1.2拥有自己的主菜单界面

1.2.1准备工作：创建GameInstance类和其他类

（1）创建相关的类

我们创建一个继承于GameInstance的C++类，命名为TutorialGameInstance，同时为了方便在编辑器中进行绑定操作，我们创建一个对应的蓝图类BP_TutorialGameInstance。

这个类将在游戏开始时创建，并且它将在整个游戏中持续存在，直到将游戏关闭。GameInstance时位于本地的，这意味着它不进行一些复制的操作。通常用于UI和会话之类的，我们需要在项目设置之中对GameInstance进行设置，如图1.2.1.1所示，在下面的GameInstance设置为BP_TutorialGameInstance。

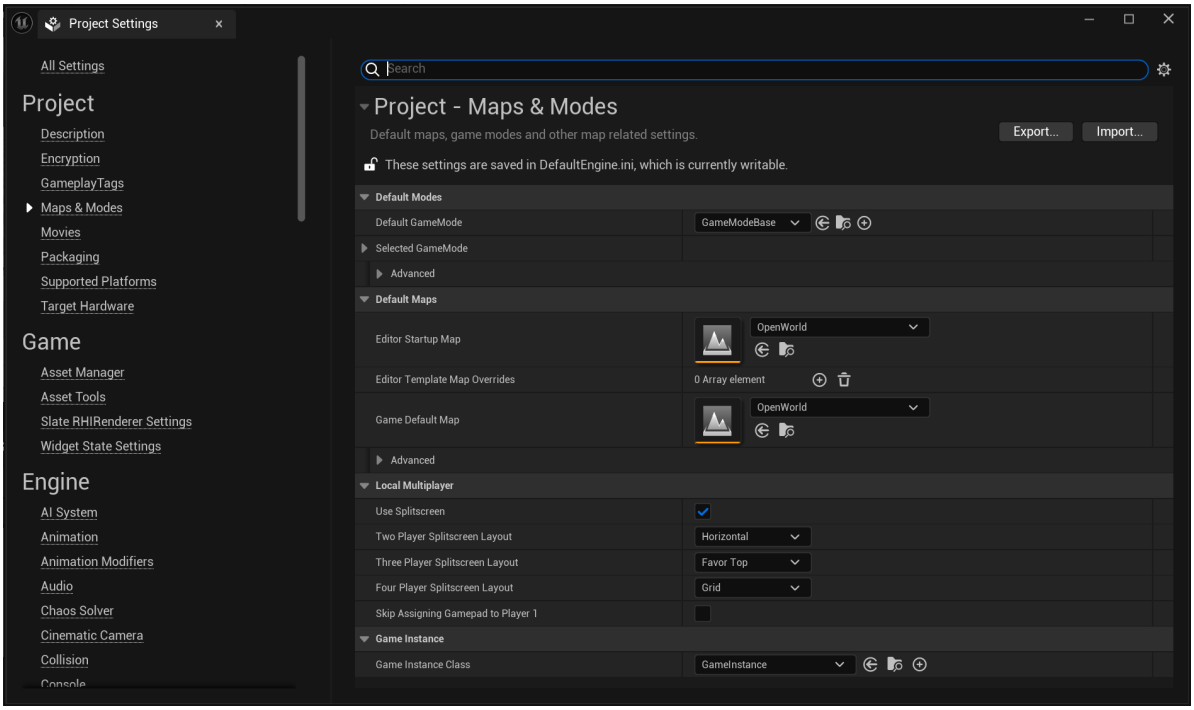

图1.2.1.1 设置GameInstance

现在我们还需要创建一个用于主菜单中的GameMode，Character，Contorller，我们分别命名为ShootMenuGameMode，ShootMenuCharacter,ShootMenuController，并都创建它们对应的蓝图类。

接着打开BP_ShootMenuGameMode中，我们设置PlayerControllerClass为BP_ShootMenuController，设置HUD_Class 为None，设置Default Pawn Class为BP_ShootMenuCharacter，如图1.2.1.2所示。然后我们创建一个新关卡EmptyLevel命名为MainMenu_Map作为我们的菜单页面，我们设置我们的WorldSettings，设置GmaeMode为我们的BP_ShootMenuGameMode。

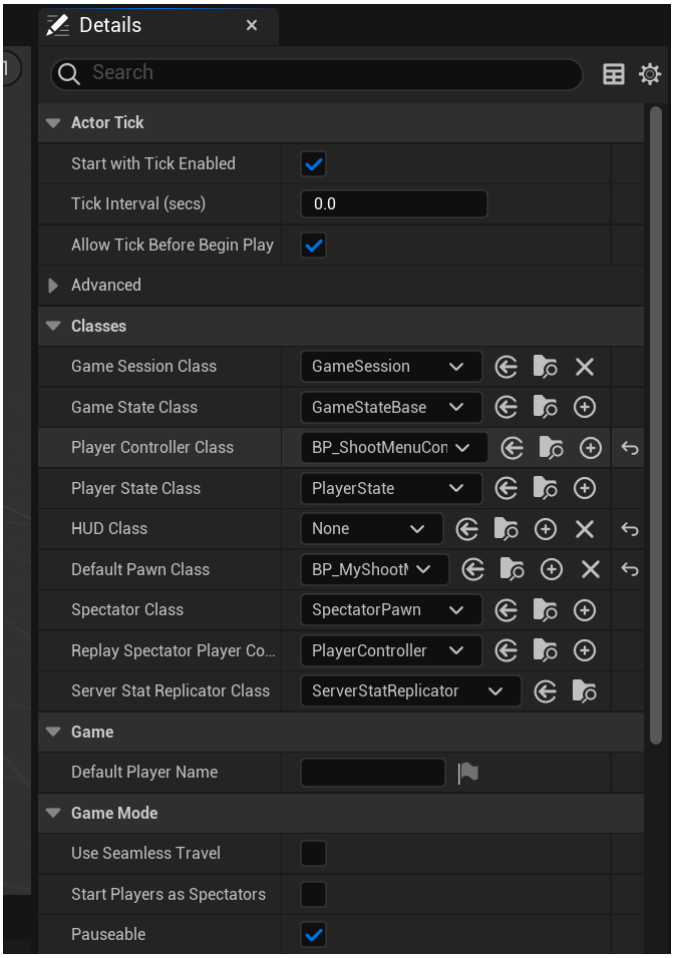

图1.2.1.2 设置BP_ShootMenuGameMode

我们再创建一个Basic关卡命名为LobbyMenu_Map作为我们的多人游戏大厅页面。先设置GameMode为BP_ShootMenuGameMode。

（2）添加相关模块

为了后续我们能够用UMG，SlateCore模块，我们需要添加一些内容在*.Build.cs文件中。

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

如同我们1.1中添加Niagara的操作一般，我们仅需要修改*.Build.cs即可。记得重启VS2022和关闭UE编辑器。

1.2.2创建UserWidget界面

（1）创建一个UserWidget蓝图和类。在ContentBrower中先创建一个UserWidget类，点击右键选择到UserInterface/ WidgetBlueprint进行创建。在弹出来的界面如图1.2.2.1中，我们可以简单地选择UserWidget选项进行创建，并重命名为WB_MainMenu。

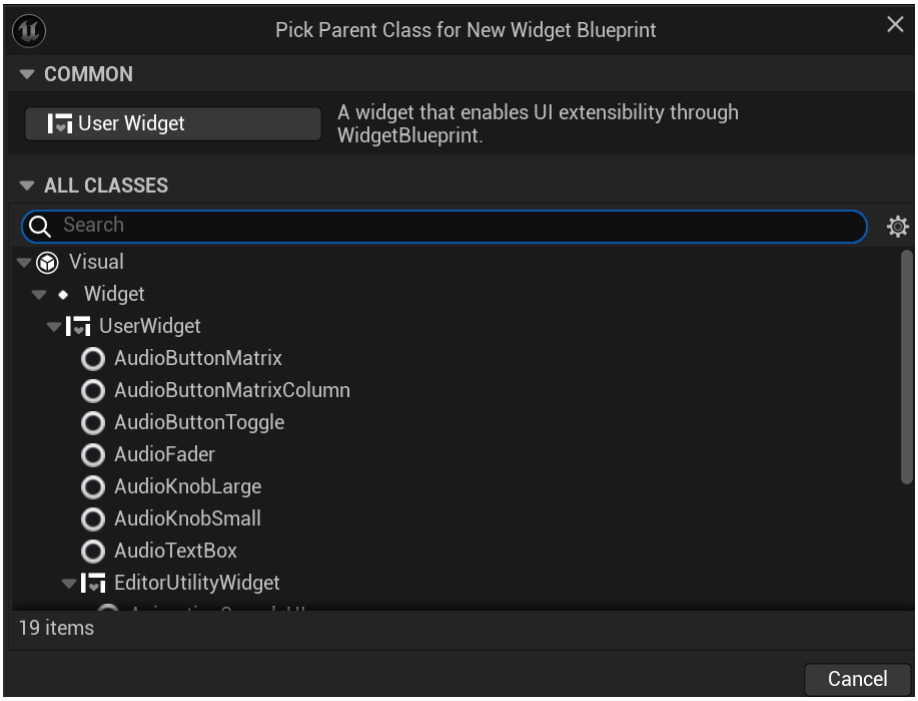

图1.2.2.1 创建一个UserWidget蓝图

在C++中，我们创建一个类UserWidget命名为MainMenuUserwWidget。然后我们在我们刚创建的WB_MainMenu蓝图中设置父类为MainMenuUserwWidget。首先，我们打开蓝图界面，我们可以看到右上角有Designer和Graph两个选项。默认情况下，这个是选择Designer的，我们需要切换到Graph界面。接着我们需要在上方的选项栏中将Class Defaults切换到Class Settings，如图1.2.2.2所示。接下来，我们在左下角的Details面板中绑定ClassOptions/ParentClass为MainMenuUserwWidget。

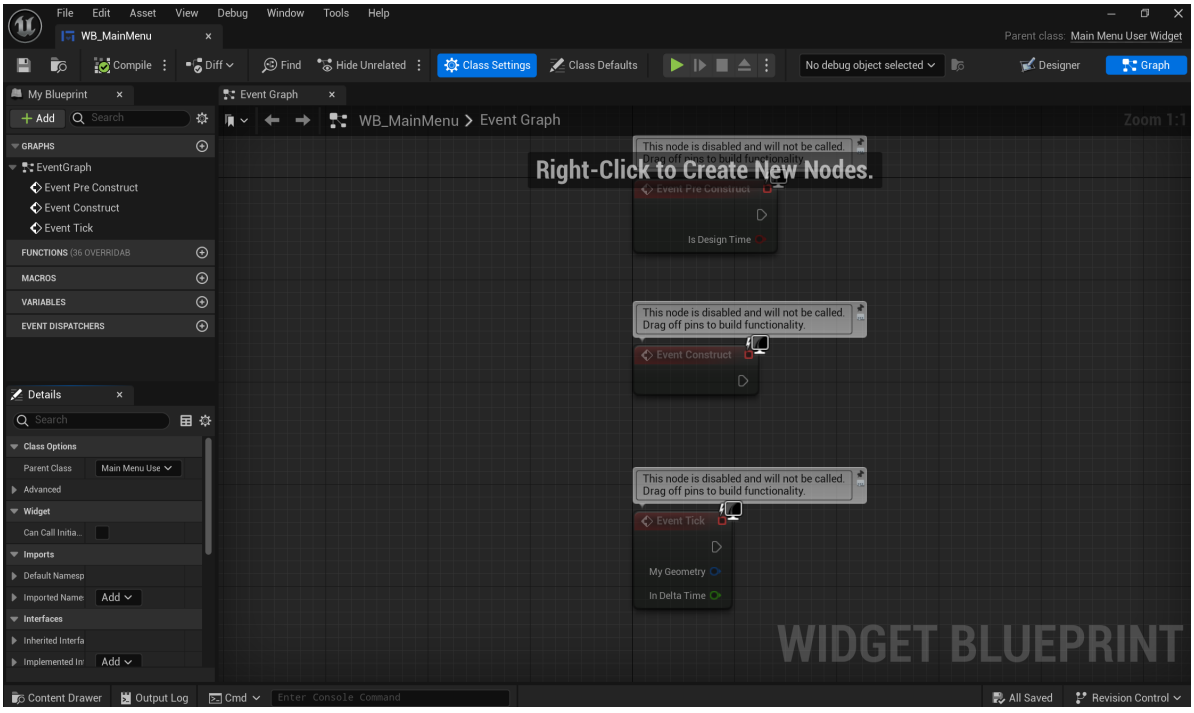

图1.2.2.2 绑定UserWidget父类

至此，我们绑定完父类，我们切换回Designer页面中，设计我们的UI界面。

（2）设计UI界面

首先，先展示我们的UI界面的Hierarchy层次结构，如图1.2.2.3所示。

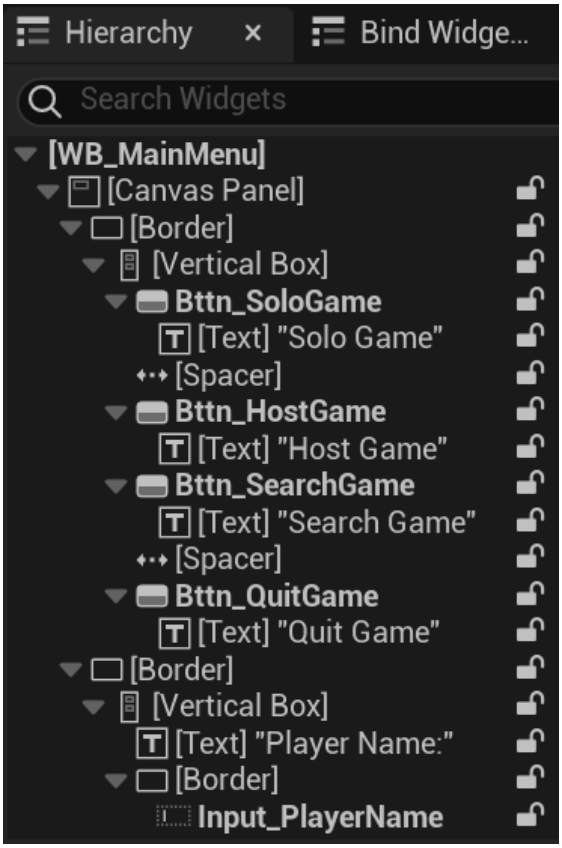

图1.2.2.3 MainMenu的Hierarchy结构

接着，我们做的第一件事就是在Palette面板中搜索，并且拖入一个Canvas Panel来，拖到Hierarchy面板中。

然后，拖入一个border到Canvas内，可以在右边Slot处设置border的高宽和位置可以在Appearance/Brush Color处的A数值设置透明度为0.5。接下来我们对我们的Border进行定位，原本是如图1.2.2.4所示的定位，我们对左上角那个花瓣图案的定位器，拖拽四个角的花瓣，拖拽成如图1.2.2.5所示的页面。这一步操作的作用是，让UI界面在不同的分辨率下也能保持一定的位置和比例。

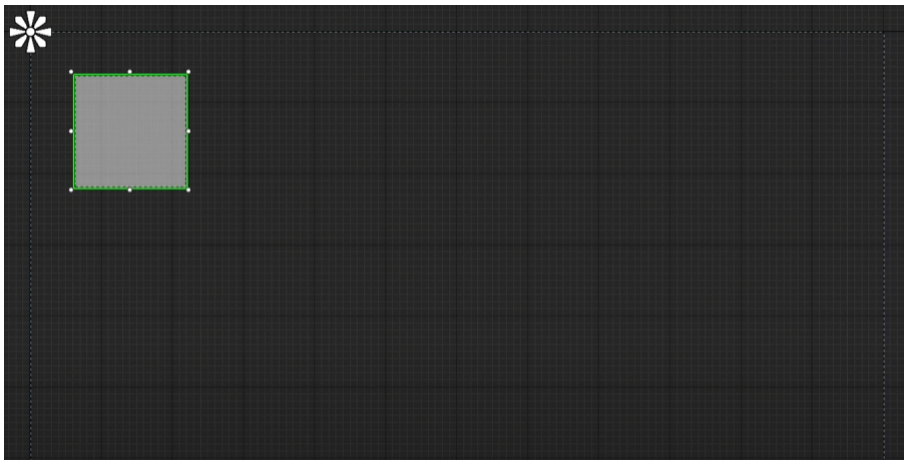

图1.2.2.4 未更改前的定位方式

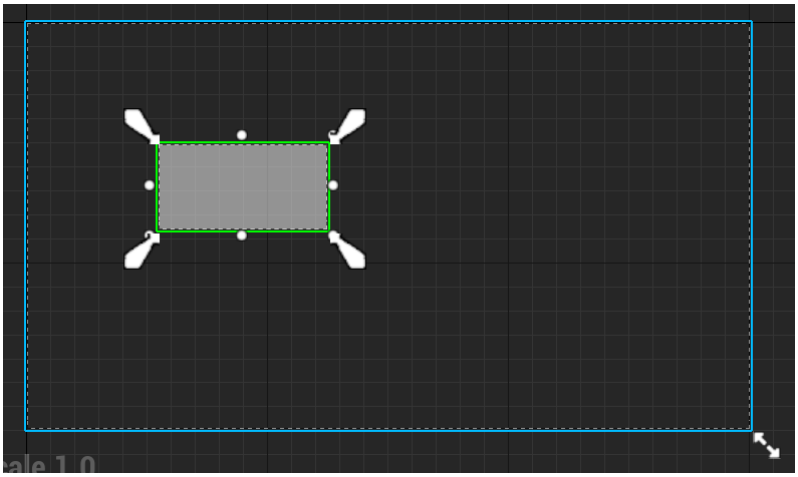

图1.2.2.5 更改后的定位方式

接下来，我们搜索垂直框Vertical Box将其拖入border内。

向垂直框内拖入一个Button并重命名为Bttn_SoloGame，同时在Details面板的上方设置该按钮为Is Variable，如图1.2.2.6所示。在Appearance/style处可以设置Normal，Hovered，Pressed状态下的button的显示，我们将normal中Tint设置为（0，0，0，0.7）黑色，透明度为0.7。Hovered我们会让Tint设置为（0，0，0，1.0）。Pressed我们会让Tint设置为（0，0，0，0.3）。

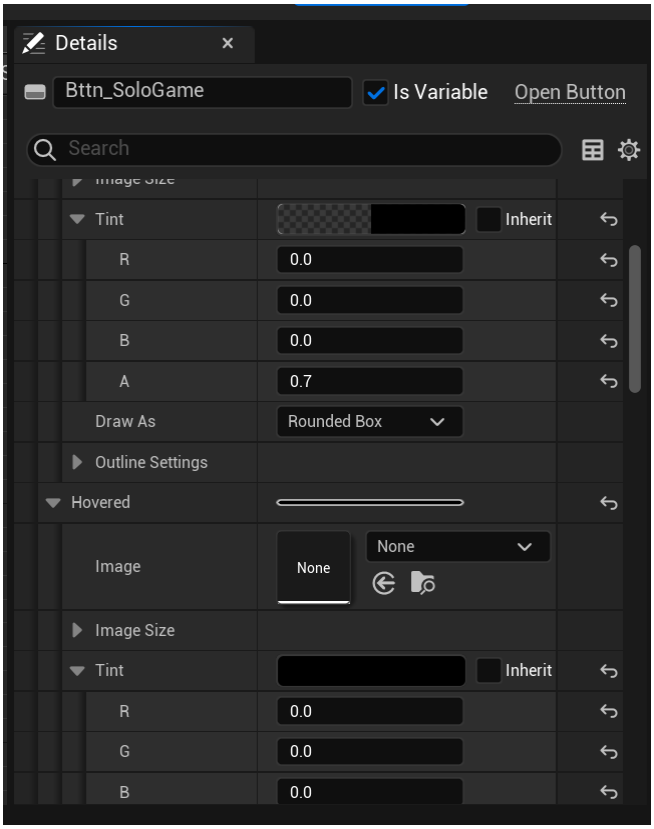

图1.2.2.6 设置Is Variable

在Button内拖入一个Text，可以在右边细节面板Content/Text处修改内容SoloGame，可以在右边Appearance/Font/Typeface处修改字体为斜体italic。

然后，我们复制Bttn_SoloGame按钮，然后粘贴在同一层次下。我们重命名按钮名字为Bttn_HostGame,修改其中的Text的内容为Host Game。再次复制粘贴出一个按钮，名字命名为Bttn_SearchGame,Text内容修改为Search Game再次复制粘贴出一个按钮，名字命名为Bttn_QuitGame，Text内容修改为Quit Game。

在Palette中拖拽出一个Spacer，放到Bttn_SoloGame和Bttn_HostGame之间，修改Appearance/Size为（1.0，15.0）。同样地，在Bttn_SearchGame和Bttn_QuitGame之间我们也拖一个Spacer进来修改Appearance/Size为（1.0，25.0）。

然后，我们调整一下Broder的高度和我们的四个按钮匹配就行。记得点击编译保存。

1.2.3展示MainMenu在开始游戏时

点击打开我们的TutorialGameInstance类，我们需要创建一个UI_ShowMainMenu函数用于展示我们的MainMenu界面。

```
public:
    UFUNCTION()
    void UI_ShowMainMenu();

    UPROPERTY(EditAnywhere, Category = "UI")
    TSubclassOf<UMainMenuUserWidget> MainMenuWidgetClass;

    UPROPERTY()
    UMainMenuUserWidget* MainMenu;
```

并且在Cpp创建它的定义，如下：

```
#include "MainMenuUserWidget.h"
void UTutorialGameInstance::UI_ShowMainMenu()
{
    if (!IsValid(MainMenu)) {
        MainMenu = CreateWidget<UMainMenuUserWidget>(GetWorld(),
MainMenuWidgetClass);
    }
    MainMenu->AddToViewport(0);
    if (APlayerController* PlayerController = GetWorld()-
>GetFirstPlayerController()) {
        FInputModeUIOnly InputMode;
        PlayerController->SetInputMode(InputMode);
        PlayerController->bShowMouseCursor = true;
    }
}
```

在这个函数中，我们首先检验了MainMenu的有效性。如果无效，那么我们创建一个MainMenu给它。如果有效，那么我们把它添加到显示界面上，AddToViewport的第一个参数的作用是，用于指示这个Userwidget的层级，数字越高的层级越高。然后，我们设置输入方式，我们设置为仅UI交互，而且设置bShowMouseCursor为true表示显示我们的光标在界面中。

编译保存后，我们记得要在编辑器的BP_TutorialGameInstance中选择我们的MainMenuWidgetClass。

接下来我们需要实现的是在Level关卡中进行调用，我们创建一个继承于LevelScriptActor的C++类，命名为MainMenuLevelScriptActor。如同我们绑定Widget父类一样我们绑定在MainMenu_Map中绑定这个地图的父类，如图1.2.3.1所示我们点开地图的Blueprint。在蓝图界面上方找到ClassSettings并且在Details面板中的ClassOptions/ParentClass中设置我们的父类为MainMenuLevelScriptActor，如图1.2.3.2所示。

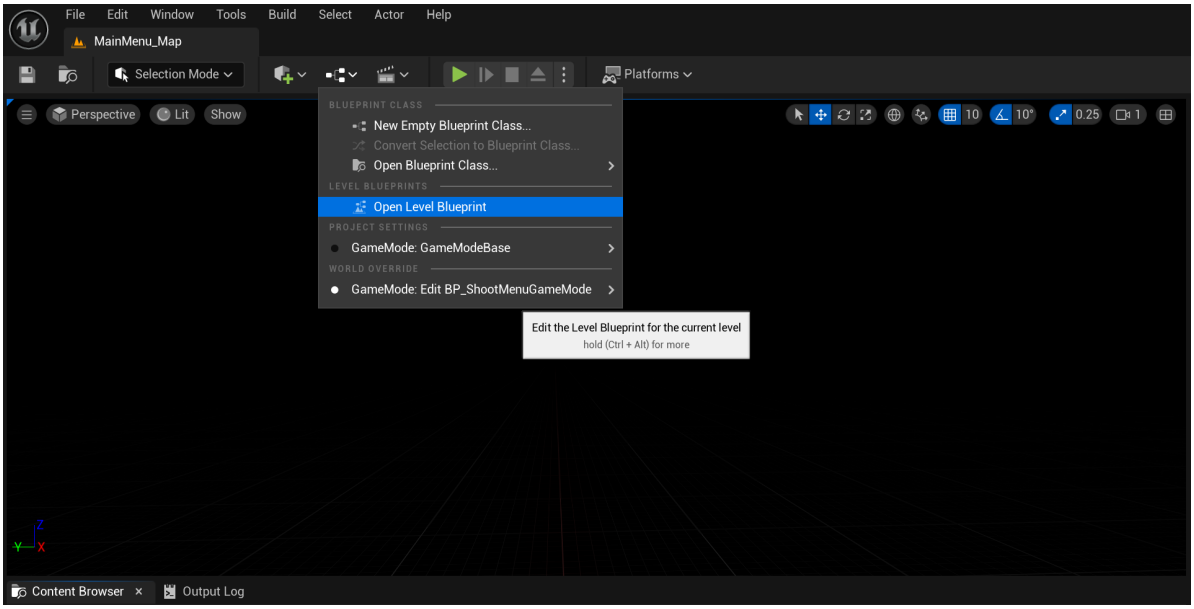

图1.2.3.1 打开MainMenu_Map的蓝图界面

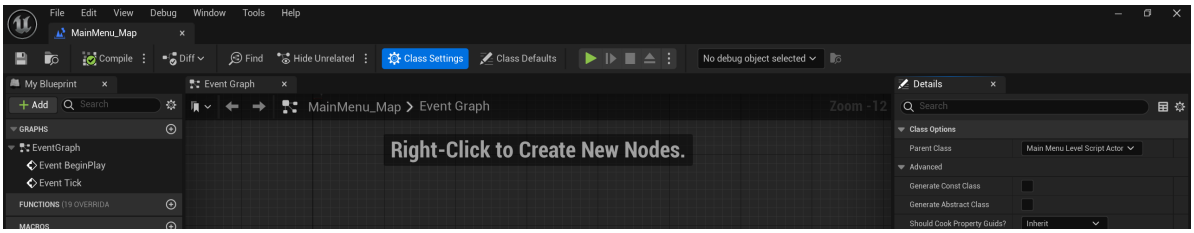

图1.2.3.2 设置MainMenu_Map的父类

然后，我们在C++代码中声明出它的BeginPlay()函数。

```
protected:
    virtual void BeginPlay() override;
```

在Cpp中创建它的定义：

```
#include "TutorialGameInstance.h"
void AMainMenuLevelScriptActor::BeginPlay()
{
    Super::BeginPlay();
    UTutorialGameInstance* MyGameInstance = Cast<UTutorialGameInstance>
(GetGameInstance());
    if (MyGameInstance) {
        MyGameInstance->UI_ShowMainMenu();
    }
}
```

我们实现父类的BeginPlay()，然后获取当前游戏的GameInstance转换为我们自己定义的类，然后调用UI_ShowMainMenu方法。这样我们就能实现在打开这个地图的时候，就能展示我们提前在UE编辑器中设置的UI界面。接下来我们编译保存一下，如果运行正常的话，我们就能看到我们的界面了，记得在编辑器的BP_TutorialGameInstance中选择我们的Widget蓝图类，运行后如图1.2.3.3的显示。

图1.2.3.3 运行界面

1.2.4按钮事件的绑定（委托事件）

（1）对Bttn_QuitGame绑定退出事件。我们想要将蓝图中对应的按钮映射到C++中，我们需要将蓝图中所需的变量用相同的名字在MainMenuUserWidget头文件中进行声明。

```
public:
    UPROPERTY(BlueprintReadOnly, meta = (BindWidgetOptional))
    UButton* Bttn_SoloGame;

    UPROPERTY(BlueprintReadOnly, meta = (BindWidgetOptional))
    UButton* Bttn_HostGame;

    UPROPERTY(BlueprintReadOnly, meta = (BindWidgetOptional))
    UButton* Bttn_SearchGame;

    UPROPERTY(BlueprintReadOnly, meta = (BindWidgetOptional))
    UButton* Bttn_QuitGame;
```

其中BlueprintReadOnly表示在蓝图中仅读。

meta = (BindWidgetOptional)指定此属性是一个 绑定到 UMG Widget 的指针。如果UMG中没有对应的名称的Widget，属性可以是nullptr，并且不会在编辑器中编译的时候报错，常用于绑定可能存在或不存在的可选Widget。如果我们使用的是BindWidget，那么我们必须保证蓝图中有对应名称的控件。

同时声明我们的初始化函数NativeOnInitialized，这个函数当 UMG Widget 完成初始化后自动调用，仅一次。我们通常会在其中绑定我们的点击事件之类的。

```
public:
    virtual void NativeOnInitialized() override;

    UFUNCTION()
    void OnBttn_QuitGameClick();
```

我们声明我们的初始化函数NativeOnInitialized和我们将要绑定的函数OnBttn_QuitGameClick。接下来我们创建NativeOnInitialized的实现。

```
#include "Components/Button.h"
void UMainMenuUserWidget::NativeOnInitialized()
{
    Super::NativeOnInitialized();
    if (Bttn_QuitGame) {
        Bttn_QuitGame->OnClicked.AddDynamic(this,
&UMainMenuUserWidget::OnBttn_QuitGameClick);
    }
}
```

在这个函数中，我们绑定了Bttn_QuitGame的点击事件，响应这个事件的函数是OnBttn_QuitGameClick。

```
#include "Kismet/KismetSystemLibrary.h"
#include "GameFramework/PlayerController.h"
#include "Engine/World.h"
void UMainMenuUserWidget::OnBttn_QuitGameClick()
{
    APlayerController* PlayerController = GetWorld()-
>GetFirstPlayerController();
    if (PlayerController)
    {
        UKismetSystemLibrary::QuitGame(GetWorld(), PlayerController,
EQuitPreference::Quit, true);
    }
}
```

这个函数先找到了PlayerController，然后调用QuitGame函数退出了游戏。至此，我们就完成了游戏的Bttn_QuitGame按钮

（2）绑定Bttn_SoloGame进入游戏事件。

我们的流程是OnClicked点击Bttn_SoloGame按钮，会调用对应的响应函数OnBttn_SoloGameClick。在这个响应函数中，我们会实现进入游戏的函数TriggerStartSoloGameEvent，这个函数的声明与定义我们放到TutorialGameInstance中。在TutorialGameInstance中，我们利用我们绑定的动态多播委Broadcast触发委托事件，通知所有已经绑定到该委托的函数（StartSoloGameEvent）执行相应的逻辑。

首先，在MainMenuUserWidget我们需要获得到我们的TutorialGameInstance。我们可以利用NativeConstruct进行获取。

```
public:
    UFUNCTION()
    void OnBttn_SoloGameClick();
protected:
    virtual void NativeConstruct() override;

    UTutorialGameInstance* MyGameInstance;
```

在上面代码中，我们声明了NativeConstruct，这个函数会在NativeOnInitialized调用后进行调用。每次显示UI界面都会调用，可以多次调用（比如重新加载或重新加入界面时）。

我们创建对应的Cpp实现.

```
void UMainMenuUserWidget::NativeConstruct()
{
    Super::NativeConstruct();
    MyGameInstance = Cast<UTutorialGameInstance>(GetGameInstance());
}

void UMainMenuUserWidget::OnBttn_SoloGameClick()
{
    if (MyGameInstance) {
        MyGameInstance->TriggerStartSoloGameEvent();
    }
}
```

我们调用GetGameInstance获取到当前游戏GameInstance。然后在OnBttn_SoloGameClick函数中，我们调用MyGameInstance的方法（该方法我们还没声明）。然后，不要忘了在NativeOnInitialized对这个按钮点击函数进行绑定。

```
void UMainMenuUserWidget::NativeOnInitialized()
{
    Super::NativeOnInitialized();
    if (Bttn_QuitGame) {...}
    if (Bttn_SoloGame) {
        Bttn_SoloGame->OnClicked.AddDynamic(this,
&UMainMenuUserWidget::OnBttn_SoloGameClick);
    }
}
```

然后，我们转到TutorialGameInstance中进行后续动态多播事件的声明和实现。

我们先讲解一下委托事件。单委托允许绑定一个函数。委托分为 单委托、多委托、动态单播委托 和 动态多播委托（以下是无参数版本）。

- DECLARE_DELEGATE：单委托。高性能，适合频繁调用的场景，仅支持 C++，不能与蓝图交互。如果绑定了多个函数，后一次绑定会覆盖之前的绑定。用于简单事件或需要快速响应的事件。

  声明：DECLARE_DELEGATE(FSimpleDelegateName);

- DECLARE_MULTICAST_DELEGATE：多委托。多委托允许绑定多个函数。所有绑定的函数会按照绑定顺序依次执行。用于需要通知多个监听者的事件。

  声明：DECLARE_MULTICAST_DELEGATE(FSimpleMulticastDelegateName);

- DECLARE_DYNAMIC_DELEGATE：动态单播委托。动态单播委托允许绑定一个函数或蓝图事件。仅能绑定一个监听者，如果多次绑定，后一次绑定会覆盖之前的。性能较低（因为使用了反射系统）。用于需要蓝图绑定但只需要一个监听者的场景。

- DECLARE_DYNAMIC_MULTICAST_DELEGATE：动态多播委托 。动态多播委托允许绑定多个函数或蓝图事件。能够同时通知多个监听者。能够同时通知多个监听者。

那么接下来，我们在TutorialGameInstance声明我们的动态多播委托。声明的位置在类的声明外部。

```
#include "xxxxx"
DECLARE_DYNAMIC_MULTICAST_DELEGATE(FStartSoloGameEventDelegate);
...
```

在类的声明中，我们声明我们的动态多播委托OnStartSoloGameEventTriggered，触发委托的函数TriggerStartSoloGameEvent，委托调用的函数StartSoloGameEvent, 是否处于单人游戏的标志IsSoloGame。同时，我们还需要声明我们的Instance的初始化函数Init，这个初始化函数用于绑定我们的动态多播委托的调用函数。

```
public:
    virtual void Init() override;
public:
    bool IsSoloGame;

    UPROPERTY(BlueprintAssignable, Category = "Events")
    FStartSoloGameEventDelegate OnStartSoloGameEventTriggered;

    void TriggerStartSoloGameEvent();

    UFUNCTION()
    void StartSoloGameEvent();
```

接下来，我们在cpp中创建对应的实现。

```
void UTutorialGameInstance::Init()
{
    Super::Init();
    OnStartSoloGameEventTriggered.AddDynamic(this,
&UTutorialGameInstance::StartSoloGameEvent);
}
```

这个Init函数中，我们绑定了我们的StartSoloGameEvent到OnStartSoloGameEventTriggered这个动态多播委托上，将来调用OnStartSoloGameEventTriggered.Broadcast();的时候，这个StartSoloGameEvent就会被调用。

接下来，我们需要实现我们的StartSoloGameEvent函数。

```
void UTutorialGameInstance::StartSoloGameEvent()
{
    IsSoloGame = true;
    UGameplayStatics::OpenLevel(this, FName("Level01"));
}
```

然后，我们需要提供一个TriggerStartSoloGameEvent的实现，供给其他人。

```
void UTutorialGameInstance::TriggerStartSoloGameEvent()
{
    OnStartSoloGameEventTriggered.Broadcast();
}
```

至此，我们可以进行编译保存，我们已经能够在UI界面中点击QuitGame和SoloGame，进行退出游戏和单人游戏了。

1.2.5创建游戏中的Controller

这个时候，我们进入游戏关卡我们会发现我们的视角是不能移动的。因为我们之前显示MainMenu界面的时候将操作更改成了仅UI输入。因此，在这里我们需要创建一个新的PlayerController的C++类命名为ShooterPlayerController。接下来，打开我们在1.1节中创建的GameMode的蓝图，在里面将游戏的PlayerControllerClass更改为我们的ShooterPlayerController。

我们打开刚创建的ShooterPlayerController的C++头文件中，添加一个BeginPlay函数。我们需要在这个函数中修改我们的输入方式。

```
public:
    virtual void BeginPlay() override;
```

并在cpp中创建下面的定义。

```
void AShooterPlayerController::BeginPlay()
{
    Super::BeginPlay();

    if (IsLocalController()) {
        FInputModeGameOnly InputMode;
        SetInputMode(InputMode);
        bShowMouseCursor = false;
    }

}
```

因为，我们的输入事件，只需要在本地发生，所以首先检查我们是否是本地玩家控制器。我们需要设置为仅游戏模式，然后设置显示鼠标光标为false。

进行编译保存，我们回到游戏中，我们应该就能够控制我们的角色了。

1.2.6添加游戏保存功能

在开始创建和搜索会话之前，我们可以设置一个玩家资料，这样我们就可以保存一些像玩家名字这样的东西。如果我们用一个玩家连接到服务器，那么需要确保连接到服务器的时候，可以简单地加载这些玩家资料的信息到服务器。

（1）创建一个USTRUCT结构体S_PlayerProfile

我们创建一个None空类命名为S_PlayerProfile。原来的在.h和.cpp的内容可以全部删去。修改成以下内容。

S_PlayerProfile.h

```
#pragma once

#include "CoreMinimal.h"
#include "S_PlayerProfile.generated.h"

/**
 *
 */
USTRUCT(BlueprintType)
struct FS_PlayerProfile
{
```

```
    GENERATED_BODY()

public:

    UPROPERTY(BlueprintReadWrite, EditAnywhere, Category = "PlayerProfile")
    FText PlayerName;

    FS_PlayerProfile()
        : PlayerName(FText::FromString(TEXT("Hello Unreal!")))
    {
    }
};
```

S_PlayerProfile.cpp

```
#include "S_PlayerProfile.h"
```

这就是我们用于存储玩家用户资料的结构体，结构体中保存了PlayerName信息。

（2）创建一个USaveGame类SG_PlayerProfile

我们创建一个继承于USaveGame的C++类，命名为SG_PlayerProfile。这个类的作用是用于保存我们结构体中的信息，它可以调用一些保存和加载的函数。我们在SG_PlayerProfile类的头文件中创建我们的结构体。

```
#include "S_PlayerProfile.h"
public:
    UPROPERTY()
    FS_PlayerProfile PlayerProfileStruct;
```

（3）在GameInstance中创建CheckForSavedProfile函数

接下来，我们需要创建CheckForSavedProfile在TutorialGameInstance中，这个函数用来加载我们的存储文件的信息。

```
UFUNCTION()
void CheckForSavedProfile();

FString PlayerProflie_Shot = FString("PlayerProflie_Shot");
```

其中，PlayerProflie_Shot这个变量是用来充当存储文件信息的文件名字。接下来我们在cpp创建它的定义。

```
#include "Kismet/GameplayStatics.h"
void UTutorialGameInstance::CheckForSavedProfile()
{
    if (UGameplayStatics::DoesSaveGameExist(PlayerProflie_Shot, 0)) {
        LoadProfile();
    }
    else {
        SaveProfile();
    }
}
```

以上代码中，我们检查目录中是否有PlayerProflie_Shot这个存档槽，第二个参数0表示用户索引。保存的目录默认是根目录下\Saved\SaveGames。如果成功保存，我们将能够在这个目录下看到对应的文件。其中，LoadProfile和SaveProfile两个函数我们接下来会进行实现。

（4）创建LoadProfile函数

如果用户文件已经存在，那么我们就加载这个文件。我们的声明如下：

```
#include "S_PlayerProfile.h"
public:
    UFUNCTION()
    void LoadProfile();
    UPROPERTY()
    USG_PlayerProfile* SG_PlayerProfile;
    UPROPERTY()
    FS_PlayerProfile PlayerProfileinfo;
```

在头文件中，我们声明了SG_PlayerProfile，这个用于存储我们加载的游戏信息，或者将要保存的游戏信息。PlayerProfileinfo这个是当前显示的游戏信息。对应的实现如下：

```
#include "SG_PlayerProfile.h"
void UTutorialGameInstance::LoadProfile()
{
    SG_PlayerProfile = Cast<USG_PlayerProfile>
(UGameplayStatics::LoadGameFromSlot(PlayerProflie_Shot, 0));
    if (SG_PlayerProfile) {
        PlayerProfileinfo = SG_PlayerProfile->PlayerProfileStruct;
    }
}
```

（5）创建SaveProfile函数

我们将会调用这个函数进行保存。声明如下：

```
UFUNCTION()
void SaveProfile();
```

我们的实现如下：

```
void UTutorialGameInstance::SaveProfile()
{
    if (IsValid(SG_PlayerProfile)) {
        ;
    }
    else {
        SG_PlayerProfile = Cast<USG_PlayerProfile>
(UGameplayStatics::CreateSaveGameObject(USG_PlayerProfile::StaticClass()));
    }
    if (SG_PlayerProfile) {
        SG_PlayerProfile->PlayerProfileStruct = PlayerProfileinfo;
        bool bTemp = UGameplayStatics::SaveGameToSlot(SG_PlayerProfile,
PlayerProflie_Shot, 0);
    }
}
```

在这段代码中，我们先确认了SG_PlayerProfile的可用性。如果不可用，我们将会创建一个USG_PlayerProfile类型的新变量。然后为这个变量赋值，赋值内容是当前的用户信息。接下来调用UGameplayStatics::SaveGameToSlot进行用户信息的保存，第一个参数是指向要保存的SaveGame的指针，第二个参数是保存的文件的名字，第三个参数是用户索引。

（6）在WB_MainMenu的蓝图添加输入框。

先给出最后的成果图，如图1.2.6.1所示。请注意组件层级等信息。

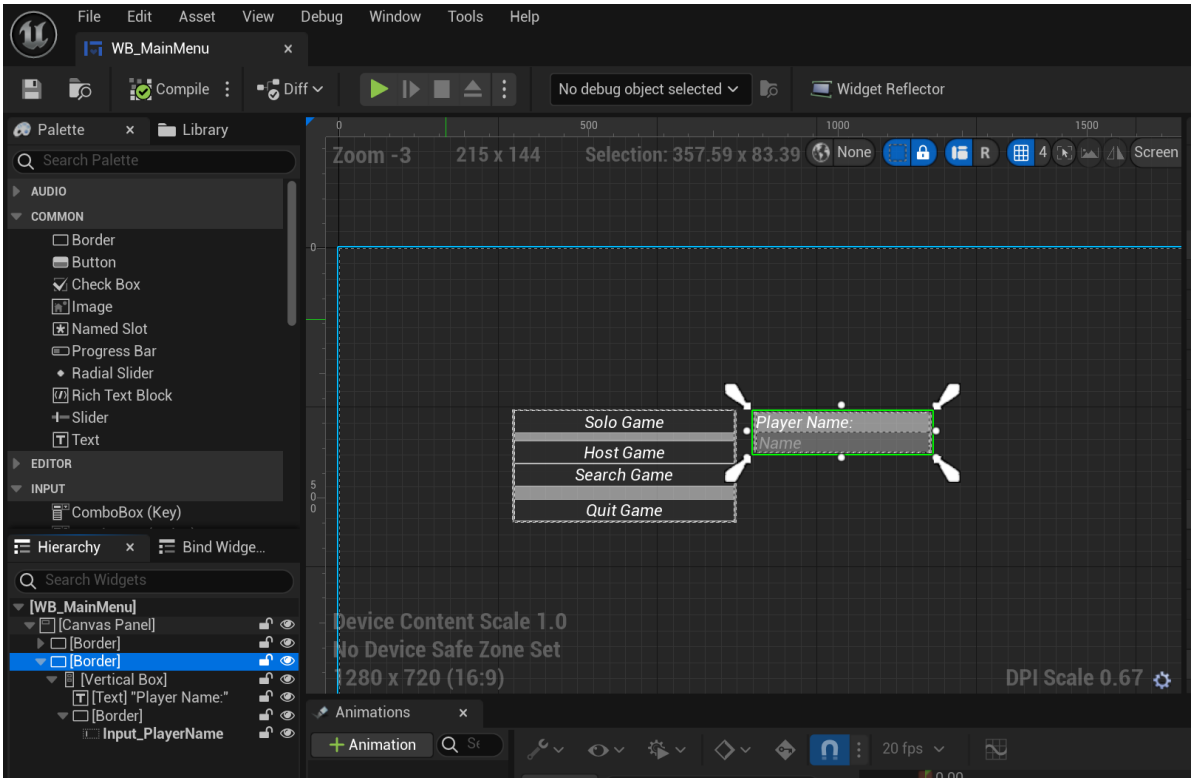

图1.2.6.1 添加输入框

首先，我们要添加一个边框Border，然后拖动到指定位置并修改定位器如图1.2.6.1所示。然后拖拽入一个垂直框VerticalBox。拖拽进一个Text用于让玩家直到他在这里应该填写什么。我们可以复制左边SoloGame的字体然后粘贴到右边，如图1.2.6.2所示。

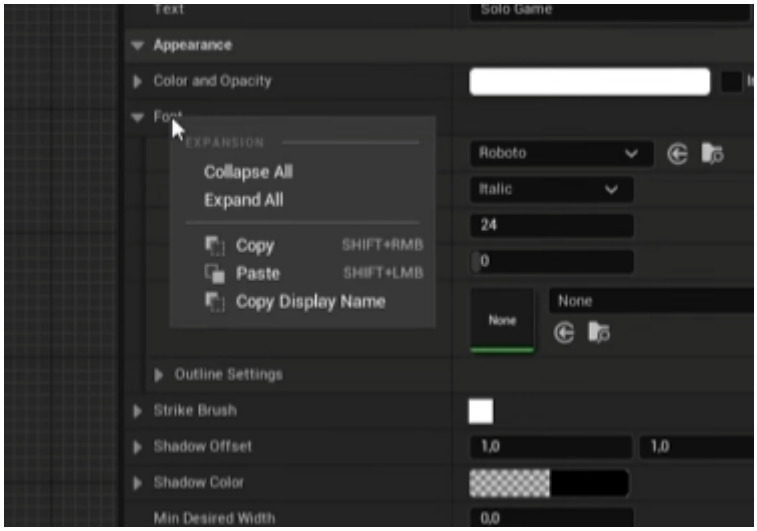

图1.2.6.2 复制字体

然后粘贴到目标Text的相同位置即可，修改文本内容为“Player Name：”。

我们拖入一个边框Border用于装放我们的可编辑文本。设置Details面板中的Appearance/Brush Color为（0，0，0，0.3）。

接下来我们拖入一个Editabe Text的可编辑文本，在Details面板的下方第一行修改名称为Input_PlayerName并且勾选上IsVariable选项。在Details面板中，我们设置Content/Hint Text为“Name”，这个会用于在用户没有输入的时候的默认文字。我们设置Appearance/Color and Opacity的Inherit的勾去掉，以便我们可以更改字体颜色，我们更改为（1.0，1.0，1.0，1.0）即可。

此时，页面应该如图1.2.6.1所示。

（7）在C++中绑定文本输入事件

在这里我们当检查到用户的焦点离开了这文本框的时候，我们需要进行保存。这时，我们的流程是，离开文本框，调用响应函数OnInput_PlayerNameCommitted，响应函数调用MyGameInstance的OnPlayerNameChanged方法并传入一个参数Text，用于改变我们的文本内容和进行保存。

在MainMenuUserWidget头文件中，我们声明一个UEditableText类型的变量。

```
public:
    UPROPERTY(BlueprintReadOnly, meta = (BindWidgetOptional))
    UEditableText* Input_PlayerName;

    UFUNCTION()
    void OnInput_PlayerNameCommitted(const FText& Text, ETextCommit::Type
CommitMethod);
```

在这个头文件中，我们声明了UI界面中对应的文本变量和可编辑文本Commit时候调用的函数。OnInput_PlayerNameCommitted函数的第二个参数ETextCommit的作用如下：

- ETextCommit::Default：默认提交方式，通常用于框架内部定义的默认行为。
- ETextCommit::OnEnter：用户按下 Enter（回车键） 提交文本。
- ETextCommit::OnUserMovedFocus：用户通过键盘或鼠标改变了输入焦点，从而完成文本提交。
- ETextCommit::OnCleared：用户清空了文本输入框的内容，可能是通过删除操作或点击清除按钮触发的提交。

我们可以用switch（CommitMethod）{case ETextCommit::Default:...}的方式对相应的提交方式做出不同的响应。

```
#include "Components/EditableText.h"
void UMainMenuUserWidget::NativeOnInitialized()
{
    Super::NativeOnInitialized();
    if (Bttn_QuitGame) {...}
    if (Bttn_SoloGame) {...}
    if (Input_PlayerName) {
        Input_PlayerName->OnTextCommitted.AddDynamic(this,
&UMainMenuUserWidget::OnInput_PlayerNameCommitted);
    }
}
void UMainMenuUserWidget::OnInput_PlayerNameCommitted(const FText& Text,
ETextCommit::Type CommitMethod)
{
    MyGameInstance->OnPlayerNameChanged(Text);
}
```

接下来，我们需要完成这个在TutorialGameInstance里面的OnPlayerNameChanged函数。同样地，我们需要先声明一个带有一个参数的动态多播委托。

```
DECLARE_DYNAMIC_MULTICAST_DELEGATE_OneParam(FChange_PlayerNameEventDelegate,
FText, PlayerName);
```

```
public:
    UPROPERTY(BlueprintAssignable, Category = "Events")
    FChange_PlayerNameEventDelegate ChangePlayerNameEvent;

    UFUNCTION()
    void OnPlayerNameChanged(FText PlayerName);

    void TriggerPlayerNameChangedEvent(FText PlayerName);
```

其中，TriggerPlayerNameChangedEvent用于给他人调用，OnPlayerNameChanged是绑定在ChangePlayerNameEvent委托上的函数。具体的实现如下：

```
void UTutorialGameInstance::Init()
{
    Super::Init();
    OnStartSoloGameEventTriggered.AddDynamic(this,
&UTutorialGameInstance::StartSoloGameEvent);

    ChangePlayerNameEvent.AddDynamic(this,
&UTutorialGameInstance::OnPlayerNameChanged);
}
void UTutorialGameInstance::TriggerPlayerNameChangedEvent(FText PlayerName)
{
    ChangePlayerNameEvent.Broadcast(PlayerName);
}
void UTutorialGameInstance::OnPlayerNameChanged(FText PlayerName)
{
    PlayerProfileinfo.PlayerName = PlayerName;
    SaveProfile();
}
```

我们在Init初始化函数中添加了对OnPlayerNameChanged的绑定，这个函数的逻辑是更改我们的显示信息，然后进行保存。TriggerPlayerNameChangedEvent函数的作用是实现绑定的所有函数。

接下来，我们需要添加我们在UI_ShowMainMenu中的逻辑。

```
void UTutorialGameInstance::UI_ShowMainMenu()
{
    if (!IsValid(MainMenu)) {...}
    MainMenu->AddToViewport(0);
    if (APlayerController* PlayerController = GetWorld()-
>GetFirstPlayerController()) {...}

    CheckForSavedProfile();
    MainMenu->Input_PlayerName->SetText(PlayerProfileinfo.PlayerName);

}
```

我们在显示的时候需要先进行CheckForSavedProfile逻辑，读取我们存档中的信息。然后更新到我们的文本显示当中。

至此，我们就完成了添加游戏保存功能的实现。进行编译保存，我们每次运行的时候都会加载到上次我们保存的玩家信息。

1.2.7总结

在这一小节中，我们拥有了自己的主菜单界面。能够进行交互，切换关卡，退出游戏，还能够保存玩家数据。本小节的代码在https://github.com/Jiejie-UE/UE5CplusplusTutorial仓库中。