Module Moteur de jeux: Unreal Engine

Travaux Pratiques 8

Unreal Engine et C++

Objectif: L'objectif de cette séance est d'être capable d'intégrer et interagir avec des éléments à partir du langage C++. Plus précisément, après cette séance, vous serez capable de :

- Créer une classe C++ et l'intégrer dans *Unreal Engine*.
- Gérer l'ajout d'effet de particule dans votre objet et rajouter un maillage.
- Rajouter des variables et les rendre éditables en *Blueprint* et *Unreal Engine*.
- Rendre des méthodes de classe C++ disponibles en *Blueprint*.
- Créer une classe *Blueprint* à partir d'une classe C++.

Travail à rendre sur moodle : à la fin de la séance vous devez déposer une archive contenant les captures d'écran permettant de noter l'avancement de chaque question avec idéalement une vidéo montrant le résultat final de votre travail.
A noter : les questions précédées de M2 (resp. DU) seront prises en compte pour la notation des étudiants en M2 (resp. DU) uniquement, les autres étudiants peuvent passer la question.

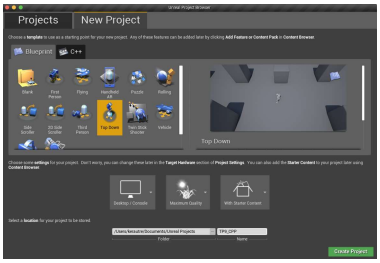
Partie I: Préliminaire

Avant de commencer ce TP, il eut être utile de vérifier que *Visual Studio* est bien installé (pour un environnement sous *Windows*). D'autres IDE peuvent être utilisés, mais dans ce TP, nous suivrons une procédure standard préconisée par *Unreal Engine* (illustrée dans ce TP avec l'interface de *Visual Studio Code*). A noter qu'un simple éditeur de texte peut être utilisé en cas de soucis de lancement de *Visual Studio* car *Unreal Engine* permet de compiler directement le code.

M2 1.1 Vérifiez que *Visual Studio* est bien installé et dans le cas contraire, tentez une installation rapide puis vérifiez que vous avez les composants illustrés sur l'image (b) ci-dessous. Sinon repérez un simple éditeur de texte que vous utiliserez pour éditer votre code (par exemple *Bracket*).

Orientation du TP :
Unreal Engine permet de créer un projet de départ orienté C++, mais il est aussi possible de rajouter des classes C++ dans un projet déjà existant qui n'a pas été configuré pour utiliser C++. Pour montrer qu'il est possible d'intégrer et mélanger du C++ et *Blueprint* dans un projet *Unreal Engine*, nous allons donc créer un nouveau projet sans utiliser le modèle C++ mais en sélectionnant un projet classique comme pour les séances précédentes.

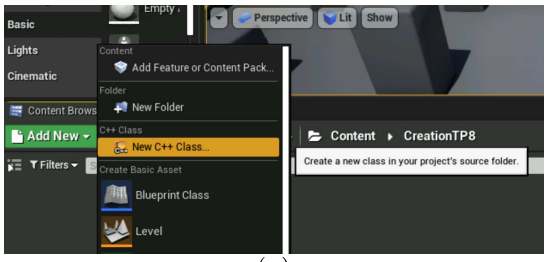
1.2 Créez un nouveau projet en utilisant le template *Top Down* de l'onglet *Blueprint*.



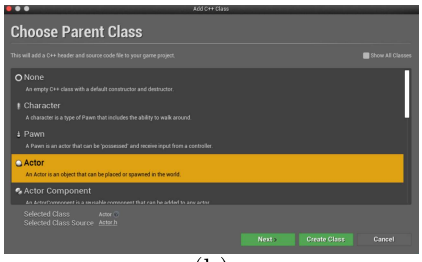
Partie II: Première classe C++

Dans cette partie, nous allons tout d'abord créer une première classe minimale associée à un Acteur (`Actor`) et testez les logs pour vérifier que la création de classe est bien fonctionnel.

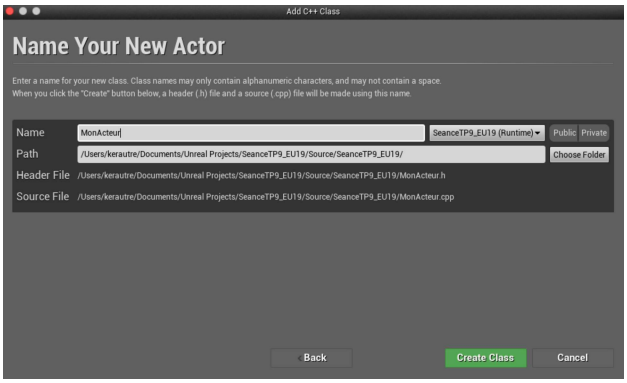
2.1 Pour créer une première classe C++, il faut procéder comme pour une classe *Blueprint*, en sélectionnant l'explorateur de contenu, puis sélectionner *New C++ Class* (voir image (a) ci-dessous).



(a)



(b)

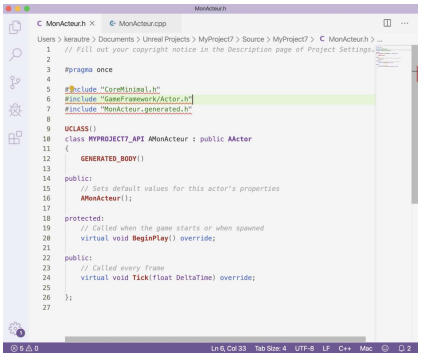


(c)

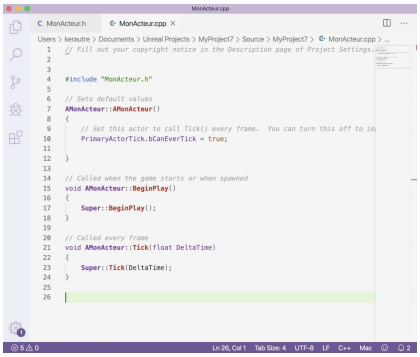
2.2 Sélectionnez ensuite la classe `Actor` pour définir la classe parent (voir image (b) ci-dessus) et nommez votre classe `MonActeur` et repérez le chemin où UE4 ajoute les deux fichiers sources `MonActeur.h` et `MonActeur.cpp`.

L'étape suivante prend un certain temps car UE4, compile vos deux nouveaux fichiers pour créer du code et les intègre dans votre projet. Si tout se passe correctement, vous devriez avoir l'éditeur associé qui se lance et vous ouvre deux fichiers sources associés aux deux fichiers précédents (voir image ci-dessous).

A noter : Il est conseiller de fermer et re ouvrir votre projet de façon à faire apparaître le répertoire contenant les objets C++ de l'image (a) de la plage suivante ou de résoudre des soucis d'affichage de log des questions suivantes.

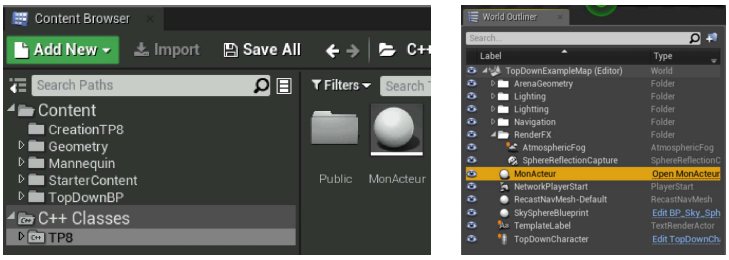


(a) fichier .h (*header*)



(b) fichier .cpp

2.3 Rajoutez l'acteur que vous venez de créer dans la scène de votre jeu.

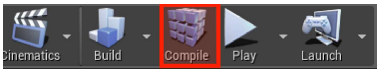


(a) (b)

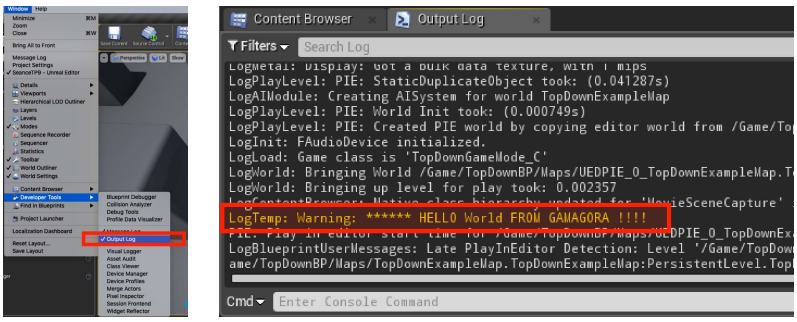
2.4 Revenez dans votre éditeur et dans le fichier `MonActeur.cpp`, et observez les points communs avec le code avec les structures *Blueprint* que vous connaissez déjà. Pour tester votre première ligne de code en C++ dans *Unreal Engine*, rajoutez le code suivant dans la méthode associée au lancement du jeu :

```
UE_LOG(LogActor, Warning, TEXT("******_HELLO_World_FROM_GAMAGORA_!!!!"));
```

Compilez ensuite votre code avec l'icone *compile* :



puis lancez votre projet après avoir activé la visualisation des *logs* (voir image (a)).



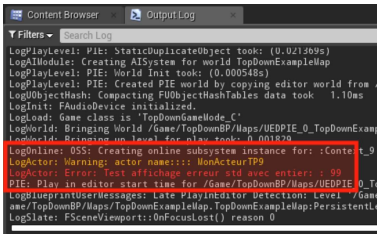
(a) (b)

Si tout se passe correctement vous devriez voir le message en jaune dans votre fichier de sortie.

2.5 Afin d'afficher le contenu d'une variable nous pouvons utiliser un système comparable à printf avec les paramètres suivants :

```
    UE_LOG(LogActor, Warning, TEXT("Actor_name_%s"), *this->GetName())
```

2.6 Sachant que le mécanisme est le même que pour printf, testez de créer une variable de type entier, que vous afficherez en sortie standard.



M2 2.7 La classe d'UE4 de type `FString` permet d'enregistrer une chaine de caractères construite de la même façon qu'avec la fonction précédente. Par exemple pour stocker le nom du joueur dans une variable de type `FString`, il faut écrire :

```
// variable nom de type FString
FString nom = FString::Printf(TEXT("Actor_name_%s"), *(this->GetName()));
```

Utilisez cette variable pour l'afficher dans les logs. Procédez avec la même syntaxe que pour la question 2.5 (ne pas oublier d'utiliser l'opérateur * comme pour l'exemple de la question 2.5).

Partie III: Acteurs, maillages et premières manipulations

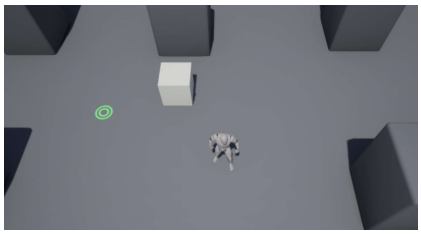
Après avoir vu comment créer son propre acteur et afficher des informations, nous allons voir comment intégrer d'autres éléments à l'objet C++ et voir comment rendre ces éléments visibles dans l'interface d'UE4.

3.1 Dans le fichier `MonActeur.h`, avant la section *public*, déclarez un nouvel objet `MonMaillage` de type `UStaticMeshComponent*`. Afin que l'objet soit visible dans l'interface d'*Unreal Engine*, rajoutez la macro `UPROPERTY(VisibleAnywhere)` juste avant votre déclaration. Cet objet sera la représentation visuelle de notre acteur.

3.2 Maintenant, il reste à créer le maillage dans l'implémentation de la nouvelle classe `MonActeur` (dans son constructeur). Recopiez les lignes 11 à 21 du code ci-dessous dans votre constructeur :

```
7  AMonActeur::AMonActeur()
8  {
9      // Set this actor to call Tick() every frame.  You can turn this off to improve performance if you don't need it.
10     PrimaryActorTick.bCanEverTick = true;
11     // Partie III: construction de l'objet VisualMesh
12     MonMaillage = CreateDefaultSubobject<UStaticMeshComponent>(TEXT("Mesh"));
13     MonMaillage->SetupAttachment(RootComponent);
14     ConstructorHelpers::FObjectFinder<UStaticMesh> CubeVisualAsset(TEXT("/Game/StarterContent/Shapes/Shape_Cube.Shape_Cube"));
15     if (CubeVisualAsset.Succeeded()) {
16         MonMaillage->SetStaticMesh(CubeVisualAsset.Object);
17         MonMaillage->SetRelativeLocation(FVector(0.0f, 0.0f, 0.0f));
18     }
19 }
20
21 //Partie III: fin de la construction de l'objet Visual Mesh
```

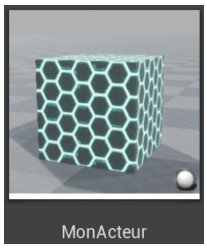
(a)



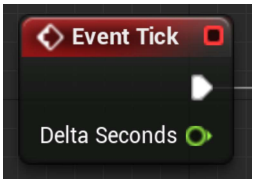
(b)

Recompilez votre projet et puis supprimez et rajoutez l'acteur dans la scène et vous devriez voir désormais le maillage rattaché à l'objet comme sur l'image (b) ci-dessus.

M2 3.3 Toujours dans le constructeur, après le réglage de la position, changez le matériel de votre cube. Pour cela vous utiliserez toujours la classe `FObjectFinder` mais avec le paramètre template `UMaterial` et en utilisant la fonction `SetMaterial` qui présente presque la même syntaxe que la commande `SetStaticMesh` utilisé dans l'exemple. Après compilation de votre projet, votre acteur devrait avoir l'apparence différente comme sur l'image suivante :



3.4 En observant le code de la classe, vous devriez voir l'équivalent la méthode permettant de mettre à jour l'objet à chaque rafraîchissement du jeu :



En rajoutant des instructions, vérifiez que cette fonction est bien comparable au bloque ci-dessous.

3.5 Dans la fonction précédente, rajoutez le code ci-dessous :

```
42    FVector NewLocation = GetActorLocation();
43    float RunningTime = GetGameTimeSinceCreation();
44    float DeltaHeight = (FMath::Sin(RunningTime + DeltaTime) - FMath::Sin(RunningTime));
45    NewLocation.Z += DeltaHeight * 100.0f;       //Scale our height by a factor of 100
46    SetActorLocation(NewLocation);
```

Testez le code et rajoutez un déplacement selon les axes X et Y de façon à avoir des déplacements aléatoires de faible amplitude (environ 15). Vous pourrez utilisez la fonction `random()` avec l'opérateur modulo (%) (par ex rand()%10 renvoi un entier aléatoire entre 0 et 10).

3.6 En vous inspirant du code de la question précédente, ajoutez une rotation à l'ojet de façon à ce qu'il effectue à la fois des mouvements verticaux mais aussi des rotations. **Indication :** pour stocker une rotation vous devez utiliser un objet de type `FRotator` qui possède un attribut `Yaw` qui défini la composante de la rotation autour de l'axe Z.

On souhaite allez un pue plus loin en rajoutant des effet de particules à votre acteur.

M2 3.7 Dans le fichier `MonActeur.h`, rajoutez les deux inclusions des fichiers entêtes suivants :

```
7    #include "Particles/ParticleSystem.h"
8    #include "Particles/ParticleSystemComponent.h"
9
10   #include "MonActeur.generated.h"
```

Rajoutez ensuite une nouvelle déclaration d'un attribut `MonEffetParticules` de type "`UParticleSystemComponent *` " (après `VisualMesh` toujours dans le fichier .h).

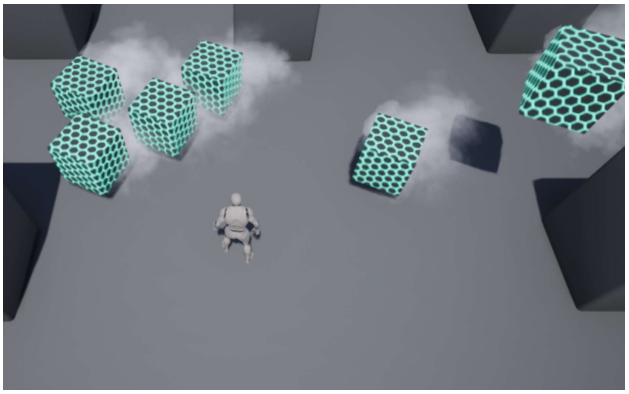
M2 3.8 Dans le fichier `MonActeur.cpp`, de la même façon que vous avez construit l'objet `VisualMesh` (dans le constructeur), créez l'objet `MonEffetParticules` en utilisant le paramètre template `UParticleSystemComponent` et le texte `MovementParticles`.

M2 3.9 Rattachez ensuite l'effet particules au maillage du cube en utilisant la méthode `SetupAttachment` sur `MonEffetParticules`.

M2 3.10 Enfin, il ne reste plus qu'à récupérer l'asset de l'effet de fumée. Pour cela, vous pouvez vous inspirer des lignes 15-20 de l'exemple de code de la question 3.2. Vous aurez à changer le type du template (`UParticleSystem`) et le chemin de l'asset `/Game/StarterContent/Particles/P_Steam_Lit`) et rattacher l'élément avec :

```
30        MonEffetParticules->SetTemplate(ParticleAsset.Object);
```

Si tout fonctionne correctement, vous devriez obtenir la visualisation suivante :



Partie IV: Intégration dans l'interface *Unreal Engine*

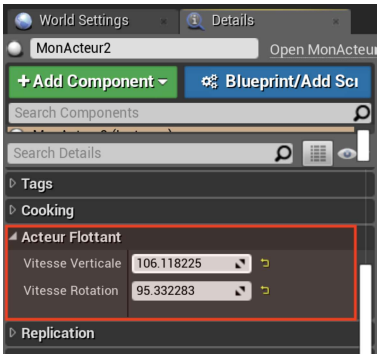
Dans cette partie, nous allons voir comment il est possible de rendre accéssible des éléments d'une classe dans l'interface d'*Unreal Engine*.

4.1 Ajoutez une (resp. deux) variable(s) de classe `VitesseDeplacement` (et `VitesseRotation` si vous avez géré la rotation) qui remplacera la valeur numérique que vous avez utilisée dans la fonction `Tick`.

4.2 Pour pouvoir voir une variable dans l'interface d'*Unreal Engine*, il suffit de rajouter une instruction `UPROPERTY()` qui permet configurer différents paramètres comme par exemple la catégorie dans laquelle doit se trouver le paramètre. Par exemple les instructions suivantes permettent de faire afficher les paramètres dans l'interface dans une nouvelle catégorie *ActeurFlottant*.

```
25      UPROPERTY(EditAnywhere, BlueprintReadWrite, Category="ActeurFlottant")
```

Testez cette instruction sur les deux attributs liés aux mouvements de votre acteur et vérifiez qu'ils sont bien fonctionnels dans l'interface et une fois le jeu lancé. Si tout fonctionne correctement vous devriez pouvoir régler les attributs dans l'interface :

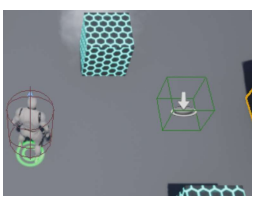


DU 4.3 Rajoutez un attribut de classe, nomé `estStatique`, de type `bool` initialisé à `false` qui permettra de désactiver tout mouvement de l'objet. Cet attribut devra être visible dans l'interface d'unreal.

M2 4.4 Rajoutez un attribut de classe, nomé `effetFume` de type `bool` initialisé à `false` et modifiez votre code de façon à prendre en compte cette valeur pour activer ou désactiver l'effet de fumée. L'effet peut être désactivé ou activé en utilisant respectivement les fonctions `Deactivate()` ou `Activate()`.

Après avoir vu la liaison des attributs de la classe, il reste à mettre en évidence l'utilisation de la classe C++ directement en *Blueprint*. Pour cela nous allons effectuer des manipulations à l'intérieur du Level BluePrint.

4.5 Dans votre scène rajoutez un trigger de type `BoxTrigger` et faire en sorte qu'il soit visible par le joueur de façon à faciliter l'activation.



4.6 Dans le LevelBluePrint, associez une action lors de l'entrée du joueur sur *trigger* de façon a rendre statique tous les éléments de type `MonActeur` dans la scène. **Indication :** comme vu dans les TP précédants, vous pouvez utiliser la fonction GetAllActorsOfClass qui permet de récupérer tous les acteurs d'une classe.

4.7 Inversement, déactivez tous les éléments de la scène lorsque le joueur quitte la zone d'activation.

Partie V: Combinaisons de C++ et *Blueprint*

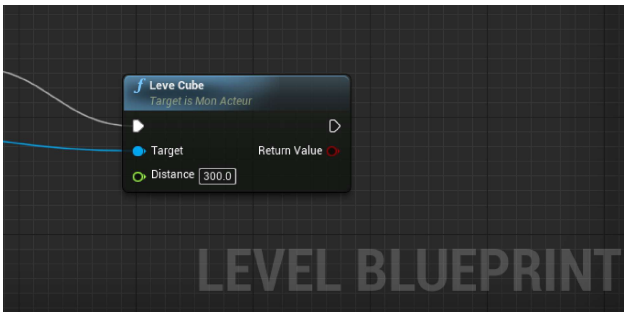
Dans cette partie, nous allons illustrer tout d'abord l'utilisation d'une fonction définie en C++ mais que l'on va utiliser en *Blueprint*. Pour illustrer cette possibilité, nous allons créer une nouvelle fonction en C++ qui va permettre de déplacer le cube vers le haut.

5.1 Dans le fichier `MonActeur.h`, déclarez une nouvelle fonction `bool leveCube()` (dans la section *public*). Pour qu'elle puisse être utilisée en *Blueprint*, il faut rajouter la macro suivante juste avant la daclaration :

```
35    UFUNCTION( BlueprintCallable,  Category = "Character" )
```

5.2 Dans le fichier `MonActeur.cpp`, implémentez votre fonction de façon à déplacer votre objet de la distance donnée en paramètre. La fonction renverra vraie si l'objet est au dessus du sol. Vérifiez que votre programme compille bien.

5.3 Dans le *Level Blueprint* à la suite des événements liés au trigger vérifiez que vous avez bien accès à votre fonction C++ comme sur l'image suivante :



5.4 Testez que votre fonction est bien fonctionnelle et rajoutez des instructions *Blueprint* de façon à exploiter la valeur de retour de votre en fonction. Vous pourrez par exemple détecter quand l'acteur à une position inférieure à 0 et dans ce cas là changer modifiez sa hauteur à 0.

5.5 Pour ceux qui ont fini, testez de créez un objet *Blueprint* à partir de la classe précédente.