1.6 无缝旅行传递信息和关卡UI

1.6.1 处理新玩家加入游戏

(1) 设置出生点

在关卡蓝图中，我们按住Alt复制4个玩家出生点出来，如图1.6.1.1所示。然后，我们点进出生点的Details面板中，设置Object/PlayerstartTag分别为0，1，2，3表示连接进来的玩家的出生位置，如图1.6.1.2所示。

图1.6.1.1 复制出生点

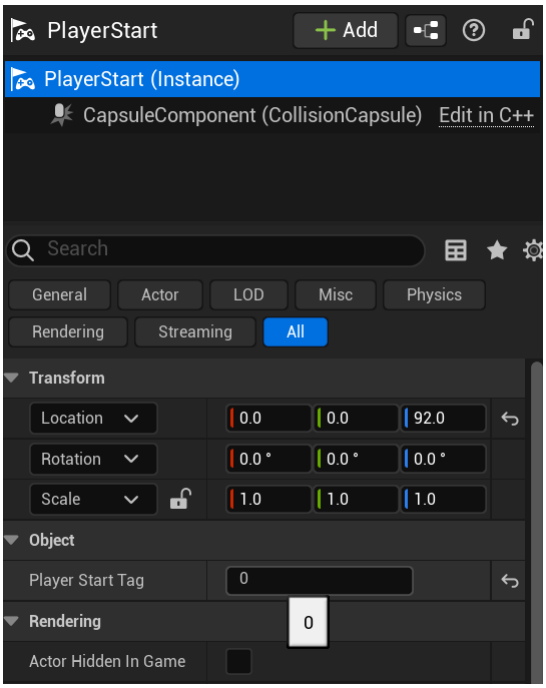

图1.6.1.2 设置出生点

(2) GameMode处理无缝旅行载入关卡

GameMode仅存在于服务器端，每个level都有一个GameMode，我们打开Level01对应GameMode的C++类（在之前的章节中我们将它命名为TutorialGameMode ）。在TutorialGameMode中我们可以重写一个函数HandleStartingNewPlayer，这个函数在无缝旅行时会被触发。

这个HandleStartingNewPlayer函数和PostLogin的区别是，PostLogin只会在连接进来的时候触发，而HandleStartingNewPlayer会在无缝旅行的时候再次触发，而PostLogin不会再次触发。

TutorialGameMode.h

```
virtual void HandleStartingNewPlayer_Implementation(APlayerController* NewPlayer)
override;

TArray<AShooterPlayerController*> All_ConnectedControllers;

UPROPERTY()
int32 Num_ExpectedPlayer;
```

在上面的定义中All_ConnectedControllers是已经连接进来的玩家Controller。Num_ExpectedPlayer是玩家数量。

TutorialGameMode.cpp

```
void ATutorialGameMode::HandleStartingNewPlayer_Implementation(APlayerController*
NewPlayer)
{
    Super::HandleStartingNewPlayer_Implementation(NewPlayer);
    AShooterPlayerController* NewController = Cast<AShooterPlayerController>
(NewPlayer);
    if (NewController) {
        All_ConnectedControllers.Add(NewController);
    }
    UTutorialGameInstance* MyGameInstance = Cast<UTutorialGameInstance>
(GetGameInstance());
    if (MyGameInstance) {
        if (MyGameInstance->IsSoloGame) {
            Num_ExpectedPlayer = 1;

RestartPlayerAtPlayerStart(All_ConnectedControllers[0],FindPlayerStart(All_Conne
ctedControllers[0]));
        }
        else {
            Num_ExpectedPlayer = MyGameInstance->MP_NumConnectedPlayers;
            if (Num_ExpectedPlayer == All_ConnectedControllers.Num()) {
                GetWorld()->GetTimerManager().SetTimer(LevelStartTimerHandle,
this, &ATutorialGameMode::HandleStartingNewPlayerDelay1, 0.5f, false);
            }
        }
    }
}
```

首先，当新玩家加入时，我们会将它添加到GameMode上的All_ConnectedControllers变量中。我们将在GameInstance上存储的MP_NumConnectedPlayers存储在GameMode的Num_ExpectedPlayer中。然后，我们判断是否是SoloGame，如果是SoloGAme那么我们直接设置Num_ExpectedPlayer为1，调用RestartPlayerAtPlayerStart重设角色位置到出生点。

如果是多人模式的话，我们将会等待所有玩家都载入地图，这个时候才开始重设角色位置。如果每个人都连接上了，我们将会等待大概半秒钟的响应时间。这个时候我们需要用到我们的计时器FTimerHandle来进行计时，当到指定时间后调用对应函数，可以选择是否重复调用。

TutorialGameMode.h

```
UPROPERTY()
FTimerHandle LevelStartTimerHandle;
UFUNCTION()
void HandleStartingNewPlayerDelay1();
```

我们声明出需要调用的函数，以及计时器。然后我们调用GetWorld()->GetTimerManager().SetTimer(LevelStartTimerHandle, this, &ATutorialGameMode::HandleStartingNewPlayerDelay1, 0.5f, false);进行计时。第一个参数表示我们的计时器。第二个参数表示当前对象的指针，表示定时器到期时要调用的函数。第三个参数表示定时器到期会执行的函数。第四个参数表示定时器的延迟时间。第五个参数表示循环时间。

TutorialGameMode.cpp

```
void ATutorialGameMode::HandleStartingNewPlayerDelay1()
{
    for (int32 i = 0; i < All_ConnectedControllers.Num(); ++i) {
        RestartPlayerAtPlayerStart(All_ConnectedControllers[i],
FindPlayerStart(All_ConnectedControllers[i]));
    }
}
```

我们会在这个函数中定义这个内容，遍历所有的控制器，然后重新设置位置到出生点。

1.6.2 制作关卡中界面UI（倒计时）

（1）绘制界面UI

首先，我们创建一个WB_IngameHUD的蓝图类，同时创建一个继承于UserWidget的蓝图重命名为InGameHUDUserWidget，并将其作为WB_IngameHUD的父类。最终我们会绘制出如图1.6.2.1所示的内容。

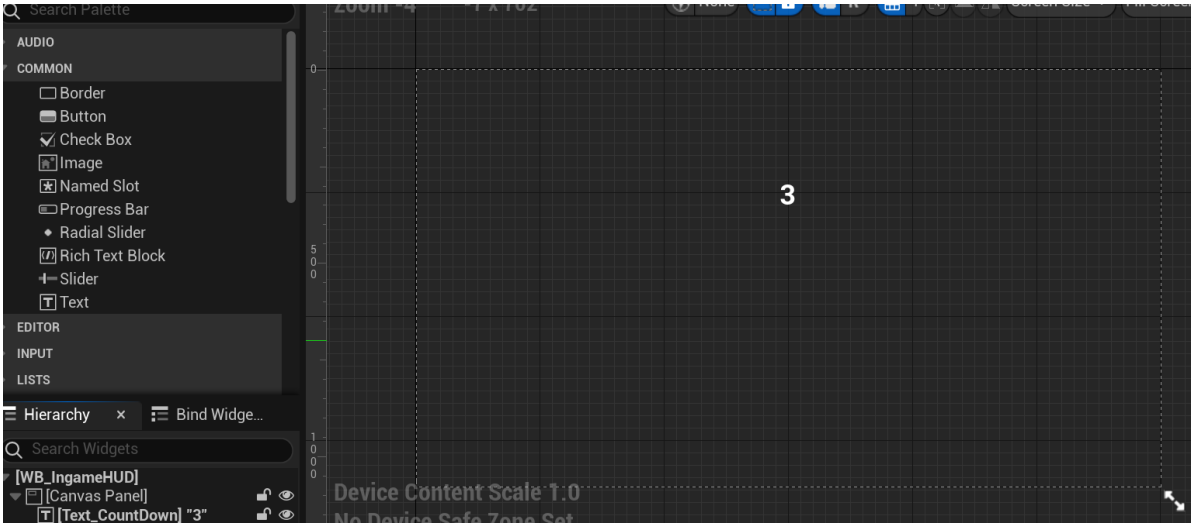

图1.6.2.1 关卡界面

暂时我们不需要非常华丽的界面，我们拖入一个Canvas和一个Text进入即可。

并且，我们需要在GameInstance中添加一个需要绑定的参数。

TutorialGameInstance.h

```
UPROPERTY(EditAnywhere, Category = "UI")
TSubclassOf<UInGameHUDUserWidget> InGameHUDWidgetClass;
```

（2）用计时器编写GameStartCountDown倒计时函数

接下来，我们会编写一个函数用于倒计时。

InGameHUDUserWidget.h

```
UFUNCTION()
void GameStartCountDown();

UPROPERTY(BlueprintReadOnly, meta = (BindWidgetOptional))
UTextBlock* Text_CountDown;

UFUNCTION()
void CountDownDelayCount();

UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Countdown")
int32 CountdownValue = 3;

UPROPERTY()
FTimerHandle CountDownTimerHandle;
```

在头文件中，我们会声明我们的计时器CountDownTimerHandle，并且定义倒计时的值，这里设置为3，声明我们的在蓝图中需要绑定的变量Text_CountDown。

InGameHUDUserWidget.cpp

```
void UInGameHUDUserWidget::GameStartCountDown()
{
    if (Text_CountDown) {
        Text_CountDown-
>SetText(FText::FromString(FString::FromInt(CountdownValue)));
        Text_CountDown->SetVisibility(ESlateVisibility::Visible);

        GetWorld()->GetTimerManager().SetTimer(CountDownTimerHandle, this,
&UInGameHUDUserWidget::CountDownDelayCount, 1.0f, true);
    }
}

void UInGameHUDUserWidget::CountDownDelayCount()
{
    if (CountdownValue > 0) {
        //3,2,1
        CountdownValue--;
        Text_CountDown-
>SetText(FText::FromString(FString::FromInt(CountdownValue)));
        return;
    }
    else if (CountdownValue == 0) {
        //0
        // 这里会让这个Round 1显示3秒
        CountdownValue--;
        Text_CountDown->SetText(FText::FromString(TEXT("Round 1")));
```

```
        return;
    }
    else if (CountdownValue >= -2) {
        //-1
        CountdownValue--;
        return;
    }
    else if (CountdownValue == -3)
    {
        GetWorld()->GetTimerManager().ClearTimer(CountDownTimerHandle);
        Text_CountDown->SetVisibility(ESlateVisibility::Hidden);
        return;
    }
}
```

调用GameStartCountDown函数时，我们会先设置文本内容为CountdownValue表示需要倒计时的值，然后设置为可见Visible。调用计时器设置1s后调用CountDownDelayCount，并且进行循环调用。在计时器调用的函数CountDownDelayCount中，我们会判断CountdownValue的值，并将CountdownValue值进行自减，然后修改文本内容，如果倒计时达到某个条件后我们可以调用ClearTimer进行定时器的关闭。

(3) 编写UI_ShowInGameHUD函数

在ShooterPlayerController中我们编写这个函数。

AShooterPlayerController.h

```
void UI_ShowInGameHUD();

UPROPERTY()
UInGameHUDUserWidget* InGameHUD;
```

AShooterPlayerController.cpp

```
void AShooterPlayerController::UI_ShowInGameHUD()
{
    UTutorialGameInstance* MyGameInstance = Cast<UTutorialGameInstance>
(GetGameInstance());
    if (MyGameInstance) {
        if (!IsValid(InGameHUD)) {
            InGameHUD = CreateWidget<UInGameHUDUserWidget>(GetWorld(),
MyGameInstance->InGameHUDWidgetClass);
        }
        if (InGameHUD) {
            InGameHUD->AddToViewport(0);
        }
    }
}
```

在这个函数中，我们创建一个刚才创建的UI。然后，我们通过AddToViewport添加到屏幕上。然后，我们在ShooterPlayerController的BeginPlay中调用这个函数，在游戏开始的时候我们就创建这个UI。

AShooterPlayerController.cpp

```
void AShooterPlayerController::BeginPlay()
{
    Super::BeginPlay();
    if (IsLocalController()) {
        UI_ShowInGameHUD();
        FInputModeGameOnly InputMode;
        SetInputMode(InputMode);
        bShowMouseCursor = false;
    }
}
```

（4）在倒计时中禁止输入

我们接下来会override一个函数PossessedBy。这个函数仅在Pawn服务端会被调用，当有一个Pawn被Controller占有时调用。

TutorialCharacter.h

```
UFUNCTION()
virtual void PossessedBy(AController* NewController) override;
```

TutorialCharacter.cpp

```
void ATutorialCharacter::PossessedBy(AController* NewController)
{
    Super::PossessedBy(NewController);
    APlayerController* PlayerController = Cast<APlayerController>
(NewController);
    if (PlayerController) {
        DisableAllInput(PlayerController);
    }
}
```

我们在这个函数中，我们会将所有的输入进行禁止，当游戏开始的时候（因为这个时候是倒计时）。我们需要通过RPC转发给客户端进行输入的禁止。

TutorialCharacter.h

```
UFUNCTION(Client, Reliable, Category = "Lobby")
void DisableAllInput(APlayerController* PlayerController);
void DisableAllInput_Implementation(APlayerController* PlayerController);
```

TutorialCharacter.cpp

```
void ATutorialCharacter::DisableAllInput_Implementation(APlayerController*
PlayerController)
{
    DisableInput(PlayerController);
}
```

这个DisableInput函数会禁止玩家进行移动。

（5）在倒计时结束后启用输入

在倒计时结束后，我们在ShooterPlayerController中启用对应的角色输入。

ShooterPlayerController.h

```
UFUNCTION(Client, Reliable, Category = "Lobby")
void EnableAllInput();
void EnableAllInput_Implementation();
```

ShooterPlayerController.cpp

```
void AShooterPlayerController::EnableAllInput_Implementation()
{
    APawn* ControlledPawn = GetPawn();
    if (ControlledPawn)
    {
        ATutorialCharacter* MyCharacter = Cast<ATutorialCharacter>
(ControlledPawn);
        if (MyCharacter)
        {
            MyCharacter->EnableInput(this);
        }
    }
}
```

Controller在服务器和客户端均有，我们需要将Controller转发到客户端进行实现。在客户端我们调用EnableInput函数启用输入即可。

（6）将GameStartCountDown转发到客户端

我们在ShooterPlayerController创建一个函数将GameStartCountDown转发到客户端。将来我们会在GameMode中调用Controller->GameStartCountDown，因此这个函数在服务端调用的，我们需要转发到客户端进行。

ShooterPlayerController.h

```
UFUNCTION(Client, Reliable, Category = "Lobby")
void GameStartCountdown();
void GameStartCountdown_Implementation();
```

ShooterPlayerController.cpp

```
void AShooterPlayerController::GameStartCountdown_Implementation()
{
    if (InGameHUD) {
        InGameHUD->GameStartCountDown();
    }
}
```

我们会调用UInGameHUDUserWidget中的GameStartCountDown函数，进行开始倒计时。

（7）编写LevelStart函数

至此，我们需要创建一个函数来开始这一切，开始这个倒计时。

TutorialGameMode.h

```
UFUNCTION()
void LevelStart();
UFUNCTION()
void LevelStartDelay1();
UFUNCTION()
void LevelStartDelay2();
```

TutorialGameMode.cpp

```
void ATutorialGameMode::LevelStart()
{
    GetWorld()->GetTimerManager().SetTimer(LevelStartTimerHandle, this,
&ATutorialGameMode::LevelStartDelay1, 0.2f, false);
}
void ATutorialGameMode::LevelStartDelay1()
{
    for (int32 i = 0; i < All_ConnectedControllers.Num(); ++i) {
        All_ConnectedControllers[i]->GameStartCountdown();
    }

    GetWorld()->GetTimerManager().SetTimer(LevelStartTimerHandle, this,
&ATutorialGameMode::LevelStartDelay2, 3.0f, false);
}
void ATutorialGameMode::LevelStartDelay2()
{
    for (int32 i = 0; i < All_ConnectedControllers.Num(); ++i) {
        All_ConnectedControllers[i]->EnableAllInput();
    }
}
```

我们会在关卡开始的时候调用这个函数，我们会有一个小的延时0.2s，已备大部分东西准备完毕。然后我们调用LevelStartDelay1开始计时，进行倒计时UI的显示。然后再等待3s，让所有玩家恢复角色控制。

（8）调用LevelStart函数

我们在HandleStartingNewPlayer中调用这个函数即可。当我们玩家载入之后，就开始这个LevelStart函数的调用。与此同时在

服务端也会调用PossessedBy函数，让新加入的玩家实现关闭输入。就完成了整个关闭输入，倒计时UI显示，启用输入的流程。

TutorialGameMode.cpp

```
void ATutorialGameMode::HandleStartingNewPlayer_Implementation(APlayerController*
NewPlayer)
{
    Super::HandleStartingNewPlayer_Implementation(NewPlayer);
    AShooterPlayerController* NewController = Cast<AShooterPlayerController>
(NewPlayer);
    if (NewController) {
        All_ConnectedControllers.Add(NewController);
    }
    UTutorialGameInstance* MyGameInstance = Cast<UTutorialGameInstance>
(GetGameInstance());
    if (MyGameInstance) {
        if (MyGameInstance->IsSoloGame) {
```

```
            Num_ExpectedPlayer = 1;

RestartPlayerAtPlayerStart(All_ConnectedControllers[0],FindPlayerStart(All_Conne
ctedControllers[0]));
            LevelStart();//调用LevelStart()
        }
        else {
            Num_ExpectedPlayer = MyGameInstance->MP_NumConnectedPlayers;
            if (Num_ExpectedPlayer == All_ConnectedControllers.Num()) {
                GetWorld()->GetTimerManager().SetTimer(LevelStartTimerHandle,
this, &ATutorialGameMode::HandleStartingNewPlayerDelay1, 0.5f, false);
            }
        }
    }
}
void ATutorialGameMode::HandleStartingNewPlayerDelay1()
{
    for (int32 i = 0; i < All_ConnectedControllers.Num(); ++i) {
        RestartPlayerAtPlayerStart(All_ConnectedControllers[i],
FindPlayerStart(All_ConnectedControllers[i]));
    }
    LevelStart();//调用LevelStart()
}
```

在这两个函数添加调用LevelStart即可，分别在SoloGame和LobbyGame中实现了这个逻辑。

1.6.3 Esc暂停界面

（1）制作Esc蓝图

我们创建一个继承于UserWidget的蓝图界面重命名为PauseMenuUserWidget，复制我们的WB_MainMenu蓝图重名为WB_PauseMenu，设置父类为PauseMenuUserWidget。并修改整个界面为如图1.6.3.1所示的样子。

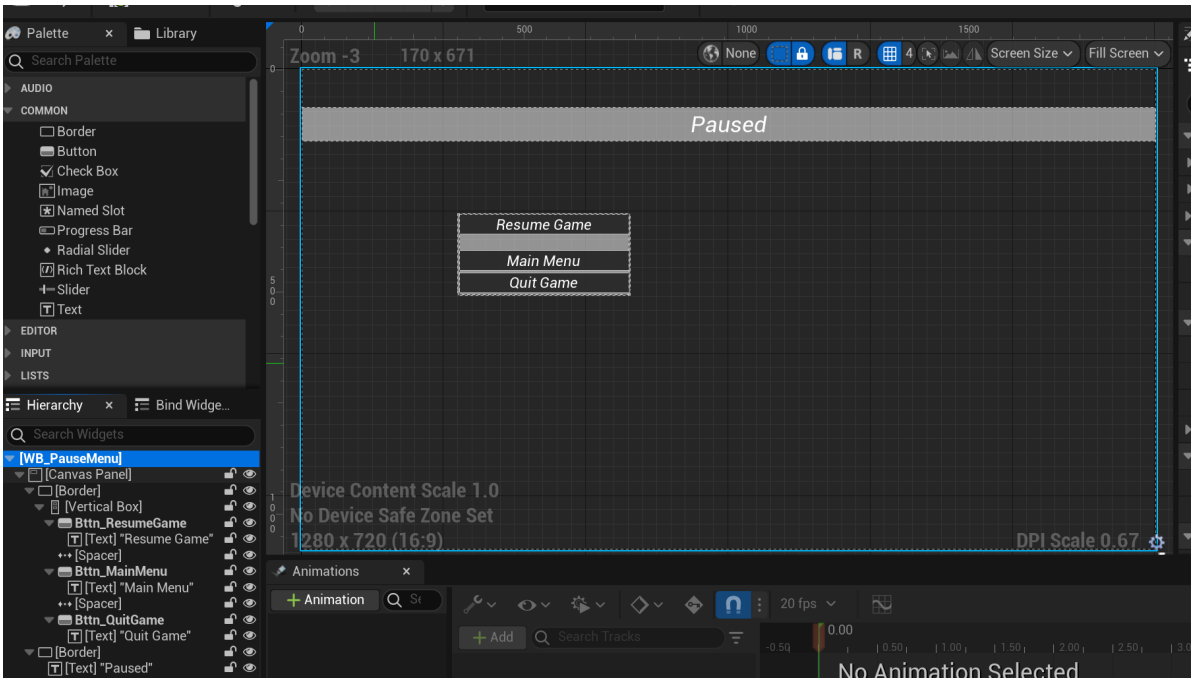

图1.6.3.1 PauseMenu界面

我们仅需要删除MainMenu中的不想关组件，然后修改按钮名称和文本内容即可。

然后，我们需要在GameInstance中添加这个蓝图的类。

TutorialGameInstance.h

```
UPROPERTY(EditAnywhere, Category = "UI")
TSubclassOf<UPauseMenuUserWidget> PauseMenuWidgetClass;
```

（2）创建UI_ShowPauseMenu函数

将来我们会在GameInstance中调用这个函数，因此我们在TutorialGameInstance中创建对应的声明和定义。

TutorialGameInstance.h

```
UFUNCTION()
void UI_ShowPauseMenu();

UPROPERTY()
UPauseMenuUserWidget* PauseMenu;
```

TutorialGameInstance.cpp

```
void UTutorialGameInstance::UI_ShowPauseMenu()
{
    if (!IsValid(PauseMenu)) {
        PauseMenu = CreateWidget<UPauseMenuUserWidget>(GetWorld(),
PauseMenuWidgetClass);
    }
    PauseMenu->AddToViewport(0);
    if (APlayerController* PlayerController = GetWorld()-
>GetFirstPlayerController()) {
        FInputModeUIOnly InputMode;
        PlayerController->SetInputMode(InputMode);
        PlayerController->bShowMouseCursor = true;
    }
}
```

在这个函数中，我们创建一个PauseMenu，然后调用AddToViewport添加到屏幕上。然后设置为仅UI交互，可供用户点击。

（3）创建TogglePauseGame函数

TogglePauseGame这个函数将会切换暂停游戏的状态。从暂停切换到继续游戏，从继续游戏切换到暂停。这个函数还会返回切换后的游戏状态，是暂停或者继续。

TutorialGameInstance.h

```
UFUNCTION()
bool TogglePauseGame();

UPROPERTY()
bool MP_IsPauseMenuVisable = false;
```

其中，MP_IsPauseMenuVisable变量是用来判断多人游戏下是否处于暂停状态的。

TutorialGameInstance.cpp

```
bool UTutorialGameInstance::TogglePauseGame()
```

```
{
    bool IsGamePausedResult = false;
    if (IsSoloGame) {
    //SoloGame
        if (UGameplayStatics::IsGamePaused(GetWorld())) {
            UGameplayStatics::SetGamePaused(GetWorld(), false);
            IsGamePausedResult = false;
            PauseMenu->RemoveFromParent();
            if (APlayerController* PlayerController = GetWorld()-
>GetFirstPlayerController()) {
                FInputModeGameOnly InputMode;
                PlayerController->SetInputMode(InputMode);
                PlayerController->bShowMouseCursor = false;
                AShooterPlayerController* ShooterPlayerController =
Cast<AShooterPlayerController>(PlayerController);
                if (ShooterPlayerController) {
                    ShooterPlayerController->UnHide_IngameHUD();
                }
            }
        }
        else {
            UGameplayStatics::SetGamePaused(GetWorld(),true);
            IsGamePausedResult = true;
            UI_ShowPauseMenu();
        }
    }
    else {
    //Lobby
        if (MP_IsPauseMenuVisable) {
            MP_IsPauseMenuVisable = false;
            IsGamePausedResult = false;
            PauseMenu->RemoveFromParent();
            if (APlayerController* PlayerController = GetWorld()-
>GetFirstPlayerController()) {
                FInputModeGameOnly InputMode;
                PlayerController->SetInputMode(InputMode);
                PlayerController->bShowMouseCursor = false;
                AShooterPlayerController* ShooterPlayerController =
Cast<AShooterPlayerController>(PlayerController);
                if (ShooterPlayerController) {
                    ShooterPlayerController->UnHide_IngameHUD();
                }
            }
        }
        else {
            MP_IsPauseMenuVisable = true;
            IsGamePausedResult = true;
            UI_ShowPauseMenu();
        }
    }

    return IsGamePausedResult;
}
```

如果在IsSoloGame中，我们会直接调用UGameplayStatics::IsGamePaused来获取当前的暂停状态，然后设置暂停状态为相反的状态，同时修改我们返回值为对应状态。如果是切换到暂停状态，我们会调用UI_ShowPauseMenu显示暂停界面。如果切换到继续状态，我们会移除暂停界面UI并重新设置输入方式，并调用UnHide_IngameHUD（下文会给出实现）显示原来就有的UI界面（这里暂时是倒计时）。

如果在LobbyGame中，大致流程也类似。不同的是，我们用自定义MP_IsPauseMenuVisable变量来判断是否处于暂停状态。

然后，我们在ShooterPlayerController实现之前的UnHide_IngameHUD函数。

ShooterPlayerController.h

```
UFUNCTION()
void UnHide_IngameHUD();
```

ShooterPlayerController.cpp

```
void AShooterPlayerController::UnHide_IngameHUD()
{
    InGameHUD->SetVisibility(ESlateVisibility::Visible);
}
```

在这个函数中，我们调用SetVisibility显示UI即可。

（4）创建Esc按键映射

在编辑器中，如图我们在1.1中一样，创建一个InputAction重命名为PauseGame，并如图1.6.3.2所示设置Triggers为Rressed，使得能够按下去在按下第二次之前仅触发一次。我们打开CharacterInputMapping，如图1.6.3.3所示，绑定一个Esc按键即可。

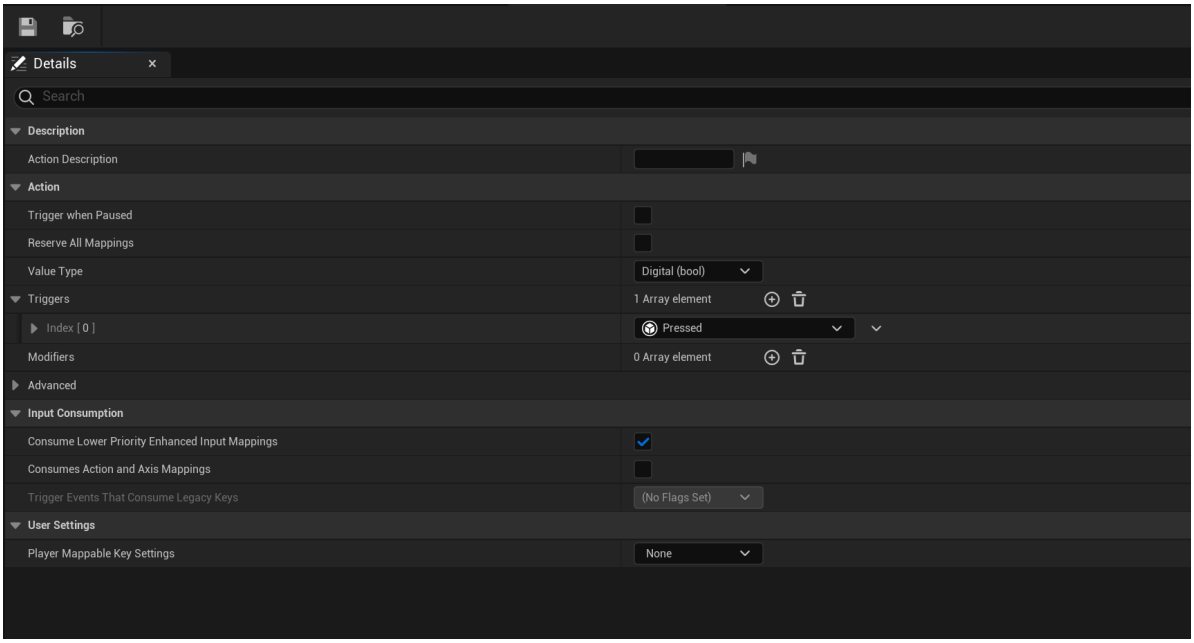

图1.6.3.2 PauseGame的Details面板

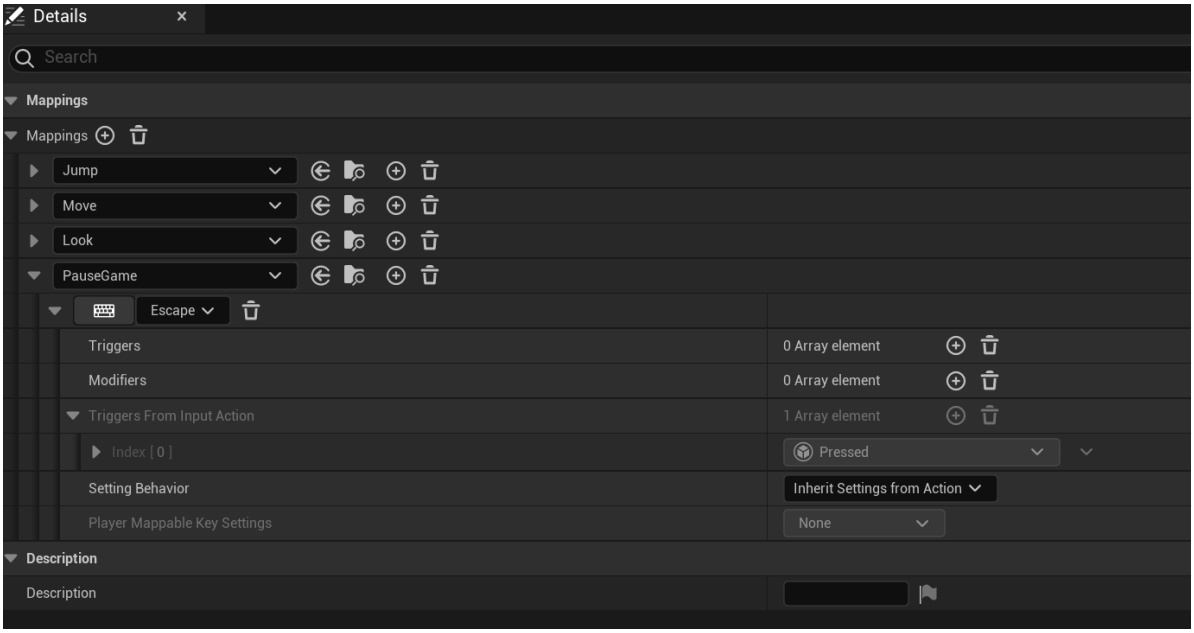

图1.6.3.3 绑定Esc按键

然后，我们在TutorialCharacter类中，添加对应的InputAction和响应函数。

TutorialCharacter.h

```
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Input, meta =
(AllowPrivateAccess = "true"))
UInputAction* PauseGameAction;

void OpenPauseGame();

UPROPERTY()
UInGameHUDUserWidget* InGameHUD;
```

TutorialCharacter.cpp

```
void ATutorialCharacter::SetupPlayerInputComponent(UInputComponent*
PlayerInputComponent)
{
    Super::SetupPlayerInputComponent(PlayerInputComponent);
    if (UEnhancedInputComponent* EnhancedInputComponent =
Cast<UEnhancedInputComponent>(PlayerInputComponent)) {
        EnhancedInputComponent->BindAction(JumpAction, ETriggerEvent::Started,
this, &ACharacter::Jump);
        EnhancedInputComponent->BindAction(JumpAction, ETriggerEvent::Completed,
this, &ACharacter::StopJumping);

        EnhancedInputComponent->BindAction(MoveAction, ETriggerEvent::Triggered,
this, &ATutorialCharacter::Move);
        EnhancedInputComponent->BindAction(LookAction, ETriggerEvent::Triggered,
this, &ATutorialCharacter::Look);

        EnhancedInputComponent->BindAction(PauseGameAction,
ETriggerEvent::Triggered, this, &ATutorialCharacter::OpenPauseGame);//绑定
PauseGameAction
    }
    BindGunInputMappingContext();
}
```

```
void ATutorialCharacter::OpenPauseGame()
{
    UTutorialGameInstance* MyGameInstance = Cast<UTutorialGameInstance>
(GetGameInstance());
    if (MyGameInstance && InGameHUD) {
        if (MyGameInstance->TogglePauseGame()) {
            InGameHUD->SetVisibility(ESlateVisibility::Hidden);
        }
        else {
            InGameHUD->SetVisibility(ESlateVisibility::Visible);
        }
    }
}
```

在头文件中的PauseGameAction记得在编辑器中进行绑定。我们在SetupPlayerInputComponent中补充上刚刚添加的绑定PauseGameAction，并实现对应的响应函数。在OpenPauseGame中，调用切换暂停状态函数TogglePauseGame，并且按照对应的状态设置InGameHUD是否显示，当显示暂停菜单时，原来界面上的UI应该进行隐藏。

在此之前，我们需要获得InGameHUD变量，我们在BeginPlay中获得这个变量。

TutorialCharacter.h

```
UPROPERTY()
FTimerHandle BeginPlayTimerHandle;
UFUNCTION()
void BeginPlayDelay1();
```

TutorialCharacter.cpp

```
void ATutorialCharacter::BeginPlay()
{
    Super::BeginPlay();
    if (APlayerController* PlayerController = Cast<APlayerController>
(Controller)) {
        if (UEnhancedInputLocalPlayerSubsystem* Subsystem =
ULocalPlayer::GetSubsystem<UEnhancedInputLocalPlayerSubsystem>(PlayerController-
>GetLocalPlayer())) {
            Subsystem->AddMappingContext(InputMappingContext, 0);
        }
    }
    bAiming = false;

    GetWorld()->GetTimerManager().SetTimer(BeginPlayTimerHandle, this,
&ATutorialCharacter::BeginPlayDelay1, 2.0f, false);//设置计时器
}
void ATutorialCharacter::BeginPlayDelay1()
{
    if (IsLocallyControlled()) {
        AShooterPlayerController* PlayerController =
Cast<AShooterPlayerController>(GetController());
        if (PlayerController) {
            InGameHUD = PlayerController->InGameHUD;
        }
    }
}
```

我们在游戏开始时有3s倒计时，因此我们可以等待2s才调用BeginPlayDelay1来获取InGameHUD变量的引用。

(5) Bttn_ResumeGame按键响应函数

PauseMenuUserWidget.h

```
public:
    virtual void NativeOnInitialized() override;
public:
    UPROPERTY(BlueprintReadOnly, meta = (BindWidgetOptional))
    UButton* Bttn_ResumeGame;
    UPROPERTY(BlueprintReadOnly, meta = (BindWidgetOptional))
    UButton* Bttn_MainMenu;
    UPROPERTY(BlueprintReadOnly, meta = (BindWidgetOptional))
    UButton* Bttn_QuitGame;
    UFUNCTION()
    void On_Bttn_ResumeGameClick();
    UFUNCTION()
    void On_Bttn_MainMenuClick();
    UFUNCTION()
    void Bttn_QuitGameClick();
```

PauseMenuUserWidget.cpp

```
void UPauseMenuUserWidget::NativeOnInitialized()
{
    Super::NativeOnInitialized();
    if (Bttn_ResumeGame) {
        Bttn_ResumeGame->OnClicked.AddDynamic(this,
&UPauseMenuUserWidget::On_Bttn_ResumeGameClick);
    }
    if (Bttn_MainMenu) {
        Bttn_MainMenu->OnClicked.AddDynamic(this,
&UPauseMenuUserWidget::On_Bttn_MainMenuClick);
    }
    if (Bttn_QuitGame) {
        Bttn_QuitGame->OnClicked.AddDynamic(this,
&UPauseMenuUserWidget::Bttn_QuitGameClick);
    }
}
```

首先，我们在NativeOnInitialized绑定对应的按键响应函数。接下来我们会一一实现对应的函数。

PauseMenuUserWidget.cpp

```
void UPauseMenuUserWidget::On_Bttn_ResumeGameClick()
{
    UTutorialGameInstance* MyGameInstance = Cast<UTutorialGameInstance>
(GetGameInstance());
    if (MyGameInstance) {
        MyGameInstance->TogglePauseGame();
    }
}
```

点击Bttn_ResumeGame的时候，我们会切换暂停状态，这地方是从暂停切换到继续。

（6）Bttn_MainMenu按键响应函数

当我们点击Bttn_MainMenu的时候，会执行如下函数。

PauseMenuUserWidget.cpp

```
void UPauseMenuUserWidget::On_Bttn_MainMenuClick()
{
    //这是在多人下的逻辑，单人需要额外写，单人不需要DestroyClientSession
    if (MyController && MyController->HasAuthority()) {
        //OnServer
        ATutorialGameMode* GameMode = Cast<ATutorialGameMode>
(UGameplayStatics::GetGameMode(GetWorld()));
        if (GameMode) {
            for (AShooterPlayerController* Controller : GameMode-
>All_ConnectedControllers) {
                if (Controller != MyController) {
                    Controller->DestroyClientSession();
                }
            }
            UTutorialGameInstance* MyGameInstance = Cast<UTutorialGameInstance>
(GetGameInstance());
            if (MyGameInstance) {
                MyGameInstance->MP_IsPauseMenuVisable = false;
            }
            MyController->DestroyClientSession();
        }
    }
    else {
        //OnClient
        if (MyController)
        {
            UTutorialGameInstance* MyGameInstance = Cast<UTutorialGameInstance>
(GetGameInstance());
            if (MyGameInstance) {
                MyGameInstance->MP_IsPauseMenuVisable = false;
            }
            MyController->DestroyClientSession();
        }
    }
}
```

这里仅写多人的逻辑。像我们在大厅界面一般，如果我们是服务器，那么我们销毁客户端会话，之后才销毁服务端会话。并且设置MP_IsPauseMenuVisable为false。如果是客户端的话，我们直接设置MP_IsPauseMenuVisable为false，然后销毁自身会话即可。其中，我们需要编写对应的DestroyClientSession逻辑。

ShooterPlayerController.h

```
UFUNCTION(Client, Reliable, Category = "Lobby")
void DestroyClientSession();
void DestroyClientSession_Implementation();
```

ShooterPlayerController.cpp

```
void AShooterPlayerController::DestroyClientSession_Implementation()
{
    UTutorialGameInstance* MyGameInstance = Cast<UTutorialGameInstance>
(GetGameInstance());
    if (MyGameInstance->CurrentSessionName.IsNone()) {
        return;
    }
    if (MyGameInstance->SessionInterface.IsValid()) {
        MyGameInstance->SessionInterface->DestroySession(MyGameInstance-
>CurrentSessionName);
    }
}
```

那么当我们调用DestroyClientSession函数时，就会通过RPC转发给客户端销毁对应的会话。

（6）Bttn_QuitGame按键响应函数

当我们按下Bttn_QuitGame的时候，我们执行的逻辑和Bttn_MainMenu很像。只不过我们不重新回到主菜单了，我们选择直接退出游戏，这个时候我们在TutorialGameInstance中维护一个变量即可。

TutorialGameInstance.h

```
UPROPERTY()
bool WillQuitGame = false;
```

TutorialGameInstance.cpp

```
void UPauseMenuUserWidget::Bttn_QuitGameClick()
{
    //这是在多人下的逻辑，单人需要额外写，单人不需要DestroyClientSession
    if (MyController && MyController->HasAuthority()) {
        //OnServer
        ATutorialGameMode* GameMode = Cast<ATutorialGameMode>
(UGameplayStatics::GetGameMode(GetWorld()));
        if (GameMode) {
            for (AShooterPlayerController* Controller : GameMode-
>All_ConnectedControllers) {
                if (Controller != MyController) {
                    Controller->DestroyClientSession();
                }
            }
            UTutorialGameInstance* MyGameInstance = Cast<UTutorialGameInstance>
(GetGameInstance());
            if (MyGameInstance) {
                MyGameInstance->MP_IsPauseMenuVisable = false;
                MyGameInstance->WillQuitGame = true;
            }
            MyController->DestroyClientSession();
        }
    }
    else {
        //OnClient
        if (MyController)
        {
            UTutorialGameInstance* MyGameInstance = Cast<UTutorialGameInstance>
(GetGameInstance());
            if (MyGameInstance) {
```

```
                MyGameInstance->MP_IsPauseMenuVisable = false;
                MyGameInstance->WillQuitGame = true;
            }
            MyController->DestroyClientSession();
        }
    }
}
```

当我们点击Bttn_QuitGame，和另一个按钮不同之处在于这里我们会让这个WillQuitGame变量变为true表示将会退出游戏。之后我们修改TutorialGameInstance中的OnDestroySessionComplete函数，在这个函数中添加对应的判断即可。如果WillQuitGame为真，那么调用QuitGame退出游戏。否则切换到主菜单。

```
void UTutorialGameInstance::OnDestroySessionComplete(FName SessionName, bool
bWasSuccessful)
{
    if (bWasSuccessful) {
        CurrentSessionName = FName();
        if (WillQuitGame) {
            APlayerController* PlayerController = GetWorld()-
>GetFirstPlayerController();
            if (PlayerController)
            {
                UKismetSystemLibrary::QuitGame(GetWorld(), PlayerController,
EQuitPreference::Quit, true);
            }
        }
        else {
            UGameplayStatics::OpenLevel(this, FName("MainMenu_Map"));
        }
    }
}
```

那么此时，我们就完成了Esc暂停界面的制作。

1.6.4 无缝旅行传递玩家状态

（1）创建PlayerState类

我们创建一个PlayerState的C++类命名为Shooter_PlayerState，然后创建一个对应的子类BP_Shooter_PlayerState。接着点开BP_TutorialGameMode和BP_ShootMenuGameMode设置它们的PlayerStateClass为BP_Shooter_PlayerState，如图1.6.4.1所示。

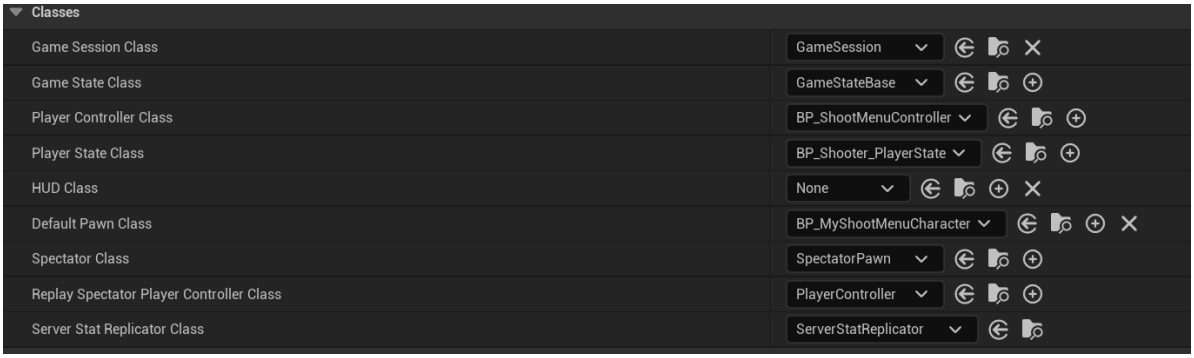

图1.6.4.1 BP_ShootMenuGameMode设置PlayerStateClass

（2）CopyProperties复制属性

在这里，我们通过PlayerState传递我们的PlayerProfile。使用CopyProperties函数，可以实现无缝旅行间通过PlayerState传递信息。

Shooter_PlayerState.h

```
UPROPERTY(Replicated)
FS_PlayerProfile PlayerProfile;

virtual void CopyProperties(APlayerState* PlayerState) override;
```

Shooter_PlayerState.cpp

```
void AShooter_PlayerState::CopyProperties(APlayerState* PlayerState) {
    Super::CopyProperties(PlayerState);
    if (AShooter_PlayerState* ShooterPS = Cast<AShooter_PlayerState>
(PlayerState))
    {
        ShooterPS->PlayerProfile = this->PlayerProfile;
    }
}
```

我们在原来的PlayerState存储的PlayerProfile，我们通过CopyProperties可以传递到新的PlayerState中。我们还需要开启复制，保证这个PlayerProfile变量复制到客户端上面。

Shooter_PlayerState.h

```
virtual void GetLifetimeReplicatedProps(TArray<FLifetimeProperty>&
OutLifetimeProps) const override;
```

Shooter_PlayerState.cpp

```
void AShooter_PlayerState::GetLifetimeReplicatedProps(TArray<FLifetimeProperty>&
OutLifetimeProps) const {
    Super::GetLifetimeReplicatedProps(OutLifetimeProps);
    DOREPLIFETIME(AShooter_PlayerState, PlayerProfile);
}
```

（3）为原来的PlayerState赋值

我们还没有为原来的PlayerState->PlayerProfile进行赋值。因此，我们在ShootMenuGameMode中进行赋值。

ShootMenuGameMode.cpp

```
void AShootMenuGameMode::UpdateLobby(bool PlayerNames, bool LevelSelection)
{
    All_PlayerProfiles.Empty();
    for (int32 i = 0; i < All_ControllerPlayers.Num(); ++i) {
        All_PlayerProfiles.Add(All_ControllerPlayers[i]->PlayerProfileinfo);
        AShooter_PlayerState* PlayerState = All_ControllerPlayers[i]-
>GetPlayerState<AShooter_PlayerState>();
        if (PlayerState) {
            PlayerState->PlayerProfile = All_ControllerPlayers[i]-
>PlayerProfileinfo;
        }
```

```
    }

    if (PlayerNames) {...}
    if (LevelSelection) {...}
}
```

我们修改UpdateLobby函数，当我们更新大厅的时候，我们会对大厅中的所有玩家的PlayerState进行更新并赋值。

（4）修改InGameHUD，添加PlayerName显示

我们传递了玩家信息过来，我们可以添加对应的玩家名称显示在界面上。我们修改InGameHUD如下图1.6.4.2所示。

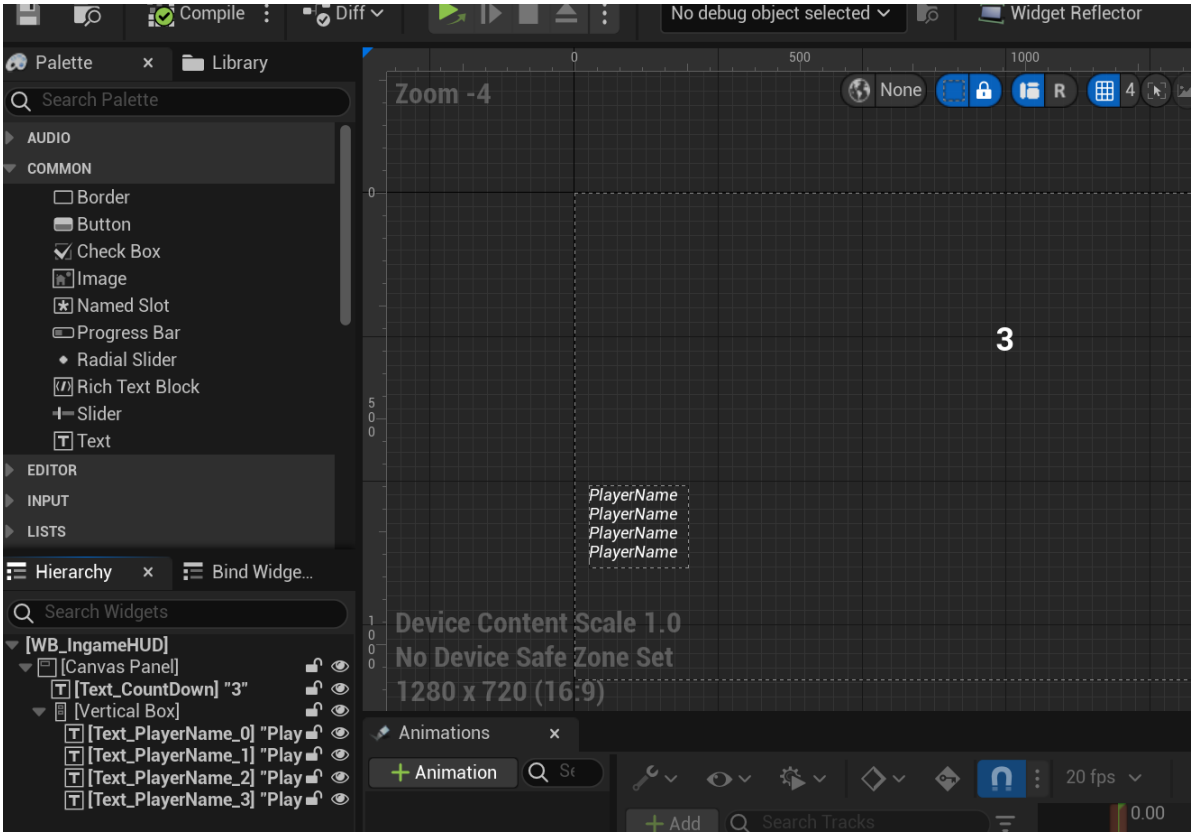

图1.6.4.2 为InGameHUD添加名称显示

拖拽一个垂直框，并创建四个PlayerName进行显示，记得设置为IsVariable。然后我们在NativeOnInitialized将它们合并到一个Array中。

InGameHUDUserWidget.h

```
public:
    virtual void NativeOnInitialized() override;
protected:
    virtual void NativeConstruct() override;
public:
    UPROPERTY(BlueprintReadOnly, meta = (BindWidgetOptional))
    UTextBlock* Text_PlayerName_0;
    UPROPERTY(BlueprintReadOnly, meta = (BindWidgetOptional))
    UTextBlock* Text_PlayerName_1;
    UPROPERTY(BlueprintReadOnly, meta = (BindWidgetOptional))
    UTextBlock* Text_PlayerName_2;
    UPROPERTY(BlueprintReadOnly, meta = (BindWidgetOptional))
    UTextBlock* Text_PlayerName_3;
    UPROPERTY()
```

```
    TArray<UTextBlock*> All_Text_PlayerName;
```

InGameHUDUserWidget.cpp

```
void UInGameHUDUserWidget::NativeOnInitialized()
{
    Super::NativeOnInitialized();
    All_Text_PlayerName.Add(Text_PlayerName_0);
    All_Text_PlayerName.Add(Text_PlayerName_1);
    All_Text_PlayerName.Add(Text_PlayerName_2);
    All_Text_PlayerName.Add(Text_PlayerName_3);
}
void UInGameHUDUserWidget::NativeConstruct()
{
    Super::NativeConstruct();
    GetWorld()->GetTimerManager().SetTimer(NativeConstructTimerHandle, this,
&UInGameHUDUserWidget::UpdateInGameHUD, 2.0f, false);
}
```

然后，我们在NativeConstruct当显示的时候，我们更新我们的玩家信息。由于刚开始载入游戏的时候玩家信息可能没有加载完毕。我们设置2s延时才调用UpdateInGameHUD函数。

InGameHUDUserWidget.h

```
UFUNCTION()
void UpdateInGameHUD();
```

InGameHUDUserWidget.cpp

```
void UInGameHUDUserWidget::UpdateInGameHUD()
{
    if (AGameStateBase* GameState = GetWorld()->GetGameState()) {
        for (int32 i = 0; i < All_Text_PlayerName.Num(); ++i) {
            if (GameState->PlayerArray.Num() - 1 < i) {
                All_Text_PlayerName[i]->RemoveFromParent();
            }
        }
        for (int32 i = 0; i < GameState->PlayerArray.Num(); ++i) {
            AShooter_PlayerState* PlayerState = Cast<AShooter_PlayerState>
(GameState->PlayerArray[i]);
            if (PlayerState) {
                All_Text_PlayerName[i]->SetText(PlayerState-
>PlayerProfile.PlayerName);
            }
        }
    }
}
```

我们声明UpdateInGameHUD函数进行界面的更新或初始化。

（5）玩家退出时处理InGameHUD显示

当玩家退出时，会触发Logout函数。因此我们有如下声明。

TutorialGameMode.h

```
virtual void Logout(AController* Exiting) override;

UPROPERTY()
FTimerHandle LogoutTimerHandle;

UFUNCTION()
void LogoutDelay1();
```

TutorialGameMode.cpp

```
void ATutorialGameMode::Logout(AController* Exiting)
{
    AShooterPlayerController* PlayerController = Cast<AShooterPlayerController>
(Exiting);
    if (PlayerController) {
        All_ConnectedControllers.Remove(PlayerController);
        Num_ExpectedPlayer--;
        GetWorld()->GetTimerManager().SetTimer(LogoutTimerHandle, this,
&ATutorialGameMode::LogoutDelay1, 0.5f, false);
    }
}
void ATutorialGameMode::LogoutDelay1()
{
    for (int32 i = 0; i < All_ConnectedControllers.Num(); ++i) {
        All_ConnectedControllers[i]->UpdateInGameHUD();
    }
}
```

当玩家退出时，我们从All_ConnectedControllers移除对应的控制器。然后等待0.5s，等待退出完毕之后调用LogoutDelay1函数。进行更新，因为UI界面时存在于客户端的。因此我们需要在Controller中编写一个客户端函数。

ShooterPlayerController.h

```
UFUNCTION(Client, Reliable, Category = "Lobby")
void UpdateInGameHUD();
void UpdateInGameHUD_Implementation();
```

ShooterPlayerController.cpp

```
void AShooterPlayerController::UpdateInGameHUD_Implementation()
{
    if (InGameHUD) {
        InGameHUD->UpdateInGameHUD();
    }
}
```

现在，我们就能够在客户端正确的显示玩家状态的刷新了。

1.6.5 总结

至此，我们已经完成了处理新玩家加入游戏，关卡界面UI制作，Esc暂停菜单的制作，玩家状态信息的传递。在这一节中，我们更进一步完善了一些基础的功能。