour l'animation du widget de gouvernail.

Lorsque l'on ouvre une ressource de type *Material* dans Unreal Engine, une fenêtre apparaît :

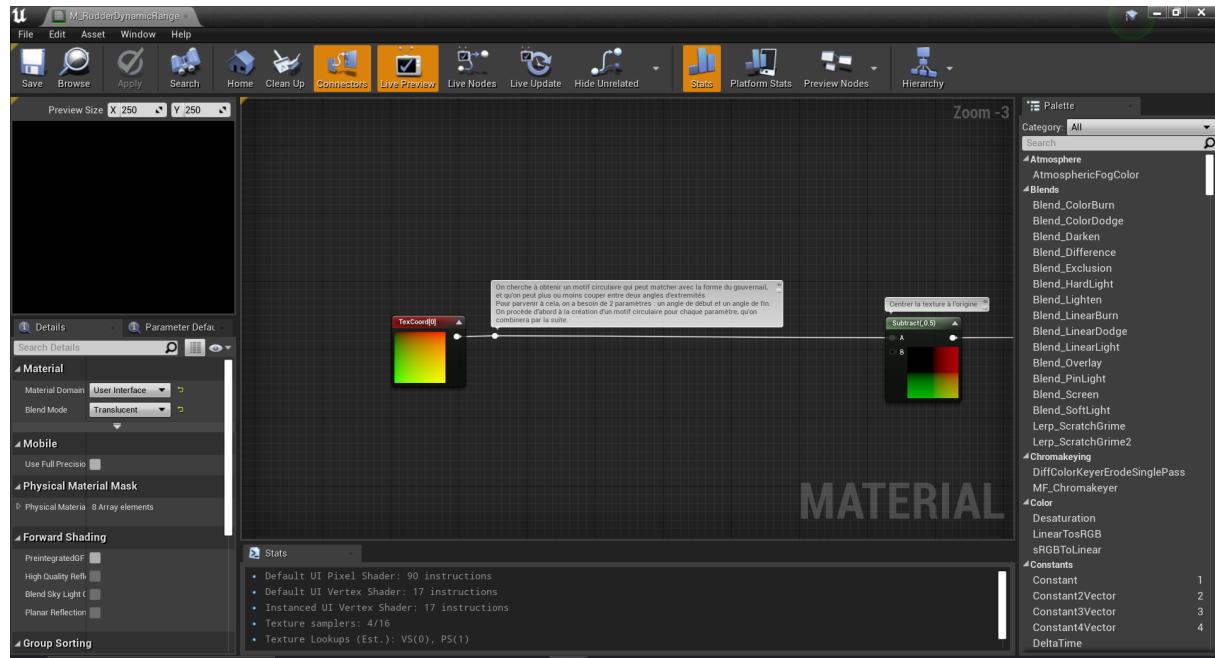


FIGURE 6 – Le matériau du widget gouvernail.

Cette fenêtre correspond à *l'éditeur de matériau*.

L'outil présent fonctionne sur un principe équivalent à la programmation shader. Le développeur utilise ici un graphe dans lequel des noeuds de structure et des noeuds fonctionnels doivent être connectés.

L'interface de l'outil dispose de plusieurs panneaux.

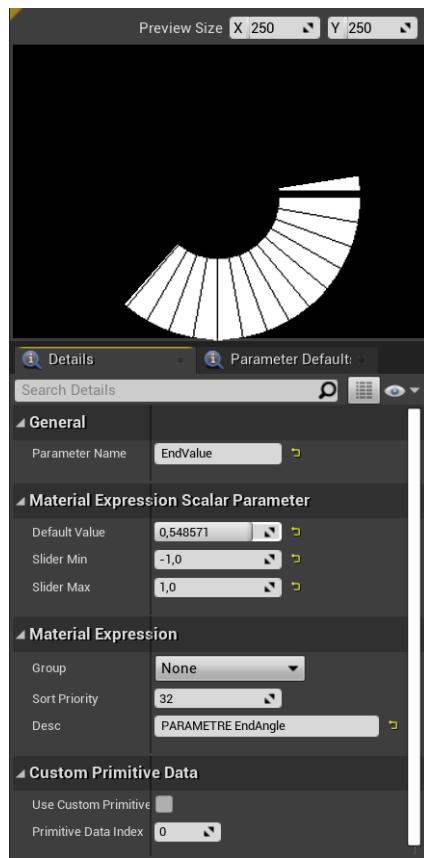


FIGURE 7 – Le panneau gauche de visualisation et des propriétés.

Le panneau gauche permet plusieurs choses :

- visualiser le résultat du matériau (en temps-réel si “Live Preview” activé dans la barre d’icônes en haut de la fenêtre)
- consulter et modifier les propriétés du matériau (ou d’un noeud s’il est sélectionné) dans l’onglet “Details”
- consulter et modifier les valeurs par défaut des paramètres passés au matériau (s’il y en a) dans l’onglet “Parameter Defaults”

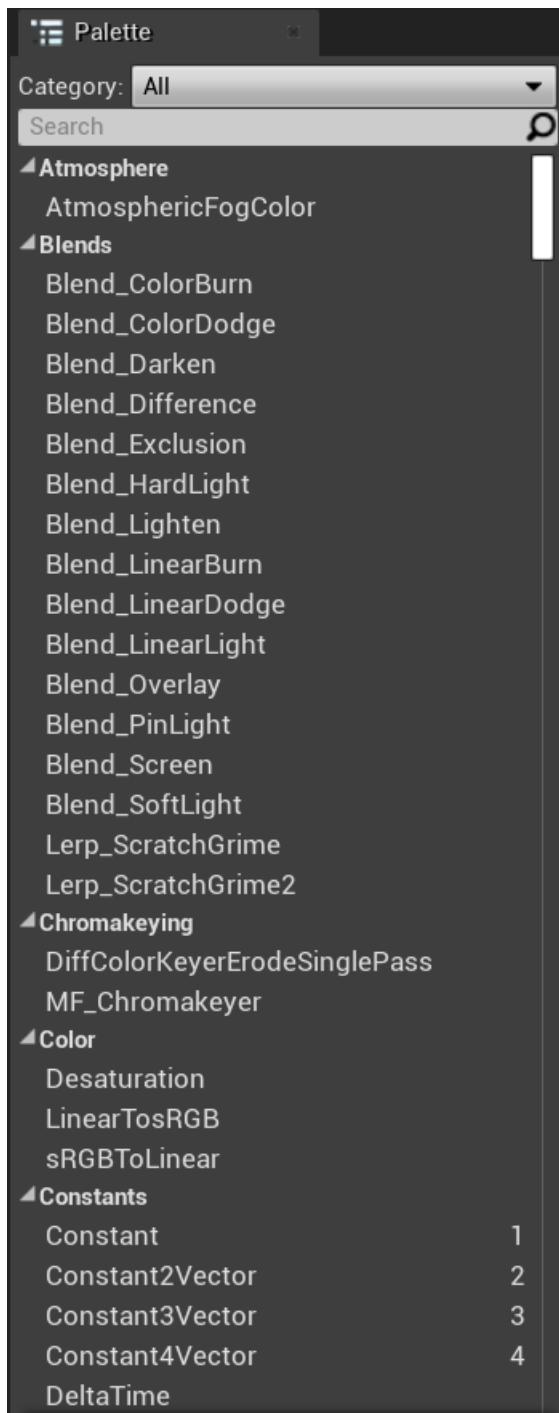


FIGURE 8 – Le panneau de la Palette.

Le panneau droite est la **Palette**.

Elle permet d'insérer dans le graphe un noeud choisi dans la collection de noeuds offerte par Unreal Engine.

Enfin se trouve au centre de la fenêtre la grille du graphe et les noeuds.

On peut également accéder à la Palette avec un menu contextuel en effectuant un clique droit dans la grille.

À l'instar d'un shader, on dispose de **noeuds d'entrées**, d'un **noeud de sortie**, de **noeuds paramètres** et de **noeuds d'instructions**.

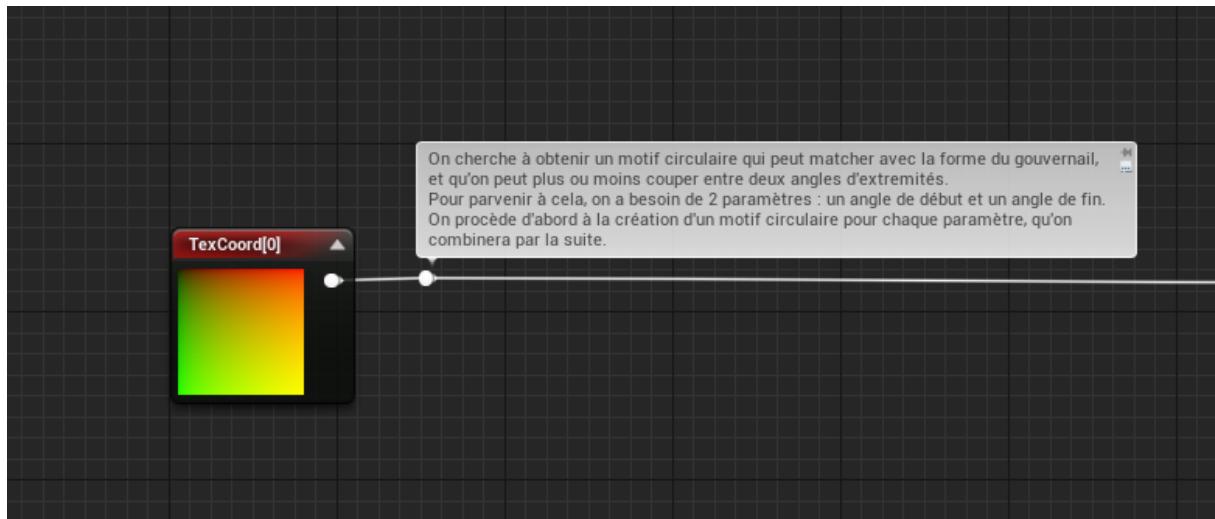


FIGURE 9 – Le noeud d'entrée du matériau. On peut aussi noter le noeud commentaire à sa droite.

Le matériau de la jauge du widget possède un seul noeud d'entrée qui est le noeud **TexCoord[0]**. Ce noeud permet d'obtenir les coordonnées des texels⁷ avec lesquels l'on va travailler.

NOTE :

Unreal Engine emploie ici le mot “texel”, et non *fragment*.

Comme le graphe de la jauge dynamique du widget gouvernail travaille sur une texture, le terme *texel* est alors approprié.

Il est important de rappeler que toutes les modifications qui suivent vont s'appliquer indifféremment à tous les texels.

L'accès aux coordonnées d'un texel va nous permettre d'effectuer toutes sortes de transformations géométriques (i.e. translation, rotation, mise à échelle de la texture) pour animer notre jauge.

À noter que chaque noeud possède au moins une **pin d'entrée** (à l'exception d'un noeud d'entrée) et au moins une **pin de sortie** (à l'exception du noeud de sortie).

Ces pins permettent de brancher les différents noeuds entre eux (similairement aux autres outils de graphes d'Unreal Engine).

7. Terme pour différencier les pixels d'une texture/image à ceux qui seront affichés à l'écran.

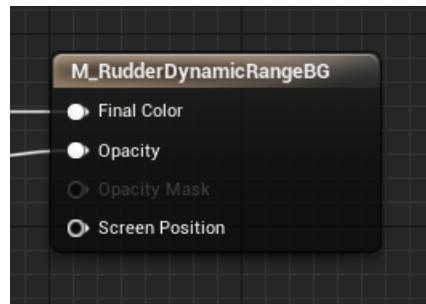


FIGURE 10 – Le noeud de sortie du matériau.

Tout matériau a un noeud de sortie pour définir la couleur finale que va porter la surface de l'objet auquel sera assigné le matériau.

Le nombre et la nature des pins d'entrée du noeud de sortie change en fonction du **domaine de matériau** renseigné (modifiable dans l'onglet “Details” du panneau gauche).

Dans le cas de la jauge dynamique du widget de gouvernail, le domaine du matériau est “User Interface”.

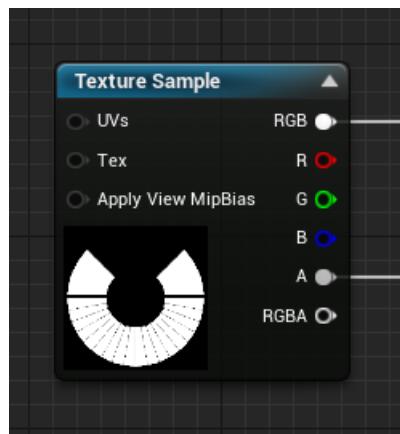


FIGURE 11 – Un noeud d'échantillon de texture.

Le développeur peut également faire intervenir des noeuds de données supplémentaires (à l'instar des variables “uniforms”), comme le noeud **TextureSample** qui permet de travailler sur une texture/image (depuis les assets du projet).

Les matériaux sont paramétrables. Le développeur peut transformer des noeuds de données en **noeuds paramètres** pour affecter le rendu du matériau depuis l'extérieur (i.e. depuis un blueprint ou une classe code).

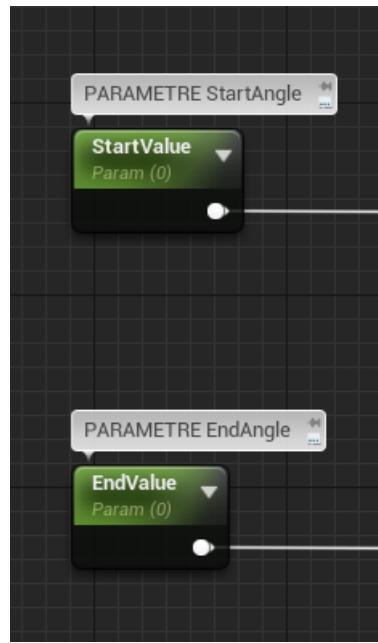


FIGURE 12 – Deux noeuds paramètres qui vont permettre de changer la valeur de la jauge pour “l’animer”.

Dans Unreal Engine, avant d’être appliqués, les shaders sont compilés une unique fois de manière asynchrone pour ne pas bloquer le moteur (en phase éditeur ou en phase de jeu/simulation).

Si l’on souhaite altérer un matériau durant la phase de jeu en modifiant ses propriétés, il est alors nécessaire de recompiler le code du shader associé (une opération coûteuse de prime abord).

Pour éviter une telle dégradation des performances, Unreal Engine propose une “surcouche” du matériau avec le **matériau instancié** (en anglais : “Instanced Material”).

Le matériau instancié permet donc d’affecter les propriétés d’un matériau depuis l’extérieur sans avoir à recompiler les shaders des matériaux.

Il reste alors les noeuds d'instructions qui sont multiples et ne seront donc pas détaillés ici.

Exemples de noeuds d'instructions :

- création de constantes : Constant (flottant simple), Constant2Vector (vecteur 2-dim), Constant3Vector (vecteur 3-dim), etc.
- déconstruction de vecteurs en flottants : BreakOutFloatXComponents (X pour la dimension)
- opérations mathématiques diverses : Add, Subtract, Multiply, Divide, Floor, Clamp, Lerp (interpolation linéaire), Blend (mélange de couleurs simples ou de textures)
- mapping de coordonnées : VectorToRadialValue
- etc.

Exemples de noeuds de données convertibles en noeuds paramètres :

- Constant, Constant2Vector, Constant3Vector, Constant4Vector
- Texture Sample, Texture Object
- et bien d'autres...

Il existe également un noeud permettant d'injecter du code HLSL⁸ pour définir des noeuds d'instructions personnalisés (noeud “Custom” dans la Palette).

Un exemple d'utilisation est consultable avec le noeud personnalisé **CustomRotation** du matériau **M_RudderDynamicRange** (jauge dynamique du widget gouvernail) du projet Navmer3D.

Tous les noeuds sont retrouvables depuis la collection fournie par la Palette (à l'exception du noeud CustomRotation p.ex., qui est personnalisé).

8. Langage de programmation shader géré par Microsoft : <https://learn.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl>

2.4 Objet HUD

L'objet HUD d'Unreal Engine peut être utilisé pour dessiner et afficher des éléments graphiques superposés à l'écran pour le joueur (le principe de l'ATH).

De base, c'est un objet historique à Unreal Engine qui propose simplement des méthodes pour tracer des primitives géométriques sur un canvas.

Il est intéressant de noter que chaque *Player Controller* dispose d'une **référence** vers un objet HUD.

L'objet **Player Controller** permet de faire l'interface entre le joueur (le navigant) et un **objet contrôlé**⁹ (p.ex. un bateau). Les valeurs des inputs effectués par le joueur sont généralement fournies via le Player Controller utilisé, ce qui permet de traduire les valeurs perçues en mouvement ou action par la suite. Il est d'ailleurs possible d'utiliser plusieurs Player Controllers (cas du jeu multijoueur local).

Dans le cas de Navmer3D, un seul et unique Player Controller est utilisé pour tous les navires.

NOTE :

Si le plugin NavmerInput est utilisé, les valeurs des inputs effectués depuis les consoles de navigation sont également fournies via le Player Controller.

Navmer3D possède son propre objet HUD : **NMHUD**.

Comme l'interface de navigation utilise des widgets, les fonctionnalités de l'objet HUD n'ont pas grand intérêt.

Cependant, dans Navmer3D, l'objet NMHUD est utilisé comme un “manager” pour l'ensemble des interfaces qui proposent un service ATH (p.ex. le menu du plugin NavmerInput, ou l'interface navigant). De ce fait, il est possible d'avoir un accès direct sur l'ensemble des widgets possédés par l'HUD.

De plus, comme l'objet HUD est attaché au Player Controller principal (qui est associé aux différents navires), il fait l'intermédiaire entre celui-ci et les widgets.

9. Dans Unreal Engine, on utilise souvent le terme “possession” pour parler d'un objet contrôlé (objet **Pawn**) par un Player Controller.

2.5 Layouts

Parce qu'il existe différents types de manœuvres ainsi que différents dispositifs de commandes, il est nécessaire d'étendre la notion d'interface de navigation.

Pour cela, Navmer3D permet d'utiliser différentes dispositions de l'interface selon le type de manœuvres utilisé.

 **NOTE :**

Les principaux sont bateaux à hélices et bateaux à pods azimutaux.

Ces dispositions sont appelées **layouts**.

 **NOTE :**

Le seul **layout** existant pour le moment dans le projet est celui des bateaux à propulsions par hélices.

Le layout est composé de différents widgets nécessaires au type de manœuvre souhaité. Il communique également avec ses widgets en leur passant les différentes structures d'états (i.e. états de simulation et états de commandes).

Si plusieurs layouts sont créés, il est possible de changer le layout actif en interrogeant le HUD du Player Controller principal (via un appel de méthode).

2.6 Minimap

La minimap de Navmer3D est réalisé à partir d'un matériau et d'une texture qui correspond à une carte ENC. De cette manière, différents effets visuels vont pouvoir être appliqués sur la minimap comme :

- Translater (i.e. déplacer) la texture en horizontale et en verticale suivant la position géographique du navire (obtenue via géoréférencement de la scène dans laquelle se trouve le bateau)
- Masquer les parties non visibles de la texture, en fonction de la position du bateau sur la minimap et de l'échelle d'affichage

Le matériau, via un matériau instancié, prend en paramètres les coordonnées **raster**¹⁰ du navire, un facteur de mise à échelle (“zoom”), ainsi qu’une texture (la carte ENC de la scène courante affichée pendant la simulation).

Concrètement, le navire est toujours affiché au milieu de la minimap : c'est la texture qui se déplace suivant sa position géographique, donnant l'impression que le bateau bouge sur la minimap.

10. Un fichier image (.png, .jpeg, .tiff, etc.) d'une carte ENC matricielle.

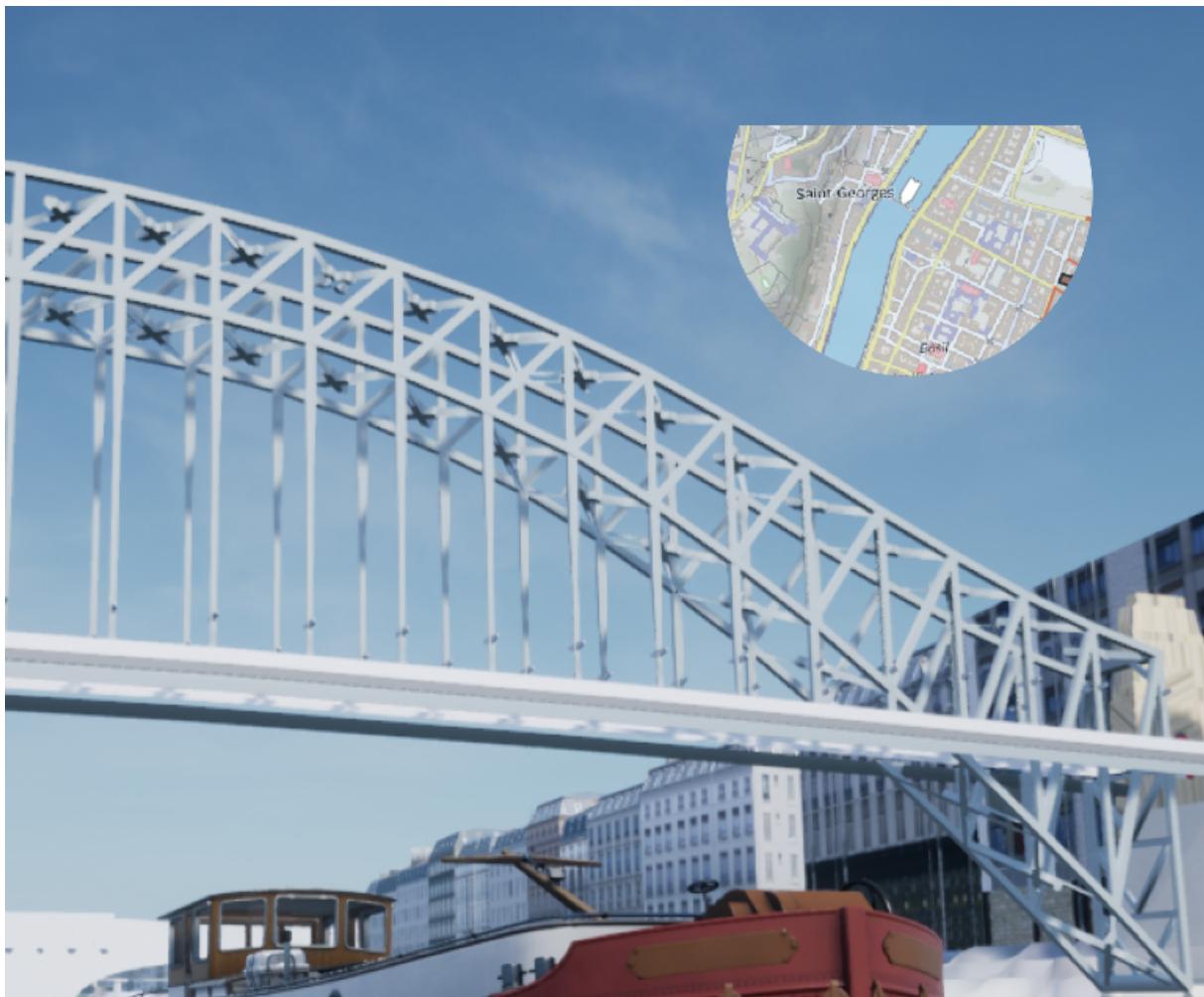


FIGURE 13 – La minimap de Navmer3D sur la scène Lyon. Le navire se trouve au bout de la carte ENC, au nord, visiblement.

 **NOTE :**

Logiquement, il est important que la texture de la carte ENC corresponde à la scène Unreal.

Même si l'utilisation du géoréférencement dans la scène permet une position fidèle du navire sur la minimap, la “fenêtre” de la région représentée avec la carte ENC peut ne pas coincider avec celle de la scène 3D.

Par exemple, dans le cas de la scène Lyon, lorsque le navire se situe précisément à l'extrême nord sur la minimap, ce n'est pas le cas du navire dans la scène 3D (il reste encore du chemin avant d'atteindre le “vide” de la scène).

Bien que ce point reste de l'ordre de la “perfection”, notamment si le point de départ du navire est décidé et son trajet est délimité dans la scène 3D, je tenais à y faire part.

2.6.1 Mapping des coordonnées

Afin d'obtenir les coordonnées *raster* dont le matériau aurait besoin pour positionner le navire contrôlé sur la minimap, il est nécessaire d'appliquer une série de conversion de coordonnées au préalable :

1. Mapper les coordonnées “world” du navire (i.e. dans la scène) dans un référentiel en coordonnées projetées (“Projected CRS”) relatives à un point de repère géographique dans la scène
2. Mapper les coordonnées projetées relatives du navire en coordonnées raster

Dans le cas **1**, il est nécessaire de géoréférencer la scène (ex. “Lyon”), de manière à ce quelle puisse représenter localement un lieu sur la surface de la Terre.

L'acteur **GeoReferencingSystem** (via un plugin natif d'Unreal Engine) permet cela lorsqu'il est présent dans une scène. À partir du point de la surface de la Terre qu'il géoréférence, on peut transformer les coordonnées “world” d'un objet en coordonnées projetées ou géographiques.

Le fait d'utiliser un référentiel en coordonnées projetées permet d'obtenir la position géographique du navire tout en conservant un système cartésien. De cette manière, il sera plus aisément de parvenir à l'étape **2**.

 **NOTE :**

Pour en savoir plus sur l'acteur GeoReferencingSystem, il est recommandé de consulter la notice opératoire de l'ancien projet.

Dans le cas **2**, il faut parser un fichier **World file**¹¹ associé à une carte ENC qui contient l'ensemble des données permettant la conversion vers le raster. De cette manière, on peut transformer les coordonnées projetées relatives du navire en coordonnées raster pour une carte ENC associée.

 **NOTE :**

Certains visualiseurs de carte ENC proposent la génération de ce type de fichier.

L'ultime étape est de transmettre, via des paramètres, les coordonnées raster du navire au matériau instancié chargé d'afficher la minimap. Le matériau s'occupe alors de convertir les coordonnées raster en coordonnées texture exploitables (i.e. dans la plage normalisée [0, 1]).

11. Fichier World file : https://en.wikipedia.org/wiki/World_file

3 Installer le projet

NOTE :

Dans le cas où cela n'a pas été notifié sur un autre document : si une nouvelle installation d'Unreal Engine 4.27 est envisagée, il existait au départ des dépendances du projet dans le dossier *Epic Games/UE_4.27/Engine/Plugins* (contenu de la version du moteur installée).

Ces dépendances sont à priori dues à l'installation de plugins dans le moteur et non dans le projet.

Il faut donc au préalable copier le contenu de *Epic Games/UE_4.27/Engine/Plugins* situé sur le poste stage dans le dossier du même nom de la nouvelle installation.

Autrement, il est nécessaire de télécharger à nouveau les plugins et de réparer les dépendances une à une dans l'éditeur.

La dernière version du projet Navmer3D (2022) contient l'implémentation de l'interface de navigation.

La maintenance du code se fait avec l'IDE Visual Studio 2022.

Lors de l'installation de Visual Studio 2022, il est nécessaire de l'associer à Unreal Engine. Une documentation est disponible via le lien suivant pour y parvenir : <https://docs.unrealengine.com/4.26/en-US/ProductionPipelines/DevelopmentSetup/VisualStudioSetup/>

Vous pouvez récupérer le dossier du projet Navmer3D depuis le poste stage, ou bien depuis l'espace Box.

Le repository git du projet Navmer3D est disponible à l'adresse <https://gitlab.cerema.fr/navmer3d/navmer3d>. La branche actuelle de développement du plugin est "stage2022".

Le contrôle de version ne s'applique qu'au code et aux blueprints.

3 INSTALLER LE PROJET

Pour l'installation d'une nouvelle configuration (i.e. pour un nouveau poste) :

1. Installer Unreal Engine dans sa version 4.27 depuis le launcher d'Epic Games (un compte est nécessaire)
2. Installer et configurer Visual Studio 2022 pour Unreal Engine en suivant la doc spécifiée auparavant.
3. Copier le contenu du dossier *Epic Games/UE_4.27/Engine/Plugins* présent sur le poste stage dans celui de la nouvelle installation.
4. Dans le dossier du projet Navmer3D, ouvrez le menu contextuel depuis le fichier **Navmer3D.uproject** et régénérez la solution VS 2022 en cliquant sur “Generate Visual Studio project files”
5. Vérifier que le repo git local du projet Navmer3D est à jour sur le code et les blueprints (“git status”), et s’assurer que c’est le cas
6. Ouvrir le projet avec le launcher d’Unreal Engine, ou depuis le fichier Navmer3D.uproject pour vérifier que tout fonctionne

Autrement :

1. Dans le dossier du projet Navmer3D, ouvrez le menu contextuel depuis le fichier **Navmer3D.uproject** et régénérez la solution VS 2022 en cliquant sur “Generate Visual Studio project files”
2. Vérifier que le repo git local du projet Navmer3D est à jour sur le code et les blueprints (“git status”), et s’assurer que c’est le cas
3. Ouvrir le projet avec le launcher d’Unreal Engine, ou depuis le fichier Navmer3D.uproject pour vérifier que tout fonctionne

Si les étapes ont bien été suivies, il ne devrait pas y avoir de problèmes à l’ouverture du projet Unreal (pas de messages d’erreurs ou de warnings sur les dépendances) ou de la solution VS 2022.

4 Vue Contenu Navmer3D

Cette partie permet d'expliquer l'ensemble du nouveau contenu de Navmer3D.

Le projet dispose de nombreuses ressources graphiques (textures, modèles de navires et de cartes, etc.), ainsi qu'un ensemble de blueprints et de classes code.

De manière à mieux répertorier son contenu, l'arborescence de Navmer3D a été mise à jour avec des sous-dossiers où y sont rangés le contenu nouveau. De plus, des préfixes ont été rajoutés à certains noms (blueprints et classes codes) pour différencier d'une part le contenu Navmer3D et celui du moteur, et d'autre part le contenu de Navmer3D et celui de l'ancien projet (dernière appellation : *PuzzlePlateform*).

Le contenu de l'ancien projet conserve cependant son état d'origine dans l'arborescence de manière à éventuellement s'accorder avec la notice technique fournie à l'époque.

4.1 Ressources et Blueprints

Depuis l'explorateur Windows, l'entièreté du contenu de Navmer3D est disponible dans le dossier *Content*, localisé directement sous la racine du projet.

Depuis l'éditeur Unreal, le contenu de Navmer3D est réparti dans les dossiers *NavmerContent* et *NavmerData* (via le *Content Browser*, sous la rubrique *Content*).

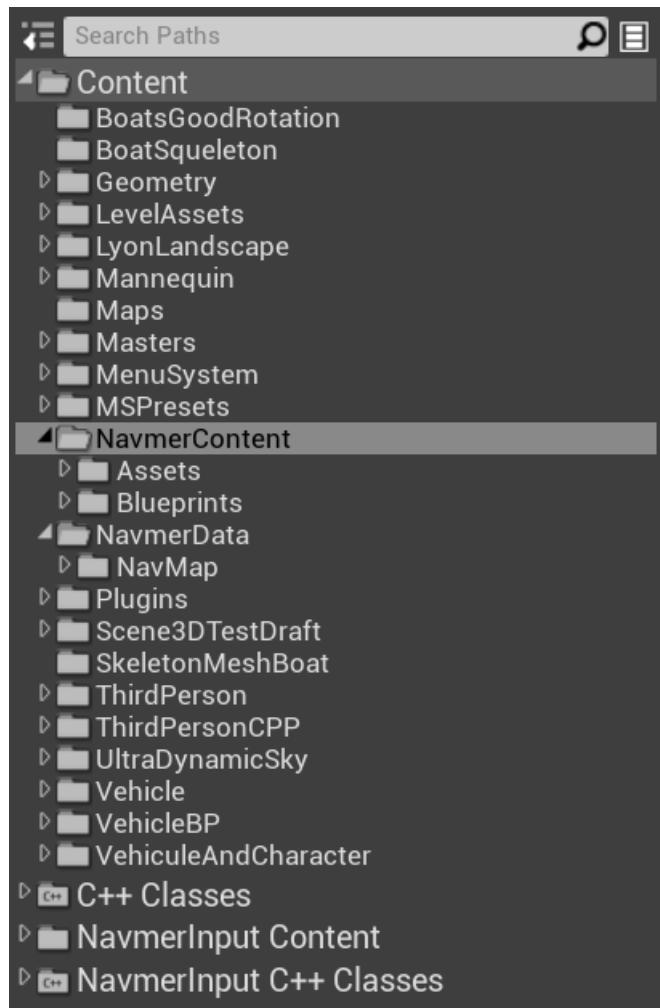


FIGURE 14 – Arborescence de fichiers de Navmer3D dans l'éditeur.

Le sous-dossier *NavmerContent/Assets/Widgets* regroupe l'ensemble des ressources graphiques du projet utilisées pour l'interface navigant, avec :

- *NavmerContent/Assets/Widgets/Materials* le sous-dossier des matériaux,
- et *NavmerContent/Assets/Widgets/Textures* le sous-dossier des textures.

Le sous-dossier *NavmerContent/Blueprints* regroupe l'ensemble des blueprints du projet utilisés pour l'interface navigant, avec :

- *NavmerContent/Blueprints/HUD/BP_HUD* le blueprint du HUD,
- *NavmerContent/Blueprints/Widgets/BoatUI/* le sous-dossier comprenant la collection de widgets personnalisés pour l'interface navigant,
- et *NavmerContent/Blueprints/Widgets/BoatUI/Layouts* le sous-dossier des layouts.

Le dossier *NavmerData* contient les données de transformation géographique utilisées pour le widget minimap de l'interface navigant. Ces fichiers (.json et .tfw) ne sont cependant pas accessibles depuis la fenêtre Content Browser de l'éditeur Unreal. Il faut utiliser l'explorateur Windows pour les découvrir.

4.2 Code source

Depuis la solution VS 2022 du projet, le code source est disponible sous *Games/Navmer3D/Source/Navmer3D*.

Autrement, le code est accessible dans le dossier *Source/Navmer3D* depuis la racine du projet via l'explorateur Windows. Depuis l'éditeur d'Unreal Engine, les classes code sont visibles depuis la rubrique "C++ Classes" via le Content Browser.

Ouvrir les classes code depuis l'éditeur d'Unreal Engine permet de lancer Visual Studio 2022, car le projet Navmer3D est lié à l'IDE.

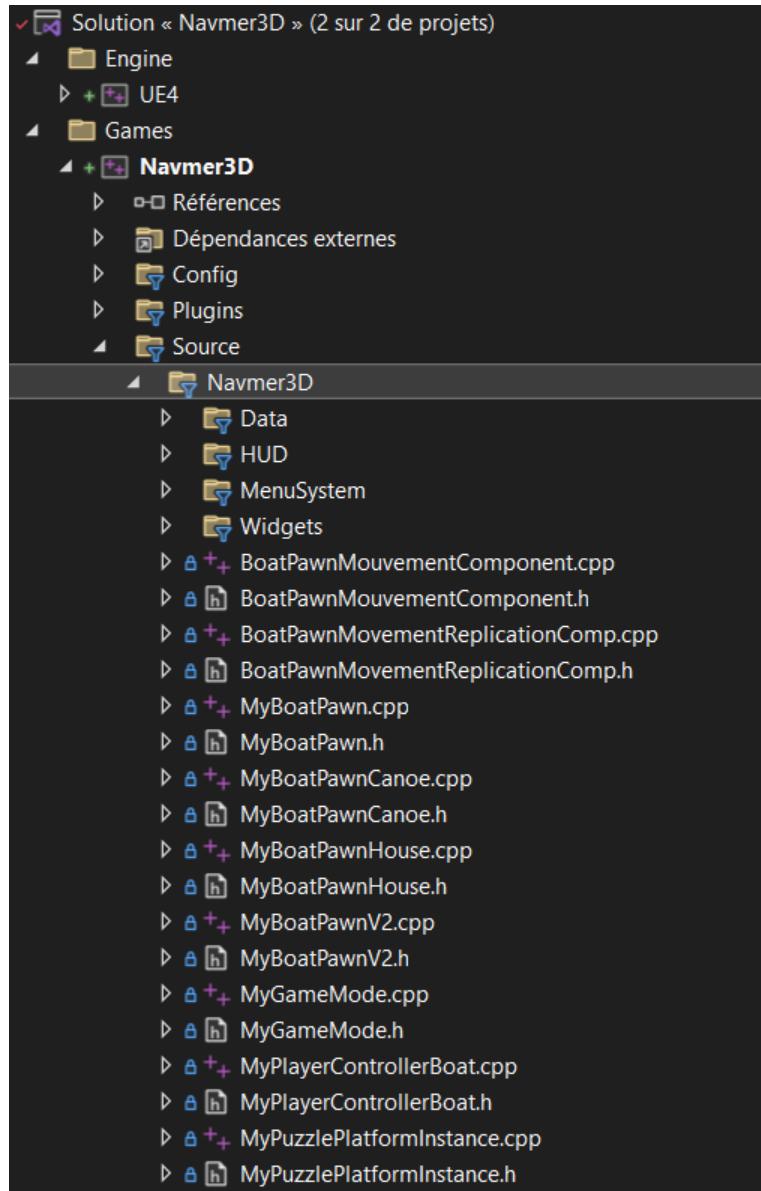


FIGURE 15 – Solution du projet Navmer3D.

On retrouve depuis la solution VS 2022 l'ensemble des fichiers sources et des headers du précédent stage. L'ensemble des nouveaux fichiers sources et headers portent le préfixe “NM_”.

Trois nouveaux dossiers sont à noter : **Data**, **HUD** et **Widgets**. Ces dossiers contiennent l'ensemble des classes utiles pour l'interface navigant.

Voici donc la liste des dossiers et fichiers de code à prendre en compte pour la maintenance de l'interface navigant :

- *Data/NMStateData.h* : Déclaration des structures d'états
- *HUD/NMHUD.h* et *HUD/NMHUD.cpp* : Classe NMHUD
- *Widgets/* : Fichiers des classes widgets
- *NMMiscHelpers.h* et *NMMiscHelpers.cpp* : Classe statique regroupant un ensemble de fonctions utilitaires
- *MyBoatPawnV2.h* et *MyBoatPawnV2.cpp* : Classe comportementale des navires
- *MyPlayerControllerBoat.h* et *MyPlayerControllerBoat.cpp* : Classe du Player Controller de Navmer3D
- *MyPuzzlePlatformInstance.h* et *MyPuzzlePlatformInstance.cpp* : Classe du GameInstance de Navmer3D
- *Navmer3D.Build.cs* : Fichier de configuration pour l'import de modules UE et plugins

4.3 Convention de nommage

L'ensemble des ressources (graphiques et autres) et du code source de Navmer3D suit une certaine convention de nommage.

Par exemple, comme dit précédemment, tous les nouveaux fichiers (par rapport à l'ancien projet) ont le préfixe “NM_”.

Tout d'abord, les ressources (“assets” en anglais), suivent la convention suivante :

- Notation “PascalCase” ou “Pascal_Case” (séparation underscore)
- Material et Instanced Material : respectivement M_XXX et MI_XXX
- Textures : pas de préfixe
- Blueprint : BP_XXX, où XXX est identique au nom de la classe code parente
- Blueprint Widget : WBP_XXX

Enfin, le code source essaye de respecter au mieux la convention suivante :

- Notation “PascalCase” (nom de fichier, nom de classe, nom de méthode, identifiant, etc.)
- Fichier headers et sources préfixés par NM
- Préfixe XNM devant le nom de classe, avec X la nature de la classe (U, F, A, etc.)

Cette convention de nommage reste en accord avec une approche usuelle ¹².

Outre la convention de nommage additionnelle employée, Unreal Engine standardise la spécification de la nature des classes codes avec le nom prefixé d'une lettre ¹³.

12. Documentation sur le nommage usuel des assets : <https://github.com/Allar/ue5-style-guide>

13. Convention de nommage du code Unreal Engine : <https://docs.unrealengine.com/4.27/en-US/ProductionPipelines/DevelopmentSetup/CodingStandard/>

5 Maintenance Ressources

5.1 Ajouter une texture

Navmer3D utilise une collection de **textures** pour l'affichage des widgets de l'interface de navigation.

Pour ajouter une texture au projet :

- Glisser la texture (au format .png, .jpeg, .tiff, etc.) dans le dossier *Content/NavmerContent/Assets/Widgets/Textures* du projet via l'explorateur Windows

Unreal Engine détecte automatiquement l'ajout d'une ressource de type image et vous propose alors d'importer la ressource dans le projet sous forme d'objet de type **Texture** (au format .uasset).

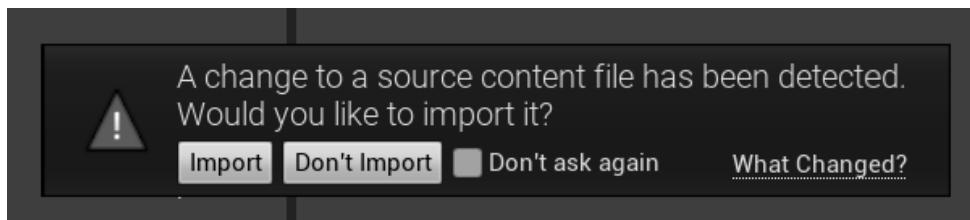


FIGURE 16 – Popup de l'éditeur après l'ajout d'une ressource dans l'arborescence du projet.

- Sélectionnez alors “Import”

Vous pouvez à présent voir votre texture importée depuis le Content Browser à l'endroit où vous l'avez déposée.

5.2 Textures Navmer3D

Plusieurs points sont à noter concernant les textures de l'interface de navigation :

1. Les textures visent à avoir des dimensions appropriées pour une définition d'image de 1920x1080 (définition commune), de base.

NOTE :

Par erreur de design au départ, il se peut que certaines textures des widgets de Navmer3D soient trop grandes pour du 1920x1080, auquel cas leurs dimensions sont redéfinies dans Unreal Engine. Si une texture est suffisamment grande au départ, diminuer légèrement sa taille n'aura pas d'impact conséquent sur les détails de la texture.

Heureusement, il n'est pas nécessaire de concevoir une texture pour chaque définition d'image souhaitée : Unreal Engine fait une mise à échelle automatique lorsque le rendu du processus passe dans une autre définition.

Cependant, il est nécessaire que votre UI soit correctement conçu avec le designer, de manière à ce que les widgets utilisent des images (c'est le cas de tous les widgets personnalisés de Navmer3D) ne se chevauchent pas lorsqu'il y a un changement de définition ou de rapport de forme (aspect ratio) (cf. [Designer un layout](#)).

2. Les textures sont blanches (ou en nuances de gris).

Ce point 2 permet de personnaliser la couleur des textures en effectuant des mélanges de couleur dans un matériau Unreal Engine (en accord avec le système de couleur additif RVB).

3. Les textures sont réutilisées.

Certaines textures sont de préférence réutilisées lorsque c'est possible, notamment lorsque les changements entre deux widgets ne sont que de l'ordre de la couleur ou de l'orientation (p.ex. les indicateurs de vent et de courant marin).

4. Les widgets d'états de commandes (cf. [Nouveaux widgets](#)) possèdent chacun une base de trois textures : une texture de fond, une texture blanche de "plage dynamique", et une texture de curseur (slider ou pointeur) placée au-dessus des autres.

La texture blanche de plage dynamique est travaillée dans un matériau pour réaliser l'effet de "plage dynamique" (un nom pas forcément approprié) qui consiste à représenter visuellement la différence de valeurs entre la consigne de commande et la valeur réelle affectée par la montée en régime du moteur sous-jacent.

Deux exceptions existent pour les widgets d'états de commandes : les widgets des manettes de propulsion. Ces widgets sont affichés de deux façons :

- Une barre “couchée” (en horizontale) : 2 textures (fond et “plage dynamique”)
- Une barre “en colonne” (en verticale) : 2 textures (fond et “plage dynamique”)
- Une seule texture de curseur pour les deux dispositions

5. Les textures de fond et de plage dynamique des widgets d'états de commandes sont de même dimensions.

Le point **5** permet de garder une cohérence entre les deux textures pour faciliter le travail dans Unreal Engine.

Si la texture de fond et la texture de plage dynamique ne sont pas de mêmes dimensions, il peut être pénible d'ajuster la taille de l'une pour correspondre à l'autre, et sans impacter les détails. Parce que la texture de plage dynamique agit visuellement comme une sorte de jauge qui progresse dans la texture de fond, il se peut également que celle-ci dépasse de l'espace qui lui est réservé sur la texture de fond lorsque les dimensions ne correspondent pas, provoquant un effet non désiré.

En prenant en compte au préalable ces différents problèmes de “format” : les textures de fond et de plage dynamique ont été dessinées au départ dans un même canevas, et exportées par la suite en plusieurs fichiers de mêmes dimensions.

5.3 Materials et Instanced Materials

À l'exception de la minimap, les widgets d'états de simulation (cf. [Nouveaux widgets](#)) ne sont pas reliés à un matériau, car cela n'était pas nécessaire au vu des effets visuels souhaités.

Par exemple, les indicateurs de vent et de courant marin donnent simplement une orientation (ils tournent simplement) et sont différenciés par une couleur propre : on peut se passer d'un matériau pour ça en utilisant les propriétés héritées des widgets d'Unreal Engine, afin de libérer du stockage et de diminuer le temps de compilation des shaders.

Ce n'est cependant pas le cas pour les widgets d'états de commandes ou le widget de la minimap, qui sont tous associés à un matériau car ils nécessitent des effets visuels plus complexes.

NOTE :

Les matériaux créés pour l'animation des widgets d'états de commandes agissent uniquement sur la composante d'opacité des textures impliquées. Les couleurs des textures sont alors affectées dans les blueprints des widgets (donc en dehors des matériaux et matériaux instanciés), ce qui est possible puisque les textures sont blanches de base.

5.3.1 Créer un Material et un Instanced Material

NOTE :

Il est recommandé de consulter la section [Utilisation des matériaux dans Unreal Engine](#) au préalable pour aborder ce qui suit.

Pour créer un matériau dans Navmer3D (reproductible dans n'importe quel projet) :

1. Depuis l'éditeur d'UE, rendez-vous dans le sous-dossier *NavmerContent/Assets/Widgets/Materials* à l'aide de la fenêtre Content Browser.

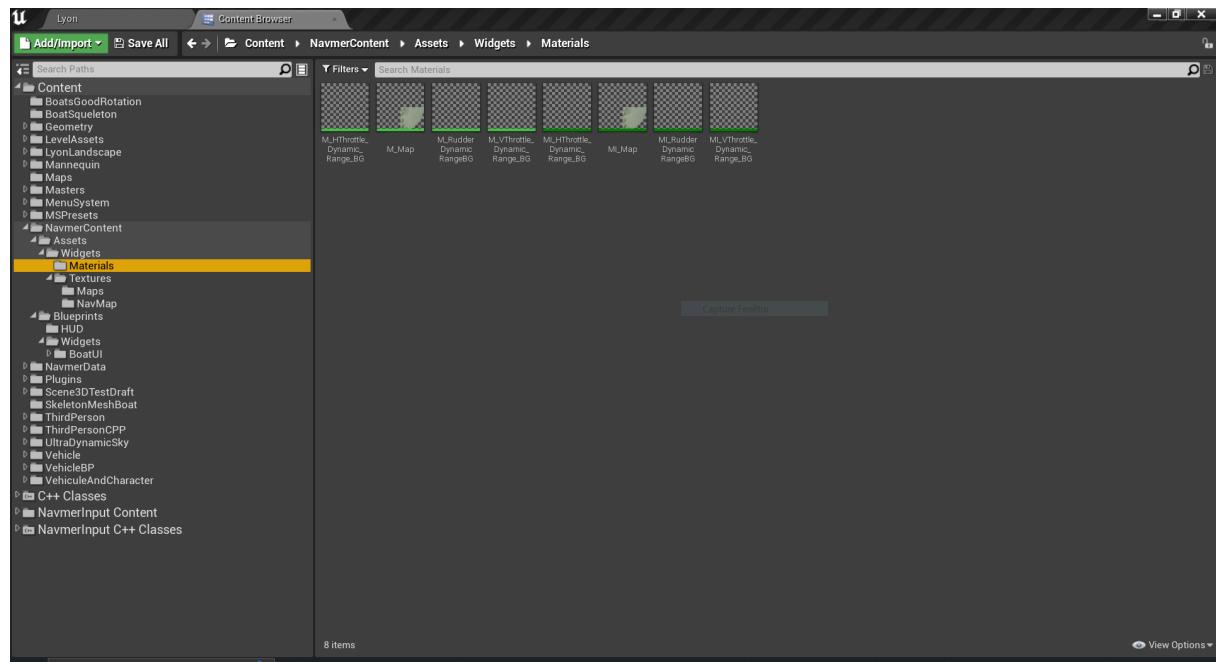


FIGURE 17 – Sous-dossier Materials pour les widgets.

2. Ouvrez le menu contextuel dans le panneau de visualisation du contenu (clic droit souris), et sélectionnez “Material”.

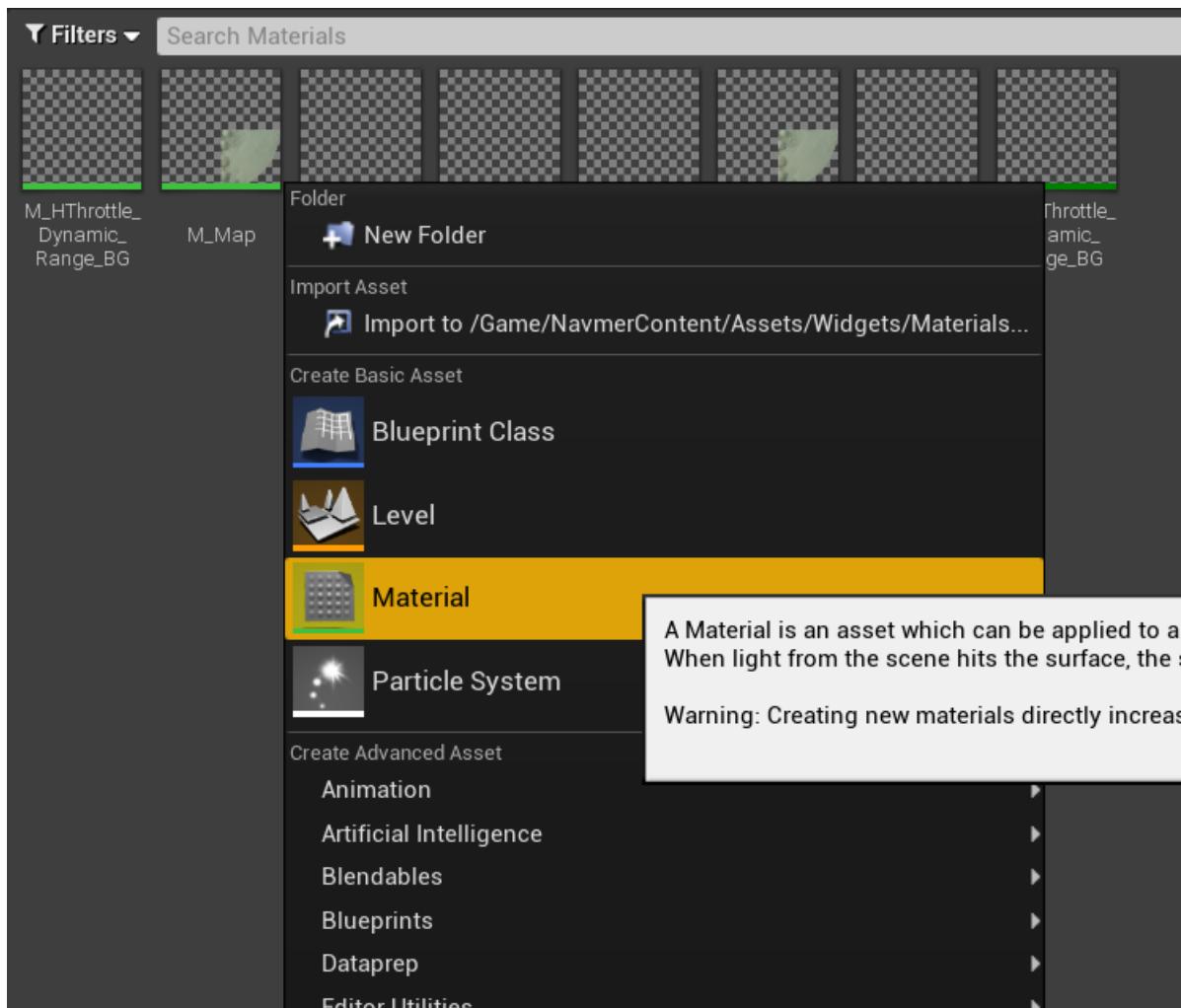


FIGURE 18 – Sélectionner “Material” permet de créer un matériau dans le dossier actuel.

Votre matériau apparaît alors dans le Content Browser.

Il nous faut maintenant passer le domaine du matériau en “User Interface”, car il sera appliqué à un widget (et non à une surface 3D). Pour cela :

3. Ouvrez le matériau pour rentrer dans l’éditeur de matériau (“Edit” depuis le menu contextuel).
4. Sous le panneau gauche des propriétés du matériau, changez dans la rubrique “Material” le “Material Domain” en “User Interface”.

5 MAINTENANCE RESSOURCES

5.3 Materials et Instanced Materials

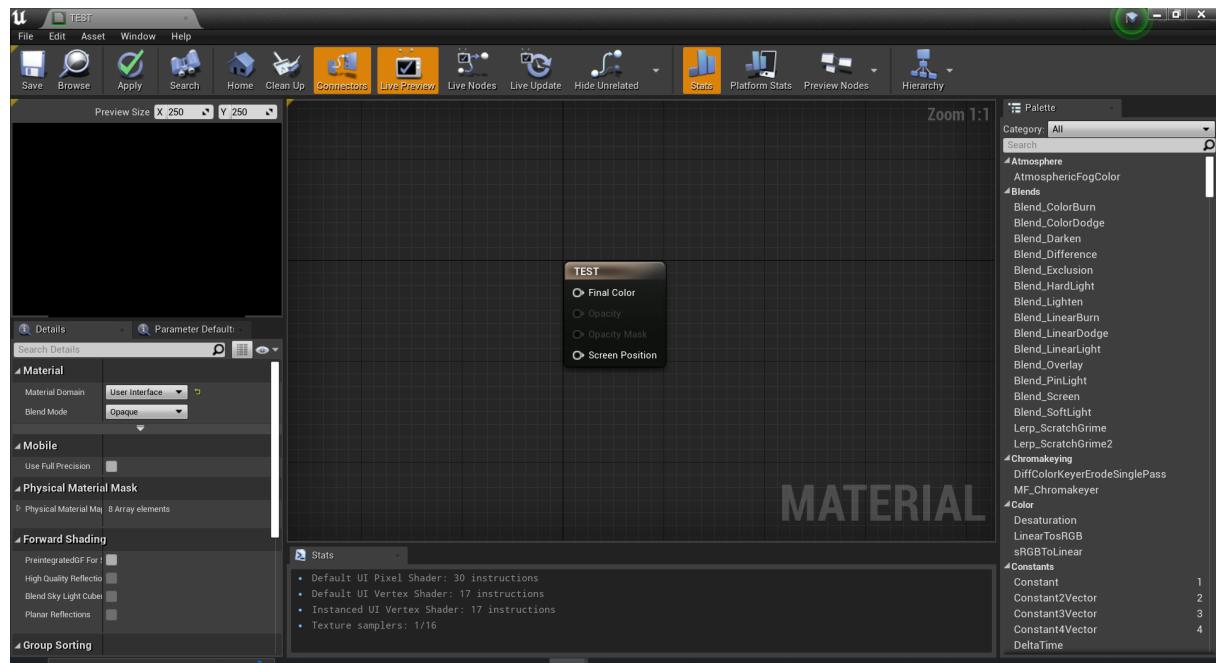


FIGURE 19 – Le matériau passe dans le domaine “User Interface”. Le noeud de sortie change également.

On notera aussi le “Blend Mode” qui permet le mélange de couleurs en ajoutant de l’opacité à la sortie. Le Blend Mode est très utile lorsque le widget doit s’afficher devant un fond en mouvement (p.ex. toute l’interface de navigation qui se comporte comme un ATH), ou lorsque l’on doit réaliser un masquage quelconque (p.ex. masquer les bords de la minimap, ou masquer les extrémités de la jauge du gouvernail).

On pourrait aussi rajouter un noeud paramètre, en supposant que l’on a un noeud de donnée. Pour ce faire :

5. Toujours depuis l’éditeur de matériau, choisissez un noeud **Constant3Vector**¹⁴ depuis la Palette et glissez-le dans le viewport (OU, choisissez un noeud Constant3Vector depuis le menu contextuel du viewport).
6. Rattachez le noeud créé (nommé dans le viewport “0,0,0”) au noeud de sortie sur le pin “Final Color”.

14. Documentation des différents types de données de l’éditeur de matériau : <https://docs.unrealengine.com/5.1/en-US/material-data-types-in-unreal-engine/>

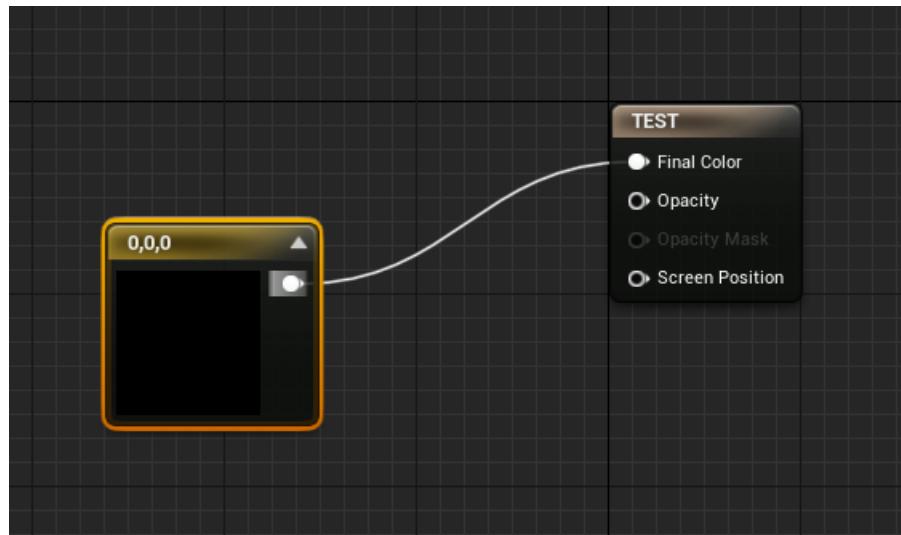


FIGURE 20 – Le noeud de donnée Constant3Vector est rattaché à la sortie du matériau, ce qui permet de changer la couleur finale du matériau.

Si vous sélectionnez le noeud, vous pourrez modifier ses valeurs depuis le panneau gauche dans l'onglet “Details”.

Maintenant, voici une manière de créer un noeud paramètre (il y en a plusieurs) :

7. Depuis le menu contextuel du noeud créé (clic droit souris sur le noeud), sélectionnez “Convert to Parameter”.

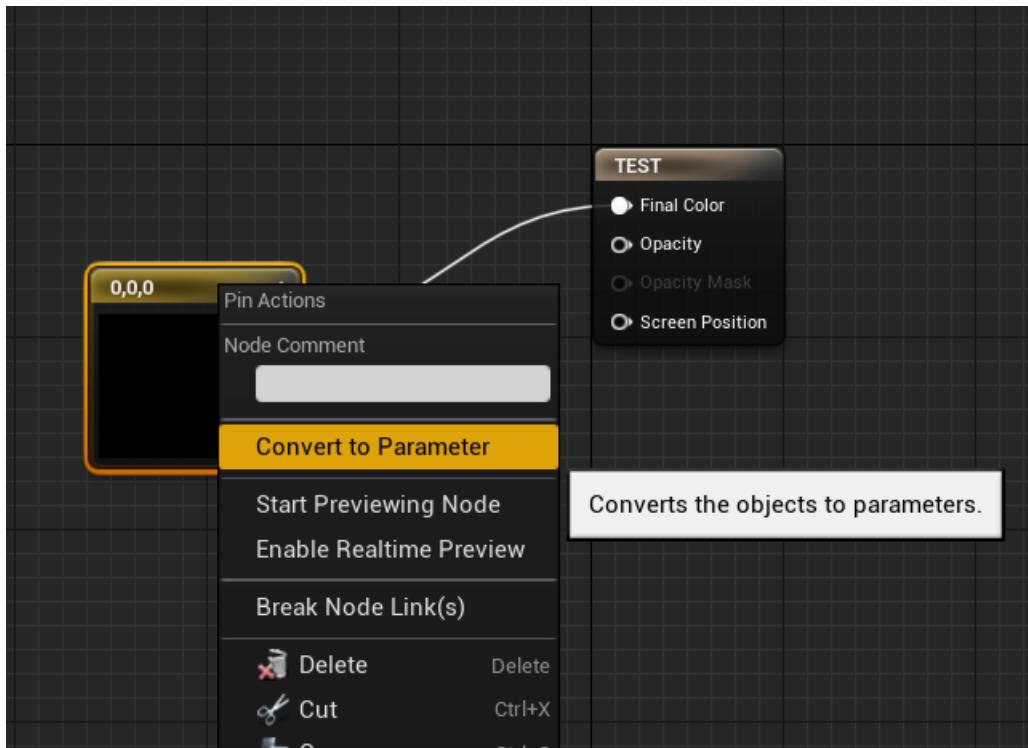


FIGURE 21 – Sélectionner “Convert to Parameter” transforme le noeud de donnée en noeud paramètre.

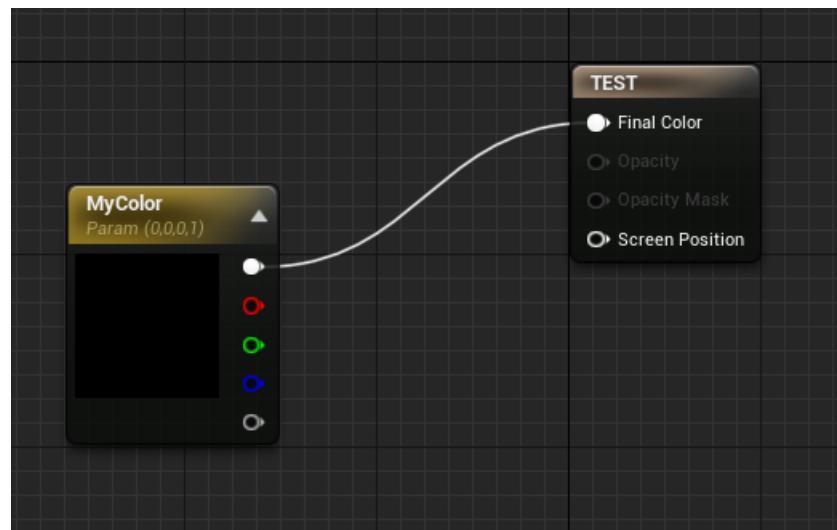


FIGURE 22 – Le noeud Constant3Vector devient le noeud paramètre “MyColor” (renommé) de type Constant4Vector.

Si le noeud créé est convertie en noeud **Constant4Vector**, c'est parce qu'il n'existe pas de noeud paramètre pour les dimensions en dessous.

N'oubliez pas d'appliquer les changements faits au matériau en cliquant sur “Save” dans la barre

principale de l'éditeur.

Enfin, pour créer un matériau instancié dans Navmer3D (reproductible dans n'importe quel projet) :

8. Depuis un matériau via le Content Browser, ouvrez le menu contextuel sur le matériau, et sélectionnez “Create Material Instance”.

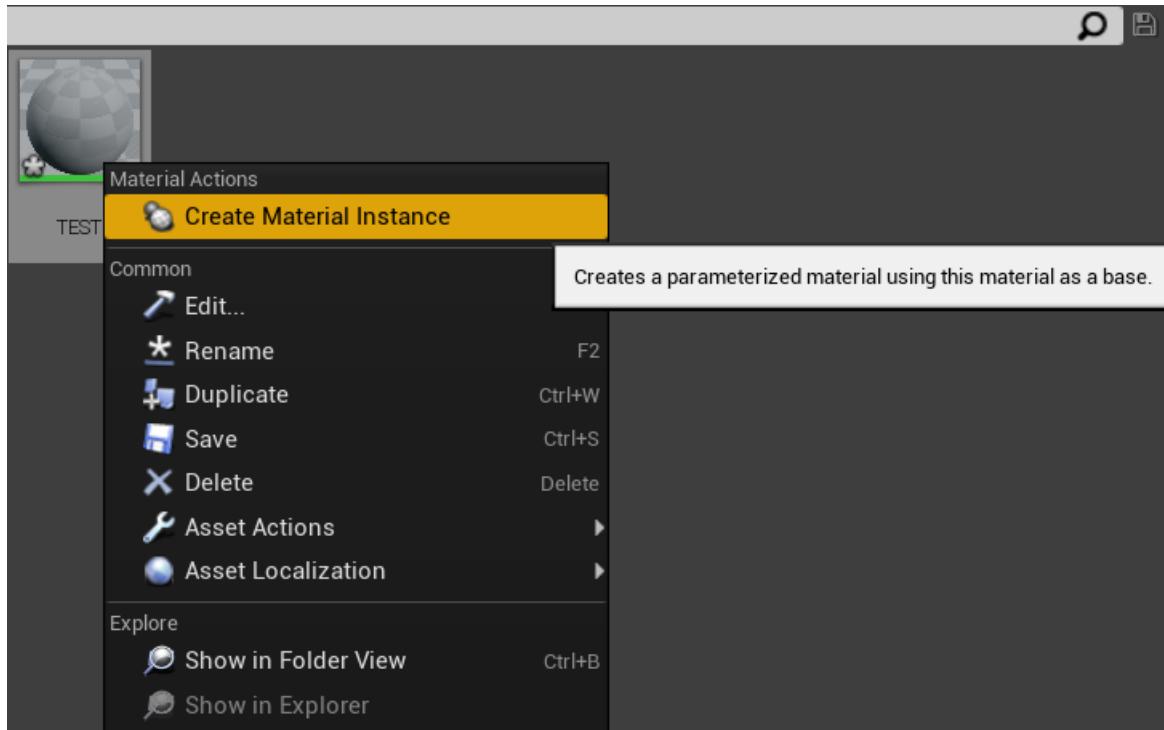


FIGURE 23 – Sélectionner “Create Material Instance” permet de créer un matériau instancié au même endroit que le matériau associé.

Votre matériau instancié apparaît alors dans le Content Browser, et est associé à son matériau.

Si vous ouvrez votre matériau instancié, vous pourrez visualiser le matériau et modifier certaines de ses propriétés sans qu'il n'en soit affecté (c'est le matériau instancié qui est modifié à la place).

Si vous avez passé le domaine du matériau en “User Interface”, vous devriez voir un fond noir.

Si vous n'avez pas changé les valeurs du noeud Constant3Vector ou MyColor (noeud paramètre), c'est normal.

Autrement, cela signifie que le pin “Final Color” de la sortie du matériau n'est relié à aucune couleur ou texture.

5 MAINTENANCE RESSOURCES

5.3 Materials et Instanced Materials

Si vous avez bien créé un noeud paramètre et que vous l'avez relié à la sortie du matériau (directement ou par l'intermédiaire d'autres noeuds), vous devriez voir le paramètre en question dans le panneau droite, sous l'onglet “Details” et sous la rubrique “Parameter Groups”.

Le paramètre est également modifiable depuis la fenêtre du matériau instancié.

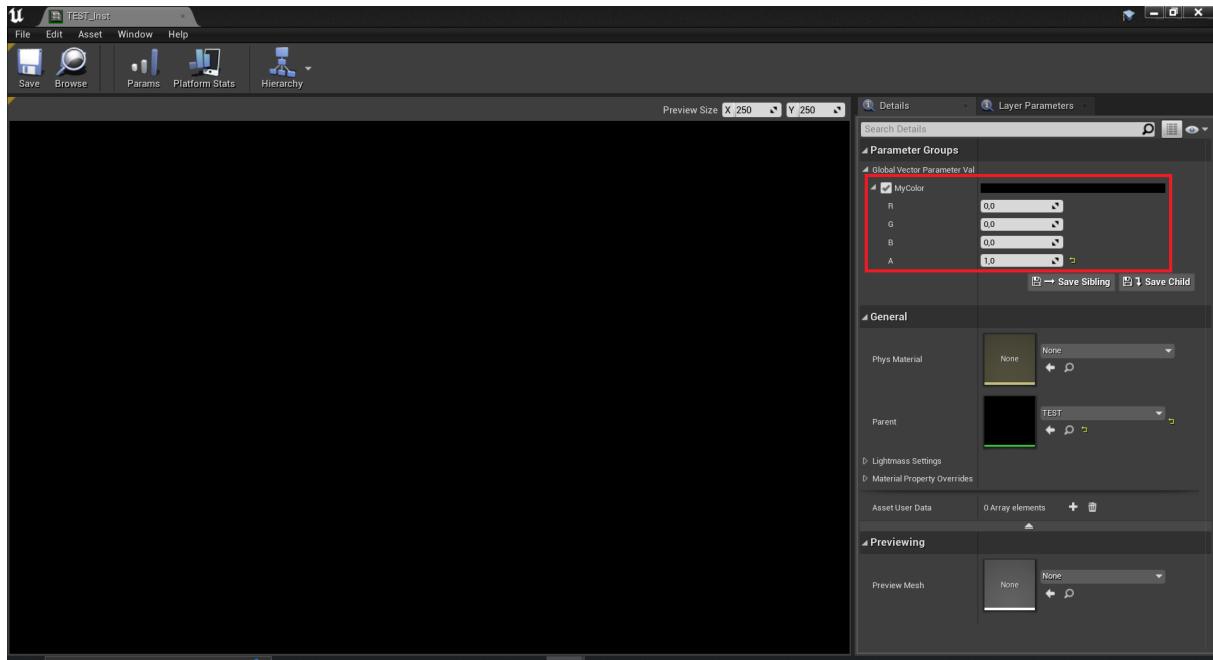


FIGURE 24 – Fenêtre du matériau instancié. Le paramètre MyColor de type **Vector4**, en rouge, a été ajouté.

L'intérêt d'avoir un noeud paramètre dans un matériau est qu'il soit accessible et modifiable depuis un blueprint ou une classe code, en accédant au matériau instancié.

Le matériau et son matériau instancié ainsi créés, vous pouvez maintenant vous référer aux [Sources utiles](#) et aux différents tutoriels sur internet pour apprendre à créer divers effets visuels pour les widgets au travers de l'éditeur de matériau.

6 Création de layout

Cette partie instructive, à la manière d'un tutoriel, permet d'expliquer comment réaliser un layout dans Navmer3D au travers de l'outil UMG d'Unreal Engine.

Il y sera d'abord expliqué comment designer un widget qui compose un layout en reprenant comme exemple un widget personnalisé du projet Navmer3D. Enfin, il y sera présenté comment designer un layout en assemblant différents widgets personnalisés.

NOTE :

Il est recommandé pour cette partie de consulter au préalable les sections [Outil UMG](#) et [Widget Blueprint](#) qui introduisent l'outil UMG et l'onglet Designer d'un widget blueprint.

Le déroulement d'instructions qui suit est basé sur ma propre façon d'utiliser le Designer. Ayant appris sur le tas à utiliser le Designer uniquement pour faire les layouts de Navmer3D, la manière générale de faire n'est sûrement pas la meilleure.

De plus, l'entièreté de la collection de widgets proposée par Unreal Engine ne sera pas explorée.

Pour s'améliorer, le lecteur est donc invité à "jouer" un peu plus avec les différents widgets disponibles dans le moteur, ainsi qu'à consulter diverses ressources et tutoriels sur internet.

L'outil de création UMG reste assez simple et est finalement vite exploré. Le lecteur peut également se renseigner sur le framework d'UI disponible dans le moteur appelé "Slate" pour aller plus loin dans la personnalisation : il n'est cependant pas utilisé pour l'interface de navigation.

6.1 Designer un widget pour layout

6.1.1 Création

Nous allons refaire ici le widget du gouvernail **WBP_Rudder**, de manière à comprendre comment fonctionne l'outil UMG et le Designer.

Tout d'abord, il faut créer le widget blueprint. Depuis Navmer3D via le Content Browser de l'éditeur d'UE :

1. Rendez-vous dans le sous-dossier *NavmerContent/Blueprints/Widgets/BoatUI*
2. Depuis le listing du contenu, ouvrez le menu contextuel et sélectionnez “Blueprint Class” sous “Create Basic Asset”

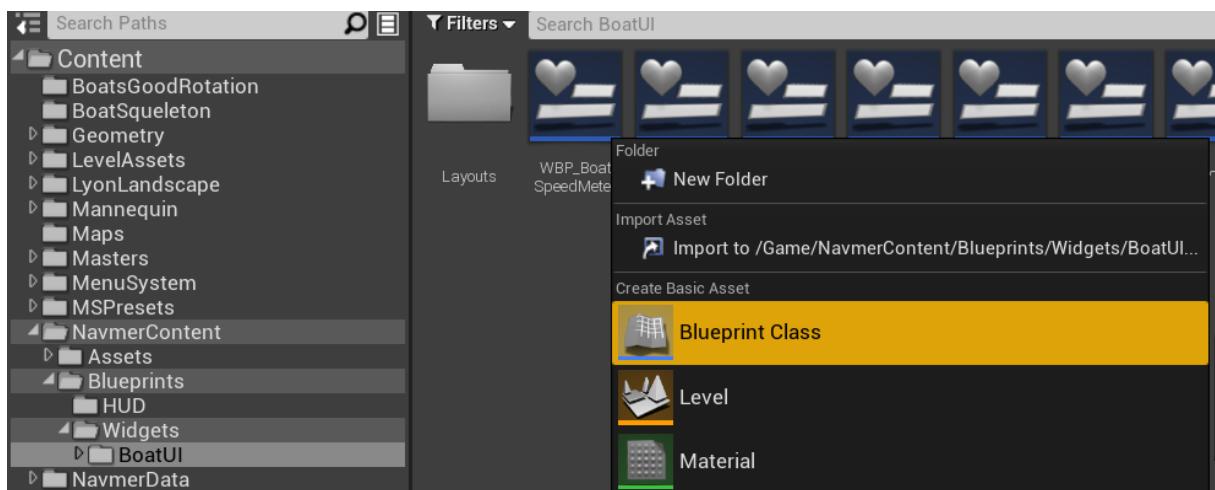


FIGURE 25 – Créer un blueprint via le Content Browser.

3. Dans la nouvelle fenêtre intitulée “Pick Parent Class”, cherchez “NMRudder” sous “All Classes”
4. Sélectionnez la classe code **NMRudder** et validez (bouton “Select”)

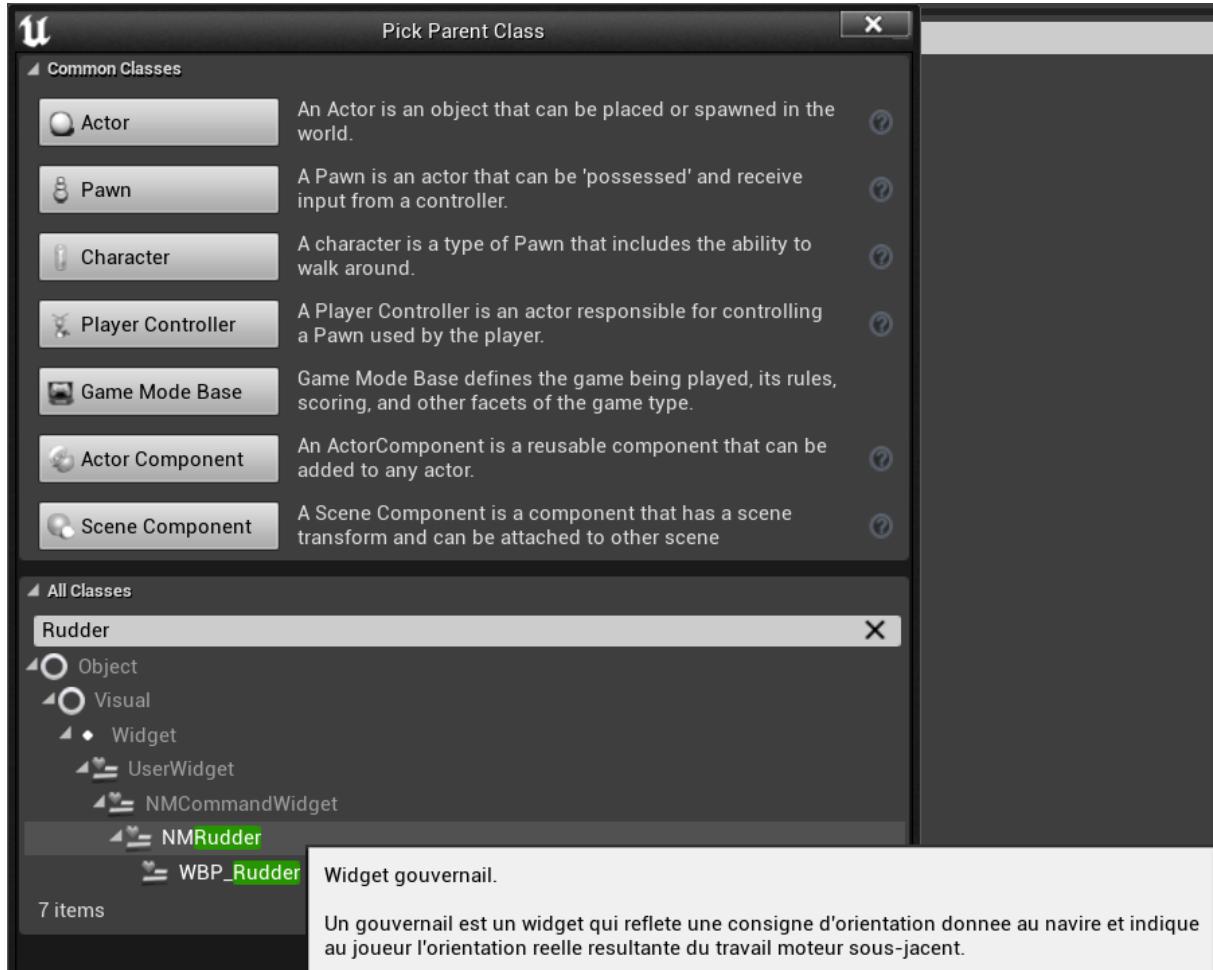


FIGURE 26 – Le blueprint doit hériter de la classe code NM_Rudder.

Le nouveau blueprint est alors créé dans le sous-dossier *NavmerContent/Blueprints/Widgets/BoatUI*.

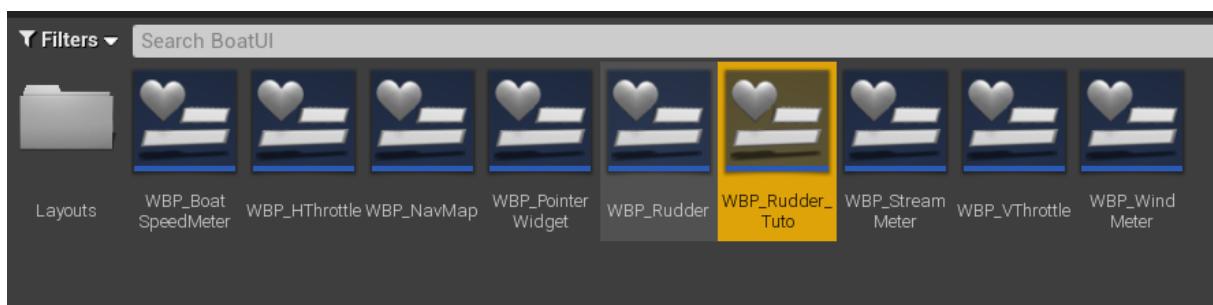


FIGURE 27 – Le blueprint ainsi créé et renommé “WBP_Rudder_Tuto” est un widget blueprint.

6.1.2 Héritage

Plusieurs choses importantes sont à noter déjà.

Premièrement, les instructions de la section **Création** permettent de créer un blueprint général qui hérite d'une classe code ou d'un autre blueprint.

Cependant, le blueprint créé est un widget blueprint (visible par l'icône de ressource représentant un cœur et des barres horizontales).

Comme notre blueprint hérite de la classe code NMRRudder (UNMRudder), qui hérite elle-même de la classe **UserWidget** (UUserWidget), le blueprint devient automatiquement un widget blueprint. Tout blueprint qui hérite de la classe UserWidget est considéré par Unreal Engine comme un widget blueprint.

Deuxièmement, tout héritage d'un blueprint (depuis une classe code ou un autre blueprint) se fait lors de la création de celui-ci.

L'héritage multiple direct pour un blueprint n'est pas possible.

Lorsqu'un blueprint hérite d'une classe code, il hérite de tous les attributs et méthodes dès lors que ceux-ci sont renseignés par un **UPROPERTY**¹⁵ spécifique dans le code source de la classe parente.

NOTE :

Dans le cadre du développement sur Unreal Engine, il est de bon usage de tirer parti de la complémentarité des classes codes et des blueprints.

Le blueprint et la classe code ont chacun des avantages distincts.

Il faut simplement être au courant que le point fort du blueprint est son accessibilité, mais que son utilisation a ses limites, notamment pour des problèmes de performances.

À savoir qu'un projet Unreal Engine peut être développé uniquement à l'aide de blueprints (suivant sa complexité).

L'usage des blueprints au détriment du C++ (et vice-versa) est à la charge du programmeur suivant les requis.

De manière à séparer les métiers, à bénéficier des avantages des deux et à accéder plus facilement aux ressources du projet, on considère un workflow où les blueprints et les classes code sont utilisés ensemble.

Les blueprints ne nécessitent pas forcément de fortes compétences en programmation ou de connaître le langage C++ (comme p.ex. les widget blueprints).

16

15. Liste assez exhaustive des UPROPERTY : <https://benui.ca/unreal/uproperty/>

16. Choisir entre un projet blueprint et un projet C++ : <<https://docs.unrealengine.com/4.27/en-US/Resources/SampleGames/ARPG/BalancingBlueprintAndCPP/>>

Dans le cas de la classe code NMRRudder, il n'y a aucun attribut et aucune méthode à hériter.

Le fait qu'un widget blueprint puisse hériter des attributs et des méthodes d'une classe code permet, dans une approche modulaire, de définir le comportement de celui-ci dans l'onglet Graph via le visual scripting.

Cependant, le widget de gouvernail étant un widget d'état de commande, il suit une implémentation C++ de sa logique sur plusieurs niveaux (plusieurs héritages) de par la direction du cahier des charges (interface de navigation différent pour chaque type de manœuvres possible, et où les éléments visuels doivent être en mesure de communiquer avec la libnavmer).

Par contre, NMRRudder est dépendant d'un widget de type **UNMPointerWidget** qui doit le composer. Ce widget sera fourni via notre widget blueprint.

L'intérêt d'une telle approche est de pouvoir séparer strictement le rendu visuel du widget et sa logique entre le blueprint et le code.

De cette manière, le design complet du widget est à la charge d'un infographiste (de la création d'une texture à la disposition dans Unreal Engine), tandis que la logique est à la charge du programmeur.

De plus, cela permet d'avoir une multitude de designs de widgets (p.ex. deux widgets de gouvernail qui n'ont pas la même apparence) sans avoir à réécrire la logique d'un blueprint à un autre.

6.1.3 Compilation et Résultats

Revenons maintenant à la création de WBP_Rudder_Tuto :

5. Ouvrez WBP_Rudder_Tuto via le Content Browser

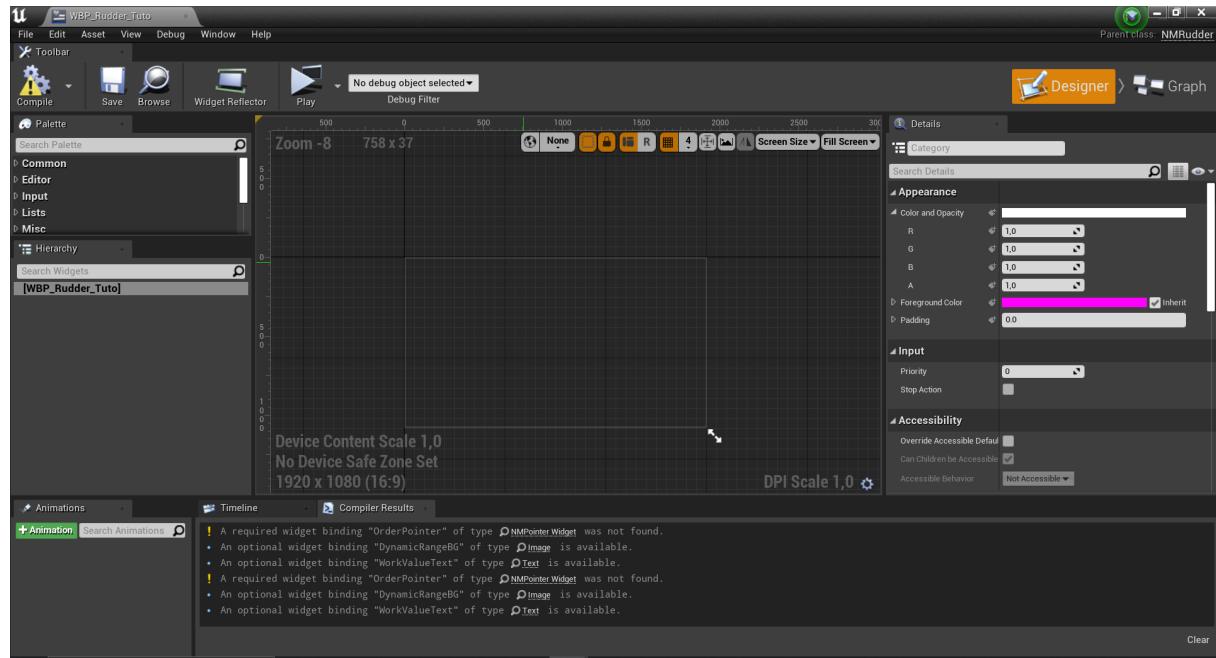


FIGURE 28 – Designer de WBP_Rudder_Tuto.

Remarquez tout d'abord le “Parent class: NM Rudder” en haut à droite de la fenêtre : le widget blueprint WBP_Rudder_Tuto hérite bien de la classe code NM Rudder.

Si vous cliquez sur “NM Rudder”, la solution du projet va s’ouvrir pour vous permettre de modifier le code de la classe (le projet Navmer3D est relié à l’IDE Visual Studio).

Par la suite, remarquez l’icone du bouton “Compile” est accompagnée d’un warning jaune.

A l’instar d’un “langage compilé”, tout blueprint doit être compilé afin que les modifications puissent prendre effet en jeu.

Pour enregistrer les modifications sur le projet, il faut cependant appuyer sur le bouton “Save”.

Il n’est à priori pas possible de compiler le blueprint pour le moment. Pour comprendre pourquoi : après avoir tenté de compiler, rendez-vous dans la fenêtre “Compiler Results” (accessible depuis le menu “Window” dans la barre principale).

```
② A required widget binding "OrderPointer" of type UNMPointerWidget was not found.  
• An optional widget binding "DynamicRangeBG" of type UImage is available.  
• An optional widget binding "WorkValueText" of type UText is available.  
② A required widget binding "OrderPointer" of type UNMPointerWidget was not found.  
• An optional widget binding "DynamicRangeBG" of type UImage is available.  
• An optional widget binding "WorkValueText" of type UText is available.  
• [1580,62] Compile of WBP_Rudder_Tuto failed. 2 Fatal Issue(s) 0 Warning(s) [in 37 ms] (/Game/NavmerContent/Blueprints/Widgets/BoatUI/WBP_Rudder_Tuto.WBP_Rudder_Tuto)
```

FIGURE 29 – Résultats de compilation de WBP_Rudder_Tuto.

Cette fenêtre rapporte les erreurs et warnings obtenus lors d'une tentative de compilation d'un blueprint.

Pour WBP_Rudder_Tuto, il est notifié dans les résultats de compilation que des **bindings** à certains widgets sont requis. Les bindings de widgets seront expliqués par la suite. Pour le moment, n'y prêtez pas attention.

6.1.4 Base du layout

Maintenant, il est temps de s'occuper de la partie “design” du widget du gouvernail.

Premièrement, puisqu'il est question de concevoir un widget qui n'a vocation qu'à être utilisé dans un autre widget (le widget sera utilisé dans un layout), il n'est pas nécessaire que celui-ci s'adapte à la configuration de l'écran (par contre, ce sera le cas pour le layout).

Parmi les boutons du viewport, cliquez sur la liste déroulante avec “Fill Screen” et sélectionnez “Desired” à la place. Le canevas dans le viewport va alors “disparaître”.

De cette façon, le canevas va englober les widgets qui seront insérés dans le viewport.

Deuxièmement, nous allons avoir besoin de 4 widgets principaux :

- 2 widgets **Image** pour la texture de fond et le matériau instancié de plage dynamique du gouvernail
- 1 widget **WBP_PointerWidget** (propre à Navmer3D) pour indiquer graphiquement la valeur d'angle en consigne
- 1 widget **Text** pour indiquer sous forme de texte la valeur d'angle réelle

Le widget Image permet de visualiser une texture ou un matériau, tandis que le widget Text agit comme un label ou une boîte de texte.

Pour commencer, il faut ajouter des **conteneurs**.

Les conteneurs sont des widgets qui ont pour utilité d'imbriquer et d'organiser dans l'espace des widgets. Ils créent une forte relation parent-enfant.

Il existe différents conteneurs proposés par Unreal Engine, qui ont des comportements différents. Nous allons en voir quelques-uns.

L'Overlay :

Ce premier conteneur permet d'afficher simplement des widgets les uns sur les autres en fonction de leur ordre dans la hiérarchie.

Plus un widget contenu dans un Overlay se trouve en bas de la hiérarchie, plus celui-ci sera affiché en premier plan par rapport aux autres.



FIGURE 30 – Exemple avec l'Overlay. L'image rouge, située en haut dans la hiérarchie, se trouve derrière le texte “Exemple” vert, situé en bas de la hiérarchie.

Le CanvasPanel :

Ce conteneur, orienté sur la composition manuelle d'un widget (et non de manière procédurale), permet d'afficher arbitrairement des widgets dans un espace défini par le conteneur.

Les widgets contenus sont manuellement disposés dans le conteneur, suivant une certaine profondeur (ordonnée z) et des **ancrages** aux bords du conteneur.

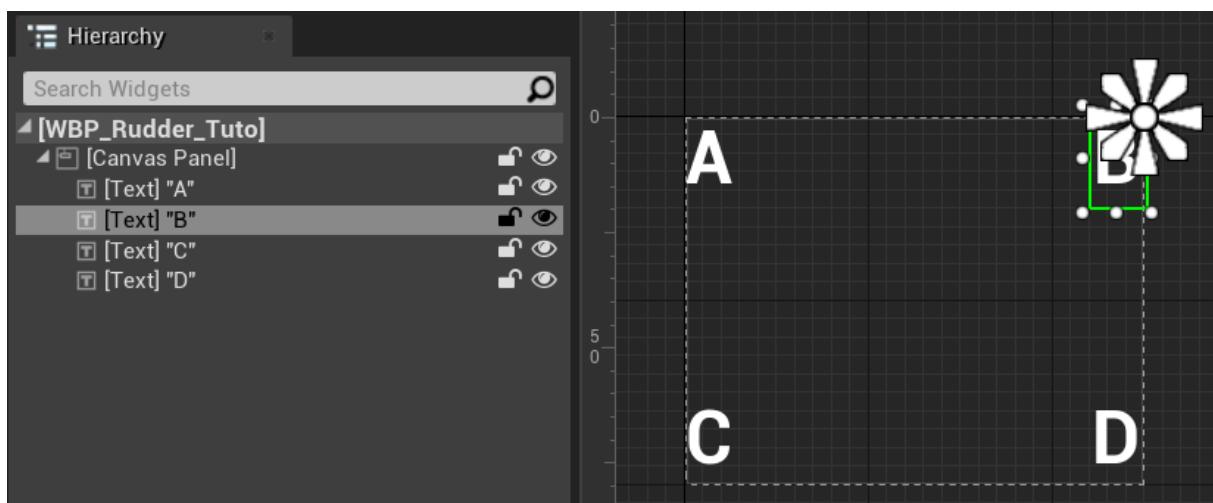


FIGURE 31 – Exemple avec le CanvasPanel. La rosace blanche indique le point d'ancrage d'un widget dans le CanvasPanel. Ici, le texte B est ancré en haut à droite.

Le Border :

Ce conteneur a la particularité de ne prendre qu'un seul widget enfant.

Son utilité est de pouvoir ajouter une couleur ou texture de fond, ou de modifier le padding du widget contenu.

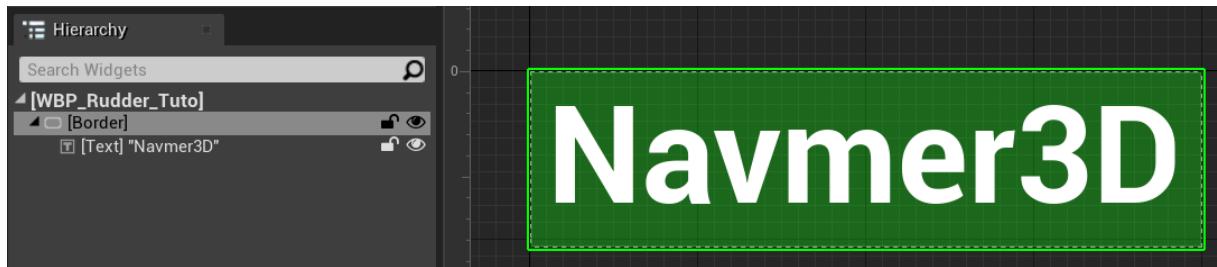


FIGURE 32 – Exemple avec le Border. Le texte “Navmer3D” se trouve sur un fond vert semi-transparent. Du padding entre les bords du fond vert et le texte est également ajouté.

La VerticalBox et HorizontalBox :

Ce dernier conteneur agit un peu comme une liste de widgets affichée verticalement.

Les widgets sont affichés dans l'ordre de la hiérarchie : le dernier widget ajouté se trouve le plus en bas de la verticalbox.

Il y a également une version horizontale avec le conteneur **HorizontalBox**.

Du point de vue du programmeur, ce type de conteneur est intéressant puisqu'il est possible de remplir celui-ci de manière procédurale.

Cependant, dans le cas où une fonctionnalité du genre est souhaitée, il serait plus avisé d'utiliser le widget **ListView** qui agit comme une zone de liste et propose une meilleure interface ainsi qu'une barre de scrolling.

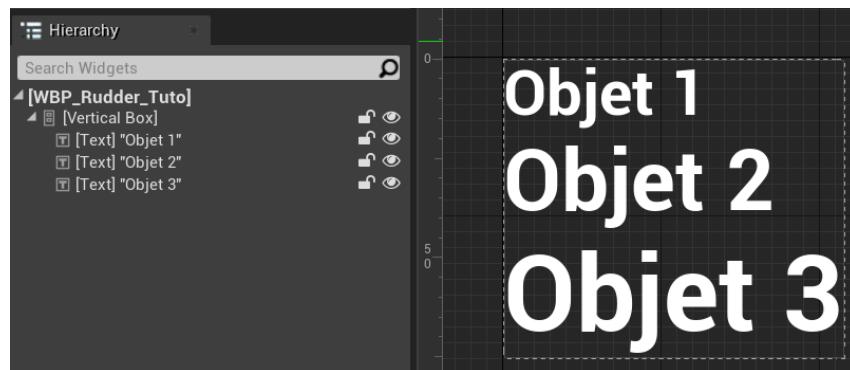


FIGURE 33 – Exemple avec la VerticalBox. Les textes (de tailles arbitraires) sont affichés à la suite, dans l'ordre de la hiérarchie.

À savoir que chaque conteneur ajoute une propriété aux widgets qu'il contient.

Cette propriété, représentée sous la rubrique "Slot", permet de modifier le comportement spatial du widget contenu dans le conteneur.

Chaque conteneur ayant un comportement différent, la rubrique Slot d'un widget contenu offre des paramètres propres à la relation avec le conteneur (ceci-dit, on peut retrouver certains paramètres sur différents conteneurs, comme le padding d'un widget par exemple).

Ajoutons en premier les différents conteneurs qui vont composer notre widget de gouvernail. Pour ce faire :

1. Dans la Palette, cherchez le conteneur Overlay et ajoutez-le en le glissant-déposant dans la hiérarchie ou dans le viewport directement.
2. Dans l'Overlay, ajoutez-y un CanvasPanel (i.e. glissez-déposez le CanvasPanel depuis la Palette sur l'Overlay présent dans la hiérarchie).

Vous pouvez renommer les conteneurs comme bon vous semble, cependant ce n'est pas nécessaire dans notre cas.

 **NOTE :**

Il peut être par exemple nécessaire de renommer les conteneurs si vous souhaitez y avoir accès depuis le graphe ou la classe code parente.

Maintenant, ajoutons les différents widgets principaux de notre widget de gouvernail :

3. Dans le CanvasPanel, ajoutez-y 2 widgets Image, 1 widget Text et 1 widget personnalisé WBP_PointerWidget (trouvable sous la rubrique “User Created” de la Palette, ou en filtrant la recherche).
4. Enfin, renommez ces 4 widgets respectivement en “Background”, “DynamicRangeBG”, “WorkValueText” et “OrderPointer” (i.e. cliquez sur chaque widget dans la hiérarchie et changez leur nom dans le panneau Details, juste en dessous de “Details”) : **cela aura son importance plus tard.**
5. Encapsulez le widget *WorkValueText* dans un Border (i.e. glissez-déposez le nouveau Border sur le CanvasPanel, puis glissez-déposez ensuite *WorkValueText* sur le Border).

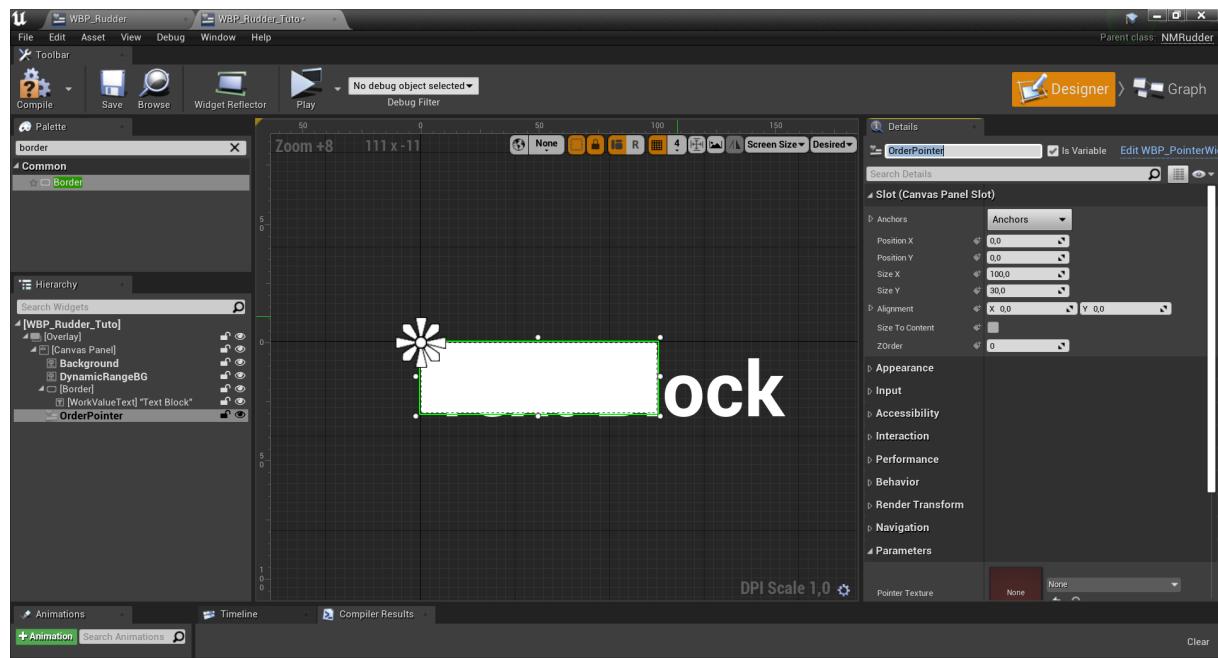


FIGURE 34 – Notre base de widget gouvernail.

6.1.5 Background

Nous allons maintenant modifier les propriétés de chaque widget, notamment leur apparence.

Commençons, en détails, par le widget *Background*.

Background va simplement héberger notre texture de fond de gouvernail :

6. Sélectionnez *Background*.
7. Dans la rubrique “Appearance” du panneau Details, déroulez “Brush”.
8. Dans le champ “Image”, ouvrez la liste déroulante qui indique “None”, recherchez sous “Browse” la texture *Default_Rudder* et sélectionnez-la.
9. Comme la texture de fond est de base un peu trop grande pour un layout défini sur du 1920x1080, dans le champ “ImageSize”, mettez X et Y à 168 (dimensions arbitraires).
10. Enfin, afin de rajouter légèrement de la transparence pour potentiellement voir ce qu'il se passe en arrière-plan : sous “ColorAndOpacity”, laissez les composantes R, G et B à 1 et mettez la composante A à 0.9 par exemple.

Le fait de laisser les composantes R, G et B (rouge, vert et bleu) à 1 n'applique aucune teinte sur la texture (i.e. la texture garde sa pleine couleur et n'est pas mélangée à une autre).

La composante A (canal alpha) agit sur l'opacité du widget.

 **NOTE :**

Le champ “Tint” du widget Image permet d'agir sur la teinte de la texture et non sur la couleur du widget, comme avec la propriété ColorAndOpacity.

Individuellement, il n'y a pas de différence visuelle entre les deux.

Le champ “DrawAs” du widget Image permet d'indiquer comment la texture doit être affichée (p.ex. en tant qu'image sans marges, ou avec marges)

Dans notre cas, on souhaite un affichage “Image”.

Le champ “Tiling” du widget Image permet d'indiquer si la texture doit se répéter.

Dans notre cas, on préfèrera l'option “No Tile”.

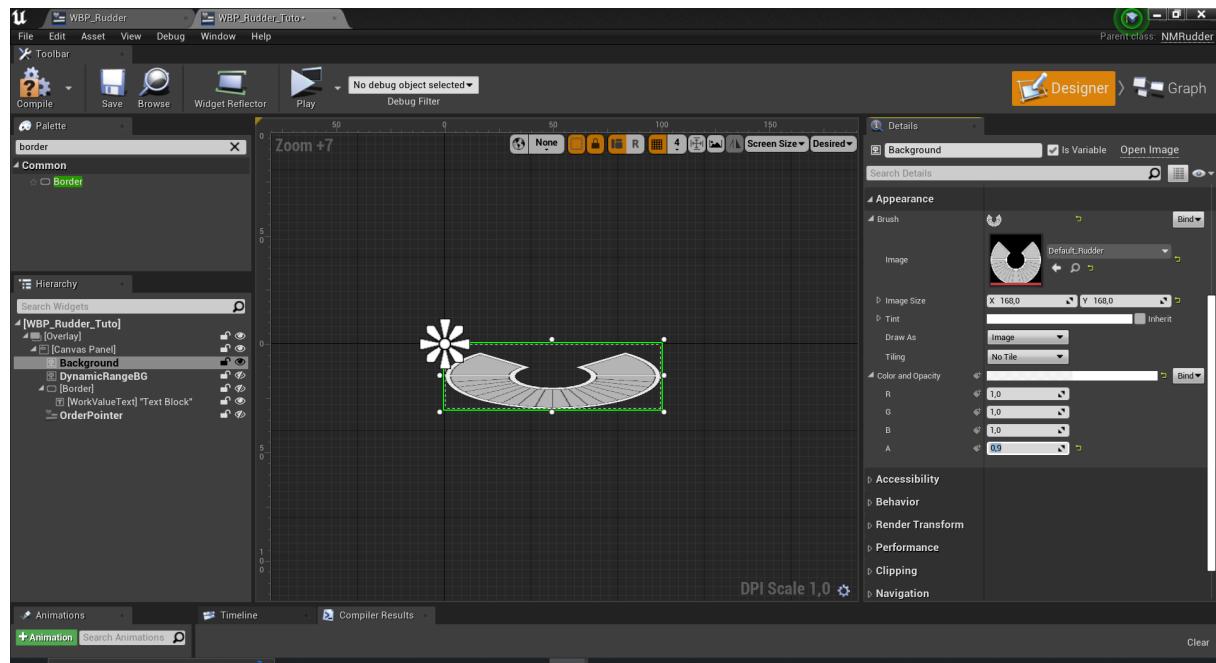


FIGURE 35 – Apparence du widget *Background*. Notez dans la hiérarchie les icônes de yeux fermés pour cacher/afficher les autres widgets.

Vous remarquerez que la texture est pour le moment “écrasée”.

De plus, lorsque vous avez modifié les dimensions du widget au travers du champ `ImageSize`, rien n'a changé.

Cela est dû au fait que le conteneur (`CanvasPanel`) ne respecte pas les dimensions du widget.

Pour corriger cela, regardons maintenant du côté de la rubrique Slot.

Ici, notre propriété Slot est reliée au conteneur (CanvasPanel).

Les champs présents sont donc dépendants et propres au CanvasPanel situé directement au dessus dans la hiérarchie :

11. Dans le champ “Anchors”, ouvrez la liste déroulante des prédefinis qui indique “Anchors” et sélectionnez un ancrage central (un dessin de petit carré placé au centre).

Suite à cela, la rosace d’ancrage se déplace au centre du widget *Background* pour indiquer que celui-ci est bien ancré au centre du CanvasPanel.

Cependant, cela modifie par la même occasion la position du widget dans le conteneur, ce qui n'est pas souhaitable :

12. Réinitialisez les valeurs de position (i.e. cliquez sur les flèches jaunes dans les champs PositionX et PositionY, ou mettez tout à zéro).

Background se retrouve malheureusement désaxé.

Pour faire les choses propres (en général pour ne pas avoir de comportements non désirés lorsque l'on change les dimensions de la fenêtre de jeu), on souhaite que le widget ait une position neutre par rapport à son ancrage dans le CanvasPanel ET que le point de pivot du widget corresponde à l'ancrage.

Par exemple, si le widget est ancré au centre du CanvasPanel, on souhaite que le point de pivot soit également au centre du widget.

Le point de pivot d'un widget dans un CanvasPanel est originellement en haut à gauche du widget, il faut donc le changer :

13. Dans le champ “Alignment”, mettez X et Y à 0.5.

Enfin, de manière à ce que le CanvasPanel respecte les dimensions du widget :

14. Cochez la case “SizeToContent”.

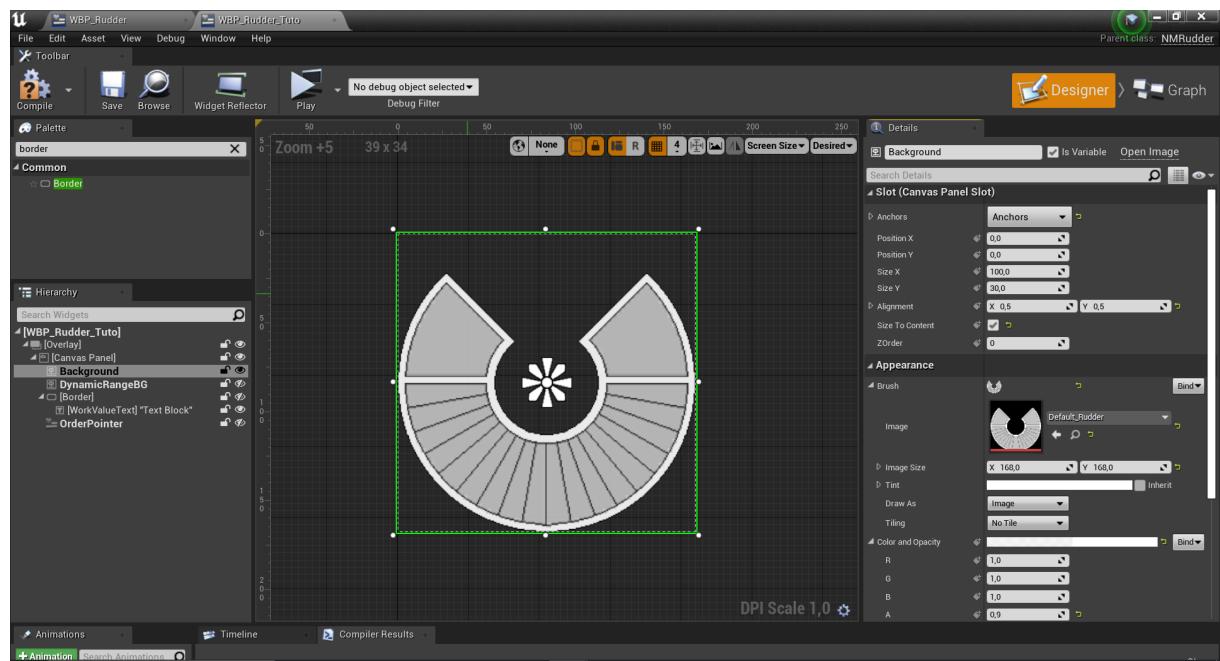


FIGURE 36 – Apparence finale du widget *Background*.

6.1.6 DynamicRangeBG

Le widget *DynamicRangeBG* répète sensiblement les mêmes opérations, avec comme différence qu'on lui passe non pas une texture mais un matériau instancié.

De cette manière, on pourra affecter plus tard le comportement du matériau en jeu.

Il prend les propriétés suivantes :

- Slot :
 - Ancrage central au CanvasPanel
 - Position à (0, 0)
 - Alignment à (0.5, 0.5)
 - SizeToContent à True
- Appearance :
 - Image avec le matériau instancié *MI_RudderDynamicRangeBG*
 - ImageSize à (168, 168) : les dimensions doivent correspondre à celles du widget *Background* (cf. point 5, section [Textures Navmer3D](#))
 - ColorAndOpacity à (1, 0, 0, 0.7) : le matériau sera affiché en rouge et légèrement transparent

À ce niveau, vous ne devriez normalement avoir aucun résultat visuel : c'est normal puisque le matériau instancié a ses valeurs de paramètres à zéro.

Vous pouvez modifier les valeurs en ouvrant le matériau instancié *MI_RudderDynamicRangeBG* pour vous assurer que la partie design du widget est bonne (la jauge doit remplir le widget *Background*).

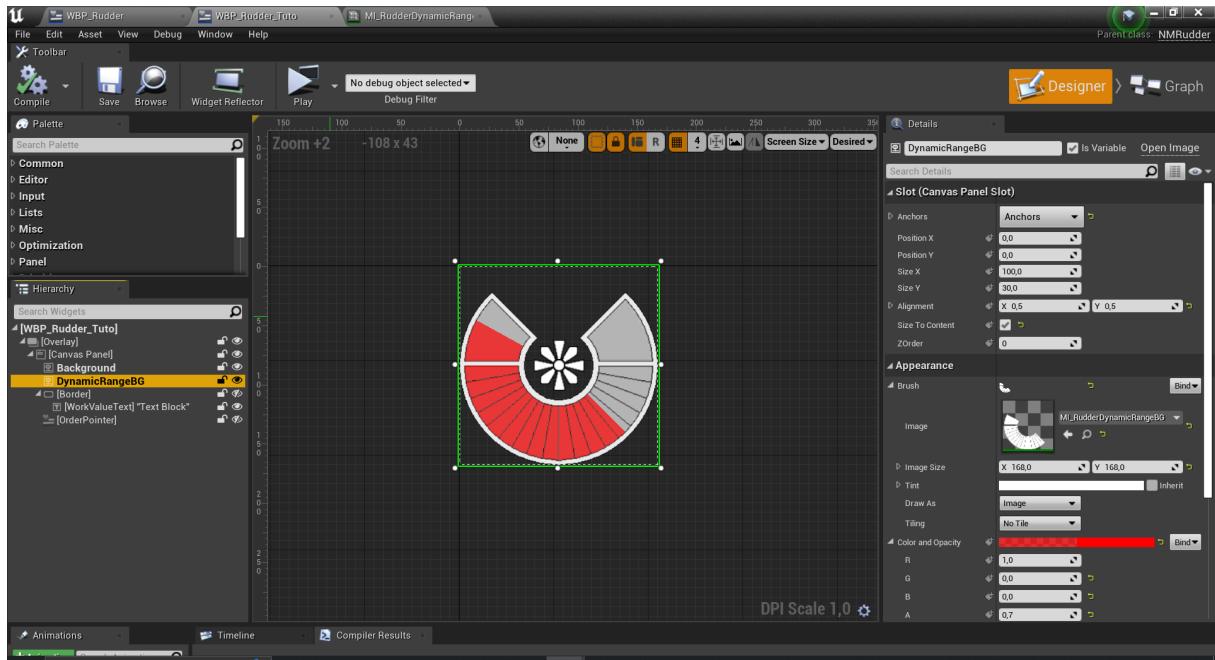


FIGURE 37 – Apparence finale du widget *DynamicRangeBG*.

À l'instar de *Background*, le widget personnalisé (pour Navmer3D) *OrderPointer* agit comme un simple widget *Image* dans l'objectif d'afficher une texture de “pointeur” (i.e. une flèche qui indique une direction).

Cependant, c'est également un widget qui hérite d'une classe code qui définit entièrement sa fonctionnalité et qui hérite de propriétés propres intéressantes.

En dehors du Slot, les propriétés à changer ici ne sont pas les mêmes; certaines nécessitent également des explications en parallèle (cf. [Propriétés de PointerWidget](#) pour en savoir plus) :

- Slot :
 - Anchrage central au CanvasPanel
 - Position à (0, 0)
 - Alignment à (0.5, 0.5)
 - SizeToContent à True
- Sous la rubrique “Parameters” :
 - **PointerTexture** avec la texture *Default_Rudder_Pointer*
 - **PointerTextureSize** à (21, 168) : la taille horizontale est arbitraire, alors que la taille verticale correspond à celle de *Background* et *DynamicRangeBG*
 - **PointerColor** à 1 pour toutes les composantes (R, G, B et A) (dérouler la propriété ou cliquer sur la couleur pour changer)
 - **PointerTextureBaseOrientation** à zéro

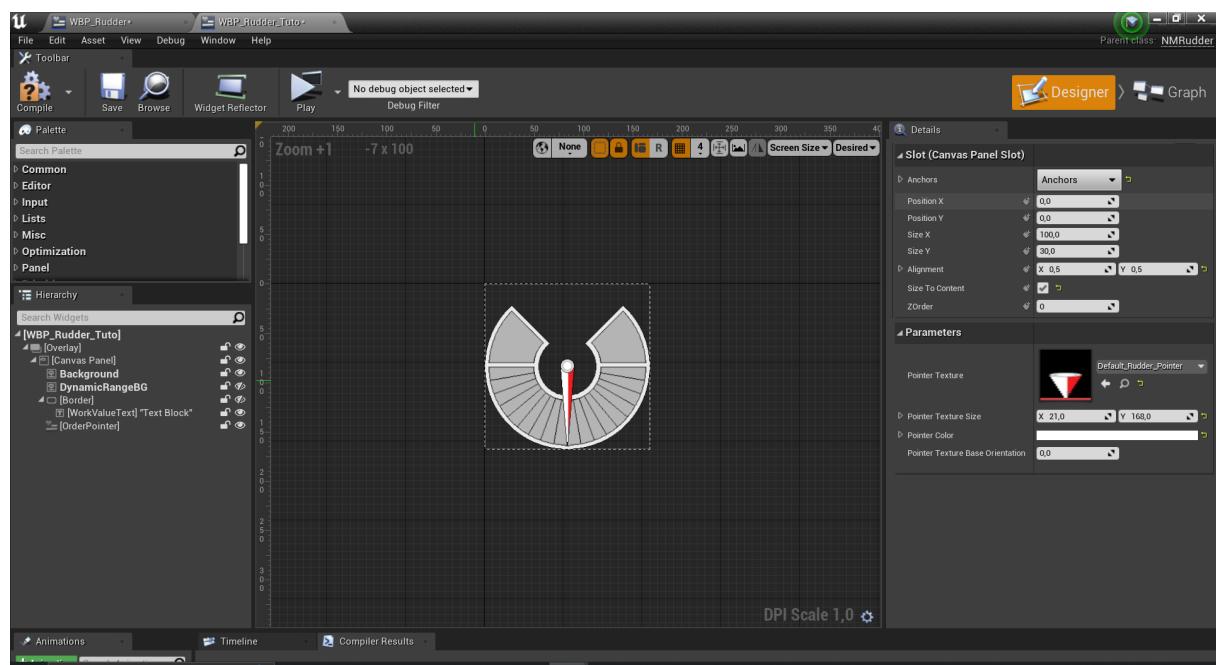


FIGURE 38 – Apparence finale du widget *OrderPointer*.

6.1.7 WorkValueText

Enfin, le widget *WorkValueText*, qui est encapsulé dans un Border, nécessite quelques instructions supplémentaires pour sa mise en forme.

Nous allons tout d'abord définir un fond semi-transparent pour que l'on puisse non seulement voir légèrement ce qui se passe en arrière-plan, mais aussi pour que l'on puisse voir le texte dans le cas où la couleur de l'arrière-plan cacherait celui-ci (p.ex du texte blanc sur un fond blanc).

Pour cela, nous allons nous servir du Border qui encapsule *WorkValueText* :

15. Sélectionnez le Border.
16. Dans la rubrique Appearance du Border, définissez “BrushColor” à (0, 0, 0, 0.5) : on obtient un fond noir et semi-transparent.

Il nous faut maintenant positionner le Border dans le CanvasPanel et faire en sorte qu'il contienne entièrement le texte de *WorkValueText* :

17. Dans la rubrique Slot du Border, ancrez le Border au centre du CanvasPanel.
18. Réinitialisez PositionX seulement pour le moment.
19. Mettez les valeurs d'Alignment à 0.5.
20. Cochez la case SizeToContent.

Nous allons ensuite changer l'apparence du texte de *WorkValueText* :

21. Sélectionnez *WorkValueText*.
22. Dans la rubrique Appearance de *WorkValueText*, sous la propriété “Font” : saisissez dans le champ “Size” une taille de police à 12.
23. Tout en bas de la rubrique Appearance : modifiez la “Justification” pour aligner le texte au centre.

Laissez la propriété ColorAndOpacity telle quelle (i.e. une couleur blanche avec aucune transparence).

Les autres champs de la propriété Font sont assez classiques et propres aux utilitaires de polices d'écriture qu'on peut retrouver dans n'importe quel logiciel de dessin assisté.

Il est également possible, via le champ “FontMaterial”, d'affecter le rendu du texte avec un matériau.

Pour le moment, le texte affiché est “Text Block” (texte par défaut du widget Text). Pour changer cela :

24. Depuis la rubrique “Content” de *WorkValueText*, changez la valeur du texte en “0°” (zéro degrés).

 **NOTE :**

Le texte d'un widget Text défini dans l'onglet Designer est un texte par défaut.

La valeur du texte n'est pas forcément fixe, notamment si celle-ci est changée via un noeud de l'onglet Graphe, ou bien depuis la classe code parente.

Le widget *WorkValueText* nécessite d'être positionné correctement dans le Border qui l'encapsule :

25. Depuis la rubrique Slot de *WorkValueText*, centrez l'alignement horizontal et vertical.

Toujours dans la rubrique Slot, vous pouvez agir sur le Padding si vous le souhaitez pour modifier les marges de *WorkValueText* dans le Border. Le Padding fonctionne de la manière suivante :

- Saisissez 1 valeur pour des marges équitables
- Saisissez 2 valeurs pour modifier dans l'ordre de saisie les marges horizontales et les marges verticales
- Saisissez 4 valeurs pour modifier dans l'ordre de saisie la marge gauche, la marge supérieure, la marge droite et la marge inférieure

Enfin, jouez avec PositionY du Border (rubrique Slot du Border) pour déplacer le border en haut du gouvernail.

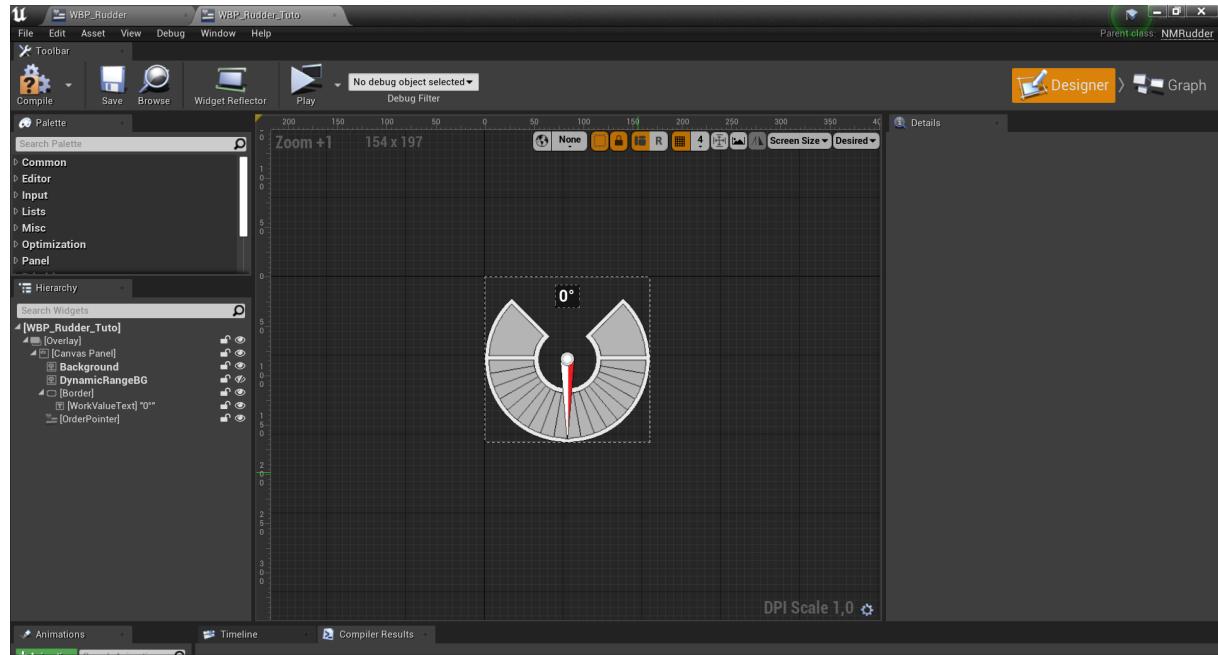


FIGURE 39 – Apparence finale de *WorkValueText*.

Les principaux éléments de notre widget WBP_Rudder_Tuto sont maintenant présents et bien définis : le principal du design est fait.

À ce niveau là, si vous ouvrez WBP_Rudder, vous remarquerez qu'il manque certains éléments : les métriques qui indiquent les paliers d'angles.

Normalement, grâce aux instructions et explications précédentes, vous devriez être capable de rajouter ces éléments par vous même à présent.

Dans l'idée, il vous faudra rajouter un CanvasPanel sur l'Overlay, avec autant de widgets Text que vous souhaitez indiquer de paliers d'angles sur le gouvernail.

Ce qui ne sera pas forcément aisement ici est le positionnement de tous les widgets Text autour du cadran du gouvernail, ainsi que trouver la bonne taille de police de manière à ce que le navigateur soit capable de lire le texte.

Utiliser un CanvasPanel est une bonne option car la gestion du positionnement manuel des widgets qu'il englobe est plus simple.

6.1.8 Bindings de widgets

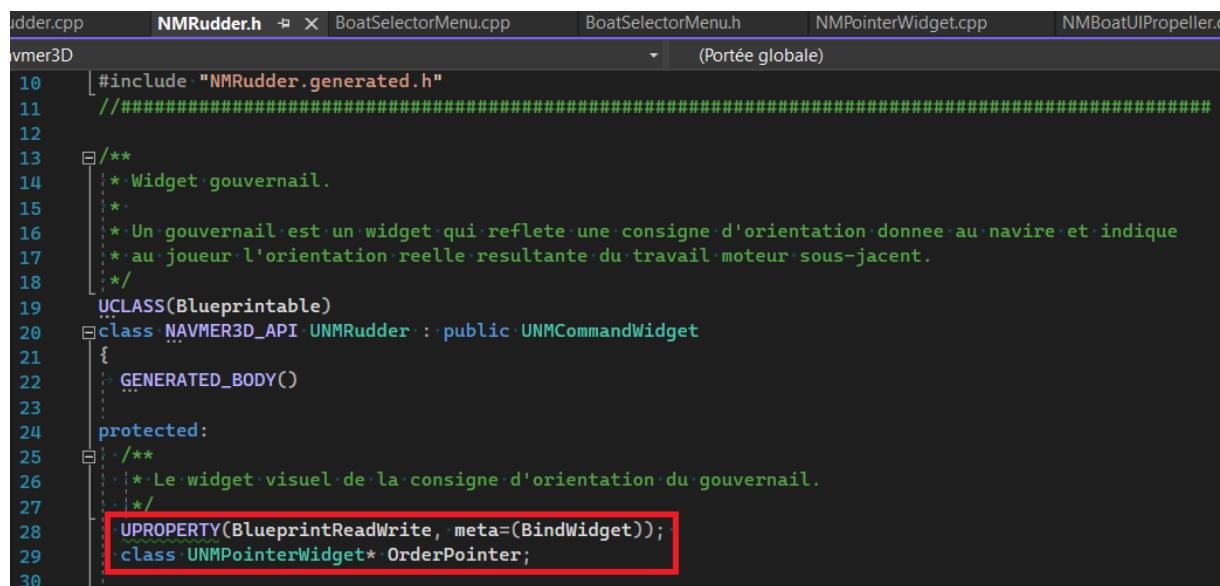
Renommer un widget dans l'onglet Design est significatif, car cela offre trois avantages :

- Une meilleure lecture et organisation dans la hiérarchie
- Une identification également appréciée dans l'onglet Graphe ou dans une classe code, lorsque le widget est utilisé en tant que variable
- Mais surtout, dans le cas de l'interface de navigation de Navmer3D : le binding de widget.

Dans la section **Compilation et Résultats**, les bindings de widgets ont brièvement été évoqués.

Le **binding de widget**, dans le contexte de la communication entre une classe code et un blueprint, permet de lier un objet widget à un attribut pointeur de même type dans la classe code parente afin d'accéder aux propriétés de ce widget pour agir sur celui-ci en run-time (i.e. en jeu) depuis le code C++.

Il se trouve que dans la classe code NMRudder, il est attendu un widget de type **UNMPointerWidget**.



```

NMRudder.cpp | NMRudder.h | BoatSelectorMenu.cpp | BoatSelectorMenu.h | NMPointerWidget.cpp | NMBoatUIPropeller.c
vmer3D                                                 (Portée globale)
10  #include "NMRudder.generated.h"
11  //#####
12
13  /**
14   * Widget·gouvernail.
15   *
16   * Un·gouvernail·est·un·widget·qui·reflekte·une·consigne·d'orientation·donnée·au·navire·et·indique
17   * au·joueur·l'orientation·réelle·résultante·du·travail·moteur·sous-jacent.
18   */
19 UCLASS(Blueprintable)
20 class NAVMER3D_API UNMRudder : public UNMCommandWidget
21 {
22     GENERATED_BODY()
23
24 protected:
25     /**
26      * Le·widget·visuel·de·la·consigne·d'orientation·du·gouvernail.
27      */
28     UPROPERTY(BlueprintReadWrite, meta=(BindWidget));
29     class UNMPointerWidget* OrderPointer;
30 }

```

FIGURE 40 – Encadré en rouge, la classe NMRudder attend un pointeur sur un objet de type **UNMPointerWidget**.

UNMPointerWidget est une classe de widget personnalisé pour Navmer3D.

On remarque la macro UPROPERTY au dessus de l'attribut **OrderPointer**, qui permet d'indiquer de la métadonnée pour le moteur. Ici, elle précise les arguments **BlueprintReadWrite** et **meta=(BindWidget)**.

L'argument BlueprintReadWrite signifie simplement qu'on autorise un blueprint qui hériterait de NM Rudder (en l'occurrence WBP_Rudder_Tuto) à obtenir un accès en lecture et écriture sur l'attribut déclaré directement sous le UPROPERTY.

 **NOTE :**

Pour le programmeur, dans le cas où celui-ci souhaite établir une partie de la logique du widget (ou d'un blueprint de manière général) via l'onglet Graph, cela a son importance.

Aussi, l'héritage des attributs d'une classe code par un blueprint ne peut se faire uniquement que si cette macro est renseignée.

Ce qui nous intéresse ici est l'argument “meta=(BindWidget)”¹⁷.

L'attribut OrderPointer de NM Rudder est un pointeur sur une valeur de type **UNMPointerWidget**.

C'est grâce à l'argument “meta=(BindWidget)” que l'on va être en capacité de fournir depuis WBP_Rudder_Tuto un objet widget à NM Rudder (i.e. la valeur pointée par l'attribut de NM Rudder sera définie dans le blueprint).

Pour que cela fonctionne, il est nécessaire que le widget **soit de même type ET qu'il porte le même nom** que l'attribut pointeur correspondant dans la classe code parente.

Pour rappel, la logique de l'interface de navigation de Navmer3D est entièrement définie dans le code C++ (i.e. le comportement des widgets en simulation est défini dans le code C++), c'est pourquoi il est important de binder les bons widgets lorsque c'est nécessaire.

 **NOTE :**

Dans le cas des widgets d'états de commandes (p.ex. WBP_Rudder_Tuto), il faut se référer à la classe code **NMCommandWidget**, puis à la classe code directement parente (p.ex. NM Rudder) pour connaître les widgets à binder.

Pour les widgets d'états de simulation, il suffit de se référer à la classe code directement parente.

De ce fait, il était important de renommer les widgets *DynamicRangeBG*, *OrderPointer* et *WorkValueText*, car ceux-ci sont demandés dans les classes NM Rudder et NMCommandWidget.

La rotation de *OrderPointer* est affectée dans la classe code NM Rudder via appel de méthode sur l'objet *OrderPointer* (de type UNMPointerWidget *).

De même pour *WorkValueText* où la valeur du texte est changée dans la classe code NM Rudder.

17. Documentation sur la macro BindWidget : <https://benui.ca/unreal/ui-bindwidget/>

Pour terminer sur la partie design de widget pour layout, rappelez-vous la fenêtre Compiler Results.

Puisque l'on a ajouté les bons widgets nécessaires, et qu'ils ont été renommés comme souhaité dans la classe code parente, si l'on souhaite compiler (via le bouton Compile) : la compilation du widget blueprint s'execute correctement.

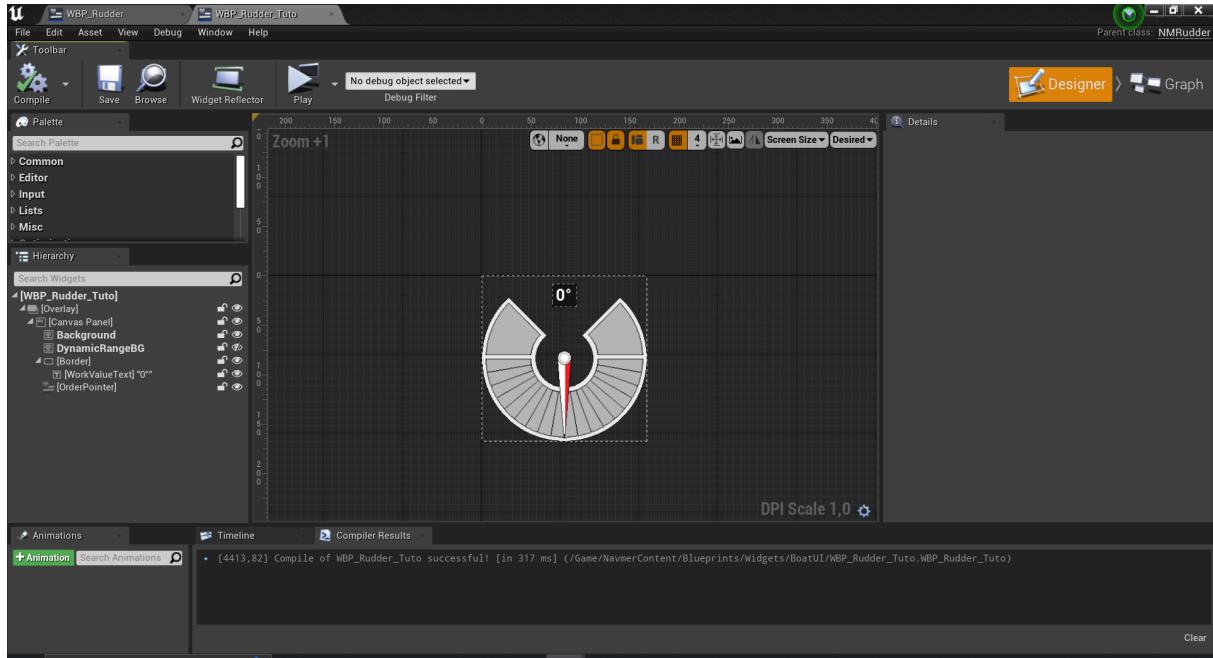


FIGURE 41 – La compilation de WBP_Rudder_Tuto s'est déroulée correctement. Vous noterez l'icone du bouton Compile qui est “cochée” en vert.

6.2 Designer un layout

6.2.1 Crédation et apparence

NOTE :

Cette section fait suite à la partie [Designer un widget pour layout](#).

Les opérations qui y figure sont sensiblement identiques à celles d'avant.

En supposant que nous avons créé tous les widgets nécessaires au layout, nous allons refaire ici le layout **WBP_BoatUIPropeller** qui est utilisé par l'interface de navigation pour une disposition “bateaux à propulsion par hélices” :

1. Rendez-vous dans le sous-dossier *NavmerContent/Blueprints/Widgets/BoatUI/Layouts*.
2. Créez le widget blueprint **WBP_LayoutPropeller_Tuto** en héritant de la classe code **NMBoatUIPropeller**.

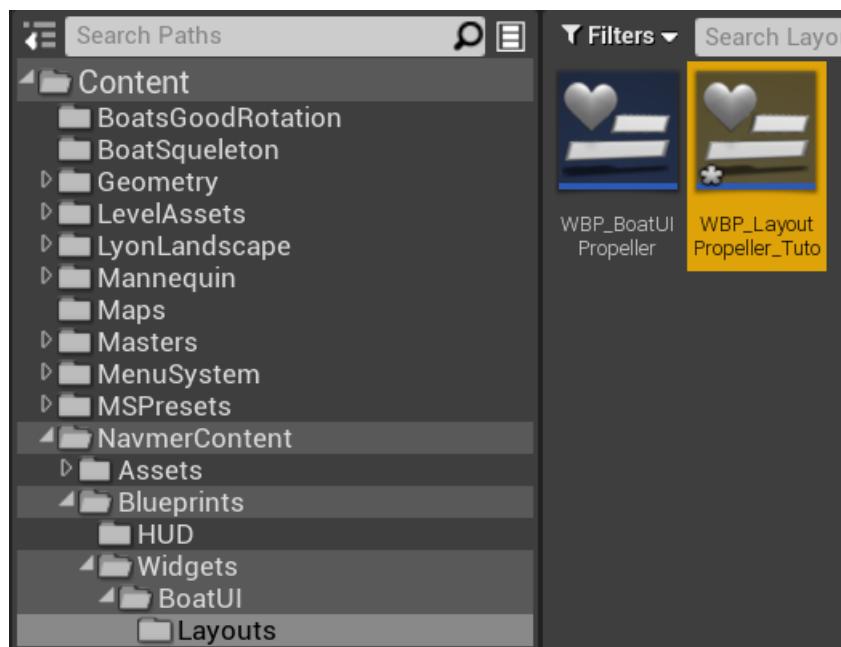


FIGURE 42 – Le nouveau widget blueprint *WBP_LayoutPropeller_Tuto* se trouve dans le sous-dossier *Layouts*.

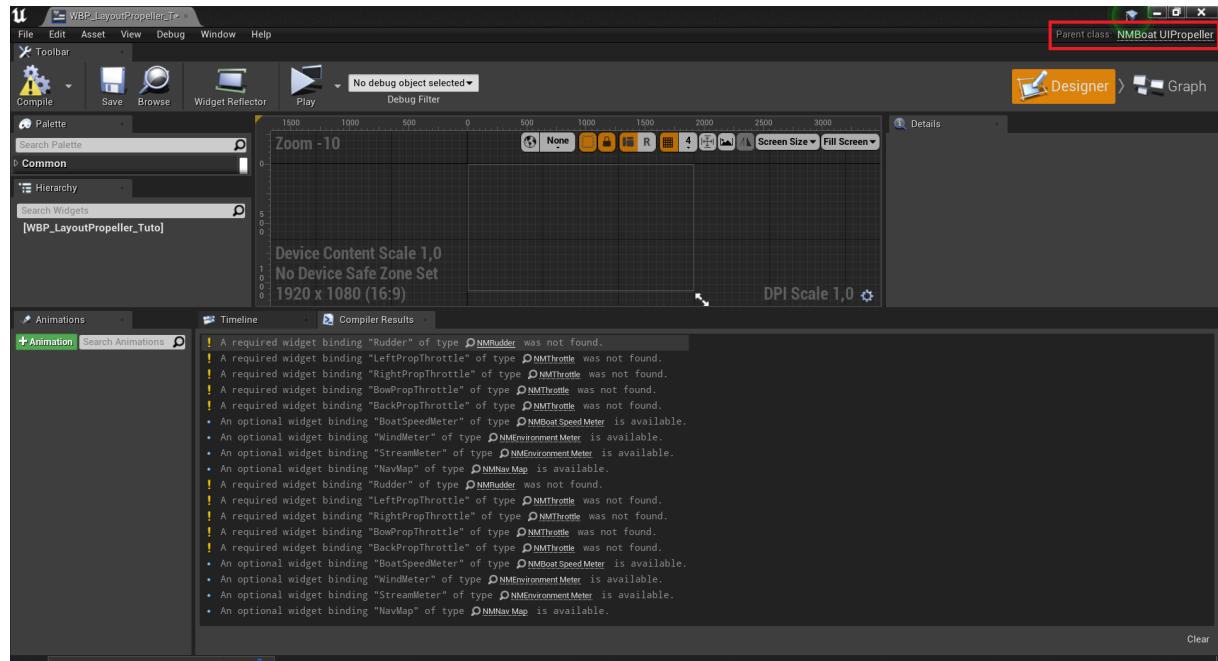


FIGURE 43 – *WBP_LayoutPropeller_Tuto* hérite de la classe code NMBoatUIPropeller (encadré rouge).

Vous remarquerez que la fenêtre Compiler Results signale un ensemble de bindings de widgets non résolus.

Il faudra donc insérer dans la hiérarchie de *WBP_LayoutPropeller_Tuto* les widgets en question, puis les renommer comme indiqué dans Compiler Results.

L'objectif ici est plus simple à atteindre que la création de widgets pour layout.

Nous allons dans un premier temps configurer le canevas pour une configuration d'écran en 1920x1080 :

3. Dans la collection de boutons en haut du viewport, assurez-vous que le canevas soit en “FillScreen” (c'est le cas par défaut normalement).

De cette manière, le canevas du viewport permettra aux widgets de remplir la surface de l'écran.

Il nous faut à présent pouvoir travailler sur une grille et des dimensions appropriées.

Nous allons disposer les widgets suivant une définition d'écran en 1920x1080 (définition d'image commune).

À priori, Unreal Engine gère une partie de la responsivité entre les différents appareils.

Cependant, ce sera à nous de faire en sorte que les éléments soient contenus dans la surface :

4. Dans la liste déroulante “ScreenSize” (à côté de FillScreen), ouvrez la liste “Monitors” et sélectionnez l'option “21.5-24 (pouces) monitor”.

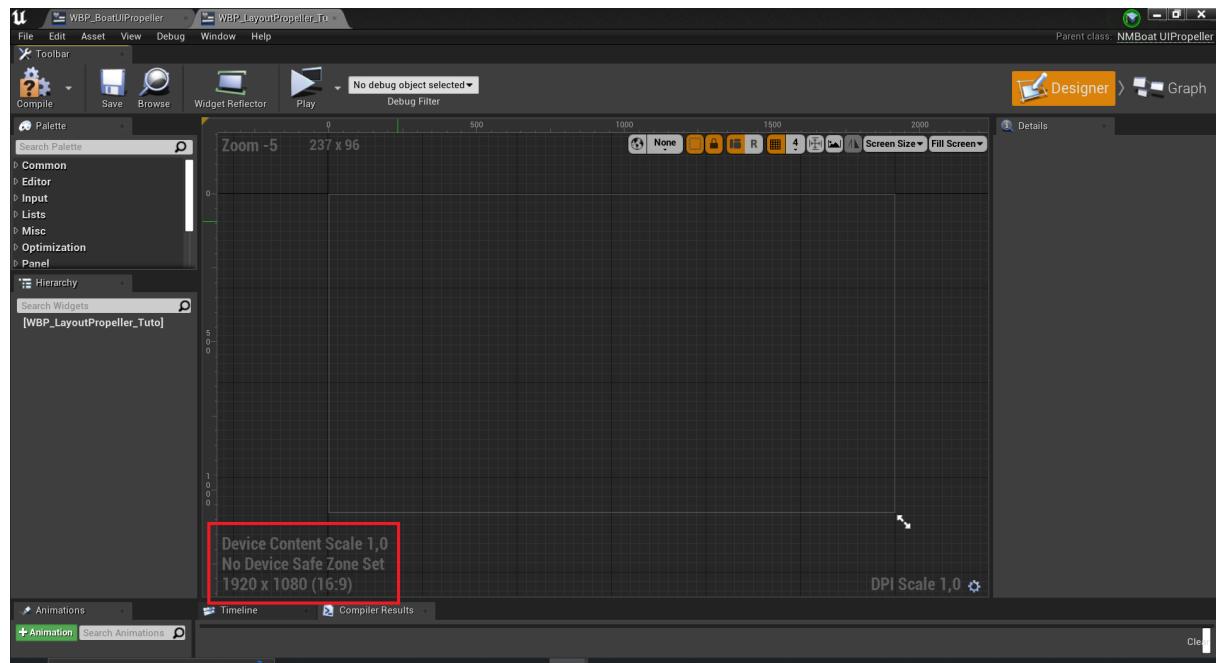


FIGURE 44 – Comme indiqué dans l’encadré rouge : le rapport de forme du canevas viewport passe en 16:9 et la définition d’écran en 1920x1080.

Dans un second temps, nous allons disposer manuellement dans le canevas les différents widgets utiles au layout à l'aide d'un CanvasPanel :

5. Ajoutez dans la hiérarchie un widget CanvasPanel.

Ce dernier prend automatiquement la forme du canevas viewport.

6. Ajoutez maintenant dans le nouveau CanvasPanel la liste suivante de widgets et renommez-les comme indiqué :
 - 1 WBP_Rudder_Tuto, à renommer en “Rudder”
 - 2 **WBP_VThrottle**, à renommer respectivement en “LeftPropThrottle” et “RightPropThrottle” (resp. manettes de propulseurs gauche et droite)
 - 2 **WBP_HThrottle**, à renommer respectivement en “BowPropThrottle” et “BackPropThrottle” (resp. manettes de propulseurs d’étrave et transversal arrière)
 - 1 *WBP_WindMeter*, à renommer en “WindMeter”
 - 1 *WBP_StreamMeter*, à renommer en “StreamMeter”
 - 1 *WBP_BoatSpeedMeter*, à renommer en “BoatSpeedMeter”
 - et enfin 1 *WBP_NavMap*, à renommer en “NavMap”.

Après cela, si vous compilez *WBP_LayoutPropeller_Tuto”, les bindings de widgets devraient être résolus.

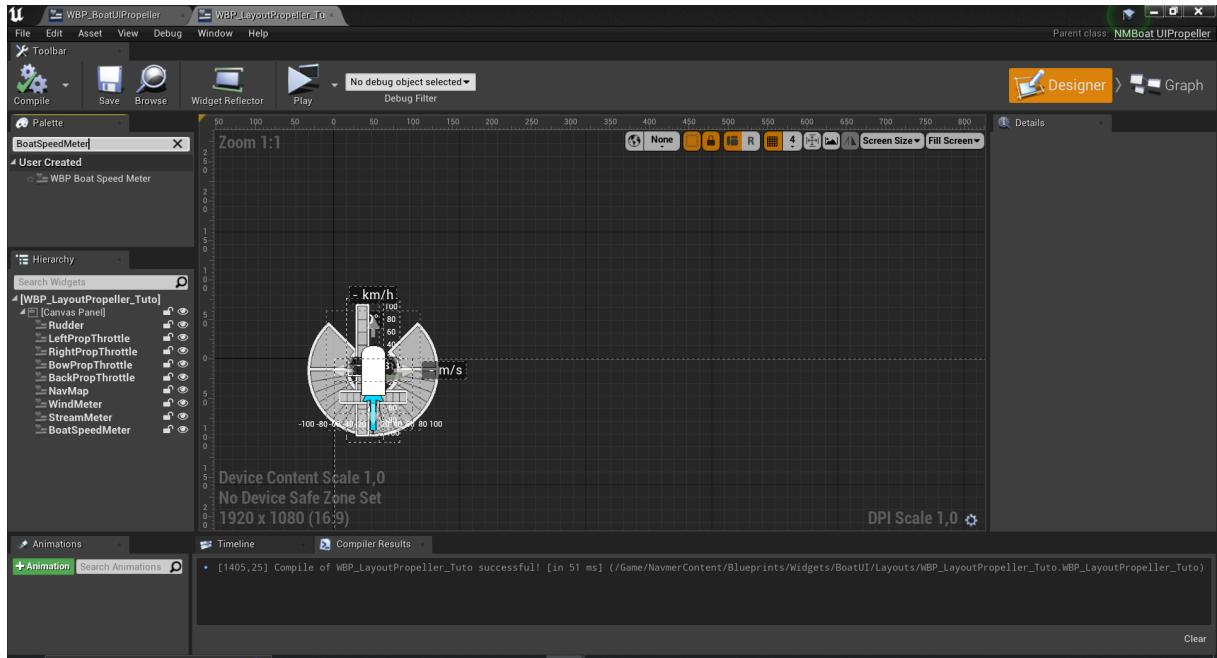


FIGURE 45 – La hiérarchie de *WBP_LayoutPropeller_Tuto* est complète.

Nous pouvons maintenant positionner les widgets dans le layout.

Nous allons respecter la disposition employée dans *WBP_BoatUIPropeller*, c'est à dire que les widgets d'états de commandes seront positionnés et ancrés par rapport au bord intérieur du CanvasPanel, et les widgets d'états de simulation seront positionnés et ancrés par rapport au bord supérieur du CanvasPanel (à l'exception de BoatSpeedMeter qui est disposé à côté des widgets d'états de commandes) :

7. Sélectionnez *Rudder* et ancrez-le au bord inférieur du CanvasPanel en choisissant un ancrage prédéfini (i.e. choisissez le petit carré en bas dans la liste déroulante Anchors).
8. Réinitialisez les valeurs des champs PositionX et PositionY.
9. Appliquez un point de pivot à 0.5 pour AlignmentX, et à 1 pour AlignmentY.
10. Cochez la case SizeToContent pour que la boîte englobante de *Rudder* (i.e. l'encadré vert lorsqu'on sélectionne le widget) prenne en compte la taille entière du widget.

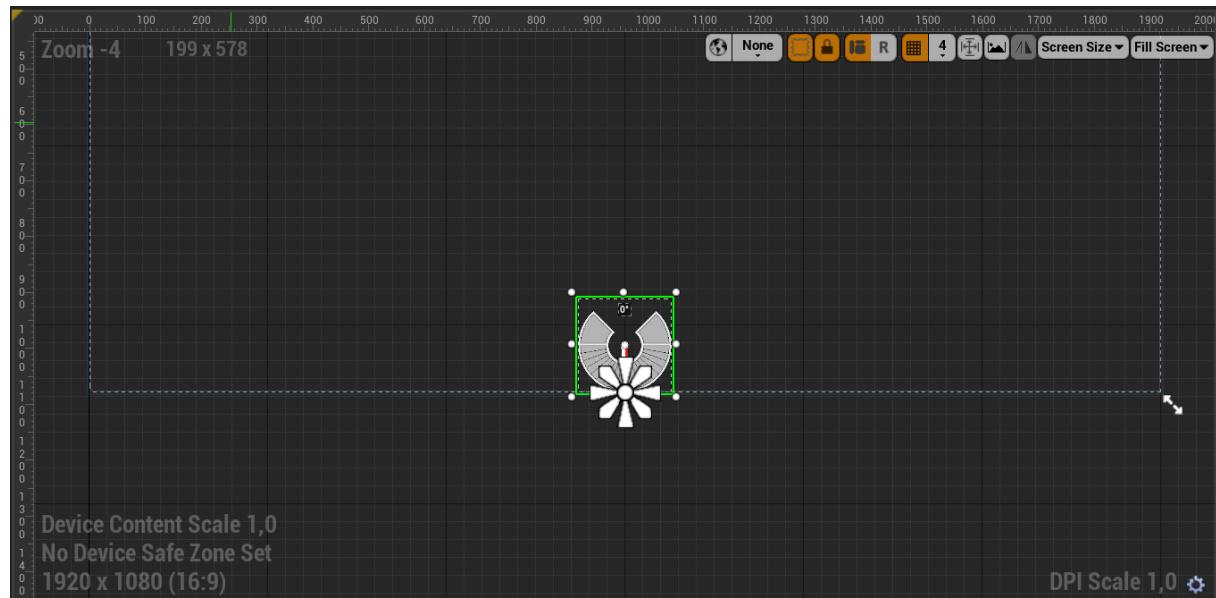


FIGURE 46 – *Rudder* est ancré au bord inférieur du *CanvasPanel*.

Vous n'avez plus qu'à jouer avec les valeurs de *PositionX* et *PositionY* pour placer votre widget vers le bas de l'écran comme bon vous semble.

Comme *Rudder* est ancré au bord inférieur et que son point de pivot est cohérent avec l'ancrage, lorsque l'on passera dans une résolution inférieure (p.ex. la fenêtre de jeu est redimensionnée) et dans le cas où d'autres widgets sont ancrés de la même façon : ceux-ci ne se chevaucheront pas et ne sortiront pas de l'écran.

Il ne vous reste alors plus qu'à faire de même avec les widgets *LeftPropThrottle*, *RightPropThrottle*, *BowPropThrottle*, *BackPropThrottle* et *BoatSpeedMeter*, en respectant l'ancrage et le point de pivot.

Pour les widgets *WindMeter*, *StreamMeter* et *NavMap*, il convient de faire un ancrage aux bords supérieurs du CanvasPanel (i.e. choisissez le petit carré en haut dans la liste déroulante Anchors de la rubrique Slot, pour chaque widget), puis de choisir un point de pivot à 0.5 pour *AlignmentX* et à 0 pour *AlignmentY*.

Vous devriez au final avoir quelque chose qui ressemble à l'image suivante :

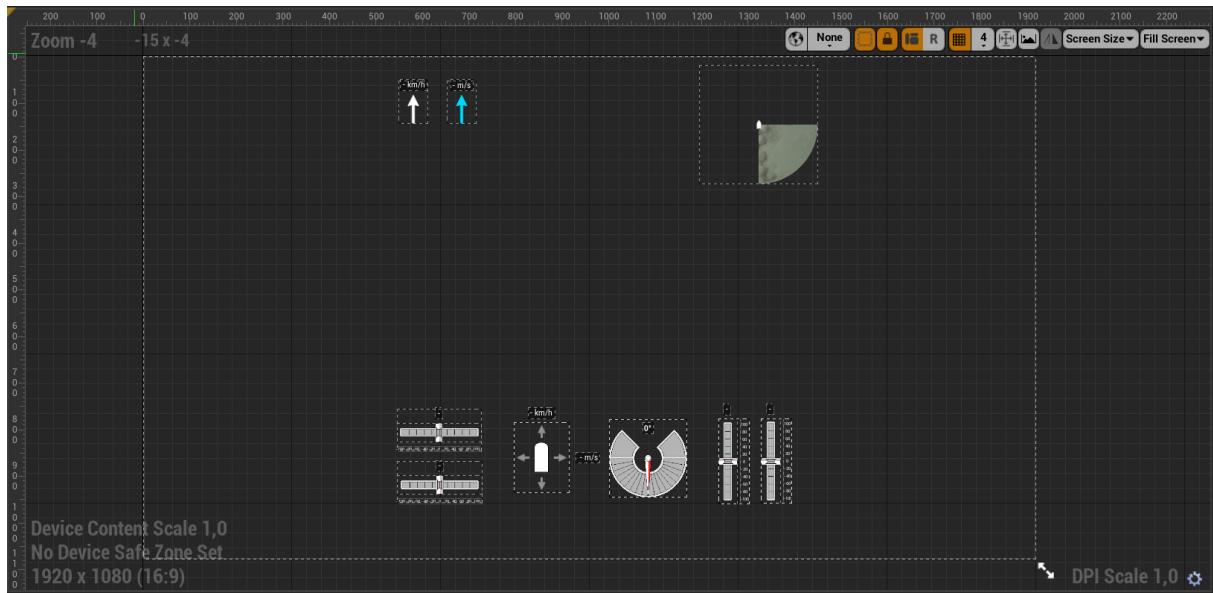


FIGURE 47 – Apparence finale de *WBP_LayoutPropeller_Tuto*.

Bien sûr, vous êtes libre d'obtenir une toute autre disposition des widgets.

Rappelez-vous cependant qu'il est nécessaire de bien utiliser les points d'ancrages et points de pivot pour éviter d'avoir des "effets de bord" indésirables.

6.2.2 Connecter le layout au HUD

Une fois le layout créé, il faut tout d'abord que celui-ci soit connecté à l'interface navigant de Navmer3D afin que l'on puisse l'invoquer, à posteriori.

Dans un premier temps donc, il faut nous rapprocher du blueprint **BP_HUD** :

1. Ouvrez le blueprint BP_HUD situé dans le sous-dossier *NavmerContent/Blueprints/HUD*.

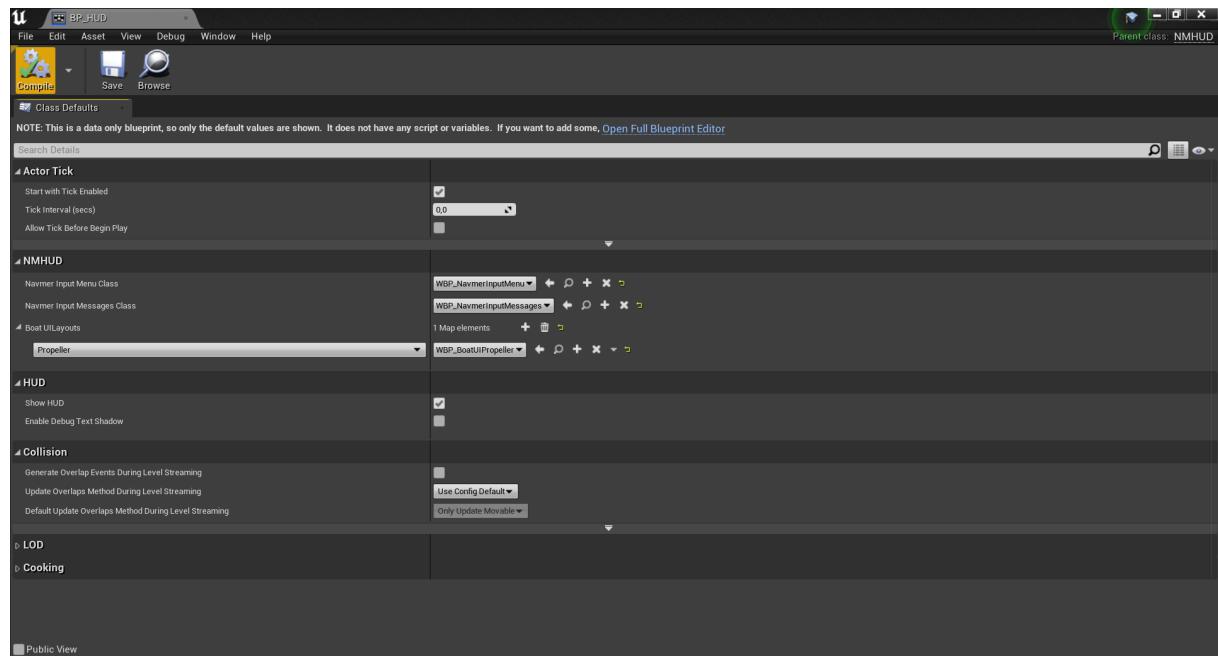


FIGURE 48 – Contenu de la fenêtre “ClassDetails” du blueprint BP_HUD.

Le blueprint BP_HUD, qui hérite de la classe code *NMHUD*, désigne un objet HUD (cf. [Objet HUD](#)).

Au travers de cet héritage : BP_HUD se comporte donc comme un “manager” de l’ensemble des interfaces graphiques ATH de Navmer3D.

Il gère l’état des différents widgets ATH de Navmer3D (i.e. les menus en simulation du plugin NavmerInput, mais aussi les layouts de l’interface navigant).

Le fait d’utiliser un blueprint comme BP_HUD permet d’assigner, depuis l’éditeur d’UE, des valeurs par défaut aux propriétés héritées de NMHUD : de cette manière, on déclare les variables dans la classe code parente et on définit ces variables dans le blueprint.

De plus, il est plus aisément d’assigner des références à des objets widgets ou blueprints dans un blueprint que depuis une classe code.

Via la fenêtre ClassDefaults de BP_HUD, il est possible de saisir les valeurs par défaut de BP_HUD.

Dans le cas de l'interface navigant, les propriétés qui nous intéressent se situent sous la rubrique "NMHUD".

 **NOTE :**

Les champs **NavmerInputMenuClass** et **NavmerInputMessagesClass** de la rubrique NMHUD sont réservés au plugin NavmerInput.

Vous pouvez remarquer la liste déroulante **BoatUILayouts**.

Ce champ, qui fonctionne comme un tableau associatif (i.e. une structure de donnée de type "Map"), permet d'affecter les **classes** de layouts pour **générer** les différents layouts à utiliser en simulation (suivant que l'on souhaite afficher une disposition "bateaux à propulsion par hélices", ou autre).

 **NOTE :**

L'unique type de layout existant actuellement dans Navmer3D (i.e. l'unique clé existante pour notre tableau associatif) est le type **Propeller**, qui attend une classe de layout pour une disposition "bateaux à propulsion par hélices".

Le type qui nous intéresse ici est le type Propeller, qui est pour le moment associé au layout WBP_BoatUIPropeller :

2. Modifiez la valeur du champ Propeller en la remplaçant par WBP_LayoutPropeller_Tuto (i.e. sélectionnez la liste déroulante WBP_BoatUIPropeller, recherchez WBP_LayoutPropeller_Tuto et sélectionnez-le).



FIGURE 49 – Le layout WBP_LayoutPropeller_Tuto prend la place du layout WBP_BoatUIPropeller pour le type Propeller.

De cette manière, ce n'est plus la classe de layout WBP_BoatUIPropeller qui est associée au type Propeller, mais la classe de layout WBP_LayoutPropeller_Tuto.

3. Compilez et sauvegardez BP_HUD.

À priori, à partir de ce moment là, si vous lancez la simulation sur la carte “Lyon” : vous devriez être en mesure de voir le layout WBP_LayoutPropeller_Tuto s’afficher à l’écran en simulation.



FIGURE 50 – Le nouveau layout WBP_LayoutPropeller_Tuto prend effet immédiatement en simulation.

Le layout affiché en jeu, vous êtes libre d’essayer de changer la taille de la fenêtre de jeu pour tester l’adaptabilité du layout au redimensionnement, mais aussi de corriger manuellement l’apparence et la disposition des widgets de WBP_LayoutPropeller_Tuto si cela ne vous convient pas.

6.2.3 Appeler l'interface navigant

À la fin de la section **Connecter le layout au HUD**, tout fonctionne d'office car l'interface navigant est déjà invoqué avec le type de layout Propeller dans la carte Lyon.

1. Ouvrez la carte Lyon.
2. Depuis la fenêtre principale de l'éditeur d'UE, ouvrez le blueprint du niveau Lyon (i.e. bouton *Blueprints > "Open Level Blueprint"* dans la barre principale).
3. Dans la fenêtre “Event Graph” du blueprint de Lyon, placez-vous au niveau de la boîte violette “Initialisation du HUD” qui part du noeud **EventBeginPlay**.

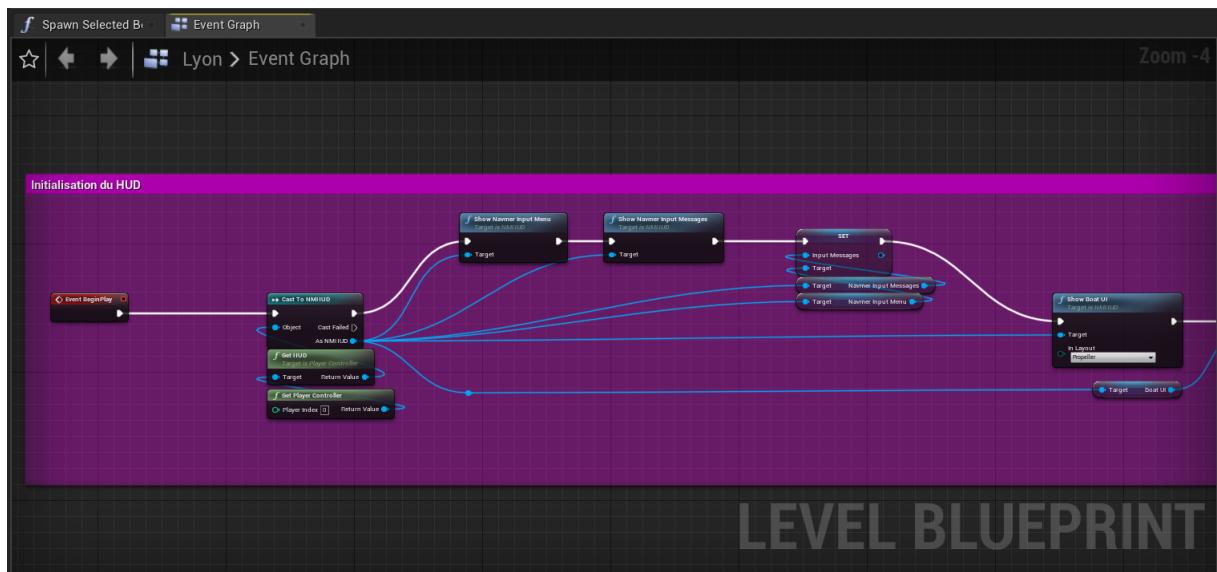


FIGURE 51 – Le graphe d'évènement du blueprint du niveau Lyon.

Sous “Initialisation du HUD” se trouve tout le fil d'exécution pour initialiser l'ATH de Navmer3D, lorsque le niveau Lyon est chargé.

NOTE :

L'ATH de Navmer3D est chargé dans un niveau de simulation pour ne pas interférer avec une autre interface graphique appelée dans un niveau dédié à un menu ou autre.

Deux choses sont à prendre en compte ici pour appeler l'interface naving avec le type de layout Propeller :

- Récuperer une référence sur l'objet BP_HUD du Player Controller unique de Navmer3D.

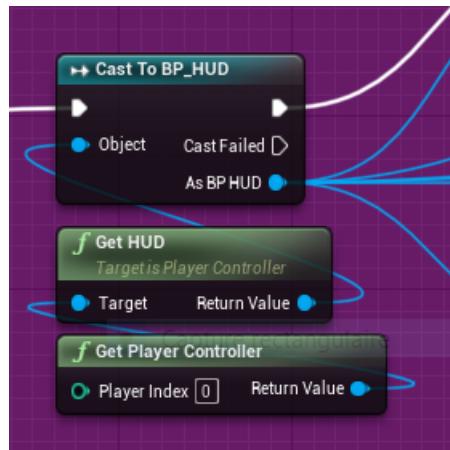


FIGURE 52 – De bas en haut : on récupère le Player Controller 0, on récupère sur le Player Controller 0 une référence sur HUD, puis on cast HUD en BP_HUD.

- Appeler la méthode **ShowBoatUI()** délivrée par BP_HUD pour invoquer l'interface naving avec le type de layout Propeller.

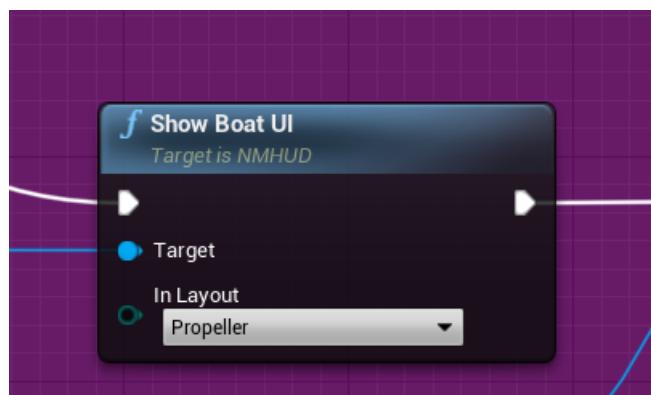


FIGURE 53 – La méthode ShowBoatUI(), “tirée” depuis la référence sur BP_HUD, est appelée avec le type Propeller.

Il est important de préciser que c'est **l'objet** BP_HUD qui est passé à la méthode ShowBoatUI() par référence via le pin d'entrée **Target**.

Lorsqu'on travaille avec le *visual scripting* d'Unreal Engine (i.e. via un graphe d'évènement) : un noeud de donnée est en fait **une référence** sur un objet. En l'occurrence ici, BP_HUD est une référence sur l'objet de même nom (la référence est obtenue via le pin de sortie "As BP_HUD" du noeud "Cast To BP_HUD").

Via sa méthode ShowBoatUI(), l'objet BP_HUD s'occupe alors de générer en simulation (i.e. lorsque le niveau Lyon est chargé) le **widget** WBP_LayoutPropeller_Tuto depuis la classe attachée à Propeller plus tôt (i.e. le **blueprint** WBP_LayoutPropeller_Tuto).

ATTENTION !

Dû à une erreur : sur le projet Navmer3D, le cast sur l'objet HUD du Player Controller est fait sur la classe code NMHUD au lieu du blueprint BP_HUD.

Puisque c'est dans BP_HUD que l'on a défini notre layout WBP_LayoutPropeller_Tuto pour la clé Propeller, il est préférable de faire la conversion dans ce sens pour éviter des comportements indésirables.

Cela fonctionne malgré tout, probablement car on incite Unreal Engine à utiliser BP_HUD par défaut pour tous les niveaux dans les paramètres globaux du projet, mais rien n'est sûr.

NOTE :

Le fil d'exécution sous "Initialisation du HUD" est à reproduire dans le graphe des blueprints de tous les niveaux de simulation pour appeler l'interface navigant.

6.2.4 Définir BP_HUD par défaut

Unreal Engine génère par défaut un objet HUD avec la classe code **HUD** (i.e l'objet HUD est de type **AHUD**).

Une référence vers l'objet HUD est alors passée au Player Controller.

Il faut donc faire en sorte qu'Unreal Engine génère par défaut un objet HUD depuis la classe blueprint **BP_HUD** au lieu de la classe code de base **HUD**.

De cette manière, le Player Controller sera assigné par défaut d'une référence sur un objet **BP_HUD** (qui hérite de la classe code **NMHUD**).

Cela permettra à Unreal Engine de savoir en runtime, dans le blueprint de niveau (p.ex. celui de Lyon) et dans le code C++, que l'HUD utilisé est **BP_HUD** : l'accès aux propriétés définies dans **BP_HUD** seront alors accessibles.

Pour cela : l'objet **GameMode** permet de spécifier différentes règles de jeu (i.e. le nombre de joueurs, les points d'apparition des joueurs, les transitions de niveaux, etc.), mais également les classes par défaut qui seront utilisées pour générer les objets Pawn, HUD et Player Controller.

Dans Navmer3D, il existe déjà un GameMode personnalisé via un blueprint : **BP_GameMode_V1**, situé dans *MenuSystem/GameModePlayercontroller/* :

1. Ouvrez le blueprint **BP_GameMode_V1**.

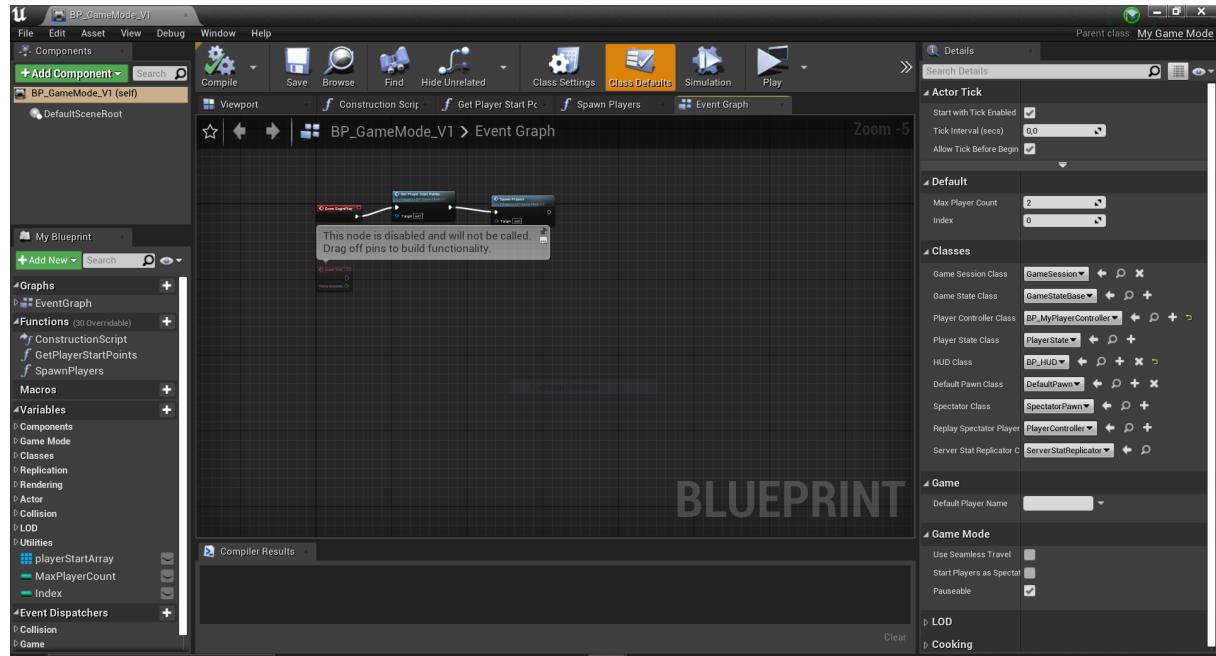


FIGURE 54 – Le GameMode principal de Navmer3D : **BP_GameMode_V1**.

Il faut s'assurer ici que c'est la classe BP_HUD qui sera utilisée par le GameMode BP_GameMode_V1 pour générer notre objet BP_HUD : c'est normalement déjà le cas.

Si vous sélectionnez dans BP_GameMode_V1 le composant marqué par "(self)" (i.e. dans le panneau "Components") et que vous consultez la rubrique "Classes" dans le panneau Details : vous pouvez remarquer que la valeur du champ "HUD Class" vaut "BP_HUD".

Pour terminer, il est nécessaire que le GameMode BP_GameMode_V1 soit assigné aux propriétés du niveau Lyon. Pour cela :

2. (Si ce n'est pas déjà fait) Ouvrez la carte Lyon.
3. Via la fenêtre principale de l'éditeur d'UE : ouvrez le menu "Settings" (icône d'engrenage dans la barre principale) et sélectionnez "World Settings".

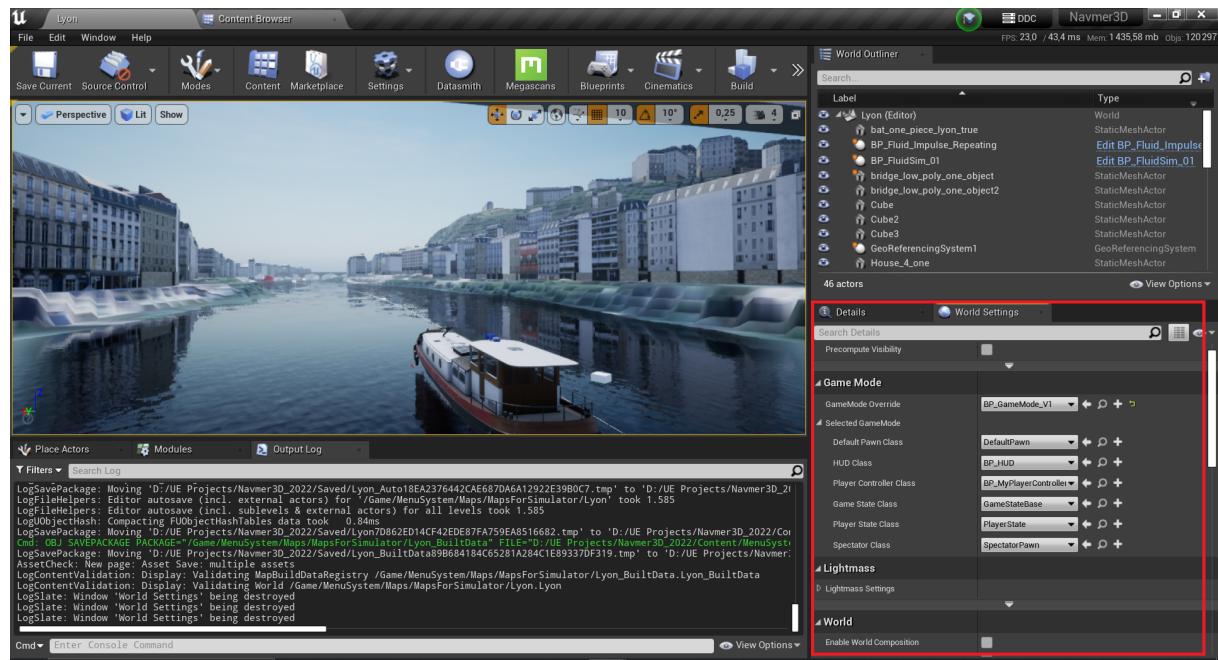


FIGURE 55 – Encadré en rouge : les propriétés de monde (i.e. de niveau).

4. Dans la fenêtre "World Settings", sous la rubrique "Game Mode" : assurez-vous que le champ "GameMode Override" ait pour valeur "BP_GameMode_V1".
5. Assurez-vous maintenant que le champ "HUD Class" ait également la valeur "BP_HUD".

Le GameMode original est alors "surcharge" par BP_GameMode_V1 (celui de Navmer3D), de même que la classe utilisée pour générer l'objet HUD.

 **NOTE :**

Pour chaque niveau, il sera nécessaire d'assigner BP_GameMode_V1 aux World Settings.

Assigner BP_GameMode_V1 aux World Settings est une manipulation “locale” : elle ne prend effet que dans le niveau Lyon (via ses World Settings).

Il est possible de surcharger le GameMode original par BP_GameMode_V1 de manière “globale” au projet Navmer3D (i.e. tous les niveaux auront cette configuration).

Pour cela, il faut se rendre dans la fenêtre *Project Settings > Project > Maps & Modes*, et assigner sous la rubrique “Defaults Modes” le GameMode BP_GameMode_V1.

Il est cependant préférable de n’impliquer que les niveaux de simulation (p.ex. comme la carte Lyon) via leurs World Settings, puisque l’ATH ne concerne que les niveaux de simulation.

6.2.5 Tester un layout

 **NOTE :**

Cette section optionnelle montre comment il est possible de tester les différents widgets d'un layout, sans passer par les structures d'états envoyées aux widgets.

Pour tester le comportement et l'animation des widgets d'un layout, il y a plusieurs manières possibles.

Lorsque le layout est connecté au HUD et que l'interface navigant est appelé, vous pouvez par exemple simplement profiter de l'échange des structures d'états avec les widgets en temps-réel.

Autrement, vous pouvez, depuis le graphe d'évènement du layout, passer des valeurs arbitraires aux widgets.

- Ouvrez le layout WBP_BoatUIPropeller, puis rendez-vous dans l'onglet Graph.

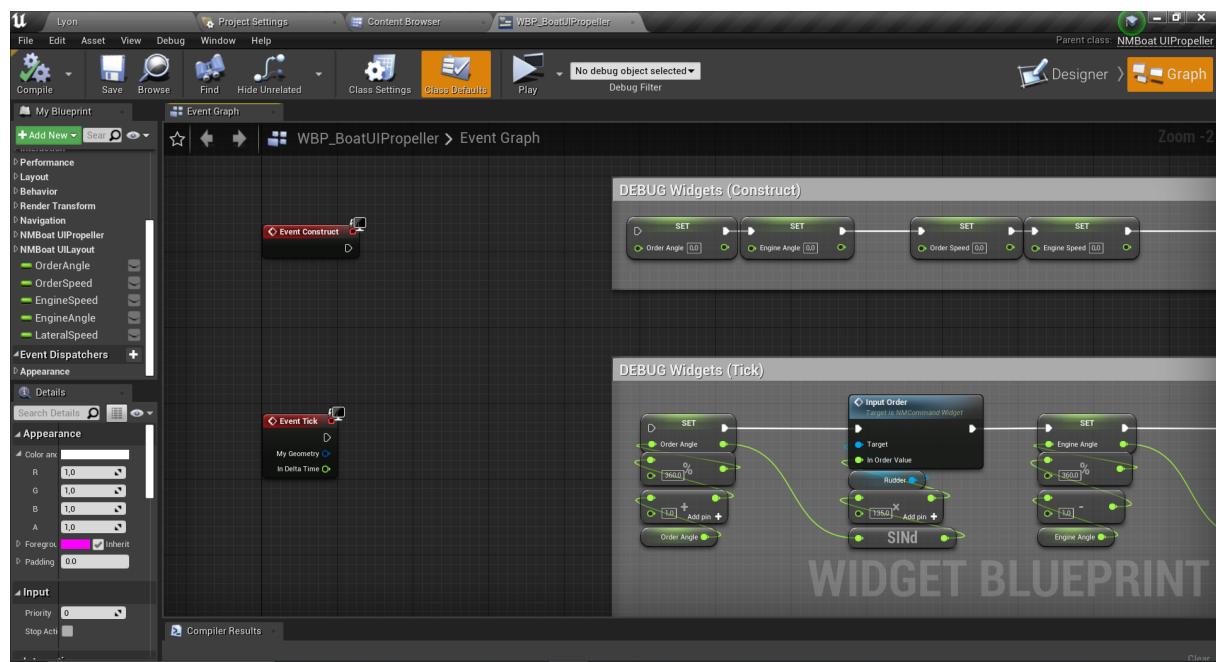


FIGURE 56 – Graphe d'évènement du layout WBP_BoatUIPropeller.

Dans son graphe, WBP_BoatUIPropeller dispose déjà d'un ensemble de noeuds d'instructions permettant de tester en simulation l'animation des différents widgets.

Le bloc d'instructions “DEBUG Widgets (Construct)” permet d'initialiser quelques variables créées, pour l'occasion, dans le graphe.

Il doit être connecté au noeud **EventConstruct**, qui est appelé lors de la phase de construction d'un widget (en l'occurrence ici, le layout WBP_BoatUIPropeller).

Les variables **OrderAngle**, **EngineAngle**, **OrderSpeed**, **EngineSpeed** et **LateralSpeed** vont nous permettre d'injecter des valeurs arbitraires respectivement aux widgets Rudder, LeftPropThrottle et BoatSpeedMeter.

Ces variables ont comme seul intérêt de tester l'animation des différents widgets souhaités.

Les variables OrderAngle, OrderSpeed et EngineSpeed sont également réutilisées pour injecter en plus des valeurs arbitraires aux widgets WindMeter et BowPropThrottle.

Le bloc d'instructions “DEBUG Widgets (Tick)” contient l'ensemble des instructions pour faire varier les valeurs des widgets dont on souhaite tester l'animation.

Il doit être connecté au noeud **EventTick** (ce noeud est appelé à chaque *frame*, i.e. toutes les 1/60 secondes ou 1/30 secondes selon le *framerate*).

Egalement, il est nécessaire que les widgets soient considérés comme des variables afin d'y avoir accès dans le graphe via référence. Pour cela, dans l'onglet Designer de WBP_BoatUIPropeller :

2. Sélectionnez le widget Rudder.
3. Assurez-vous que la case “IsVariable” est cochée (la case est située à côté du nom du widget, directement sous le panneau Details).
4. Faites de même avec tous les autres widgets.

Maintenant, vous pouvez, depuis le panneau “My Blueprint” de l’onglet Graph, retrouver les widgets et les placer à votre guise dans le graphe pour faire appel à leurs méthodes (les méthodes des widgets sont définies dans leur classe code parente).



FIGURE 57 – Test de comportement et d’animation du layout WBP_BotUIPropeller, en simulation.

 **NOTE :**

Il est bien sûr possible de tester localement les widgets depuis le graphe d’évènement de leur widget blueprint, plutôt que depuis un layout.

7 Maintenance Widgets

7.1 Hiérarchie des classes

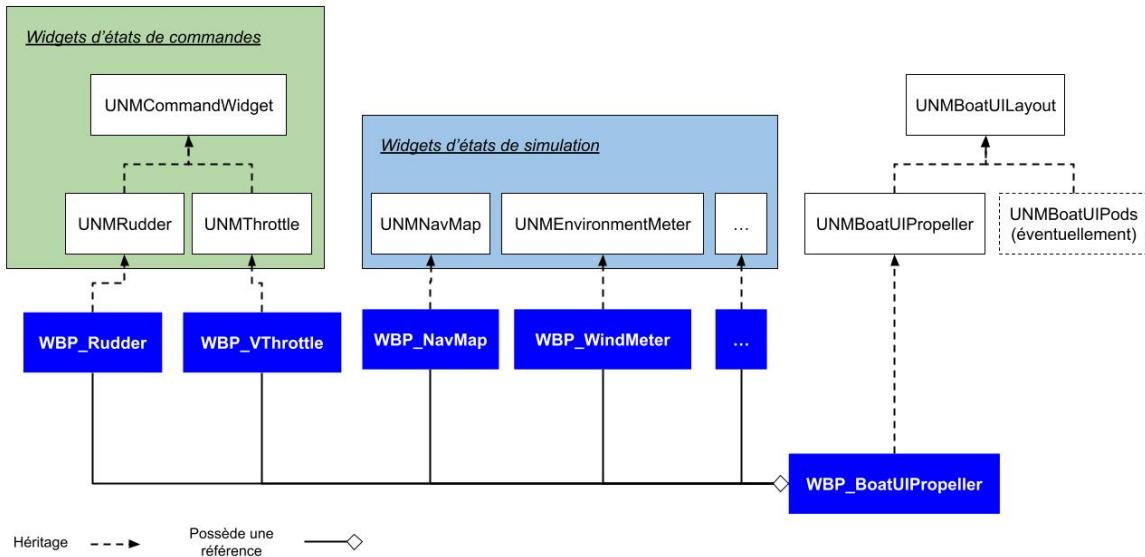


FIGURE 58 – Vue sur la hiérarchie des classes de l’interface navigant.

La figure ci-dessus montre globalement comment sont hiérarchisées les classes des widgets de l’interface navigant, avec tout en bas les widget blueprints.

On a d’un côté les widgets d’états de commandes, qui ont en commun la classe parente : **UNMCommandWidget**, et de l’autre côté les widgets d’états de simulation.

Les classes de layout ont toutes comme classe de base **UNMBoatUILayout**.

Chaque classe de layout (p.ex. **UNMBoatUIPropeller**) **implémente** sa manière de représenter à l’écran un état de la simulation ainsi que les états de commandes, suivant les widgets qui compose le layout.

Chaque classe de layout nécessite d’ailleurs un ensemble de références vers les widgets qui compose le layout. Les widgets sont fournis au travers du widget blueprint qui hérite de la classe de layout (cf. [Bindings de widgets](#)).

Enfin, toutes les classes de widgets héritent, directement ou indirectement, de la classe **UUUserWidget**. Cela permet d’une part d’accéder aux propriétés et fonctionnalités utiles à la personnalisation de widgets, et d’autre part d’avoir la possibilité de dériver une classe de widget en un widget blueprint.

7.2 Communication des classes

Les widgets de l'interface navigant s'animent suivant les valeurs qui leurs sont passées, via des *structures d'états*.

Les structures d'états sont déclarées dans le fichier *Data/NMStateData.h*.

Une structure **FSimState** contient les paramètres de simulation, tandis qu'une structure **FCommand-State** contient les paramètres de commandes pour un widget de commande.

Les structures d'états peuvent être créées au sein du programme depuis les classes code de Navmer3D, ou depuis un module externe gréffé au projet en tant que librairie *Third Party*.

NOTE :

L'objectif souhaité est que les paramètres renvoyés par la *libnavmer* soient encapsulés dans des structures d'états de simulation et de commandes.

Actuellement, deux exemples d'utilisations de structures d'états de commandes existent dans la classe code **AMyBoatPawnV2**, où seulement les inputs navigant (obtenus depuis le Player Controller unique de Navmer3D) sont envoyés :

- dans la méthode **Server_AddCustomForceForward_Implementation** (manette de propulsion gauche),
- et dans la méthode **Server_AddCustomForceRight_Implementation** (gouvernail).

Autrement, les structures d'états de simulation ne sont pas utilisées pour le moment et il n'existe pas d'exemples.

Egalement, la position géographique du navire (nécessaire à la minimap), qui pourrait être transmise via une structure d'état de simulation, est générée depuis le blueprint de niveau (cf. [Navmap et Carte ENC](#)).

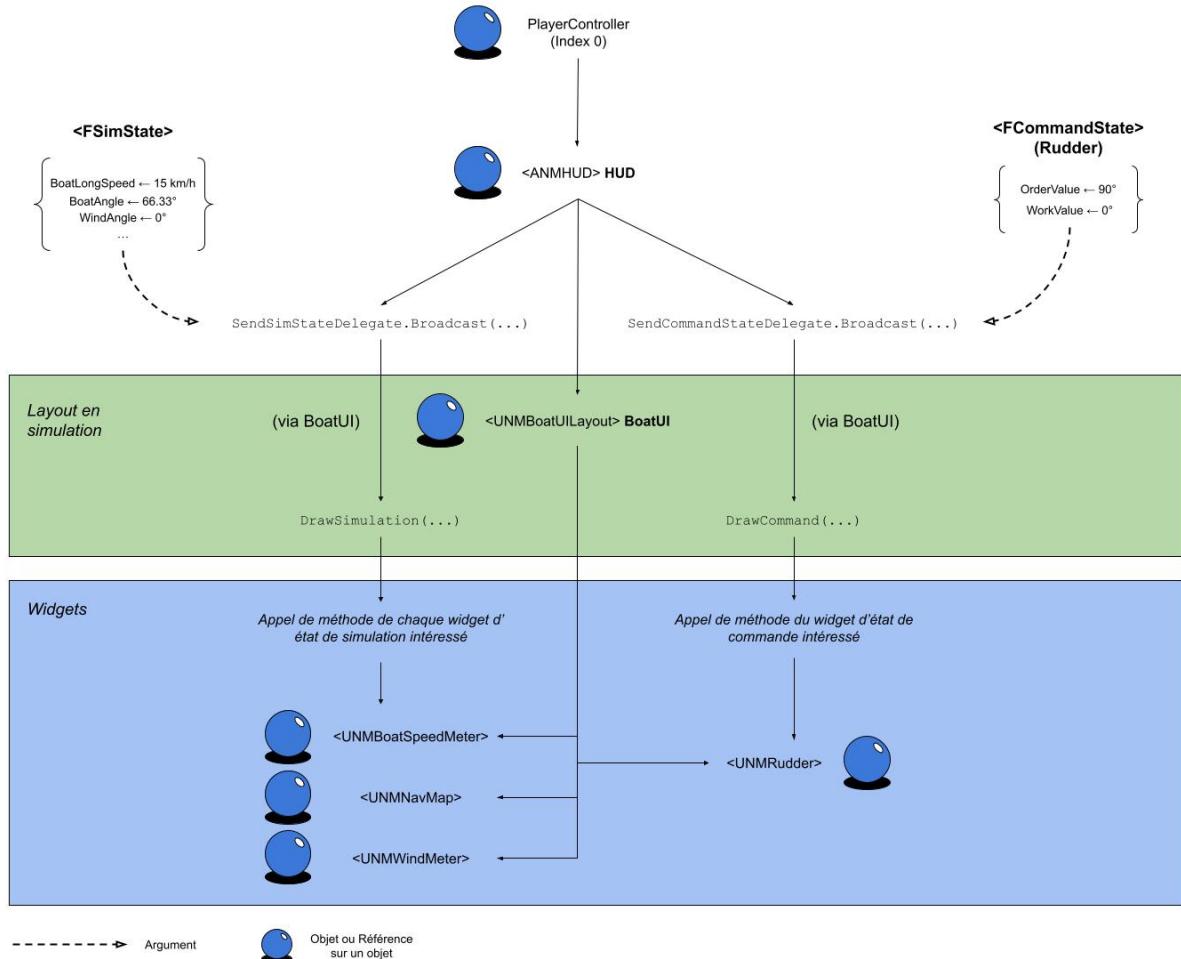


FIGURE 59 – Schéma de communication via l'utilisation de structures d'états.

Le schéma de communication fonctionne ainsi :

1. Une structure d'état est déclarée et définie avec ses paramètres.
2. La structure d'état est passée par référence à une des **Delegates** ¹⁸ (**SendSimStateDelegate** ou **SendCommandStateDelegate**) de l'objet NMHUD ou BP_HUD qui se charge, à la réception, de passer la structure d'état au layout attaché au viewport en simulation (attribut **BoatUI** de UNMHUD).
3. Le layout utilisé en simulation fait appel respectivement à son *implémentation de la méthode* ¹⁹ :
 - **DrawSimulation** si la structure d'état transmise est une structure d'état de simulation,
 - ou **DrawCommand** si la structure d'état transmise est une structure d'état de commande.
4. Enfin, les valeurs de la structure d'état sont passées aux différents widgets du layout via des appels à leurs méthodes (des implémentations de *mutateurs*) qui sont chargées d'animer ceux-ci.

18. Type dédié à la diffusion de données pour des objets ayant "souscrits" : <https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/ProgrammingWithCPP/UnrealArchitecture/Delegates/>

19. En utilisant la macro UPROPERTY avec l'argument **BlueprintNativeEvent**, la méthode peut être implémentée dans une classe enfant (virtual) ET être aussi appelée dans un blueprint qui hérite (impossible autrement).

7.3 Propriétés de PointerWidget

Le widget **PointerWidget**, défini dans le widget blueprint WBP_PointerWidget, est une base de widget qui sert à représenter un “pointeur” (i.e. une simple direction) personnalisable dans un widget blueprint extérieur.

Pour cela, il ne se sert que d'un simple widget *Image* pour afficher une texture quelconque pouvant représenter un pointeur, et de quelques paramètres hérités de la classe code **NMPointerWidget** pour affecter cette texture.

Le widget PointerWidget est utilisé par exemple dans le widget blueprint WBP_Rudder ou encore dans WBP_WindMeter et WBP_StreamMeter.

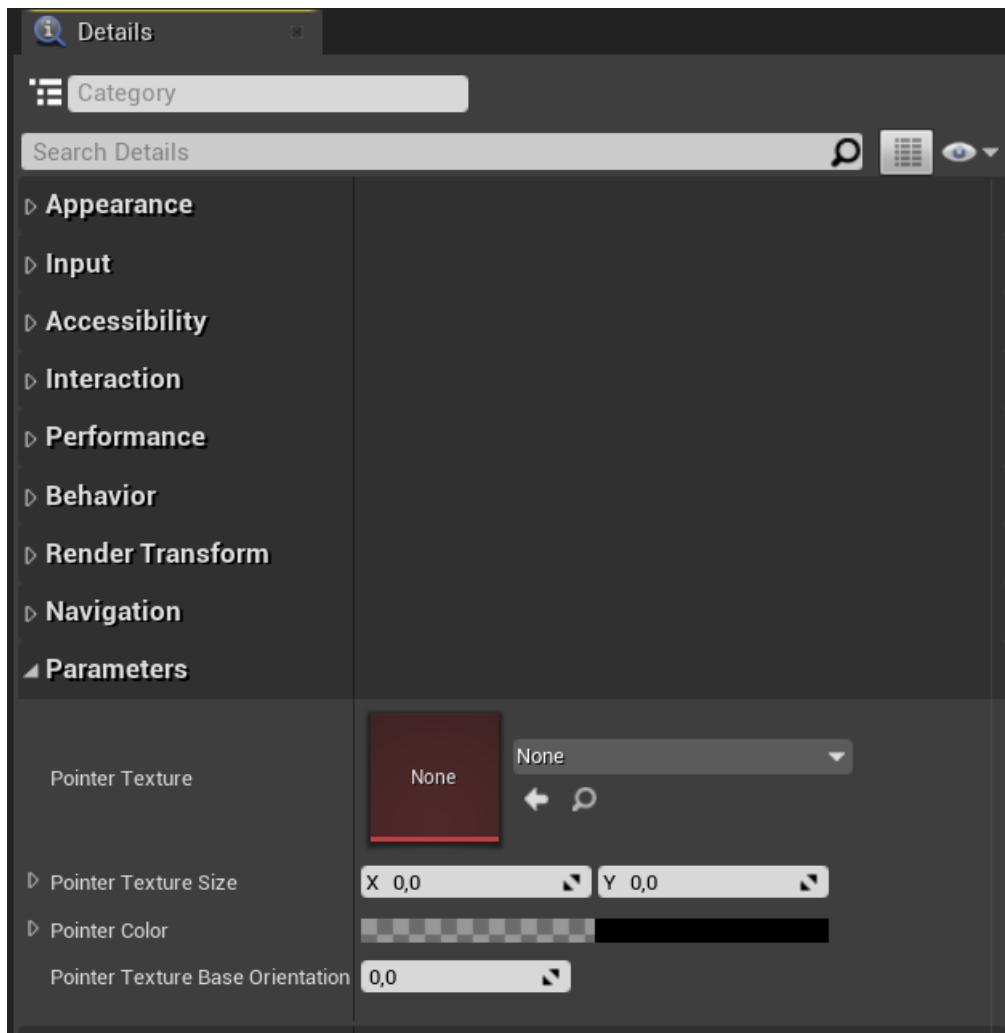


FIGURE 60 – Vue sur les paramètres hérités dans le widget blueprint WBP_PointerWidget.

Les paramètres hérités sont les suivants :

- **PointerTexture** : permet de définir une texture pour le pointeur
- **PointerTextureSize** : permet de définir les dimensions de la texture du pointeur
- **PointerColor** : permet de définir la couleur et l'opacité de la texture du pointeur
- **PointerTextureBaseOrientation** : permet de spécifier l'angle de départ du pointeur selon **l'apparence de la texture**

 **ATTENTION !**

L'identifiant de la variable *PointerTextureBaseOrientation* est assez ambigu.

Dans le cas où la texture représente une direction quelconque, l'intérêt de la variable *PointerTextureBaseOrientation* est de pouvoir redéfinir l'angle de départ en appliquant un **décalage sur la nouvelle valeur d'angle** afin que toute rotation du widget soit cohérente avec le cercle trigonométrique.

Par exemple, une texture d'une flèche qui pointe vers l'est a un angle de départ de 0° sur le cercle trigonométrique.

Cependant, si on prend une texture d'une flèche qui pointe vers le sud, l'angle de départ est de -90° (sens trigonométrique) : définir *PointerTextureBaseOrientation* à 90° permet d'appliquer un "décalage" et d'orienter le pointeur à 0° sur le cercle trigonométrique.

La méthode **OnAngleChanged_Implementation()** (située dans le fichier source *NMPointerWidget.cpp*), qui est appelée lorsque l'angle d'orientation de la texture change (Delegate), applique le décalage sur la rotation de la texture du pointeur.

7.4 Graphe WBP de WindMeter et StreamMeter

Les widgets **WindMeter** et **StreamMeter**, définis dans les widgets blueprints WBP_WindMeter et WBP_StreamMeter, sont des widgets d'états de simulation qui permettent d'indiquer respectivement la direction (exprimée en degrés) et la vitesse du vent (exprimée en km/h), et la direction du courant marin et sa force.

Les widgets blueprints ont en commun l'héritage de la classe code **NMEnvironmentMeter**.

Egalement, ils ont en commun une partie de la logique des widgets qui est définie dans l'onglet Graph.

Le fil d'exécution présent dans l'onglet Graph des deux widgets blueprints est identique.

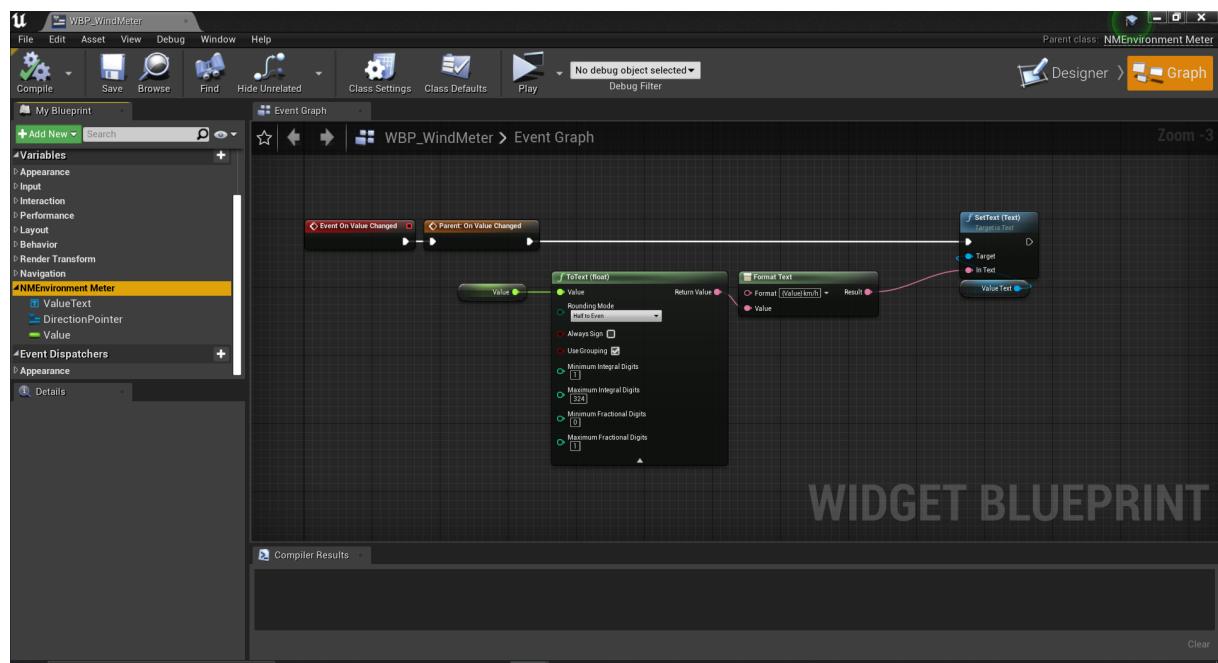


FIGURE 61 – Graphe du widget blueprint WBP_WindMeter (équivalent dans WBP_StreamMeter).

Ce fil d'exécution sert à formater la valeur de vitesse (ou de force) afin de l'afficher dans le widget **ValueText**.

NOTE :

Ce formatage peut tout aussi bien être réalisée dans la classe code parente.

Cependant, comme les widgets WindMeter et StreamMeter n'affichent pas les mêmes unités, il est préférable que le formatage soit réalisé dans le widget blueprint, ou bien dans une classe code parente qui hériterait de NMEnvironmentMeter.

Voici quelques explications sur les noeuds utilisés :

- Le noeud **Event OnValueChanged** exécute le fil lorsque la valeur de la propriété héritée **Value** est affectée (méthode **OnValueChanged()** attachée à la Delegate **OnValueChangedDelegate** de WBP_WindMeter, s'il existe une surcharge).
- Le noeud **Parent: OnValueChanged** spécifie que c'est la méthode de la classe parente que l'on souhaite utiliser (méthode **OnValueChanged()** attachée à la Delegate **OnValueChangedDelegate** de NMEnvironmentMeter).
- Le noeud **ToText (float)** génère une chaîne de caractères représentant un flottant fourni : en l'occurrence ici la valeur de l'attribut hérité *Value*.
- Le noeud **FormatText** effectue le formatage de la chaîne de caractères (à l'instar de la fonction *printf* dans le langage de programmation C).
- Le noeud **SetText (Text)** modifie la valeur du texte du widget *ValueText* avec la valeur de la chaîne de caractères précédemment formatée.



NOTE :

Afin d'ajouter un noeud “Event” personnalisé dans un blueprint pour une Delegate héritée d'une classe code, il est nécessaire que :

- la Delegate soit déclarée comme “Dynamic” et “Multicast” (macro DECLARE_DYNAMIC_MULTICAST_DELEGATE),
- la classe code parente ait la propriété **Blueprintable** (macro UCLASS(Blueprintable)),
- l'attribut associé au type de la Delegate ait la propriété **BlueprintAssignable** (macro UPROPERTY(BlueprintAssignable)),
- et enfin que la méthode attachée à la Delegate (et non l'implémentation de cette méthode) ait la propriété **BlueprintNativeEvent** (macro UFUNCTION(BlueprintNativeEvent)).

Après compilation du code C++ du projet, vous pouvez par la suite ajouter dans le graphe du blueprint un noeud évènement relié à votre Delegate qui porte le même nom et se comporte à l'identique (i.e. sous la liste déroulante “Add Event” après avoir fait un clic-droit souris dans le viewport du graphe).

7.5 NavMap et Carte ENC

NOTE :

Le fonctionnement de la minimap et les étapes de conversion des coordonnées navire sont expliqués dans la partie [Minimap](#).

Le widget **NavMap** correspond au widget de la minimap affichée à l'écran en simulation.

C'est un widget d'état de simulation qui nécessite principalement une texture de carte ENC (en initialisation), et la position géographique du navire contrôlé (en simulation).

L'apparence “par défaut” du widget est définie dans le Designer de **WBP_NavMap**.

Cette apparence change en fonction de la texture de carte ENC passée en paramètre du matériau instancié **MI_Map**.

Tout repose sur la programmation du matériau **M_Map** et les appels aux méthodes de la classe code **NMNavMap**.

Un focus sur certains aspects est cependant nécessaire, notamment comment est parsé un fichier Worldfile, ou bien où ce déroule la mise à jour de la position du navire sur la minimap.

7.5.1 Parser

L'utilisation d'une carte ENC dans Unreal Engine implique plusieurs choses :

- une texture ENC (i.e. une texture de carte ENC)
- la lecture d'un fichier texte WorldFile exterieur

Les textures ENC sont insérées manuellement dans le projet (cf. [Ajouter une texture](#)).

Concernant la lecture d'un fichier Worldfile, la méthode employée permet d'automatiser le processus pour chaque niveau Unreal (p.ex. Lyon). Elle consiste en :

1. Parser un fichier JSON qui répertorie l'ensemble des chemins vers les Worldfile, et qui associe chaque chemin à un nom de niveau (p.ex. la chaîne de caractères "Lyon")
2. Parser les données d'un fichier Worldfile pour un nom de niveau donné

NOTE :

Les fichiers textes et JSON ne sont ni visibles depuis la solution Visual Studio, ni visibles au sein du projet (i.e. via le Content Browser). Ils peuvent être consultés depuis l'explorateur Windows.

Le fichier JSON "paths.json" regroupant les chemins des Worldfile est situé dans le sous-dossier *NavmerData/NavMap*.

L'unique fichier Worldfile disponible (pour le niveau Lyon pour le moment) est situé dans le sous-dossier *NavmerData/NavMap/Lyon*.

Les étapes de lectures sont effectuées en "off" :

- Lorsque **l'instance de jeu** est initialisée pour l'étape **1**
- Lorsqu'un niveau est chargé pour l'étape **2**

L'instance de jeu (objet **GameInstance**) permet de stocker et de partager des données entre tous les niveaux.

L'étape 1 est réalisée depuis la définition de la méthode **Init()** de la classe code **MyPuzzlePlatformInstance**.

```

74 void UMyPuzzlePlatformInstance::Init()
75 {
76     Super::Init();
77
78     UE_LOG(LogTemp, Warning, TEXT("Found class %s"), *MenuClass->GetName());
79
80     // Charger les chemins des fichiers de données raster pour le widget NavMap
81     FString JsonFile;
82
83     if (UNMMiscHelpers::LoadFileToString(JsonFile, NAVMAP_PATHS)) {
84         // Fichier Json chargé
85         TSharedPtr<FJsonObject> jsonData_Ptr = MakeShareable(new FJsonObject);
86         TSharedRef<TJsonReader>> Reader = TJsonReaderFactory::Create(JsonFile);
87
88         if (!FJsonSerializer::Deserialize(Reader, jsonData_Ptr))
89             // Echec de la déserialisation
90             UE_LOG(LogNavmerInstance, Warning, TEXT("Could not deserialize file \"%s\"."), NAVMAP_PATHS);
91
92         if (jsonData_Ptr.IsValid()) {
93             // Enregistrer les chemins de fichiers dans RasterDataPaths
94             for (auto It = jsonData_Ptr->Values.CreateConstIterator(); It; ++It) {
95                 RasterDataPaths.Add(It->Key, It->Value->AsString());
96             }
97         }
98
99         jsonData_Ptr = nullptr;
100    }
101    else {
102        UE_LOG(LogNavmerInstance, Warning, TEXT("Could not load file \"%s\"."), NAVMAP_PATHS);
103    }
104 }

```

FIGURE 62 – Contenu de la méthode **Init()** avec, encadré en rouge, le bloc dédié à la lecture des chemins.

Pour lire un fichier JSON, il faut inclure au projet le module natif à Unreal Engine “Json”. Le module est à priori déjà inclus, cependant pour s’assurer que c’est le cas :

- Ouvrez le fichier **Navmer3D.Build.cs**
- Inspectez la ligne **PublicDependencyModuleNames.AddRange(new string[] { ... });**
- Si la ligne contient la chaîne “Json” : le module peut être alors utilisé dans le projet
- Optionnellement, vous pouvez ajouter le module “**JsonUtilities**” pour accéder à des fonctions utilitaires

Les chaînes de caractères des chemins Worldfiles sont stockés dans le tableau associatif **RasterDataPaths** du GameInstance MyPuzzlePlatformInstance, par nom de niveau.

NOTE :

La clé du tableau associatif doit être identique au nom de niveau.

Par exemple, la clé pour le niveau ***MenuSystem/Maps/MapsForSimulator/Lyon.umap** sera “Lyon”.

L'étape 2 est réalisée depuis la définition de la méthode **ChangeMap()** de la classe code **NMNavMap**.

Concrètement, depuis le GameInstance (accessible avec la classe helper UE **UGameplayStatics**) :

- RasterDataPaths est récupéré
- On récupère, depuis RasterDataPaths, le chemin du Worldfile avec le nom de niveau donné en argument de ChangeMap()
- On fait appel à la fonction utilitaire Navmer3D **LoadRasterData()** pour parser les données du Worldfile
- La matrice RasterData, regroupant les données de transformations du Worldfile, est alors récupérée

La conversion des coordonnées projetées relatives en coordonnées raster ce fait à partir de la matrice RasterData.

Cependant, cette matrice permet de faire la conversion inverse : coordonnées raster en coordonnées projetées.

Il faut donc réaliser les transformations inverses pour obtenir les coordonnées désirées. Ces transformations géométriques sont réalisées avec la fonction utilitaire Navmer3D **Conv_GeoCoordsToRaster()**.

7.5.2 Configuration

Maintenant qu'il est possible de parser les données de transformations d'un Worldfile en fournissant un nom de niveau, il ne reste plus qu'à faire appel à la méthode ChangeMap() quelque part.

En outre, ChangeMap() est appelée dans le blueprint d'un niveau, pour paramétrer le widget NavMap lors de l'initialisation du niveau.

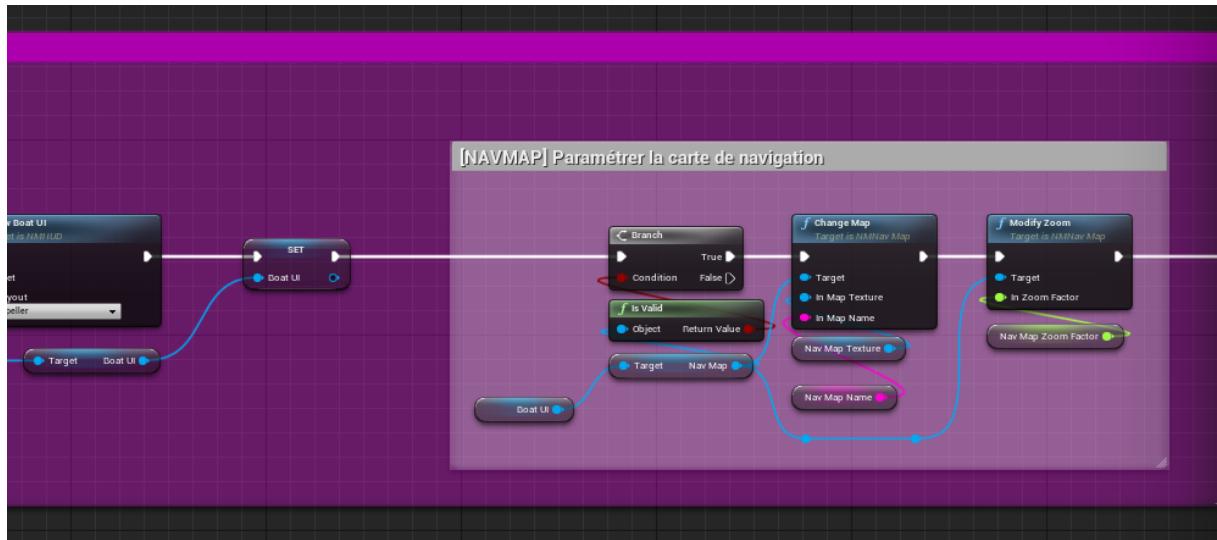


FIGURE 63 – Initialisation du widget NavMap dans le blueprint de niveau de Lyon.

Le bloc sur fond blanc effectue la configuration initiale du widget NavMap.

Tout le fil d'exécution qui précède ce bloc est rattaché au noeud “Event BeginPlay” : il s'execute donc une seule fois dès que le niveau est chargé.

Le noeud **Branch** ici permet de contrôler si une référence au widget NavMap existe dans le layout BoatUI récupéré auparavant (i.e. dans le layout à afficher en simulation).

Si tel est le cas : on fait appel à la méthode ChangeMap() de NavMap et on lui donne en entrées :

- le nom du niveau (noeud type string **NavMapName**)
- la texture ENC à afficher (noeud type texture **NavMapTexture**)

On appelle ensuite la méthode **ModifyZoom()** pour définir l'échelle d'affichage de la minimap en fournissant un flottant (**NavMapZoomFactor**).

7.5.3 Mise à jour

La mise à jour de la minimap consiste simplement à passer en temps-réel les coordonnées raster du navire au matériau instancié MI_Map, qui est chargé alors de l'affichage en bonne est due forme de la minimap.

Toujours dans le blueprint d'un niveau (p.ex. celui de Lyon), la mise à jour des coordonnées navire dans la minimap se fait dans le fil d'exécution "Event Tick".

Egalement, c'est dans ce fil d'exécution que la conversion des coordonnées "world" du navire en coordonnées projetées relatives est réalisée.

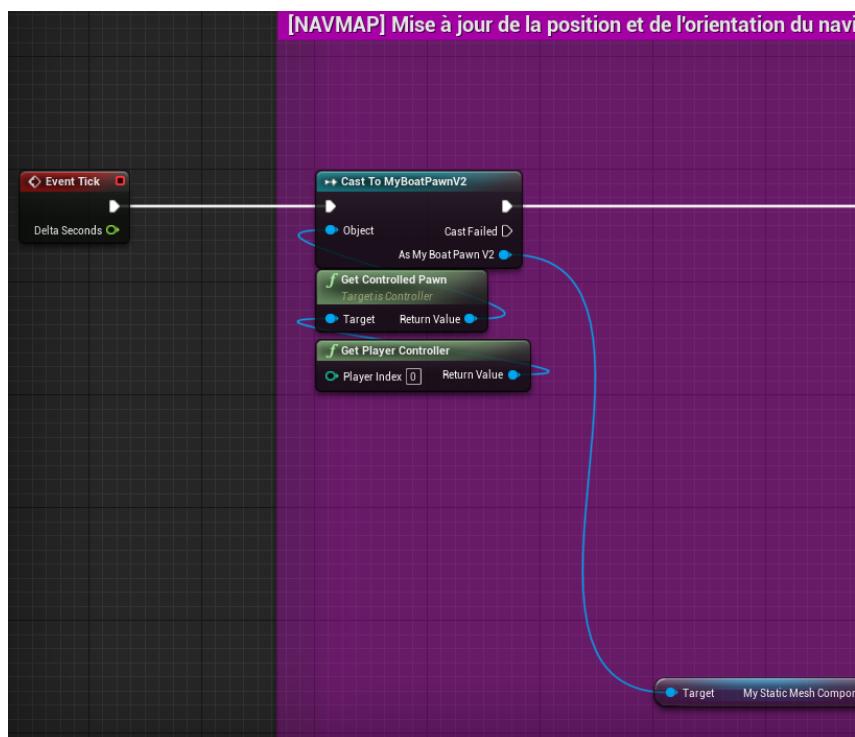


FIGURE 64 – Première partie du fil d'exécution de mise à jour de la minimap.

Ce fil est executé pour chaque *frame* en simulation.

On récupère tout d'abord la référence au **Pawn** du navire contrôlé de manière à pouvoir accéder à son *transform* (i.e. ses informations géométriques dans l'espace), via référence à son **MeshComponent** (noeud **MyStaticMeshComponent**).

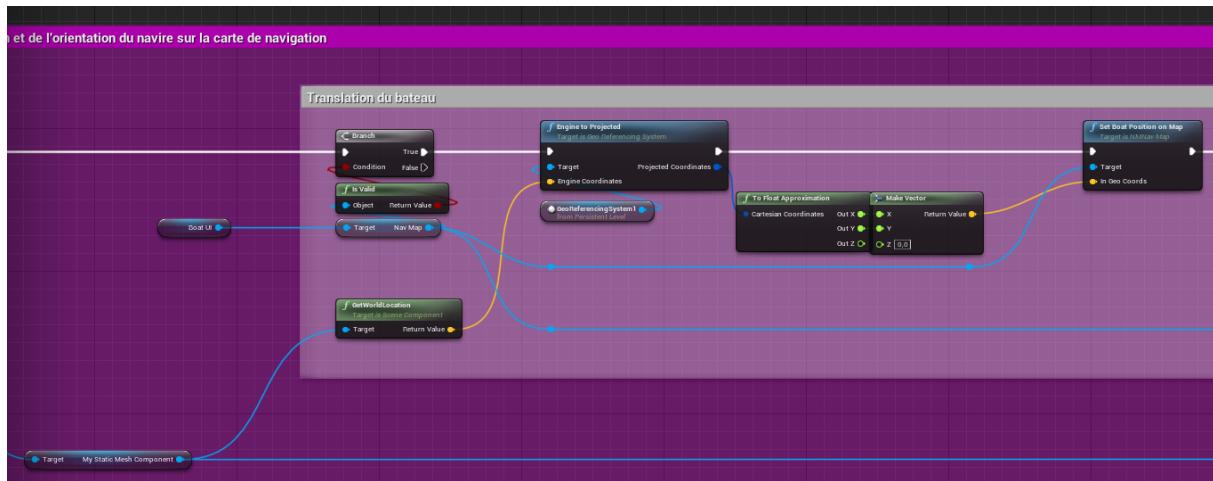


FIGURE 65 – Translation du bateau sur la minimap.

On récupère par la suite une référence au widget NavMap depuis BoatUI (on contrôle son existence avec le noeud Branch), et la position du navire avec le noeud **GetWorldLocation()** via MyStaticMesh-Component.

Viens alors la conversion des coordonnées “world” du navire en coordonnées projetées relatives.

Via une référence à l’objet **GeoReferencingSystem1** (présent dans la scène du niveau) : on appelle le noeud **EngineToProjected()** en passant en argument la position du navire donnée par GetWorldLocation().

EngineToProjected() donne alors les coordonnées projetées relatives du navire sous forme de structure de données.

Avant de passer ces coordonnées à la méthode **SetBoatPositionMap()** (noeud tirée depuis la précédente référence au widget NavMap), on casse la structure de données pour récupérer uniquement les composantes X (horizontale) et Y (verticales) sous forme de **Vector2** (seules les coordonnées 2D sont nécessaires pour la minimap).

Le Vector2 contenant les coordonnées X et Y projetées relatives du navire est alors passé à la méthode SetBoatPositionMap(), qui à son tour est chargée de convertir les coordonnées projetées relatives en coordonnées raster, puis de passer les coordonnées raster du navire au matériau instancié (cf. fichier *NMNavMap.cpp*).

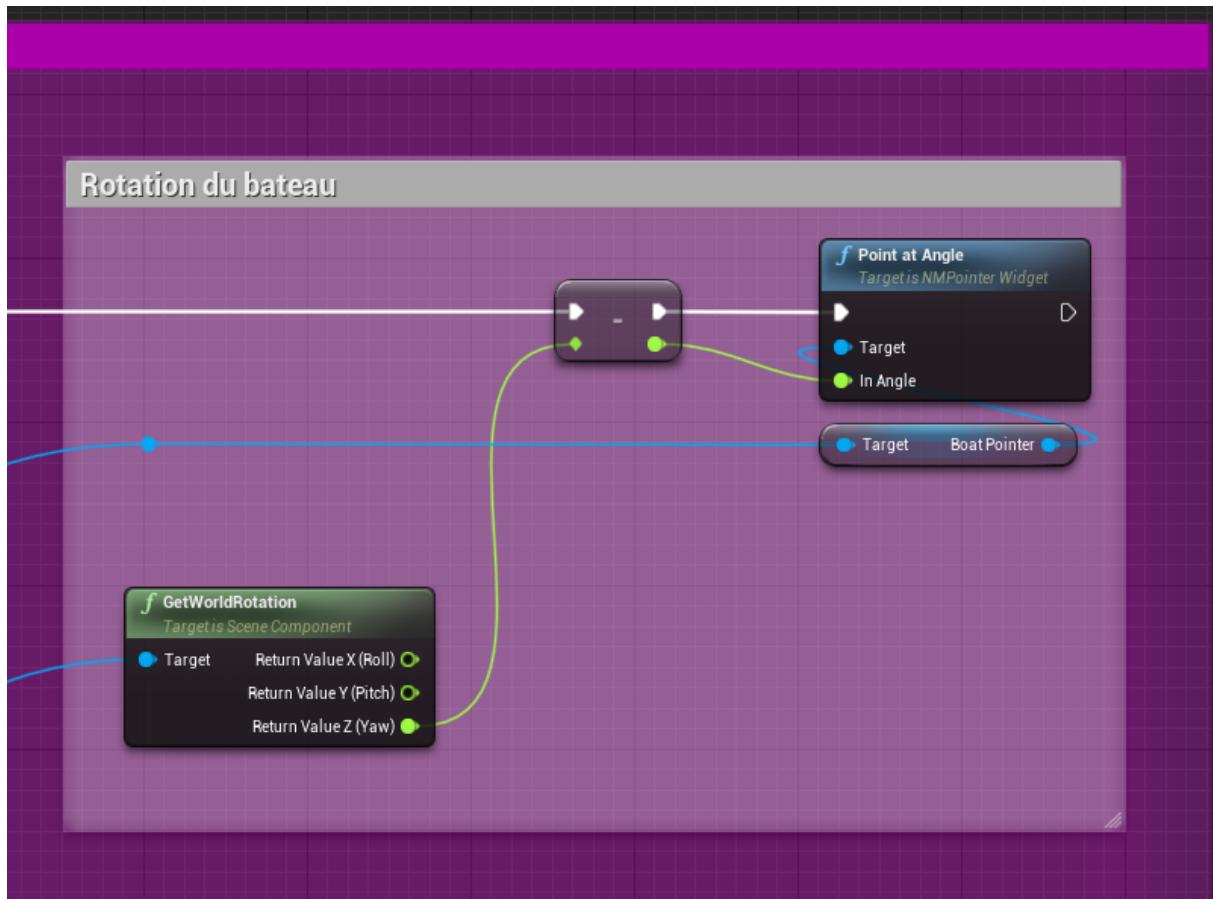


FIGURE 66 – Rotation du bâteau sur la minimap.

Enfin, il ne reste plus que la rotation du navire sur la minimap.

Comme évoqué dans la partie [Minimap](#), c'est la texture qui est translatée et non le navire.

Concernant la rotation du navire, un widget **PointerWidget** est utilisé en parallèle du matériau.

Via **MyStaticMeshComponent**, on récupère l'angle d'embardée du navire (i.e. la composante "Yaw") avec le noeud **GetWorldRotation()**.

On passe ensuite cet angle à la méthode **PointAtAngle()**, obtenue depuis une référence au widget **PointerWidget** de NavMap (**BoatPointer**), qui s'occupe de faire la rotation du pointeur du widget Nav-Map.

Le noeud "**-**" inverse l'angle d'embardée car la méthode **PointAtAngle()** effectue les rotations en sens trigonométrique (ce qui n'est pas le cas de base).

8 Améliorations

Différentes améliorations peuvent être apportés à l'interface de navigation de Navmer3D.

Principalement, on peut noter l'ajout de nouveaux layouts pour les autres types de manoeuvres (p.ex. la propulsion avec pods azimutaux, où le navire ne nécessite pas de gouvernail), et l'ajout d'un binding sur un bouton pour changer à tout moment de layout (p.ex. pour passer d'un layout pour bateaux à hélices à un layout pour bateau à pods azimutaux).

Egalement, après avoir greffer la **libnavmer** au projet, il faudrait faire passer dans les structures d'états (*CommandState*) les valeurs physiques des commandes calculées par la libnavmer pour l'animation des widgets d'états de commandes.

Bien que le zoom du navire sur la minimap est paramétrable, l'icône du navire ne change pas d'échelle. De manière à correspondre avec les dimensions du navire dans la scène (et donc des dimensions du navire sur la carte ENC), il serait intéressant que l'icône change également de taille en fonction du facteur de zoom.

Enfin, de manière optionnelle : séparer davantage le contenu de Navmer3D dans de nouveaux sous-dossiers.

9 Sources utiles

- “The Book of Shaders”. Guide en français sur les Fragment Shaders (utile pour la programmation de matériaux UE) : <https://thebookofshaders.com/?lan=fr>
- “The Book of Shaders”. Partie explications sur les shaders et la programmation parallèle GPU : <https://thebookofshaders.com/01/?lan=fr>
- “The Book of Shaders”. Partie du guide sur le dessin de formes (utile pour traiter les bords de la minimap dans le matériau M_Map) : <https://thebookofshaders.com/07/?lan=fr>
- Un site web de référence sur la création d’UI avec Unreal Engine : <https://benui.ca/>
- Introduction au développement d’interfaces graphiques avec Unreal Engine : <https://benui.ca/unreal/ui-introduction/>
- Méthodologie pour mettre en place son UI : <https://benui.ca/unreal/ui-setup/>
- Tutoriel vidéo sur la création d’une jauge à partir d’une texture et d’un matériau : <https://www.youtube.com/watch?v=0bLBQXP9pWY>
- Série de tutoriels vidéos sur la création d’une minimap à partir d’une texture et d’un matériau (“Advanced”) : https://www.youtube.com/watch?v=Z6qzaT4ZOho&list=PL4G2bSPE_8unhcUuYl6fNPnSNCyJhCPDl&index=2
- Référence HLSL pour la syntaxe (création de noeuds “custom” dans l’éditeur de matériaux) : <https://learn.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl-language-syntax>
- Géoréferencer un niveau et explications sur les différents systèmes de coordonnées : <https://docs.unrealengine.com/4.27/en-US/BuildingWorlds/Georeferencing/>
- Documentation officielle sur les matériaux instanciés : <https://docs.unrealengine.com/4.27/en-US/RenderingAndGraphics/Materials/MaterialInstances/>
- Article sur l’utilisation des interfaces dans les classes code : <https://unrealistic.dev/posts/the-many-faces-of-interfaces-in-unreal-engine-4>
- Introduction aux Delegates : <https://benui.ca/unreal/delegates-intro/>

Notes de pieds de pages

- [Shaders, p14] Page wikipédia sur le pipeline graphique : https://en.wikipedia.org/wiki/Graphics_pipeline
- [Shaders, p15] Article sur le fonctionnement du GPU : <https://cvw.cac.cornell.edu/GPUArch/threadcore>
- [Utilisation des matériaux dans Unreal Engine, p22] Référence HLSL : <https://learn.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl>
- [Mapping des coordonnées, p27] Page wikipédia sur le World file et son fonctionnement : https://en.wikipedia.org/wiki/World_file
- [Convention de nommage, p35] Documentation sur le nommage usuel des assets : <https://github.com/Allar/ue5-style-guide>
- [Convention de nommage, p35] Convention de nommage du code Unreal Engine : <https://docs.unrealengine.com/4.27/en-US/RenderingAndGraphics/Materials/MaterialInstances/>

9 SOURCES UTILES

- unrealengine.com/4.27/en-US/ProductionPipelines/DevelopmentSetup/CodingStandard/
- [Créer un Material et un Instanced Material, p42] Documentation des différents types de données de l'éditeur de matériau : <https://docs.unrealengine.com/5.1/en-US/material-data-types-in-unreal-engine/>
- [Héritage, p50] Documentation d'une liste assez exhaustive des UPROPERTY : <https://benui.ca/unreal/uproperty/>
- [Héritage, p50] Choisir entre un projet blueprint et un projet C++ : <<https://docs.unrealengine.com/4.27/en-US/Resources/SampleGames/ARPG/BalancingBlueprintAndCPP/>
- [Bindings de widgets, p70] Documentation et tutoriel sur la macro BindWidget : <https://benui.ca/unreal/ui-bindwidget/>
- [Communication des classes, p93] Documentation officielle sur les Delegates : <https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/ProgrammingWithCPP/UnrealArchitecture/Delegates/>

10 Annexes

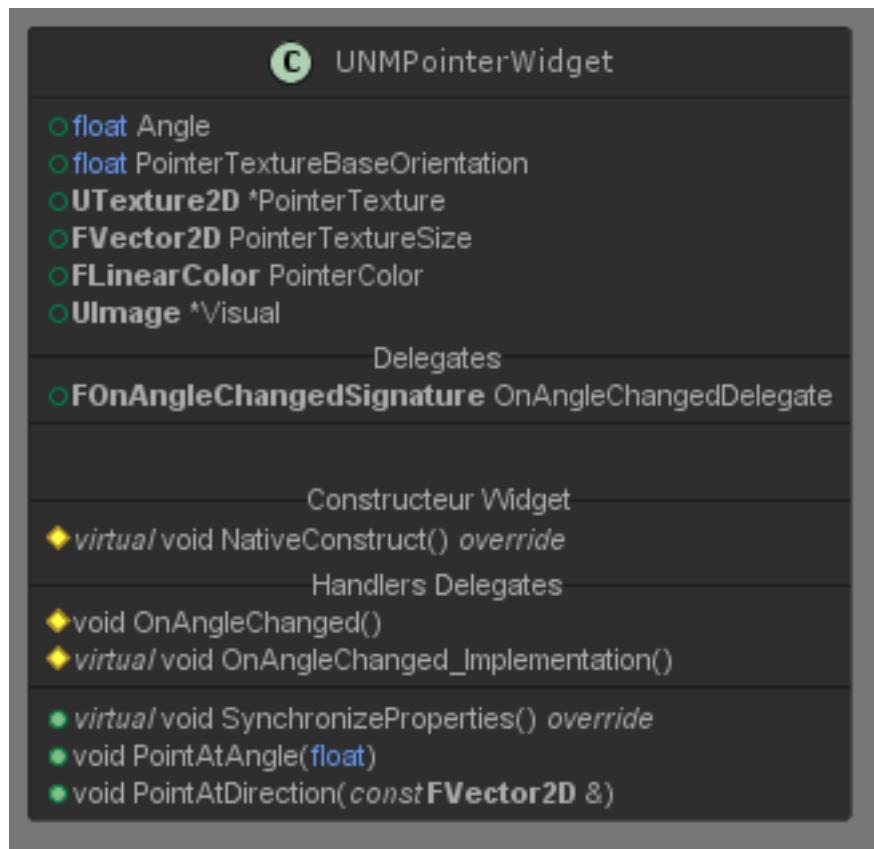


FIGURE 67 – Diagramme de classe de NMPointerWidget.

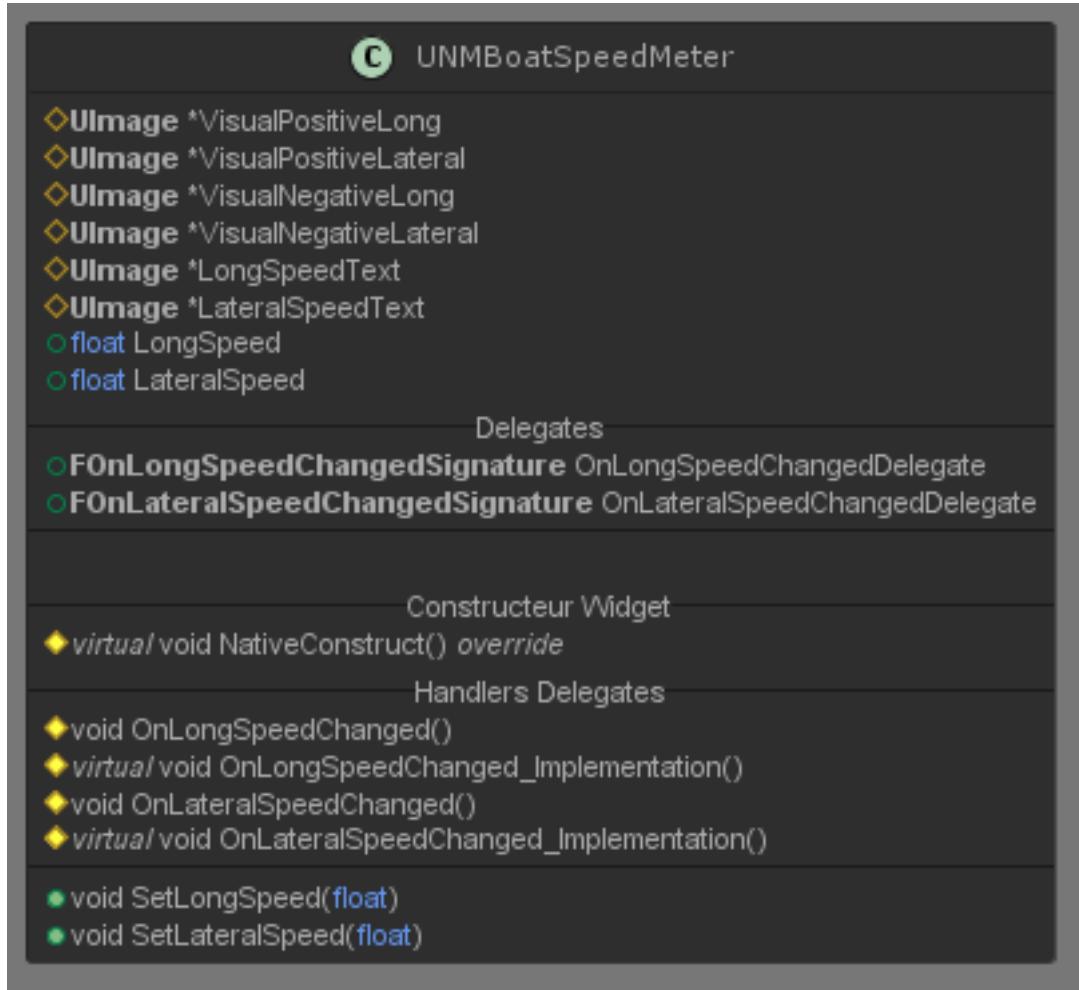
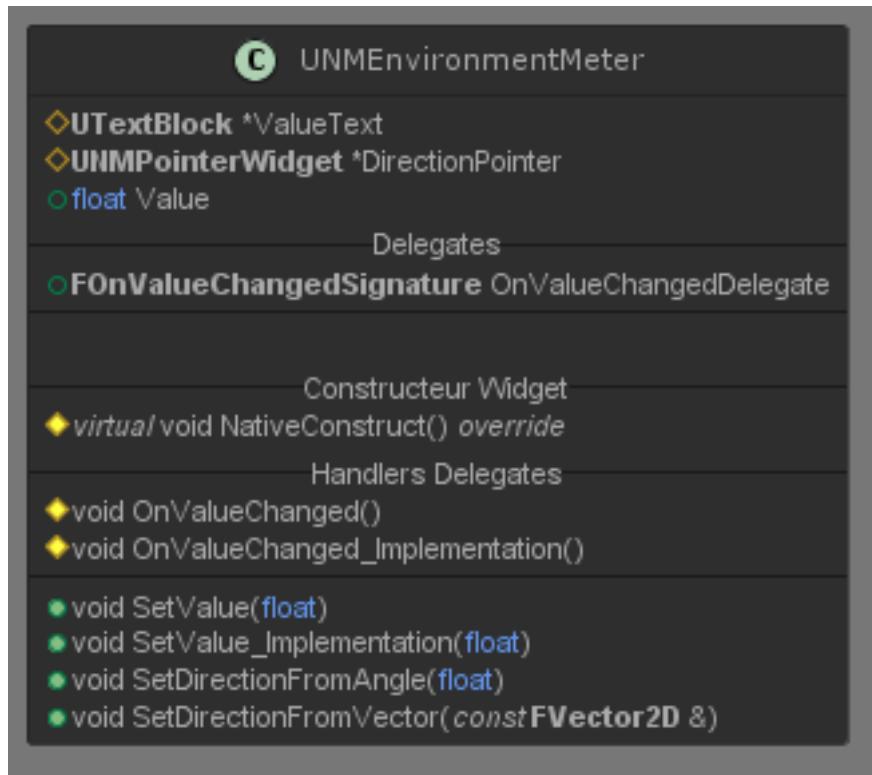
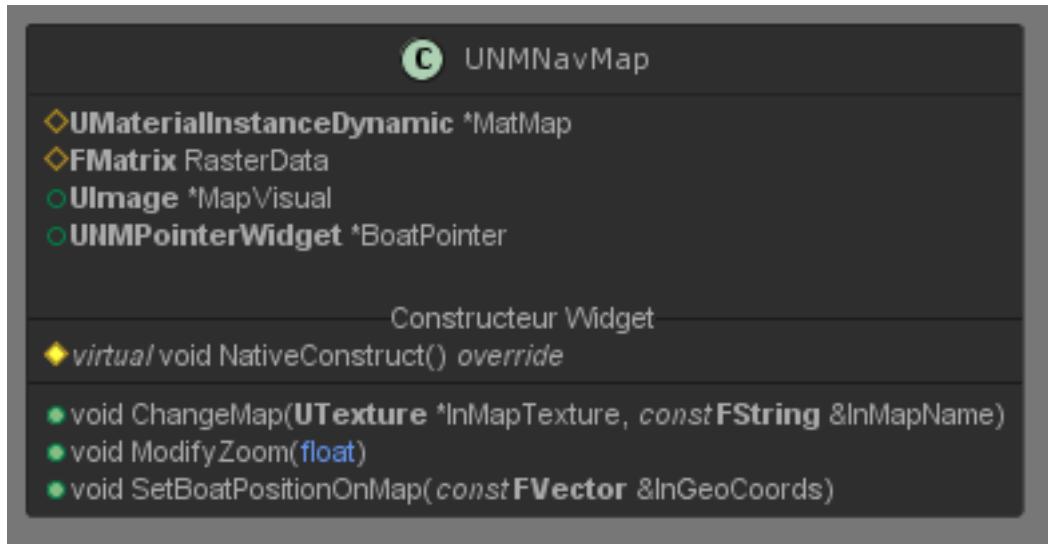


FIGURE 68 – Diagramme de classe de NMBoatSpeedMeter.

**FIGURE 69** – Diagramme de classe de NMEnvironmentMeter.**FIGURE 70** – Diagramme de classe de NMNavMap.

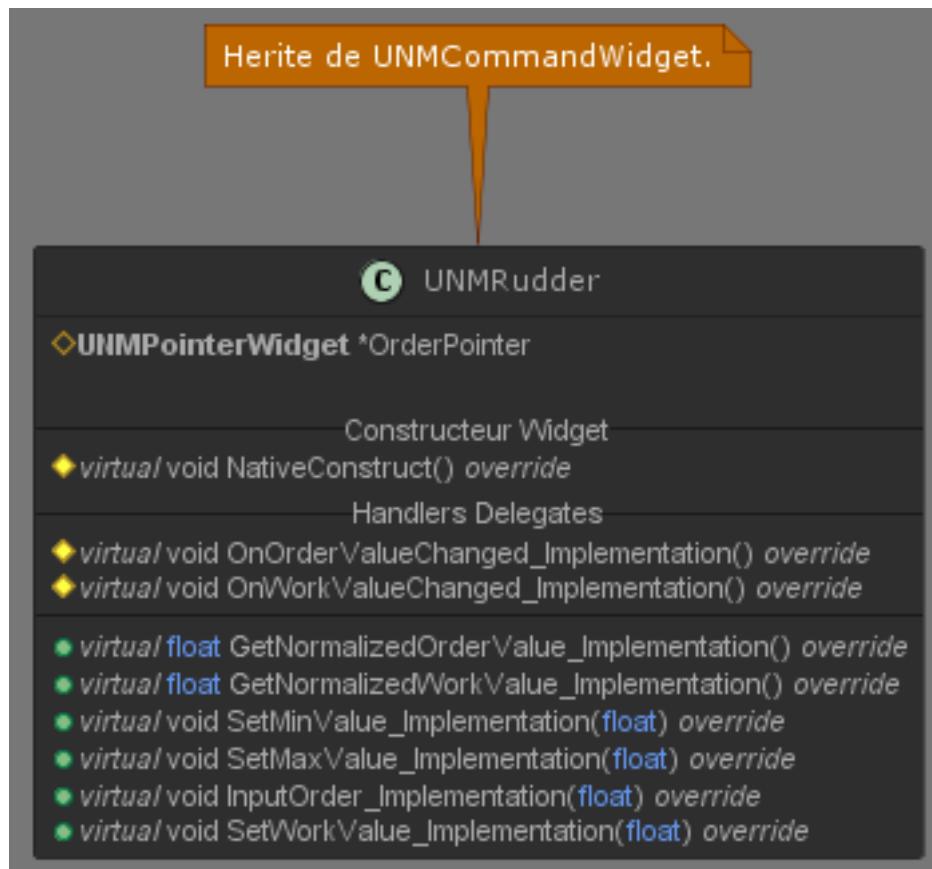


FIGURE 71 – Diagramme de classe de NMRRudder.

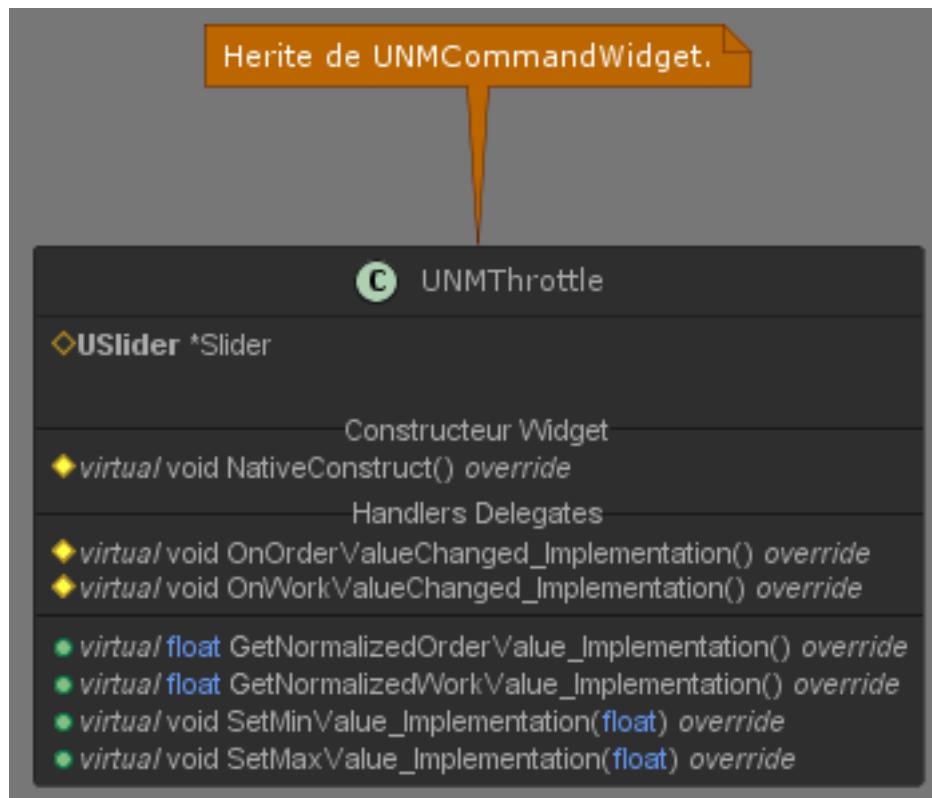


FIGURE 72 – Diagramme de classe de NMThrottle.

10 ANNEXES

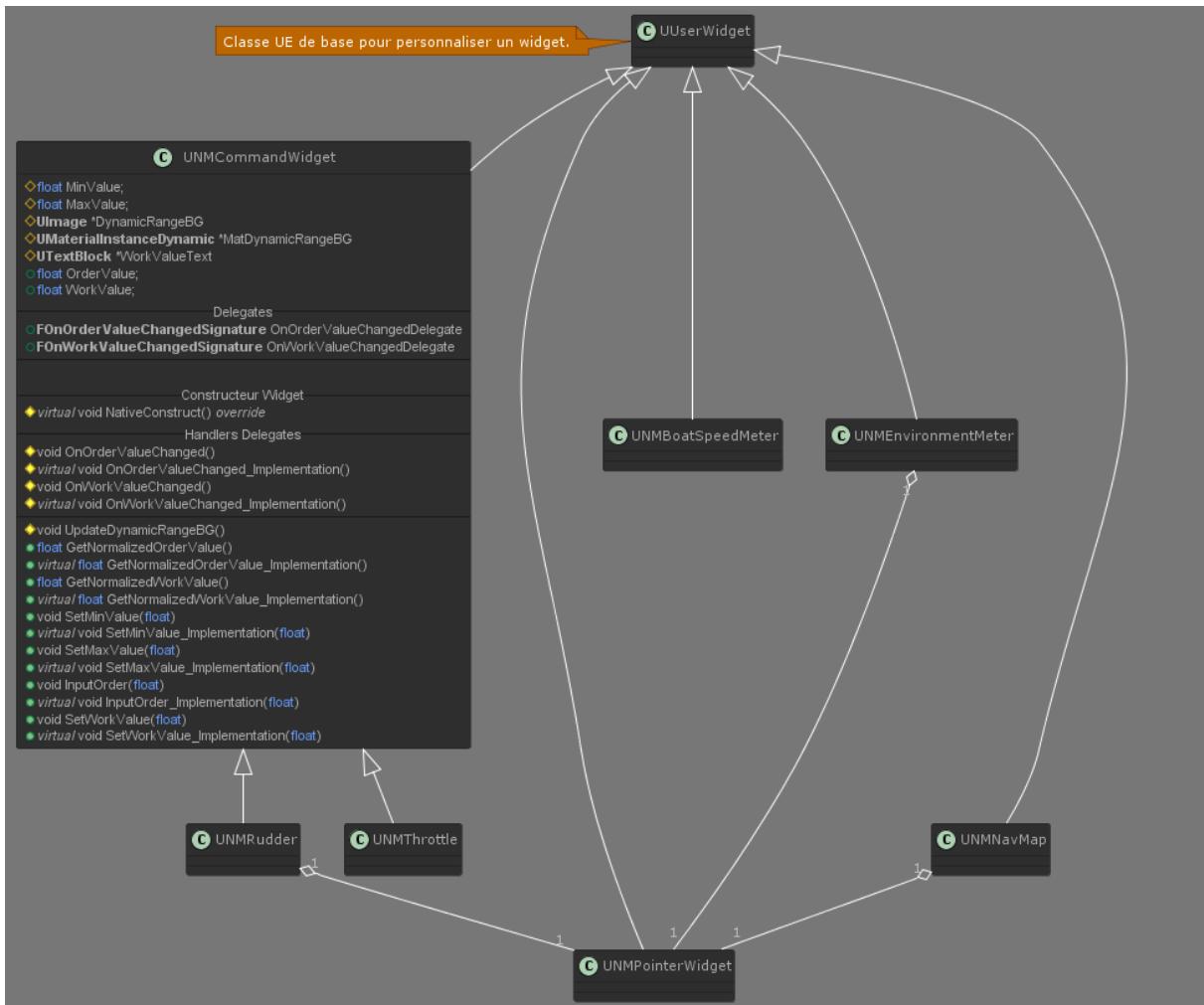
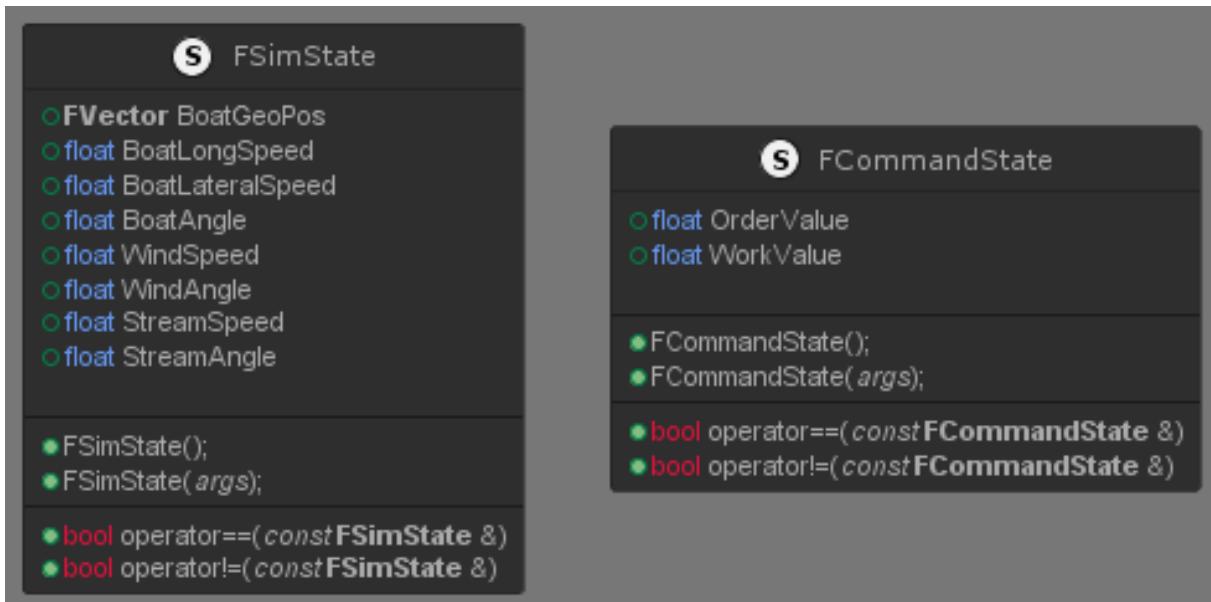


FIGURE 73 – Diagramme de classes des widgets.

**FIGURE 74** – Diagramme de classes des structures d'états.**FIGURE 75** – Diagramme de classes des layouts.

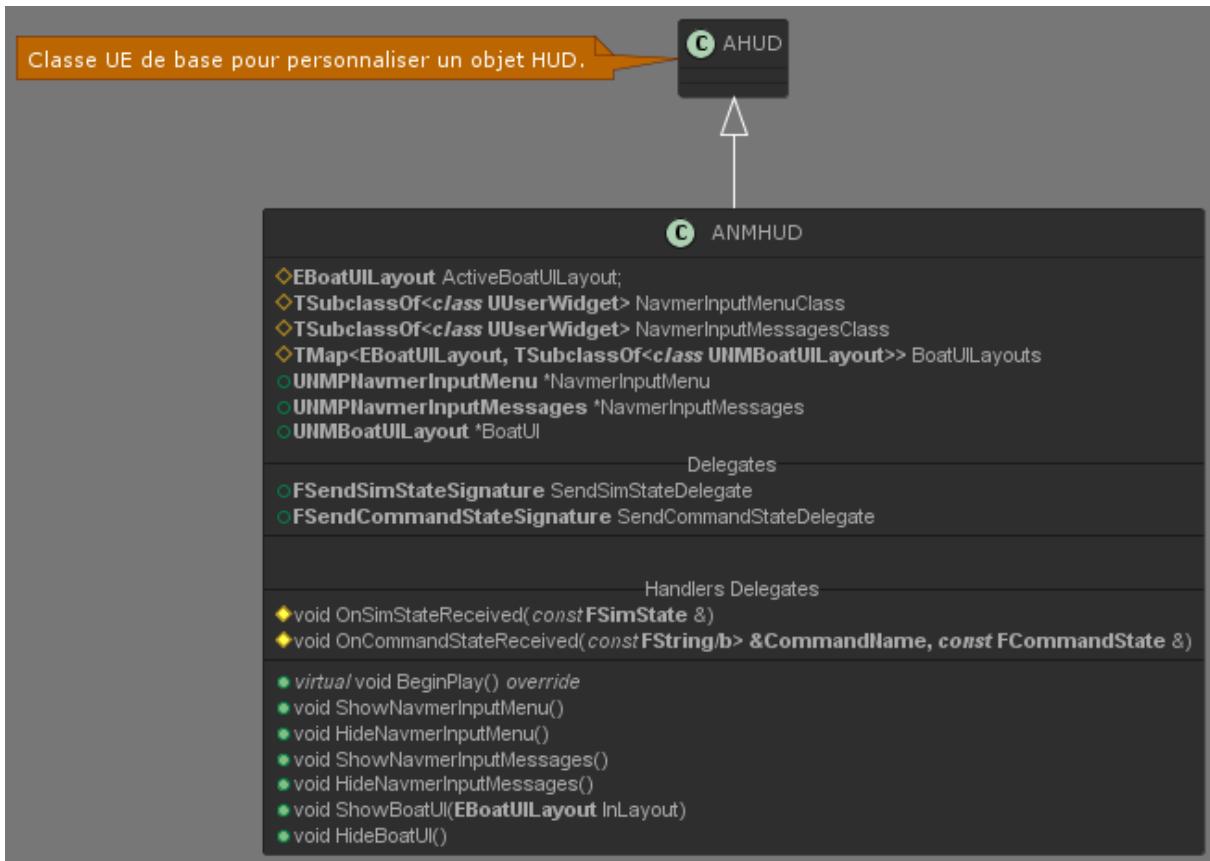


FIGURE 76 – Diagramme de classe de NMHUD.

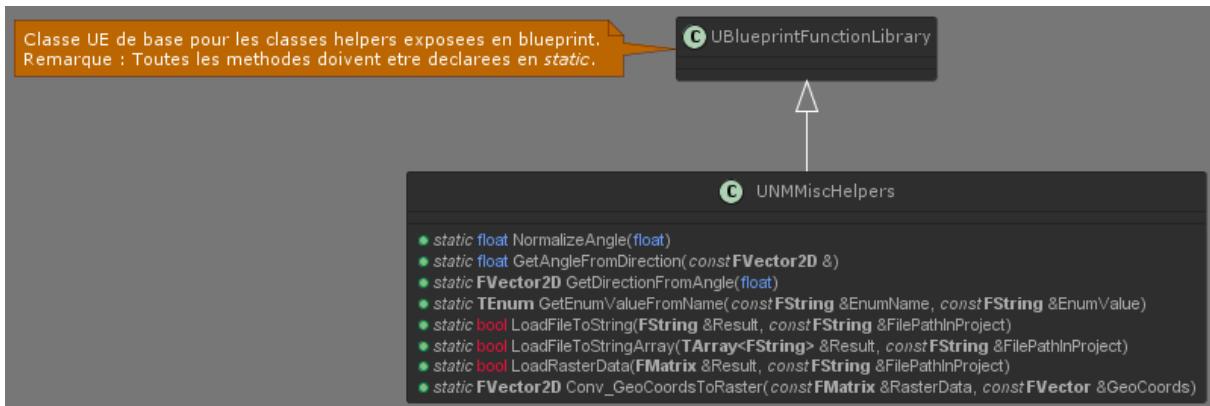


FIGURE 77 – Diagramme de classe de NMMiscHelpers.