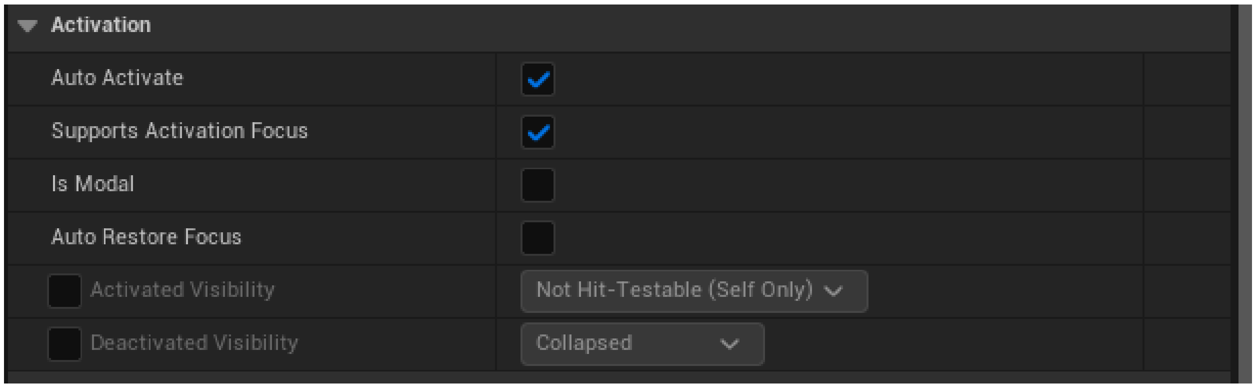
Additional Lesson

During the second week, we saw several things regarding the UI on UE5. But unfortunately, I wasn't able to deliver my entire chapter due to a lack of time so here's the rest of it.
You can either follow this lesson and add those new features to your project, or, just take mine and see what was added with this document.

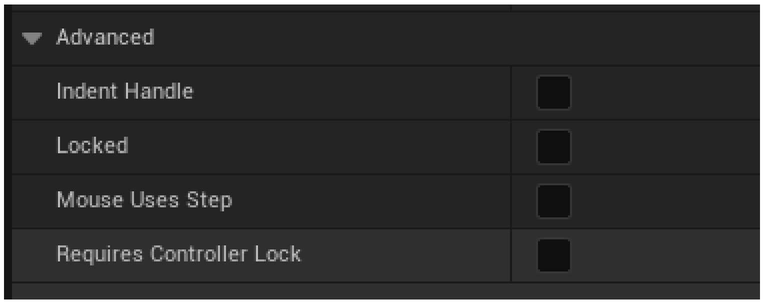
Note

In order to be sure that your UI works with a gamepad, please be sure to have the **Auto Activate** set to true in all your Widgets !



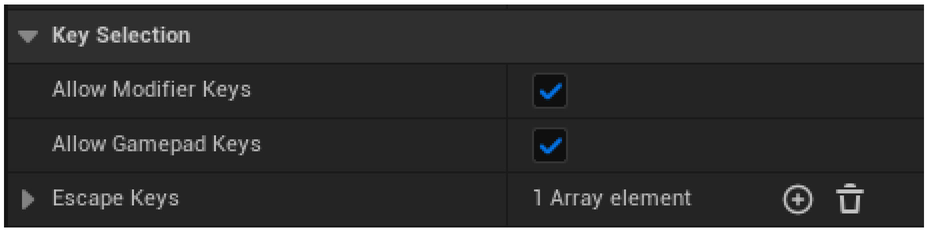
Activation	
Auto Activate	✓
Supports Activation Focus	✓
Is Modal	
Auto Restore Focus	
Activated Visibility	Not Hit-Testable (Self Only)
Deactivated Visibility	Collapsed

In addition to that, to use your Slider with a gamepad, please be sure in your **Option Menu** that for **BIND_XSlider** and **BIND_YSlider** the value **Require Controller Lock** is set to false.



Advanced	
Indent Handle	
Locked	
Mouse Uses Step	
Requires Controller Lock	

And finally, please be sure that the **Allow Gamepad Keys** is set to true for the **BIND_InputKeySelector** in the **WBP_KeyMapping**.



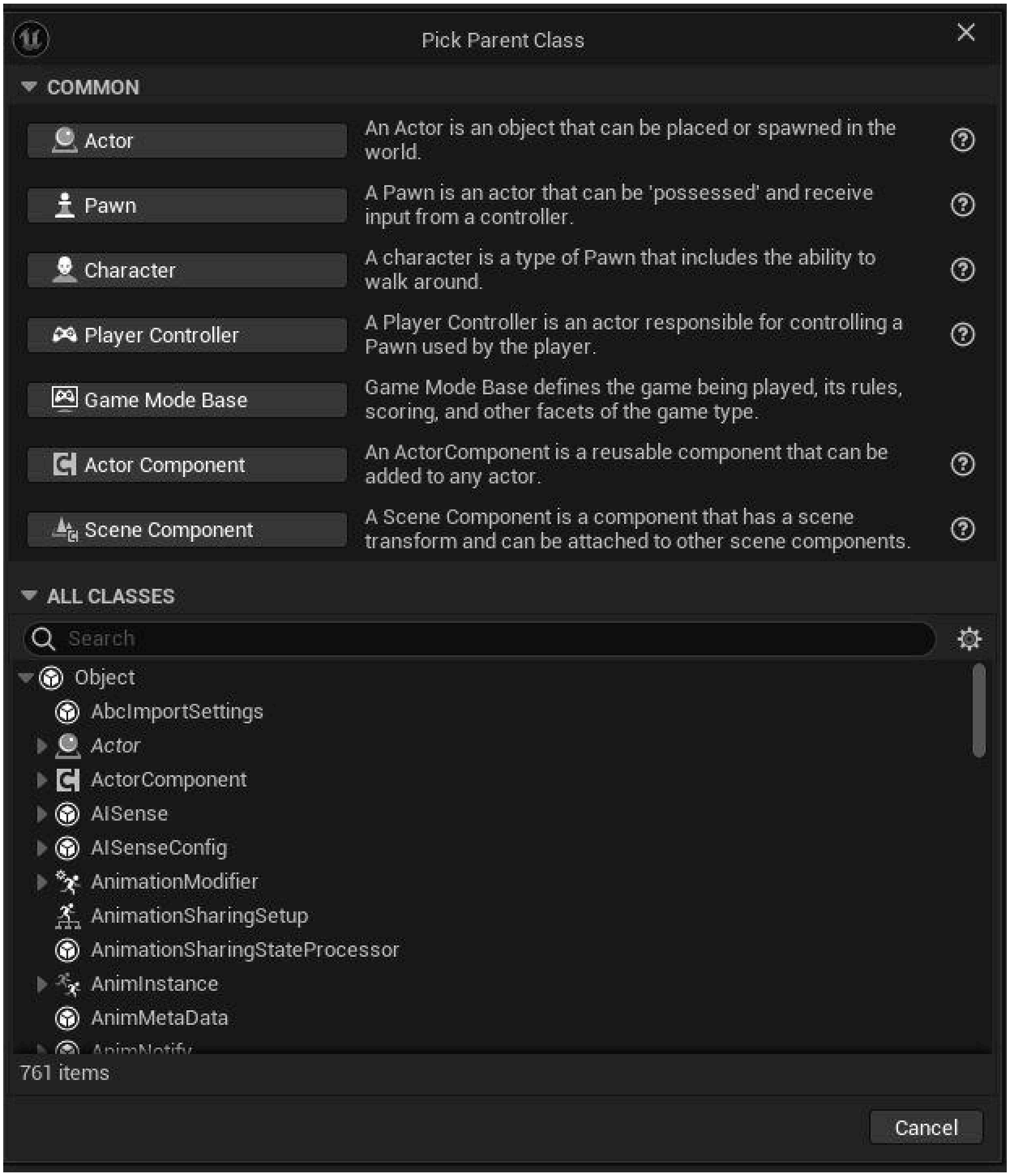
Key Selection	
Allow Modifier Keys	✓
Allow Gamepad Keys	✓
Escape Keys	1 Array element

Add a Main Menu

We will create a main menu from which we will have access to the options and from where we can launch the game. We will also have a map that will serve as background.

Create the Game Mode

Let’s start by creating a special **GameMode** for the main menu. I’m going to call it **BP_MainMenuGameMode**, which will be used to group the HUD and Character of the main menu.


Pick Parent Class

COMMON

Actor	An Actor is an object that can be placed or spawned in the world.
Pawn	A Pawn is an actor that can be 'possessed' and receive input from a controller.
Character	A character is a type of Pawn that includes the ability to walk around.
Player Controller	A Player Controller is an actor responsible for controlling a Pawn used by the player.
Game Mode Base	Game Mode Base defines the game being played, its rules, scoring, and other facets of the game type.
Actor Component	An ActorComponent is a reusable component that can be added to any actor.
Scene Component	A Scene Component is a component that has a scene transform and can be attached to other scene components.

ALL CLASSES

Search

- Object
  - AbcImportSettings
  - *Actor*
  - ActorComponent
  - AISense
  - AISenseConfig
  - AnimationModifier
  - AnimationSharingSetup
  - AnimationSharingStateProcessor
  - AnimInstance
  - AnimMetaData

761 items

Cancel

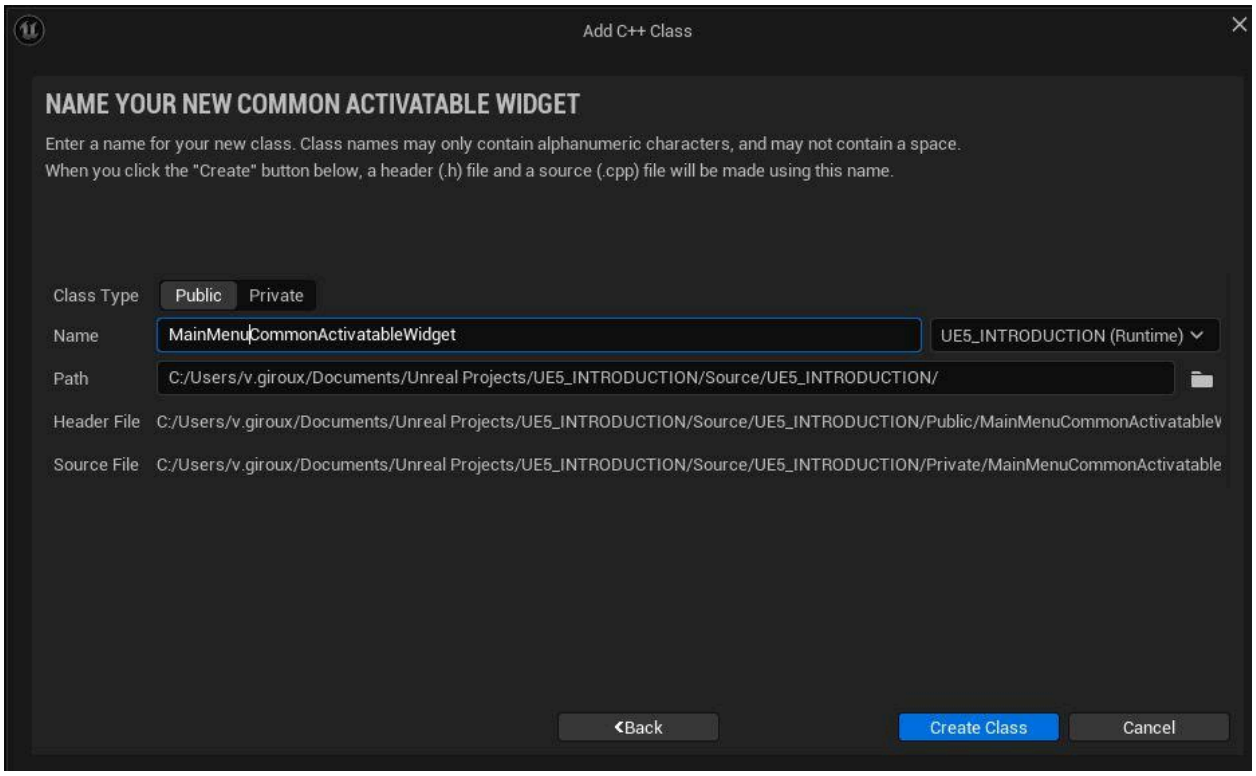
Create the Character

We will create a special **Character** for our menu that we will name **BP_MainMenuCharacter**. We can directly inherit it from Character.

We create a new character because we don’t need all the complexity of our basic character for the main menu. We create it directly in BP because we do not need to specialize its behavior either.

Create the User Widget

For the Menu, we will need a new **Common Activatable Widget** because we are going to create a different menu from the **Pause menu**. Let’s create a new class c++ **MainMenuCommonActivatableWidget** that inherits from **Common Activatable Widget** .

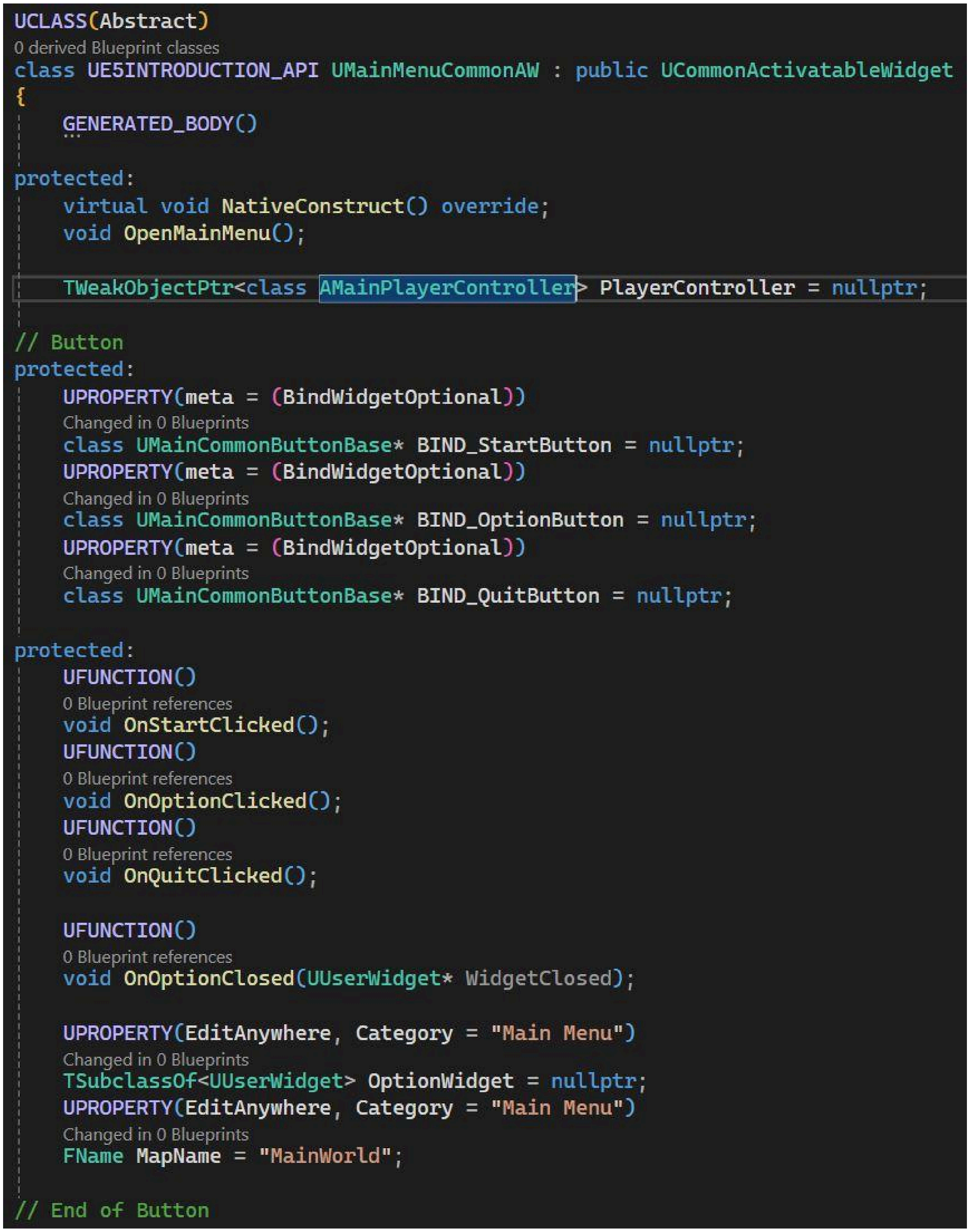


NAME YOUR NEW COMMON ACTIVATABLE WIDGET

Enter a name for your new class. Class names may only contain alphanumeric characters, and may not contain a space.
When you click the "Create" button below, a header (.h) file and a source (.cpp) file will be made using this name.

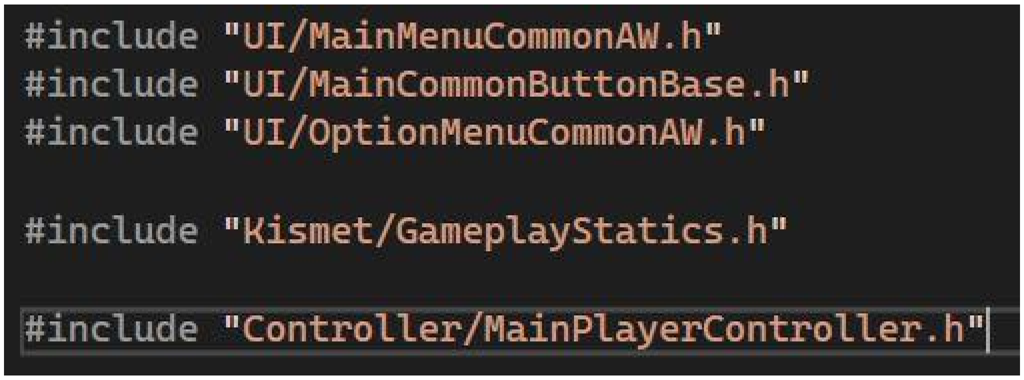
Class Type: [✓ Public] Private
Name: MainMenuCommonActivatableWidget — UE5_INTRODUCTION (Runtime)
Path: C:/Users/v.giroux/Documents/Unreal Projects/UE5_INTRODUCTION/Source/UE5_INTRODUCTION/
Header File: C:/Users/v.giroux/Documents/Unreal Projects/UE5_INTRODUCTION/Source/UE5_INTRODUCTION/Public/MainMenuCommonActivatableW
Source File: C:/Users/v.giroux/Documents/Unreal Projects/UE5_INTRODUCTION/Source/UE5_INTRODUCTION/Private/MainMenuCommonActivatable

In our menu, we will have 3 buttons, one to start the game, one for options and one to quit. We will also have a reference to the **Widget** menu option so that it can be opened and finally the name of the map of the game that will open when you launch a game.
In addition to that we have the 3 functions for the buttons and a callback function when we close the pause menu.
We will need a reference to the **Player Controller**. Please note that here it’s also a **AMainPlayerController.** This is because we need its reference in order to properly update the options.



```
UCLASS(Abstract)
0 derived Blueprint classes
class UE5INTRODUCTION_API UMainMenuCommonAW : public UCommonActivatableWidget
{
    GENERATED_BODY()

protected:
    virtual void NativeConstruct() override;
    void OpenMainMenu();

    TWeakObjectPtr<class AMainPlayerController> PlayerController = nullptr;

// Button
protected:
    UPROPERTY(meta = (BindWidgetOptional))
    Changed in 0 Blueprints
    class UMainCommonButtonBase* BIND_StartButton = nullptr;
    UPROPERTY(meta = (BindWidgetOptional))
    Changed in 0 Blueprints
    class UMainCommonButtonBase* BIND_OptionButton = nullptr;
    UPROPERTY(meta = (BindWidgetOptional))
    Changed in 0 Blueprints
    class UMainCommonButtonBase* BIND_QuitButton = nullptr;

protected:
    UFUNCTION()
    0 Blueprint references
    void OnStartClicked();
    UFUNCTION()
    0 Blueprint references
    void OnOptionClicked();
    UFUNCTION()
    0 Blueprint references
    void OnQuitClicked();

    UFUNCTION()
    0 Blueprint references
    void OnOptionClosed(UUserWidget* WidgetClosed);

    UPROPERTY(EditAnywhere, Category = "Main Menu")
    Changed in 0 Blueprints
    TSubclassOf<UUserWidget> OptionWidget = nullptr;
    UPROPERTY(EditAnywhere, Category = "Main Menu")
    Changed in 0 Blueprints
    FName MapName = "MainWorld";

// End of Button
```

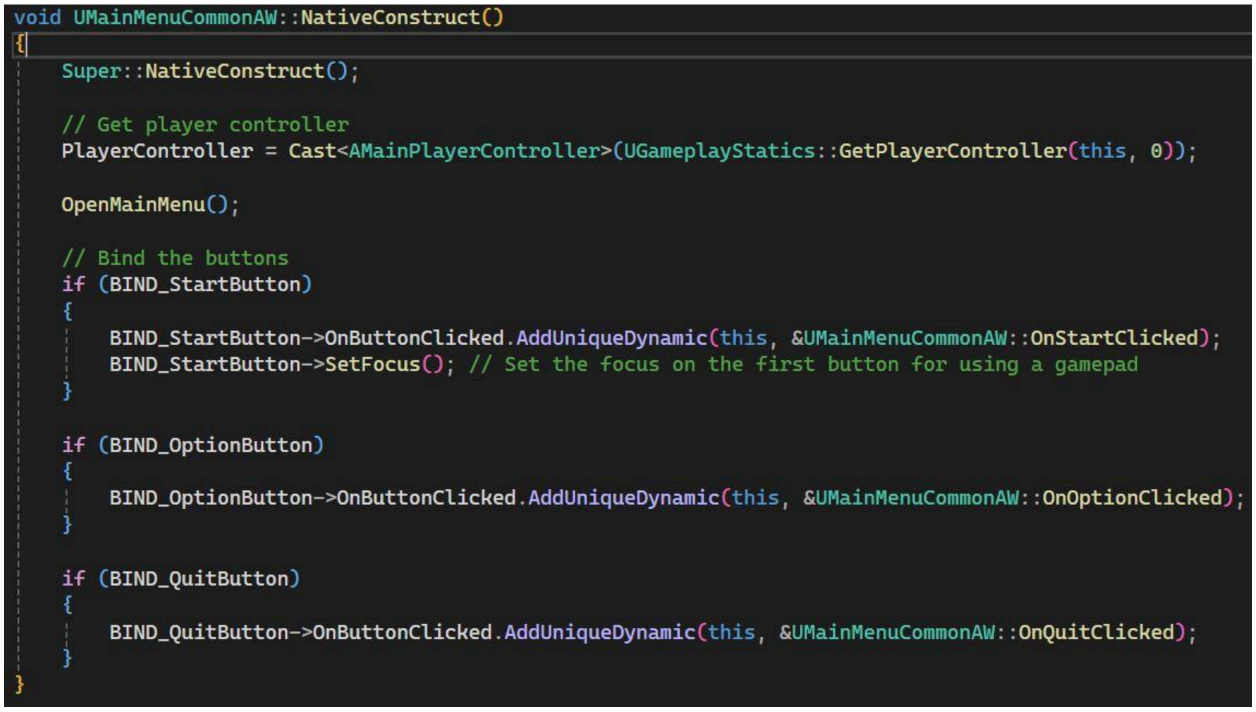
Let's start with the **Construct**. We will here bind on the Buttons, get a reference to **PlayerController** or open our main menu. The focus for the Gamepad will also be set on the **Start Button**.



```
#include "UI/MainMenuCommonAW.h"
#include "UI/MainCommonButtonBase.h"
#include "UI/OptionMenuCommonAW.h"

#include "Kismet/GameplayStatics.h"

#include "Controller/MainPlayerController.h"
```

When you open the menu, no need to pause the game. We could want to use a background scene in the main menu. We will also and pass the inputs in **UIOnly**.

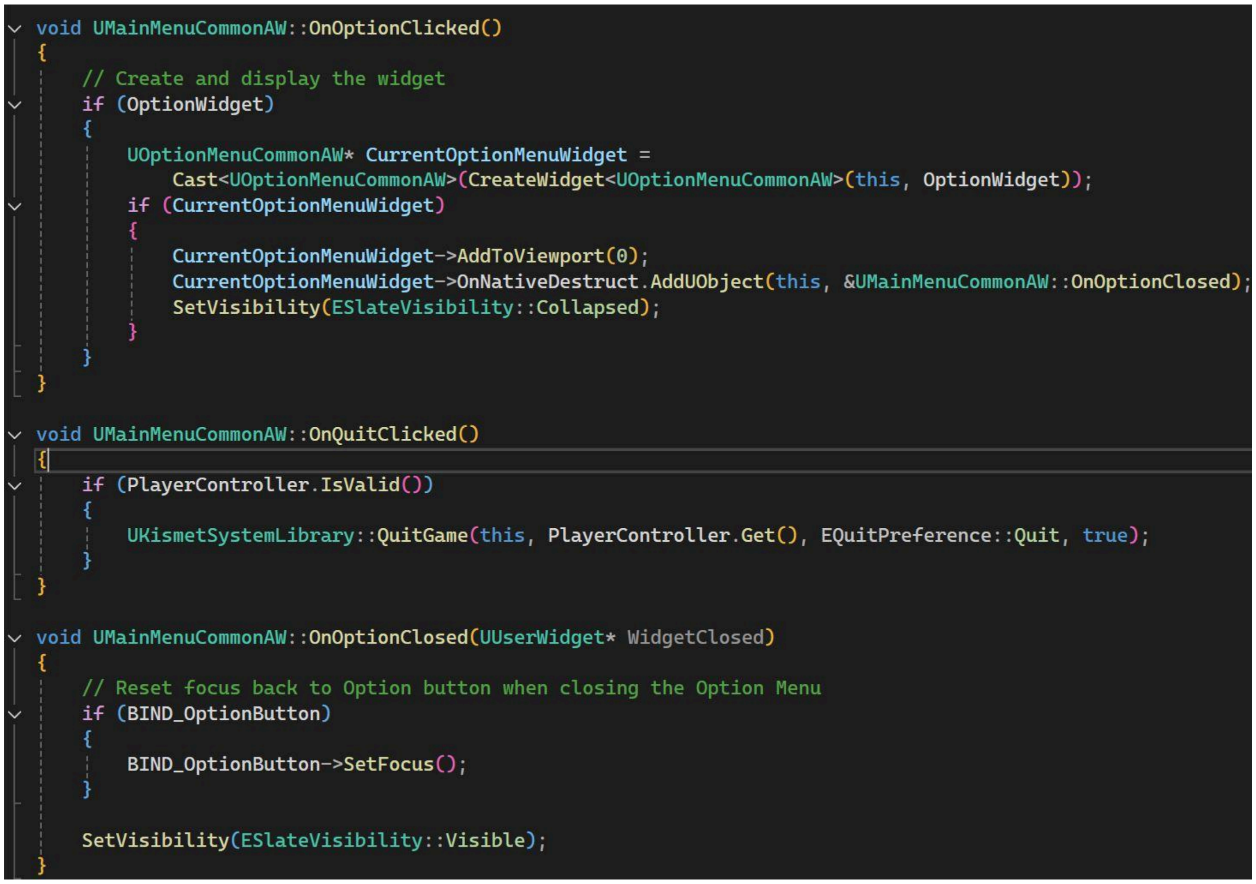
For the **Start**, we reset the Inputs in **GameMode**, remove the pause and open the base level.



```
void UMainMenuCommonAW::NativeConstruct()
{
    Super::NativeConstruct();

    // Get player controller
    PlayerController = Cast<AMainPlayerController>(UGameplayStatics::GetPlayerController(this, 0));

    OpenMainMenu();

    // Bind the buttons
    if (BIND_StartButton)
    {
        BIND_StartButton->OnButtonClicked.AddUniqueDynamic(this, &UMainMenuCommonAW::OnStartClicked);
        BIND_StartButton->SetFocus(); // Set the focus on the first button for using a gamepad
    }

    if (BIND_OptionButton)
    {
        BIND_OptionButton->OnButtonClicked.AddUniqueDynamic(this, &UMainMenuCommonAW::OnOptionClicked);
    }

    if (BIND_QuitButton)
    {
        BIND_QuitButton->OnButtonClicked.AddUniqueDynamic(this, &UMainMenuCommonAW::OnQuitClicked);
    }
}
```

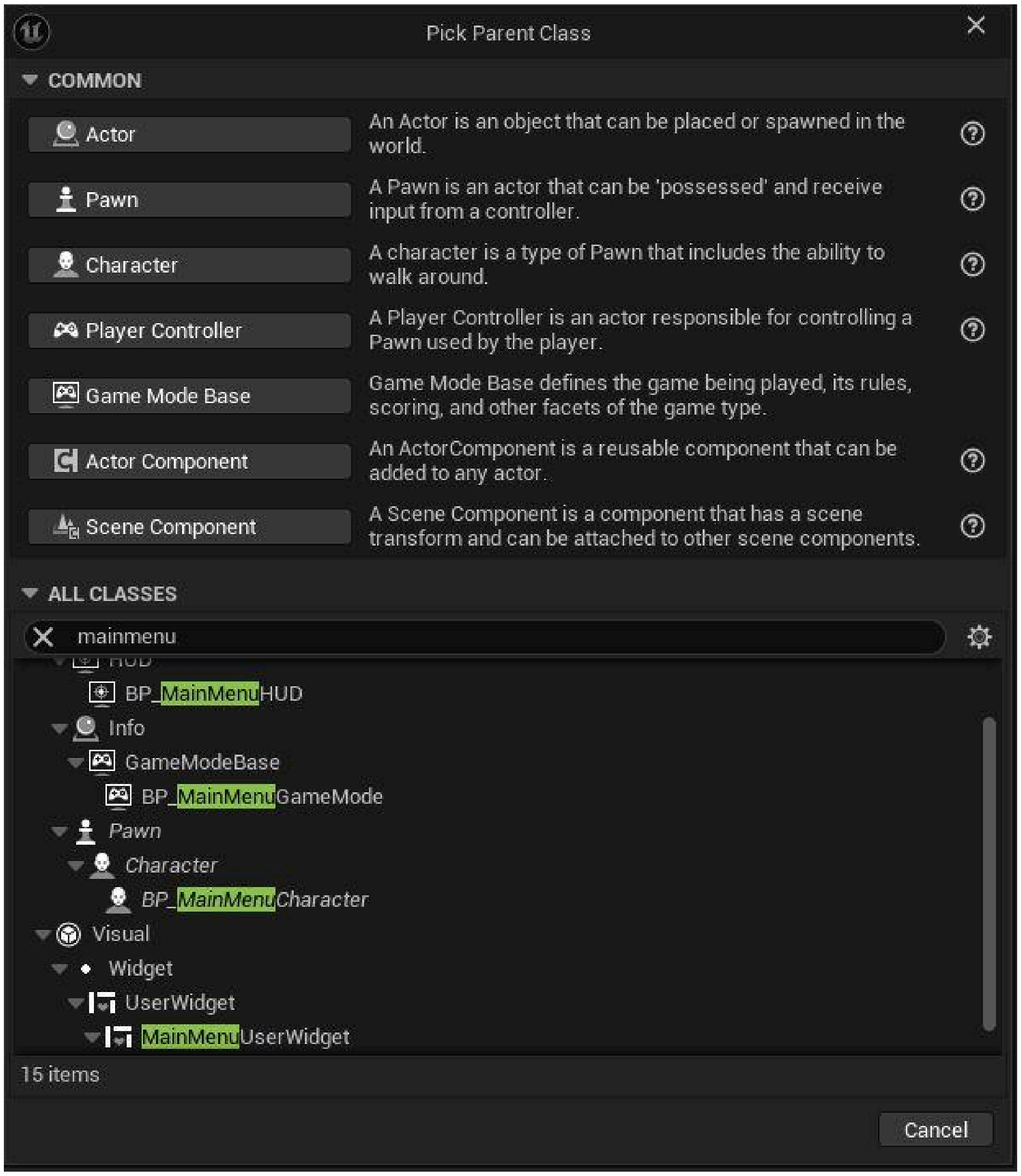
For the **Option**, we perform the same operation as in the pause menu, we open the **Widget**. Note that we will display a menu that will hide the screen, while our main menu will have a background image. We can recreate the option menu so that it does not have a grey image in the background if we really wanted to see the background of the Main Menu.
For the callback from closing the option menu, we will put the focus on the **Start** button.

Finally, to **Exit** the game, we use the same function as for the Pause Menu.



```
void UMainMenuCommonAW::OnOptionClicked()
{
    // Create and display the widget
    if (OptionWidget)
    {
        UOptionMenuCommonAW* CurrentOptionMenuWidget =
            Cast<UOptionMenuCommonAW>(CreateWidget<UOptionMenuCommonAW>(this, OptionWidget));
        if (CurrentOptionMenuWidget)
        {
            CurrentOptionMenuWidget->AddToViewport(0);
            CurrentOptionMenuWidget->OnNativeDestruct.AddUObject(this, &UMainMenuCommonAW::OnOptionClosed);
            SetVisibility(ESlateVisibility::Collapsed);
        }
    }
}

void UMainMenuCommonAW::OnQuitClicked()
{
    if (PlayerController.IsValid())
    {
        UKismetSystemLibrary::QuitGame(this, PlayerController.Get(), EQuitPreference::Quit, true);
    }
}

void UMainMenuCommonAW::OnOptionClosed(UUserWidget* WidgetClosed)
{
    // Reset focus back to Option button when closing the Option Menu
    if (BIND_OptionButton)
    {
        BIND_OptionButton->SetFocus();
    }

    SetVisibility(ESlateVisibility::Visible);
}
```

We can compile and go to take care of the display, we will create a **Widget** that inherits from this class. We will name it **WBP_MainMenu.**



Pick Parent Class

COMMON

Actor	An Actor is an object that can be placed or spawned in the world.
Pawn	A Pawn is an actor that can be 'possessed' and receive input from a controller.
Character	A character is a type of Pawn that includes the ability to walk around.
Player Controller	A Player Controller is an actor responsible for controlling a Pawn used by the player.
Game Mode Base	Game Mode Base defines the game being played, its rules, scoring, and other facets of the game type.
Actor Component	An ActorComponent is a reusable component that can be added to any actor.
Scene Component	A Scene Component is a component that has a scene transform and can be attached to other scene components.

ALL CLASSES

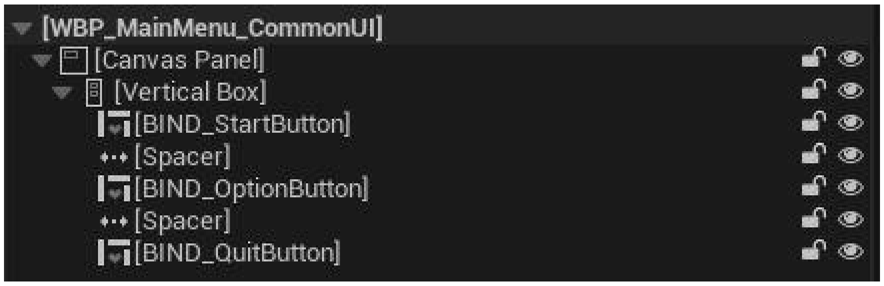
mainmenu

```
  HUD
    BP_MainMenuHUD
  Info
    GameModeBase
      BP_MainMenuGameMode
  Pawn
    Character
      BP_MainMenuCharacter
Visual
  Widget
    UserWidget
      MainMenuUserWidget
```

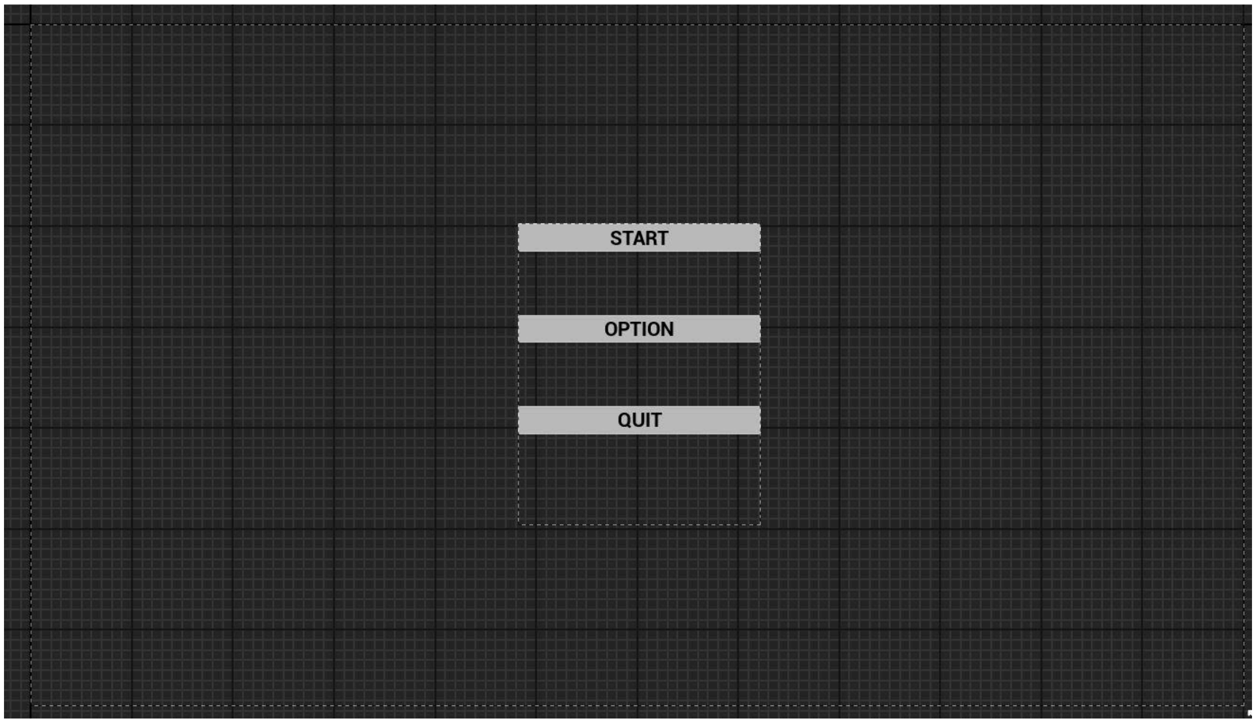
15 items

Cancel

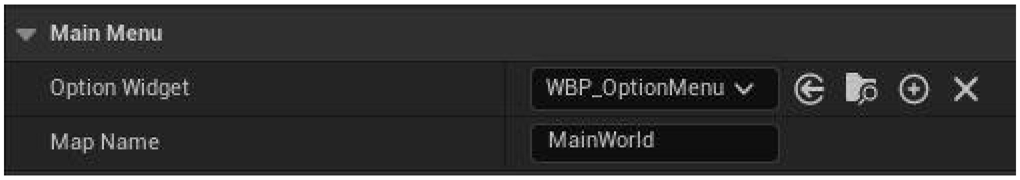
We can create a **Canvas Panel**, a **Vertical Box** that will center then the **Buttons** for display. I'll add some **Spacer** with a **Y** size of **100**.



```
[WBP_MainMenu_CommonUI]
  [Canvas Panel]
    [Vertical Box]
      [BIND_StartButton]
      [Spacer]
      [BIND_OptionButton]
      [Spacer]
      [BIND_QuitButton]
```



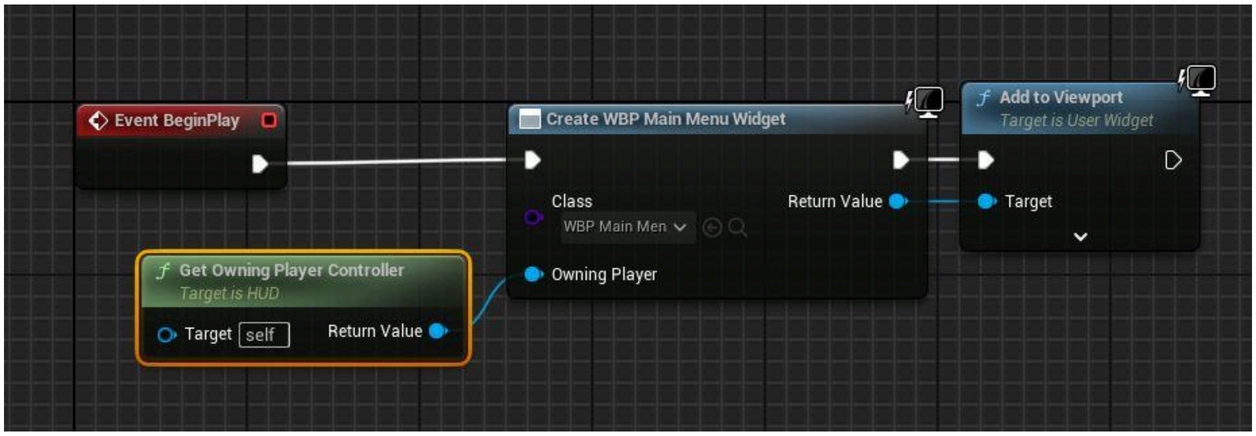
We can then provide our BP with the information regarding the menu option and the name of the map.


Main Menu

Option Widget	WBP_OptionMenu
Map Name	MainWorld

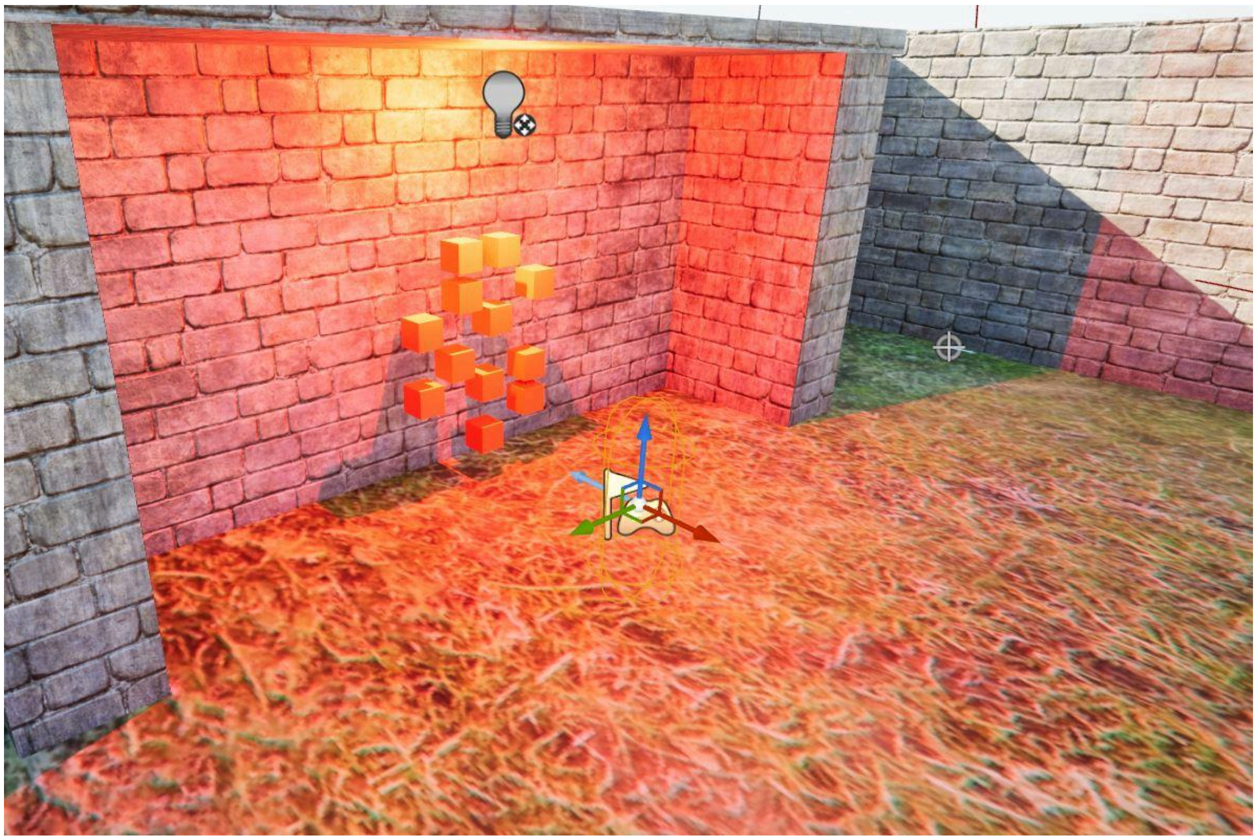
Create HUD

Our **User Widget** is ready, we can now create a **HUD** that will display it, we'll name it **BP_MainMenuHUD**. We will just take care of displaying our **Widget** eu **BeginPlay**.



Create the Map

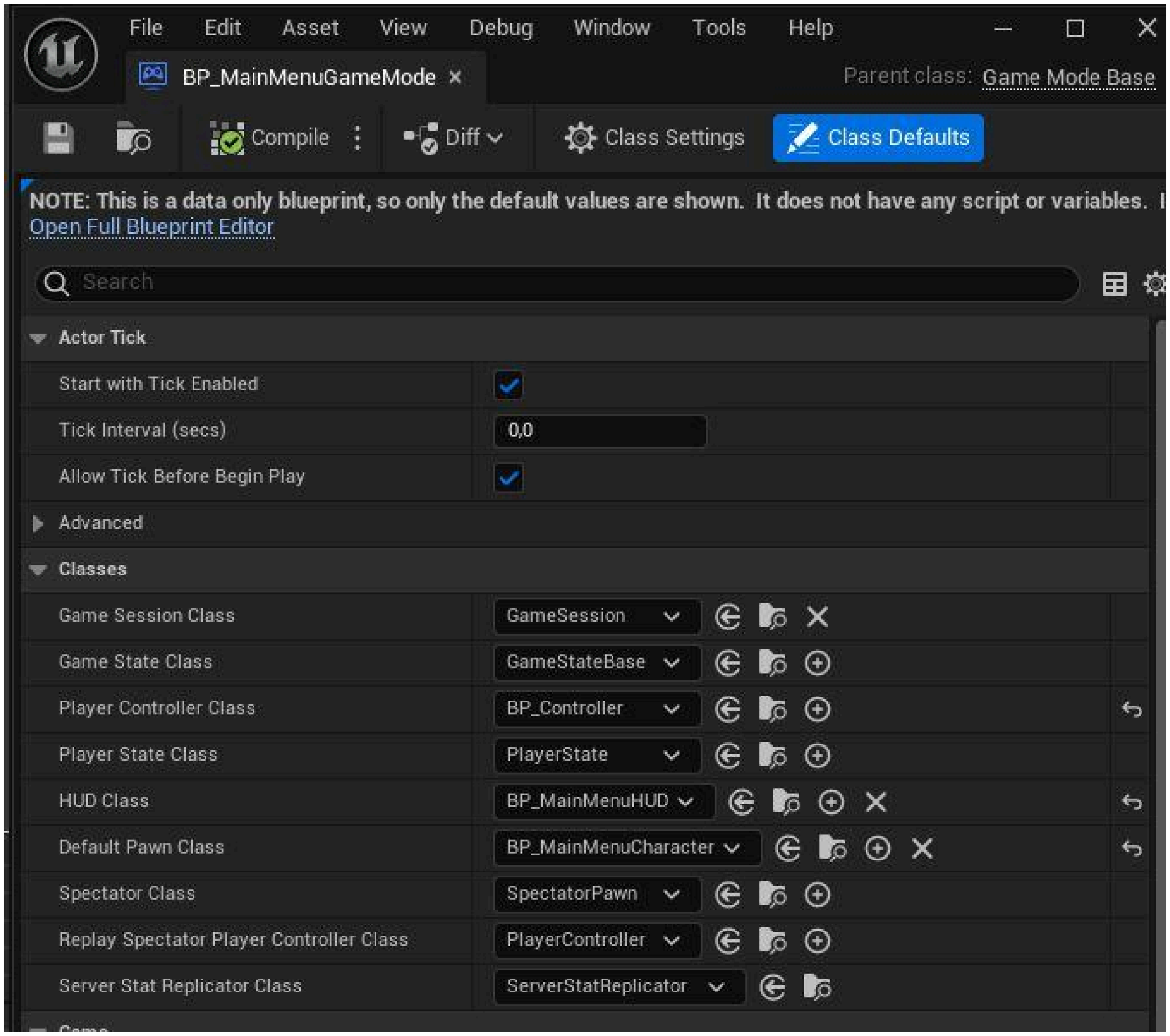
We will duplicate our current map and call the new **MainMenu**. This will allow us to have a background and be able to link our **GameMode** to this map.
Here the player will not be able to move, so we'll move the **PlayerStart** and put it in front of something that will serve as a background for the Main Menu. Here's a simple example.



Link the GameMode to the Map

Each map can have a **GameMode** assigned to it, as soon as we arrive on this map, the **GameMode** will be activated and we will then find ourselves with a different **HUD**, Character etc.
This is particularly useful here because we want to have a different **Character** and **HUD** for this **Main Menu.**

Let's start by updating our **GameMode** with the **HUD**, the **Controller** and the **Chara**.



NOTE: This is a data only blueprint, so only the default values are shown. It does not have any script or variables.
Open Full Blueprint Editor

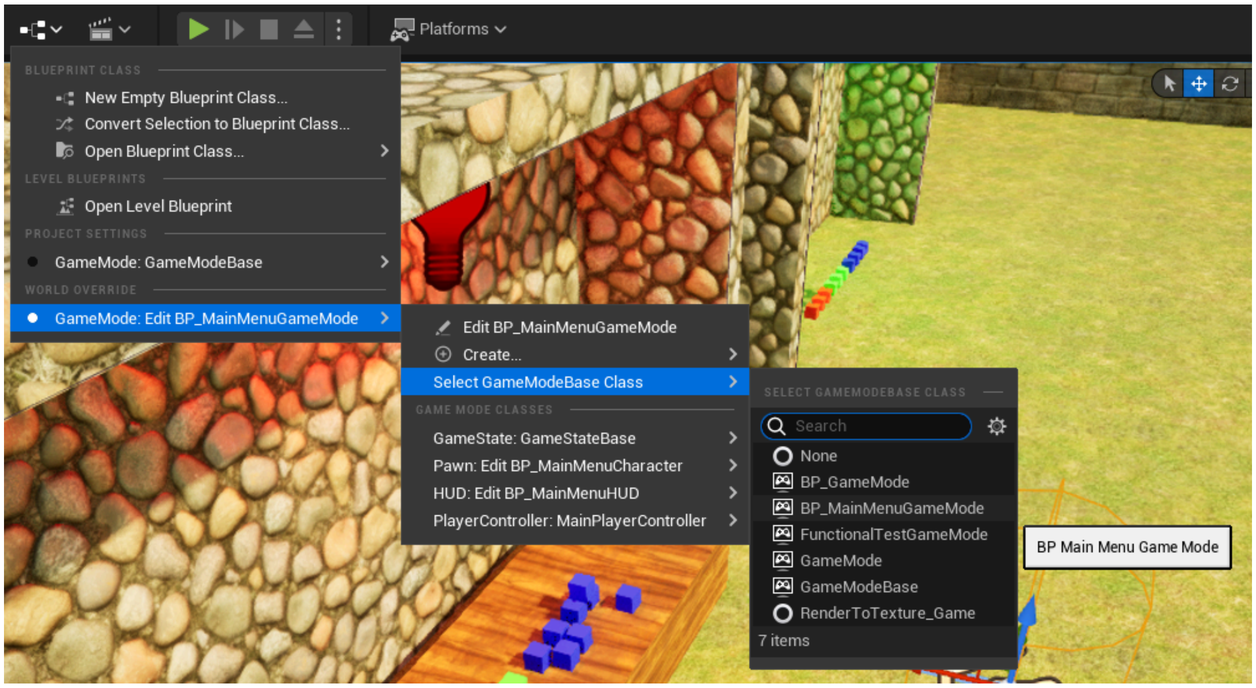
Actor Tick

Start with Tick Enabled	✓
Tick Interval (secs)	0,0
Allow Tick Before Begin Play	✓
Advanced	

Classes

Game Session Class	GameSession
Game State Class	GameStateBase
Player Controller Class	BP_Controller
Player State Class	PlayerState
HUD Class	BP_MainMenuHUD
Default Pawn Class	BP_MainMenuCharacter
Spectator Class	SpectatorPawn
Replay Spectator Player Controller Class	PlayerController
Server Stat Replicator Class	ServerStatReplicator

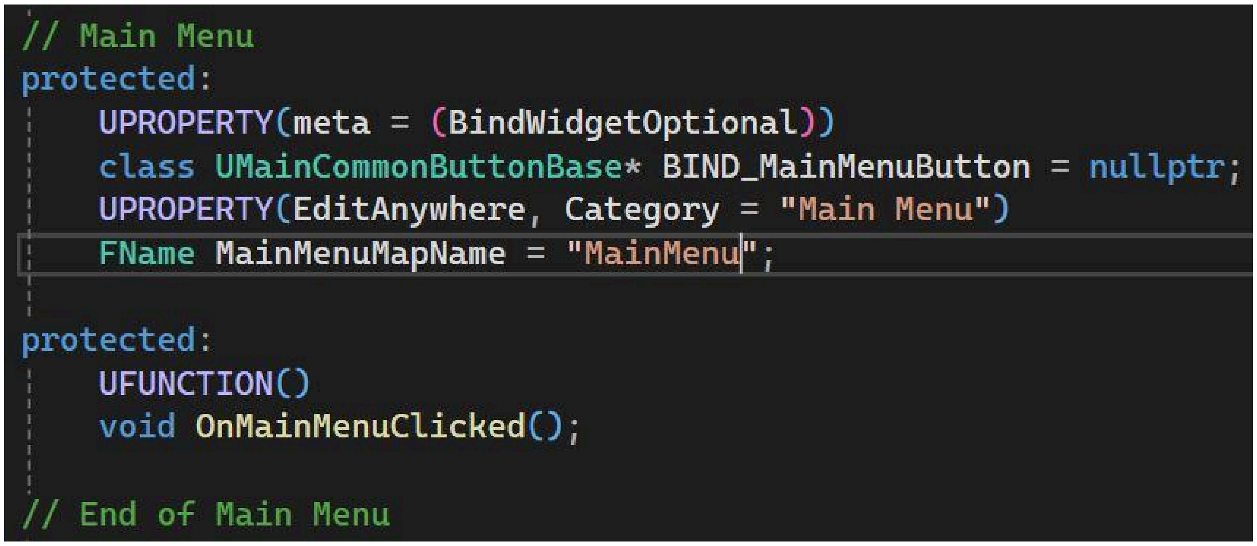
Now let's put the basic **GameMode** on our map. Open the **MainMenu** map and select the options of the map, then in **World Override > GameMode > SelectGameModeBaseClass**, to we can put our new and test.



Update Menu Pause

We're almost there! All we have to do is add a new button in our basic Pause Menu that allows you to return to the **Main Menu**.

Let's start by adding the button in our **PauseMenuUserWidget** and the name of the map to open.



```
// Main Menu
protected:
    UPROPERTY(meta = (BindWidgetOptional))
    class UMainCommonButtonBase* BIND_MainMenuButton = nullptr;
    UPROPERTY(EditAnywhere, Category = "Main Menu")
    FName MainMenuMapName = "MainMenu";

protected:
    UFUNCTION()
    void OnMainMenuClicked();

// End of Main Menu
```

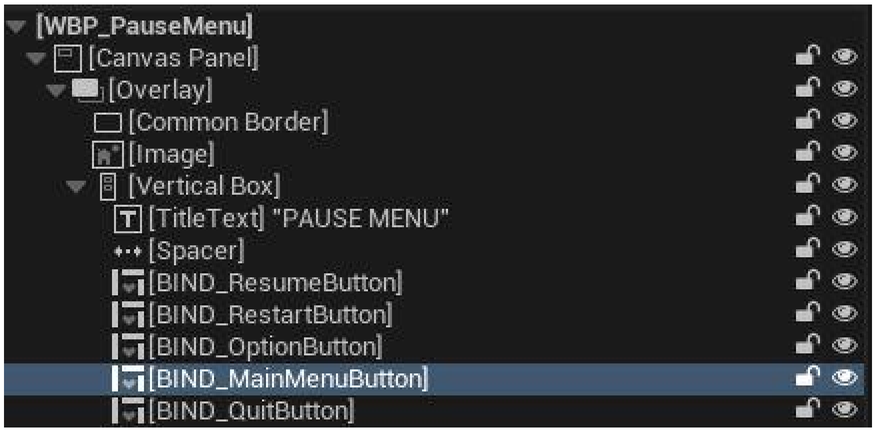
Bind our button.

```
if (BIND_MainMenuButton)
{
    BIND_MainMenuButton->OnButtonClicked.AddUniqueDynamic(this, &UPauseMenuCommonAW::OnMainMenuClicked);
}
```

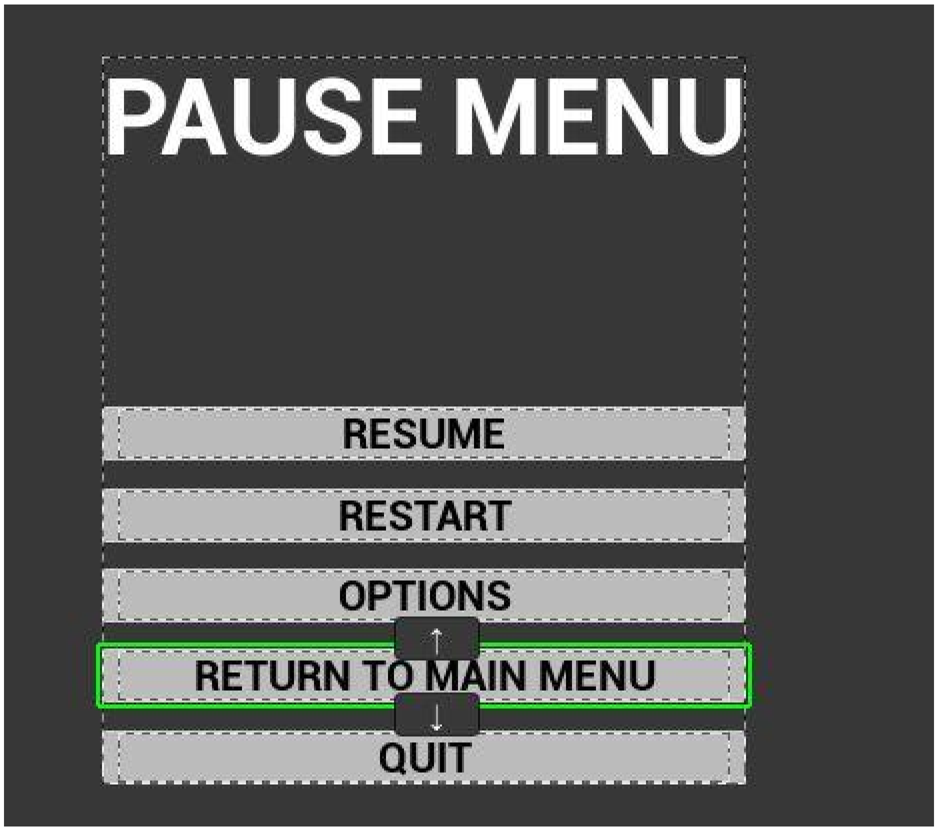
And open the Main Menu.

```
void UPauseMenuCommonActivatableWidget::MainMenu()
{
    UGameplayStatics::OpenLevel((UObject*)PlayerController->GetWorld(), MainMenuMapName);
}
```

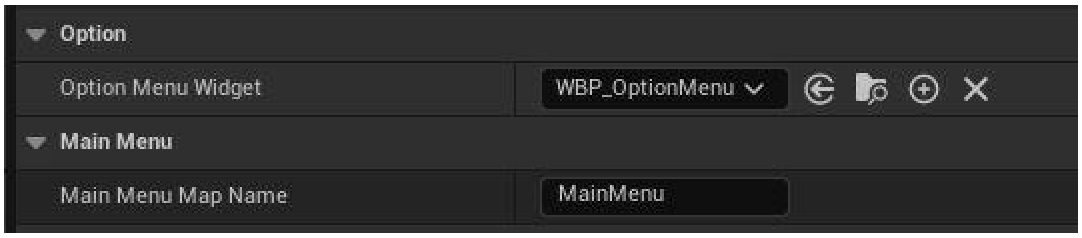
Let's add it in our Widget.



```
[WBP_PauseMenu]
  [Canvas Panel]
    [Overlay]
      [Common Border]
      [Image]
      [Vertical Box]
        [TitleText] "PAUSE MENU"
        [Spacer]
        [BIND_ResumeButton]
        [BIND_RestartButton]
        [BIND_OptionButton]
        [BIND_MainMenuButton]
        [BIND_QuitButton]
```



And finally, add the name of this map in the data.


Option
Option Menu Widget: WBP_OptionMenu
Main Menu
Main Menu Map Name: MainMenu

You can now try to launch the **Main Menu** and you should be able to navigate between the **Main World** and the **Main Menu** and update the **Options** in both maps.

Create Menu Keybindings

Now that our menus are ready, we will improve the Option menu a little.

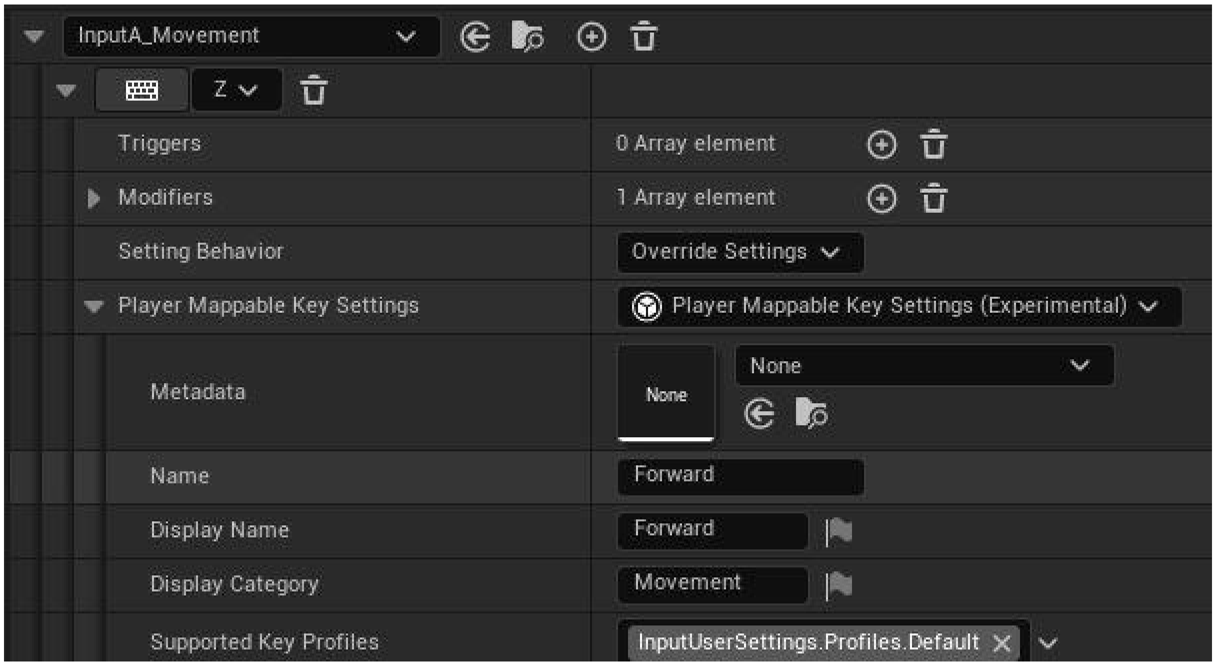
We will add two new inputs to rebind, **Take** and **Throw Input**. All bindable inputs will have a **Display Category** assigned to them. We'll be able to use it in code in order to sort them (we

can put the current inputs in the **Movement** category and the two new ones in an **Action** category).
We'll be able to get it with **FEnhancedActionKeyMapping::GetDisplayCategory**.

We will create a new **Option Menu** for **Keybindings** via **Common Activatable Widget**. Inside it, we will display the bindable keys by Category.

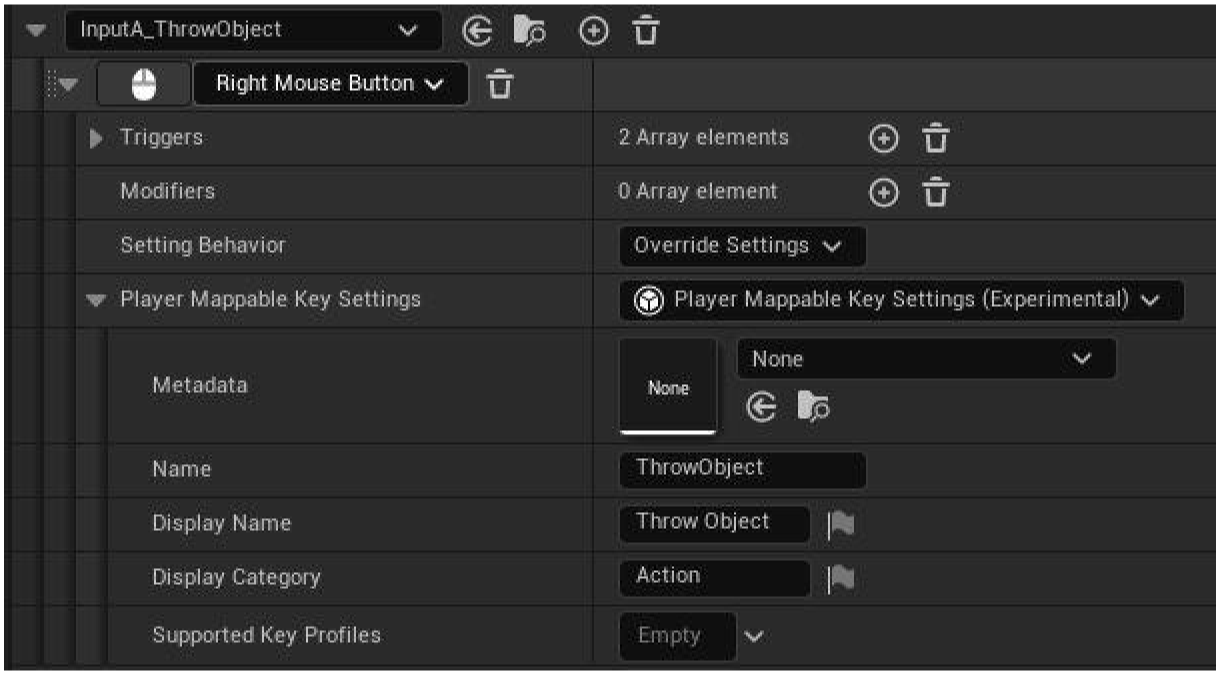
Prepare the Inputs

To start, I add the category **Movement** to all the movements inputs (Z, Q, S and D).



InputA_Movement
Z
Triggers: 0 Array element
Modifiers: 1 Array element
Setting Behavior: Override Settings
Player Mappable Key Settings: Player Mappable Key Settings (Experimental)
Metadata: None
Name: Forward
Display Name: Forward
Display Category: Movement
Supported Key Profiles: InputUserSettings.Profiles.Default

Then the category **Action** to the mouse inputs **IA_TakeObject** and **IA_ThrowObject**.


InputA_ThrowObject

Right Mouse Button

Triggers: 2 Array elements

Modifiers: 0 Array element

Setting Behavior: Override Settings

Player Mappable Key Settings: Player Mappable Key Settings (Experimental)

Metadata: None

Name: ThrowObject

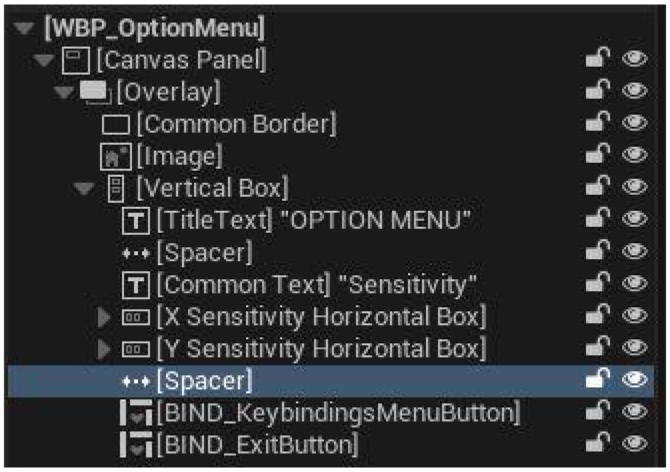
Display Name: Throw Object

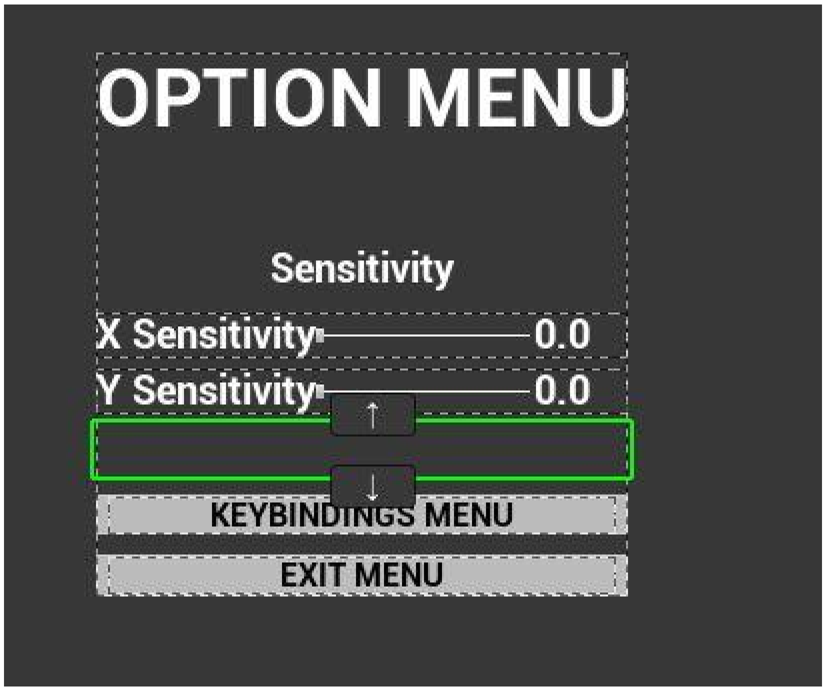
Display Category: Action

Supported Key Profiles: Empty

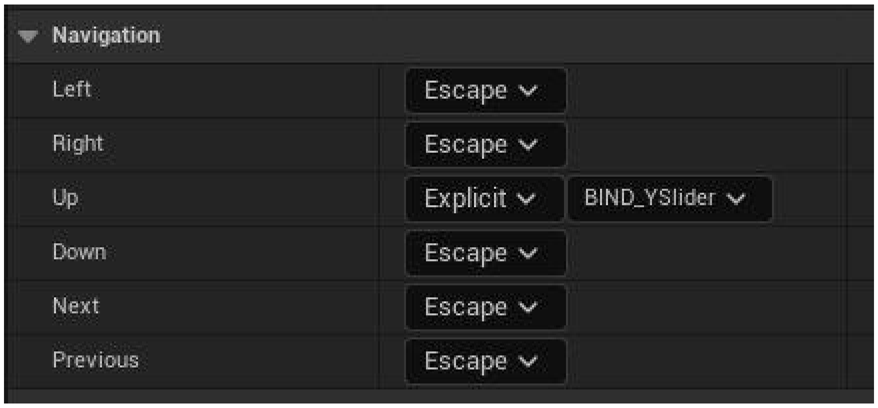
Update Existing Option Menu

Open the **WBP_OptionMenu** and remove the **Vertical Box** for keys. Then, create a new button for the new **Keybinding Menu** and name it **BIND_KeybindingsMenuButton.**


```
[WBP_OptionMenu]
  [Canvas Panel]
    [Overlay]
      [Common Border]
      [Image]
      [Vertical Box]
        [TitleText] "OPTION MENU"
        [Spacer]
        [Common Text] "Sensitivity"
        [X Sensitivity Horizontal Box]
        [Y Sensitivity Horizontal Box]
        [Spacer]
        [BIND_KeybindingsMenuButton]
        [BIND_ExitButton]
```

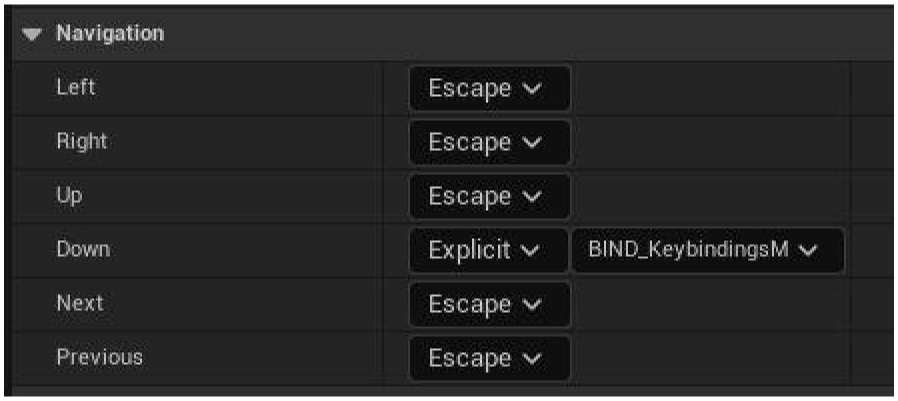


For the gamepad navigation to work properly, it is necessary to ignore the **Spacer** between the **Keybindings** button and the **Y Slider**. To do this, we will modify their **Navigation** settings.
In these parameters, it is possible to assign a direction from one element of the UI to a function or explicitly to another element of the UI. That’s what we’re going to do here.
For the **Button**, we will pass **Up** en **Explicit** and we will indicate the **Y Slider.**
And vice versa for the **Y Slider**, we will switch the **Button** on **Down**.
You can check the other Navigation type that does exist, it might be useful to know the options that do exist.



Navigation

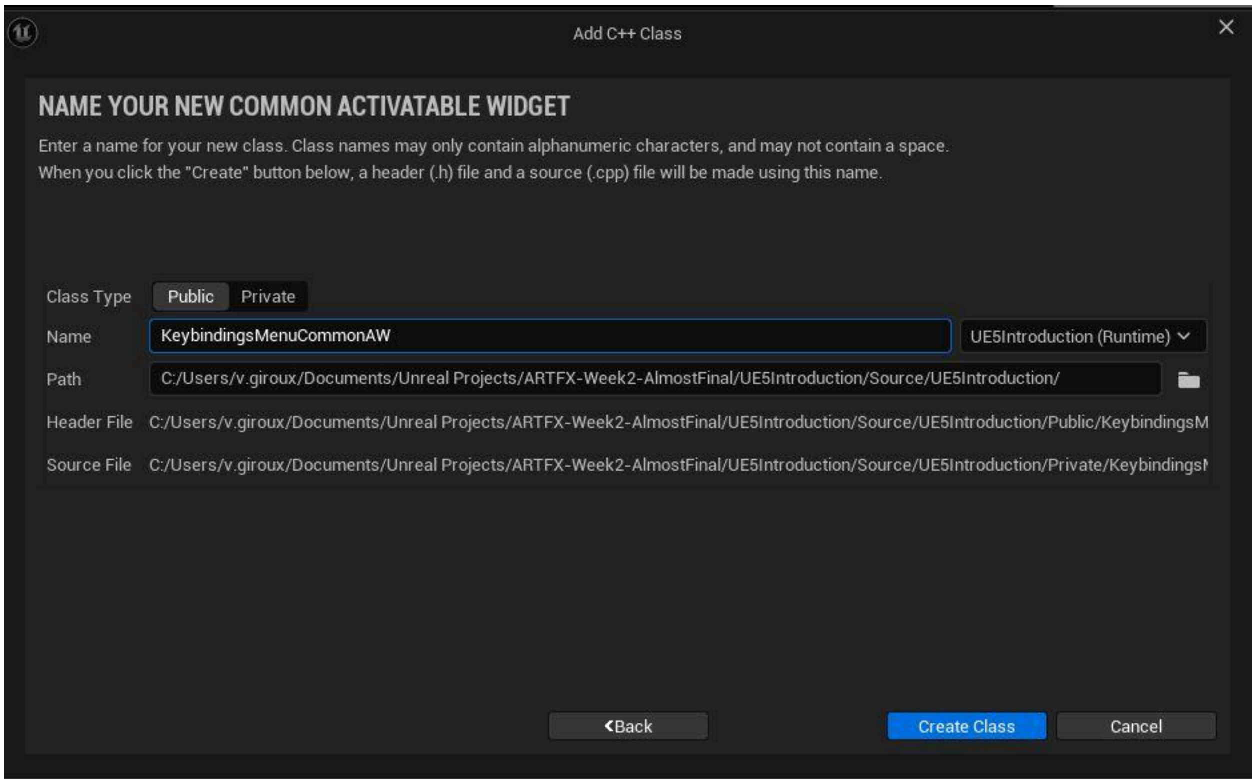
Left	Escape	
Right	Escape	
Up	Explicit	BIND_YSlider
Down	Escape	
Next	Escape	
Previous	Escape	


Navigation

Left	Escape	
Right	Escape	
Up	Escape	
Down	Explicit	BIND_KeybindingsM
Next	Escape	
Previous	Escape	

Create new Keybinding Menu

Let's create the new menu. Create a **KeybindingsMenuCommonAW** from **Common Activatable Widget** and put it in the **UI folder**.

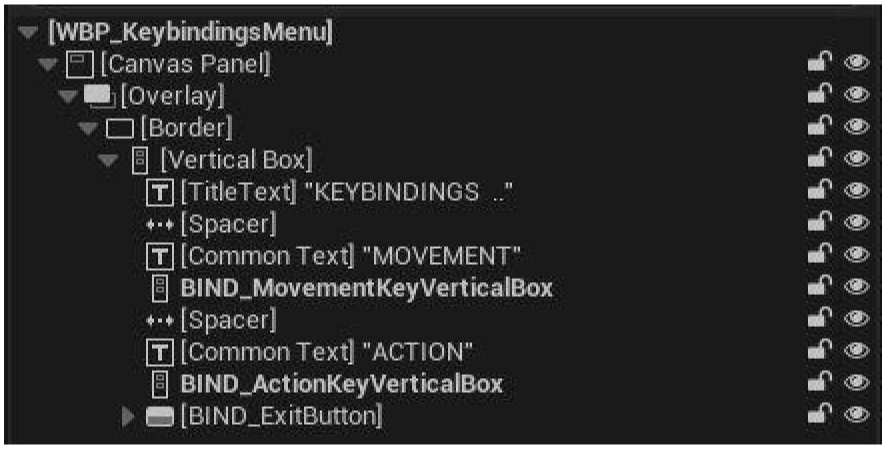

Add C++ Class

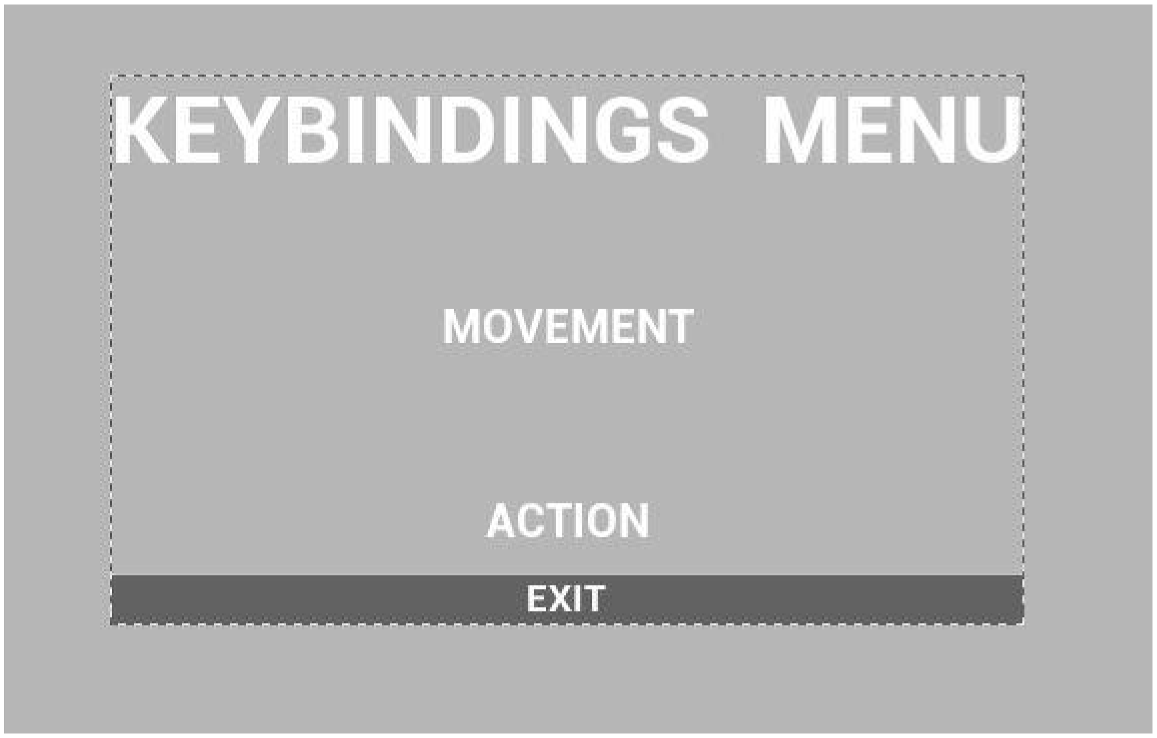
NAME YOUR NEW COMMON ACTIVATABLE WIDGET

Enter a name for your new class. Class names may only contain alphanumeric characters, and may not contain a space.
When you click the "Create" button below, a header (.h) file and a source (.cpp) file will be made using this name.

Class Type	Public / Private
Name	KeybindingsMenuCommonAW — UE5Introduction (Runtime)
Path	C:/Users/v.giroux/Documents/Unreal Projects/ARTFX-Week2-AlmostFinal/UE5Introduction/Source/UE5Introduction/
Header File	C:/Users/v.giroux/Documents/Unreal Projects/ARTFX-Week2-AlmostFinal/UE5Introduction/Source/UE5Introduction/Public/KeybindingsM
Source File	C:/Users/v.giroux/Documents/Unreal Projects/ARTFX-Week2-AlmostFinal/UE5Introduction/Source/UE5Introduction/Private/Keybindings

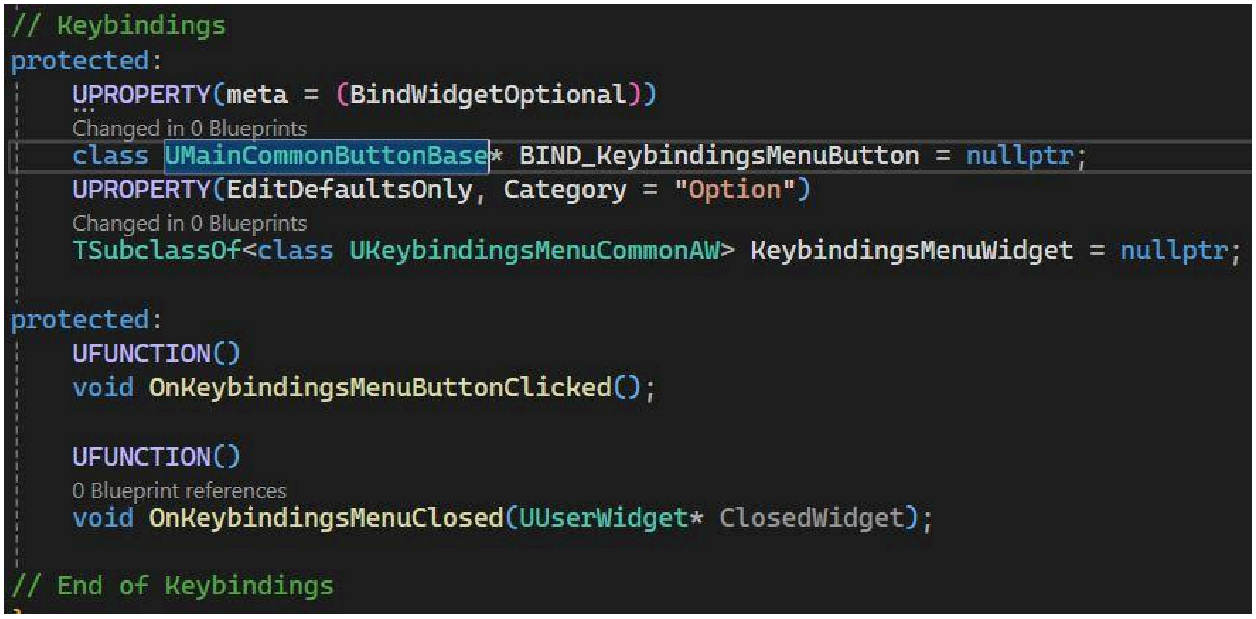
We will create the UI first. Create a new BP from this class and name it **WBP_KeybindingsMenu**.
Inside, we'll add **Vertical Box** to bind and an **Exit button**. Name them respectively **BIND_MovementKeyVerticalBox, BIND_ActionKeyVerticalBox** and **BIND_ExitButton**.


[WBP_KeybindingsMenu]
[Canvas Panel]
[Overlay]
[Border]
[Vertical Box]
[TitleText] "KEYBINDINGS .."
[Spacer]
[Common Text] "MOVEMENT"
BIND_MovementKeyVerticalBox
[Spacer]
[Common Text] "ACTION"
BIND_ActionKeyVerticalBox
[BIND_ExitButton]



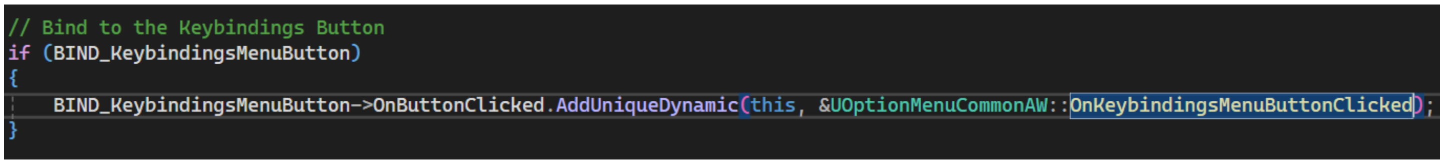
Update Option Menu Code

Before implementing the code for the **Keybinding Menu**, let's update the code in the **Option Menu**.
First, remove everything that was in the **Keybindings Section**, we'll not need it anymore.
Since we will open the **Keybindings Menu** from here, we'll have a reference to the widget.
Then we'll have a **Button** and its function to open it. In addition to that, we'll also add a function called when the menu will be closed



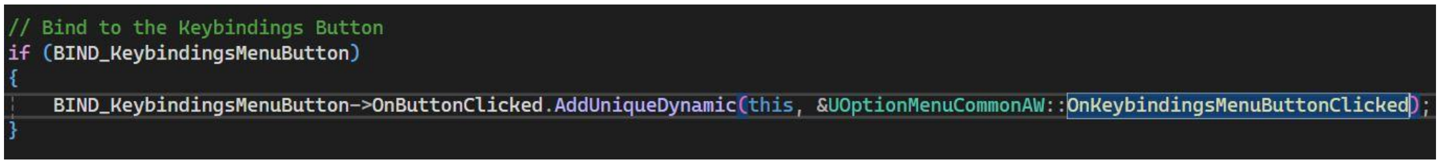
```
// Keybindings
protected:
    UPROPERTY(meta = (BindWidgetOptional))
    Changed in 0 Blueprints
    class UMainCommonButtonBase* BIND_KeybindingsMenuButton = nullptr;
    UPROPERTY(EditDefaultsOnly, Category = "Option")
    Changed in 0 Blueprints
    TSubclassOf<class UKeybindingsMenuCommonAW> KeybindingsMenuWidget = nullptr;

protected:
    UFUNCTION()
    void OnKeybindingsMenuButtonClicked();

    UFUNCTION()
    0 Blueprint references
    void OnKeybindingsMenuClosed(UUserWidget* ClosedWidget);

// End of Keybindings
```

Start by binding yourself to the new Button in the **Native Construct**.

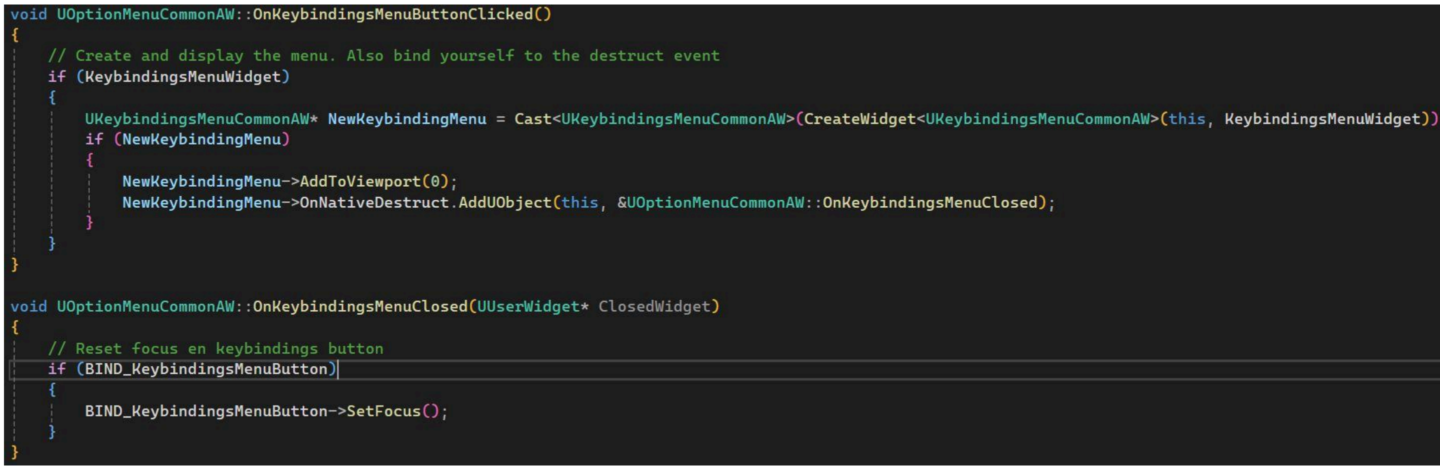


```
// Bind to the Keybindings Button
if (BIND_KeybindingsMenuButton)
{
    BIND_KeybindingsMenuButton->OnButtonClicked.AddUniqueDynamic(this, &UOptionMenuCommonAW::OnKeybindingsMenuButtonClicked);
}
```

Then, open the menu in **OnKeybindingsMenuButtonClicked** and bind yourself to the **Destruct** event. After that set back the focus to this same button when using the callback **OnKeybindingsMenuClosed**.

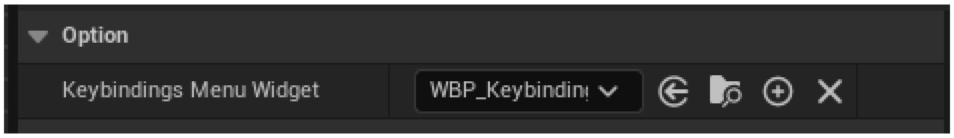


```
// Bind to the Keybindings Button
if (BIND_KeybindingsMenuButton)
{
    BIND_KeybindingsMenuButton->OnButtonClicked.AddUniqueDynamic(this, &UOptionMenuCommonAW::OnKeybindingsMenuButtonClicked);
}
```



```
void UOptionMenuCommonAW::OnKeybindingsMenuButtonClicked()
{
    // Create and display the menu. Also bind yourself to the destruct event
    if (KeybindingsMenuWidget)
    {
        UKeybindingsMenuCommonAW* NewKeybindingMenu = Cast<UKeybindingsMenuCommonAW>(CreateWidget<UKeybindingsMenuCommonAW>(this, KeybindingsMenuWidget))
        if (NewKeybindingMenu)
        {
            NewKeybindingMenu->AddToViewport(0);
            NewKeybindingMenu->OnNativeDestruct.AddUObject(this, &UOptionMenuCommonAW::OnKeybindingsMenuClosed);
        }
    }
}

void UOptionMenuCommonAW::OnKeybindingsMenuClosed(UUserWidget* ClosedWidget)
{
    // Reset focus en keybindings button
    if (BIND_KeybindingsMenuButton)
    {
        BIND_KeybindingsMenuButton->SetFocus();
    }
}
```

Let’s quickly add the reference to the **Keybinding Menu** and we’re done !

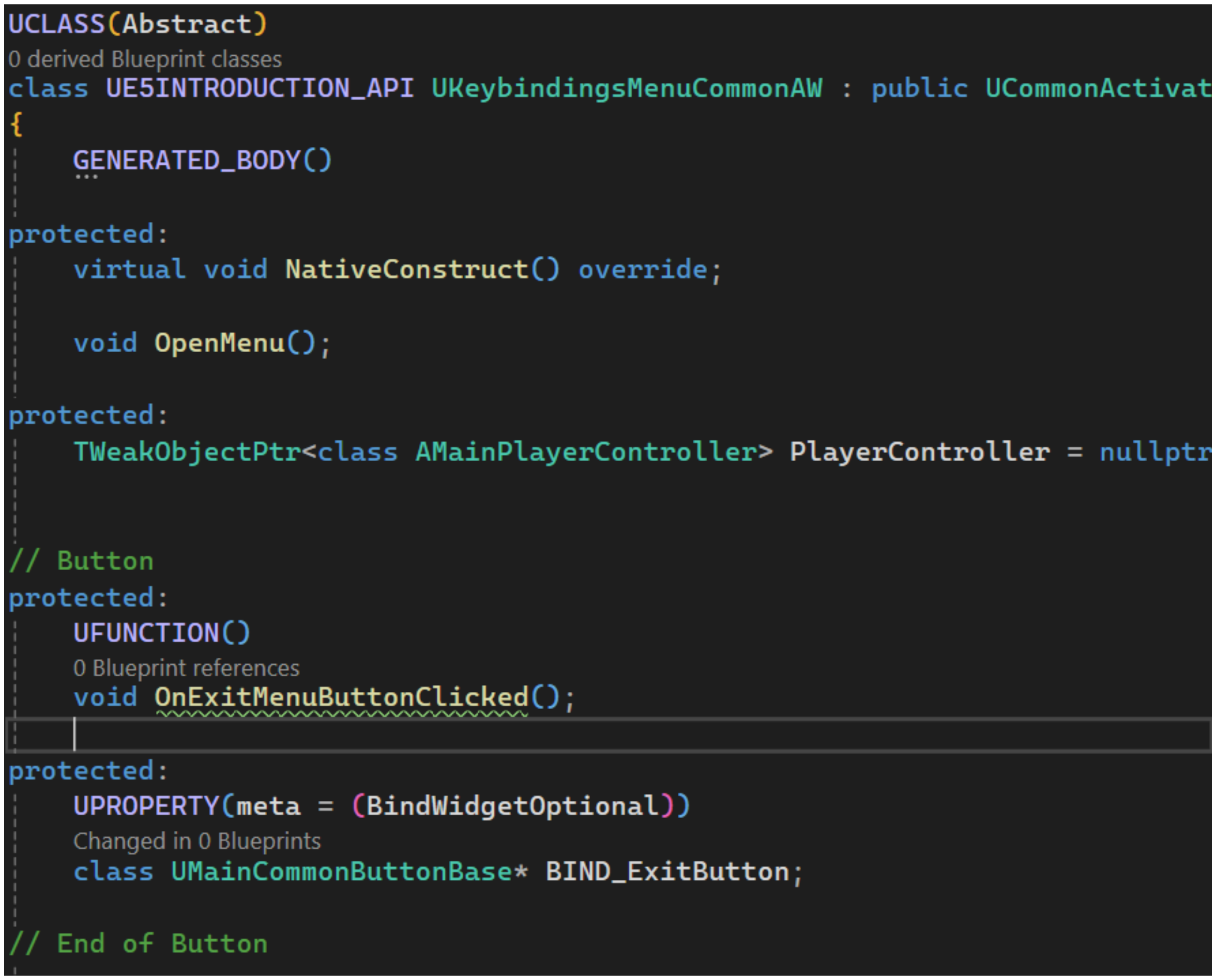


Implement the code for the Keybindings Menu

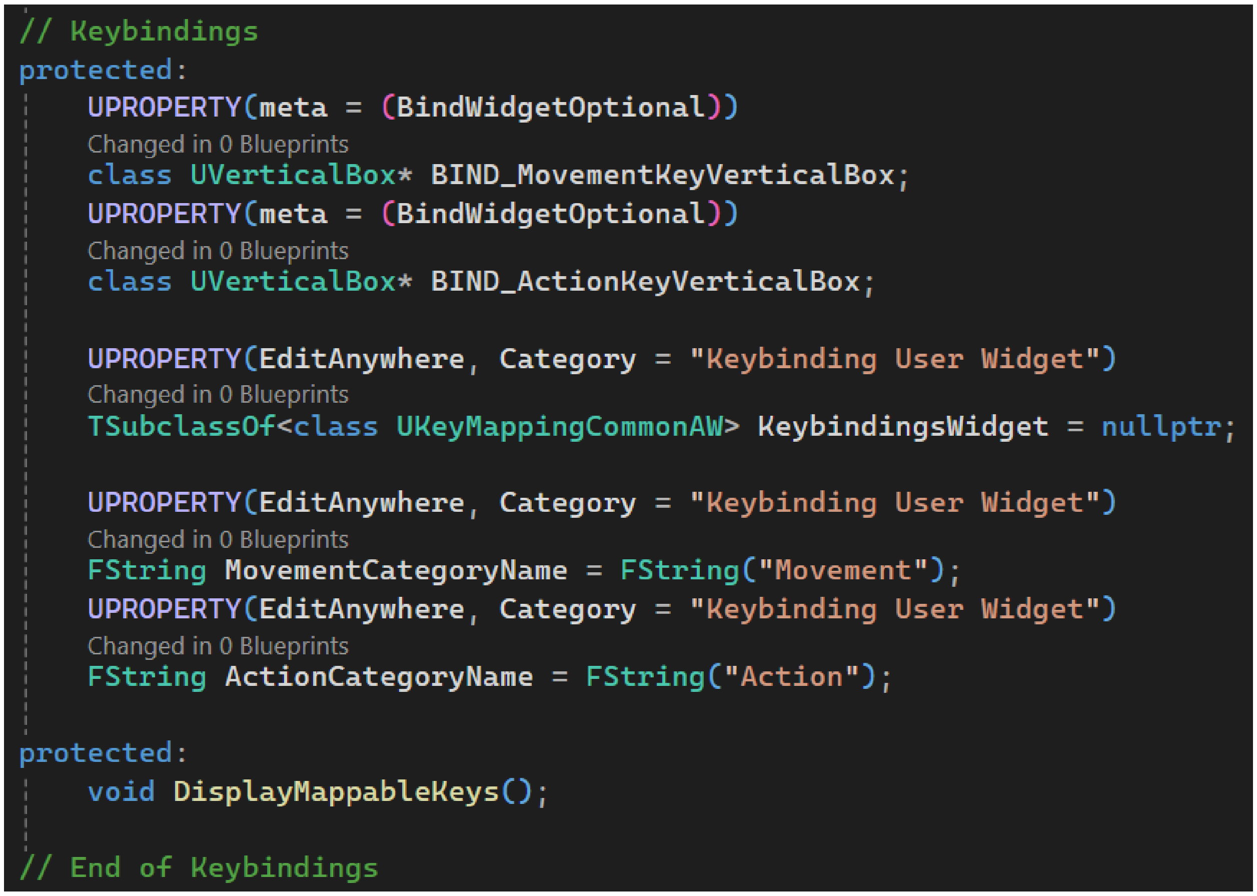
Now that everything is set in the **Option Menu**, let’s implement the code for the **Keybindings Menu**.

The class will be **Abstract** as usual.

We will add in our class a pointer to the **Player Controller**, as well as a reference to the **Exit Button.** We’ll add the two **Vertical Box** with and an **FString** to indicate their Category and a function to display everything.
The reference to the **Key Mapping widge**t will also be required.

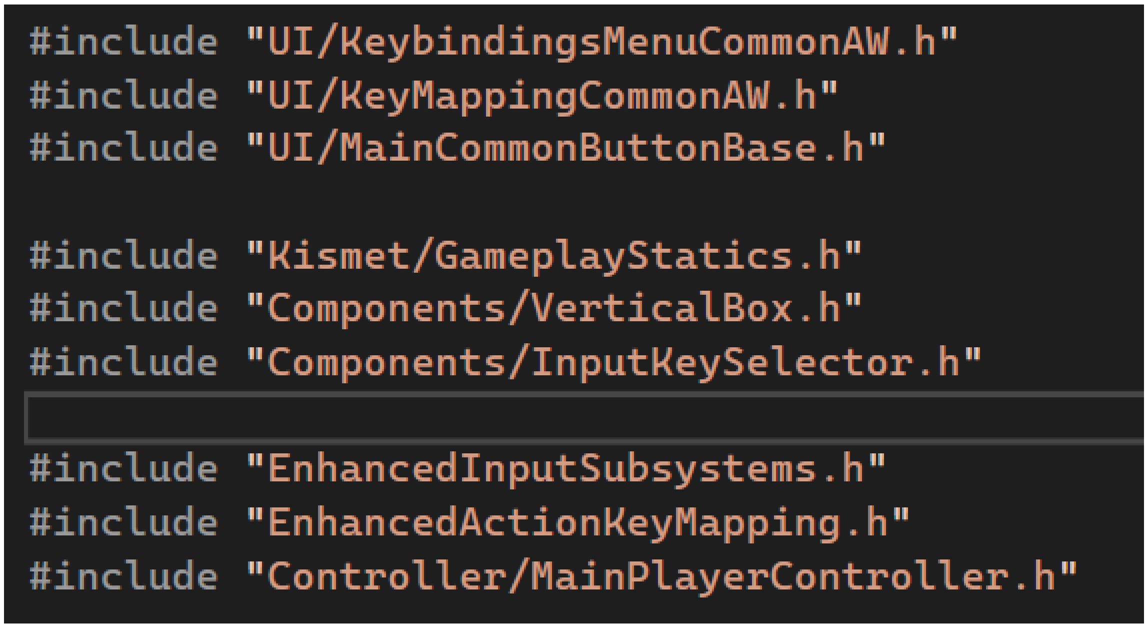


```
UCLASS(Abstract)
0 derived Blueprint classes
class UE5INTRODUCTION_API UKeybindingsMenuCommonAW : public UCommonActivat
{
    GENERATED_BODY()

protected:
    virtual void NativeConstruct() override;

    void OpenMenu();

protected:
    TWeakObjectPtr<class AMainPlayerController> PlayerController = nullptr


// Button
protected:
    UFUNCTION()
    0 Blueprint references
    void OnExitMenuButtonClicked();

protected:
    UPROPERTY(meta = (BindWidgetOptional))
    Changed in 0 Blueprints
    class UMainCommonButtonBase* BIND_ExitButton;

// End of Button
```

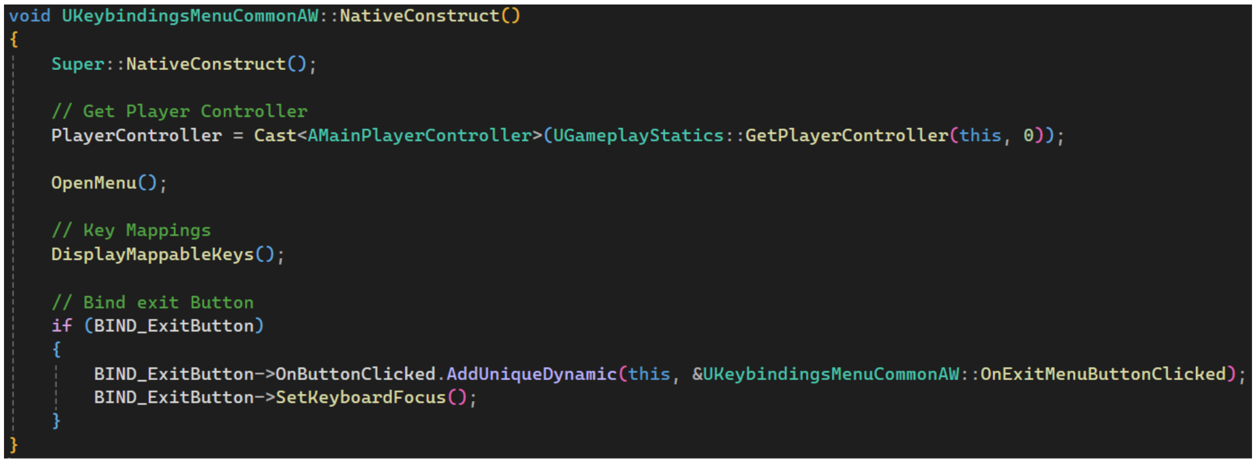


```
// Keybindings
protected:
    UPROPERTY(meta = (BindWidgetOptional))
    Changed in 0 Blueprints
    class UVerticalBox* BIND_MovementKeyVerticalBox;
    UPROPERTY(meta = (BindWidgetOptional))
    Changed in 0 Blueprints
    class UVerticalBox* BIND_ActionKeyVerticalBox;

    UPROPERTY(EditAnywhere, Category = "Keybinding User Widget")
    Changed in 0 Blueprints
    TSubclassOf<class UKeyMappingCommonAW> KeybindingsWidget = nullptr;

    UPROPERTY(EditAnywhere, Category = "Keybinding User Widget")
    Changed in 0 Blueprints
    FString MovementCategoryName = FString("Movement");
    UPROPERTY(EditAnywhere, Category = "Keybinding User Widget")
    Changed in 0 Blueprints
    FString ActionCategoryName = FString("Action");

protected:
    void DisplayMappableKeys();

// End of Keybindings
```

For the **Construct**, we will get the controller and call the function to display the keys, then bind the button.

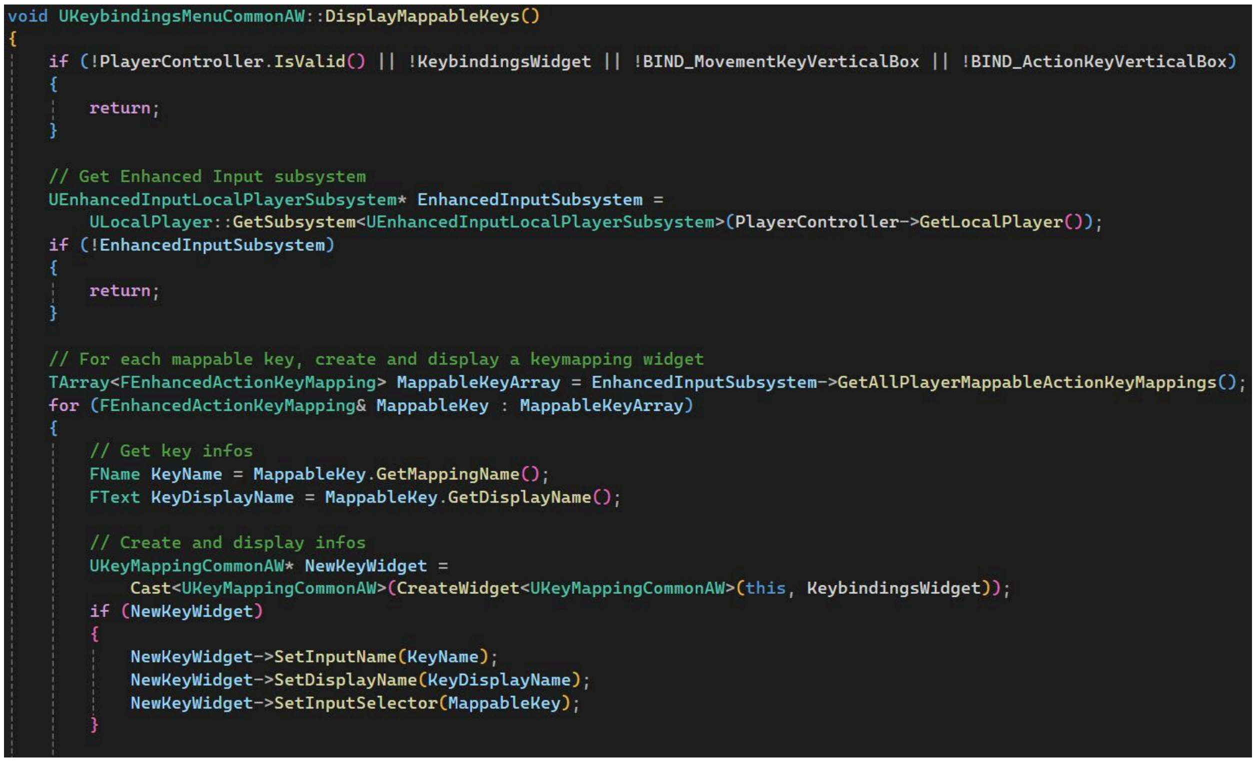


```
#include "UI/KeybindingsMenuCommonAW.h"
#include "UI/KeyMappingCommonAW.h"
#include "UI/MainCommonButtonBase.h"

#include "Kismet/GameplayStatics.h"
#include "Components/VerticalBox.h"
#include "Components/InputKeySelector.h"

#include "EnhancedInputSubsystems.h"
#include "EnhancedActionKeyMapping.h"
#include "Controller/MainPlayerController.h"
```

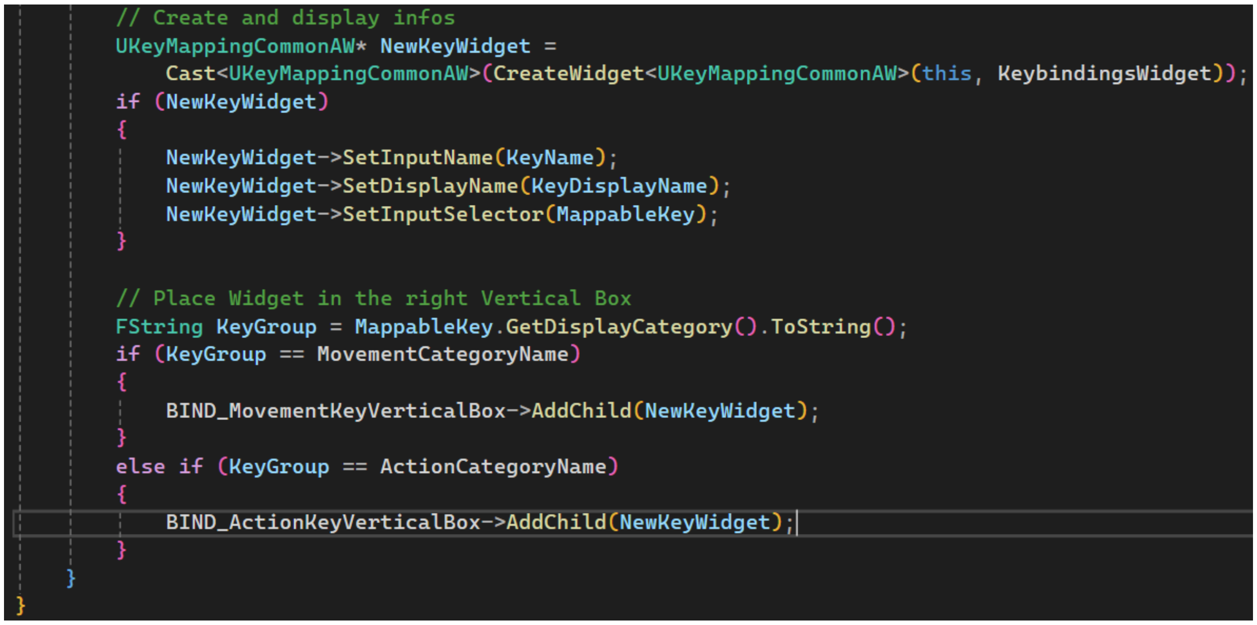


```cpp
void UKeybindingsMenuCommonAW::NativeConstruct()
{
    Super::NativeConstruct();

    // Get Player Controller
    PlayerController = Cast<AMainPlayerController>(UGameplayStatics::GetPlayerController(this, 0));

    OpenMenu();

    // Key Mappings
    DisplayMappableKeys();

    // Bind exit Button
    if (BIND_ExitButton)
    {
        BIND_ExitButton->OnButtonClicked.AddUniqueDynamic(this, &UKeybindingsMenuCommonAW::OnExitMenuButtonClicked);
        BIND_ExitButton->SetKeyboardFocus();
    }
}
```

For the **Display Mappable Keys**, we can use the previous code that we had in the **Option Menu** and just update the end by checking the Category.

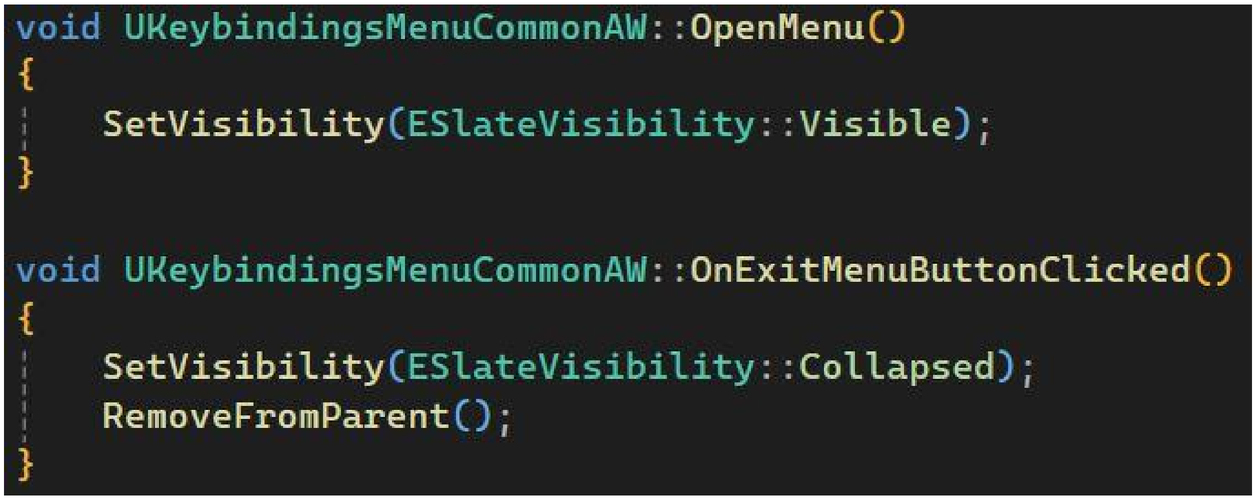


```cpp
void UKeybindingsMenuCommonAW::DisplayMappableKeys()
{
    if (!PlayerController.IsValid() || !KeybindingsWidget || !BIND_MovementKeyVerticalBox || !BIND_ActionKeyVerticalBox)
    {
        return;
    }

    // Get Enhanced Input subsystem
    UEnhancedInputLocalPlayerSubsystem* EnhancedInputSubsystem =
        ULocalPlayer::GetSubsystem<UEnhancedInputLocalPlayerSubsystem>(PlayerController->GetLocalPlayer());
    if (!EnhancedInputSubsystem)
    {
        return;
    }

    // For each mappable key, create and display a keymapping widget
    TArray<FEnhancedActionKeyMapping> MappableKeyArray = EnhancedInputSubsystem->GetAllPlayerMappableActionKeyMappings();
    for (FEnhancedActionKeyMapping& MappableKey : MappableKeyArray)
    {
        // Get key infos
        FName KeyName = MappableKey.GetMappingName();
        FText KeyDisplayName = MappableKey.GetDisplayName();

        // Create and display infos
        UKeyMappingCommonAW* NewKeyWidget =
            Cast<UKeyMappingCommonAW>(CreateWidget<UKeyMappingCommonAW>(this, KeybindingsWidget));
        if (NewKeyWidget)
        {
            NewKeyWidget->SetInputName(KeyName);
            NewKeyWidget->SetDisplayName(KeyDisplayName);
            NewKeyWidget->SetInputSelector(MappableKey);
        }
```



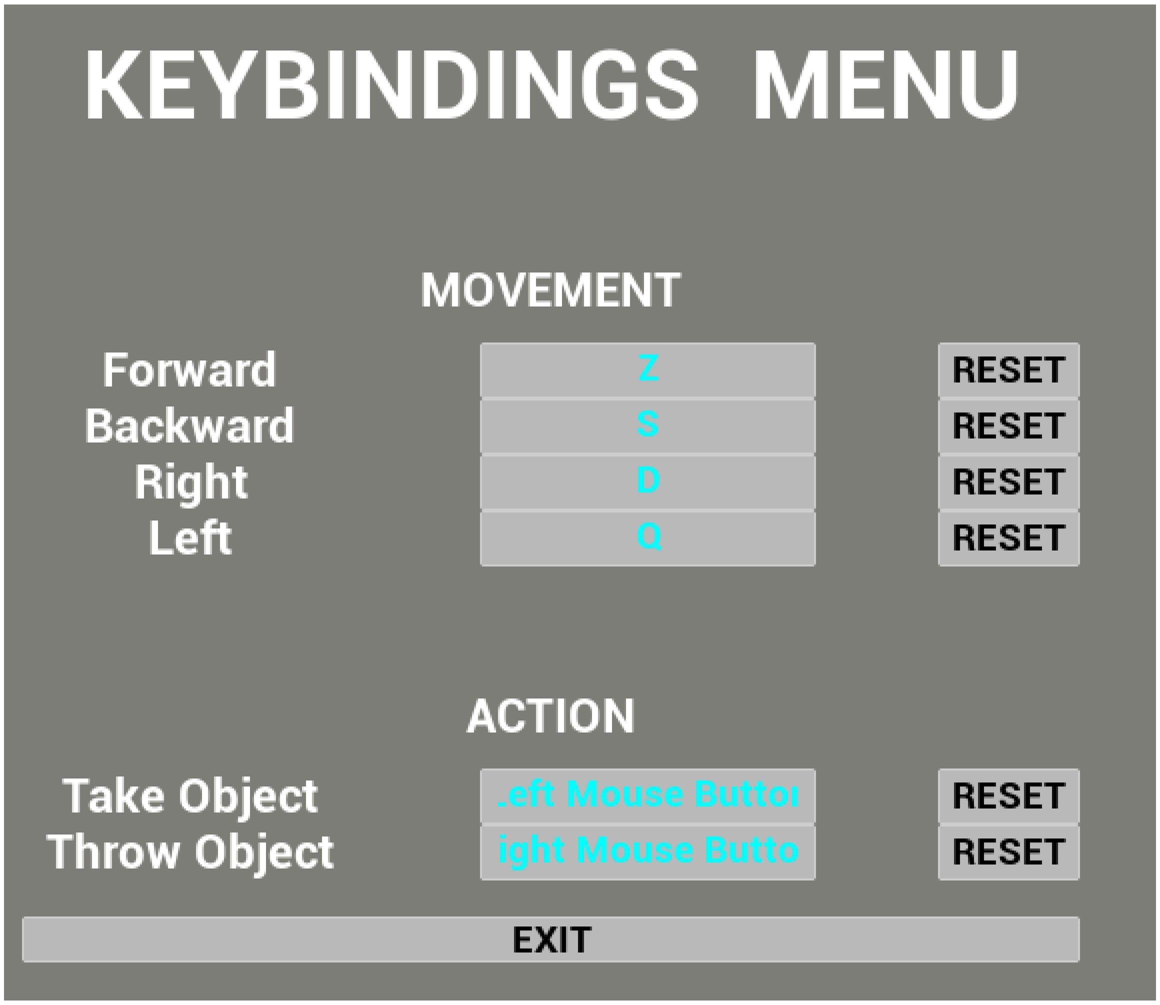
```
        // Create and display infos
        UKeyMappingCommonAW* NewKeyWidget =
            Cast<UKeyMappingCommonAW>(CreateWidget<UKeyMappingCommonAW>(this, KeybindingsWidget));
        if (NewKeyWidget)
        {
            NewKeyWidget->SetInputName(KeyName);
            NewKeyWidget->SetDisplayName(KeyDisplayName);
            NewKeyWidget->SetInputSelector(MappableKey);
        }

        // Place Widget in the right Vertical Box
        FString KeyGroup = MappableKey.GetDisplayCategory().ToString();
        if (KeyGroup == MovementCategoryName)
        {
            BIND_MovementKeyVerticalBox->AddChild(NewKeyWidget);
        }
        else if (KeyGroup == ActionCategoryName)
        {
            BIND_ActionKeyVerticalBox->AddChild(NewKeyWidget);
        }
    }
}
```

The **Open** and **Exit** functions are as usual.



```
void UKeybindingsMenuCommonAW::OpenMenu()
{
    SetVisibility(ESlateVisibility::Visible);
}

void UKeybindingsMenuCommonAW::OnExitMenuButtonClicked()
{
    SetVisibility(ESlateVisibility::Collapsed);
    RemoveFromParent();
}
```

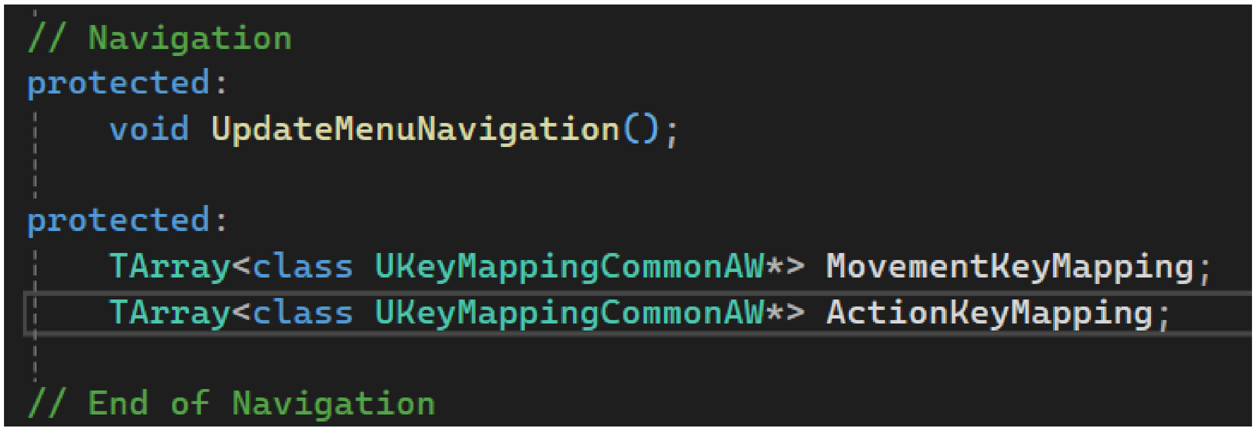
Now you can try and should see your inputs in this new menu.



Fix Gamepad Movement Issues

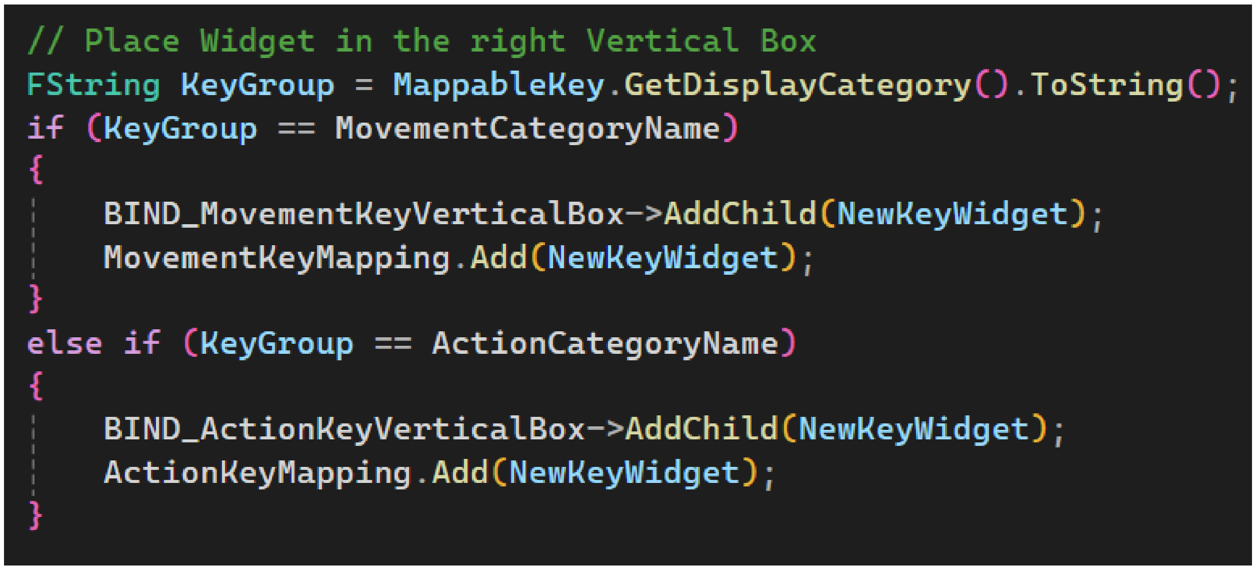
This is almost good, but there is still a problem with the navigation with a gamepad.
If you try to navigate in the **Menu**, you'll see that between the **Take Object** and the **Left** key, you'll focus the **Spacer.** In order to ignore it we need to override the **Navigation** data, like we did in the **Option Menu**.
Since the **Key Mapping** widgets are created in code, it's not possible to bind it in the editor, we must do it in the code.

In order to achieve this, let's start with modifying the **Key Binding Menu**. For that, we'll create a new function and we'll also need an Array that will contain the Widget's reference for each section.



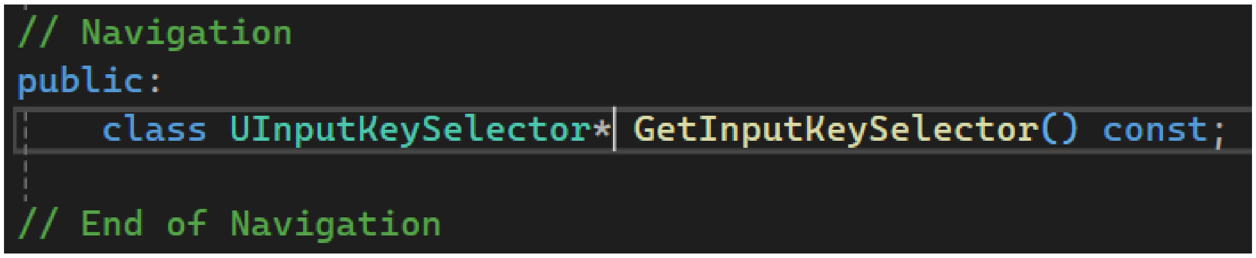
```
// Navigation
protected:
    void UpdateMenuNavigation();

protected:
    TArray<class UKeyMappingCommonAW*> MovementKeyMapping;
    TArray<class UKeyMappingCommonAW*> ActionKeyMapping;

// End of Navigation
```

Let’s start by filling the arrays in **DisplayMappableKeys**.

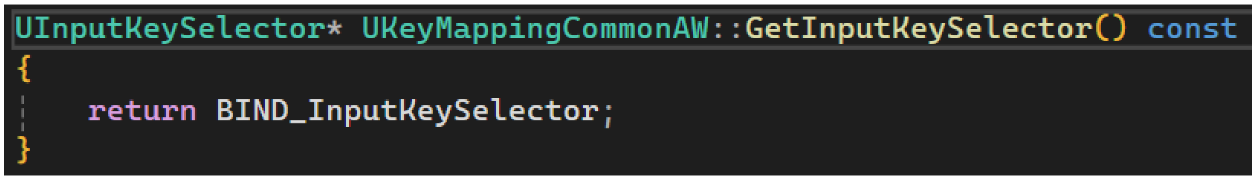


```
// Place Widget in the right Vertical Box
FString KeyGroup = MappableKey.GetDisplayCategory().ToString();
if (KeyGroup == MovementCategoryName)
{
    BIND_MovementKeyVerticalBox->AddChild(NewKeyWidget);
    MovementKeyMapping.Add(NewKeyWidget);
}
else if (KeyGroup == ActionCategoryName)
{
    BIND_ActionKeyVerticalBox->AddChild(NewKeyWidget);
    ActionKeyMapping.Add(NewKeyWidget);
}
```

Before filling the **UpdateMenuNavigation** function, we need to update the **UKeyMappingCommonAW.**
In this function, we’ll link the **Input Selector**, we need to add a Getter for this in the **Key Mapping** class.

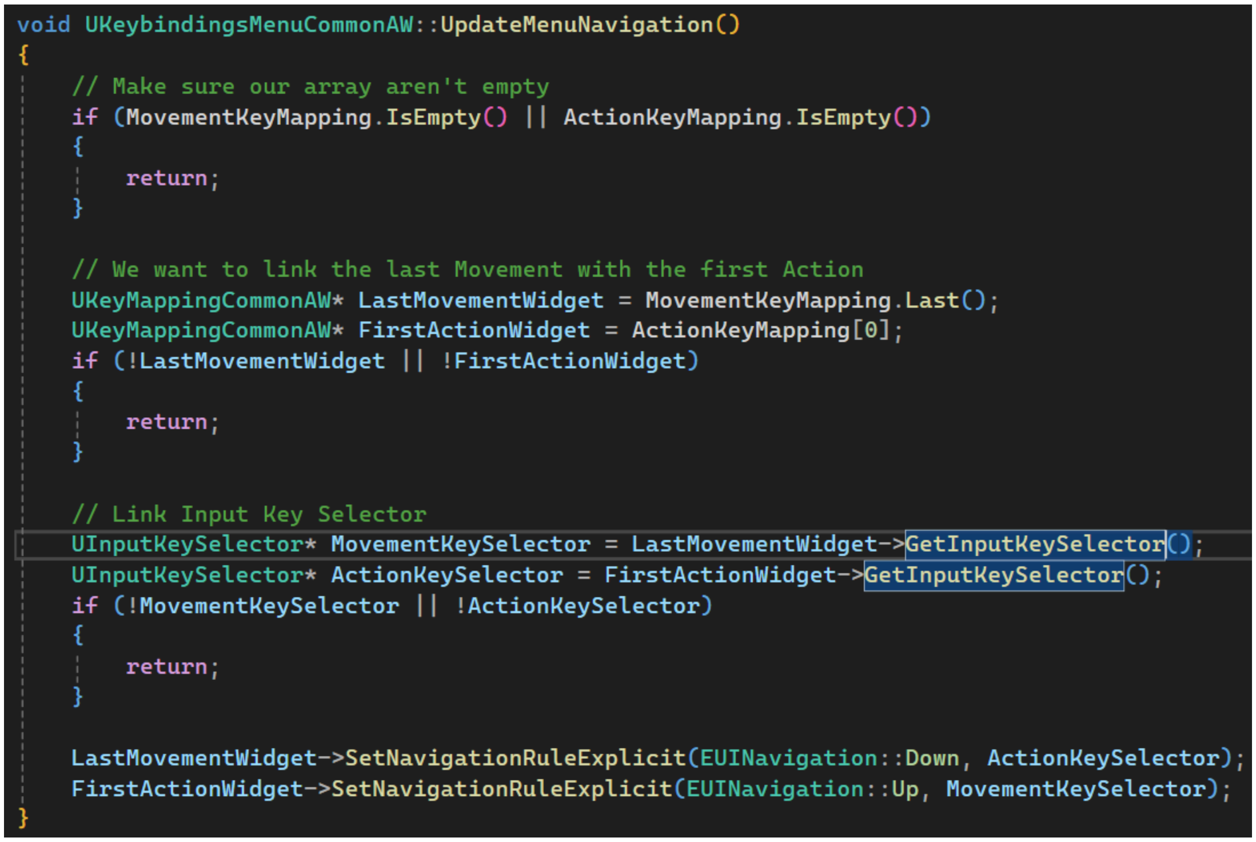


```
// Navigation
public:
    class UInputKeySelector* GetInputKeySelector() const;

// End of Navigation
```



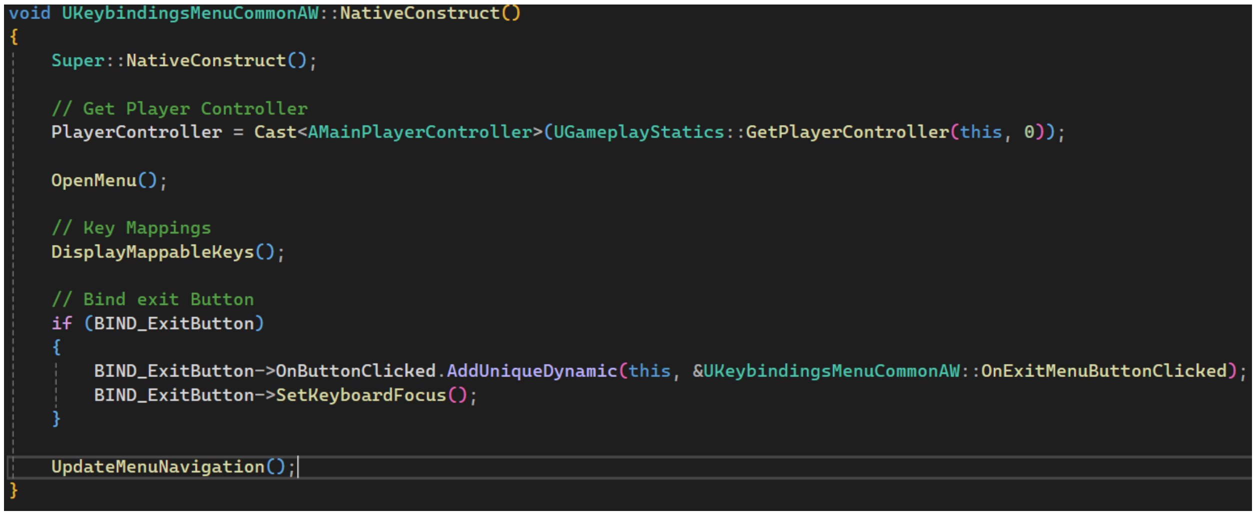
```
UInputKeySelector* UKeyMappingCommonAW::GetInputKeySelector() const
{
    return BIND_InputKeySelector;
}
```

Now let's go back to **UpdateMenuNavigation** in **KeybindingsMenuCommonAW.**
First, we check the array. Then we get the last **Movement Key** and the first **Action Key**.
Finally, we link them with the **SetNavigationRuleExplicit.** This function allows us to use an **Explicit Navigation** between those two elements.



```cpp
void UKeybindingsMenuCommonAW::UpdateMenuNavigation()
{
    // Make sure our array aren't empty
    if (MovementKeyMapping.IsEmpty() || ActionKeyMapping.IsEmpty())
    {
        return;
    }

    // We want to link the last Movement with the first Action
    UKeyMappingCommonAW* LastMovementWidget = MovementKeyMapping.Last();
    UKeyMappingCommonAW* FirstActionWidget = ActionKeyMapping[0];
    if (!LastMovementWidget || !FirstActionWidget)
    {
        return;
    }

    // Link Input Key Selector
    UInputKeySelector* MovementKeySelector = LastMovementWidget->GetInputKeySelector();
    UInputKeySelector* ActionKeySelector = FirstActionWidget->GetInputKeySelector();
    if (!MovementKeySelector || !ActionKeySelector)
    {
        return;
    }

    LastMovementWidget->SetNavigationRuleExplicit(EUINavigation::Down, ActionKeySelector);
    FirstActionWidget->SetNavigationRuleExplicit(EUINavigation::Up, MovementKeySelector);
}
```

Let's call this function at the end of the **Construct.**



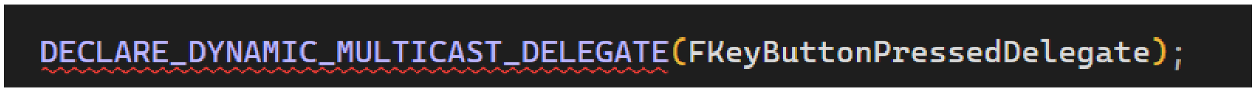
```cpp
void UKeybindingsMenuCommonAW::NativeConstruct()
{
    Super::NativeConstruct();

    // Get Player Controller
    PlayerController = Cast<AMainPlayerController>(UGameplayStatics::GetPlayerController(this, 0));

    OpenMenu();

    // Key Mappings
    DisplayMappableKeys();

    // Bind exit Button
    if (BIND_ExitButton)
    {
        BIND_ExitButton->OnButtonClicked.AddUniqueDynamic(this, &UKeybindingsMenuCommonAW::OnExitMenuButtonClicked);
        BIND_ExitButton->SetKeyboardFocus();
    }

    UpdateMenuNavigation();
}
```

We can now try to navigate through this menu with a gamepad and see that everything works fine !

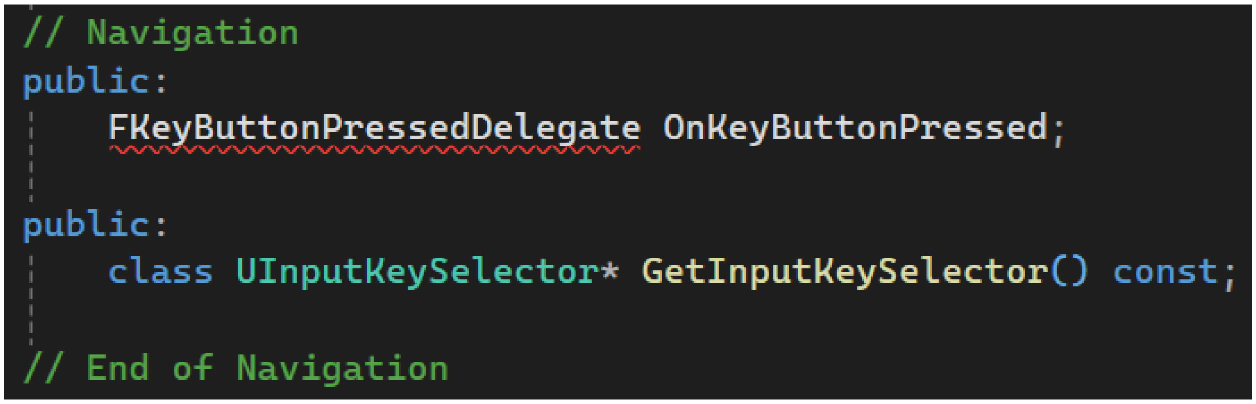
Reset focus in Keybindings Menu

Ok, one last thing.
If you try to rebind a **Key** with a gamepad, you'll see that you'll lose the focus after that. That's a bug in Unreal but there's an easy way to fix it.
We'll simply send an event when a **Key** is rebind and we'll reset the focus to the **Exit Button**.

Go back to the **KeyMappingCommonAW** and add this event.

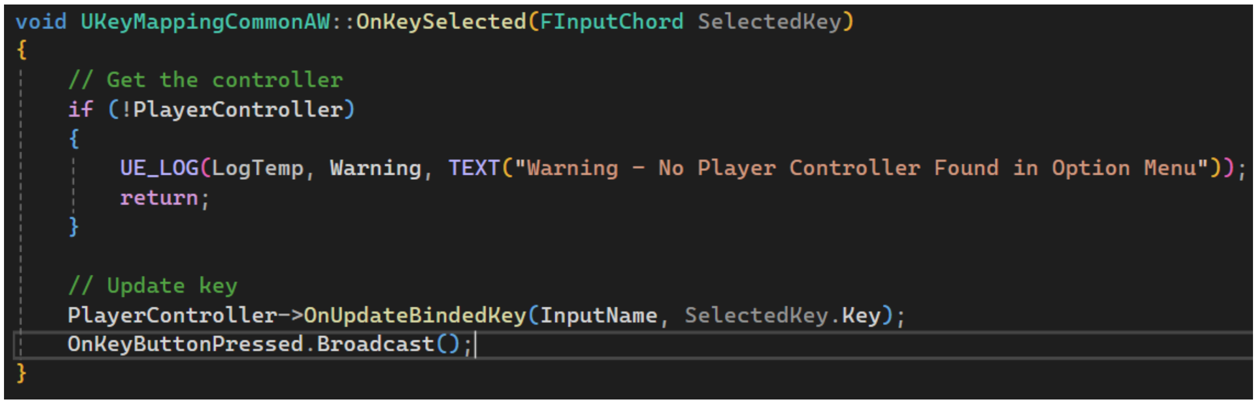


```
DECLARE_DYNAMIC_MULTICAST_DELEGATE(FKeyButtonPressedDelegate);
```



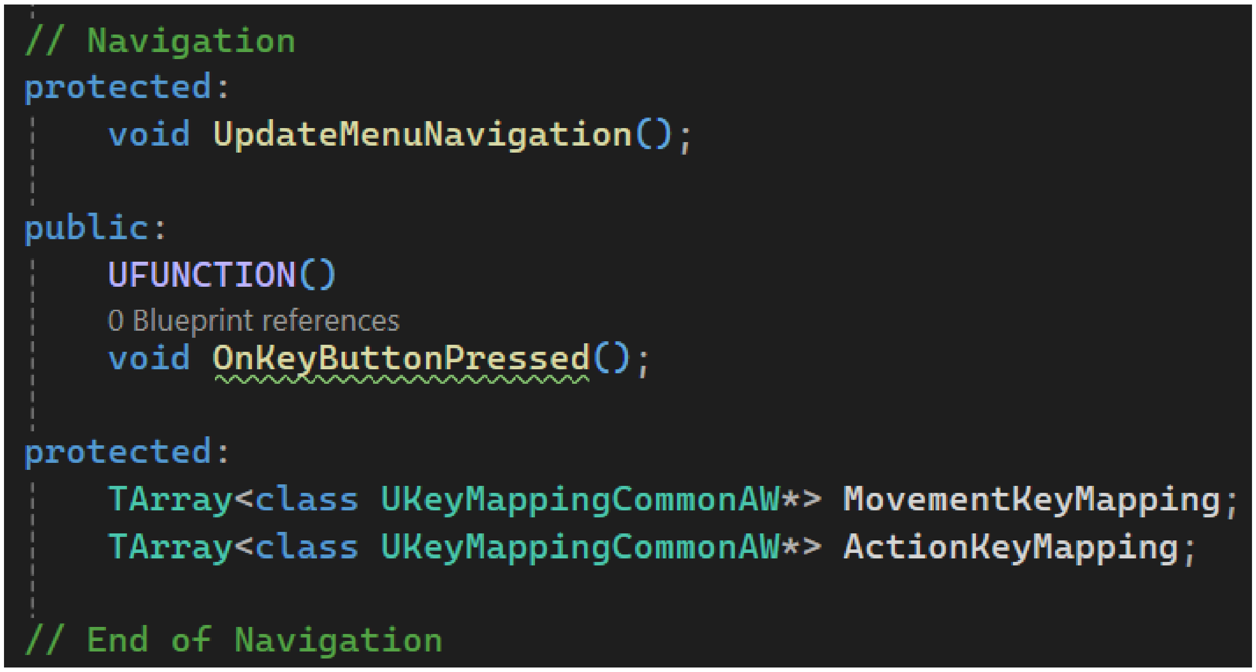
```
// Navigation
public:
    FKeyButtonPressedDelegate OnKeyButtonPressed;

public:
    class UInputKeySelector* GetInputKeySelector() const;

// End of Navigation
```

We'll broadcast it in the **OnKeySelected** function.



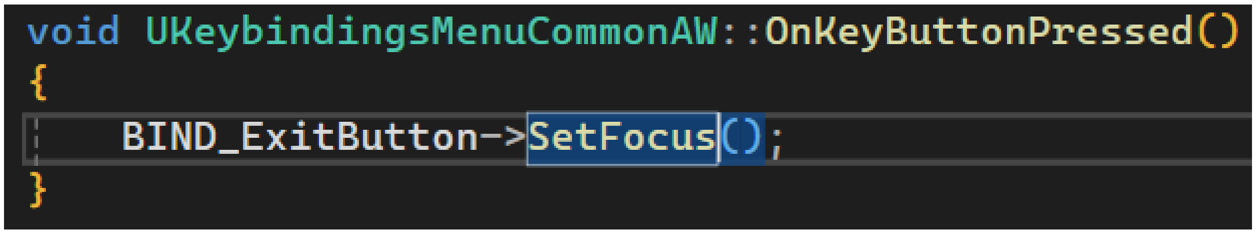
```
void UKeyMappingCommonAW::OnKeySelected(FInputChord SelectedKey)
{
    // Get the controller
    if (!PlayerController)
    {
        UE_LOG(LogTemp, Warning, TEXT("Warning - No Player Controller Found in Option Menu"));
        return;
    }

    // Update key
    PlayerController->OnUpdateBindedKey(InputName, SelectedKey.Key);
    OnKeyButtonPressed.Broadcast();
}
```

Now let's go back to the **KeybindingsMenuCommonAW.**
Let's create a callback function



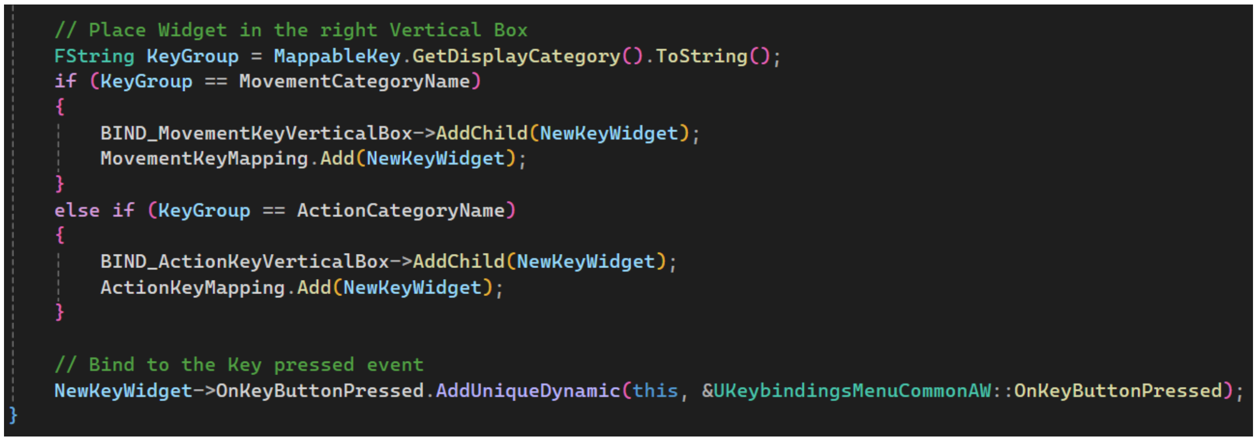
```
// Navigation
protected:
    void UpdateMenuNavigation();

public:
    UFUNCTION()
    0 Blueprint references
    void OnKeyButtonPressed();

protected:
    TArray<class UKeyMappingCommonAW*> MovementKeyMapping;
    TArray<class UKeyMappingCommonAW*> ActionKeyMapping;

// End of Navigation
```

This function will simply reset the focus back to the **Exit Button**.
We could send back a pointer to the **Key Selector** to set back the focus on it if we wanted.



```
void UKeybindingsMenuCommonAW::OnKeyButtonPressed()
{
    BIND_ExitButton->SetFocus();
}
```

Now we can simply bind ourselves at the end of the **DisplayMappableKeys** function.



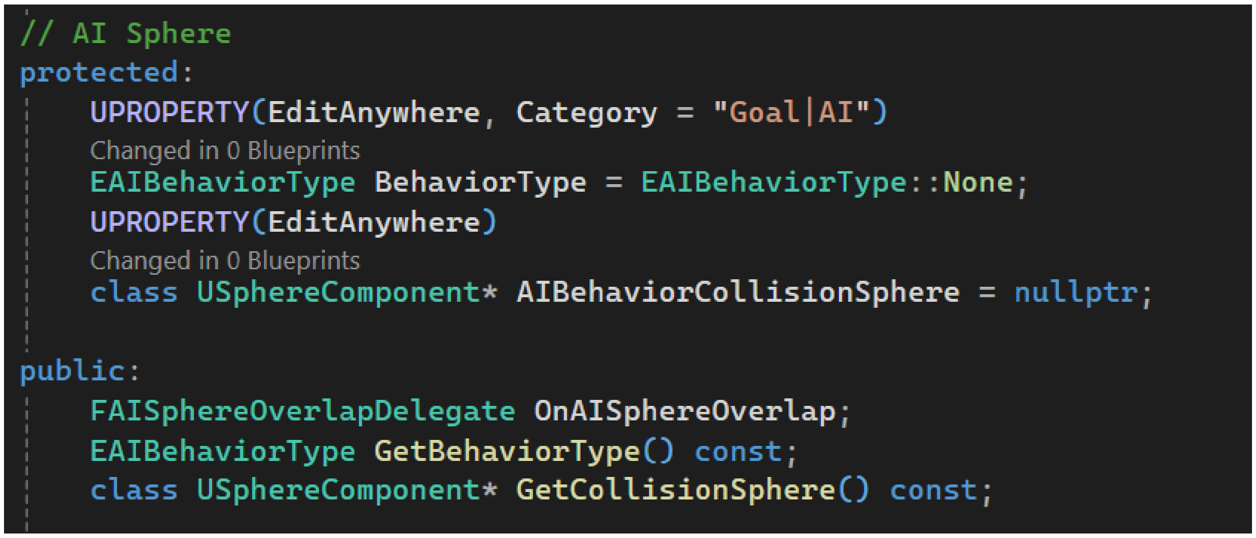
```
    // Place Widget in the right Vertical Box
    FString KeyGroup = MappableKey.GetDisplayCategory().ToString();
    if (KeyGroup == MovementCategoryName)
    {
        BIND_MovementKeyVerticalBox->AddChild(NewKeyWidget);
        MovementKeyMapping.Add(NewKeyWidget);
    }
    else if (KeyGroup == ActionCategoryName)
    {
        BIND_ActionKeyVerticalBox->AddChild(NewKeyWidget);
        ActionKeyMapping.Add(NewKeyWidget);
    }

    // Bind to the Key pressed event
    NewKeyWidget->OnKeyButtonPressed.AddUniqueDynamic(this, &UKeybindingsMenuCommonAW::OnKeyButtonPressed);
}
```

Perfect ! Now we'll reset the focus after each new key binded with a gamepad !

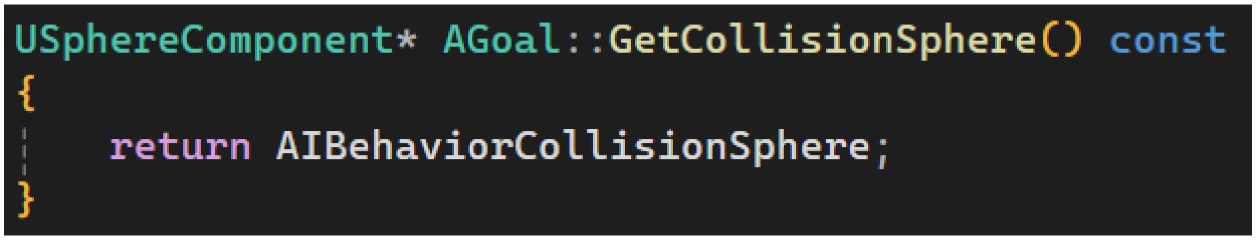
Fix AI Controller Begin Play

Ok this a quick fix for something that we've done in the **EnemyController**.
If you remember, in the **Begin Play** we set the **Blackboard** value of **EnemyIsInDefenseSphereName** to true automatically which is not ideal. If a designer decides to move the AI a bit, our value will not be correct and will break the AI's Behavior.
Instead, we'll check if the AI is inside the **Collision Sphere** at the **Begin Play.**

Start by adding this Getter in the **Goal**.

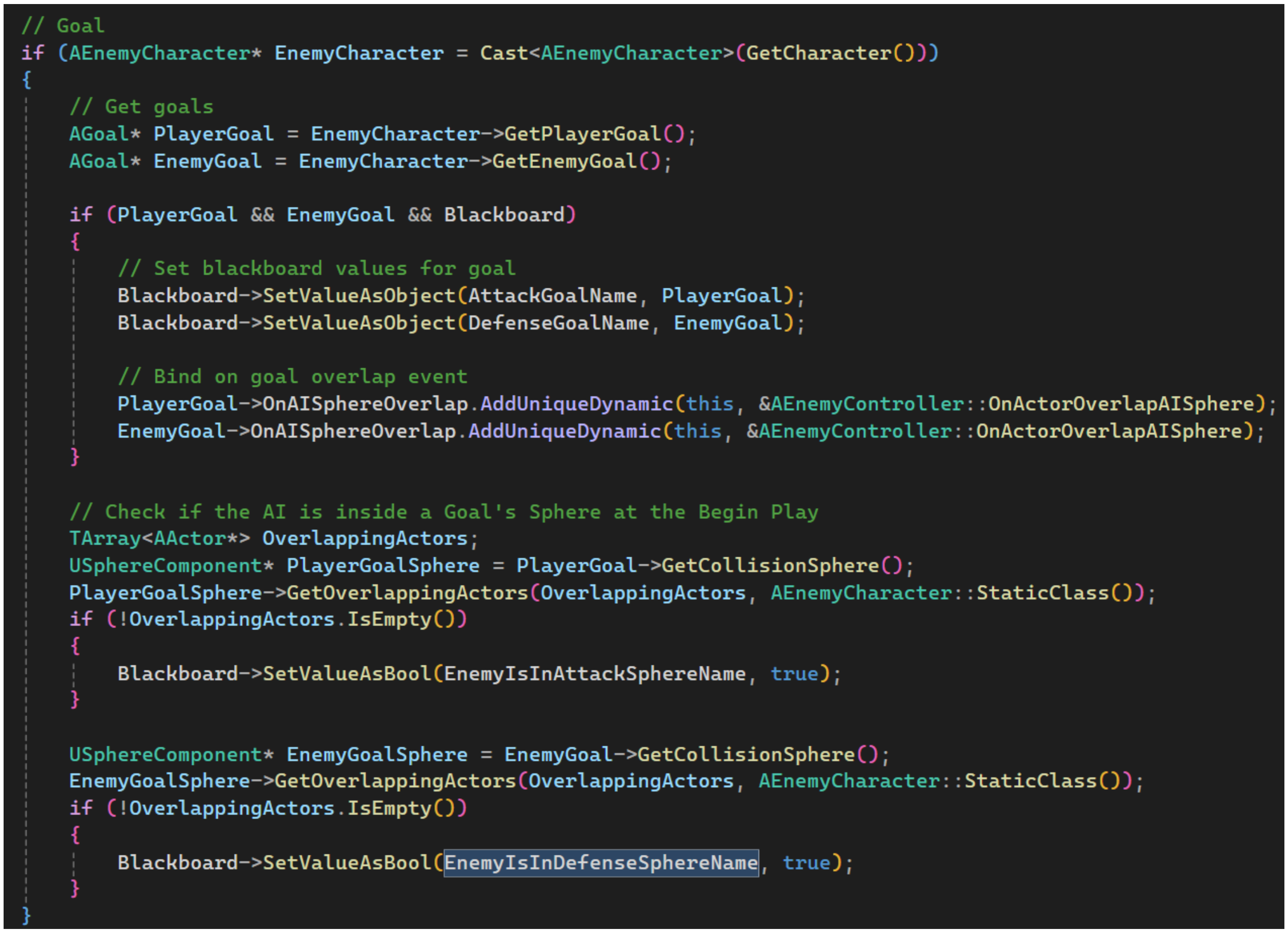


```
// AI Sphere
protected:
    UPROPERTY(EditAnywhere, Category = "Goal|AI")
    Changed in 0 Blueprints
    EAIBehaviorType BehaviorType = EAIBehaviorType::None;
    UPROPERTY(EditAnywhere)
    Changed in 0 Blueprints
    class USphereComponent* AIBehaviorCollisionSphere = nullptr;

public:
    FAISphereOverlapDelegate OnAISphereOverlap;
    EAIBehaviorType GetBehaviorType() const;
    class USphereComponent* GetCollisionSphere() const;
```



```
USphereComponent* AGoal::GetCollisionSphere() const
{
    return AIBehaviorCollisionSphere;
}
```

Then, in the **Enemy Controller's Begin Play**, we'll get it and check if the **Enemy Character** is inside.
For that we'll simply use the **GetOverlappingActors** function.
Please note here that we use a small trick. The second argument of this function is the class we're looking for. Since here we only have one AI, checking if the **OverlappingActors** array is empty is enough. But if we had more than one, we should check the pointers inside this array !



```
// Goal
if (AEnemyCharacter* EnemyCharacter = Cast<AEnemyCharacter>(GetCharacter()))
{
    // Get goals
    AGoal* PlayerGoal = EnemyCharacter->GetPlayerGoal();
    AGoal* EnemyGoal = EnemyCharacter->GetEnemyGoal();

    if (PlayerGoal && EnemyGoal && Blackboard)
    {
        // Set blackboard values for goal
        Blackboard->SetValueAsObject(AttackGoalName, PlayerGoal);
        Blackboard->SetValueAsObject(DefenseGoalName, EnemyGoal);

        // Bind on goal overlap event
        PlayerGoal->OnAISphereOverlap.AddUniqueDynamic(this, &AEnemyController::OnActorOverlapAISphere);
        EnemyGoal->OnAISphereOverlap.AddUniqueDynamic(this, &AEnemyController::OnActorOverlapAISphere);
    }

    // Check if the AI is inside a Goal's Sphere at the Begin Play
    TArray<AActor*> OverlappingActors;
    USphereComponent* PlayerGoalSphere = PlayerGoal->GetCollisionSphere();
    PlayerGoalSphere->GetOverlappingActors(OverlappingActors, AEnemyCharacter::StaticClass());
    if (!OverlappingActors.IsEmpty())
    {
        Blackboard->SetValueAsBool(EnemyIsInAttackSphereName, true);
    }

    USphereComponent* EnemyGoalSphere = EnemyGoal->GetCollisionSphere();
    EnemyGoalSphere->GetOverlappingActors(OverlappingActors, AEnemyCharacter::StaticClass());
    if (!OverlappingActors.IsEmpty())
    {
        Blackboard->SetValueAsBool(EnemyIsInDefenseSphereName, true);
    }
}
```

Conclusion

That’s it for the bonus, they’re still a lot of things to see with the **UI** like the **Tabs** or the **Common Input Settings**. And a lot with the **AI** like the **Perception Component** and the **Nav Link Proxy**.

If you’ve got any questions regarding this or something else in UE5, feel free to reach me with the Discord server or by mail via victor.giroux@artfx.fr.
Good luck with your graduation projects !