MCR - Structure du programme

Unreal Engine

Nous utilisons Unreal Engine comme fondation pour notre programme. Cela signifie que la quasi-totalité de nos choix d'implémentation prennent en compte les restrictions et opportunités qu'impose Unreal Engine. Un exemple de ceci est que la norme dans Unreal Engine est d'utiliser le came case pour le nommage des variables.

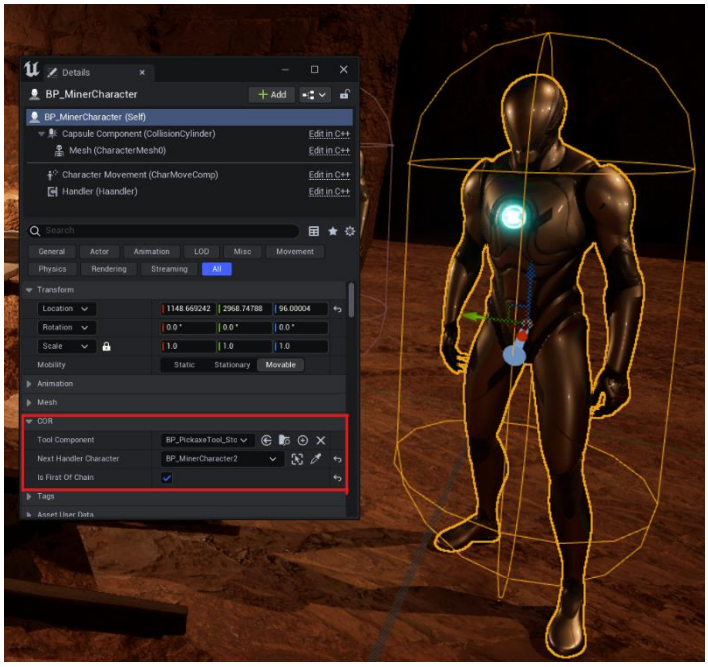
Blueprints

Les Blueprints sont un système offert par Unreal. Comme leur nom indique, ils servent à créer des plans d'objets se basant sur des classes c++. Ils sont donc une couche entre les objets et leurs classes. De plus, ils permettent d'accueillir une logique supplémentaire sous forme de programmation visuelle à base de nœuds. L'avantage de ce langage, outre sa simplicité, est qu'il permet l'accès à des fonctions de Unreal Engine très complexe à faire en c++ mais de manière très simple. Parmi ces fonctions, on peut trouver par exemple la génération d'un nouvel acteur dans la scène Unreal à l'exécution, ou bien encore des fonctions asynchrones comme demander à un acteur de se déplacer vers un endroit.

UPROPERTY et UFUNCTION

Pour définir ce que les Blueprints ont le droit et doivent faire depuis les classes c++, il faut utiliser les macros UPROPERTY et UFUNCTION.

UPROPERTY indique les droits que la version Blueprint de la classe et que l'éditeur a sur les membres de cette classe. Par exemple, `EditAnywhere` signifie que la valeur du membre peut être édité à tout moment, même instance par instance dans la scène Unreal Engine.

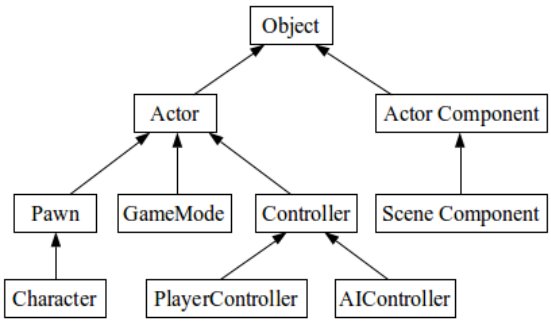


On peut voir sur l'image que les trois membres COR (Chain Of Responsability) de l'instance du Blueprint de MinerCharacter sont modifiables directement dans l'éditeur de la scène.

Les mots clés `BlueprintReadWrite` et `BlueprintReadOnly` définissent si le membre est accessible en écriture ou en lecture depuis les scripts des Blueprints.

UFUNCTION indique quel doit être le comportement du Blueprint par rapport à une fonction. `BlueprintCallable` signifie que la fonction peut aussi être appelée depuis le script d'un Blueprint. `BlueprintImplementableEvent` signifie que la fonction n'est pas implémentée en c++ mais seulement sous forme d'un `Event` dans les Blueprints. Un `Event` est une fonction asynchrone sans valeur de retour est peux ne pas être implémentée dans certains Blueprints. `BlueprintNativeEvent` se comporte exactement comme une fonction normale mais est constituée de deux parties. La partie implémentée dans le Blueprint et la partie *<NomDeLaFonction>_Implementation* écrite en c++ qui est utilisé uniquement si Unreal Engine ne trouve pas la fonction implémentée en Blueprint.

Hiérarchie des classes Unreal Engine



L'utilisation d'un préfix est de norme pour ces classes. Tout Actor doit être préfixé de la lettre A, Actor compris (un Actor est donc un AActor). Et il en va de même pour la lettre U avec les Object (UObject, UActorComponent, …).

Structure hors chaine de responsabilité

ControllableCharacter et MinerCharacter

Les Characters sont les acteurs (AActor) qui se trouverons dans le Level Unreal et qui ont la possibilité de se déplacer. Ils nous servent de véhicule pour nos Handler.

Nous avons décidé que la classe MinerCharacter hérite de la classe abstraite ControllableCharacter dans le but de faciliter l'ajout de plusieurs types de Characters par le futur.

Un MinerCharacter dispose de deux composants (outre ceux qui leurs sont déjà attachés par défaut par Unreal Engine) que nous lui attachons à sa création et au début de la partie. Le premier qui peut être créé statiquement est le MinerHandler. C'est le UObject qui nous servira de maillon dans notre chaine de responsabilité. Il peut être créé statiquement (et donc en c++) car nous savons qu'un MinerCharacter aura toujours un MinerHandler.

Dans le cas du deuxième composant (qui n'est pas réellement un UActorComponent mais seulement un AActor gardé en mémoire par le biais d'un pointeur), la pioche qu'il transporte, nous utilisons un pointer sur une UClass (TSubClassOf<>) qui sera définie par l'utilisateur lors de la création de ses MinerCharacter. Dans ce cas-là, la solution la plus propre est de passer par un script stocké dans le Blueprint pour initialiser cet AActor. Nous avons donc créé la fonction que nous avons ensuite implémentée dans le Blueprint :

```
UFUNCTION(BlueprintImplementableEvent)
void UnEquipTool();
```

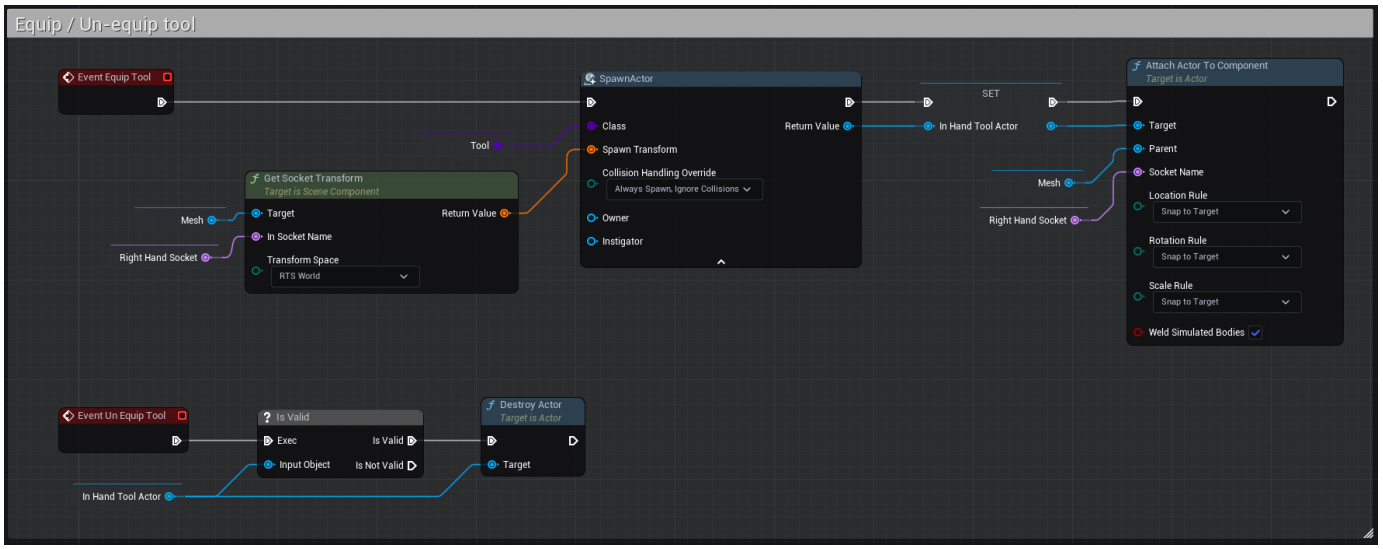


Figure 1 Fonctions EquipTool() et UnEquipTool()

La logique de déplacement, par soucis de simplicité à nouveau, a aussi été codée dans le Blueprint puisque ce langage est fait pour gérer ce genre d'événements étroitement liés à Unreal Engine. Ci-dessous la fonction de déplacement prenant une requête contenant le Block devant être miné en paramètre.

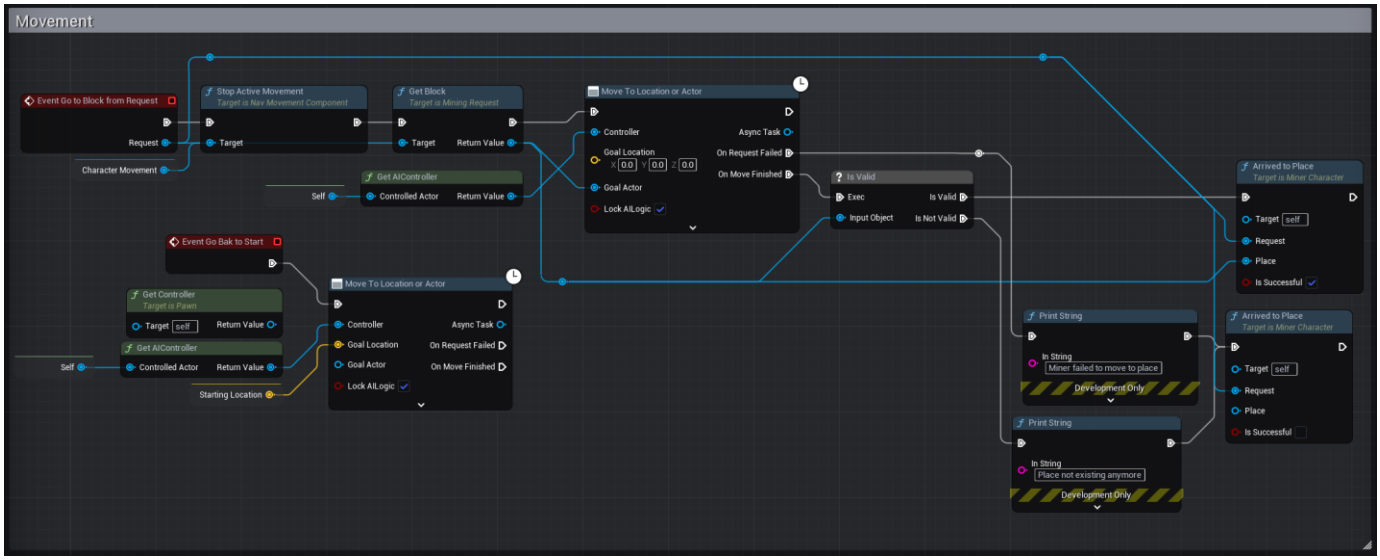


Figure 2 Fonctions GoToBlockFromRequest() et GoBackToStart()

Finalement, Ils possèdent un pointeur vers un MinerCharacter représentant le prochain mineur dans la chaine de responsabilité. Le Handler de ce mineur pointé est envoyé au Handler attaché par le biais de la fonction SetNext().

Tools

Les Tools sont les outils que nous pourrons donner à nos Characters.

À nouveau, la classe PickaxeTool hérite de la classe abstraite Tool dans le but de faciliter l'ajout de plusieurs types de Tool dans le futur.

La classe PickaxeTool est très simpliste puisqu'elle ne contient qu'une valeur définissant la qualité de la pioche. Cette classe sert surtout en tant que parente à plusieurs Blueprints définissant des pioches avec différentes qualités.

VisitablePlace et Block

La classe StoneBlock hérite de la classe abstraite Block qui hérite elle-même de la classe abstraite VisitablePlace, vous l'aurez compris, par soucis de simplicité pour de futurs sous-classes.

La seule logique contenue dans ces classes est leur destruction. Si un Block peut être miné par une pioche, alors le Block notifie le MineGameMode de recréer une nouvelle requête et se détruit.

Structure de la chaine de responsabilité

AbstractMinerHandler et MinerHandler

La classe MinerHandler hérite de AbstractMinerHandler. Leur but est de gérer une requête qu'ils reçoivent par le biais de leur fonction Handle().

Comme expliqué auparavant, ils sont attachés en tant que composant à un MinerCharacter et possèdent donc un pointeur sur leur MinerCharacter grâce à la fonction GetOwner(). Cela leur permet donc de contrôler ce dernier pour gérer la Request qui leur a été fournise.

Un problème qui est vite survenu par rapport aux plans originaux du modèle de conception est le fait que le déplacement d'un ACharacter se fait en appelant une fonction asynchrone. Cela signifie que comme les mineurs doivent se déplacer pour miner les blocs, le traitement d'une requête ne peut pas se faire intégralement dans la fonction Handle() puisque cette fonction ne peut pas attendre que le mineur aie finit son déplacement. Nous avons donc introduit les fonctions ArrivedToPlace() de la classe ControllableCharacter et ForwardRequest() de la classe AbstractMinerHandler qui servent comme fonctions de callback une fois que le déplacement du mineur est terminé.

La fonction ForwardRequest() a donc pour but de passer la Request plus loin dans la chaine de responsabilité.

Request et MiningRequest

La classe MiningRequest hérite de Request. Elles ont pour but d'être les objets qui sont passés dans la chaine de commandement.

MiningRequest a comme attribut le Block devant être miné.

MineGameMode

Les game modes ou AGameModeBase sont l'endroit où il est possible d'écrire toute la logique du jeu. On peut les considérer comme les programmes Main d'une application traditionnelle.

Dans notre cas, MineGameMode se charge d'envoyer les premières requêtes. Pour cela, il scanne tous les mineurs et tous les blocs de la scène dans la fonction FindAllActors() et les stocks dans des listes de pointeurs vers objets. Il utilise ensuite ces listes pour trouver le mineur que nous avons définit en tant que premier de la chaine grâce à l'attribut bIsFirstOfChain. Une fois ce mineur trouvé, il cherche le bloc le plus proche de ce mineur et construit une requête contenant ce bloc et la transmet au Handler de ce mineur. Comme expliqué ci-dessus, c'est lorsqu'un bloc est détruit qu'il indique au game mode de refaire une nouvelle requête.