Whiskey Buisness/ Character Controller

Architecture/Design Document

Table of Contents

Contents

Change History

Version: 0.1

Modifier: Gabriel Hernandez

Date: 3 / 07 / 2025

Description of Change: Module Design Document started.

1. Introduction

This document delineates the architectural framework and design specifics of the character controller mechanics integrated within the ACharacterController class. This class is responsible for handling player input, movement, and interaction with the game environment.

The controller system processes input actions such as movement, jumping, attacking, and dashing while seamlessly integrating with the game's physics and animation systems.

For developers, this documentation provides insights into the implementation and expansion points for future development.

Project managers will gain an understanding of how this system impacts gameplay and performance, assisting in development planning. Maintenance programmers are equipped with a structured breakdown of the system for debugging and modifications.

2 Design Goals

The design of the Character Controller mechanic class is driven by several key objectives aimed at enhancing gameplay and ensuring technical robustness:

- **Ease of use:** The controller aims to select the best key binds for best enjoyment and an easy learning curve
- **Simple Implementation:** The controller is designed in such a way that adding new inputs is easy

- **Dual Controller Usability:** The controller blueprint was set up so that the player can use either a controller or the mouse and keyboard

3 System Behavior

The ACharacterController class processes input and directs the player's character within the game world. Below is an overview of its functional behavior:

Activation:

- Possession of Character
- Binding Actions
- Gamepad and Keyboard Support

Movement Mechanics:

- Directional Movement
- Dash Mechanic
- Jumping

Combat System Integration:

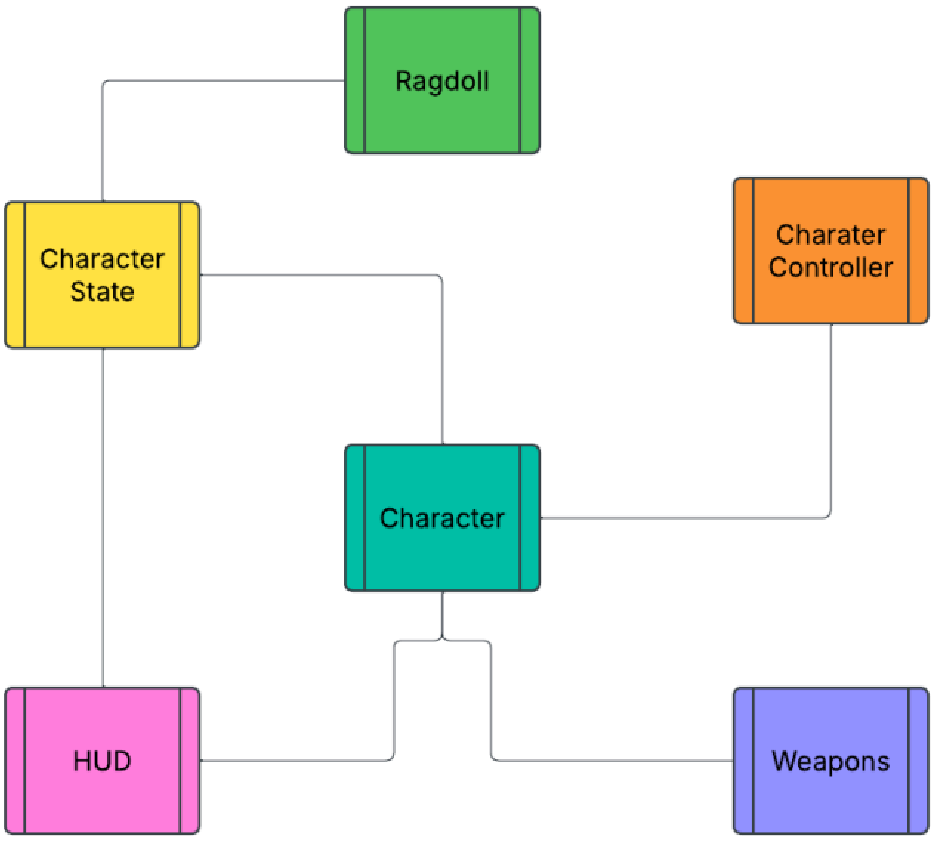
- Melee Attacks
- Throwing Objects
- Stun Mechanic

Mouse Interaction:

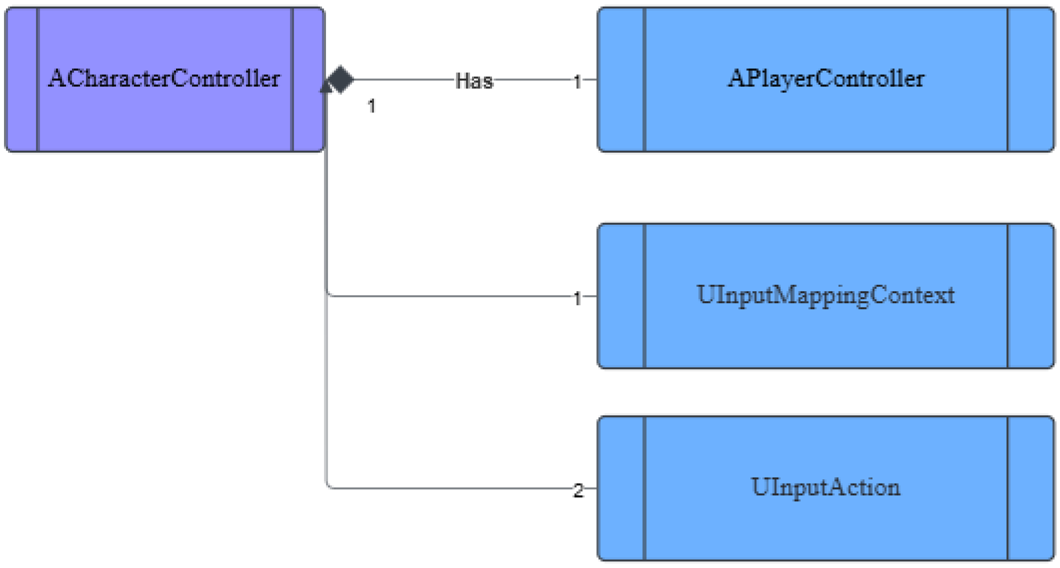
- Character Facing Direction

4 Logical View

4.1 High-Level Design



4.2 Mid-Level Setup Controller

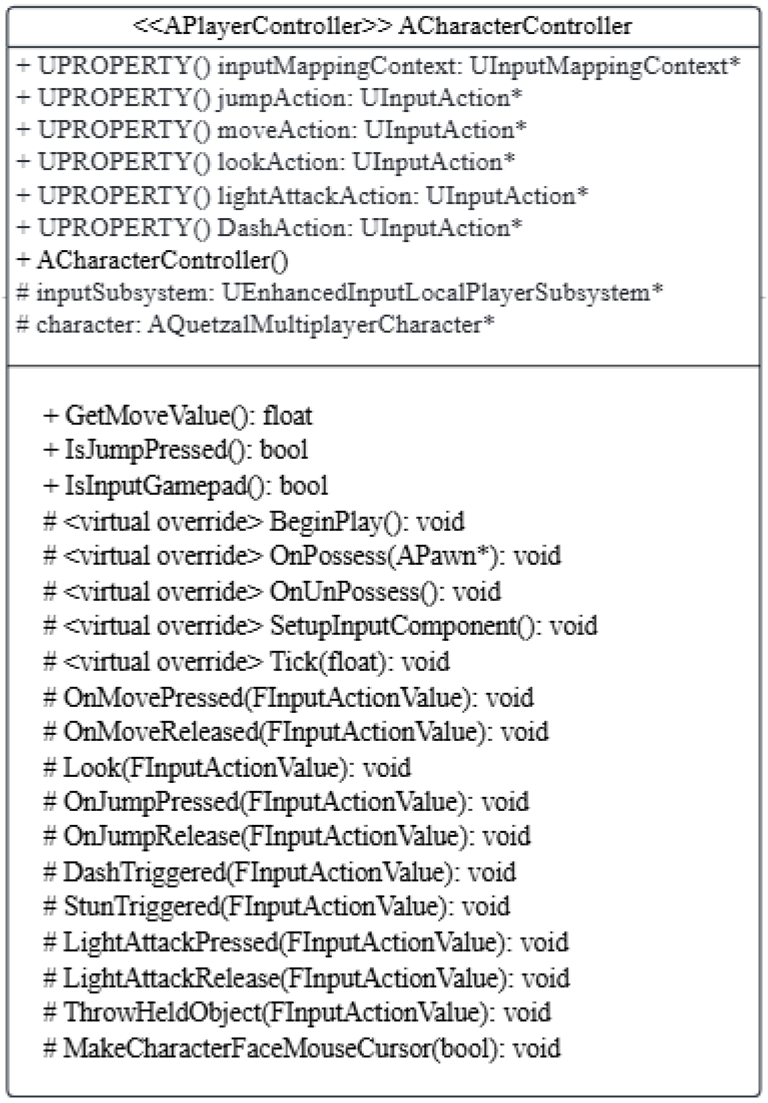


High-Level Design of varying systems in the game.

The Character Controller will handle the input from the user that will control the player. The HUD will keep the players informed of the current health and score of the players using values from Character and Character State.

Weapons will interact with the player by activating equipped weapons or receiving damage from weapons. Depending on the Character State the player can enter a Ragdoll state. At the center is the Character interacting with all systems in some varying ways.

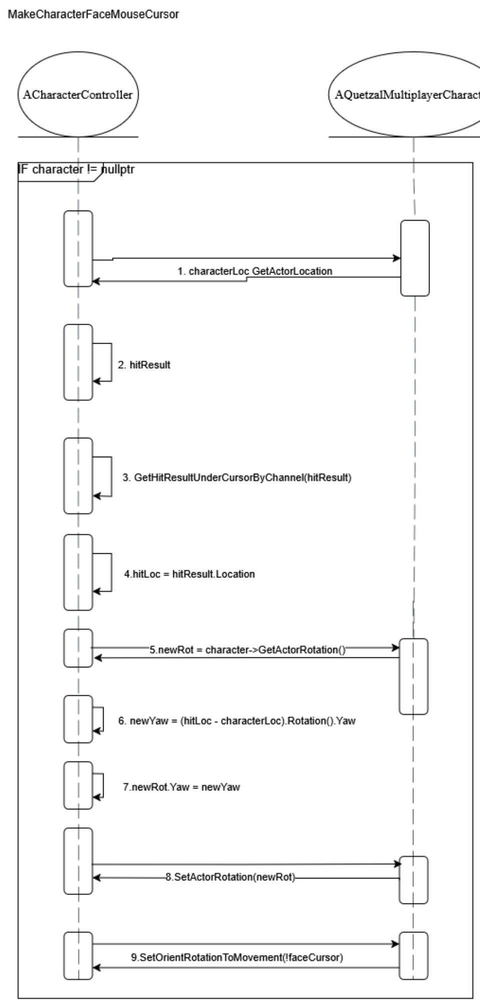
4.3 Detailed Class Design of Character Controller



```
<<APlayerController>> ACharacterController
+ UPROPERTY() inputMappingContext: UInputMappingContext*
+ UPROPERTY() jumpAction: UInputAction*
+ UPROPERTY() moveAction: UInputAction*
+ UPROPERTY() lookAction: UInputAction*
+ UPROPERTY() lightAttackAction: UInputAction*
+ UPROPERTY() DashAction: UInputAction*
+ ACharacterController()
# inputSubsystem: UEnhancedInputLocalPlayerSubsystem*
# character: AQuetzalMultiplayerCharacter*

+ GetMoveValue(): float
+ IsJumpPressed(): bool
+ IsInputGamepad(): bool
# <virtual override> BeginPlay(): void
# <virtual override> OnPossess(APawn*): void
# <virtual override> OnUnPossess(): void
# <virtual override> SetupInputComponent(): void
# <virtual override> Tick(float): void
# OnMovePressed(FInputActionValue): void
# OnMoveReleased(FInputActionValue): void
# Look(FInputActionValue): void
# OnJumpPressed(FInputActionValue): void
# OnJumpRelease(FInputActionValue): void
# DashTriggered(FInputActionValue): void
# StunTriggered(FInputActionValue): void
# LightAttackPressed(FInputActionValue): void
# LightAttackRelease(FInputActionValue): void
# ThrowHeldObject(FInputActionValue): void
# MakeCharacterFaceMouseCursor(bool): void
```

5 Process View of the CharacterController

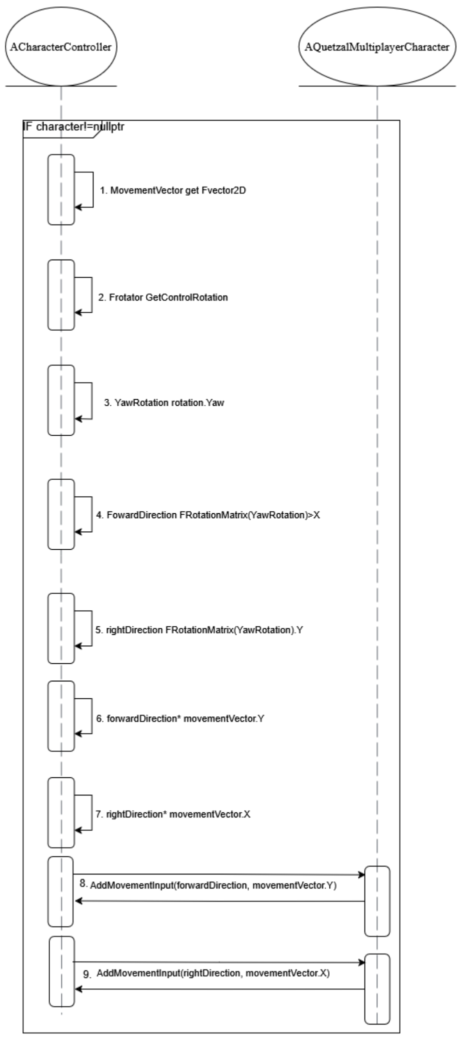
MakeCharacterFaceMouseCursor


MakeCharacterFaceMouseCursor
ACharacterController
AQuetzalMultiplayerCharacter
IF character != nullptr
1. characterLoc GetActorLocation
2. hitResult
3. GetHitResultUnderCursorByChannel(hitResult)
4.hitLoc = hitResult.Location
5.newRot = character->GetActorRotation()
6. newYaw = (hitLoc - characterLoc).Rotation().Yaw
7.newRot.Yaw = newYaw
8.SetActorRotation(newRot)
9.SetOrientRotationToMovement(!faceCursor)

1. If the character is not nullptr, get the actor location from character
2. Setup the hit result
3. Check the hit result putting the hitResult variable into the GetHitResultUnderCursorByChannel function using ConvertToTraceType(ECC_Visibility)
4. Set the hit location to the hit result from the function
5. Get the character actor rotation
6. Set up a new variable (newYaw) to the difference between the hit location and the character location
7. Set newRot.yaw to newYaw

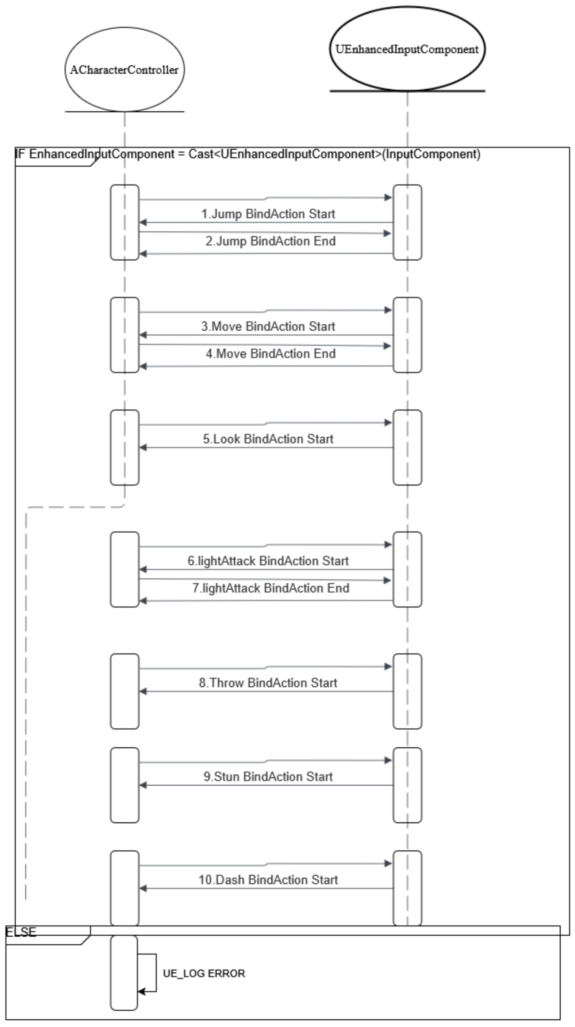
8. Set the actor rotation to the newRot to make the player look at the mouse cursor
9. Set the orientation rotation to the movement

OnMovePressed


ACharacterController
AQuetzalMultiplayerCharacter
IF character!=nullptr
1. MovementVector get Fvector2D
2. Frotator GetControlRotation
3. YawRotation rotation.Yaw
4. FowardDirection FRotationMatrix(YawRotation)>X
5. rightDirection FRotationMatrix(YawRotation).Y
6. forwardDirection* movementVector.Y
7. rightDirection* movementVector.X
8. AddMovementInput(forwardDirection, movementVector.Y)
9. AddMovementInput(rightDirection, movementVector.X)

1. If the character is not nullptr, get the movement vector passed in through the parameter
2. Set the rotation (as const FRotator) to the player control rotation (GetControlRotation())
3. Set the yawRoation Y value (as a const FRotator) from the rotation variable previously obtained.
4. Get the forward direction to orient the player controller (as a const FVector)
5. Get the right direction to orient the player controller (as a const FVector)
6. Set the forward direction variable to the movementVector.Y
7. Set the right direction variable to the movementVector.X
8. Set the foreward direction movement to the player (movementVector.Y)
9. Set the right direction movement to the player (movementVector.X)

SetupInputComponent



1-2. The jump action gets bound and returned to the controller.

3-4. The jump action gets bound and returned to the controller.

5. The look action gets bound

6-7. The light attack gets bound and returned to the controller

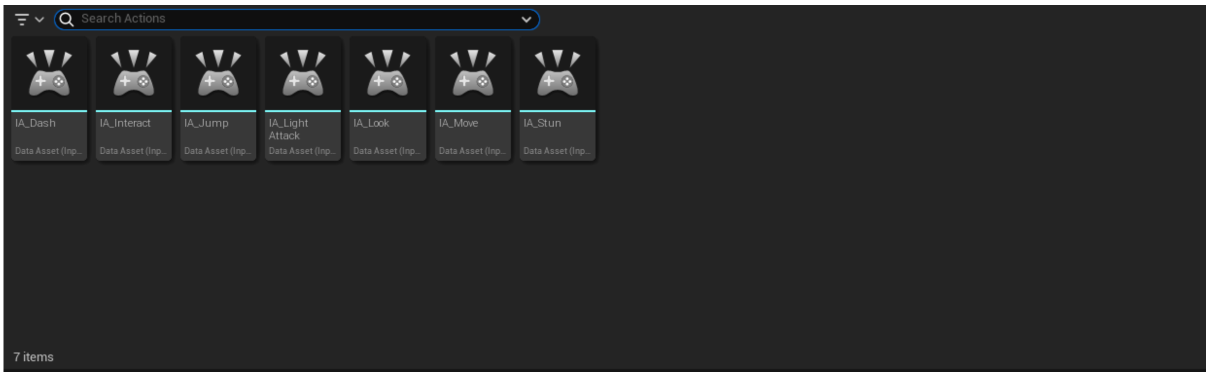
8. The throw action gets bound

9. The Stun action gets bound

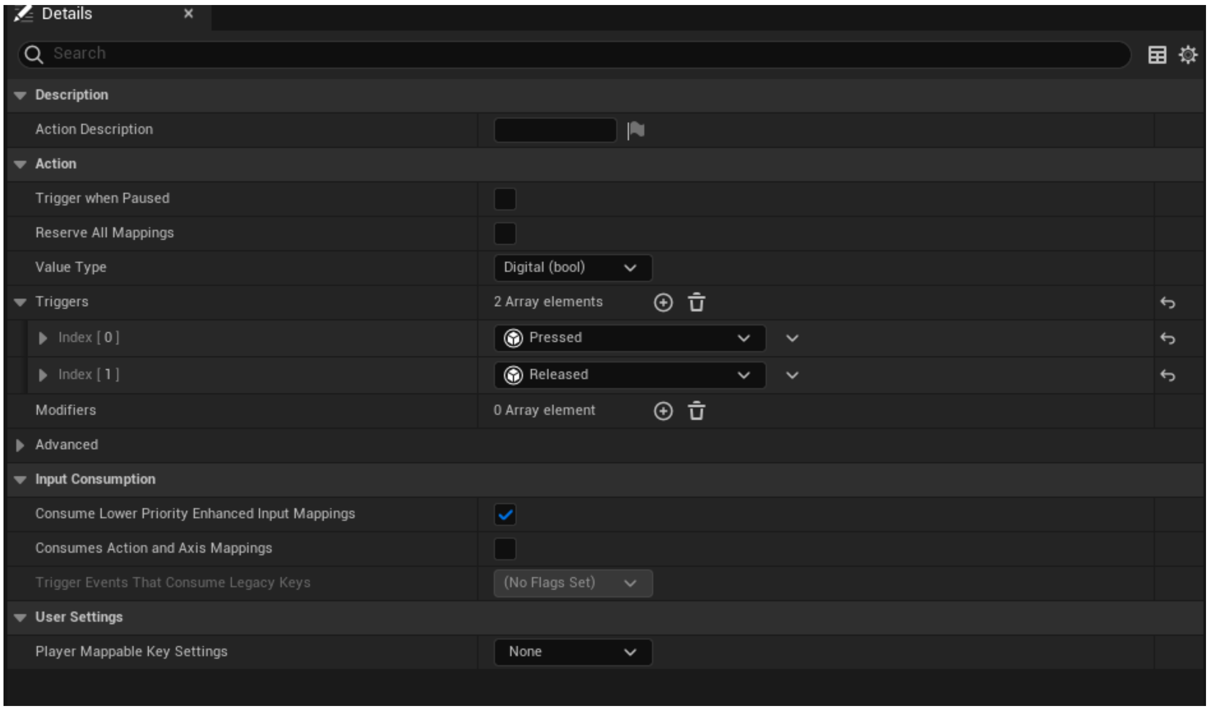
10. The dash action gets bound

6 Set Up Guide

1. Add a new input action



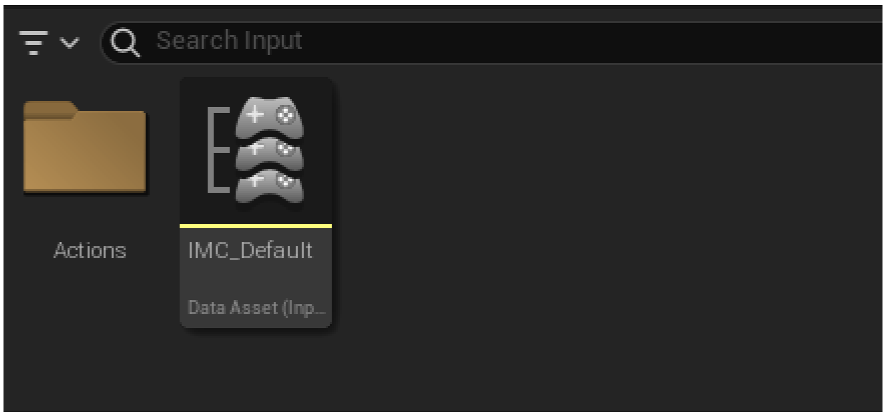
2. Set the value type (bool for a button, or axis for joystick movement) and the triggers

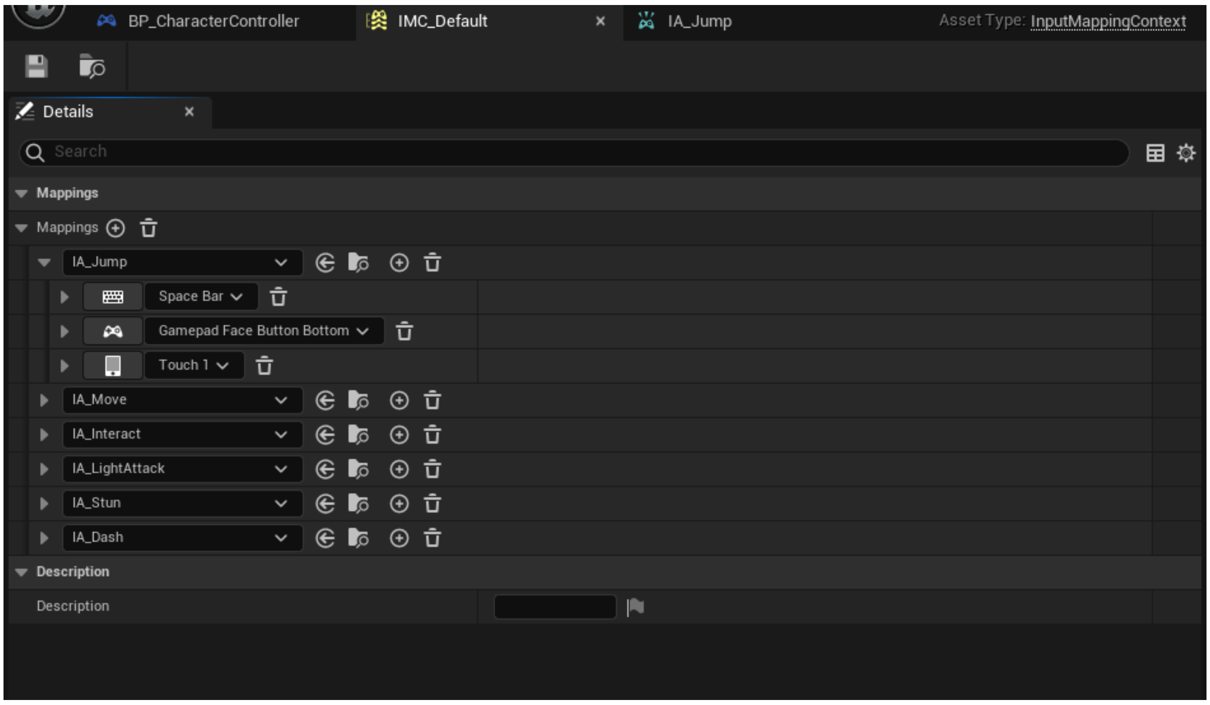

Details

Search

Description	
Action Description	
Action	
Trigger when Paused	☐
Reserve All Mappings	☐
Value Type	Digital (bool)
Triggers	2 Array elements
Index [0]	Pressed
Index [1]	Released
Modifiers	0 Array element
Advanced	
Input Consumption	
Consume Lower Priority Enhanced Input Mappings	✓
Consumes Action and Axis Mappings	☐
Trigger Events That Consume Legacy Keys	(No Flags Set)
User Settings	
Player Mappable Key Settings	None

3. Create a new input mapping context and add new mappings with the input actions. Here you will also set the hotkeys for each input action.



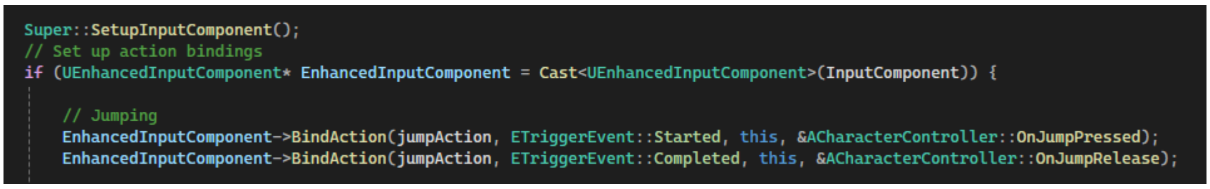


5. Create a character controller C++ class

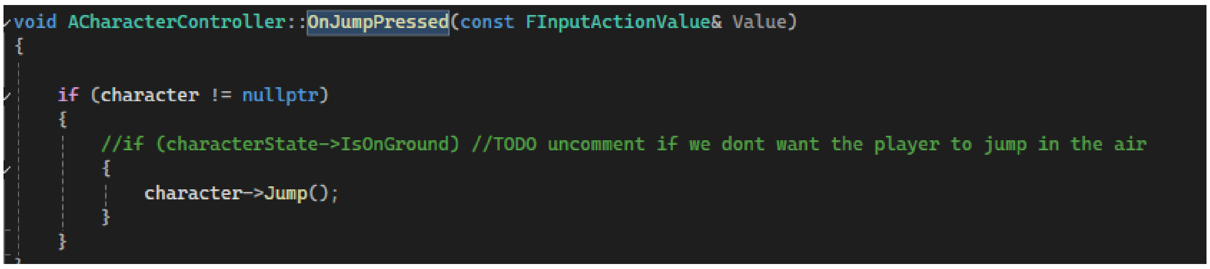
6. In the Character Controller.h file, add an input mapping context as a UPROPERTY

```
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = Input, meta = (AllowPrivateAccess = "true"))
UInputMappingContext* inputMappingContext;
```

7. Setup the input context and create a function for the input to call when hit

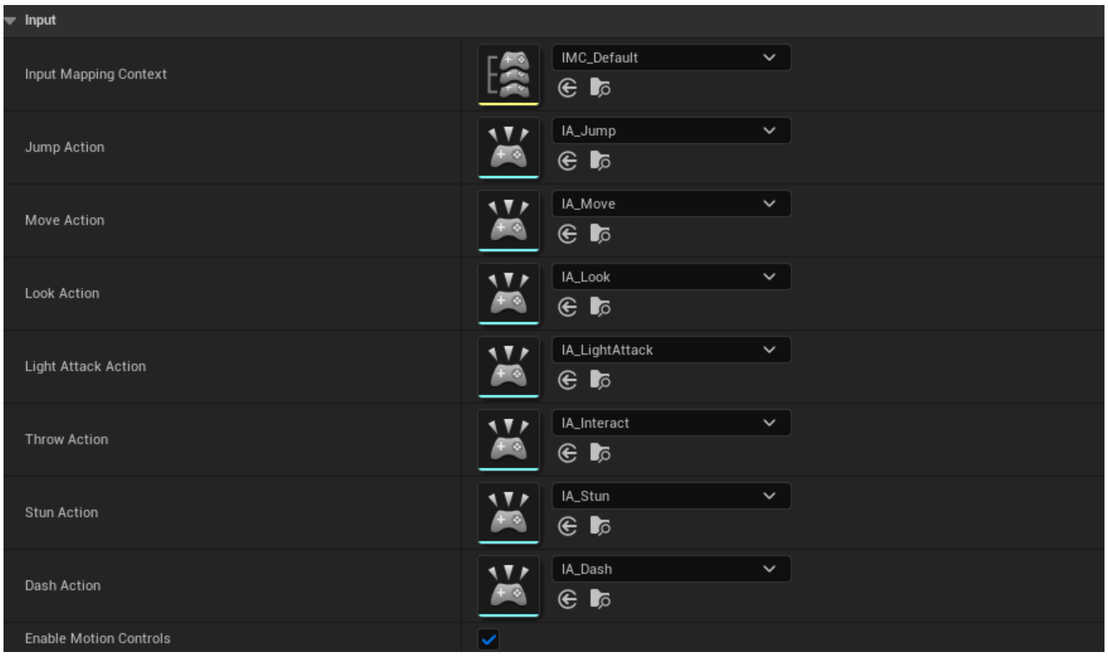


```
Super::SetupInputComponent();
// Set up action bindings
if (UEnhancedInputComponent* EnhancedInputComponent = Cast<UEnhancedInputComponent>(InputComponent)) {

    // Jumping
    EnhancedInputComponent->BindAction(jumpAction, ETriggerEvent::Started, this, &ACharacterController::OnJumpPressed);
    EnhancedInputComponent->BindAction(jumpAction, ETriggerEvent::Completed, this, &ACharacterController::OnJumpRelease);
```



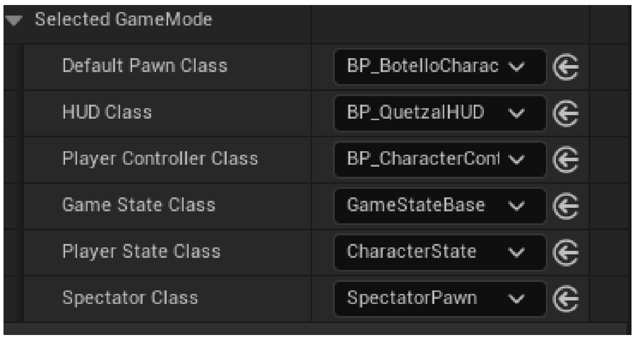
```
void ACharacterController::OnJumpPressed(const FInputActionValue& Value)
{

    if (character != nullptr)
    {
        //if (characterState->IsOnGround) //TODO uncomment if we dont want the player to jump in the air
        {
            character->Jump();
        }
    }
```

8. Create a blueprint for the created Controller class and set the IMC and all the action blueprints



Input	
Input Mapping Context	IMC_Default
Jump Action	IA_Jump
Move Action	IA_Move
Look Action	IA_Look
Light Attack Action	IA_LightAttack
Throw Action	IA_Interact
Stun Action	IA_Stun
Dash Action	IA_Dash
Enable Motion Controls	✓

9. Go to the world settings or the gamemode blueprint and set the created controller as the selected gamemode



Selected GameMode	
Default Pawn Class	BP_BotelloCharac
HUD Class	BP_QuetzalHUD
Player Controller Class	BP_CharacterCont
Game State Class	GameStateBase
Player State Class	CharacterState
Spectator Class	SpectatorPawn