

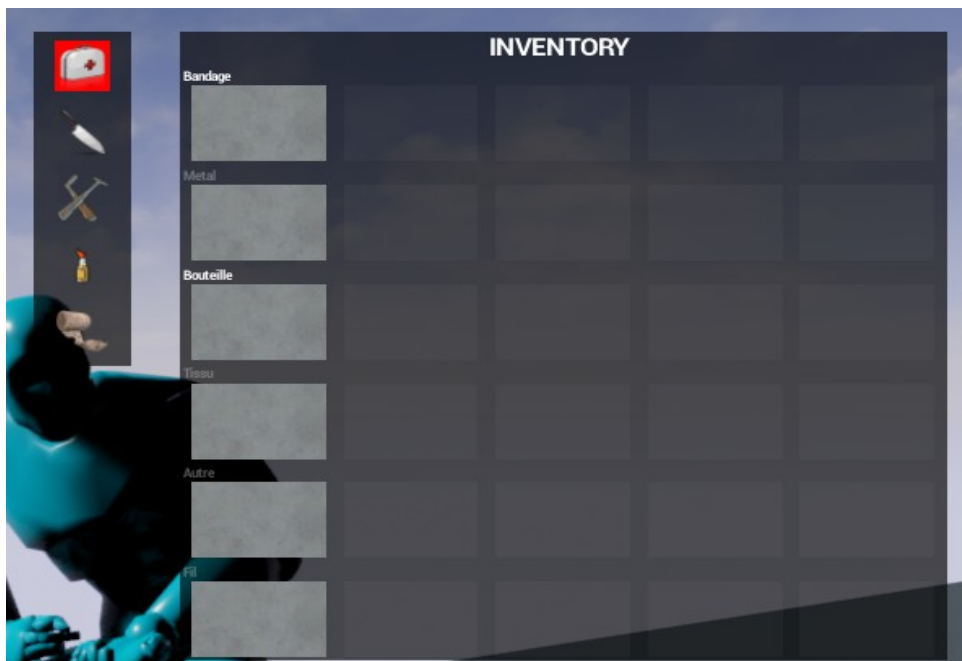
Introduction

(NB : Ceci constitue mon premier tutoriel sur UE4, j'espère qu'il vous sera utile et qu'il sera compréhensible par tous. Si vous avez des questions n'hésitez pas à m'envoyer un message sur Facebook « Antoine Gargasson » ou par mail antoine.gargasson@gmail.com)

(NB2 : Ce tuto est réalisé sur la version 4.8.3 de UE4)

Dans ce premier tutoriel, nous allons aborder l'inventaire et le système de craft en C++.

Nous allons réaliser un inventaire dans le style The Last Of Us. Voici le résultat final :



On distingue donc sur la barre de gauche la sélection de l'objet à crafter avec une couleur indiquant la possibilité de créer ce craft et au milieu le nombre d'unités de chaque objet dans l'inventaire, représenté par une barre remplie, ainsi que la coloration de l'objet nécessaire au craft.

Attention, ce tutoriel est le même que le Tuto 1 en blueprint excepté que tout le travail fait en blueprint sera fait en C++.

Il est donc possible que certaines images ne correspondent pas **exactement** à ce que vous aurez mais rassurez vous ce n'est que le nom. Je compte sur vous pour **adapter** et ne pas **recopier simplement** le code ;)

Préparation

Avant toute chose il est important de préparer le terrain en posant quelques termes et quelques critères de production afin de bien s'y retrouver.

Définitions :

- Un **craft** est un objet réalisé grâce à un Item. Il n'est pas ramassable.
- Un **Item** est un objet ramassable dans le monde qui est stocké dans l'inventaire et s'utilise pour réaliser des craft.

Critères :

- Nous voulons que les **Items** se stockent dans l'inventaire, qu'ils aient un nom, une représentation physique, que leur nombre s'incrémente et que ça soit visible.
- Nous voulons pouvoir naviguer entre les **craft**, voir sur quel craft nous nous sommes arrêté, voir quels Items sont nécessaires et si le craft est **réalisable** ou non.
- Nous voulons que le tout soit modulable assez facilement.

Ces bases posées, nous allons pouvoir commencer à tailler dans le vif du sujet ! :)

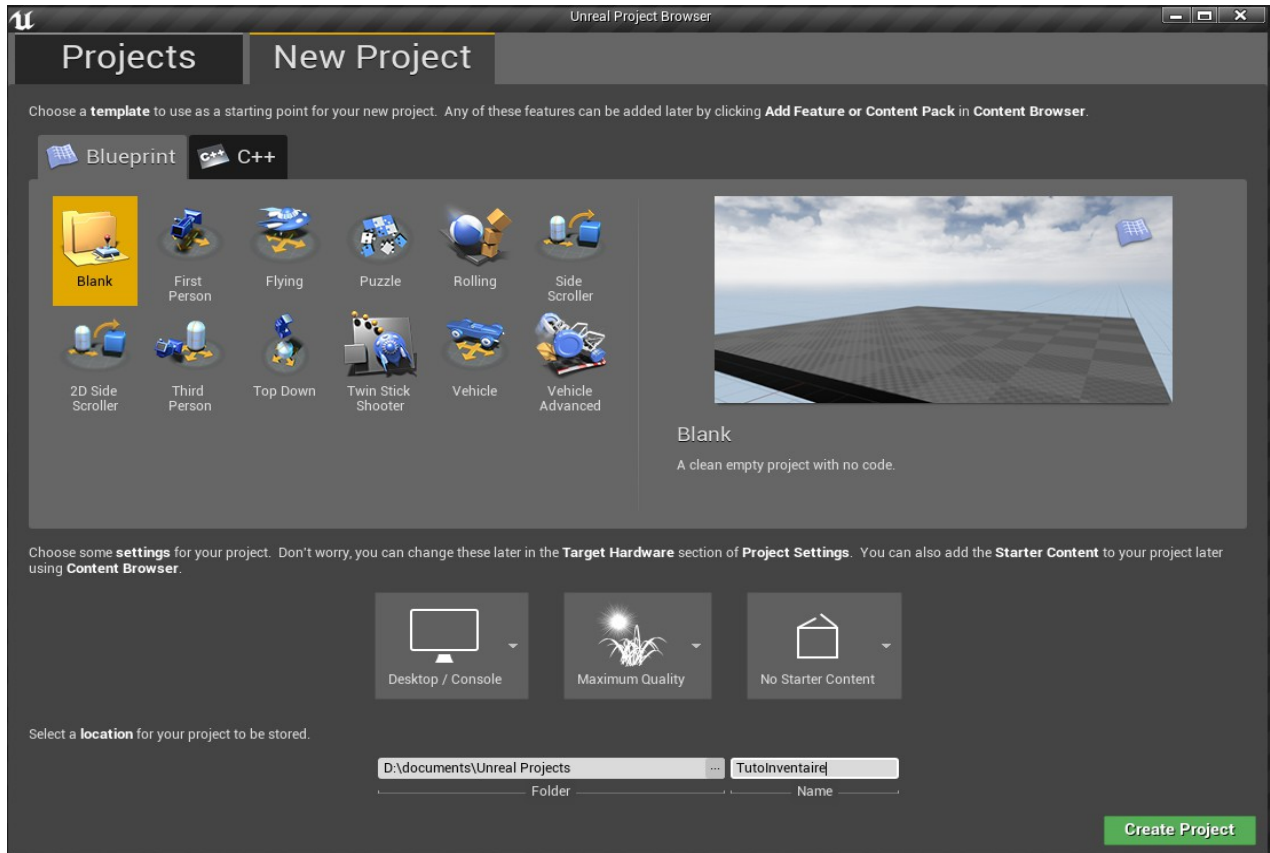
Nous allons tout d'abord commencer par poser les bases de l'inventaire puis nous lui ajouterons des fonctionnalités pour enfin lui donner une représentation visuelle à l'aide des UMG.

Puis, dans un prochain tuto, nous ajouterons la possibilité de crafter un item et sa représentation visuelle.

(NB : Un tuto sur le ramassage d'objets sera effectué plus tard car ce n'est pas le propos du tutoriel d'aujourd'hui.)

L'inventaire

Tout d'abord, créez nouveau projet vide sans le starter content et nommez le à votre convenance.

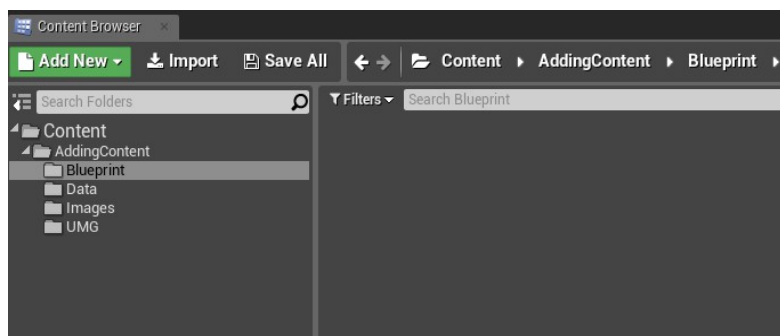


Actuellement, si vous lancez le jeu en cliquant sur « **Play** », vous incarnez une caméra que vous pouvez déplacer à l'aide de la souris et de WASD (si vous laissez les contrôles par défaut)

Il est important de bien classer et trier les objets du **Content Browser**. Nous allons donc créer des dossiers où nous rangerons le tout plus tard.

Je crée toujours un dossier « Adding Content » pour ne pas mélanger ce que j'ajoute et ce qui est de base dans UE4. A vous de voir si vous faites de même ou non.

Nous créons donc un dossier « **Blueprint** », « **UMG** », « **Data** » et « **Images** ». Pour ça il existe 2 solutions : clique droit sur un dossier et **New Folder** ou clique sur **Add New** et **New Folder**.



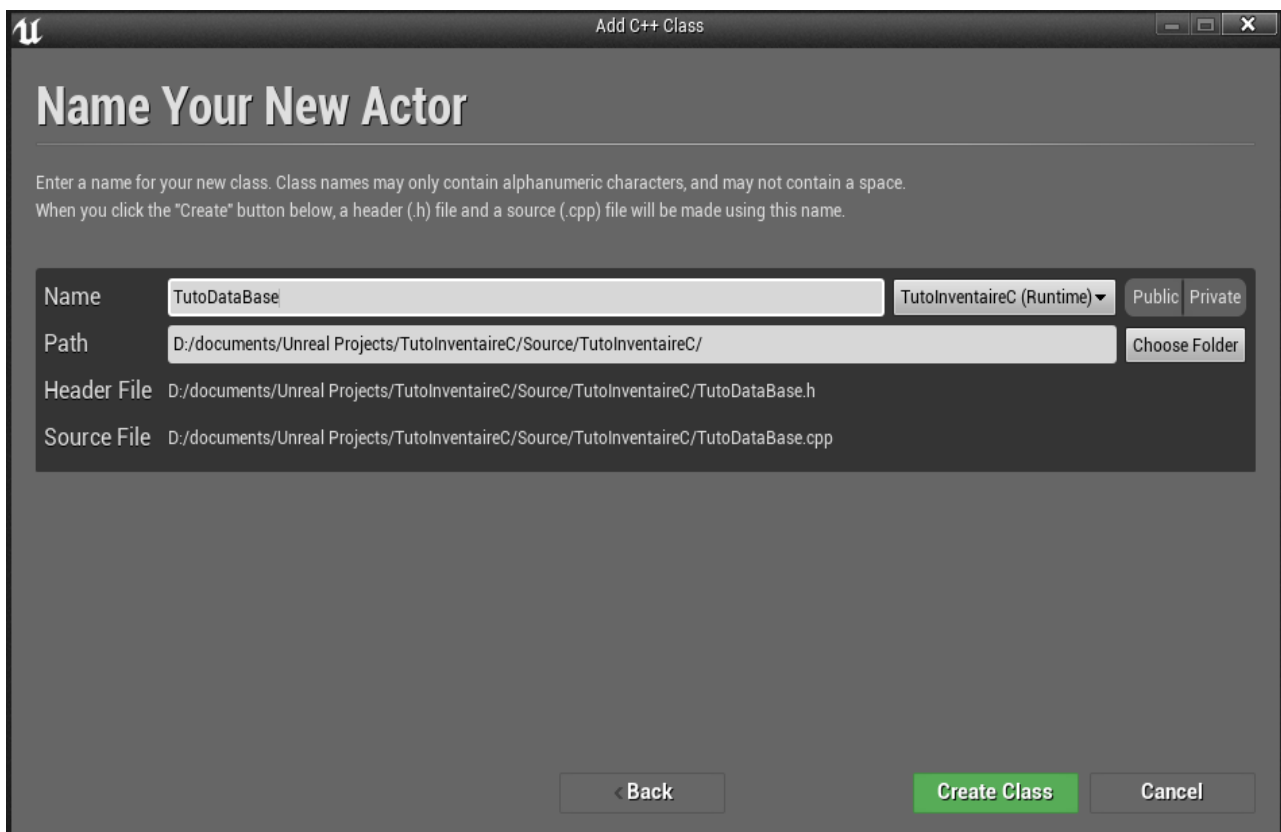
Le dossier *Blueprint* va contenir tous les blueprint des **Items**. *Images* va contenir toutes vos ressources graphiques. *UMG* stockera tout le visuel utilisant UMG et enfin *Data* va contenir toutes les données utilisées pour les Items.

Avant tout, nous allons sauvegarder car UE4 crash de temps en temps et il est important de tout le temps sauvegarder son travail (**ctrl+s**) . Faites un File puis Save As et enregistrez votre map.

Commençons par les **Data**. Ce sont le cœur de l'inventaire. en effet, l'inventaire va manipuler des données non présentes dans le monde et il est important de les stocker afin de ne pas les perdre.

Nous allons commencer par utiliser les **structures**. Les structures servent à rassembler plusieurs **types primitifs** dans un seul bloc. On peut par exemple rassembler une chaîne de caractères (string) et un entier (integer).

Faites **File** et **New C++ Class...** C'est ainsi que vous allez créer toutes vos classes. Choisissez **Actor**, faites **Next** et nommez la **TutoDataBase**.

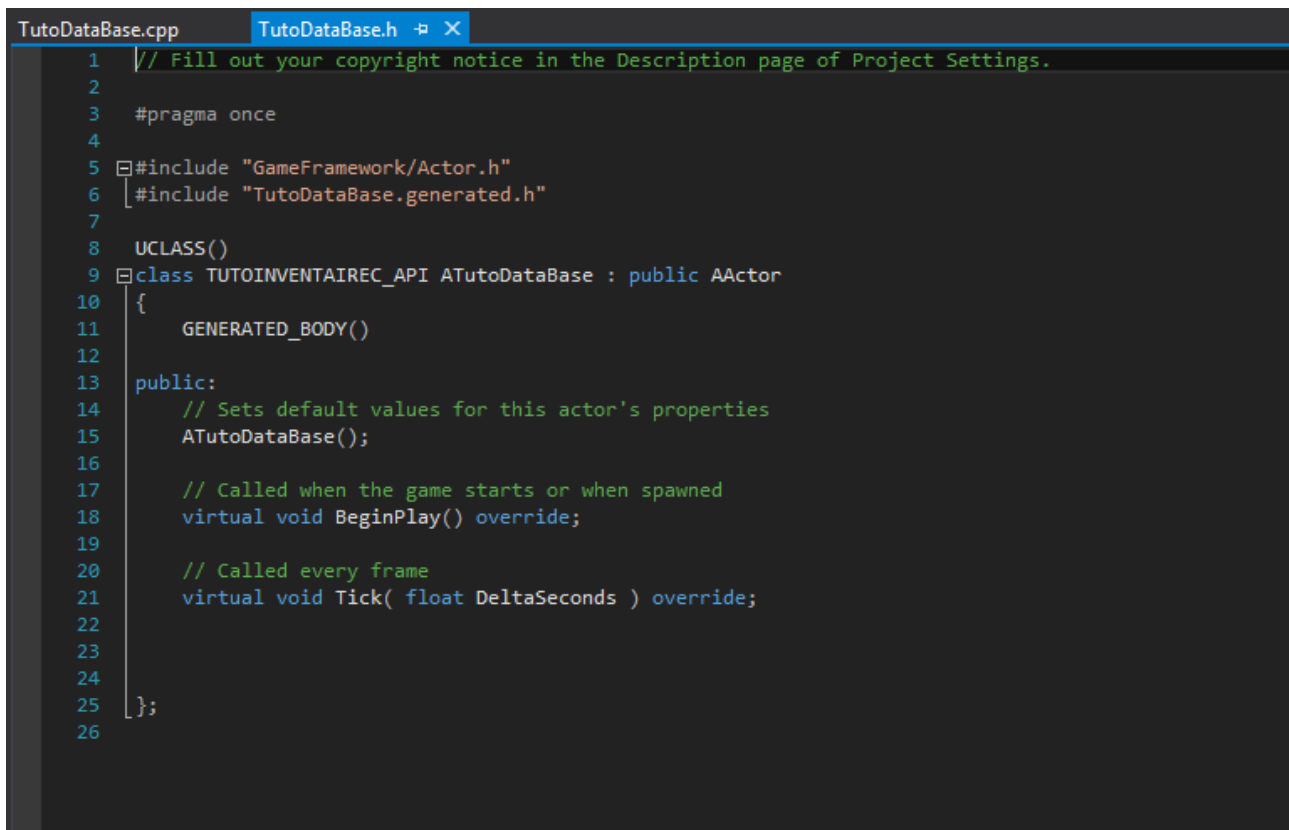


Une classe de type **Actor** est **plaçable** dans le monde. Il est important de pouvoir **retrouver** vos classes **rapidement**, donnez un mot en **préfixe** de vos nom, ici ça sera le mot **Tuto**.

Notre **DataBase** va **stocker** nos **items** et nous **aider** à les **créer**.

Cliquez sur **Create Class**. UE4 devrait **compiler** votre projet (toujours un peu long la première fois) et **ouvrir** Visual Studio (j'ai la version 13 mais la 15 marchera si vous avez cette version).

Vous vous trouvez à présent dans **Visual Studio**. Il a dû vous ouvrir **automatiquement** le .h et le .cpp. Si ce n'est pas le cas, rendez vous **l'explorateur** de projet, Déroulez **Games**, **NomDeVotreProjet**, **Source**, **NomDeVotreProjet**. Vous devriez trouver **toutes** vos **classes** ici :



```
TutoDataBase.cpp  TutoDataBase.h -# X
1  // Fill out your copyright notice in the Description page of Project Settings.
2
3  #pragma once
4
5  #include "GameFramework/Actor.h"
6  #include "TutoDataBase.generated.h"
7
8  UCLASS()
9  class TUTOINVENTAIREC_API ATutoDataBase : public AActor
10 {
11     GENERATED_BODY()
12
13 public:
14     // Sets default values for this actor's properties
15     ATutoDataBase();
16
17     // Called when the game starts or when spawned
18     virtual void BeginPlay() override;
19
20     // Called every frame
21     virtual void Tick( float DeltaSeconds ) override;
22
23
24
25 };
26
```

Par défaut, vous allez avoir déjà des fonctions écrites pour que **l'Actor** puisse **fonctionner** de base. Vous avez dans l'ordre : Le **constructeur** pour **initialiser** les **variables**, la réécriture de la fonction **BeginPlay** (au **démarrage** du jeu) et la réécriture de la fonction **Tick** (à chaque **frames**).

A présent c'est à vous de décider de quoi sera composé votre Item. J'ai choisi de caractériser mon Item comme suit :

<u>Item</u> :
ID : Integer
Nom : String
Quantite : Integer

Voici une représentation de mon Item sous forme simplifié (UML).

Il est donc composé d'un ID qui est un entier et qui servira à récupérer rapidement un objet et à les différencier facilement, d'un nom qui est une chaîne de caractères et qui sera affiché dans l'inventaire, d'une quantité qui est un entier et qui représentera le nombre de cet objet dans l'inventaire.

Pour écrire une structure, je vous le rappelle, il faut procéder comme ceci :

```
USTRUCT()
struct FUneStructure
{
    GENERATED_USTRUCT_BODY()

    //mettre vos variables et fonctions ici

    FUneStructure()
    {
        //mettre l'initialisation des variables ici
    }
};
```

Vous **devez** écrire tout ceci dans un **.h** . Nous allons donc écrire la **structure** dans la classe **TutoDataBase.h** .

Attention ! Vous devez écrire vos structures en dehors de la classe et le nom de votre structure doit commencer par un F (en majuscule) ! Ou votre projet ne compilera pas !

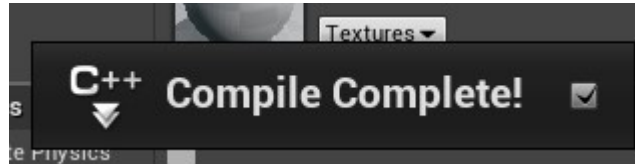
Vous obtenez, après écriture et ajout des variables, quelque chose comme ça :

```
4
5 #include "GameFramework/Actor.h"
6 #include "TutoDataBase.generated.h"
7
8
9 USTRUCT()
10 struct FItem
11 {
12     GENERATED_USTRUCT_BODY()
13
14     //mettre vos variables et fonctions ici
15
16     UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Item")
17     int32 ID;
18
19     UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Item")
20     FString Nom;
21
22     UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Item")
23     int32 Quantite;
24
25     FItem()
26     {
27         //mettre l'initialisation des variables ici
28         ID = -1;
29         Nom = "UNKNOWN";
30         Quantite = 1;
31     }
32 };
33
34
35 UCLASS()
36 class TUTOINVENTAIREC_API ATutoDataBase : public AActor
37 {
38     GENERATED_BODY()
39 }
```

Vous remarquez que ma structure **commence** bien par un **F** (qui ne sera pas traité à la compilation), qu'elle se situe **entre** la **classe** et les **includes** et que j'ai bien **initialisé** mes **variables**.

Sauvegardez, retournez sur UE4 et **cliquez** sur le petit icône en forme de **rubiks cube** où vous pouvez lire **Compile**.

Si tout se passe bien vous devriez obtenir ceci :



Créez une nouvelle **classe** (**File, New C++ Class**).

Sélectionnez un **Character** et nommez le « **TutoJoueur** ».

Une fois encore, Visual studio s'ouvre avec votre classe.

Nous allons maintenant affecter l'inventaire à notre joueur. Un inventaire n'est qu'un tableau d'Items. Nous allons simplement créer un tableau qui va contenir tous nos items.

Nous allons rajouter, en C++, un **tableau d'Items** qui soit **éditable** et **visible** en **blueprint**. C'est encore plus simple que en blueprint. Il suffit de **savoir où** mettre le code.

Nous voulons que **l'inventaire** soit sur le **personnage**, nous allons donc écrire le code **dans** le **TutoJoueur**.

Ouvrez votre fichier **TutoJoueur.h** . Vous allez avoir un **GENERATED_BODY()** et en public un **constructeur** et 3 **fonctions** : **BeginPlay**, **Tick** et **SetupPlayerInputComponent** (nous y reviendrons sur le second tuto). Il va nous falloir rajouter à cet **endroit** le **tableau d'Items**.

Avant de pouvoir utiliser la structure, nous devons **spécifier** où elle est **créée**. Il faut ajouter dans le fichier **TutoJoueur.h** la ligne suivante :

```
#include "TutoBase.h"
```

Attention à bien le rajouter **après l'include** du **framework** et **avant l'include** du **generated.h** qui **doit** être le **dernier include**.

Rajoutez à la suite de **SetupPlayerInputComponent** :

```
// Inventory as an array
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Gameplay
Tarray<FItem> Inventaire ;
```

On compile et on sauvegarde.

Ici on a spécifié que l'**inventaire** est **éditable partout** et **public**, qu'il peut être **set** et **get** par les **blueprints** et j'ai choisi de le mettre dans la **catégorie Gameplay**.
J'utilise un **Tarray** de **type FItem** car j'ai envie de pouvoir jouer avec la **taille** du **tableau**.

```
1 // Fill out your copyright notice in the Description page of Project Settings.
2
3 #pragma once
4
5 #include "GameFramework/Character.h"
6 #include "TutoDataBase.h"
7 #include "TutoJoueur.generated.h"
8
9 UCLASS()
10 class TUTOINVENTAIREC_API ATutoJoueur : public ACharacter
11 {
12     GENERATED_BODY()
13
14 public:
15     // Sets default values for this character's properties
16     ATutoJoueur();
17
18     // Called when the game starts or when spawned
19     virtual void BeginPlay() override;
20
21     // Called every frame
22     virtual void Tick( float DeltaSeconds ) override;
23
24     // Called to bind functionality to input
25     virtual void SetupPlayerInputComponent(class UInputComponent* InputComponent) override;
26
27     // Inventory as an array
28     UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Gameplay)
29     TArray<FItem> Inventaire;
30
31 };
32
```

Maintenant que vous avez vos **classes**, il faut les **créer** et les **instancier** dans le **monde** !

De retour dans UE4, allez dans votre **dossier** de **blueprint**. Un **clic droit** et on crée une **classe** de **blueprint**. **Déroulez** le volet "**All Classes**" et **cherchez** **TutoJoueur**. Sélectionnez et nommez le **BP_TutoJoueur**. Ouvrez ce dernier blueprint.

Vous aurez remarqué que dans les **détails**, quand votre **BP_TutoJoueur** est **sélectionné** dans la liste des **Components**, vous pouvez **ajouter** des **éléments** à votre tableau. En **cliquant** sur la **croix**, vous remarquez qu'on peut **modifier** les **valeurs** par défaut de l'Item actuel. Vous l'aurez compris c'est **ici** qu'il va falloir **ajouter** vos **Items**.

Si quand vous cliquez sur la croix, rien ne se passe ou que vous ne voyez aucun ID ou autre juste une ligne supplémentaire, **sauvegardez** et **relancez** votre **projet**, les fichiers .h se **compilent mal** parfois ;)

Dans mon exemple j'ai 6 Items différents. Le Bandage, le Metal, la Bouteille, le Tissu, le Fil et une catégorie Autre. Ajoutez 6 items à votre inventaire et changez les noms. Laissez la quantité à 0 et

augmentez l'ID à chaque item (Bandage : 0, Metal : 1, etc..).

Inventaire		6 elements	+	-	↺
▲ 0		3 members			↺
ID	0				↺
Nom	Bandage				↺
Quantite	0				↺
▲ 1		3 members			↺
ID	1				↺
Nom	Metal				↺
Quantite	0				↺
▲ 2		3 members			↺
ID	2				↺
Nom	Bouteille				↺
Quantite	0				↺

On compile et on sauvegarde (ça doit devenir un réflexe :P)

Félicitation vous avez votre inventaire ! :)

Il ne reste plus qu'à le tester et à le montrer au joueur !

Tests

Il est important de bien tester avant de conclure et de partir dans le visuel. Les tests permettent de voir si votre réalisation correspond à ce que vous attendiez et surtout de créer des fonctions qui serviront pour le visuel.

Toujours dans le `TutoJoueur`, nous allons créer des fonctions. C'est très pratique, cela permet d'utiliser tout un calcul rapidement sans avoir à le réécrire à chaque fois. Comme un professeur à moi dit « Si vous utilisez plus de deux fois une valeur, elle doit être variable et si vous utilisez plus de deux fois un calcul, il doit être fonction ».

Pour commencer nous allons créer une fonction qui va nous retourner le nombre d'unité d'un objet en fonction de son ID.

Vous allez me dire « et pourquoi pas de son nom ? » parce qu'il va falloir effectuer un travail sur le nom pour qu'il n'y est pas d'erreur de casse (Majuscule et Minuscule) ce qui peut être très long si vous avez 600 objets différents.

Retournez dans le `TutoJoueur.h`, ajoutez une fonction qui **retourne** un **entier**, qui prend en **paramètre** un **entier** et **nommez** la « **GetNumberFromID** » qui sera plus facile que de la traduire « **AvoirNombreDepuisID** », enfin c'est à vous de voir..

En paramètres de la **UFUNCTION**, rajoutez **BlueprintCallable** et une **Category** afin de pouvoir l'utiliser plus tard dans des **blueprints**.

Dans le `.h`, nous aurons donc la fonction suivante à ajouter :

```
// Return the Number of Item From a specific ID
UFUNCTION(BlueprintCallable, Category = Gameplay)
int32 GetNumberFromID(int32 TheID);
```

Et dans le `.cpp` :

```
// Return the Number of Item from a specific ID
int32 ATutoJoueur::GetNumberFromID(int32 TheID)
{
    int32 retour = -1;
    for (int32 index = 0; index < Inventaire.Num(); index++)
    {
        if (Inventaire[index].ID == TheID)
        {
            retour = Inventaire[index].Quantite;
            break;
        }
    }
    return retour;
}
```

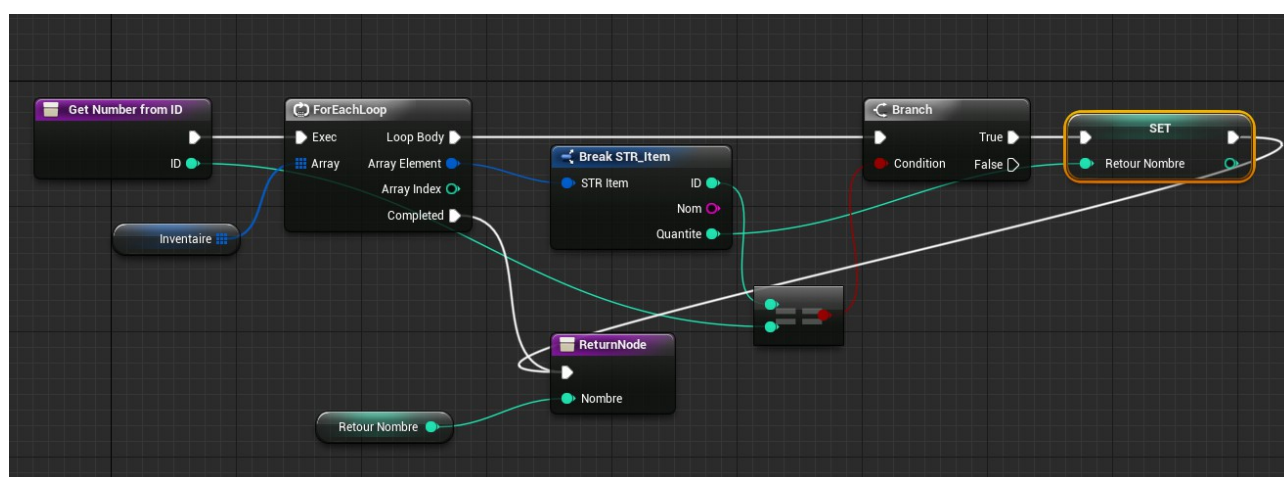
Dans le **.h**, que du **classique**, une fonction qui renvoie un entier et qui prend en paramètre un entier. Dans le **.cpp**, je **crée** une **variable** qui sera **retournée** dans tous les cas et je **l'initialise** à **-1** pour détecter des **problèmes**, je **parcours** mon **tableau**. **Inventaire.Num()** va me **renvoyer** le nombre **d'éléments** du tableau. Je vérifie la **correspondance** et **j'affecte** le **résultat** si c'est bon. Un **break** de boucle pour **éviter** du travail qui serait inutile et enfin je **renvoi** la valeur **finale**.

C'est quand même plus rapide que de faire chaque bloc séparément.. Voici le C++ et le blueprint comparés :

```

36 // Return the Number of Item from a specific ID
37 int32 ATutoJoueur::GetNumberFromID(int32 TheID)
38 {
39     int32 retour = -1;
40     for (int32 index = 0; index < Inventaire.Num(); index++)
41     {
42         if (Inventaire[index].ID == TheID)
43         {
44             retour = Inventaire[index].Quantite;
45             break;
46         }
47     }
48     return retour;
49 }

```



Ces deux fonctions font exactement le même travail. C'était à titre d'exemple, je ne comparerai plus puisqu'il n'y a pas matière car il s'agit d'un tuto C++.

J'ai également créé de la même manière une fonction **GetItemIndexWithID** qui me renvoi l'index dans le tableau de l'item (et oui être programmer c'est savoir être fainéant quand il le faut ;))

Elle se construit pareil mais vous récupérez **l'index**. Attention à bien **insérer** la ligne correspondante dans le **.h** également :

```
// Return the Number of Item from a specific ID
int32 ATutoJoueur::GetNumberFromID(int32 TheID)
{
    int32 retour = -1;
    for (int32 index = 0; index < Inventaire.Num(); index++)
    {
        if (Inventaire[index].ID == TheID)
        {
            retour = index;
            break;
        }
    }
    return retour;
}
```

Nous allons maintenant ajouter deux fonctions qui vont ajouter et supprimer un item dans l'inventaire en utilisant son ID.

Créez deux fonctions **AddItemWithID** et **RemoveItemWithID**. Ajoutez également une variable **NombreMaxItem** (Integer). Initialisez-la à 5.

(NB : Il est important de bien nommer ses fonctions et ses variables pour maintenir son code et que si une personne doit passer après vous puisse comprendre votre code et s'y retrouver)

En paramètre, pour les deux fonctions, vous mettrez un **Integer ID** et le type de retour est **void**.

L'ajout d'un Item

```
// Add an Item in the inventory with the specific ID
void ATutoJoueur::AddItemWithID(int32 TheID)
{
    int32 Number = GetNumberFromID(TheID);
    if (Number < NombreMaxItem)
    {
        int32 index = GetItemIndexWithID(TheID);
        Inventaire[index].Quantite++;
    }
    else
    {
        if (GEngine)
        {
            GEngine->AddOnScreenDebugMessage(-1, 5.f,
FColor::Yellow, TEXT("Impossible d'ajouter un Item avec l'id : " +
FString::FromInt(TheID)));
        }
    }
}
```

Remarque : j'aurais pu renvoyer un **integer** qui me dirait si l'ajout s'est **bien** passé.

On récupère le nombre d'unités et on le compare au nombre max pour voir si on peut ajouter un item, si c'est le cas, on incrémente le nombre d'unité sinon on affiche un message.

(NB : vous pouvez aussi ajouter en entrée le nombre que vous souhaitez ajouter mais il faudra modifier un peu votre code pour qu'il s'adapte)

Suppression de l'Item

```
// Remove an Item in the inventory with the specific ID
void ATutoJoueur::RemoveItemWithID(int32 TheID)
{
    int32 Number = GetNumberFromID(TheID);
    if (Number > 0)
    {
        int32 index = GetItemIndexWithID(TheID);
        Inventaire[index].Quantite--;
    }
    else
    {
        if (GEngine)
        {
            GEngine->AddOnScreenDebugMessage(-1, 5.f,
FColor::Yellow, TEXT("Impossible de supprimer un Item avec l'id : " +
FString::FromInt(TheID)));
        }
    }
}
```

Une fois ces fonctions créées, nous allons pouvoir les tester in-game.

Tout d'abord il faut rajouter des bind d'actions. Ces bind sont des touches que l'ont va assigner et qui vont effectuer des actions spéciales. Pour binder une touche, retournez sur UE4 et rendez vous dans **Edit, Project Settings**. Ici vous trouverez toutes les infos relatives à votre projet. Sous la catégorie **Engine**, cliquez sur **Input**.

C'est dans cette fenêtre que vous allez trouver tous les binds (aussi appelé mapping) d'actions et d'axes. Si vous êtes dans un projet vide comme moi, il ne devrait y avoir aucun bind.

Pour rajouter un bind, il suffit de cliquer sur le + à côté de la section choisie. Un bind d'action se réfère à une touche du clavier ou de la manette alors qu'un bind d'axe se réfère à un mouvement de souris ou de joystick.

Nous allons rajouter 3 binds. Un pour l'ouverture et la fermeture de l'inventaire, un pour ajouter un objet et un pour retirer un objet.

Cliquer sur le + à côté de **action mapping**, déroulez le menu si besoin et changez le nom du premier en **Inventaire**. Déroulez encore une fois et là vous pouvez assigner une touche en cliquant sur le menu déroulant et en choisissant votre touche. J'ai choisi de binder la touche I.

Faites de même pour **AjoutObjet** bindé sur P et **SuppressionObjet** bindé sur M.

Une fois vos inputs prêt, rendez vous dans l'**event graph** du BP_Joueur. Un clique droit sur le graph et on trouve **AjoutObjet** en cherchant le début du mot. Vous pouvez désormais lier une action à votre touche.

Quand la touche AjoutObjet sera pressée, nous allons ajouter un bandage à l'inventaire (ID 0) et quand la touche SuppressionObjet sera pressée, nous allons supprimer un bandage.

Pour cela, nous allons **affecter** des **actions** en C++. **Retournez** sur Visual Studio et allez sur l'onglet **Tuto_Joueur.cpp**.

Pour affecter une action à une touche, nous allons **rajouter** des lignes dans la fonctions **SetupPlayerInputComponent**.

Quand vous voulez **affecter** une **fonction** à une **touche**, ce qu'on appel **bind** une **touche**, vous allez utiliser **deux** commandes en fonction de si vous bindez un **axe** ou une **action**.

```
//Bind la fonction MoveRight sur le character (this) sur la touche MoveRight
InputComponent->BindAxis("MoveRight",this,&Character::MoveRight);

//Bind la fonction Fire sur le character (this) sur la touche Fire
InputComponent->BindAction("Fire",IE_Pressed,this,&Character::Fire);
```

A vous d'adapter donc ce que vous souhaitez faire à ces deux schémas :

BindAxis("NomBouton", this, &NomClasse::NomFonction) ;

BindAction("NomBouton", TypeExecution, this, &NomClasse::NomFonction) ;

En **type d'exécution** on trouve **IE_Pressed** (à l'appuie), **IE_Released** (au relâchement), **IE_Repeat** (en répétition).

Nous ne pouvons pas ajouter de paramètres lors de Bind de touche, in va falloir créer une fonction si vous souhaitez appeler une fonction qui utilise des paramètres.

Créez donc deux **fonctions** de type **void**, nommez les **AjoutObjet** et **SuppressionObjet**. On les nomme comme les touches pour les **reconnaître**. Ne les mettez **pas BlueprintCallable**, ces fonctions seront utiles que dans le C++.

Nous bindons les touches en C++ pour n'utiliser que du C++, rien ne vous empêche de le faire en blueprint si vous trouvez cela plus rapide ou plus simple.

Petit résumé à suivre :

Dans le .h on a donc ça :

```
// Add an Item in the inventory with the specific ID
UFUNCTION(BlueprintCallable, Category = Gameplay)
void AddItemWithID(int32 TheID);

// Remove an Item in the inventory with the specific ID
UFUNCTION(BlueprintCallable, Category = Gameplay)
void RemoveItemWithID(int32 TheID);

// TEST : Test the adding of an object
UFUNCTION()
void AjoutObjet();

// TEST : Test the removing of an object
UFUNCTION()
void SuppressionObjet();
```

Et dans le .cpp :

```
// Called to bind functionality to input
void ATutoJoueur::SetupPlayerInputComponent(class UInputComponent*
InputComponent)
{
    Super::SetupPlayerInputComponent(InputComponent);

    InputComponent->BindAction("AjoutObjet", IE_Pressed, this,
&ATutoJoueur::AjoutObjet);
    InputComponent->BindAction("SuppressionObjet", IE_Pressed, this,
&ATutoJoueur::SuppressionObjet);
}

// TEST : Test the adding of an object
void ATutoJoueur::AjoutObjet()
{
    AddItemWithID(0);
}

// TEST : Test the removing of an object
void ATutoJoueur::SuppressionObjet()
{
    RemoveItemWithID(0);
}
```

Avant de lancer le jeu et pouvoir admirer le résultat, vous devez spécifier un nouveau **GameMode** si vous êtes dans un projet vide car votre personnage BP_TutoJoueur n'est **pas utilisé** par le jeu.

Créez une **nouvelle** classe **C++** de type **GameMode**. Nommez la **TutoGameMode**. Votre Visual Studio, une fois de plus, est ouvert et la surprise, il n'y a rien dans vos fichiers (sauf des includes). C'est parce que le GameMode ne gère pas grand-chose de base et qu'il est déjà bien rempli sans qu'on le voit.

Rajoutons donc le **constructeur** et le **BeginPlay** dans le **.h**.

```
UCLASS()
class TUTOINVENTAIREC_API ATutoGameMode : public AGameMode
{
    GENERATED_BODY()

public:

    ATutoGameMode(const FObjectInitializer& ObjectInitializer);

    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

};
```

Dans le **.cpp**, dans le **constructeur**, nous allons **récupérer** la **classe** du **BP_TutoJoueur** afin de l'utiliser comme joueur **par défaut**. Nous allons ensuite **afficher** un **message** dans le **BeginPlay** afin de bien se rendre compte que l'on utilise le bon **GameMode**.

```
ATutoGameMode::ATutoGameMode(const FObjectInitializer& ObjectInitializer) :
Super(ObjectInitializer)
{
    //set the default pawn class to our Blueprinted character
    static ConstructorHelpers::FClassFinder<APawn>
PlayerPawnObject(TEXT("Pawn'/Game/AddingContent/Blueprint/BP_TutoJoueur.BP_TutoJo
ueur_C'"));
    if (PlayerPawnObject.Class != NULL)
    {
        DefaultPawnClass = PlayerPawnObject.Class;
    }
}

void ATutoGameMode::BeginPlay()
{
    Super::BeginPlay();

    StartMatch();

    if (GEngine)
    {
        GEngine->AddOnScreenDebugMessage(-1, 5.f, FColor::Yellow, TEXT("HELLO
WORLD"));
    }
}
```

Sauvegardez et compilez ! (je ne le dit pas à chaque fois, mais quand vous revenez sur UE4, ça doit être un réflexe ;)

Allez dans le dossier **blueprint** et créez un nouveau **blueprint** de type **TutoGameMode**. Nommez le BP_TutoGameMode.

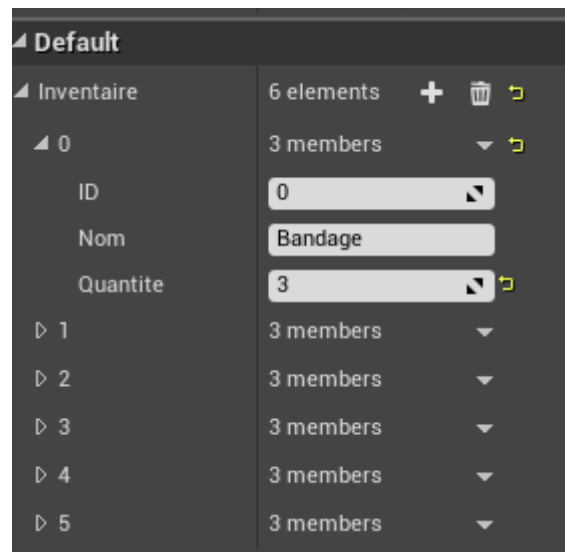
Une fois ouvert vous pourrez modifier les **defaults** (fenêtre Details). Changer le **pawn class** par BP_TutoJoueur s'il ne s'est pas modifié.

Une fois fait vous devrez aussi modifier, dans la fenêtre de **projet settings** de tout à l'heure, le Game Mode utilisé. Rendez vous cette fois dans **Maps & Modes**. Remplacez le « **Default Game Mode** » par le vôtre (BP_TutoGameMode).

Lancez le jeu une fois toute cette préparation faite. Vous remarquerez que vous ne bougez plus et que en haut de la fenêtre on voit "HELLO WORLD", c'est normal, cela signifie que votre personnage est bien remplacé.

Dans le **world outliner**, sélectionnez le **BP_TutoJoueur** que vous verrez et repérez dans ces **details** l'inventaire. Déroulez le et vous constaterez que tout est bien en place. Vous allez pouvoir vérifier que vos fonctions fonctionnent en surveillant les quantités. Déroulez le 0 et vérifiez que vous êtes bien sur le bandage et qu'il a bien l'ID 0.

Appuyez sur P et M pour **faire varier** les quantités de bandages. Si vous avez tout bien fait, votre quantité ne devrait pas excéder 5 et ne pas descendre en dessous de 0. Votre **message** doit apparaître si vous essayez d'aller plus loin.



Votre inventaire est fonctionnel et vous pouvez désormais vous attaquer à la partie visuelle ! :)

Pour la partie visuelle, je travail en Blueprint, c'est bien plus facile ;)

Visuel

Nous voici dans la dernière partie de ce tuto, un dernier petit effort et vous aurez enfin votre inventaire !

Si vous avez déjà utilisé les umg avec ue4, cette partie devrait être assez rapide, sinon suivez le guide !

Restons dans l'optique du résultat final. Je vous remet la photo pour que nous puissions commenter et préparer la chose.



Il va nous falloir donc récupérer les noms des items de l'inventaire, leur nombre et afficher le nombre et que le tout soit flexible.

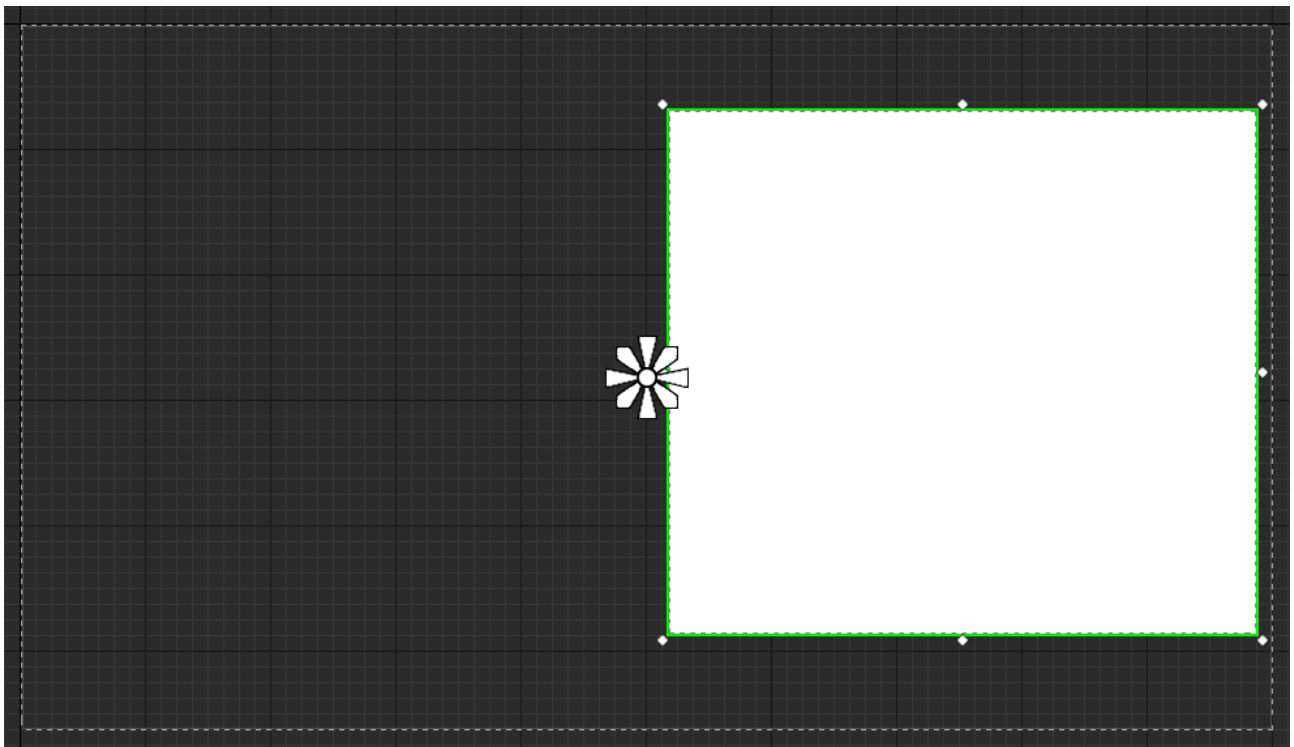
Nous allons donc commencer par créer le HUD que le joueur verra dans le jeu. Il est composé d'une fenêtre d'inventaire dans lequel on encapsulera des Items.

Dans le dossier **UMG**, commencez par cliquer droit et créez un **widget blueprint** que vous trouverez dans la catégorie **User Interface**. Nommez le **UMG_HUD**.

Ouvrez le widget HUD. Sur votre gauche vous constaterez qu'il y a la fenêtre **panel**. Elle va vous servir à créer facilement votre HUD. En haut vous avez les éléments que vous allez pouvoir inclure dans votre HUD et en bas la hiérarchie. Tout fonctionne à l'aide du **drag&drop**.

De base vous avez le Canvas Panel. Cet élément représente votre écran et peut être supprimé pour des plus petits objets (nous en reviendrons plus tard). Laissez le en place et commencez par ajouter une bordure (Border). Mettez la au centre de l'écran et agrandissez la. Vous remarquez qu'une « fleur » est apparu en haut de l'écran. C'est l'encrage de la fenêtre. Mettez l'ancrage au centre pour que la fenêtre s'adapte à tous les écrans. J'ai choisi de faire une fenêtre de 900 par 800.

Placez votre fenêtre sur la droite de l'écran. Les pointillés délimitent la taille de l'écran.



Pour un meilleur effet cliquer dans la hiérarchie sur la bordure et cherchez dans les **details** l'onglet **Appearance** et **Brush Color**. Mettez la fenêtre en noire avec 0.3 d'opacité.



Une petite sauvegarde et on continu !

Ajoutez sous la bordure une **Vertical Box** pour pouvoir mettre des widgets dedans. Ajoutez dedans ensuite un **Text** que vous modifierez en « Inventaire » et une **Vertical Box** que vous nommerez conteneur. Pour le text vous pouvez le mettre les **alignment** au milieu et laissez la taille en auto.



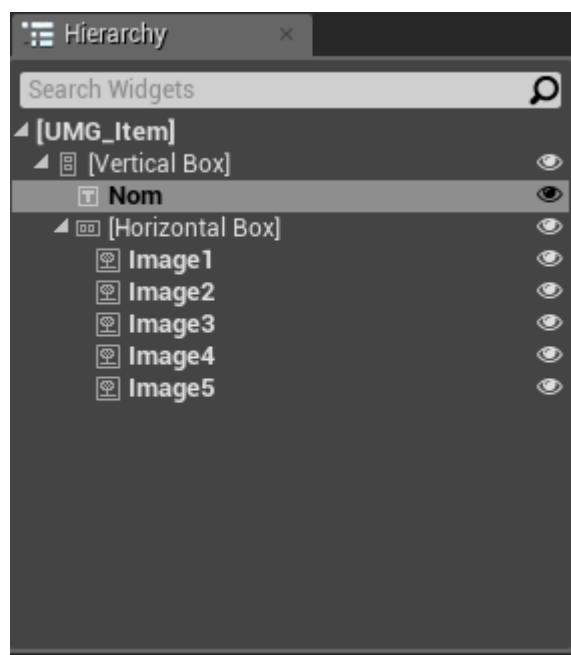
Mettez votre conteneur en **Is Variable** et sa taille en **Fill**. C'est dans ce conteneur qu'on va mettre les items.

Créez maintenant dans le dossier UMG un nouveau widget nommé **UMG_Item**.

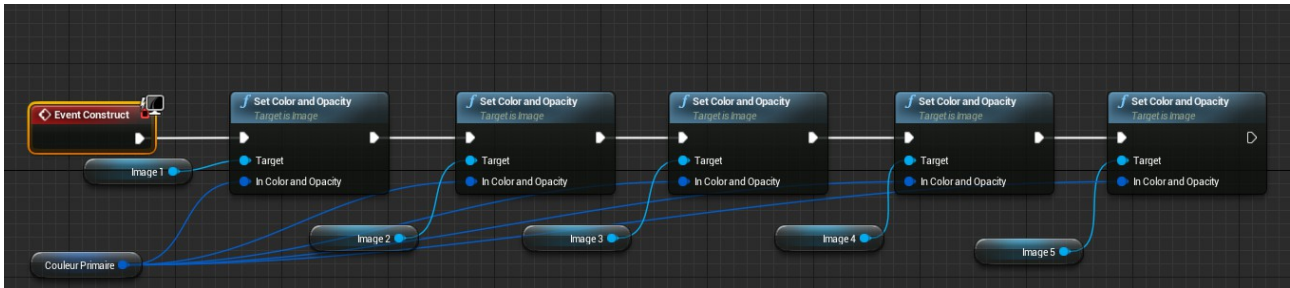
Ouvrez-le et **supprimez** le Canvas Panel. Ajoutez un **vertical box** et dedans mettez un **Text** et un **Horizontal Box**. L'horizontal box va stocker les images qui vont changer de couleur en fonction du nombre de cet item que vous aurez. Nous voulons 5 unités max donc nous allons ajouter 5 **images** dans l'**horizontal box**.

Ensuite sélectionnez toutes vos images, mettez la taille en **Fill** et un padding **left** et **right** de 10. Il ne reste plus qu'à mettre votre **Text** en **Is Variable** et mettre toutes vos **images** en **Is Variable** après les avoir renommés.

Vous devriez avoir ceci :

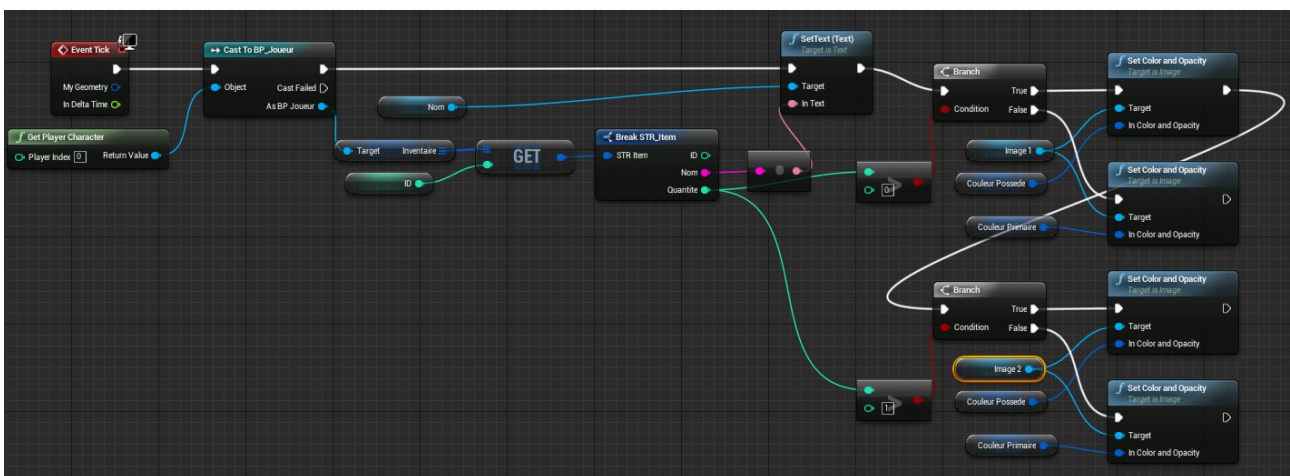


Passez dans le graph de l'UMG_Item et commencez par ajouter une variable de type **LinearColor** et de nom **CouleurPrimaire**. Cette couleur va servir quand votre item n'aura pas atteint la bonne quantité. Dans les **default value** mettez en **blanc** avec un alpha à **0.1**. Ajoutez une deuxième variable du même type et nommez là **CouleurPossede**. Mettez là en **blanc** avec un alpha à **1**. Récupérez toutes vos images et mettez les dans le graph. En partant de chaque image récupérez la fonction **SetColorAndOpacity**. Récupérez la **CouleurPrimaire** et reliez là à chaque **SetColorAndOpacity**. Reliez ensuite tous les **SetColorAndOpacity** entre eux et le premier à **l'Event Construct**. Toutes vos images seront à présent grisées dès le début de la partie.



Rajoutez maintenant un **ID** éditable (le petit œil) qui soit un **Integer**. Cet ID va représenter l'ID de l'Item correspondant.

Il va falloir maintenant actualiser le nombre d'objets possédés et changer la couleur des images en conséquence. Pour ça je vous propose un grand graph en image que je vous explique dans 2 secondes.

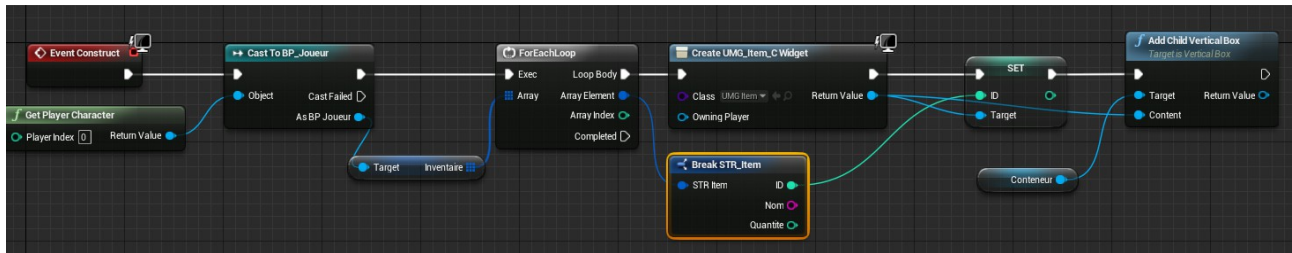


Dans ce graph on récupère le joueur, puis son inventaire et finalement l'item actuel. On met le nom de l'Item au bon endroit et on vérifie la quantité. Pour **chaque palier** on colore l'image de la bonne couleur. J'ai fais pour les 2 premières images sur la photo mais il faut le faire pour **toutes** les images en oubliant pas de **changer la valeur** que l'on test (0, 1, 2, etc..) et l'image.

Une fois ce schéma reproduit, votre Item est fonctionnel. Il ne reste plus qu'à ajouter le bon nombre d'Item dans le HUD au début de la partie.

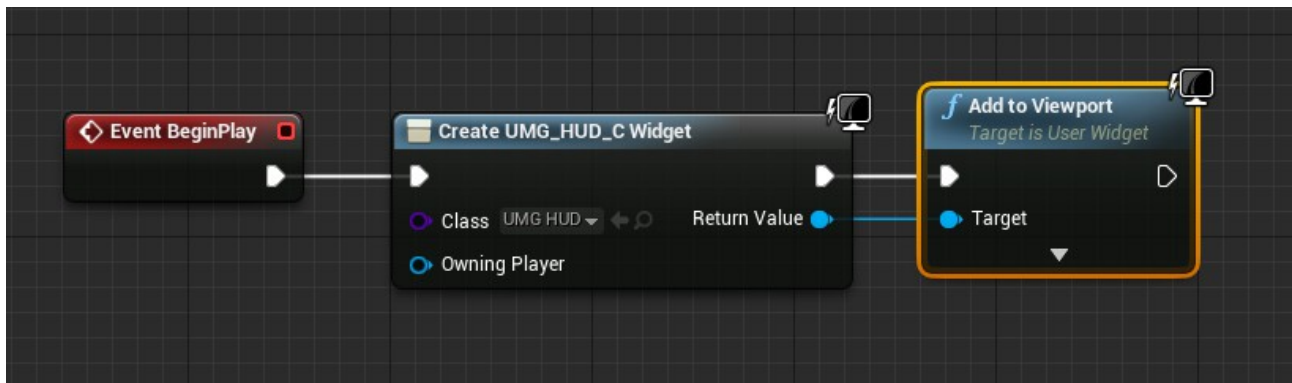
Pour cela on retourne dans le **HUD**.

Dans le graph on va récupérer le joueur et affecter les Items dès le début du jeu.

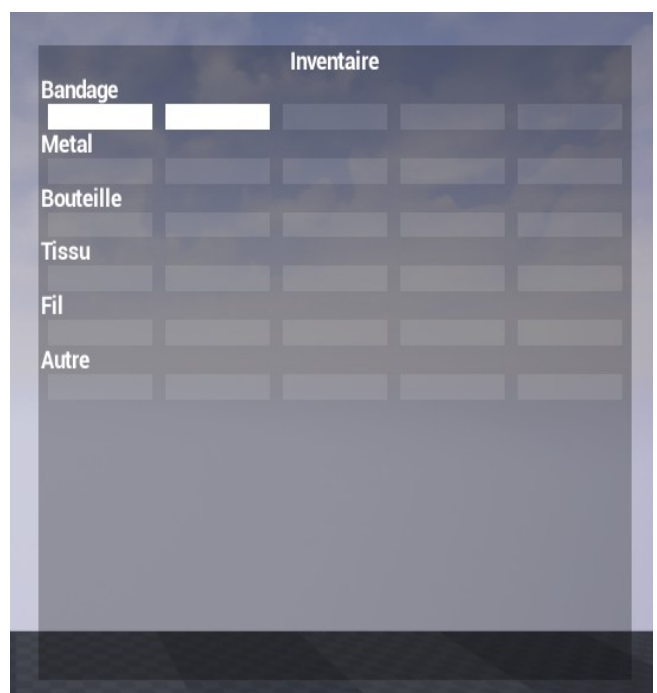
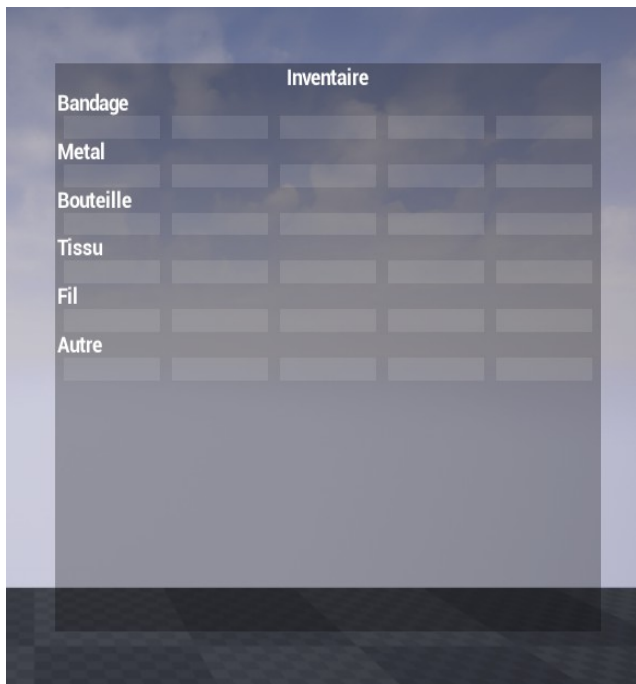


Voilà, c'est fini !

Pour pouvoir voir votre inventaire il faut pas oublier d'ajouter votre **HUD** à votre personnage. Rien de plus simple, on le crée sur le **BP_TutoJoueur** :)



Le résultat est le suivant :



On n'oublie pas de sauvegarder. Voilà, vous avez votre système d'inventaire ! :)

Petit bonus pour **ouvrir** et **fermer** votre inventaire.

Retournez dans le TutoJoueur sur Visual Studio. Nous allons changer la valeur d'une variable booléenne afin de dire au jeu si l'inventaire est ouvert ou non. Créez une variable de type booléen qui se nomme **InventaireVisuel**. Créez une fonction ChangeInventoryState de type void. Dans le TutoJoueur toujours ajoutez ceci :

```
// Change the inventory state
void ATutoJoueur::ChangeInventoryState()
{
    InventaireVisuel = !InventaireVisuel;
}
```

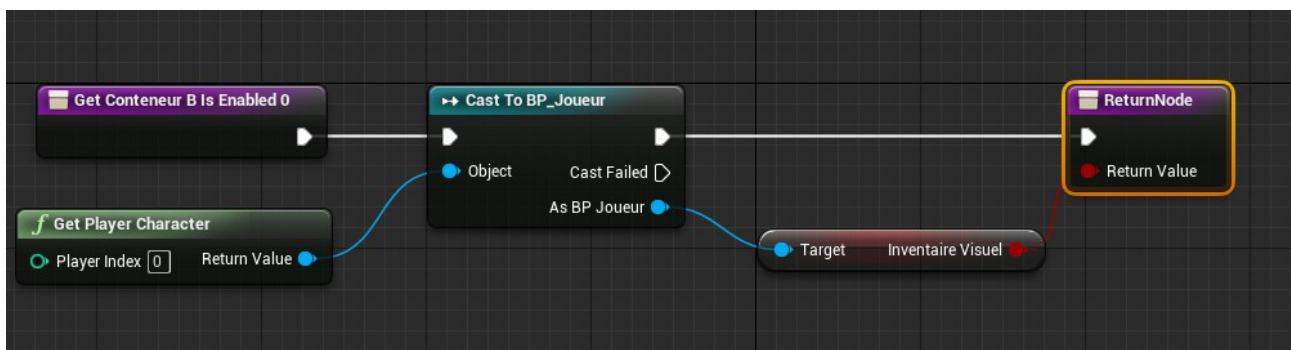
Ici, on inverse simplement la valeur du booléen.

Sauvegardez et compilez.

De retour dans le **UMG_HUD**, on va Binder (encore une fois) la visibilité et la fonctionnalité de l'inventaire.

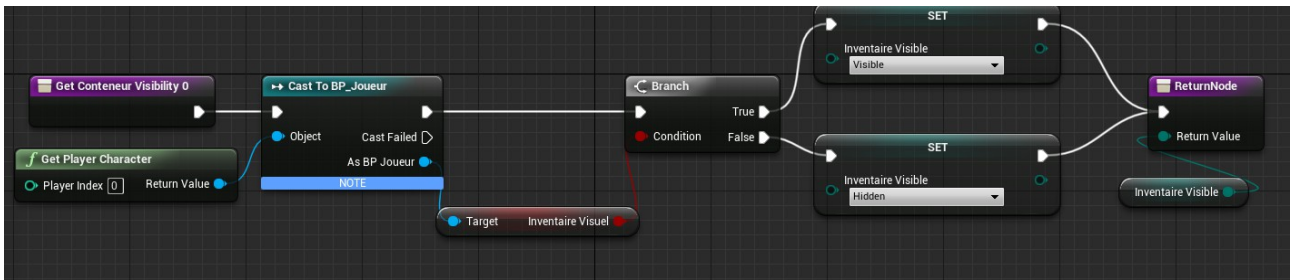
Cliquez tout d'abord sur la bordure et ensuite cliquez sur **Bind** à côté de **Is Enabled** et cliquez sur **Create Binding**.

Rajoutez ceci :



Pour le visuel, le graph va être un petit peu différent :

Créez une variable locale **InventaireVisible** de type **EslateVisibility** et mettez la à **Hidden** par défaut.



Si vous n'avez pas encore **bindé** votre touche **Inventaire**, c'est le moment !
Retournez dans **VS** et allez sur **TutoJoueur.cpp** .

```
InputComponent->BindAction("Inventaire", IE_Pressed, this,
&ATutoJoueur::ChangeInventoryState);
```

Lancez le jeu et appuyez sur I et magie ! Votre inventaire fonctionne ! :D

Pour toute éventuelles questions, remarques, détails ou demandes, n'hésitez pas à prendre contact avec moi ! (Cf : début du tuto)