Lab 5 - Managing Actors, Replication Properties

COMP280-Multiplayer Game Programming

Purpose: Managing Actors

Contents

1. Intro

We will:

- Manage Actors in a Multiplayer Environment
  - Set up the character
  - Add interactions (input actions, input mapping context)
  - Controll the connection to an actor
  - Manage the Actor Relevancy and Authority
  - Add character stats in a stats data table

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**:
  - **Camera**

- player **input** logic.
- **ownership** in a multiplayer environment
- **relevancy** in the level.

In this lab, the following is covered:

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

2.2. Setting up the character

Before dealing with topics such as connections, authority, and roles, we need to set up the player character. So, in this section, you will:

- add a camera
- add user input handling
    - set up movement

2.2.1 Adding basic settings to the character

We'll add the components that will make up the third-person camera behavior and implement their logic. After that, we'll set some default values for the components that are already available in the Character class: the Arrow, the Capsule, and the SkeletalMesh components.

2.2.1.1 Adding a Camera component to the character

We will add a Camera component and a Spring component that will connect the camera to the Capsule component available in the Character class and leter, getters. To do this:

- Open the **project** from Lab 4 (for me is US_LOTL_AT).
- Open the **US_Character.h.**
- Add these two component declarations after the **GENERATED_BODY()** macro:

```
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = Camera, meta = (AllowPrivateAccess = "true"))
TObjectPtr<class USpringArmComponent> CameraBoom;
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = Camera, meta = (AllowPrivateAccess = "true"))
TObjectPtr<class UCameraComponent> FollowCamera;
```

- *Take Snapshot*

The above code uses the `UPROPERTY` annotation with a few *specifiers*. The meaning of them is:

Name	Type	Description
UPROPERTY()	Annotation	Property of the class
VisibleAnywhere	Attribute	Property visible in all UE windows, but not editable
BlueprintReadOnly	Attribute	Property can be read by Blueprints but not modified
Category	Attribute	The category of the property when displayed in the Blueprint Details panel
meta	meta	Metadata Specifiers exists only in the Editor; control interactions of the property with Unreal Engine
AllowPrivateAccess	meta Attr.	A private member should be accessible from a Blueprint

We need the last one, as, these properties' **accessibility** is not explicitly declared, and so, they default to **private**.

An exhaustive list of **Property** Specifiers and **Metadata Specifiers** can be found here: **UPROPERTY Specifiers**

class keyword before the type (`class USpringArmComponent`) is a C++ class forward declaration — a way to declare the class name and its members without providing the full class definition. This can be useful in situations where you want to use a class in a header file but do not want to include the entire class definition, which can make compilation slower and create unnecessary dependencies.

`TObjectPtr<T>` template – is a new addition in UE5, introduced to replace raw pointers (for example, **USpringComponent***) in header files with UProperties. The `TObjectPtr<T>` template is intended only for member properties declared in the headers of your code. Elsewhere (functions etc) you can use raw pointers.

Camera and Spring components are private, so we need to add **getter** methods. Inside the public declaration of the header, locate this line of code:

```
virtual void SetupPlayerInputComponent(UInputComponent* PlayerInputComponent) override;
```

- Then, below this line, add the following:

```
// Getters
FORCEINLINE USpringArmComponent* GetCameraBoom() const { return CameraBoom; }
FORCEINLINE UCameraComponent* GetFollowCamera() const { return FollowCamera; }
```

- *Take Snapshot*

These two methods will let you access the pointer components.

FORCEINLINE macro forces the code to be inlined; this is going to give your code some performance benefits as you will avoid a function call when using this method. You can see this macro definition (and any macro for that matter) as follows: right-click over `FORCEINLINE` and click on `Peek Definition`, or press `Alt+F12`; you'll see this line (that comes from **WindowsPlatform.h**):

```
#define FORCEINLINE __forceinline /* Force code to be inline */
```

2.2.1.2 Implementing the camera behavior

- Let's add the code logic to handle these properties. Open the .cpp file and add the following includes at its top:

```
#include "Camera/CameraComponent.h"
#include "Components/CapsuleComponent.h"
#include "GameFramework/CharacterMovementComponent.h"
#include "GameFramework/SpringArmComponent.h"
```

- *Take Snapshot*
- Then, inside the constructor (i.e., AUS_Character::AUS_Character()), add this code:

```
// CameraBoom
CameraBoom = CreateDefaultSubobject<USpringArmComponent>(TEXT("CameraBoom"));
CameraBoom->SetupAttachment(RootComponent);
CameraBoom->TargetArmLength = 800.0f;
CameraBoom->bUsePawnControlRotation = true;
```

- *Take Snapshot*

> **CreateDefaultSubobject()** is a function that is used to create a new **subobject** of a class that will be owned by another object.

> A **Subobject** is essentially a component or a member variable of an object, and the method is typically called within an object's constructor to initialize its subobjects (in this case, the components).

> **Reminder**
>
> Measurements in Unreal are in **centimeters**, so **800** centimeters in Unreal, would be **8 meters** in Unity.

> **SetupAttachment()** method re-parents a component to another one. In this case, you are attaching the **Camera** component to **RootComponent**, which is actually the **Capsule** component.

- Continuing, add this code block just after the previous lines of code:

```
// FollowCamera
FollowCamera = CreateDefaultSubobject<UCameraComponent>(TEXT("FollowCamera"));
FollowCamera->SetupAttachment(CameraBoom, USpringArmComponent::SocketName);
FollowCamera->bUsePawnControlRotation = false;
```

- *Take Snapshot*

> The difference here is that you are reparenting the camera to the Spring component instead of the root one.

We have created some sort of **"chain of command"** where the camera is connected to the **Spring** component that is connected to the **root** of the Actor – this will let the camera follow the character with a **"springy" behavior** whenever the camera hits an obstacle and provide a better feel for the player.

2.2.1.3 Setting up the default component properties

Let's modify some properties to create a default setup for your Character class:

- In the constructor, add these lines of code:

```
bUseControllerRotationPitch = false;
bUseControllerRotationYaw = false;
bUseControllerRotationRoll = false;
GetCapsuleComponent()->InitCapsuleSize(60.f, 96.0f);
GetMesh()->SetRelativeLocation(FVector(0.f, 0.f, -91.f));
static ConstructorHelpers::FObjectFinder<uskeletalmesh>
       SkeletalMeshAsset(TEXT("/Game/KayKit/Characters/rogue"));
if (SkeletalMeshAsset.Succeeded())
{
   GetMesh()->SetSkeletalMesh(SkeletalMeshAsset.Object);
}
GetCharacterMovement()->bOrientRotationToMovement = true;
GetCharacterMovement()->RotationRate = FRotator(0.0f, 500.0f, 0.0f);
GetCharacterMovement()->MaxWalkSpeed = 500.f;
GetCharacterMovement()->MinAnalogWalkSpeed = 20.f;
GetCharacterMovement()->BrakingDecelerationWalking = 2000.f;
```

- *Take Snapshot*

Here, we just changed some of the default values of the character Actor and its components. Of note is that we got the character model and assigned it to **SkeletalMeshComponent** by means of the **FObjectFinder()** utility method available in the **ConstructorHelpers** class we've used before.

2.2.1.4 Updating the Character Blueprint

- **Save** your files, go back to the Unreal Editor, and
- Click the **Compile** button.
- Open the **BP_Character** Blueprint
- Notice that your changes didn't show up. (Blueprint has not been updated yet).
- Select **File** → **Refresh All** nodes.
- Check that **Camera Boom** and **Follow Camera** elements added to the hierarchy in the **Components** panel
- $Take\ Snapshot$

If we still can't see the updated mesh in the SkeletalMesh component we do the following steps:

- Select the **Mesh** elements in the Components list and look for the **Skeletal Mesh Asset** field in the Details panel.
- If Skeletal Mesh Asset shows a value of None, click on the Reset Property arrow next to it.
- Double-check the Use Controller Rotation Yaw Y property, as it may also need a reset.
- Check that you are able to see the viewport updated with the selected mesh added to it
- $Take\ Snapshot$

Let's make our character it move by adding some user interaction.

2.2.2 Adding interaction to the character

Now we will add **input settings** for the character. To do this, we will be using the new **Enhanced Input System** that has been introduced in UE5. This new system provides developers with more advanced features than the previous one (known simply as the *Input System*), such as:

- complex input handling and
- runtime control remapping
- radial dead zones,
- chorded actions,
- contextual input,
- prioritization,
- extending filtering and processing of raw input data

The old system is being deprecated, (it will probably be removed from Unreal Engine sooner or later), so it is best to stay updated on the changes.

The Enhanced Input System communicates with your code via **Input Actions**.

Input Actions represent what the character can do during play (i.e., walk, jump, or attack). A group of Input Actions can be collected inside an **Input Mapping Context**.

Input Mapping Context represents a set of rules for what will trigger the included **actions**.

At runtime, UE5 will:

- Check a list of **Input Triggers** to determine how user input activated an **Input Action**,
- Validate patterns such as:
  - long presses,
  - release events, or
  - double-clicks.
- Pre-process the raw input through a list of **Input Modifiers** that will alter the data, such as:
  - setting a custom dead zone for the thumbstick or
  - getting a negative value from the input itself.

Let's create the following basic interactions for our character:

Interaction	Control Type	Description
Move/Look	Keyboard/Mouse combo	WASD, Mouse Look
Sprint	Button	
Interaction	Button presses	pickup, use etc.

We'll implement the attack actions in later labs.

> If you want to explore the full range of possibilities offered by this System check: Enhanced Input System

2.2.2.1 Creating Input Actions

To start creating **Input Actions**, take the following steps:

- Right-click on **Content Drawer** → **All** → **Content**
- Click **New Folder** to add a *folder*
- Name it **Input**.
- Inside the folder, right-click and select **Input** → **Input Action** to create an *Input Action* asset.
- Name it **IA_Interact.**
- Create three other Input Actions and name them **IA_Look**, **IA_Move**, and **IA_Sprint**.
- *Take Snapshot*

> ⓘ **IA_** prefix stands for **Input Action**.

Setting Up IA_Interact

We need **IA_Interact** to be activated by a single press of a button (or key), and this action should be dispatched the moment the button has been pressed. To do so, double-click on the asset to open it and do the following:

- Double-Click on **IA_Interact** to set it up
- On the **Action** → **Triggers** click on the **+** button to add a trigger.
- Click the dropdown that should have been created and select **Pressed** – to avoid dispatching multiple events if the player holds the button.
- Leave the rest as it is: – check that **Value Type** has been set to the default value of **Digital (bool)**.
  - Check that **User Settings** → **Player Mappable Key Settings** is at **None** for now.
- *Take Snapshot*

> Note that Triggers' type drop-down box has, among others, the following options: None (default), Down, Hold, **Pressed** (that we used above), Pulse, Released, Tap.

> Note that **Value Type** drop-down has the following options: **Digital (bool)** (default - we checked this above), Axis1D (float), Axis2D (Vector2D), and, Axis3D (Vector)

Setting Up IA_Sprint

The **IA_Sprint** action is pretty similar to the **IA_Interact** one but needs to trigger a **press** event when the character starts sprinting and trigger a **release** event when the character stops sprinting.

- Double-click on the **IA_Sprint** asset to open it
- On the **Action** → **Triggers** click on the **+** button **twice** to add **two** triggers.
- Select **Pressed** for the first trigger (as in IA_Interact's case).
- Select **Released** for the second trigger (to stop sprinting).
- Leave the rest as it is, checking that Value Type has been set to the default value of **Digital (bool)**.
- *Take Snapshot*

Setting Up IA_Move - Open **IA_Move -** Change Value Type to **Axis2D (Vector2D) -** $Take\ Snapshot$

Setting Up IA_Look - Open **AI_Look -** Change Value Type to **Axis2D (Vector2D) -** $Take\ Snapshot$

Basic actions have been defined. Let's create the **Mapping Context** and set it up.

2.2.2.2 Setting up the Input Mapping Context

Mapping Context refers to a set of **Input Actions** that identifies a specific situation in which the player may find themselves. The "context" you need to create here is the base actions a character can do (move, look around, and interact):

- Open the **Content Browser** and create this asset:
  - Right-click on the Input folder and select **Input** → **Input Mapping Context**.
  - Name the newly created asset **IMC_Default**
- Double-click on it to **set it up**:
  - Click the **Mappings** → **+** button
  - In the drop-down menu created, select **IA_Interact**.
  - Repeat the above two steps three more times to add **IA_Sprint**, **IA_Move**, and **IA_Look**.
- Press **Ctrl + S** to save (you need to do this for all assets).
- $Take\ Snapshot$

> ⓘ If you forgot to save any assets, at the bottom right (in the Status Line), you'll have a button **nn Unsaved**. Click that to get the **Save Content** dialog box with the list of all **nn** unsaved assets selected. Just press the **Save Selected** button when ready, to save all.

Let's now map the actions with the input(s) the player will be using. For this game we will allow the player to use:

- Keyboard + Mouse, or,
- Controller (XBox, etc)

So far, all mappings are set to None; this means that no input will pass through this context.

Mapping IA_Interact:

- Expand **IA_Interact** (left triangle) and click on the **Keyboard** icon therein; select **I** (for **Interact**).
- Click on the **+** button to the right of the IA_Interact field to add another mapping.
- From the drop-down menu, select **Gamepad** → **Gamepad Face Button Bottom**.
- Alternatively, if you have a game controller connected to your PC, you can simply click on the keyboard icon and then press the corresponding button (for instance, the **A** button for an Xbox controller).

Mapping for IA_Sprint:

- Set **Left Shift** for if the player is using the keyboard.
- Set **Gamepad** → **Gamepad Left Thumbstick Button** for if the player is using the **controller** (this second option will allow the player to press the **left thumbstick** to sprint).

Mapping for IA_Move:

- Add **Gamepad** → **Gamepad Left Thumbstick 2D-Axis** to the mappings.
  - Additionally, add a **modifier** from the Modifiers list with the value **Dead Zone**.
- Add a **Keyboard** → **D** mapping with no modifiers to move **right**.
- Add a **Keyboard** → **A** mapping to move **left**.
  - Add a **modifier** from the Modifiers list with the value **Negate**.
- Add a **Keyboard** → **W** mapping to move **forward**.
  - Add a **modifier** from the Modifiers list with the value **Swizzle Input Axis Values**.
- Add a **Keyboard** → **S** mapping to move **backward**.
  - Add a **modifier** from the Modifiers list with the value **Swizzle Input Axis Values**.
  - Add another **modifier** from the Modifiers list with the value **Negate**.

> ⓘ **Dead Zone** modifier forces the thumbstick to not send data when in the rest position.

> ⓘ

Negate modifier does just that, makes the values obtained (that are normally positive), negative.

Swizzle Input Axis Values modifier will convert **x** values into **y** (and vice versa), so you'll get a "forward" value for your character.

Mapping for IA_Look:

- Select **Gamepad** → **Gamepad Right Thumbstick 2D-Axis** for the *controller*.
    - Add a modifier from the Modifiers list with a value of **Dead Zone**
- Select a **Mouse** → **Mouse XY 2D-Axis** interaction for the mouse.
    - Add a modifier from the Modifiers list with the value **Negate**
        - Uncheck the **X** and **Z** checkboxes, leaving just the **Y** value selected. This will grant negative values for the mouse interaction – for instance, moving it forward will let the character move the camera down, and moving it backward will move the camera up.
- $Take\ Snapshot$ of all mappings so far.

Let's now set up the character so that it can receive input from the player.

2.2.3 Importing the Enhanced Input Module

As we are using the **Enhanced Input System**, we need to add it to the module declaration in the **Build.cs** file. - Open **VS IDE** (f.e. . - Open **US_LOTL_{YourInitials}.Build.cs** in the Source folder of your C++ project. - Locate the following line of code:

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

- Change it by adding the Enhanced Input module (if not there already):

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

This will make the Enhanced Input module available to your project, and you'll be ready to start implementing the user interaction, something that you'll do right now.

2.2.4 Adding user interaction to the character

To add user interaction to the character, we will:

- Declare the Mapping Context and action references to your code, along with the corresponding methods.
- Next, we'll implement the code logic needed to handle all actions.
- Finally, we'll declare these actions inside the character Blueprint.

2.2.4.1 Declaring input properties and functions

- Open the **US_Character.h** header file.
- Check that it has the following line of code (if not, add it as a public declaration):

```
virtual void SetupPlayerInputComponent(UInputComponent* PlayerInputComponent) override;
```

- Declare a pointer to the **Input Mapping Context** and a pointer for each **Input Action**.
    - Add the following code in the class implicit private section (i.e., just after the GENERATED_BODY() macro), just after the components declarations:

```
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Input", meta = (AllowPrivateAccess = "true"))
TObjectPtr<class UInputMappingContext> DefaultMappingContext;
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Input", meta = (AllowPrivateAccess = "true"))
TObjectPtr<class UInputAction> MoveAction;
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Input", meta = (AllowPrivateAccess = "true"))
TObjectPtr<UInputAction> LookAction;
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Input", meta = (AllowPrivateAccess = "true"))
TObjectPtr<UInputAction> SprintAction;
```

```
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Input", meta = (AllowPrivateAccess = "true"))
TObjectPtr<UInputAction> InteractAction;
```

- Add the following methods in the protected section, just after the BeginPlay() method declaration:

```
void Move(const struct FInputActionValue& Value);
void Look(const FInputActionValue& Value);
void SprintStart(const FInputActionValue& Value);
void SprintEnd(const FInputActionValue& Value);
void Interact(const FInputActionValue& Value);
```

> We added a method for each of the interactions you defined before. All these methods accept one argument named **Value** which is a reference to the structure **FInputActionValue**.

> In the **IA_Sprint** asset, we declared a **Pressed** and a **Released** trigger, so you will need to handle them with two corresponding methods (i.e., **SprintStart()** and **SprintEnd()**).

2.2.4.2 Implementing the Mapping Context for the character

Let's implement the above methods:

- Open **US_Character.cpp**.
- Add the following block of code, which includes all the classes you'll be using in the next steps:

```
// Inputs
#include "Components/InputComponent.h"
#include "GameFramework/Controller.h"
#include "EnhancedInputComponent.h"
#include "EnhancedInputSubsystems.h"
```

- Look for the *BeginPlay()* method and, after the *Super* declaration, add this block of code:

```
if (APlayerController* PlayerController = Cast<APlayerController>(Controller))
{
   if (UEnhancedInputLocalPlayerSubsystem* Subsystem =
   ULocalPlayer::GetSubsystem<UEnhancedInputLocalPlayerSubsystem>(PlayerController->GetLocalPlayer()))
   {
      Subsystem->AddMappingContext(DefaultMappingContext, 0);
   }
}
```

Here above, the first line of code checks that the Controller is a PlayerController by means of a Cast template.

> The Unreal's **Cast** is slightly different that the **cast** in pure C++; It is possible to safely cast to types that may not be valid. While in regular C++ that results in a crash, in Unreal we get as a return a safer **nullptr**, which in the context of **if** behaves as **false**.

Then, the code above:

- Tries to get an **Enhanced Input Subsystem** from the player (same **Cast** comment applies) , and,
- If successful, adds a **Mapping Context** to it.
- Note that from this point on, all the actions declared inside the context will be **"tracked"** by the *input system*.

Let's **bind** these actions to a corresponding method implementation (i.e., move, sprint, interact, etc.). - Look for the **SetupPlayerInputComponent()** method. - After the Super() declaration, add this block of code:

```
if (UEnhancedInputComponent* EnhancedInputComponent = Cast<UEnhancedInputComponent>(PlayerInputComponent))
{
   EnhancedInputComponent->BindAction(MoveAction, ETriggerEvent::Triggered, this, &AUS_Character::Move);
   EnhancedInputComponent->BindAction(LookAction, ETriggerEvent::Triggered, this, &AUS_Character::Look);
   EnhancedInputComponent->BindAction(InteractAction, ETriggerEvent::Started, this, &AUS_Character::Interact);
   EnhancedInputComponent->BindAction(SprintAction, ETriggerEvent::Started, this, &AUS_Character::SprintStart);
   EnhancedInputComponent->BindAction(SprintAction, ETriggerEvent::Completed, this, &AUS_Character::SprintEnd);
}
```

We are calling the **BindAction()** method on the enhanced input component pointer to **bind** each action to the corresponding method; also, we are using the values of the enumeration **ETriggerEvent** to specify the type of the trigger event. You can inspect this by hoovering over it, right-clicking and selecting **Peek Definitin** (or press **Alt+F12**).

2.2.4.3 Implementing the actions

Let's implement the methods for each action.

Move method:

- Add the following block of code:

```
void AUS_Character::Move(const FInputActionValue& Value)
{
  const auto MovementVector = Value.Get<FVector2D>();
  GEngine->AddOnScreenDebugMessage(0, 5.f, FColor::Yellow
      , FString::Printf(TEXT("MovementVector: %s"), *MovementVector.ToString()));
  if (Controller != nullptr)
  {
    const auto Rotation = Controller->GetControlRotation();
    const FRotator YawRotation(0, Rotation.Yaw, 0);
    const auto ForwardDirection = FRotationMatrix(YawRotation).GetUnitAxis(EAxis::X);
    const auto RightDirection = FRotationMatrix(YawRotation).GetUnitAxis(EAxis::Y);
    AddMovementInput(ForwardDirection, MovementVector.Y);
    AddMovementInput(RightDirection, MovementVector.X);
  }
}
```

Above we:

- Get the **2D vector** from the **Value** parameter. This vector contains the x and y directions for the left thumbstick (or the keyboard) and indicates the direction in which the character should move.
- We show an **onscreen message** to keep track of this value.
- If there is a Controller possessing this Actor, we:
    - compute the **forward** and **right** directions of the Character, and
    - **move** it in the corresponding direction

Look() method:

- Add these lines just after the Move() function:

```
void AUS_Character::Look(const FInputActionValue& Value)
{
  const auto LookAxisVector = Value.Get<FVector2D>();
  GEngine->AddOnScreenDebugMessage(1, 5.f, FColor::Green
      , FString::Printf(TEXT("LookAxisVector: %s"), *LookAxisVector.ToString()));
  if (Controller != nullptr)
  {
    AddControllerYawInput(LookAxisVector.X);
    AddControllerPitchInput(LookAxisVector.Y);
  }
}
```

Above we:

- Get the **2D vector** from the **Value** parameter (comes from the **right thumbstick** or the **mouse**).
- Again we show an **onscreen message** to keep track of this value.
- If there is a Controller possessing this Actor, we:
    - Add a **yaw/pitch** to the Controller; this will cause the Spring component and, consequently, the Camera component, to rotate around the character.

Sprint() method:

- Add this code block after the previous function:

```
void AUS_Character::SprintStart(const FInputActionValue& Value)
{
   GEngine->AddOnScreenDebugMessage(2, 5.f, FColor::Blue, TEXT("SprintStart"));
```

```
    GetCharacterMovement()->MaxWalkSpeed = 3000.f;
}
void AUS_Character::SprintEnd(const FInputActionValue& Value)
{
    GEngine->AddOnScreenDebugMessage(2, 5.f, FColor::Blue, TEXT("SprintEnd"));
    GetCharacterMovement()->MaxWalkSpeed = 500.f;
}
```

This code simply increases the maximum speed of the character when it is sprinting, otherwise goes back to the initial value.

> The walk and sprint values are hardcoded; we will need to get these values from a dataset later.

Interact() method:

- Since we don't have anything to interact with, we'll just add an on screen message inside the function for the time being:

```
void AUS_Character::Interact(const FInputActionValue& Value)
{
    GEngine->AddOnScreenDebugMessage(3, 5.f, FColor::Red, TEXT("Interact"));
}
```

The last thing you need to do to make the character fully functional is to add the input assets to the Blueprint.

2.2.4.4 Updating the character Blueprint

To update the Blueprint, take the following steps:

- **Save** all the files you have modified and get back to the Unreal Editor.
- Click the **Compile** button and wait for the success message.
- Open your **BP_Character** Blueprint and select the **Class Defaults** section.
- Search for the **Input** category in the **Details** panel. You should get the **Default Mapping Context** property along with the four actions that have been created.
- Click the drop-down button for **Default Mapping Context** and select the corresponding asset (there should be only one to choose).
- For each action property, select the corresponding **action asset** from the drop-down menu.
- $Take\ Snapshot$
- Save and Close the blueprint.

The character is finally complete! Let's test it.

2.2.5 Testing the character's movement

- Open the **Level_01** map and do the following:
  - Locate for the **Label - Spawn Point 1" label in the level**
  - **Add a **Player Start Actor** near it (via **Quickly Add to the Project** → **Basic** → **Actor -** the cube with the **+** green sign).
- Hit the **Play** button to test the game.
- Check that:
  - Characters can **Move** with WASD
  - They can **Sprint**
  - They can **Look** around
- $Take\ Snapshot$
- Stop the game.
- Set **Net Mode** to **Listen Server** with **3** players.
- Playtest as before.
- $Take\ Snapshot$ in all the windows.

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 you 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.

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