1.4 新用户加入和退出Session

1.4.1 将null系统改为Steam系统

我们按照以下步骤可以切换到Steam系统。

如果我们尝试运行1.3的那个程序的话，我们会发现，即使运行了3个窗口，两个进程进入同一个会话之后，第三个进程再次搜索会话，发现当前的搜索结果的人数还是1/4，并没有变为我们预期的2/4。这由于null系统的简单性导致的，我们接下来将其切换为Steam系统进行实现。

首先，我们将DefaultEngine.ini文件中添加以下内容，并删除之前添加的那行NULL。

```
[/Script/Engine.GameEngine]
+NetDriverDefinitions=
(DefName="GameNetDriver",DriverClassName="OnlineSubsystemSteam.SteamNetDriver",D
riverClassNameFallback="OnlineSubsystemUtils.IpNetDriver")

[OnlineSubsystem]
DefaultPlatformService=Steam

[OnlineSubsystemSteam]
bEnabled=true
SteamDevAppId=480
GameServerQueryPort=27015
bInitServerOnClient=true

[/Script/OnlineSubsystemSteam.SteamNetDriver]
NetConnectionClassName="OnlineSubsystemSteam.SteamNetConnection"
```

然后，我们在*.Build.cs中添加以下插件，这里添加到Public也可以。

```
PrivateDependencyModuleNames.AddRange(new string[] { "OnlineSubsystem",
"OnlineSubsystemUtils", "OnlineSubsystemNull", "OnlineSubsystemSteam" });
```

接下来在*.Target.cs中添加`bUsesSteam = true;`一行，如下。

```
public class UETutorialTarget : TargetRules
{
    public UETutorialTarget(TargetInfo Target) : base(Target)
    {
        Type = TargetType.Game;
        DefaultBuildSettings = BuildSettingsVersion.V5;

        bUsesSteam = true;

        IncludeOrderVersion = EngineIncludeOrderVersion.Unreal5_4;
        ExtraModuleNames.Add("UETutorial");
    }
}
```

将创建会话和搜索会话时的IsLan调整成false。在我们创建会话的时候判断IsLan，选择是否进行`SessionSettings.BuildUniqueId = rand();`这一句的执行。

```
FOnlineSessionSettings SessionSettings;

SessionSettings.bIsLANMatch = bIsLanConnection;                         // 使用局域网
SessionSettings.bShouldAdvertise = true;                    // 广播会话信息
SessionSettings.NumPublicConnections = NumPublicConnections; // 最大玩家数
SessionSettings.bAllowJoinInProgress = true;                // 允许加入进行中的会话
SessionSettings.bUsesPresence = true;                      // 不使用在线状态
SessionSettings.bIsDedicated = false; // 设置为非专用服务器,
SessionSettings.bAllowJoinViaPresence = true;   //...
SessionSettings.bUseLobbiesIfAvailable = true;  //...

SessionSettings.Set(FName(TEXT("SERVER_NAME")),
PlayerProfileinfo.PlayerName.ToString(),
EOnlineDataAdvertisementType::ViaOnlineServiceAndPing);
//1.4.1
if (!bIsLanConnection) {//bIsLanConnection==false, Online
    SessionSettings.BuildUniqueId = rand();
}

// 创建会话
bool bResult = SessionInterface->CreateSession(0, SessionName, SessionSettings);
```

完成以上设置后，保存并重新编译运行。在打包之后，我们就可以在两个设备上通过Steam连接。当不需要使用Steam系统的时候，我们将DefaultPlatformService改为NULL这一行即可。

1.4.2 大厅菜单界面制作

（1）修改IsLan状态

在MPUserWidget中为Connect按钮添加一个点击函数，用于修改当前的IsLan状态，将Lan改为Online文本。

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

UFUNCTION()
void OnBttn_ConnectionTypeClick();
```

并且在NativeOnInitialized函数中绑定这个点击事件

```
if (Bttn_ConnectionType) {
    Bttn_ConnectionType->OnClicked.AddDynamic(this,
&UMPUserWidget::OnBttn_ConnectionTypeClick);
}
```

接着在MPUserWidget.cpp中创建对应的定义。

```
void UMPUserWidget::OnBttn_ConnectionTypeClick()
{
    MyGameInstance->ChangeConnectionType();
    SetConnectionType_Text(MyGameInstance->bIsLanConnection);
}
```

（2）制作LobbyMenu蓝图

复制并粘贴一个新的WB_MainMenu蓝图出来并重命名为WB_LobbyMenu。在这个蓝图中，修改我们的界面为如下图1.4.2即可，可以设置成自己喜欢的样子，前面的内容我们做过很多次了，这里读者可以自由发挥。

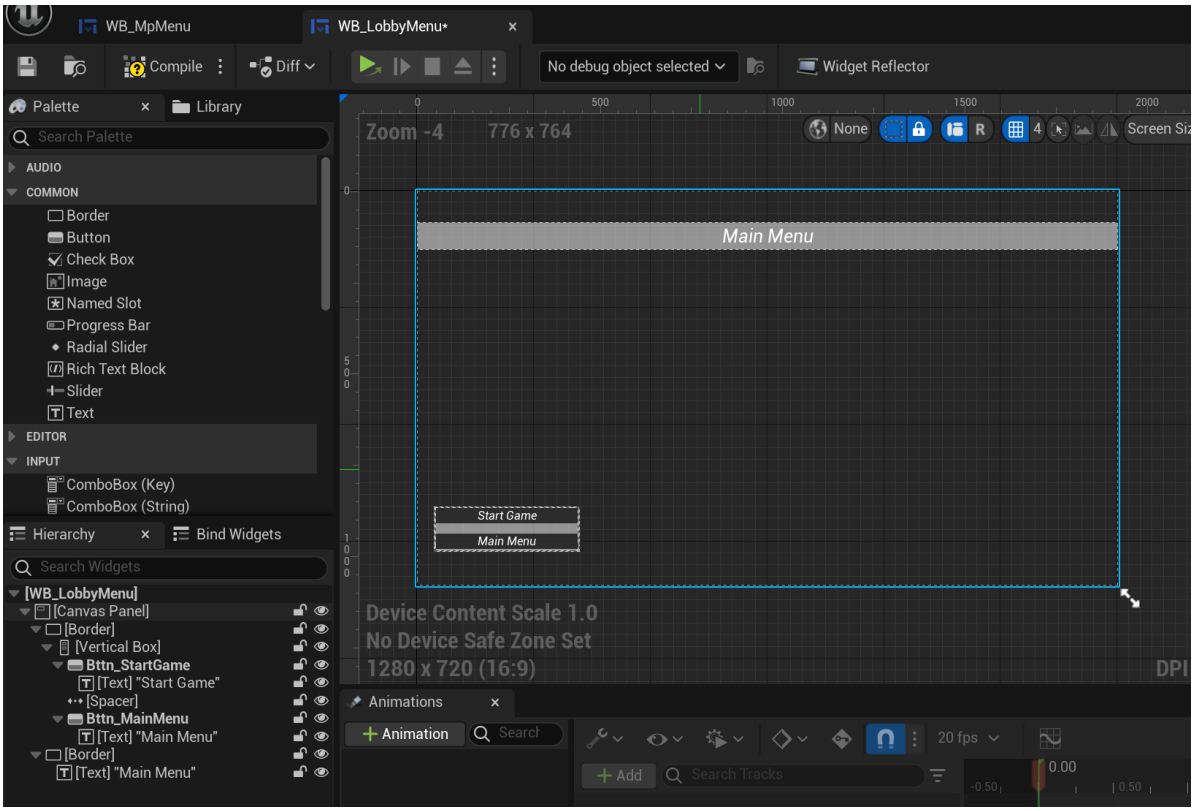

图1.4.2.1 修改蓝图界面

我们拖拽一个Border到CanvasPanel层级下，然后设置这个框的透明度为0.5，即Details/Appearance/BrushColor的第四个维度A为0.5，然后我们设置一个合适的大小和位置如图1.4.2.2所示，这个Border中我们后续会添加正在等待的玩家名称。

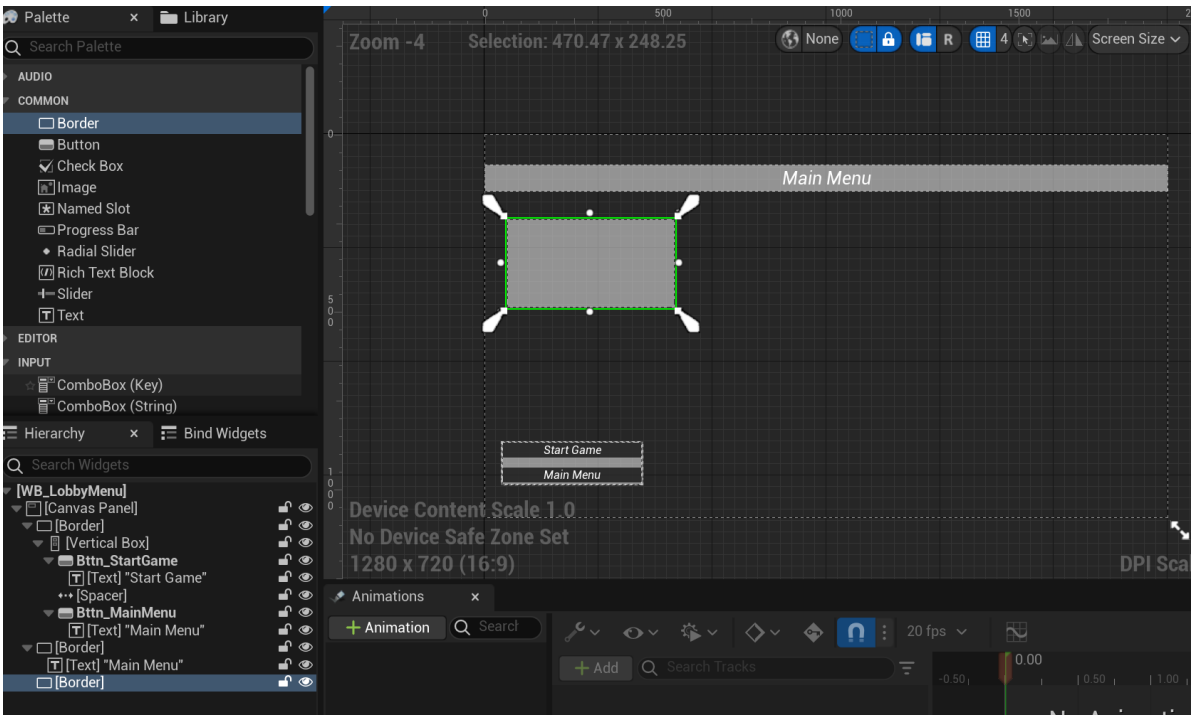

图1.4.2.2 添加Border

接着我们拖拽一个新的Border到我们刚才创建的Border的层级下，并且拖拽一个Text文本进来，修改刚创建的Boder的Details/Appearance/Brush Color为（0，0，0，0.7），设置字体和其他文本的字体一致（这里直接复制字体即可），并且修改Border的变量名为Border_PlayerName，修改Text变量名为Text_PlayerName，并都设置为IsVariable如图1.4.2.3所示。

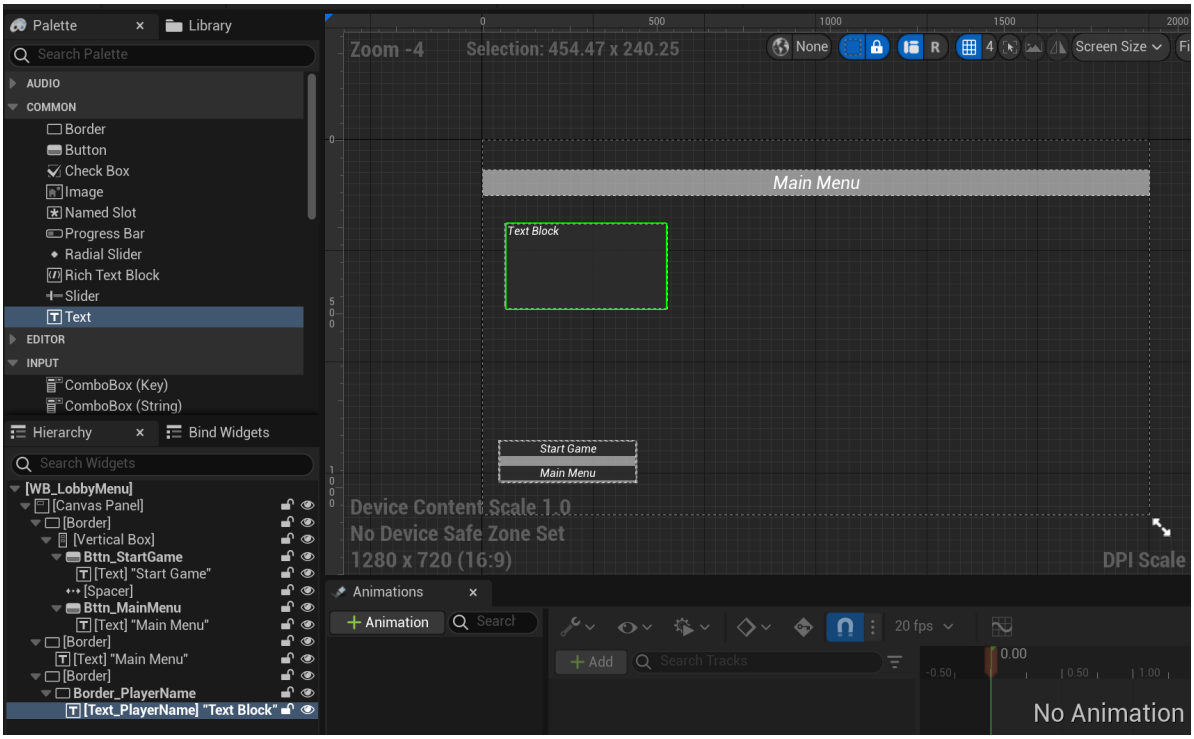

图1.4.2.3 添加用于显示玩家名字组件

在Border_PlayerName上包裹一个垂直框，然后复制Border_PlayerName和其子组件，然后粘贴出3份出来。然后选中这四个Border_PlayerName，设置一下Details/Slot/Padding属性，让这四个框更加美观。把这四个Text_PlayerName的文本内容设置为Waiting for player...,然后将文本对其到中心，如图1.4.2.4所示。

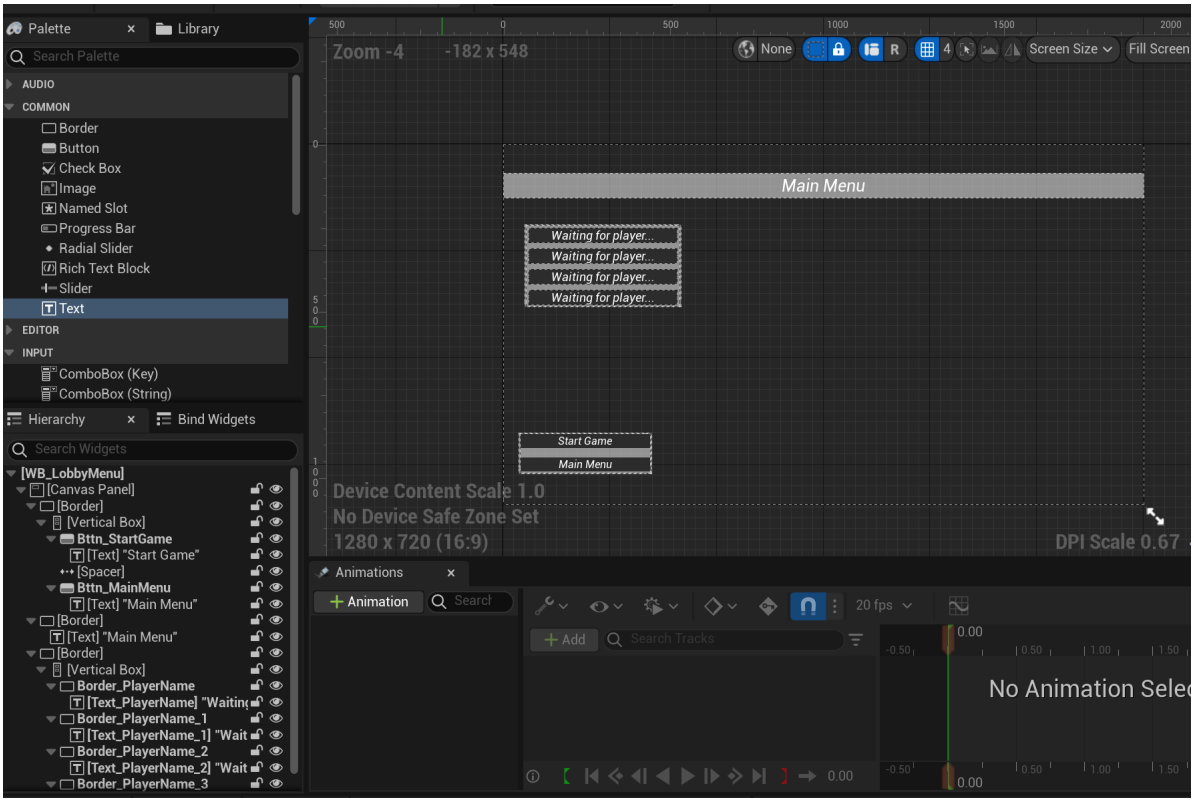

图1.4.2.4 复制粘贴出额外的3份组件

修改StartGameBttn的文本的变量名为Text_StartGameBttn，并设置为变量。新建UserWidget的C++类命名为LobbyMenuUserWidget,并绑定为刚才创建的多人菜单蓝图的父类。

1.4.3 DestroySession退出会话

（1）DestroySession函数

首先，我们打开ShootMenuController.h和ShootMenuController.cpp文件。我们需要在客户端进行退出会话连接的操作，那么我们需要用到RPC告诉客户端结束会话。ShootMenuController.h在头文件中，我们声明以下内容，正如我们之前章节中提到的那样。

ShootMenuController.h

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

这样，我们会确保这个函数在客户端运行，结束会话。

ShootMenuController.cpp

```
void AShootMenuController::DestroyClientSession_Implementation()
{
    if (MyGameInstance->CurrentSessionName.IsNone()) {
        return;
    }

    if (MyGameInstance->SessionInterface.IsValid()) {
        MyGameInstance->SessionInterface->DestroySession(MyGameInstance-
>CurrentSessionName);
    }
}
```

当然，我们需要在之前进入会话的时候，就记住当时我们进入的会话名称CurrentSessionName，用以后续退出会话。故我们需要声明以下变量。

TutorialGameInstance.h

```
FName CurrentSessionName;
```

当我们加入到会话或者创建会话之后，我们就修改这个变量为我们加入的会话名称。

TutorialGameInstance.cpp

```
void UTutorialGameInstance::OnCreateSessionComplete(FName SessionName, bool
bWasSuccessful)
{
    if (bWasSuccessful)
    {
        CurrentSessionName = SessionName;

        GetWorld()->ServerTravel("LobbyMenu_Map?listen");
    }
}
void UTutorialGameInstance::OnJoinSessionComplete(FName SessionName,
EOnJoinSessionCompleteResult::Type Result)
{

    if (APlayerController* PController =
UGameplayStatics::GetPlayerController(GetWorld(), 0)) {
        FString JoinAddress = "";
        SessionInterface->GetResolvedConnectString(SessionName, JoinAddress);
        if (JoinAddress != "") {
            PController->ClientTravel(JoinAddress,
ETravelType::TRAVEL_Absolute);
        }
        //1.4.2.3
        CurrentSessionName = SessionName;
    }

}
```

此时在上面的DestroySession的传参处，我们就有了正确的传参。

(2) OnDestroySessionComplete

当我们退出了会话之后，我们应该在退出结束之后，我们应该让CurrentSessionName重置为空，并且重新打开最开始的主菜单。

TutorialGameInstance.cpp

```
void UTutorialGameInstance::OnDestroySessionComplete(FName SessionName, bool
bWasSuccessful)
{
    if (bWasSuccessful) {
        CurrentSessionName = FName();

        UGameplayStatics::OpenLevel(this, FName("MainMenu_Map"));
    }
}
```

这个函数需要绑定到OnDestroySessionCompleteDelegates委托上，因此我们还需要在Init函数中添加一行绑定。

TutorialGameInstance.cpp

```
void UTutorialGameInstance::Init()
{
    Super::Init();
    ...
    if (IOnlineSubsystem* Subsystem = IOnlineSubsystem::Get("")) {
        SessionInterface = Subsystem->GetSessionInterface();
        if (SessionInterface.IsValid())
        {
            ...
            //OnDestroySessionCompleteDelegates委托绑定
            SessionInterface->OnDestroySessionCompleteDelegates.AddUObject(this,
&UTutorialGameInstance::OnDestroySessionComplete);
        }
    }
}
```

(3) 绑定到Bttn_MainMenu按钮

在LobbyMenuUserWidget.h中进行如下的声明。

LobbyMenuUserWidget.h

```
virtual void NativeOnInitialized() override;

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

UFUNCTION()
void OnBttn_MainMenuClick();
```

声明我们初始化函数，按钮以及点击按钮的响应函数。

LobbyMenuUserWidget.cpp

```
void ULobbyMenuUserWidget::NativeOnInitialized()
{
    Super::NativeOnInitialized();
    if (Bttn_MainMenu) {
        Bttn_MainMenu->OnClicked.AddDynamic(this,
&ULobbyMenuUserWidget::OnBttn_MainMenuClick);
    }
    MyGameInstance = Cast<UTutorialGameInstance>(GetGameInstance());
}
void ULobbyMenuUserWidget::OnBttn_MainMenuClick()
{
    AShootMenuController* MyController = Cast<AShootMenuController>(GetWorld()-
>GetFirstPlayerController());
    if (MyController && MyController->HasAuthority()) {
        AShootMenuGameMode* GameMode = Cast<AShootMenuGameMode>
(UGameplayStatics::GetGameMode(GetWorld()));
        if (GameMode) {
```

```
            for (AShootMenuController* Controller : GameMode-
>All_ControllerPlayers) {
                if (Controller != MyController) {
                    Controller->DestroyClientSession();
                }
            }
            MyController->DestroyClientSession();
        }
    }
    else {
        if (MyController)
        {
            //这里还需要删除服务器上的All_ControllerPlayers列表的对应元素...
            MyController->DestroyClientSession();
        }
    }
}
```

当点击这个按钮的时候，我们会判断当前是否处于服务器上。如果在服务器上的话，我们会先断开所有连接到客户端的会话，最后再断开我们服务器的会话。如果处于客户端上的话，我们会直接断开会话。

在这里我们有一个还没声明的变量All_ControllerPlayers，我们打开ShootMenuGameMode.h进行声明。这个数组用于存储连接到这个会话中的控制器。

ShootMenuGameMode.h

```
UPROPERTY()
TArray<AShootMenuController*> All_ControllerPlayers;
```

那么，我们就完成了MainMenu按钮的制作。

1.4.4 StartGame按钮

（1）编写TravelToMap事件

打开TutorialGameInstance.h文件，我们需要创建一个TravelToMap事件，用以让服务端带着所有连接到的客户端进行地图切换。

TutorialGameInstance.h

```
DECLARE_MULTICAST_DELEGATE_TwoParams(FTravelToMapEventDelegate,int32,FString);
```

首先，我们在类的外面声明这个委托。然后在类中定义以下成员，分别是委托，触发函数，绑定函数，当前连接的玩家数量和对应的UE蓝图。这个UE的蓝图需要记得在编辑器中进行绑定。

TutorialGameInstance.h

```
FTravelToMapEventDelegate TravelToMapEventDelegate;

void TriggerTravelToMap(int32 NumConnectdPlayers, FString MapName);

void TravelToMap(int32 NumConnectdPlayers, FString MapName);

UPROPERTY()
int32 MP_NumConnectedPlayers;

UPROPERTY(EditAnywhere, Category = "UI")
TSubclassOf<ULobbyMenuUserWidget> LobbyMenuUserWidgetClass;
```

在Init函数中，我们需要给委托绑定响应函数。

TutorialGameInstance.cpp

```
void UTutorialGameInstance::Init()
{
    Super::Init();

    OnStartSoloGameEventTriggered.AddDynamic(this,
&UTutorialGameInstance::StartSoloGameEvent);
    ChangePlayerNameEvent.AddDynamic(this,
&UTutorialGameInstance::OnPlayerNameChanged);
    OnCreateSession.AddUObject(this ,&UTutorialGameInstance::OnCreateMPSession);
    OnJoinSession.AddUObject(this, &UTutorialGameInstance::OnJoinMPSession);
    //委托绑定响应函数
    TravelToMapEventDelegate.AddUObject(this,
&UTutorialGameInstance::TravelToMap);

    if (IOnlineSubsystem* Subsystem = IOnlineSubsystem::Get("")) {
        ...
    }
}
```

接下来，我们需要实现我们的响应函数和触发函数。

```
void UTutorialGameInstance::TriggerTravelToMap(int32 NumConnectdPlayers, FString
MapName)
{
    TravelToMapEventDelegate.Broadcast(NumConnectdPlayers, MapName);
}

void UTutorialGameInstance::TravelToMap(int32 NumConnectdPlayers, FString
MapName)
{
    MP_NumConnectedPlayers = NumConnectdPlayers;
    GetWorld()->ServerTravel(MapName + "?listen", true);
}
```

用ServerTravel切换到指定的地图中。

（2）绑定StartGame按钮

在LobbyMenuUserWidget.h头文件中，创建按钮和点击函数的声明。

LobbyMenuUserWidget.h

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

UFUNCTION()
void OnBttn_StartGameCilck();

UPROPERTY(BlueprintReadOnly, meta = (BindWidgetOptional))
UTextBlock* Text_StartGameBttn;
```

LobbyMenuUserWidget.cpp

```
void ULobbyMenuUserWidget::NativeOnInitialized()
{
    Super::NativeOnInitialized();
    if (Bttn_MainMenu) {...}
    if (Bttn_StartGame) {
        Bttn_StartGame->OnClicked.AddDynamic(this,
&ULobbyMenuUserWidget::OnBttn_StartGameCilck);
    }
    MyGameInstance = Cast<UTutorialGameInstance>(GetGameInstance());
}
void ULobbyMenuUserWidget::OnBttn_StartGameCilck()
{
    if (MyGameInstance) {
        AShootMenuGameMode* GameMode = Cast<AShootMenuGameMode>
(UGameplayStatics::GetGameMode(GetWorld()));
        if (GameMode) {
            MyGameInstance->TriggerTravelToMap(GameMode-
>All_ControllerPlayers.Num(), TEXT("Level01"));
        }
    }
}
```

在NativeOnInitialized中绑定函数到委托上。接着编写OnBttn_StartGameCilck函数逻辑，那就是让服务端点击这个按钮，然后进入到游戏地图中。那么，我们需要避免客户端按下这个按钮，因此，在打开这个蓝图的时候，我们会判断是服务端还是客户端，一边可供点击，而客户端不行。

LobbyMenuUserWidget.h

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

LobbyMenuUserWidget.cpp

```
void ULobbyMenuUserWidget::NativeConstruct()
{
    Super::NativeConstruct();
    AShootMenuController* MyController = Cast<AShootMenuController>(GetWorld()->GetFirstPlayerController());
    if (MyController && MyController->HasAuthority()) {
        //OnServer
    }
    else {
        //OnClient
        Text_StartGameBttn->SetText(FText::FromString(TEXT("Please wait")));
        Bttn_StartGame->SetIsEnabled(false);
    }
}
```

这样我们实现了开始按钮的点击，而且只能由服务器进行点击。

1.4.5 利用RPC在客户端创建蓝图

(1) 绑定一些蓝图组件

我们需要将蓝图中的那些PlayerName的框放到一个数组中，方便后续管理。

LobbyMenuUserWidget.h

```
UPROPERTY(BlueprintReadOnly, meta = (BindWidgetOptional))
UBorder* Border_PlayerName;
UPROPERTY(BlueprintReadOnly, meta = (BindWidgetOptional))
UBorder* Border_PlayerName_1;
UPROPERTY(BlueprintReadOnly, meta = (BindWidgetOptional))
UBorder* Border_PlayerName_2;
UPROPERTY(BlueprintReadOnly, meta = (BindWidgetOptional))
UBorder* Border_PlayerName_3;
UPROPERTY()
TArray<UBorder*> All_Border_PlayerName;

UPROPERTY(BlueprintReadOnly, meta = (BindWidgetOptional))
UTextBlock* Text_PlayerName;
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;
```

在头文件中，我们将所有的Border和Text都放到数组中。然后，再Init函数中进行添加。

LobbyMenuUserWidget.cpp

```
void ULobbyMenuUserWidget::NativeOnInitialized()
{
    Super::NativeOnInitialized();
    if (Bttn_MainMenu) {...}
    if (Bttn_StartGame) {...}
    MyGameInstance = Cast<UTutorialGameInstance>(GetGameInstance());
```

```
    All_Border_PlayerName.Add(Border_PlayerName);
    All_Border_PlayerName.Add(Border_PlayerName_1);
    All_Border_PlayerName.Add(Border_PlayerName_2);
    All_Border_PlayerName.Add(Border_PlayerName_3);

    All_Text_PlayerName.Add(Text_PlayerName);
    All_Text_PlayerName.Add(Text_PlayerName_1);
    All_Text_PlayerName.Add(Text_PlayerName_2);
    All_Text_PlayerName.Add(Text_PlayerName_3);
}
```

（2）UI_ShowLobbyMenu函数构建

接着，我们需要在客户端创建LobbyMenu蓝图，因此我们需要创建一个函数用来展示蓝图。

ShootMenuController.h

```
void CreateLobbyMenu();

UPROPERTY()
ULobbyMenuUserWidget* LobbyMenu;

UFUNCTION(Client, Reliable, Category = "Lobby")
void ClientCreateLobbyMenu();
void ClientCreateLobbyMenu_Implementation();
```

这里采用RPC的方式，将ClientCreateLobbyMenu发送到拥有这个ShootMenuController的客户端上。

ShootMenuController.cpp

```
void AShootMenuController::CreateLobbyMenu()
{
    if (IsLocalController()) {
        MyGameInstance = Cast<UTutorialGameInstance>(GetGameInstance());
        if (MyGameInstance) {
            if (!IsValid(LobbyMenu)) {
                LobbyMenu = CreateWidget<ULobbyMenuUserWidget>(GetWorld(),
MyGameInstance->LobbyMenuUserWidgetClass);
            }
            LobbyMenu->AddToViewport(0);
            FInputModeUIOnly InputMode;
            SetInputMode(InputMode);
            bShowMouseCursor = true;

        }
    }
}
void AShootMenuController::ClientCreateLobbyMenu_Implementation() {
    CreateLobbyMenu();
}
```

在ClientCreateLobbyMenu_Implementation中，我们仅调用CreateLobbyMenu函数即可。在CreateLobbyMenu函数中，我们会创建LobbyMenu蓝图并AddToViewport添加到视图中，并设置UI输入。

1.4.6 处理新玩家加入会话

（1）PostLogin

在ShootMenuGameMode.h中，我们声明PostLogin函数，这个函数当有新用户加入时就会执行（包括自身）。

ShootMenuGameMode.h

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

ShootMenuGameMode.cpp

```
void AShootMenuGameMode::PostLogin(APlayerController* NewPlayer)
{
    Super::PostLogin(NewPlayer);
    FString LevelName = UGameplayStatics::GetCurrentLevelName(this, true);
    if (LevelName.Equals(TEXT("LobbyMenu_Map")))
    {
        AShootMenuController* NewPlayerController = Cast<AShootMenuController>
(NewPlayer);
        if (NewPlayerController)
        {
            All_ControllerPlayers.Add(NewPlayerController);
            NewPlayerController->ClientCreateLobbyMenu();
            NewPlayerController->Init_Setup();
        }
    }
}
```

在这个函数中，我们判断当前是否处于LobbyMenu_Map中。如果处于其中，我们就添加这个新的用户到All_ControllerPlayers列表中，然后在对应的客户端ClientCreateLobbyMenu展示大厅UI。然后，调用Init_Setup函数（该函数后续会实现）进行大厅的初始化。

（2）创建Init_Setup函数，初始化大厅

在这之前，在ShootMenuController.h我们声明一个函数用以加载我们在MyGameInstance存放的PlayerProfileinfo

ShootMenuController.h

```
UFUNCTION()
void LoadSavedProfile();

UPROPERTY()
FS_PlayerProfile PlayerProfileinfo;
```

ShootMenuController.cpp

```
void AShootMenuController::LoadSavedProfile()
{
    if (MyGameInstance) {
        PlayerProfileinfo = MyGameInstance->PlayerProfileinfo;
    }
}
```

接着，我们在ShootMenuController.h声明Init_Setup用以创建大厅。

ShootMenuController.h

```
UFUNCTION(Client, Reliable, Category = "Lobby")
void Init_Setup();
void Init_Setup_Implementation();
```

ShootMenuController.cpp

```
void AShootMenuController::Init_Setup_Implementation()
{
    LoadSavedProfile();
    Server_UpdateLobby(PlayerProfileinfo);
}
```

（3）Server_UpdateLobby函数

在客户端上调用LoadSavedProfile函数加载GameInstance中的数据。然后调用Server_UpdateLobby更新我们的大厅页面（这个函数我们马上就会实现它）。

ShootMenuController.h

```
UFUNCTION(Server, Reliable)
void Server_UpdateLobby(FS_PlayerProfile PlayerProfile);
void Server_UpdateLobby_Implementation(FS_PlayerProfile PlayerProfile);
```

ShootMenuController.cpp

```
void AShootMenuController::Server_UpdateLobby_Implementation(FS_PlayerProfile
PlayerProfile)
{
    PlayerProfileinfo = PlayerProfile;
    AShootMenuGameMode* GameMode = Cast<AShootMenuGameMode>
(UGameplayStatics::GetGameMode(GetWorld()));
    GameMode->UpdateLobby(true);
}
```

Server_UpdateLobby函数中，我们利用RPC转发到服务器上，修改服务端上的对应数据。然后在服务端上调用UpdateLobby函数，该函数会修改会话中所有客户端的界面。

（4）UpdateLobby函数

这个函数会将All_ControllerPlayers连接进来的玩家的UI界面进行更新。

ShootMenuGameMode.h

```
UFUNCTION()
void UpdateLobby(bool PlayerNames);

UPROPERTY()
TArray<FS_PlayerProfile> All_PlayerProfiles;
```

ShootMenuGameMode.cpp

```
void AShootMenuGameMode::UpdateLobby(bool PlayerNames)
{
    for (int32 i = 0; i < All_ControllerPlayers.Num(); ++i) {
        All_PlayerProfiles.Add(All_ControllerPlayers[i]->PlayerProfileinfo);
    }
    if (PlayerNames) {
        for (int32 i = 0; i < All_ControllerPlayers.Num(); ++i) {
            All_ControllerPlayers[i]->UpdatePlayerNames(All_PlayerProfiles);
        }
    }
}
```

我们新建All_PlayerProfiles属性，用以存放所有连接进来的玩家的信息。然后对每一个All_ControllerPlayers的玩家，进行UpdatePlayerNames这个函数会实现UI界面更新，传入所有玩家的信息作为参数。

（5）AShootMenuController::UpdatePlayerNames函数

这个函数会利用RPC转发到客户端上，进行UI的更新。

ShootMenuController.h

```
UFUNCTION(Client, Reliable, Category = "Lobby")
void UpdatePlayerNames(const TArray<FS_PlayerProfile>& PlayerProfiles);
void UpdatePlayerNames_Implementation(const TArray<FS_PlayerProfile>&
PlayerProfiles);
```

ShootMenuController.cpp

```
void AShootMenuController::UpdatePlayerNames_Implementation(const
TArray<FS_PlayerProfile>& PlayerProfiles)
{
    if (IsValid(LobbyMenu)) {
        LobbyMenu->ClearPlayerNames();
        LobbyMenu->UpdatePlayerNames(PlayerProfiles);
    }
}
```

首先，我们会ClearPlayerNames先重置UI。然后，再UpdatePlayerNames更新UI内容。

（6）ClearPlayerNames和ULobbyMenuUserWidget::UpdatePlayerNames函数

这两个函数负责重置UI和更新UI内容。

LobbyMenuUserWidget.h

```
UFUNCTION()
void UpdatePlayerNames(const TArray<FS_PlayerProfile>& PlayerProfiles);

UFUNCTION()
void ClearPlayerNames();
```

LobbyMenuUserWidget.cpp

```
void ULobbyMenuUserWidget::UpdatePlayerNames(const TArray<FS_PlayerProfile>&
PlayerProfiles)
```

```
{
    for (int32 i = 0; i < PlayerProfiles.Num();++i) {
        All_Text_PlayerName[i]->SetText(PlayerProfiles[i].PlayerName);
        All_Border_PlayerName[i]->SetBrushColor(FLinearColor(0,0,0,1.f));
    }
}

void ULobbyMenuUserWidget::ClearPlayerNames()
{
    for (int32 i = 0; i < All_Text_PlayerName.Num(); ++i) {
        All_Text_PlayerName[i]->SetText(FText::FromString(TEXT("Wariting for
player...")));
        All_Border_PlayerName[i]->SetBrushColor(FLinearColor(0,0,0,0.7f));
    }
}
```

这样子，我们就完成了UI的更新，成功处理了新玩家加入会话。我们切换到Steam子系统然后进行测试，如下图1.4.6.1所示。

图1.4.6.1 游戏界面

1.4.7 总结

在这一节中，我们完成了大厅菜单界面的制作。我们还有一些部分没有完成，读者可自行尝试完成。例如，当连接进来的其他用户点击MainMenu时，退出了会话，但是这个时候仍在会话中的人那边的All_ControllerPlayers和All_PlayerProfiles还没有删除对应的角色，以及UI也需要进行更新。