UE4 TRUCS ET ASTUCES

utilisation de la version 4.8.3

- faire les structures en dehors des classes et commencer le nom par un 'F' :

```
/** REQUIREMENT */
USTRUCT(BlueprintType)
struct FRequirement
{
	GENERATED_USTRUCT_BODY()

	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Requirement")
	int32 Id;

	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Requirement")
	int32 Quantity;

	FRequirement()
	{
		Id = -1;
		Quantity = 1;
	}
};
```

- les types primitifs sont :

```
	float TeaWeight;

	int32 TeaCount;

	bool bDoesTeaStink;

	FName TeaName;

	FString TeaFriendlyName;

	UClass* TeaClass;

	USoundCue* TeaSound;

	UTexture2D* CrosshairTex;

	UTexture* TeaTexture;

	AActor* Player;

	TSubclassOf<class AFPSProjectile>

	TArray<FItem> Inventory;

	USkeletalMeshComponent* FirstPersonMesh;

	UCameraComponent* FirstPersonCameraComponent;
```

- les fonctions utilisables dans les blueprints s'écrivent ainsi :

```
UFUNCTION(BlueprintCallable, Category = Gameplay)
void OnFire();
```

- les variables utilisables dans les blueprints s'écrivent ainsi :

```
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category=Camera)
      UCameraComponent* FirstPersonCameraComponent;
```

- une variable peut être `VisibleAnywhere` (visible dans les defaults et in game) ou `VisibleDefaultsOnly` (visible dans les defaults uniquement) ou `EditAnywhere` (éditable n'importe où) ou `EditDefaultsOnly` (éditable uniquement dans les défaults)

- une variable peut être `BlueprintReadOnly` (uniquement lisible dans les blueprints) ou `BlueprintReadWrite` (lisible et scriptable dans les blueprints)

- fonctions utiles :

```
//utilisée en début de jeu
virtual void BeginPlay() override;

//utilisée à chaque frame
virtual void Tick( float DeltaSeconds ) override;

//!!utilisée dans les classes qui extandent AHUD!!
virtual void DrawHUD() override;

//utilisée pour détécter une collision
void OnHit(class AActor* OtherActor, class UPrimitiveComponent* OtherComp, FVector
NormalImpulse, const FHitResult& Hit);
```

- constructeur d'une classe (utilisé pour créer des objets, les attacher et définir les variables) :

```
//dans le .h
AFPSProjectile(const FObjectInitializer& ObjectInitializer);


//dans le .cpp
AFPSProjectile::AFPSProjectile(const FObjectInitializer& ObjectInitializer) :
Super(ObjectInitializer)
{
...
}
```

- créer un composant et l'attacher :

```
//create a CameraComponent
FirstPersonCameraComponent =
ObjectInitializer.CreateDefaultSubobject<UCameraComponent>(this,
TEXT("FirstPersonCamera"));
//attach it to the root component
FirstPersonCameraComponent->AttachParent = CapsuleComponent;
```

- variables utiles :

```
// Set this character to call Tick() every frame.  You can turn this off to improve
performance if you don't need it.
PrimaryActorTick.bCanEverTick = true;

//Allow the pawn to control rotation
FirstPersonCameraComponent->bUsePawnControlRotation = true;

//only the owning player will see this mesh
FirstPersonMesh->SetOnlyOwnerSee(true);

//don't use shadow for this model
FirstPersonMesh->bCastDynamicShadow = false;
FirstPersonMesh->CastShadow = false;

//the owner don't see the model
Mesh->SetOwnerNoSee(true);
```

- retrouver un objet créé :

```
// Set the crosshair texture
static ConstructorHelpers::FObjectFinder<UTexture2D>
CrosshairTextObj(TEXT("Texture2D'/Game/models/crosshair.crosshair'"));
CrosshairTex = CrosshairTextObj.Object;
```

- afficher un message à l'écran :

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

- utiliser une classe :

```
HUDClass = AFPSHUD::StaticClass();
```

- retrouver une classe d'un blueprint :

```
//set the default pawn class to our Blurprinted character
static ConstructorHelpers::FClassFinder<APawn>
PlayerPawnObject(TEXT("Pawn'/Game/Blueprint/BP_FPSCharacter.BP_FPSCharacter_C'"));
if (PlayerPawnObject.Class != NULL)
{
	DefaultPawnClass = PlayerPawnObject.Class;
}
```

- créer un objet et l'affecter en tant que root :

```
CollisionComp = ObjectInitializer.CreateDefaultSubobject<USphereComponent>(this,
TEXT("SphereComp"));
RootComponent = CollisionComp;
```

- caster un objet :

```
auto MyPC = Cast<AFPSCharacter>(OtherActor);
if (MyPC)
{
//cast réussi
}
```

- conversions :

int32 en string : `Fstring::FromInt(int32) ;`

float en string : `FString::SanitizeFloat(YourFloat);`

- conseils divers :

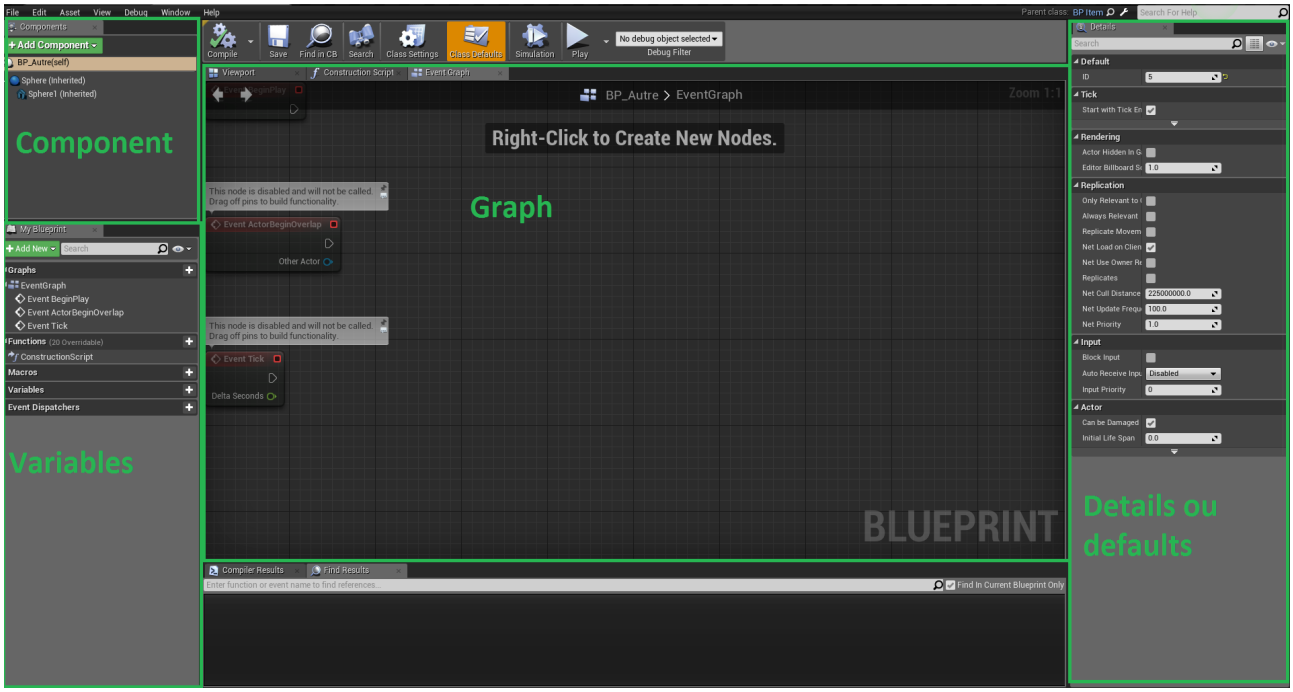
* En cas d'override de méthodes, utiliser le "Super" de cette méthode avant tout !

* Pour trouver le chemin d'un objet : clique droit dessus → copy reference (ex : Blueprint'/Game/Blueprint/BP_FPSProjectile.BP_FPSProjectile')

* Se référer au wiki et à la doc pour trouver des fonctions et des variables utiles.

* Utiliser des composants utiles et déjà présents tels que les mouvements components et autre.

* Le tuto "First Person Shooter" est un bon entrainement, voir https://wiki.unrealengine.com/First_Person_Shooter_C%2B%2B_Tutorial



Voici les noms des sections dans la fenêtre d'un blueprint. J'utilise ces noms dans tous mes tutos.

- Pour utiliser des **USoundCue*** dans votre code et votre blueprint, vous devez spécifier votre variable en ***EditDefaultOnly***. Autrement votre variable ne marchera pas.

```
UPROPERTY(EditDefaultOnly, Category = Gameplay)
UsoundCue* son ;
```

- Pour utiliser les **UMediaPlayer** vous devez les spécifiez en **ReadOnly** et **ajouter** la **librairie** dans votre [projet].build.cs de votre projet qui se trouve dans [projet]/Source/[projet]/[projet].Build.cs

Vous trouverez une ligne ressemblant à celle-ci :

PublicDependencyModuleNames.AddRange(new string[] { "Core", "CoreUObject", "Engine", "InputCore", "MediaAssets"});

Rajoutez le MediaAssets comme écrit ci-dessus.

Il se peut que vous ayez des erreurs en voulant utiliser le UMediaPlayer dans votre .cpp .
La petite astuce est de mettre le mot clé "**class**" devant le type de votre variable pour qu'elle soit acceptée.

Cette astuce est valable pour pleins d'autres type. Si votre code ne compile pas un type de variable, essayez de mettre "class" devant peut vous sauver.

- Pour faire un **random** dans votre code, il faut utiliser la librairie **FMath**.
Utilisez la comme ceci :

```
int rand = Fmath::RandRange(x,y);
```

A noter que le RandRange vous renvoi que des int de base.

- Pour avoir un **focus** sur le jeu directement sans avoir à cliquer sur la fenêtre, vous pouvez vous rendre dans les blueprint et ajouter les deux nodes suivant :

SetFocusToGameViewport et ***SetInputModeGameOnly***. Ce dernier a besoin du PlayerController pour fonctionner.

- Pour faire des **logs** dans votre jeu, ce qui peut être plutôt pratique, vous pouvez utiliser les méthodes suivantes :

```
UE_LOG(LogTemp, Warning, Text(" … "));
```

Pour afficher des **variables** dans vos **logs**, vous devez utiliser un formatage particulier :

string : `UE_LOG(LogTemp, Warning, Text(" …%s"), leString);`
int : `UE_LOG(LogTemp, Warning, Text(" …%d"), leInt);`

plus de formatage à cette page :

http://www.cplusplus.com/reference/cstdio/printf/

Pour les logs, je recommande de définir une **convention** de **nommage**. Par exemple voici la mienne :

```
UE_LOG(LogTemp, Warning, Text("Fichier.cpp : Methode : Log") ;
```

Il vous sera ainsi plus facile de vous retrouvez dans vos logs et de savoir d'où vient quel log.

- Pour les gens qui voudraient faire de la **prédiction client** ça marche comme ceci :

1) On fait l'action sur le client
2) On check sur le serveur la validité du mouvement
3) Le serveur renvoi une confirmation
4) Si le serveur affirme votre mouvement, on l'annule sur le client.

- Si vous faites du réseau, n'oubliez pas d'utiliser le **PlayerState** qui est très utile et qui est **répliqué** à tous les clients. N'oubliez pas de le **recompiler après** la compilation de **visual**, il peut être un peu capricieux.

- Pour écrire une **fonction** qui doit être **exécutée** sur le **serveur** et appelée par un client, je procède comme suit :

Fire *→ Si Role == Authority → Do action*
→ Si Role < Authority → `ServFire()`

ServFire() *=> cette fonction ne doit pas être définie, elle va appeler l'implémentation et le validate.*
ServFire_Implementation() *→* `Fire()` *=> cette fonction appelle Fire avec l'authority*
ServFire_Validate() *→* `return true` *=> cette fonction renvoi toujours true*

- Si vous utilisez un **joueur comme serveur**, ne faites **pas** de **multicast** ou vous vous retrouverez avec une **boucle infinie** car votre fonction fera Serveur → joueur, Serveur → joueur, etc.

- Si vous avez **deux constructeurs** dans votre header, utilisez celui qui a des **paramètres** pour définir vos variables et ajouter des composants à votre classe.

- Pour faire des headshots et visez certaines parties du corps, utilisez le **physical body** pour avoir des **collisions localisées**.

- Le **GameInstance** est **persistant** entre les scènes, n'hésitez à l'utiliser pour **sauvegarder** des données.

- Si vous avez des **problèmes d'animOffset**, vérifiez de bien mettre la **même animation** de référence sur le **SelectedAnimationFrame**.

- Quand vous vous déplacez, si vous voyez que vous avez des **bugs** de **ralentissements** un peu aléatoire, c'est que votre **pitch** n'est **pas bloqué** et donc votre capsule frotte le sol !

- Si vous faites du **réseau** en **serveur dédié**, vous n'avez **pas** besoin de **répliquer** en **multicast** tous les états et fonctions. Le serveur réplique déjà à tous les joueurs.

- Pour **afficher** la **sourie** en jeu, utilisez le **ShowMouseCursor** du **PlayerController**. Vous voudrez peut-être également ajouter le **SetInputGame/UIOnly** pour avoir les input sur l'UI de votre jeu.

- Et n'hésitez pas à tester par vous même pleins de choses ! :D

- Pour faire une recherche sur tous les objets d'un type, équivalent au GetAllObjectOfClass en blueprint, vous pouvez faire en code :

```
for (TActorIterator<MaClasse> ActorItr(GetWorld()); ActorItr; ++ActorItr)
{
      Camera = *ActorItr;
}
```

ici ma caméra va référencer un objet de type MaClasse (utile pour trouver un seul objet d'une classe ou pour faire des tableaux d'objets)