Introduction

(NB : J'espère que ce tuto 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 le premier tuto nous avons mis en place un système d'inventaire et une manière de le visualiser. Dans le tuto d'aujourd'hui nous allons continuer sur ce projet et rajouter le ramassage des objets ainsi qu'un déplacement basique.

Je rappelle l'objectif final du tuto : un système inventaire/craft à la The last of us.

La dernière fois nous avions mis en place le carré de droite avec les items et leur nombre.
Aujourd'hui nous allons pouvoir remplir l'inventaire et se déplacer dans le monde pour aller chercher du loot.

Attention, ce tutoriel est le même que le Tuto 2 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 ;)*

Ramassage des objets dans le monde

Nous allons donc commencer par rendre un objet récupérable dans le monde et qu'il s'ajoute à l'inventaire.
Rappelez vous que vous avez créé des fonctions qui seront fort utiles dans le tuto d'aujourd'hui.

Avant de créer les objets nous allons rajouter un peu de **mouvement** dans notre scène.

Pour cela il faut créer **4 axes de mapping**. Pour rappel vous trouverez l'éditeur d'Input dans **Edit**, **Project Settings** et **Input**. Le premier va s'appeler Avant, il va être bindé sur Z avec 1.0 en puissance (scale).

Je vais résumer ce bind comme suit : **Axe (Avant, Z, 1.0)** , c'est comme ça que je vous donnerai les informations sur un bind dorénavant : Type(Nom, Touche, (optionnel)Valeur).

Vous allez donc devoir créer 4 axes comme suit :

Axe (Avant, Z, 1.0)
Axe (Avant, S, -1.0)

Axe (Droite, D, 1.0)
Axe (Droite, Q, -1.0)

Axe (Tourner, MouseX, 1.0)

Axe (Lever, MouseY, -1.0)

une fois ces axes créés, **rendez vous** dans **Visual Studio** dans la classe **TutoJoueur**. Ici nous allons rajouter du mouvement. Ce sont des déplacements basiques qui pourront vous servir dans vos projets futurs.

On va utiliser le **character movement** déjà présent pour gérer les déplacements.
Pour commencer avec les déplacements, Ajoutez **2 fonctions** à votre classe.

La première sera **Avant** de type **void** et prendra en **paramètre** un **float**. La seconde sera **Droite** de type **void** également et prendra aussi un **float** en **paramètre**.

```
//Move the player forward and backward
UFUNCTION(BlueprintCallable, Category = Deplacment)
void Avant(float value);

//Move the player left and right
UFUNCTION(BlueprintCallable, Category = Deplacment)
void Droite(float value);
```

Rajoutez vos deux bind de mouvement sur le joueur au bon endroit :

```
InputComponent->BindAxis("Avant", this, &ATutoJoueur::Avant);
InputComponent->BindAxis("Droite", this, &ATutoJoueur::Droite);
```

Ici je vais un peu décomposer les deux fonctions Avant et Droite afin que vous les compreniez bien :

```
//Move the player forward and backward
void ATutoJoueur::Avant(float value)
{
	if ((Controller != NULL) && (value != 0.0f))
	{
		//Find out which way is forward
		FRotator Rotation = Controller->GetControlRotation();
		//limit pitch when walking or falling
		if (GetCharacterMovement()->IsMovingOnGround() ||
GetCharacterMovement()->IsFalling())
		{
			Rotation.Pitch = 0.0f;
		}
		//add movement in that direction
		const FVector Direction =
FRotationMatrix(Rotation).GetScaledAxis(EAxis::X);
		AddMovementInput(Direction, value);
	}
}
```

Dans cette fonction, on **vérifie** tout d'abord qu'il **existe** un **controller** pour le **joueur** (pas la manette hein ;)) et que la **valeur** ne soit **pas nulle**. On récupère ensuite la **rotation** du **joueur** et on **l'affecte** à **Rotation**. On **bloque** la **rotation** avant/arrière quand on est en l'air ou que l'on marche. On récupère ensuite le **vecteur** de **devant** (x) par **rapport** à notre **rotation**. Finalement on **applique** un **mouvement** vers **l'avant** de la **valeur** de la touche (vers l'arrière si c'est négatif).

Pour aller à droite, on fait globalement la même chose avec le vecteur de droite (y) sans se soucier du blocage de rotation.

```
//Move the player left and right
void ATutoJoueur::Droite(float value)
{
	if ((Controller != NULL) && (value != 0.0f))
	{
		//Find out which way is forward
		FRotator Rotation = Controller->GetControlRotation();
		//add movement in that direction
		const FVector Direction =
FRotationMatrix(Rotation).GetScaledAxis(EAxis::Y);
		AddMovementInput(Direction, value);
	}
}
```

Si vous testez votre projet maintenant, vous pouvez voir que vous pouvez vous **déplacer** mais pas bouger la sourie. Rajoutons un peu de mouvement à la **caméra** pour avoir un déplacement basique et pouvoir ramasser des objets plus facilement :

On va simplement utiliser les **fonctions** de **base** du **character**.
On ajoute donc **Tourner** et **Lever** aux bind déjà existants :

```
InputComponent->BindAxis("Tourner", this, &ATutoJoueur::AddControllerYawInput);
InputComponent->BindAxis("Lever", this, &ATutoJoueur::AddControllerPitchInput);
```

Ici on demande au **character** d'ajouter de la **rotation horizontale** quand on **tourne** et de la **rotation verticale** quand la sourie **monte** ou **descend**.

(Si comme moi vous avez des erreurs de compilation sur vos blueprint, fermez UE4 et réouvrez votre projet, parfois le quick reload marche pas. Le **quick reload** c'est la **compilation** des fichiers .**h** et .**cpp**. Et si vous avez encore des erreurs après, vérifiez que sur vos blueprint concernés, tous les pins soient connectés)

Vous avez maintenant un **déplacement** de personnage basique et classique. (C'est le déplacement utilisé dans les projets de base 3^{e} et 1e personne)

Le Loot

Il est temps de passer aux choses sérieuses et de revenir sur nos objets **ramassables** (loot).

Un **loot** va être constitué d'un **static mesh** qui sera la forme que verra le joueur dans le jeu, d'une **sphère de collision** afin de détecter que le joueur est près de l'objet et enfin d'un **ID** qui sera utilisé pour ajouter le bon loot à l'inventaire.

On va commencer par créer l'Item. C'est une classe parent. Nous allons faire de l'héritage par la suite pour les autres loot. Je vous explique toutes les étapes pour créer un loot et je vous laisserai le soin de créer les autres objets par vous même ou de simplement créer des classes filles ;)

Pour **l'Item**, on va utiliser une représentation de **sphère** et rajouter une **sphère de collision** autour pour détecter le joueur. Sans oublier de mettre un **ID** pour ajouter le loot au moment du ramassage.

On commence par ajouter une nouvelle **classe** de type **Actor**. Nommez là **TutoItem**.

On va **ajouter** une **sphère collision**. Elle servira à **détecter** quand un **acteur** touche l'objet et donc quand le **joueur** pourra le **ramasser**. On va mettre cette **collision** en **root** (racine) sur l'objet et on lui ajoutera une **sphère** pour avoir un **visuel** dans le blueprint.

Dans le .**h** ajoutez le **constructeur** de la classe, la **sphère component** ainsi que les deux **fonctions** qui serviront à **récupérer** les **collisions** :

```
// constructor of the class
ATutoItem(const FObjectInitializer& ObjectInitializer);

// Sphere collision component
UPROPERTY(VisibleDefaultsOnly, Category = Projectile)
USphereComponent* CollisionComp;

// called when something overlap the item
UFUNCTION()
void OnBeginOverlap(class AActor* OtherActor, class UPrimitiveComponent*
OtherComp, int32 OtherBodyIndex, bool bFromSweep, const FHitResult &
SweepResult);

// called when something end overlap the item
UFUNCTION()
void OnEndOverlap(class AActor* OtherActor, class UPrimitiveComponent*
OtherComp, int32 OtherBodyIndex);
```

Ensuite, on va **créer** la **sphère collision** dans le **constructeur** :

```
ATutoItem::ATutoItem(const FObjectInitializer& ObjectInitializer)
{
	//Use a sphere as a simple collision representation
	CollisionComp =
ObjectInitializer.CreateDefaultSubobject<USphereComponent>(this,
TEXT("SphereComp"));
	CollisionComp->InitSphereRadius(100.0f);
	CollisionComp->OnComponentBeginOverlap.AddDynamic(this,
&ATutoItem::OnBeginOverlap);
	CollisionComp->OnComponentEndOverlap.AddDynamic(this,
&ATutoItem::OnEndOverlap);
	RootComponent = CollisionComp;
}
```

On créé donc l'objet et on l'appel SphereComp. On lui donne un diamètre de 100 unités et on bind les deux fonctions sur l'objet. Pour finir on l'affecte en tant que root.

De retour dans le **.h**, on va ajouter les **informations** de **l'objet** :

```
// Item information
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Item")
FItem info;
```

N'oubliez pas d'ajouter en haut de **TutoItem.h** les deux **entêtes** :

```
#include "TutoDataBase.h"
#include "TutoJoueur.h"
```

Sans ces entêtes votre code ne **compilera pas**. Mettez les **avant** le **generated.h** et **après Actor.h** .

Avant de continuer, on va rajouter, sur le joueur, **l'item** qu'il va pouvoir **ramasser**.
Rendez vous donc dans **TutoJoueur.h** et ajoutez les lignes suivantes :

```
// The item to pickup
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Gameplay)
AActor* ItemToPickUp;
```

Maintenant que le joueur sait quel item il va pouvoir ramasser, il va falloir lui dire quand il rentre dans le collider d'un item qu'il peut le ramasser.

Pour ça on va **tester** la **collision** de l'Item, **vérifier** qu'il s'agit du **joueur** et lui **affecter** le nouvel **Item**.

Ci-dessous, le code quand le joueur entre dans la collision :

```
void ATutoItem::OnBeginOverlap(class AActor* OtherActor, class
UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromSweep, const
FHitResult & SweepResult)
{
	auto MyPC = Cast<ATutoJoueur>(OtherActor);
	if (MyPC)
	{
		if (GEngine)
		{
			GEngine->AddOnScreenDebugMessage(-1, 5.f, FColor::Blue,
TEXT("PLAYER ENTER!!"));
		}
		MyPC->ItemToPickUp = this;
	}
}
```

On cast l'objet qui rentre (otherActor) en AtutoJoueur. Le type est en auto, c'est le compilateur qui va gérer le type tout seul. On vérifie que le cast à fonctionné, on affiche un message de debug et on affecte l'Item sur l'Item à ramasser sur le joueur.

Pour la sortie c'est globalement la même chose :

```
void ATutoItem::void OnEndOverlap(class AActor* OtherActor, class
UPrimitiveComponent* OtherComp, int32 OtherBodyIndex)
{
	auto MyPC = Cast<ATutoJoueur>(OtherActor);
	if (MyPC)
	{
		if (GEngine && DebugMessage)
		{
			GEngine->AddOnScreenDebugMessage(-1, 5.f, FColor::Blue,
TEXT("PLAYER EXIT!!"));
		}
		MyPC->ItemToPickUp = nullptr;
	}

}
```

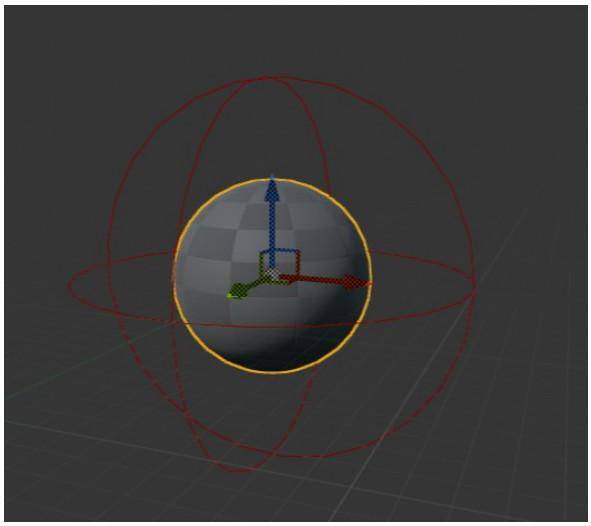
On affecte nullptr qui est un pointer null sur l'Item à ramasser sur le joueur. Il ne pourra donc rien ramasser.

Il y a pas mal de nouvelles choses, essayons de **compiler** pour voir si tout se passe bien ;)
La compilation peut être longue. Prenez vous un petit café ! Le temps peut varier en fonction de votre ordinateur, de votre version de UE4, de ce que vous avez d'ouvert en terme d'applications, si vous avez le focus sur UE4 (gardez le dessus, ça ralenti beaucoup de faire autre chose pendant la compilation), etc.
Si vous avez aucunes erreurs, félicitations ! Vous avez bien fait votre boulot !
Si vous avez des erreurs, ce n'est pas bien grave, ça doit être une petite erreur ou une coquille. Regardez où se situe votre erreur grâce à la fenêtre de message (Window, Developper tools, Message Log) et prenez l'erreur la plus haute.

Vous pouvez maintenant créer un blueprint de votre classe TutoItem. Appelez le BP_TutoItem.

Ajoutez une sphère en enfant et sauvegardez.

Vous devriez avoir ceci :



Il est l'heure de mettre à profit votre mémoire !

Il faut maintenant ajouter un mapping d'action pour ramasser l'objet. Rajoutez ce qui suit :

Action (Action, E)

Il fallait en effet rajouter un bind d'action avec la touche E, bravo à ceux qui l'avait déjà fait ! Ça commence à rentrer ;)

De **retour** dans le code de **TutoJoueur**, on va faire en sorte de **ramasser l'objet** et de **l'ajouter** dans l'inventaire **si c'est possible**.

Commencez par ajouter une **fonction** dans **TutoJoueur.h** :

```
//handles use
UFUNCTION()
void OnUse();
```

Cette fonction va être **utilisée** quand vous allez **appuyer** sur **E**.

Devinez quoi ? Il faut la **binder** dans **TutoJoueur.cpp** sur la touche **Action**.

Je vous laisse le faire et vous donne la solution page suivante…

Il fallait rajouter la ligne suivante :

```
InputComponent->BindAction("Action", IE_Pressed, this, &ATutoJoueur::OnUse);
```

Il va falloir vous concentrez à partir de maintenant car je vais vous balancez un pavé de code dans la figure. Ça va faire mal je sais mais ne paniquez pas, je vais tout vous expliquez, si vous n'avez pas déjà tout compris par vous même ;)

Juste avant cela, ajoutez cet **include** dans **TutoJoueur.h** pour que ça fonctionne :

```
#include "TutoItem.h"
```

Voici le code :

```
// handle use
void ATutoJoueur::OnUse()
{
	//if we are in a pickup collision
	if (ItemToPickUp)
	{
		if (GEngine)
		{
			GEngine->AddOnScreenDebugMessage(-1, 5.f, FColor::Yellow,
TEXT("picked Up!"));
		}
		//cast the item to a pickup item
		ATutoItem* TheItem = Cast<ATutoItem>(ItemToPickUp);
		if (TheItem)
		{
			if (GEngine)
			{
				GEngine->AddOnScreenDebugMessage(-1, 5.f, FColor::Blue,
TEXT("Infos retreived, id:" + FString::FromInt(TheItem->info.ID)));
			}
			//update the inventory of the player
			int32 index = GetItemIndexWithID(TheItem->info.ID);
			//the item don't exist, we can add it to the inventory
			if (index == -1)
			{
				FItem NewItem = TheItem->info;
				int32 TheIndex = Inventaire.Add(NewItem);
			}
			//the item already exist, we can update the quantity
			else
			{
				Inventaire[index].Quantite++;
			}
		}
		else
		{
			if (GEngine)
			{
				GEngine->AddOnScreenDebugMessage(-1, 5.f, FColor::Red,
TEXT("Fail to retreive infos"));
			}
		}
		ItemToPickUp->Destroy();
		ItemToPickUp = nullptr;
	}
}
```

Qu'est-ce qu'on fait dans ce code ?

Tout d'abord on vérifie qu'on a un objet à récupérer, on affiche un message si c'est le cas, on cast l'item pour pouvoir utiliser ses attributs, si l'item est bien casté, on affiche un message avec son ID, on récupère son index dans l'inventaire et si l'index est négatif nous ajoute l'objet dans l'inventaire, sinon on lui ajoute 1 en quantité. Si l'item ne se caste pas on affiche un message d'erreur. Pour finir on détruit l'Item et on nettoie l'Item à ramasser.

Facile non ? ;)

On sauvegarde et on compile !

Vous pouvez à présent faire un **clique droit** sur le blueprint **BP_TutoItem** et faire un **Create Child Blueprint Class** . Une fois nommé **Bandage** et ses infos modifiés, vous pouvez faire une **drag&drop** du **bandage** dans le monde et jouer. Approchez vous, faite **E** et magie ! L'objet **disparaît** et si vous ouvrez votre **inventaire** vous avez un bandage en plus ! :)

Essayez en ajoutant les autres loot et en lançant le jeu. N'oubliez pas de **changer** l'ID du nouveau loot.

Vous pouvez maintenant mettre des loot dans le monde et remplir votre inventaire. :)

Dans le prochain tuto vous allez réutiliser tout ce qu'on a fait jusqu'à présent et faire du craft !

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