# Lab 6 – Managing Actors, Replication Properties – Part 1

*COMP280-Multiplayer Game Programming*

**Purpose:** Managing Actors (continued), Replication Properties

## Contents



# 1. Intro

We will:

- Manage Actors in a Multiplayer Environment
  - ~~Set up the character~~ DONE
  - ~~Add interactions (input actions, input mapping context)~~ DONE
  - Control the connection to an actor
  - Manage the Actor Relevancy and Authority
  - Add character stats in a stats data table
- Replicate Properties Over the Network
  - Enable property replication
  - Handle the level-up of the character
  - Add a HUD to the game

# 2. Managing Actors in a Multiplayer Environment

## 2.1. Technical requirements

- We are going to use Unreal Engine 5.4.

- You have to have finished the Lab4.

We'll tackle Actor's connection management and attributes that are relevant during a game session. The player character is now just an empty shell. Let's enrich it with adding/testing:

- ~~more **components**~~: DONE`
- ~~player **input** logic.~~ DONE
- **ownership** in a multiplayer environment
- **relevancy** in the level.

In this lab, the following is covered:

- ~~**Setting up** the character~~ DONE
- Controlling the **connection** of an Actor
- Understanding Actor **relevancy**
- Introducing **authority**

## 2.2. ~~Setting up the character~~ DONE

## 2.3. Controlling the connection of an Actor

Note that we are already playing a networked game even though you did not add any multiplayer code logic. **Why?** The **Character** class is already set to be replicated — just open the **BP_Character** Blueprint and look for the **Replication** category. You will find out that Replicate Movement has been set by default and also that the Replicates property is set to true.

Note that:

- the character on the server window moves and sprints **smoothly**
- the one on the client's window, moves a bit **jumpy** when running.

This happens because we are trying to execute the **sprint** action on the **client**, but the server is the one who is actually in command. So, the client will make the character move faster, but the server will bring it back to its move position. Basically, at the moment, we are trying to "cheat" on the client, but the server, which is **authoritative**, will forbid you from doing this. To deal with this we need to know more about **replication** and **authority** settings in the next Lab.

In Unreal Multiplayer games, each **connection** has its own **PlayerController** that has been created expressly for it; in this case, we say that the PlayerController is **"owned"** by that connection. In Unreal Engine, Actors can have an **Owner**: if the outermost Owner of an Actor is a **PlayerController**, then the PlayerController becomes the Owner of that Actor. This means that the first Actor is also **owned** by the same connection that owns the PlayerController. So ownership chain is:

**Actor → PlayerController → Connection**

The concept of **ownership** is used during Actor replication to determine which connections receive updates for each Actor: for instance, an Actor may be flagged so that only the connection that owns that Actor will be sent property updates for it.

As an example, let's imagine that your thief character (which is basically an Actor) is possessed by a PlayerController – this PlayerController will be the Owner of the character. During gameplay, the thief gets a pickup that grants a magical dagger: once equipped, this weapon will be owned by the Character. This means that the PlayerController will also own the dagger. In the end, both the thief Actor and the dagger will be owned by the PlayerController connection. As soon as the thief Actor is no longer possessed by the Player Controller, it will cease to be owned by the connection, and so will the weapon.

In standalone games, we retrieve the **Player Controller** or the **character** by:

- Blueprint - using nodes such as:
  - Get Player Controller, or
  - Get Player Character
- C++:
  - UGameplayStatics::GetPlayerController(), or
  - UGameplayStatics::GetPlayerCharacter().

In a networked environment we may have issues if we don't know what we are doing, as we'll get different results depending on the context.

Example: calling the **Get Player Controller** function with **Player Index** equal to **0** will results in:

- The **Listen server's PlayerController** if you are calling it from a **listen server**
- The **First client's PlayerController** if you are calling it from a **dedicated server**
- The **client's PlayerController** if you are calling it from a **client**

Furthermore, the index is not consistent across the server and different clients, so more confusion can arise.

Solution: In Multiplayer games in Unreal Engine, we need to use some of the following functions (or their corresponding nodes):

- **AActor::GetOwner()**, which returns the **Owner** of an Actor instance
- **APawn::GetController()**, which returns the **Controller** for the Pawn or Character instance
- **AController::GetPawn()**, which returns the **Pawn** possessed by the Controller
- **APlayerState::GetPlayerController()**, which will return the **Player Controller** that created the Player State instance (remote clients will return a null value)

Components, on the other hand, have their own way of determining their **owning connection**: – They start by following the component's outer chain until they find the Actor that owns them. - Then, the system determines the owning connection of that Actor via **UActorComponent::GetOwner()**.

In this section, we have just "scratched the surface" of what an **Owner** is and how to get info about it, but you should be aware that **connection ownership** is so important that it will be pervasive throughout the rest of the course: in other words, the idea of owning a connection is deemed crucial enough to be addressed throughout the multiplayer project we are developing.

In the next section, we're going to deal with a strongly connected topic: **relevancy**.

## 2.4. Understanding Actor relevancy

Definition: **Relevancy** is the process of determining which objects in a scene should be visible or updated based on their importance to the player.

This is an important concept in Unreal Engine, and by understanding how it works, you can make sure your game runs efficiently. In this section, we will explore this topic and show an example of how it works depending on its settings.

### 2.4.1 Understanding relevancy

**Relevancy** refers to how the Engine determines **which Actors** in the game world **should be replicated** to which clients, based on their current locations, and which Actors **are relevant** to the player's current **view** or **area**.

A game level can have a size varying from **very small** to **really huge**. This may pose a problem in updating everything on the network and for every client connected to the server. As the playing character may not need to know every single thing that's happening in the level, most of the time, it's just enough to let it know what is near.

So, the Engine uses several factors to let the player know if something has changed on an Actor:

- the distance to the Actor itself,
- its visibility,
- is the Actor currently active?

An Actor that is deemed **irrelevant** will NOT be replicated to the player's client, and this will::

- reduce network traffic and
- improve game performance.

Unreal uses a virtual function named **AActor::IsNetRelevantFor()** to test the relevancy of an Actor. It goes throw some checks/tests that can be summarized as follows:

- **First** Check: The Actor is relevant if the following applies:

    - Its **bAlwaysRelevant** flag is set to true
    - Or, it is **owned** by the Pawn or PlayerController
    - Or, it is the **Pawn** object
    - Or, the Pawn object is the **instigator** of an action such as *noise* or *damage*
- **Second** Check: If the Actor's **bNetUseOwnerRelevancy** property is **true** and the Actor itself has an **Owner**, the **owner's relevancy** will be used.
- **Third** Check: If the Actor has the **bOnlyRelevantToOwner** property set to true and does not pass the first check, then it is not relevant.
- **Fourth** Check: If the Actor is **attached** to another Actor's skeleton, then its relevancy is determined by the relevancy of its **parent**.
- **Fifth** Check: If the Actor's **bHidden** property is set to true and the root component is not colliding with the checking Actor, then the Actor is **not relevant**.
- **Sixth** Check: if **AGameNetworkManager** is set to use **distance-based relevancy**, the Actor is **relevant** if it is **closer** than the **net cull distance**.

> The **Pawn/Character** and **PlayerController** classes have slightly **different relevancy checks** as they need to consider additional information, such as the **movement** component.

This system is not perfect:

- as the distance check may give a false negative when dealing with large Actors.
- the system does not take into account sound occlusion
- the system does not take into account other complexities related to ambient sounds.

Nevertheless, the approximation is precise enough to get good results during gameplay. Let's begin implementing a concrete example to see relevancy in action by testing your character.

## 2.4.2 Testing relevancy

To test the effect of **relevancy** during gameplay, you'll create a simple **pickup** and play around with its settings.

### 2.4.2.1 Creating the Pickup Actor

- Creating a new **C++ class** inheriting from **AActor**; name it **US_BasePickup**.
- Open the generated header file and add these two component declarations in the private section:

```
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category="Components", meta=(AllowPrivateAccess = "true"))
TObjectPtr<class USphereComponent> SphereCollision;
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category="Components", meta=(AllowPrivateAccess = "true"))
TObjectPtr<class UStaticMeshComponent< Mesh;
```

We just declared the:

- **SphereCollision** component for triggering the pickup, and
- **Mesh** component for its visual aspect.

- Next, in the protected section, just after the BeginPlay() declaration, add a declaration that will handle the **character overlap** with the Actor:

```
UFUNCTION()
void OnBeginOverlap(UPrimitiveComponent* OverlappedComponent, AActor* OtherActor
  , UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromSweep, const FHitResult& SweepResult);
```

- Immediately after that, add the declaration for the **pickup action**:

```
UFUNCTION(BlueprintCallable, BlueprintNativeEvent, Category = "Pickup", meta=(DisplayName="Pickup"))
void Pickup(class AUS_Character* OwningCharacter);
```

We need this function to be callable inside a Blueprint, so we use the **BlueprintCallable** specifier. Then, the **BlueprintNativeEvent** specifier states that the function can be overwritten by a Blueprint, but it also has a default native C++ implementation that will be called if the Blueprint does not implement anything. To natively implement the method, in the **US_BasePickup.cpp** file, we will need to implement a C++ function with the same name as the primary function but with **_Implementation** added to the end. Finally, to the

public section – and after the corresponding properties, in order to avoid forward declarations – add two **getters** for the components we declared previously:

```
FORCEINLINE USphereComponent* GetSphereCollision() const { return SphereCollision; }
FORCEINLINE UStaticMeshComponent* GetMesh() const { return Mesh; }
```

- Open the **US_BasePickup.cpp** file and add the necessary includes below *US_BasePickup.h*:

```
#include "US_Character.h"
#include "Components/SphereComponent.h"
```

- Then, inside the constructor, add the following block of code, which creates the two components and attaches them to the Actor:

```
SphereCollision = CreateDefaultSubobject<uspherecomponent>("Collision");
RootComponent = SphereCollision;
SphereCollision->SetGenerateOverlapEvents(true);
SphereCollision->SetSphereRadius(200.0f);
Mesh = CreateDefaultSubobject<ustaticmeshcomponent>("Mesh");
Mesh->SetupAttachment(SphereCollision);
Mesh->SetCollisionEnabled(ECollisionEnabled::NoCollision);
```

- Immediately after that, set **bReplicates** to true (as Actors do not replicate by default):

```
bReplicates = true; //Actors have this false by default
```

- Inside the BeginPlay() function, add a *dynamic multi-cast delegate* for the overlap event:

```
SphereCollision->OnComponentBeginOverlap.AddDynamic(this, &AUS_BasePickup::OnBeginOverlap);
```

> We will give proper attention and focus to replication, later.

- Now add the **overlap handler** just after the closing bracket of the BeginPlay() function:

```
void AUS_BasePickup::OnBeginOverlap(UPrimitiveComponent* OverlappedComponent, AActor* OtherActor,
    UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromSweep, const FHitResult& SweepResult)
{
    if (const auto Character = Cast<AUS_Character>(OtherActor))
    {
        Pickup(Character);
    }
}
```

- Notice that the previous block of code is quite straightforward: after having checked that the overlapping Actor is AUS_Character (i.e., our multiplayer hero), we simply call the **Pickup()** method (to be implemented next). As a comparison, this is similar to Unity's **OnCollisionEnter** method.
- Add the **Pickup()** C++ implementation:

```
void AUS_BasePickup::Pickup_Implementation(AUS_Character * OwningCharacter)
{
   SetOwner(OwningCharacter);
}
```

Let's create a blueprint out of this C++ class.

### 2.4.2.2 Creating a pickup Blueprint class

To test **IA_Interact** input action, as well as the effects of **relevancy** in action, we'll create a Blueprint pickup:

- Open the Unreal Engine Editor.
- **Compile** your project to add the pickup to the available classes of your Blueprints.
- In your Blueprints folder, create a new **Blueprint Class** inheriting from **AUS_BasePickup** and name it **BP_SpellBook**.
- Open it and in the Blueprint **Details** panel, select a **mesh** for the **Static Mesh** property – I opted for the **spellBook** model.

To make the book float, we are going to move the mesh up and down by using a Timeline node. To do so, follow these steps:

- Open the **Blueprint Event Graph**, right-click on the canvas, and add a **Timeline node** – give it a name such as **Float**.
- Double-click on the node to **open** the corresponding editor.

- Click the **+ Track** button to add a new **Float** track and name it **Alpha**.
- Click on the **Loop** button to enable the loop mode (becomes blue).
- Right-click on the curve panel and select the **Add key to...** option. Then, set **Time** to **0** and **Value** to **0**.
- Create another **key**, but this time set **Time to 2.5** and **Value to 0.5**.
- Create one **last key**, this time with **Time equal to 5** and **Value equal to 0**.
- Right-click on **each** of the keys and set the **Key Interpolation value** to **Auto**.

You have just created a **sinusoidal value** that will indefinitely **loop** between 0 and 1 values; you'll use this floating value to move the book up and down. To implement this floating movement, return to the Event Graph and do the following:

- Connect the **Event Begin Play** node to the **Timeline** node.
- Drag the **Mesh** component from the Components panel onto the **Event Graph canvas**. Click and drag from its **outgoing pin** to add a Set Relative Location node.
- Connect **Timeline** → **Update** outgoing pin to the **Set Relative Location** incoming execution pin.
- Drag the **Timeline** → **Alpha** pin to a **Multiply** node — $\alpha$ goes to first parameter; set the second parameter to **100** (so the **Multiply** node calculates $\alpha \cdot 100$).
- Right-click on the **New Location** pin of the **Set Relative Location** node and select **Split Struct Pin** to expose the $X$, $Y$, and $Z$ values.
- Connect the **Result** pin of the **Multiply** node to **New Location Z** of the **Set Relative Location** node (so, make $Z$ of the **BP_SpellBook** move up and down).

> This floating animation is purely a **visual effect**, so we just won't worry about whether it is synchronized over the network.

Now that the Blueprint item has been created, it's time to add it to the level and test its pickup functionality – something we are going to do in the next subsection.

### 2.4.3 Testing the relevancy settings

It's now time to test how the spell book behaves in a multiplayer environment when relevancy settings are changed. First of all, drag an instance of the BP_SpellBook Blueprint into the level, near the PlayerStart Actor, so that the player will be in the line of sight once it has been spawned. Open the PB_SpeelBook Blueprint and, with the Class Defaults panel selected, look for the Replication category. Try playing the game as a listen server with three players, and every player should see the book as expected. Things are going to get a bit trickier in a moment... Stop the application from playing and get back to the BP_SpellBook Blueprint. Look for the Net Load on Client property and uncheck it. As this property will load the Actor during map loading, we need to disable it, so the Actor will be loaded only when it becomes relevant for the client. You are now ready to test different situations, depending on the properties you change in the next steps.

#### 2.4.3.1 Setting the net cull distance

The first situation you will be testing is about distance culling – at the moment, your object is set to be relevant at a very far distance. To check this, run the game again, and you should see no difference since your last gameplay. But what happens if you lower Net Cull Distance Squared to a very low number, for instance, 500? You will get really "weird" behavior: the server window will show the book, while the two clients will not! With one of the client windows active, try walking near the zone where the book should be, and it will immediately pop up! Didn't I already warn you that this book was nothing short of magical?

# 3. Zip Project and Submit

- Click **File** → **Zip Project**
- Keep the default folder, select the name **US_LTOL_{YourInitials}_Lab5.zip**
- When the zipping finishes click on the provided link **Show in Explorer**
- Upload the zip in the corresponding folder (Lab5).

# 4. Summary

In this Lab, we:

- Managed Actors in a Multiplayer Environment
    - Set up the character
    - Added interactions (input actions, input mapping context)
    - Controlled the connection to an actor

Next week, we will:

- Continue to Manage Actors in a Multiplayer Environment
    - Manage the Actor Relevancy and Authority
    - Add character stats in a stats data table
- Deal with Replication and Authority