

游戏工程创建

026 创建工程

我们有了之前创建好的插件，接下来就是制作整个游戏的过程。

创建工作并验证插件

创建一个新的 UE5.1.1 空工程

将我们制作的插件文件夹拷贝到新的工程 Blaster 中，打开工程，添加插件。Online Subsystem Steam，并修改 DefaultEngine.ini 文件(详见：008 为 steam 配置工程)，之后再 DefaultGame.ini 中添加如下代码：

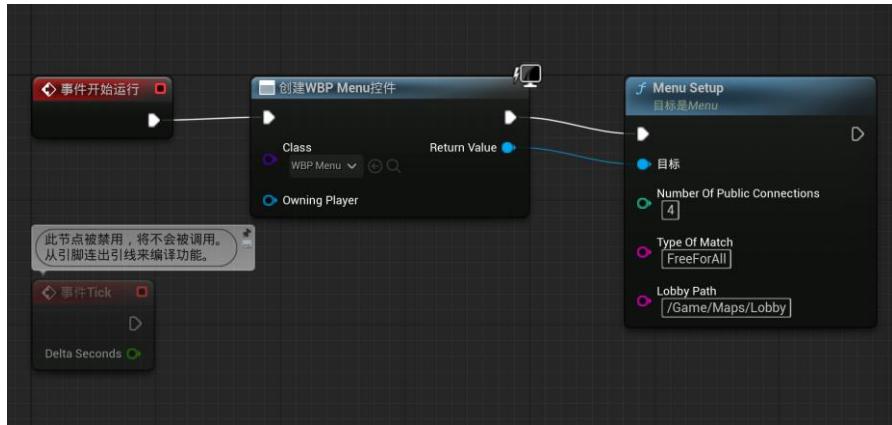
```
[/Script/Engine.GameSession]
```

```
MaxPlayers=100
```

在编辑器中添加两个新关卡 Lobby 和 GameStartupMap，并在编辑器设置中将两个地图添加到打包的地图中（详见：013 加入会话），同时修改项目设置的默认初始地图为 GameStartupMap：



打开 GameStartupMap 的关卡蓝图，连接蓝图并修改 Lobby 地图的地址的值：

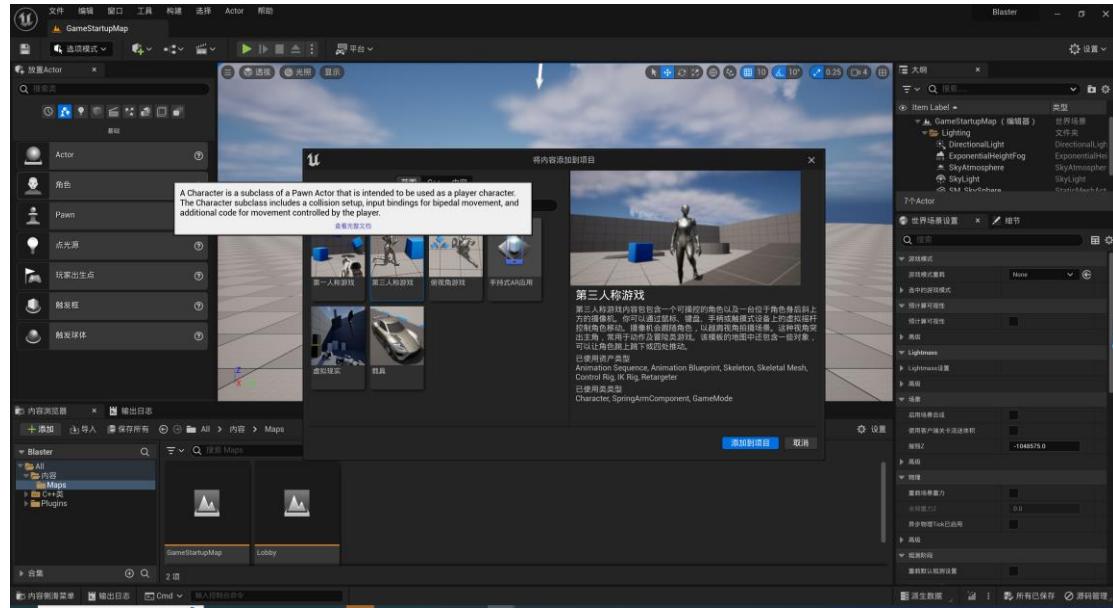


打包项目，实验能否连上 Steam。

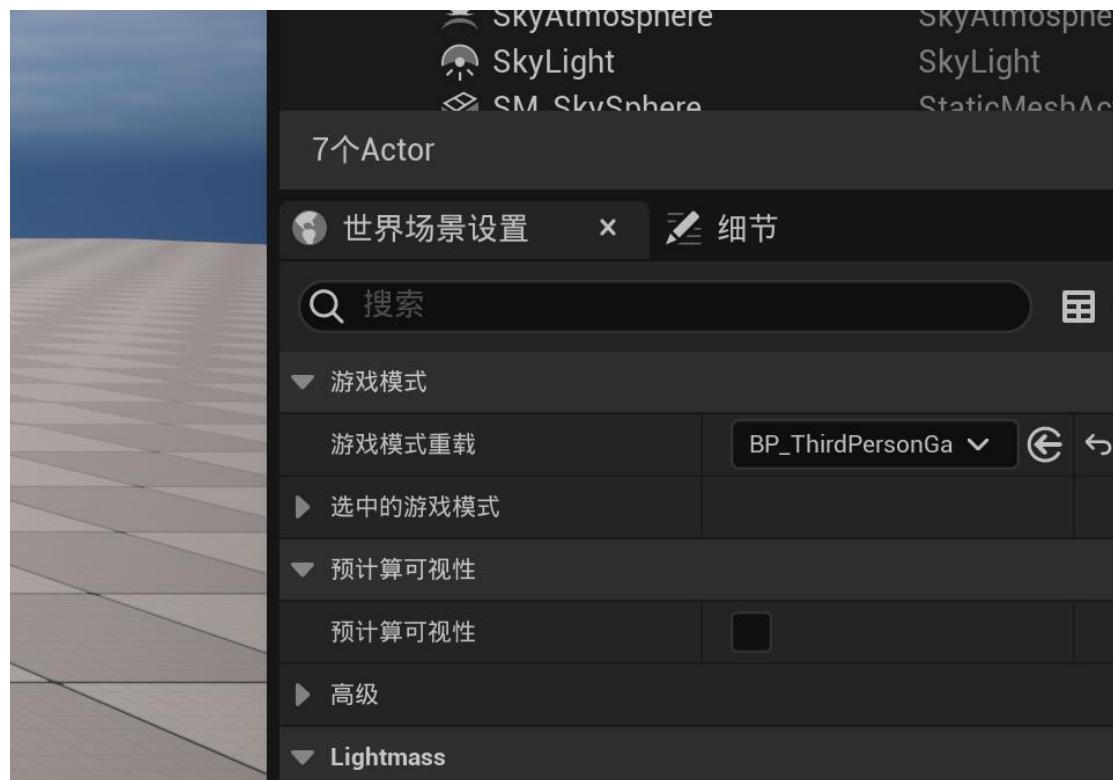


027 测试一个会话

由于默认 pawn 没有运动复制，我们需要添加第三人称游戏的资源。



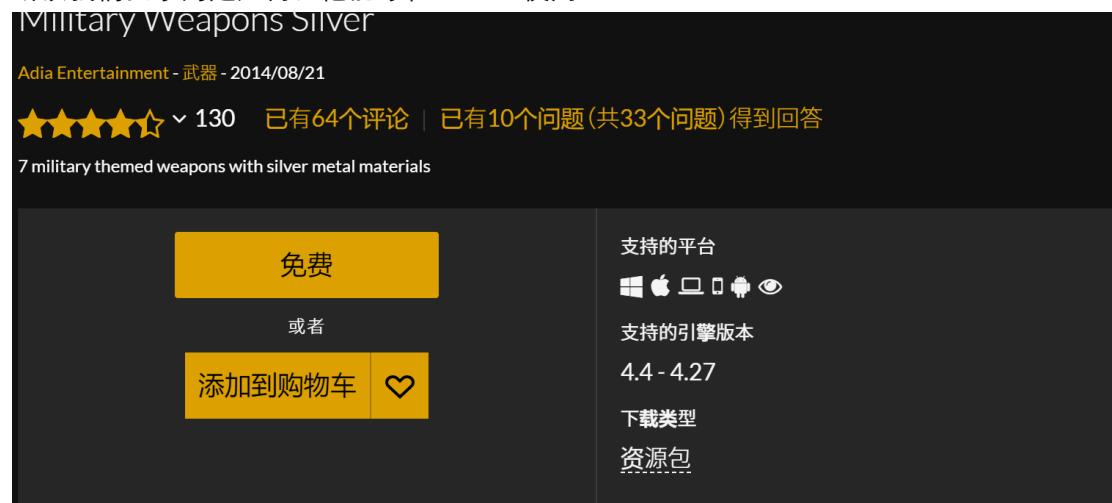
修改 Lobby 的默认游戏模式：



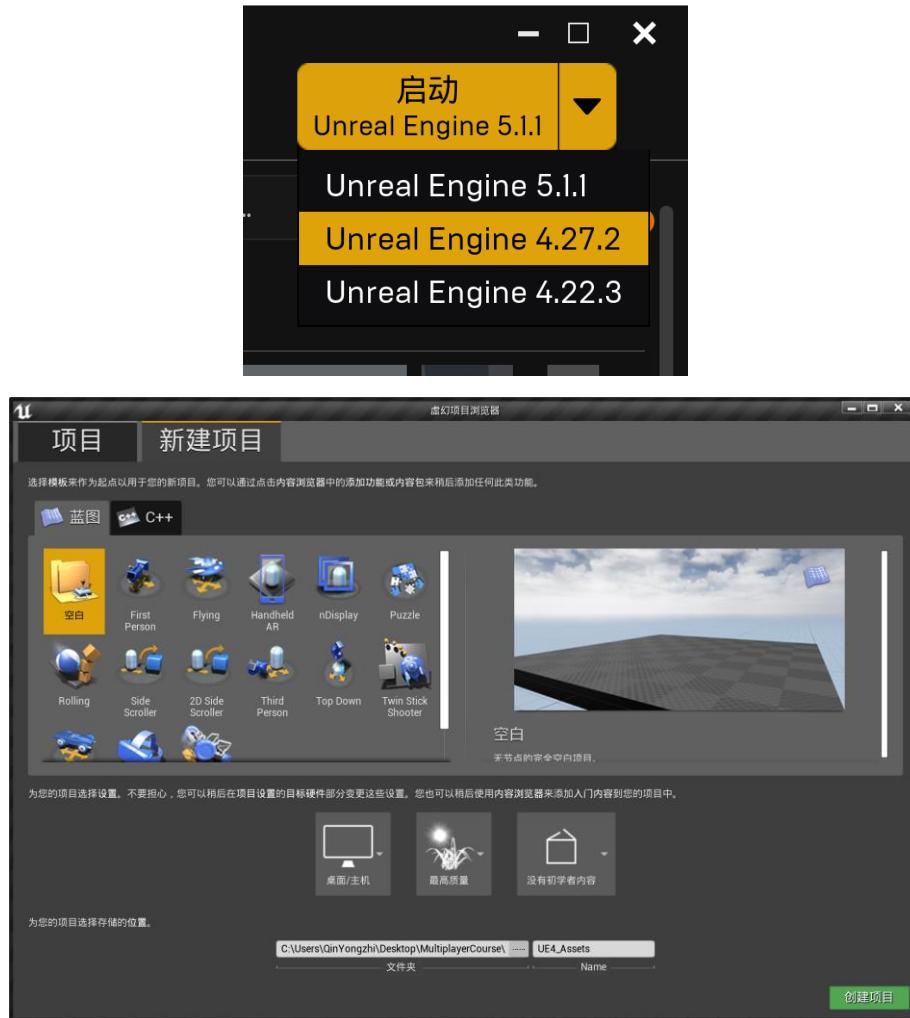
之后再打包游戏就能发现运动成功复制了。

028 资产

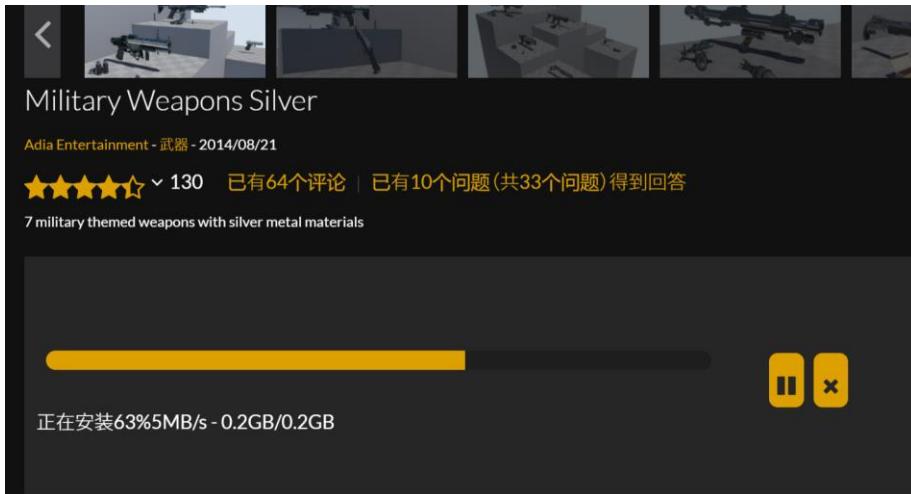
在虚幻商城里搜索 Military Weapons Silver, 我们可以看到对应的资源包, 他并不支持 UE5.1, 所以我们要学的是如何让他能够在 UE5 上使用:



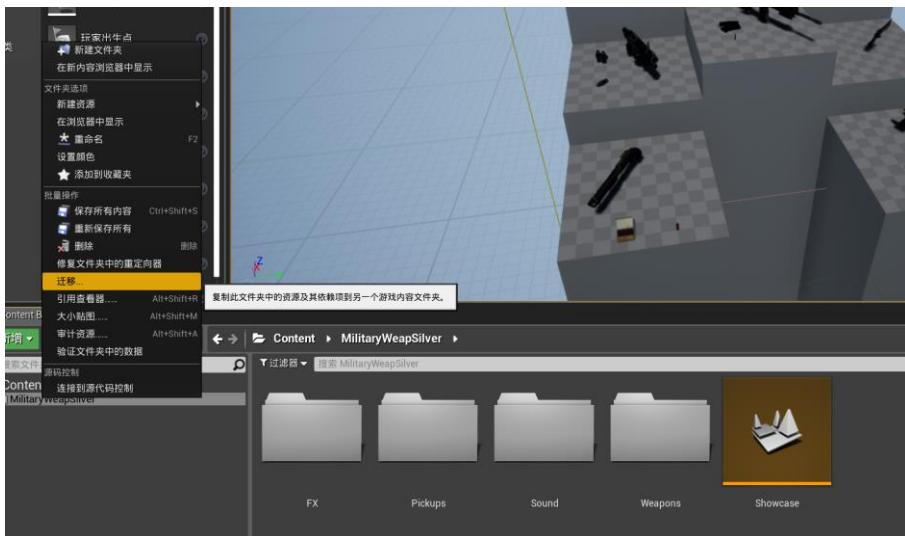
创建一个适合的版本的蓝图空项目：



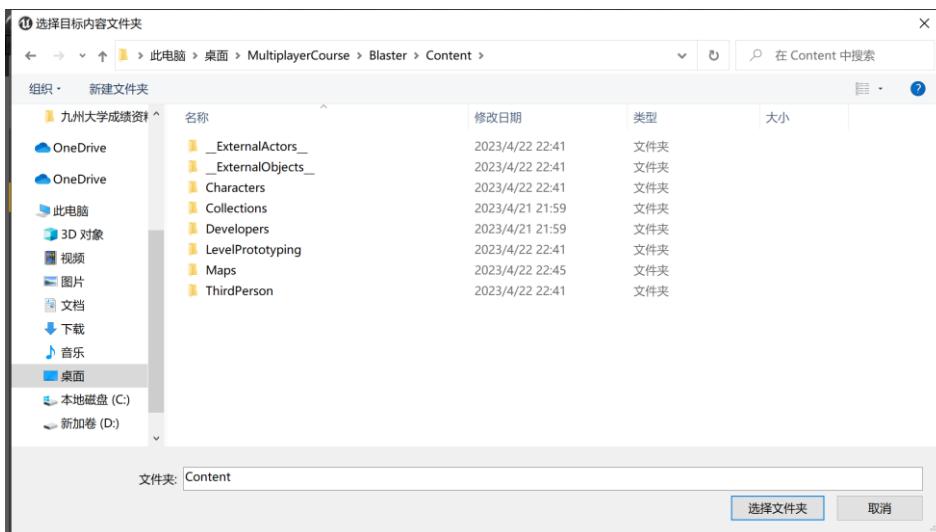
将资源包添加到我们创建的工程。



在工程中将添加的资源文件夹，选择迁移：



选择迁移到 Blaster 的 content 文件夹



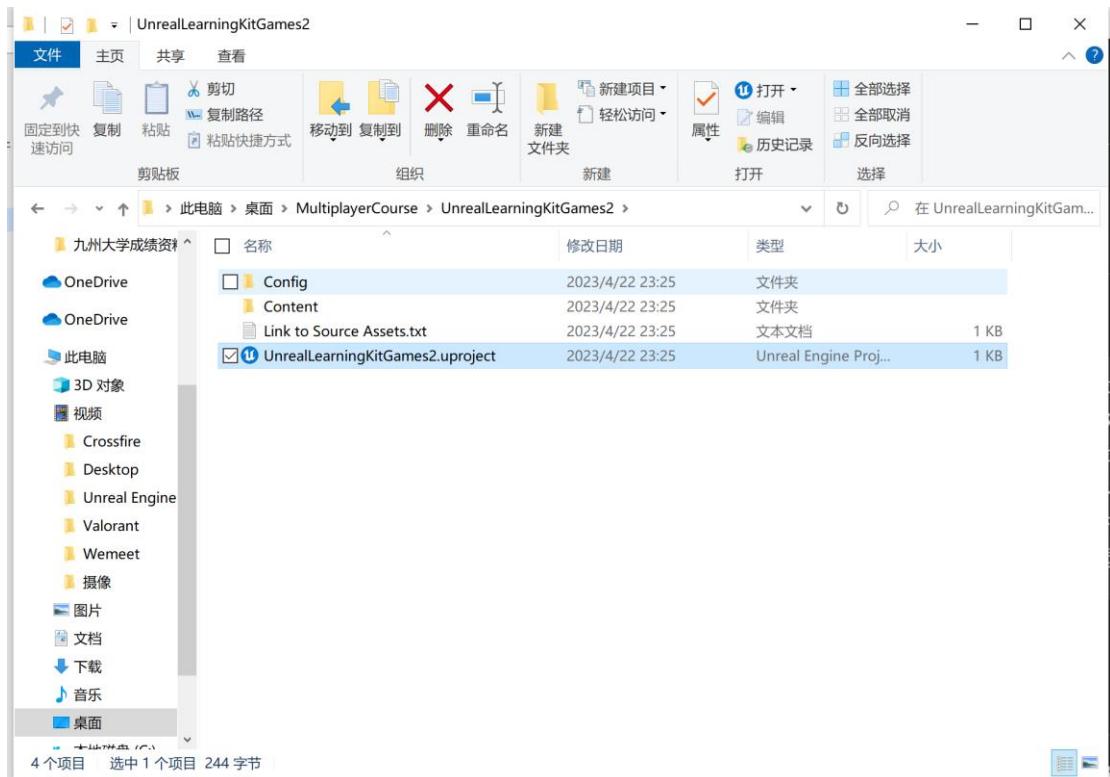
虚幻学习工具包

我们可以使用虚幻学习工具包 (unreal learning kit games) 中的许多资产。

在虚幻商城中搜索，然后创建工程，不要选用 UE5.1，用之前的版本

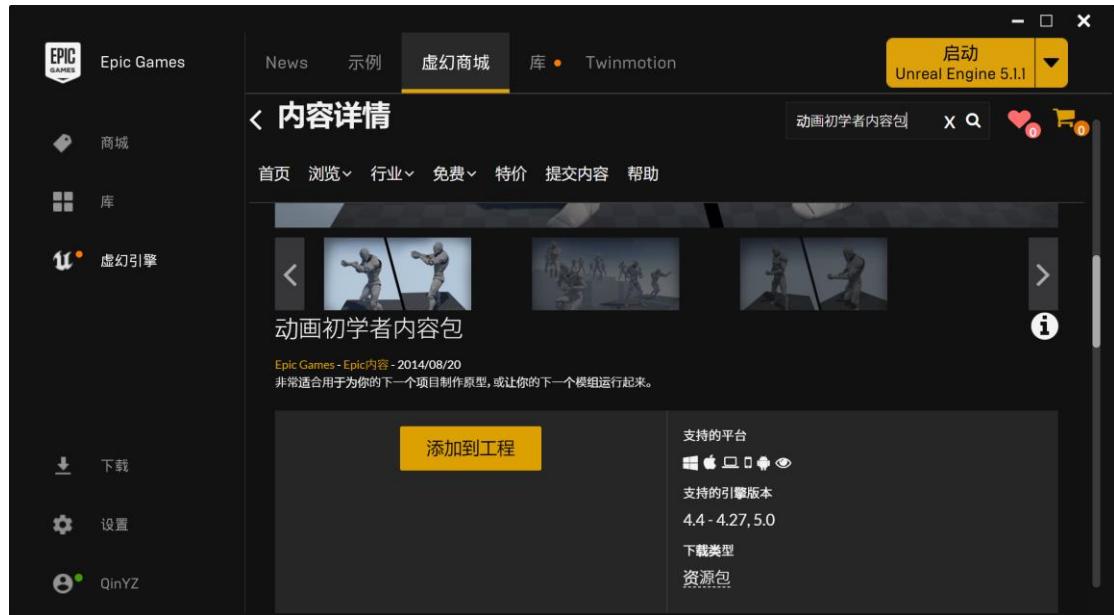


打开项目：



里面有许多的资产，如果我们还需要别的资产我们可以把它们添加到项目中，之后再迁移到我们自己的项目里。

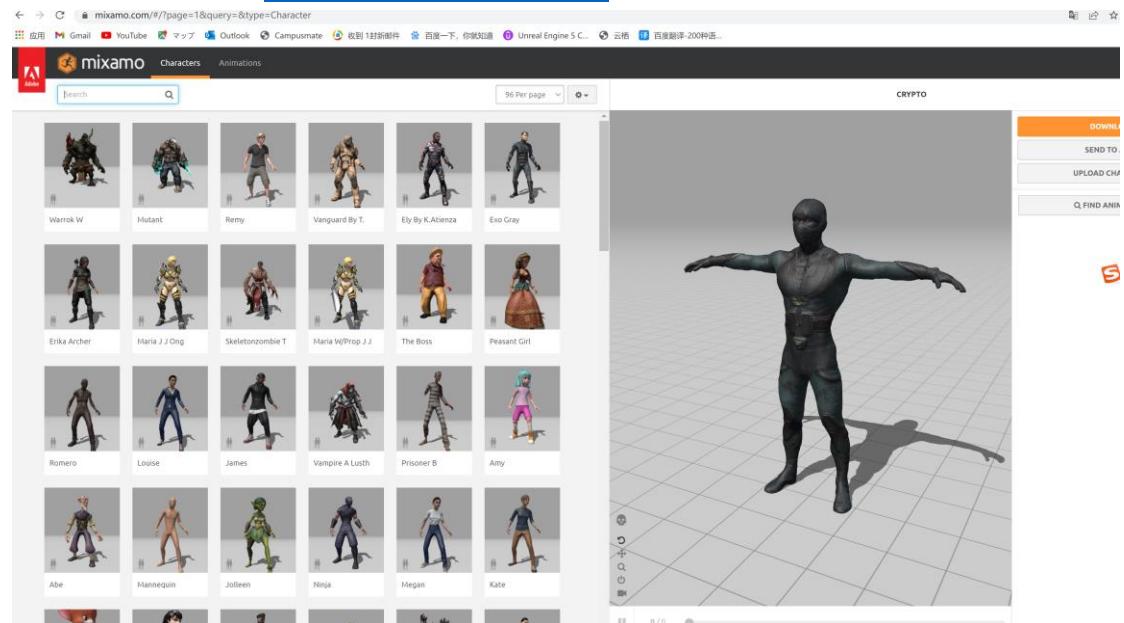
我们可以在添加一些动画资产：



但是这个动画包中还缺少我们所需的一些动画，比如在站立状态和蹲下状态的转身，我们将从其他外部网站寻找一些资源。

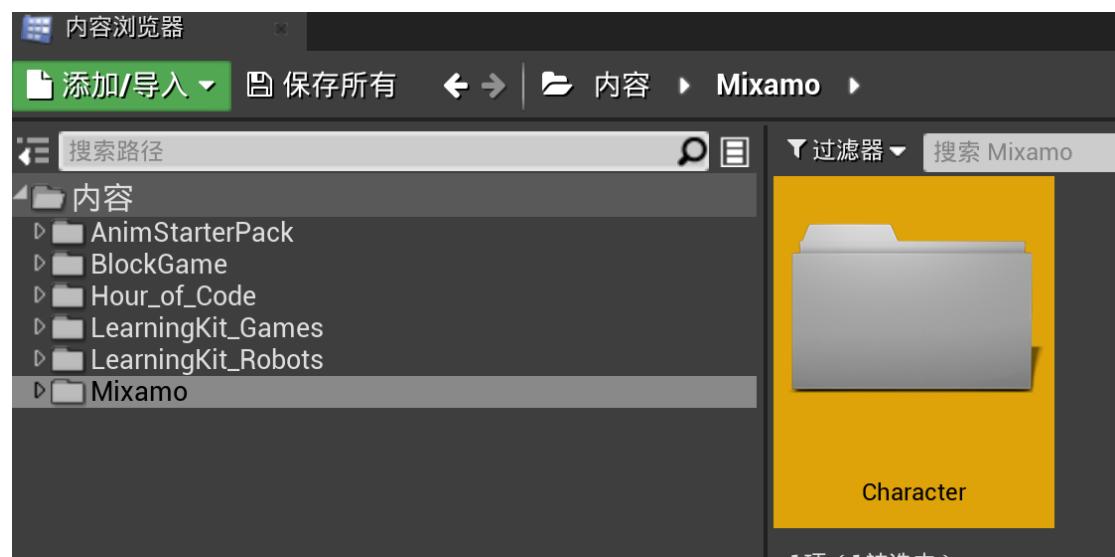
029 动画重定向

免费动画资源网站：<https://www.mixamo.com/#/>

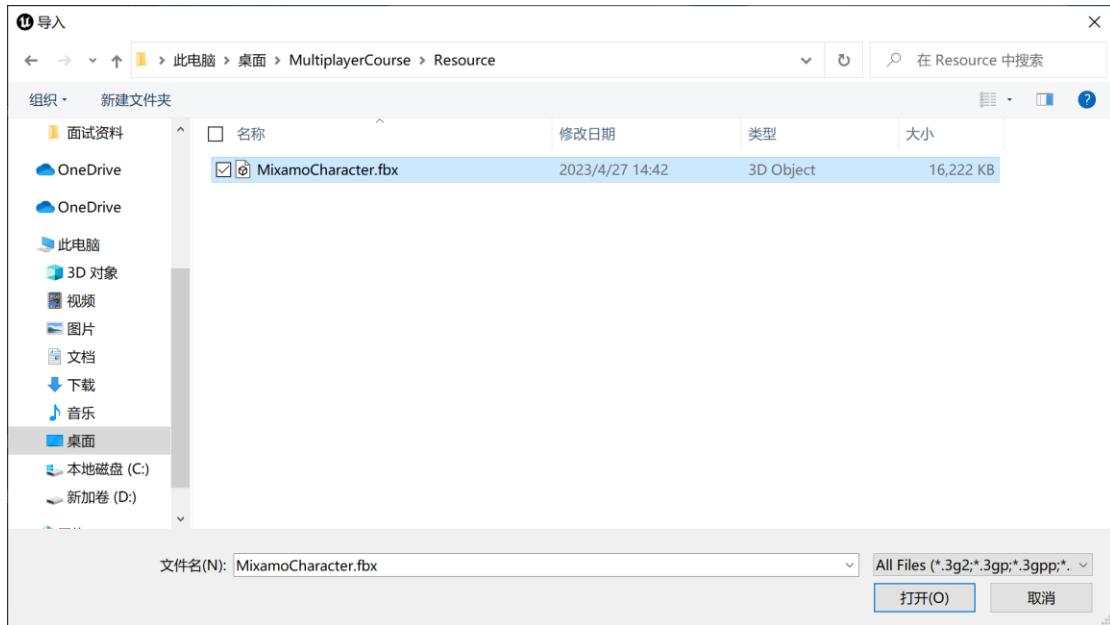


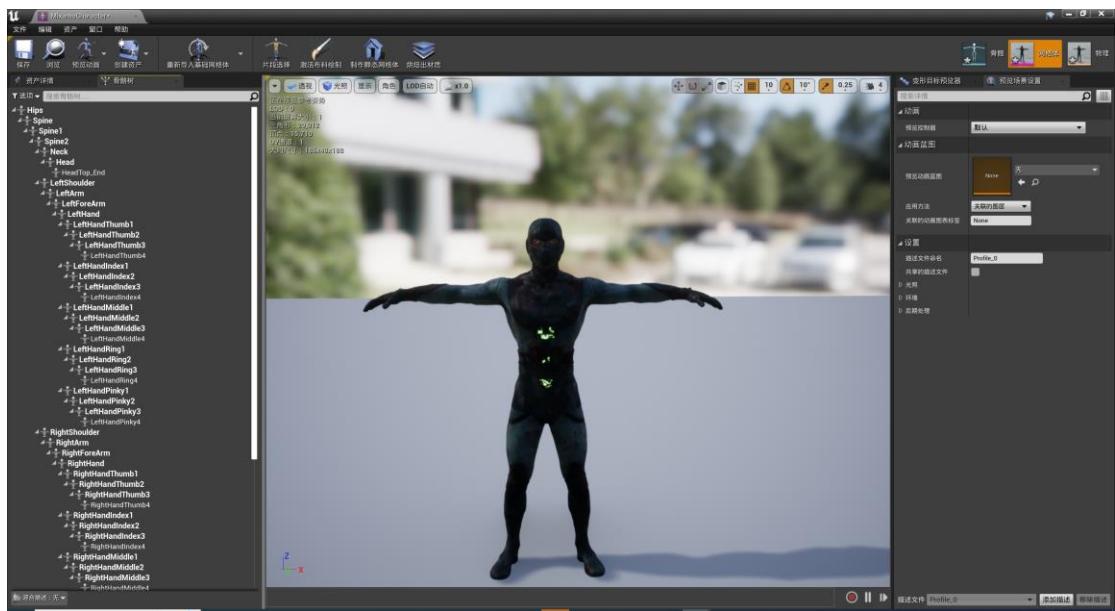
注册登录后先选择角色下载 T-pose 角色文件

打开 UE 编辑器，新建文件夹 Mixamo 然后再其下再添加一个文件夹 Character

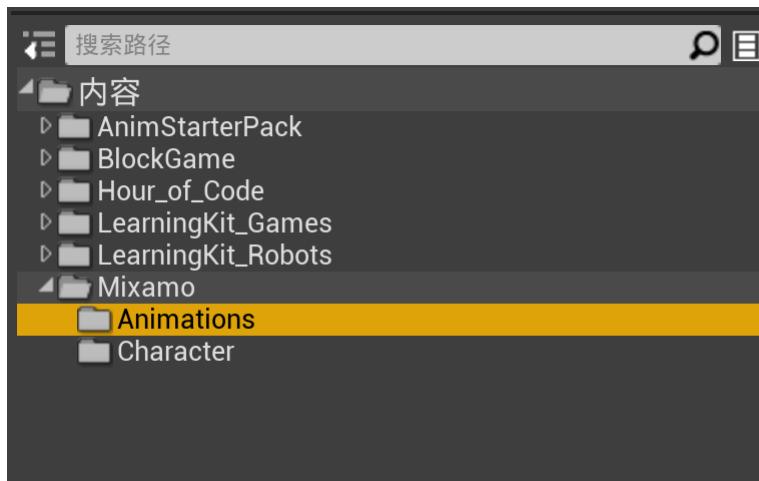


点击右键，选择我们的角色并添加到 UE 中



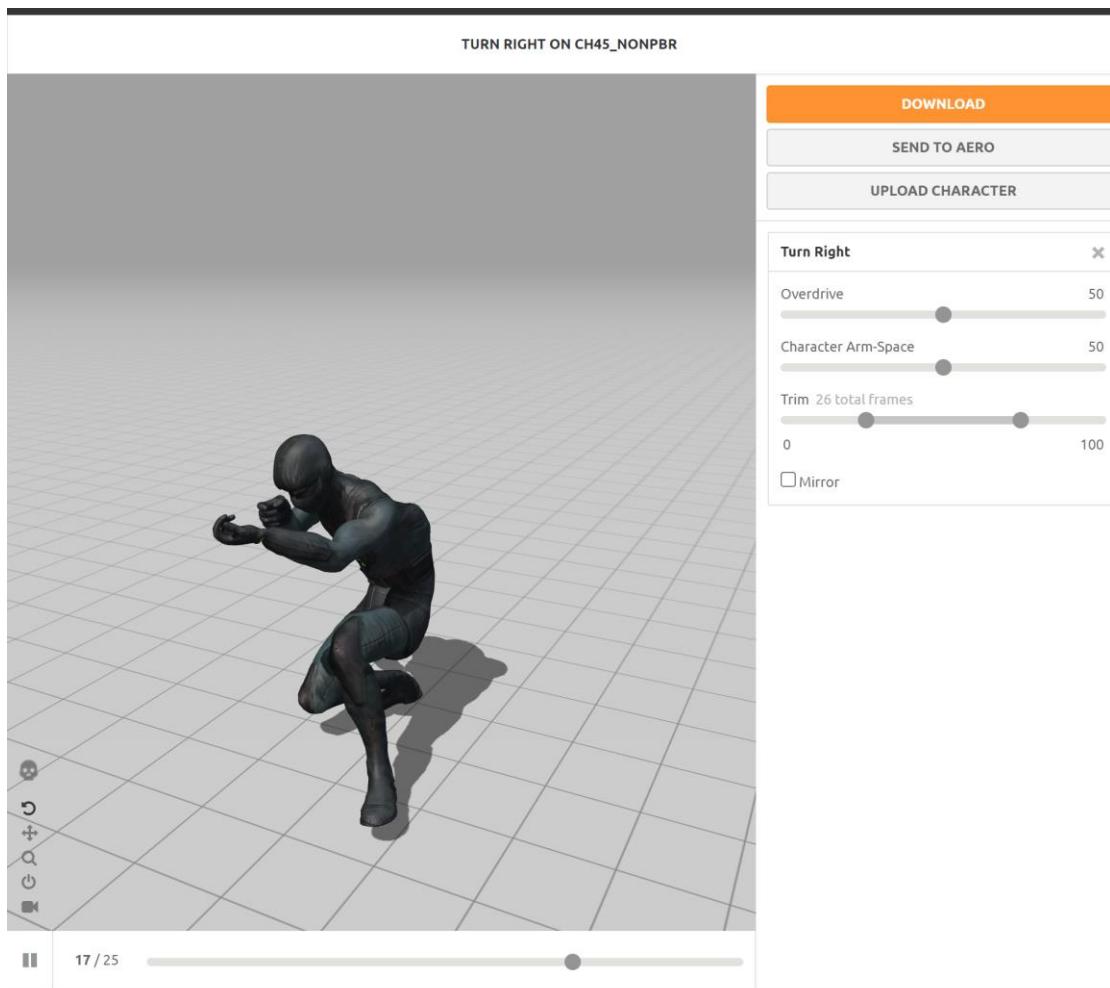


之后再在 mixamo 文件夹中创建一个新文件夹 Animations

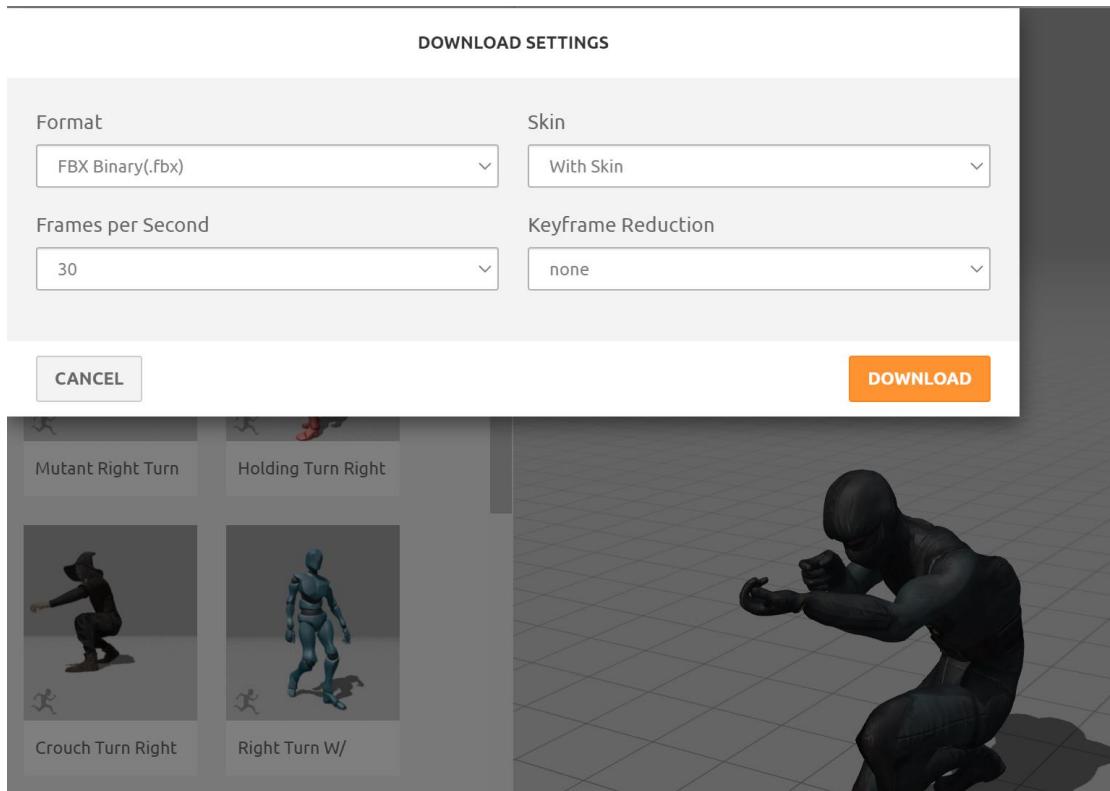


回到 mixamo 中，来到动画资源中查找 turn right

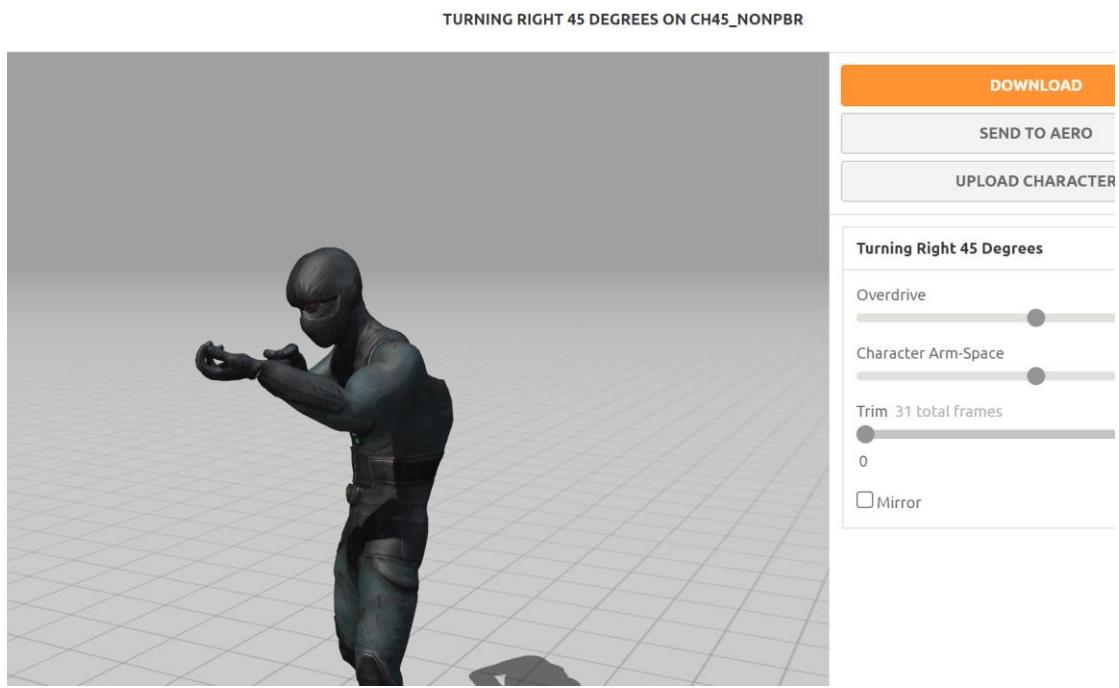
我们需要的是 in-place 的动画，因为我们在游戏逻辑中实现角色的旋转。



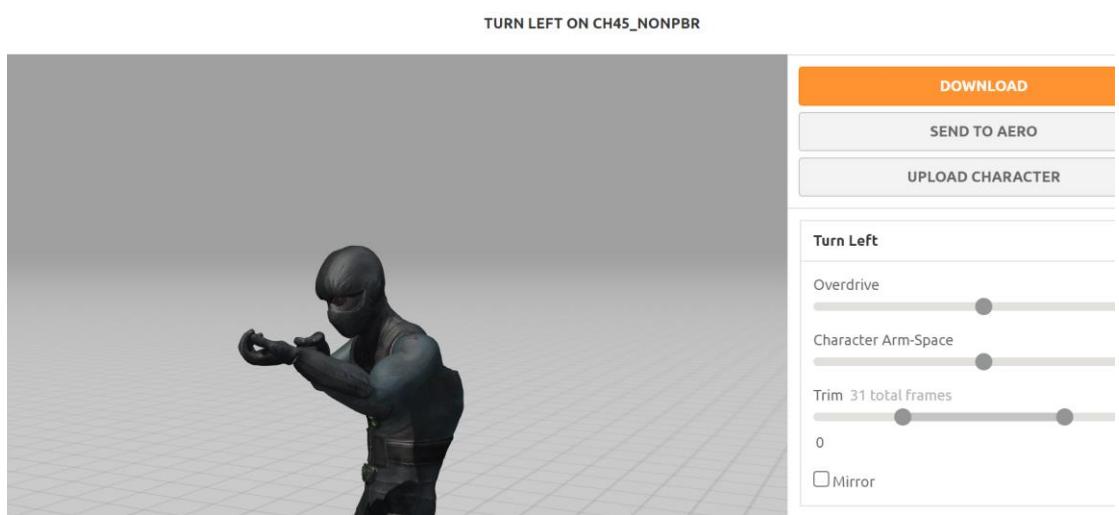
我们看到了一个蹲着的动画，点击下载：



继续往下拉，又看到了站着的动画：

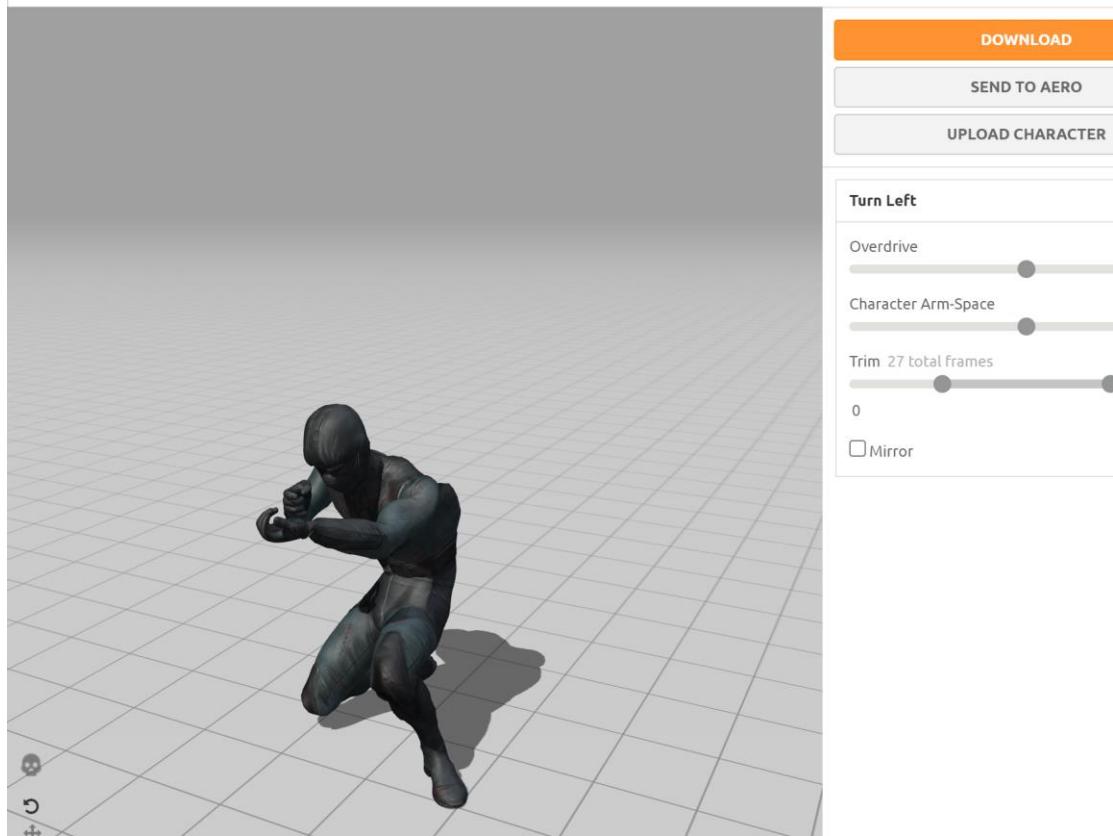


同理，搜索 turn left 的动画



还要下载一个蹲着的动画：

TURN LEFT ON CH45_NONPBR



接下来下载跳跃的动画：

JUMP UP ON CH45_NONPBR

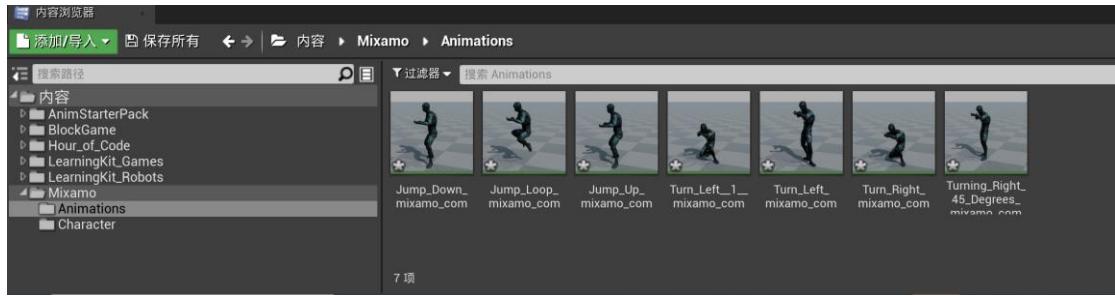


JUMP DOWN ON CH45_NONPBR



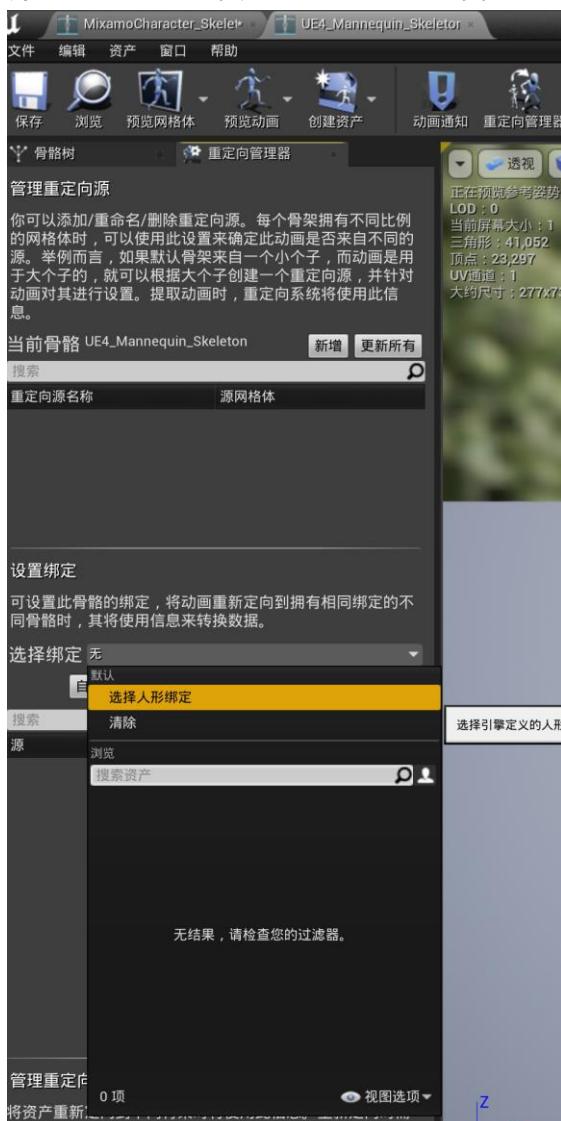


删除多余的 Tpose 动画，剩下的就是我们需要的动画：

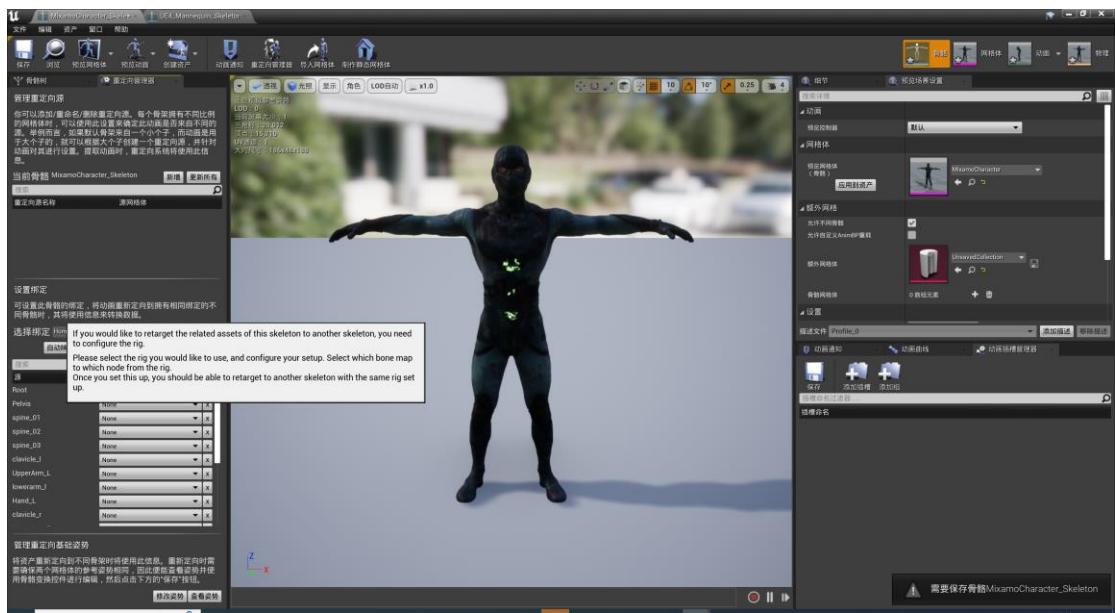


重定向

为了将 mixamo 的动画重定向到 UE 的小白人上，我们需要做一些事情：



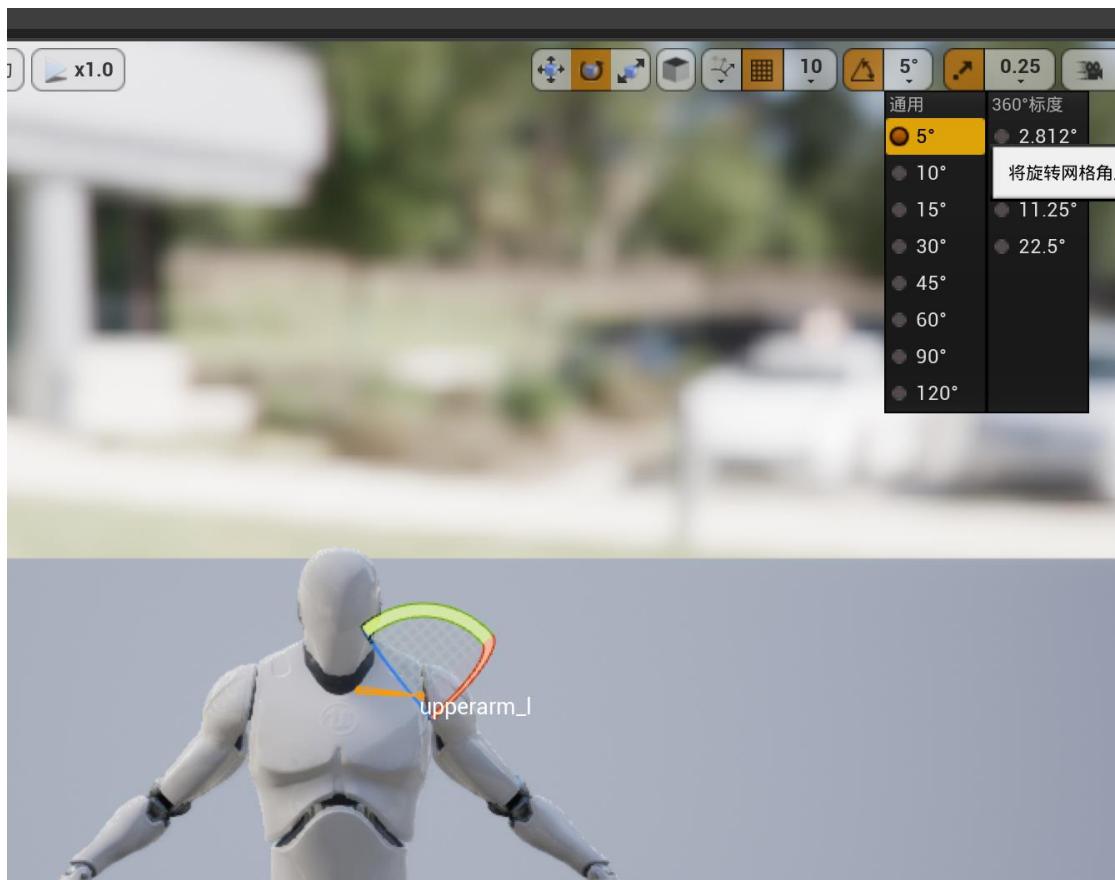
打开小白人的骨骼，选择重定向管理器，【选择绑定人形】，同样操作用在 mixamo 的人物骨骼上。



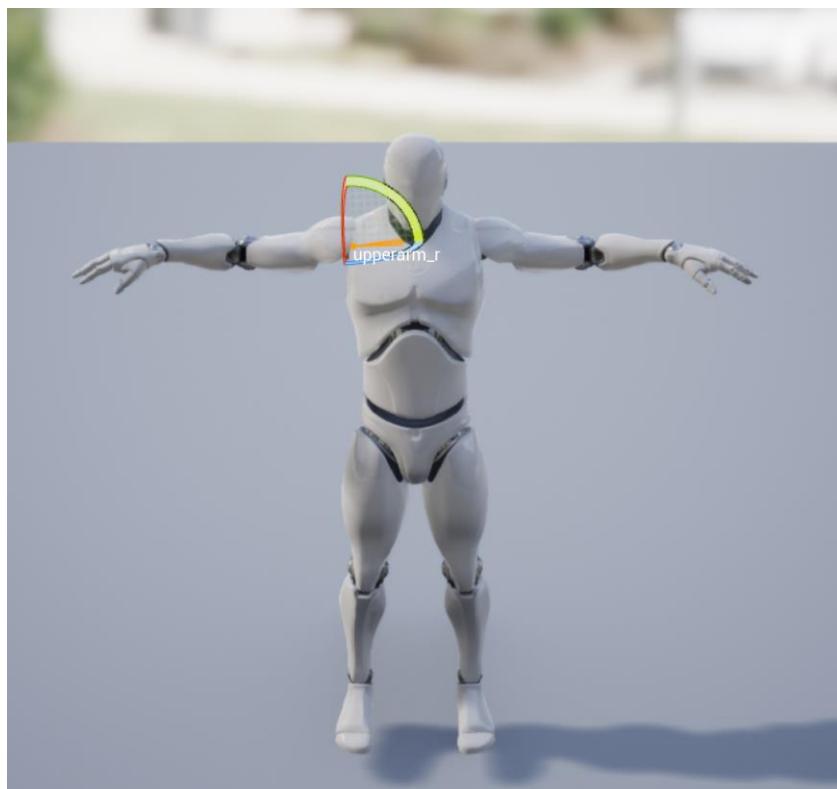
接下来需要绑定 mixamo 人物的对应人形骨骼。



接下来由于 UE 小白人默认采用了 T-pose，而 mixamo 默认采用了 A-pose，所以我们选择吧 UE 小白人的造型更改为 T-pose，之后再进行动画重定向。

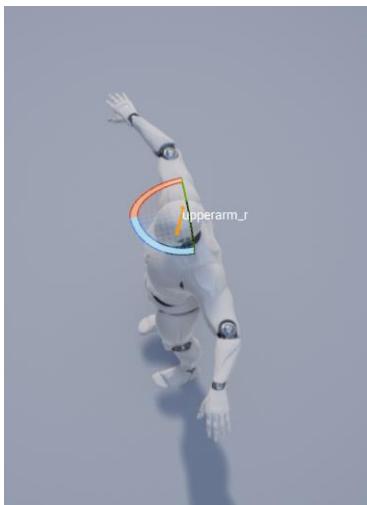


选择 upperarm_l 骨骼，将每次旋转角度设为 5 度，将其向上旋转 45 度。



对右手进行同样操作。

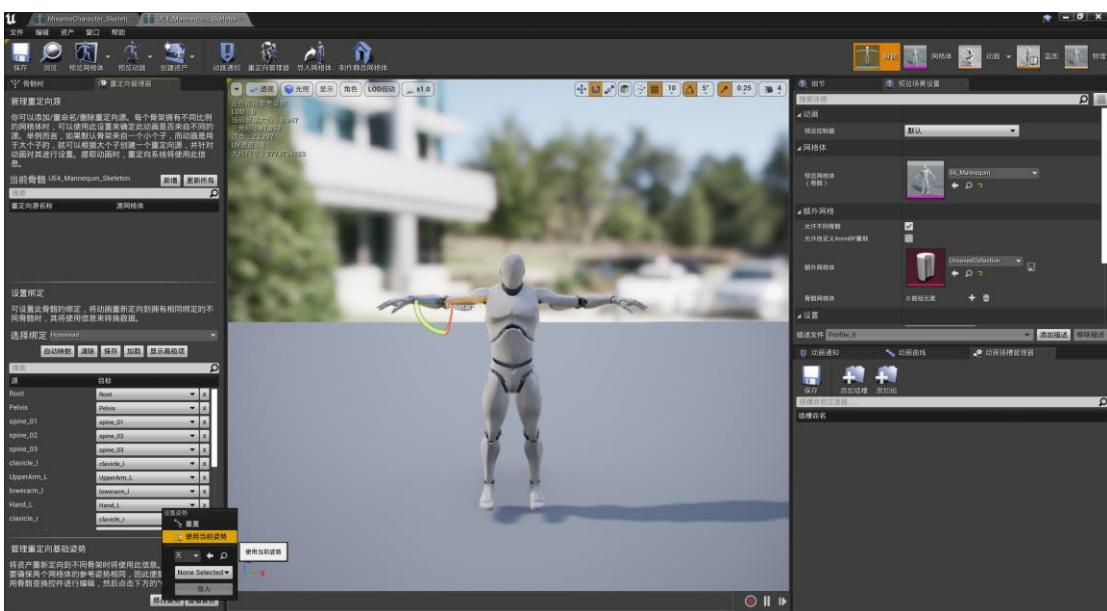
然后将视角朝向下方，我们可以发现，手臂并不是直的。



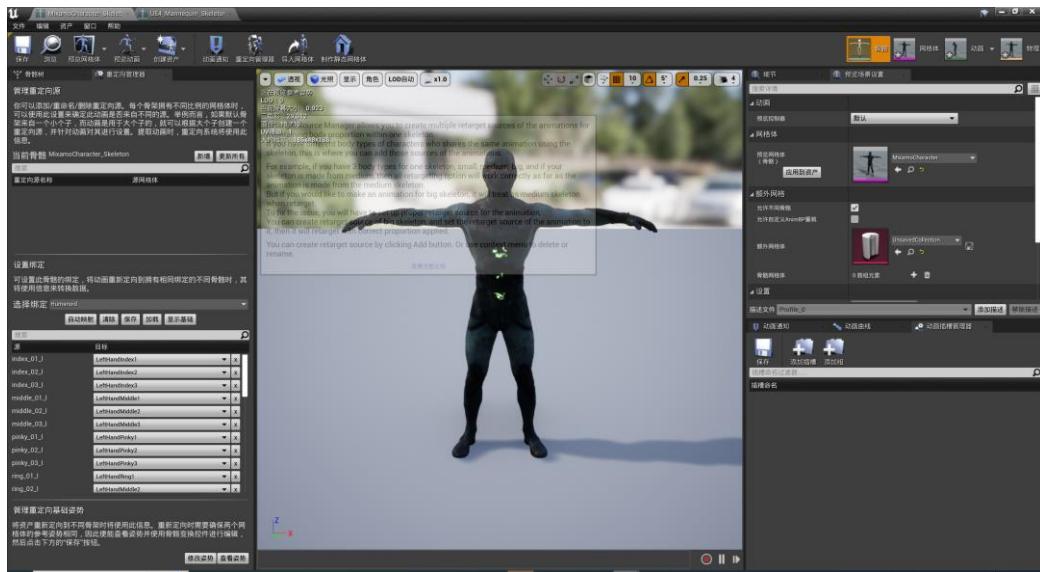
所以我们还要再旋转手的小臂



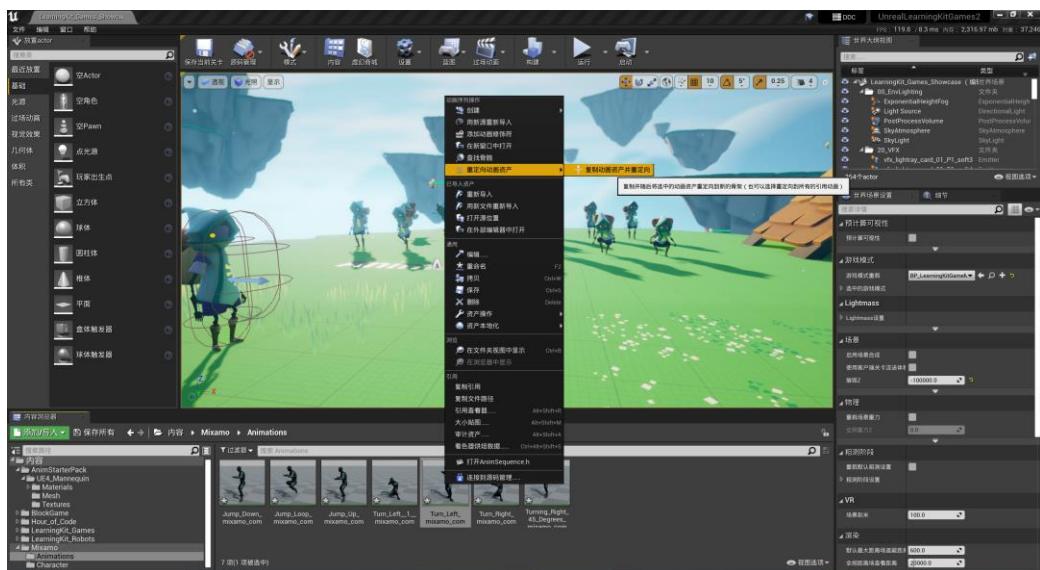
将两边小臂各旋转 10 度。



设置完成后回到重定向管理器，选择使用当前姿势。



回到 mixamo 角色，点击左边的应用到资产。



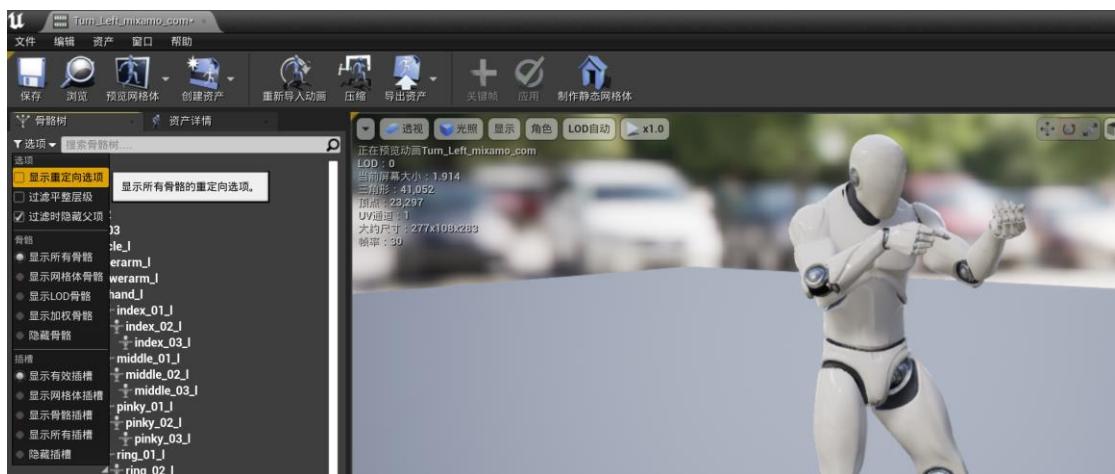
之后就可以选择重定向动画。



选择 ue 小白人骨骼并重定向。



可以点击查看我们重定向的效果，如果有个别骨骼特别奇怪，可以回头查看是否在设置骨骼对应的时候出现了错误。

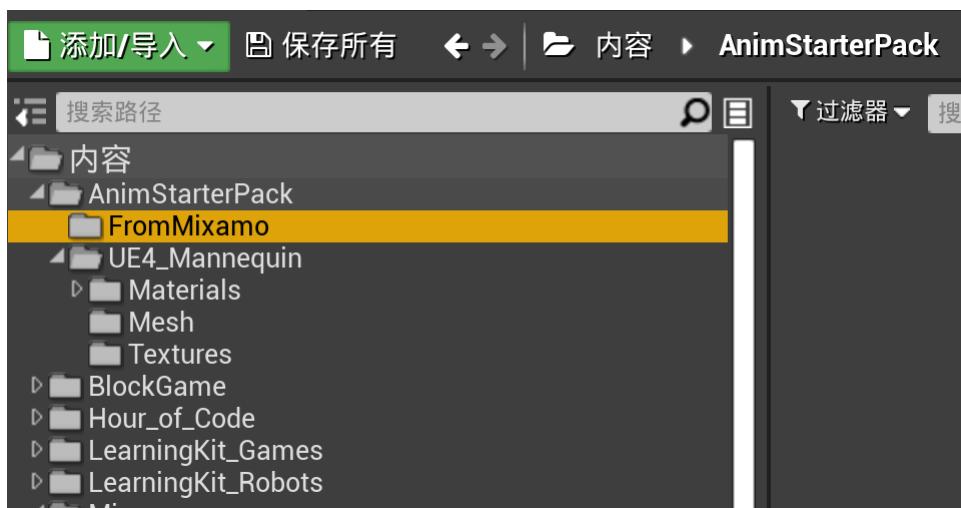


点击 root 右键选择

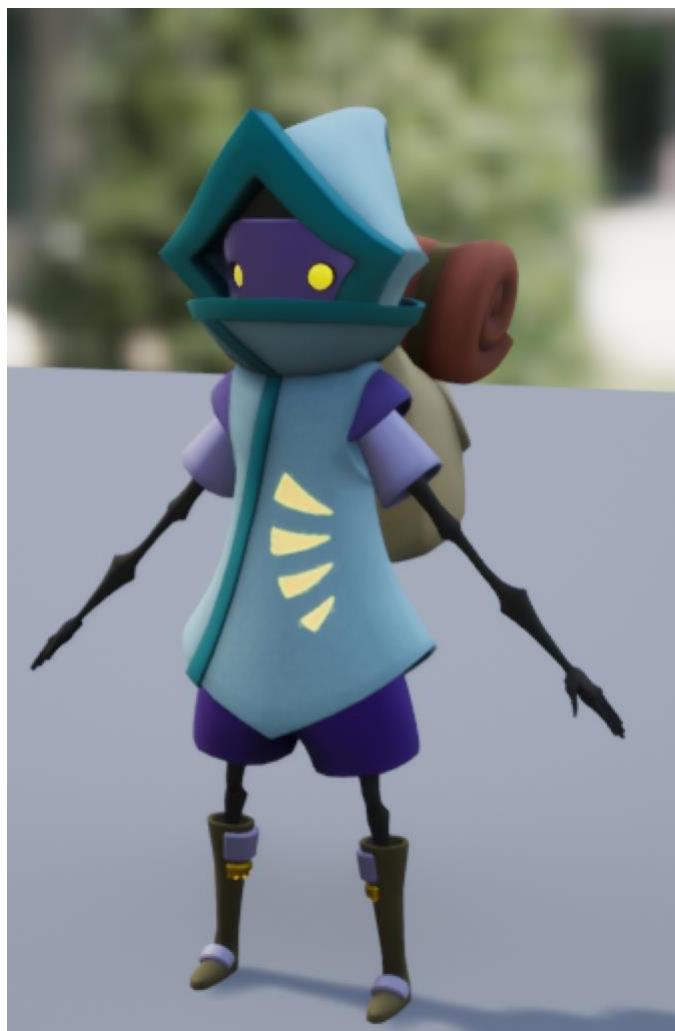


之后设置 pelvis 为动画缩放，再设置 root 为动画。

确认没问题后将所有的动画重定向，之后创建一个新的文件夹用于保存重定向的动画：



我们不直接重定向到学习包中的角色是因为学习包中的角色有些不同：



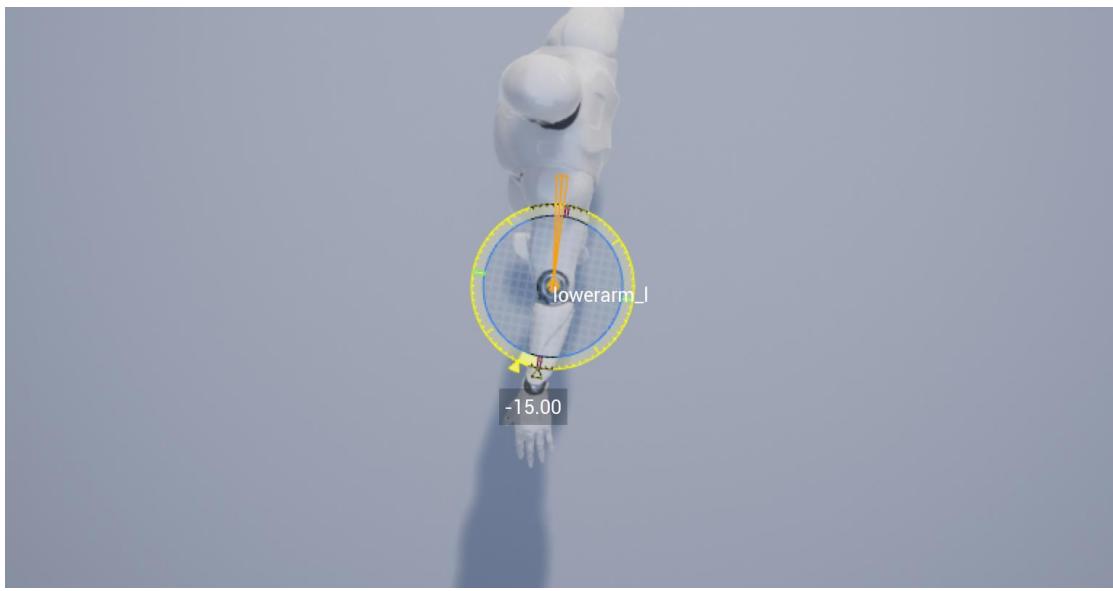
他有一个直的手臂，所以重定向之前需要确保其设置的正确。



同样将其手臂宣传 45 度成 T-pose，之后设置重定向管理器使用当前造型。



将我们之前设置的小白人两个手臂向下旋转 10 度。

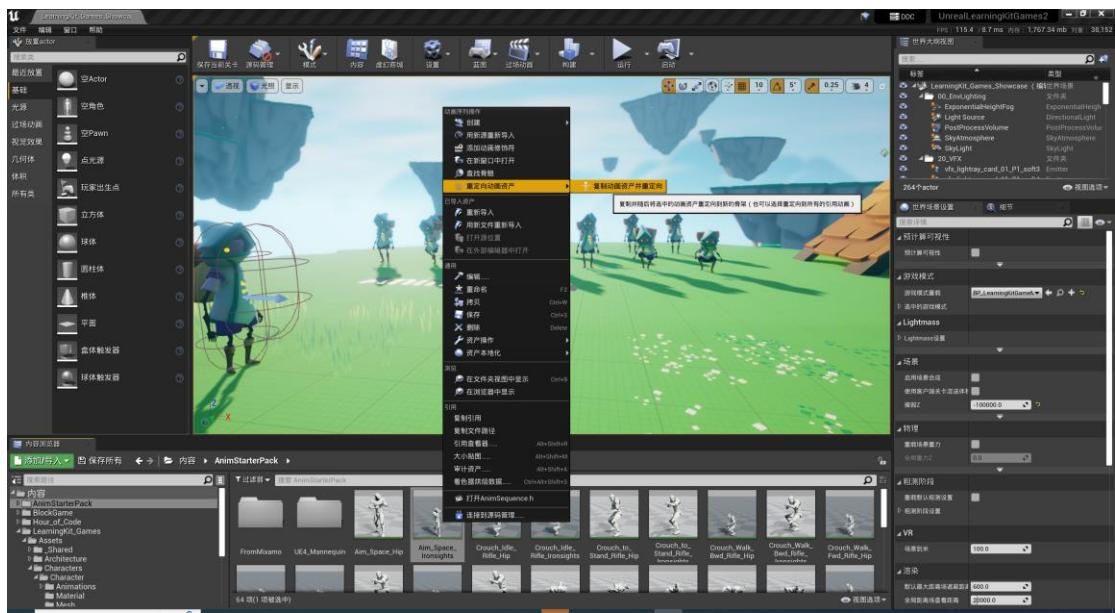


小臂则再往后 15 度

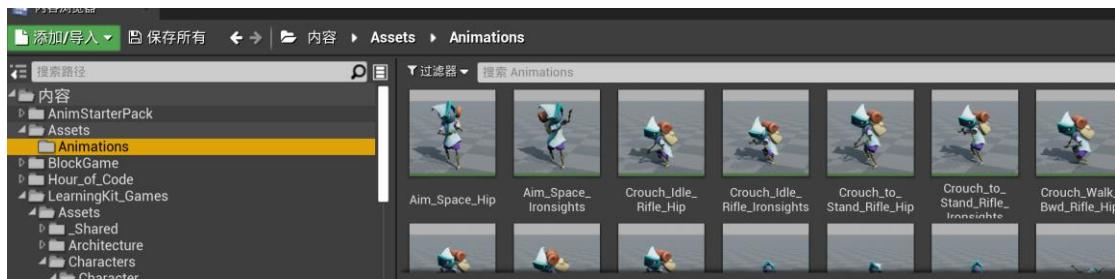
回到 AnimStarterPack,删除除了动画以外的文件。



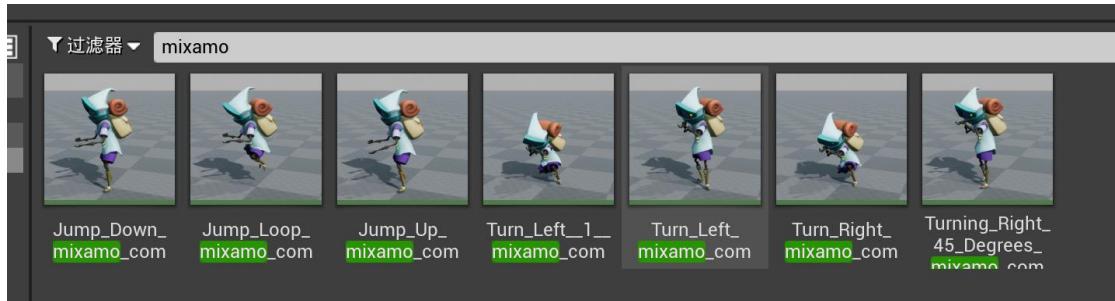
选择一个动画重定向：



瞄准动作具有代表性，所以我们选择瞄准的动画，再确认没有问题之后我们可以重定向其他的所有动画。

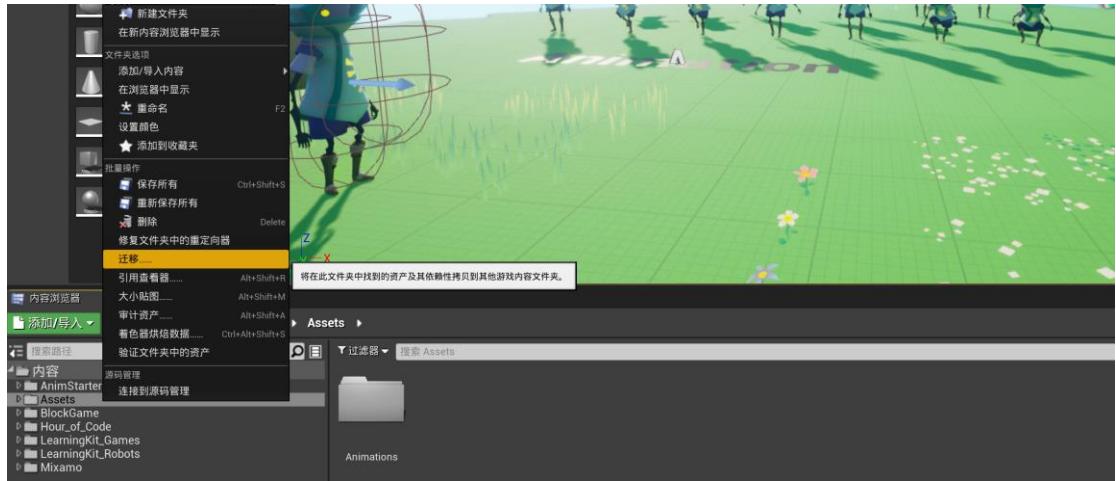


创建一个新文件夹，将动画全部移动到这里。搜索 mixamo 的动画，并重命名。



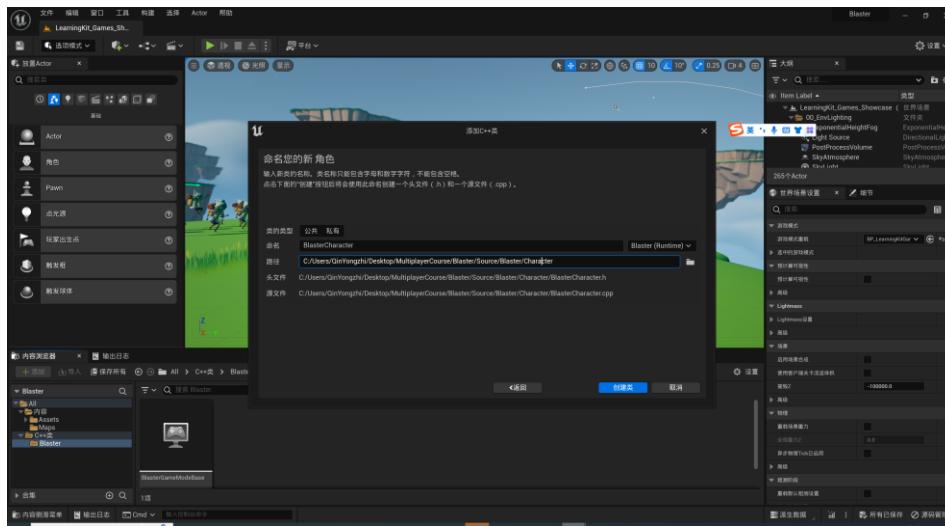
接下来进行迁移：





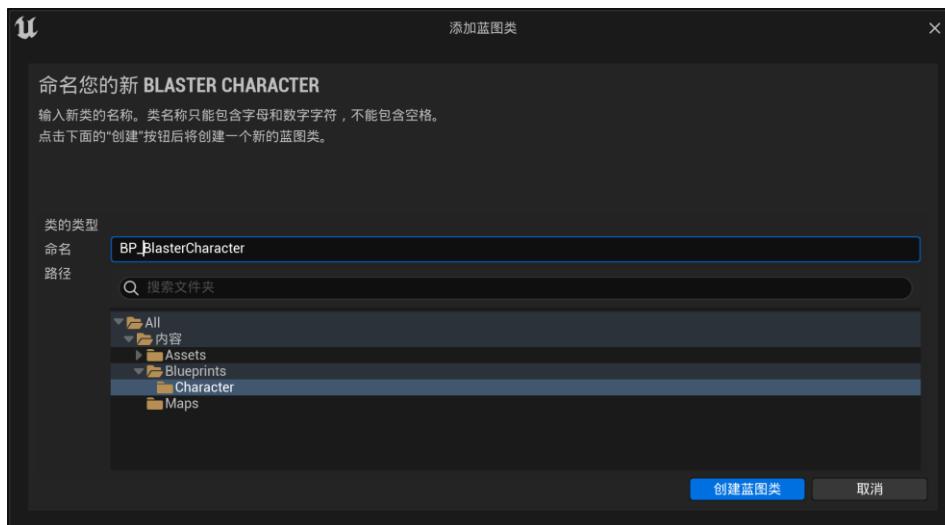
之后打开 Blaster，删除多余的不需要的文件即可。

030 Blaster 角色



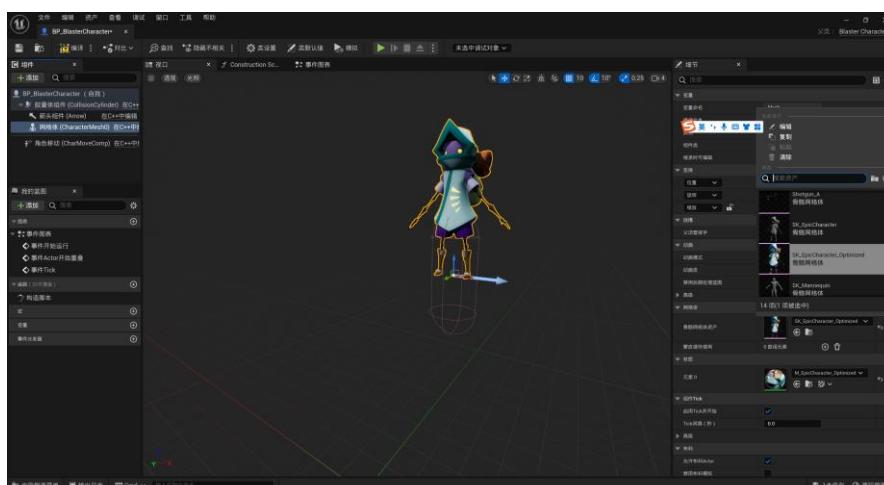
创建一个新 Character 类

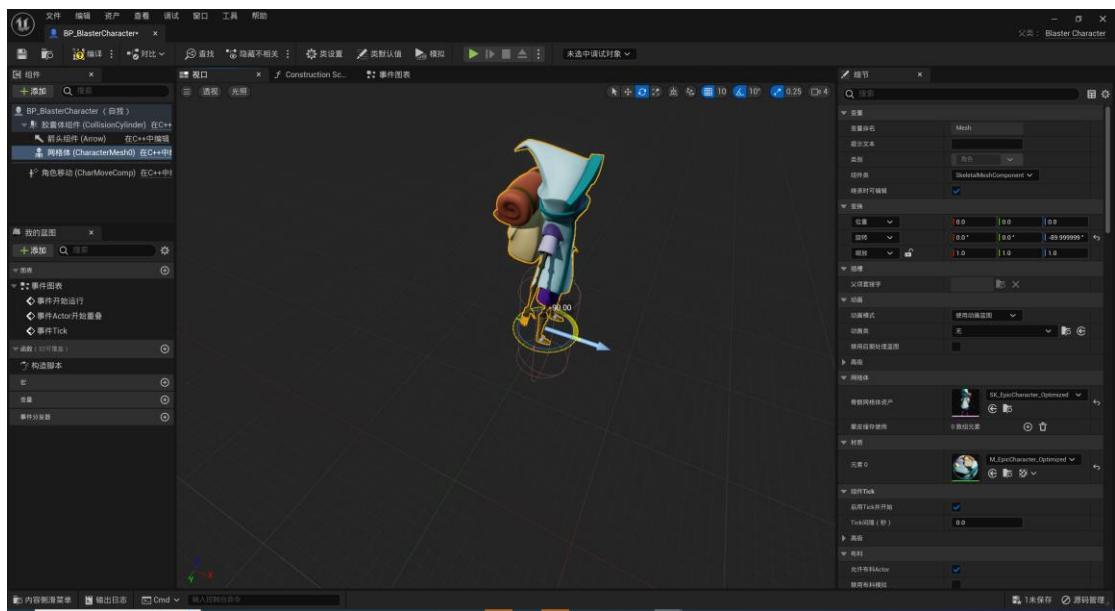
删除 VS 文件中的 include 中的 Character/这一段。



重新编译后创建文件夹，并用 C++ 类生成蓝图。

打开蓝图类，设置角色：



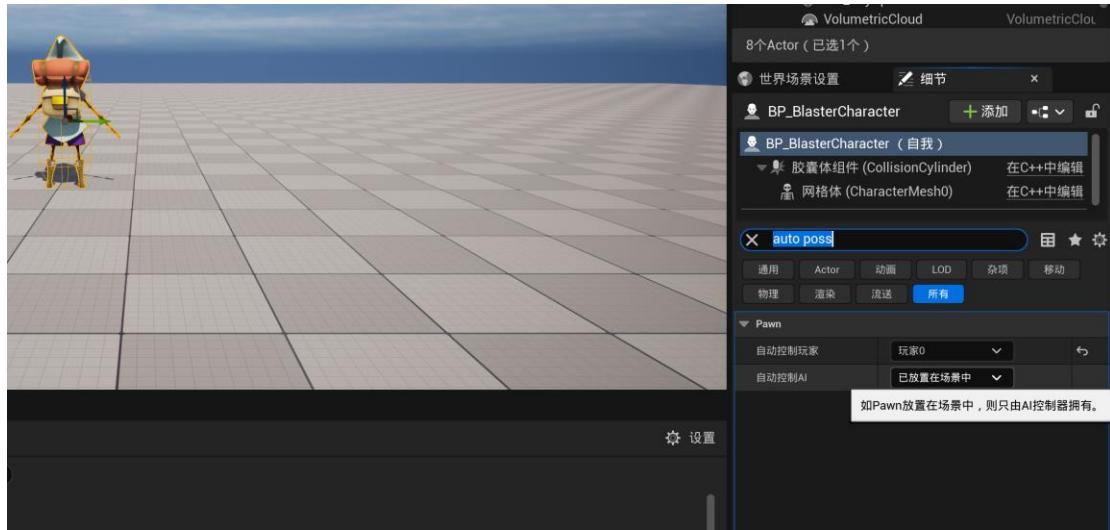


旋转角色正对正方向。



由于默认胶囊体半高为 88，所以这里设置位移-88。

031 相机与弹簧臂



在 Lobby 中放置一个 Character 并设置自动控制玩家。

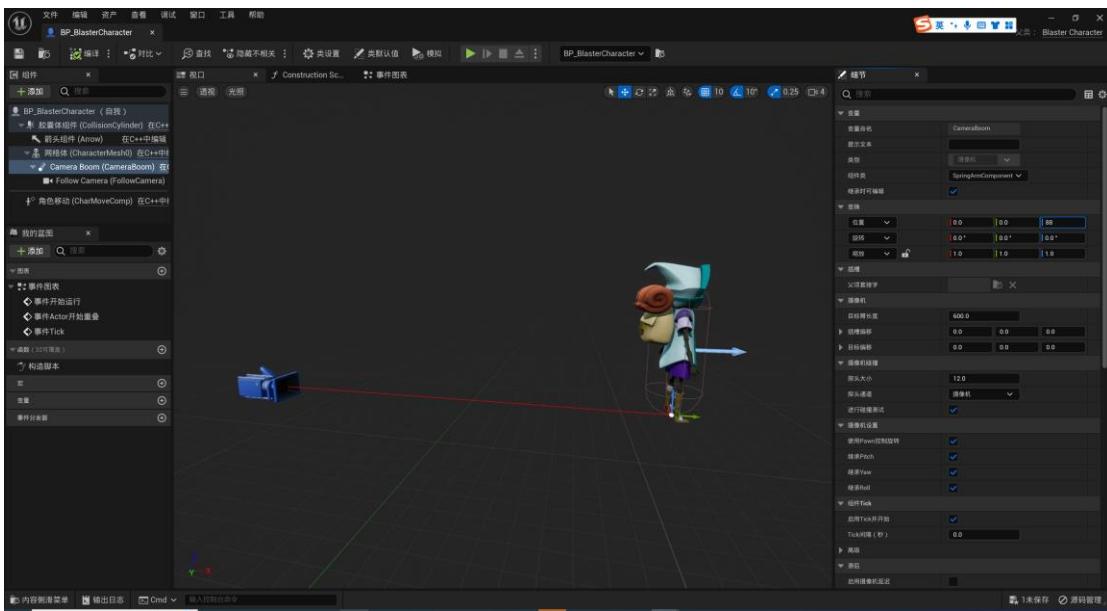
在 BlasterCharacter.h 中添加相机和弹簧臂成员：

```
private:  
    UPROPERTY(VisibleAnywhere, Category = Camera)  
    class USpringArmComponent* CameraBoom;  
  
    UPROPERTY(VisibleAnywhere, Category = Camera)  
    class UCamerComponent* FollowCamera;
```

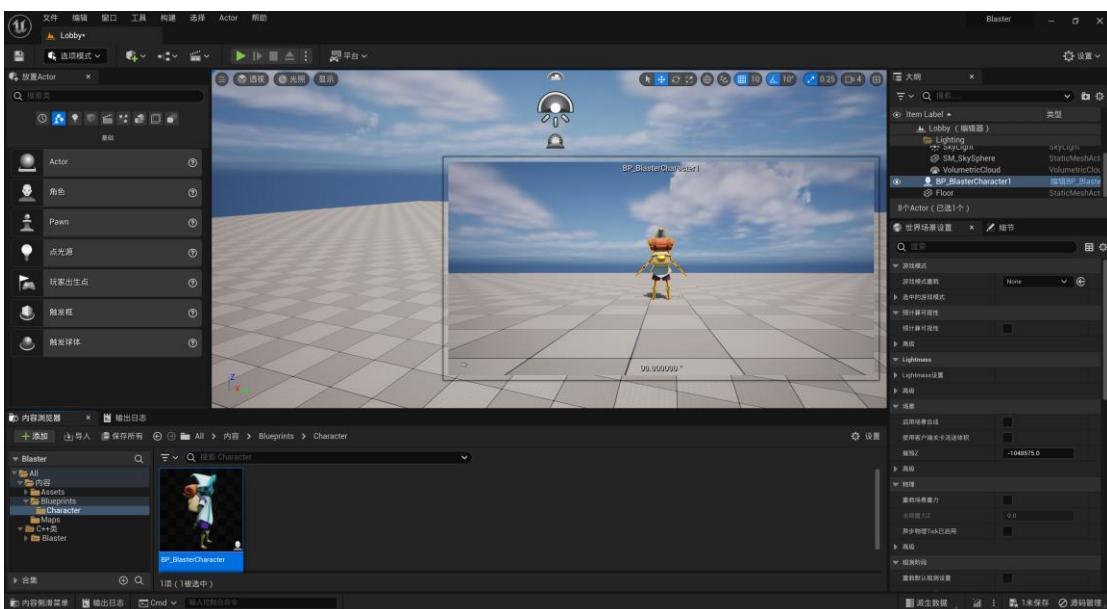
在 BlasterCharacter.cpp 中添加头文件并修改构造函数：

```
#include "GameFramework/SpringArmComponent.h"  
#include "Camera/CameraComponent.h"  
  
ABlasterCharacter::ABlasterCharacter()  
{  
    PrimaryActorTick.bCanEverTick = true;  
  
    CameraBoom = CreateDefaultSubobject<USpringArmComponent>(TEXT("CameraBoom"));  
    CameraBoom->SetupAttachment(GetMesh());  
    CameraBoom->TargetArmLength = 600.f;  
    CameraBoom->bUsePawnControlRotation = true;  
  
    FollowCamera = CreateDefaultSubobject<UCamerComponent>(TEXT("FollowCamera"));  
    FollowCamera->SetupAttachment(CameraBoom, USpringArmComponent::SocketName);  
    FollowCamera->bUsePawnControlRotation = false;  
}
```

编译，并运行。



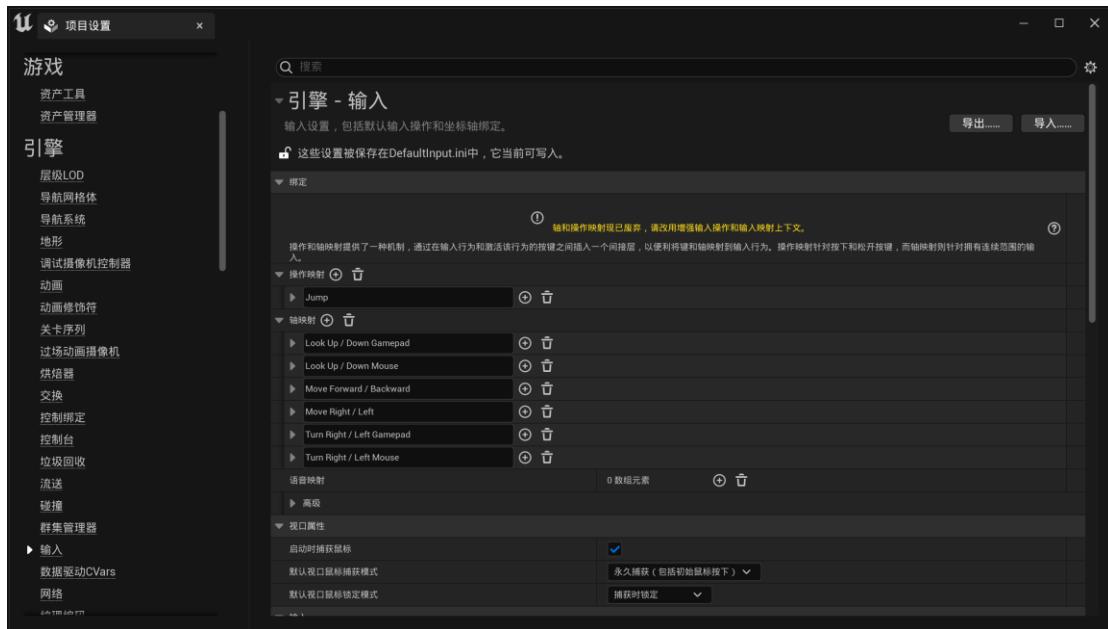
打开角色蓝图，设置相机弹簧臂高度为 88



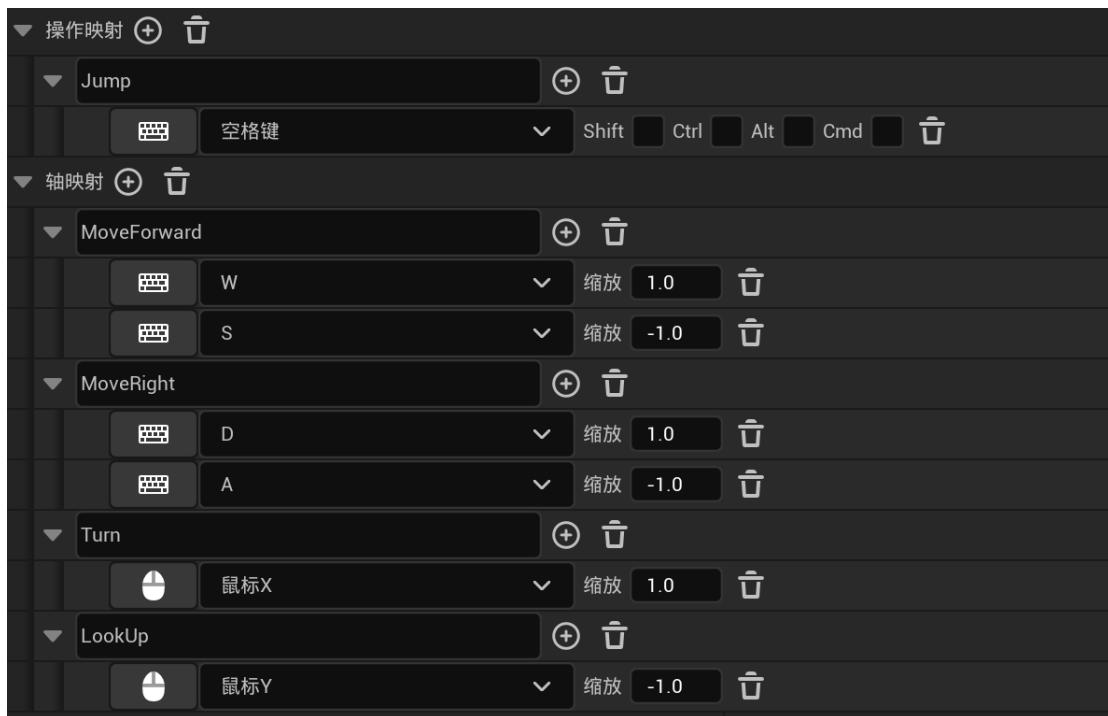
现在的角色还不能移动，因为我们还没有做那部分的功能，我们将在下一节做这些功能。

032 角色移动

配置我们的输入



打开项目设置-引擎-输入：



如图设置，之后可以在函数中绑定这些输入。

移动功能

在.h 文件中创建 4 个 protected 函数，来处理输入：

```
void MoveForward(float Value);
void MoveRight(float Value);
void Turn(float Value);
void LookUp(float Value);
```

在.cpp 文件中，在 SetupPlayerInputComponent 中完成绑定，并完成输入函数的实现。

```
void ABlasterCharacter::SetupPlayerInputComponent(UInputComponent*
PlayerInputComponent)
{
    Super::SetupPlayerInputComponent(PlayerInputComponent);

    PlayerInputComponent->BindAction("Jump", IE_Pressed, this, &ACharacter::Jump);

    PlayerInputComponent->BindAxis("MoveForward", this, &ABlasterCharacter::MoveForward);
    PlayerInputComponent->BindAxis("MoveRight", this, &ABlasterCharacter::MoveRight);
    PlayerInputComponent->BindAxis("Turn", this, &ABlasterCharacter::Turn);
    PlayerInputComponent->BindAxis("LookUp", this, &ABlasterCharacter::LookUp);
}

void ABlasterCharacter::MoveForward(float Value)
{
    if (Controller != nullptr && Value != 0.f)
    {
        const FRotator YawRotation(0.f, Controller->GetControlRotation().Yaw, 0.f);
        const FVector Direction(FRotationMatrix(YawRotation).GetUnitAxis(EAxis::X));
        AddMovementInput(Direction, Value);
    }
}

void ABlasterCharacter::MoveRight(float Value)
{
    if (Controller != nullptr && Value != 0.f)
    {
        const FRotator YawRotation(0.f, Controller->GetControlRotation().Yaw, 0.f);
        const FVector Direction(FRotationMatrix(YawRotation).GetUnitAxis(EAxis::Y));
        AddMovementInput(Direction, Value);
    }
}

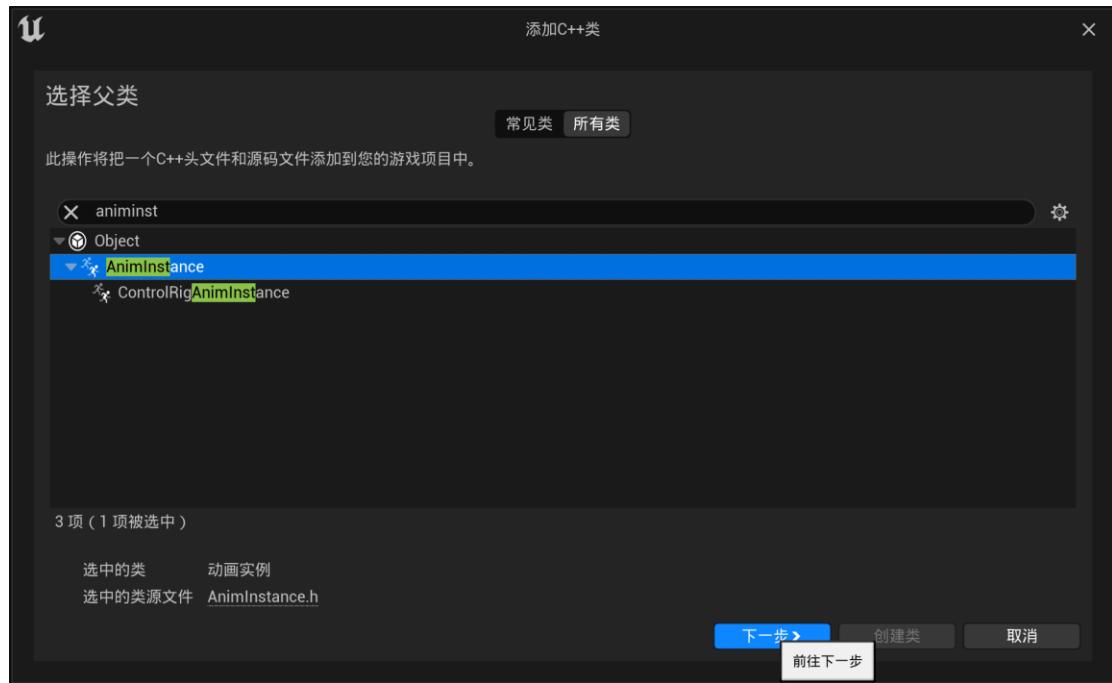
void ABlasterCharacter::Turn(float Value)
{
```

```
    AddControllerYawInput(Value);  
}  
  
void ABlasterCharacter::LookUp(float Value)  
{  
    AddControllerPitchInput(Value);  
}
```

编译，并运行。我们会发现上一节我们放在场景中的小人可以移动了。

033 动画蓝图

添加一个继承于 AnimInstance 的类：



在.h 文件中添加：

```
public:  
    virtual void NativeInitializeAnimation() override;  
    virtual void NativeUpdateAnimation(float DeltaTime) override;  
  
private:  
    UPROPERTY(BlueprintReadOnly, Category = Character, meta = (AllowPrivateAccess =  
"true"))  
    class ABlasterCharacter* BlasterCharacter;  
  
    UPROPERTY(BlueprintReadOnly, Category = Movement, meta = (AllowPrivateAccess =  
"true"))  
    float Speed;  
  
    UPROPERTY(BlueprintReadOnly, Category = Movement, meta = (AllowPrivateAccess =  
"true"))  
    bool bIsInAir;
```

```
    UPROPERTY(BlueprintReadOnly, Category = Movement, meta = (AllowPrivateAccess =  
"true"))  
    bool bIsAccelerating;
```

在.cpp 文件中添加：

```
#include "BlasterCharacter.h"  
#include "GameFramework/CharacterMovementComponent.h"  
void UBlasterAnimInstance::NativeInitializeAnimation()
```

```

{
    Super::NativeInitializeAnimation();

    BlasterCharacter = Cast<ABlasterCharacter>(TryGetPawnOwner());
}

void UBlasterAnimInstance::NativeUpdateAnimation(float DeltaTime)
{
    Super::NativeUpdateAnimation(DeltaTime);

    if (BlasterCharacter == nullptr)
    {
        BlasterCharacter = Cast<ABlasterCharacter>(TryGetPawnOwner());
    }
    if (BlasterCharacter == nullptr) return;

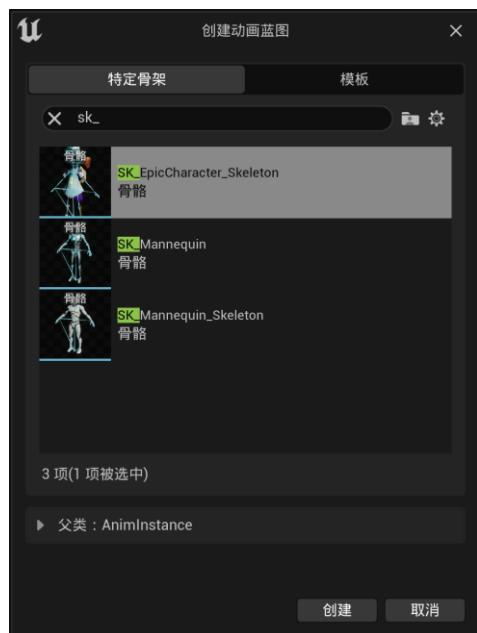
    FVector Velocity = BlasterCharacter->GetVelocity();
    Velocity.Z = 0.f;
    Speed = Velocity.Size();

    bIsInAir = BlasterCharacter->GetCharacterMovement()->IsFalling();
    bIsAccelerating =
    BlasterCharacter->GetCharacterMovement()->GetCurrentAcceleration().Size() > 0.f ?
    true : false;
}

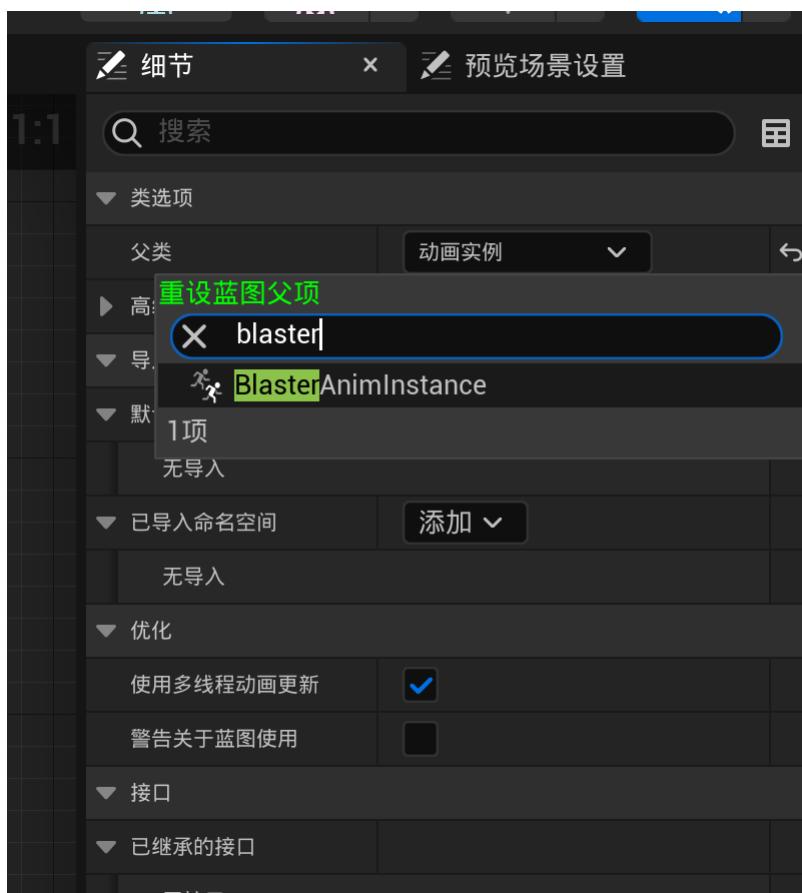
```

之后编译程序。

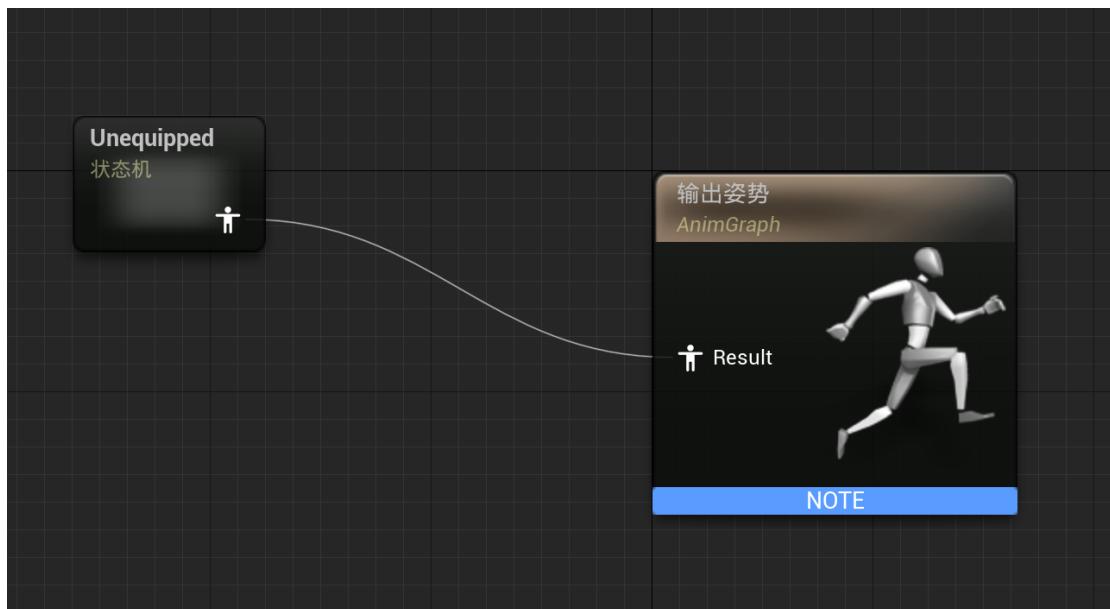
打开之后创建一个动画蓝图：

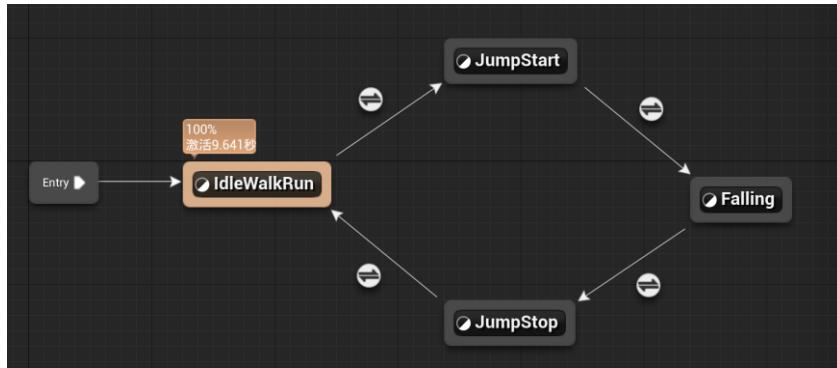


打开镭射纸，设置父类为我们自己的 C++ 类

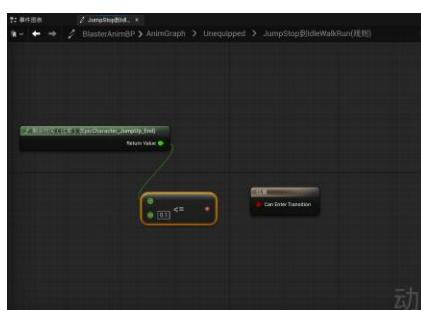


创建一个状态机：





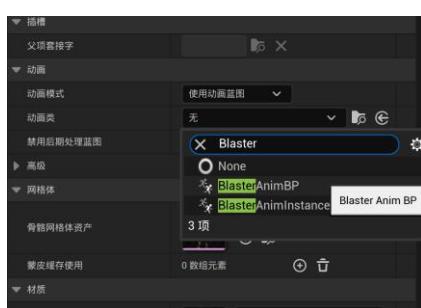
连一个教科书式的状态，然后设置每个状态的变化：



动画

然后给出了 idle 之外的状态添加动画。

之后制作一个混合空间并连接给 idleWalkRun 并保存
然后打开角色蓝图，设置动画蓝图。



034 无缝切换与大厅 (SEAMLESS TRAVEL AND LOBBY)

无缝切换：

在切换地图时不需要断开连接之后重连。

在 GameMode 中设置 bUseSeamlessTravel = true;

需要一个额外的切换地图，因为任何时间都需要有一个已经加载完成的地图。

UWorld::ServerTravel

- Server Only
- Jumps the Server tp a new level
- All connected clients will follow
- Server calls APlayerContriller::ClientTravel

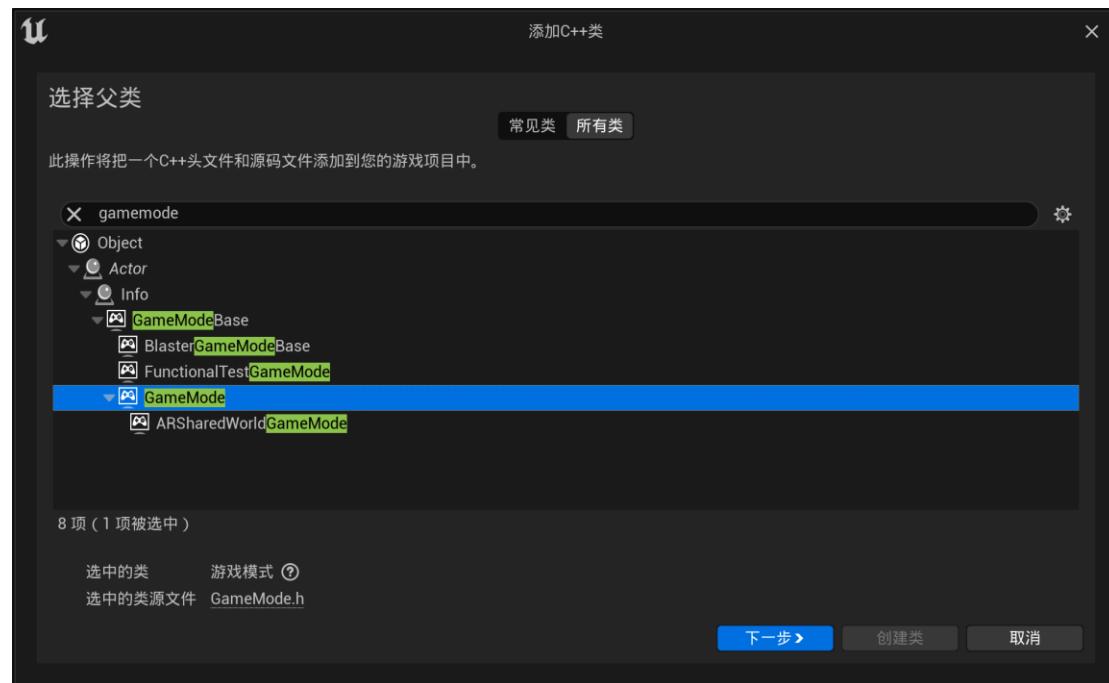
APlayerContriller::ClientTravel

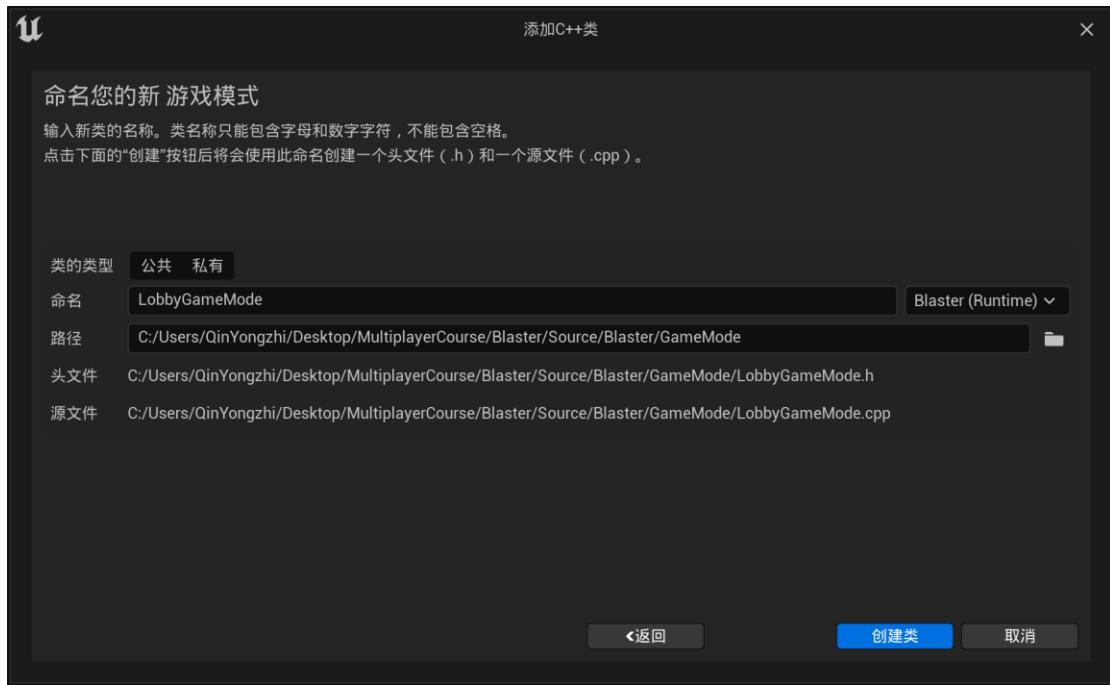
当发起者是客户端时，会加入一个新的服务器

当发起者是服务器时，会加入一个新的地图

因为 GameMode 只存在于服务器，所以我们使用 GameMode 来调用 ServerTravel。

创建一个 LobbyGameMode





.h 中:

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

.cpp 中:

```
#include "GameFramework/GameStateBase.h"
```

```
void ALobbyGameMode::PostLogin(APlayerController* NewPlayer)
```

```
{
```

```
    Super::PostLogin(NewPlayer);
```

```
    int32 NumberOfPlayers = GameState.Get()>PlayerArray.Num();
```

```
    if (NumberOfPlayers == 2)
```

```
{
```

```
        UWorld* World = GetWorld();
```

```
        if (World)
```

```
{
```

```
            bUseSeamlessTravel = true;
```

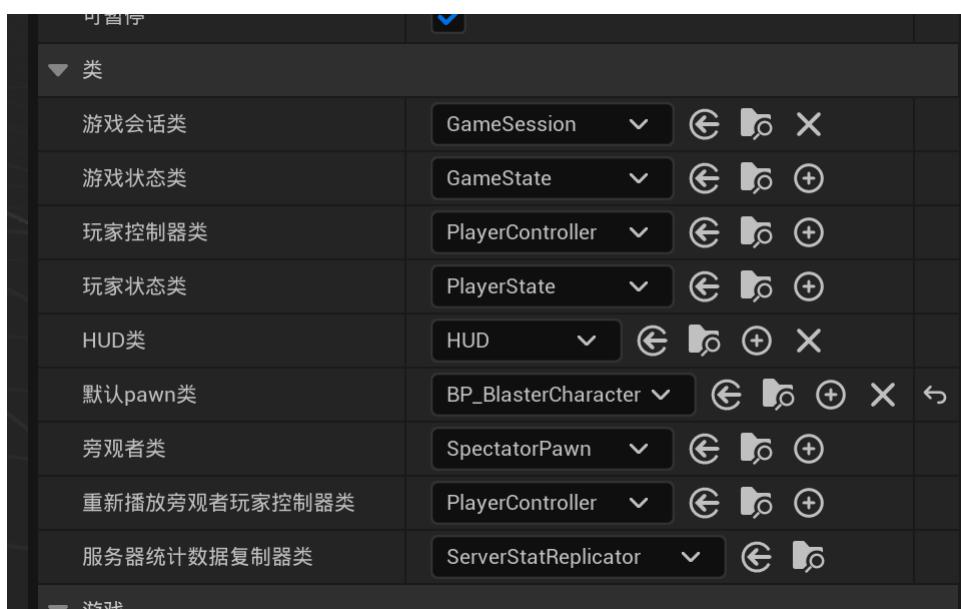
```
            World->ServerTravel(FString("/Game/Maps/BlasterMap?listen"));
```

```
}
```

```
}
```

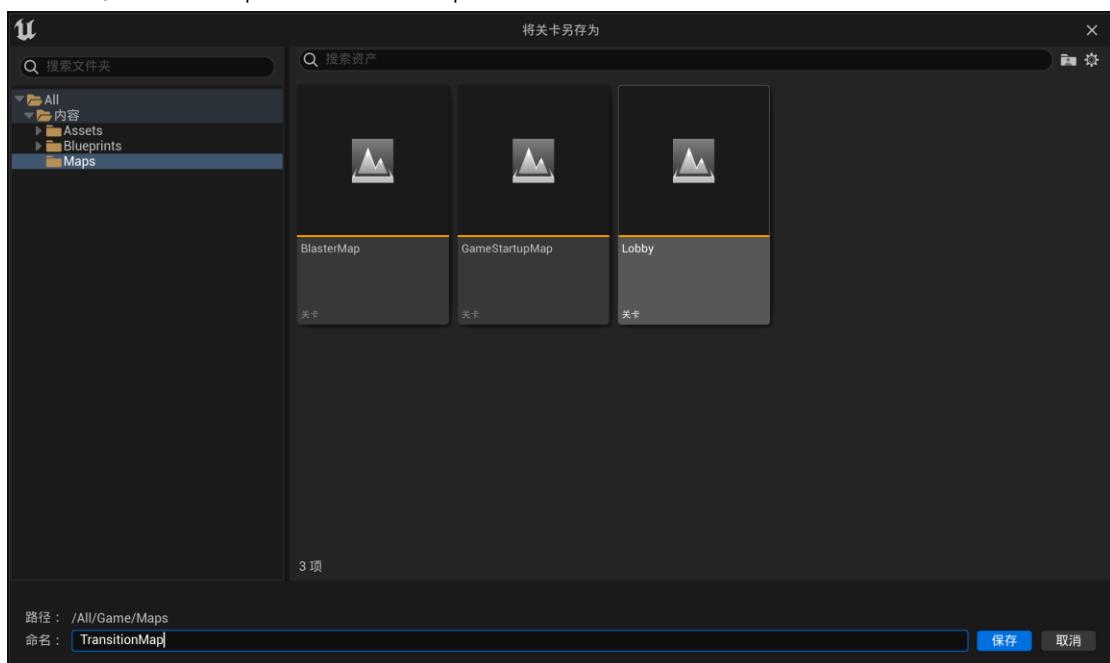
```
}
```

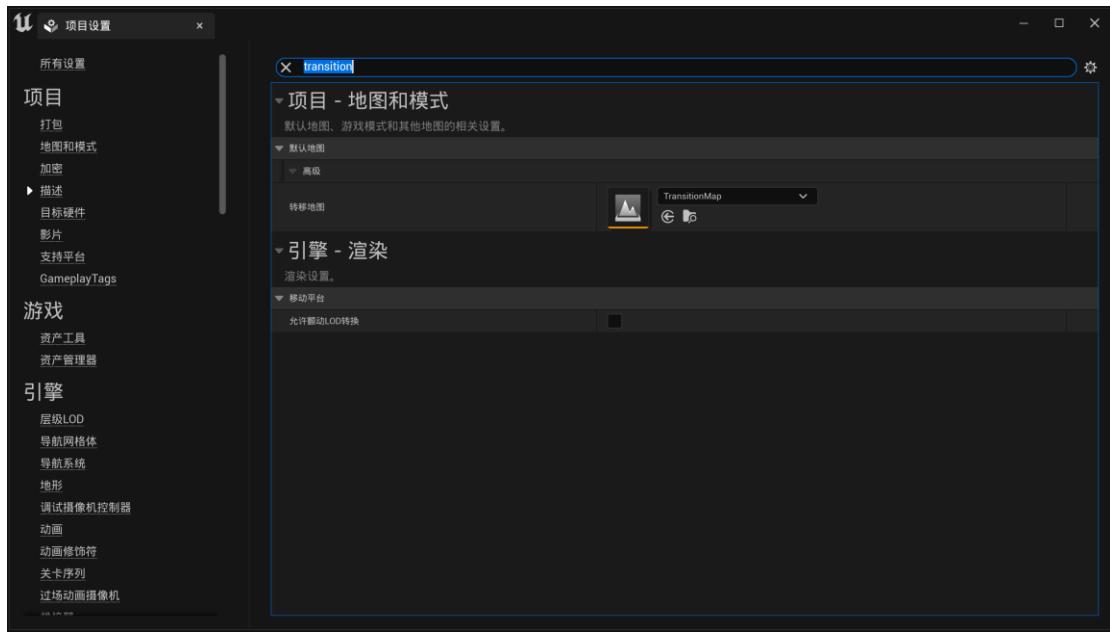
创建 gamemode 蓝图类：



修改 gamemode 蓝图中的默认 pawn，并勾选无缝转移。

创建一个 BlasterMap 和 transitionmap





在项目设置中设置转移地图。

Q1:

TObjectPtr?

035 网络角色

当我们在玩多人游戏的时候我们的客户端会有一个我们控制的 Pawn，在服务端也会有一个 Pawn，在其他客户端也会有一个 Pawn 的复制。

UE 中有一个 ENetRole 来鉴别每个 Character 或 Pawn 的网络角色。

ENetRole::ROLE_Authority 存在于服务端的权威角色。

ENetRole::ROLE_SimulateProxy 说明这个 Character 在一个不控制它的机器上。

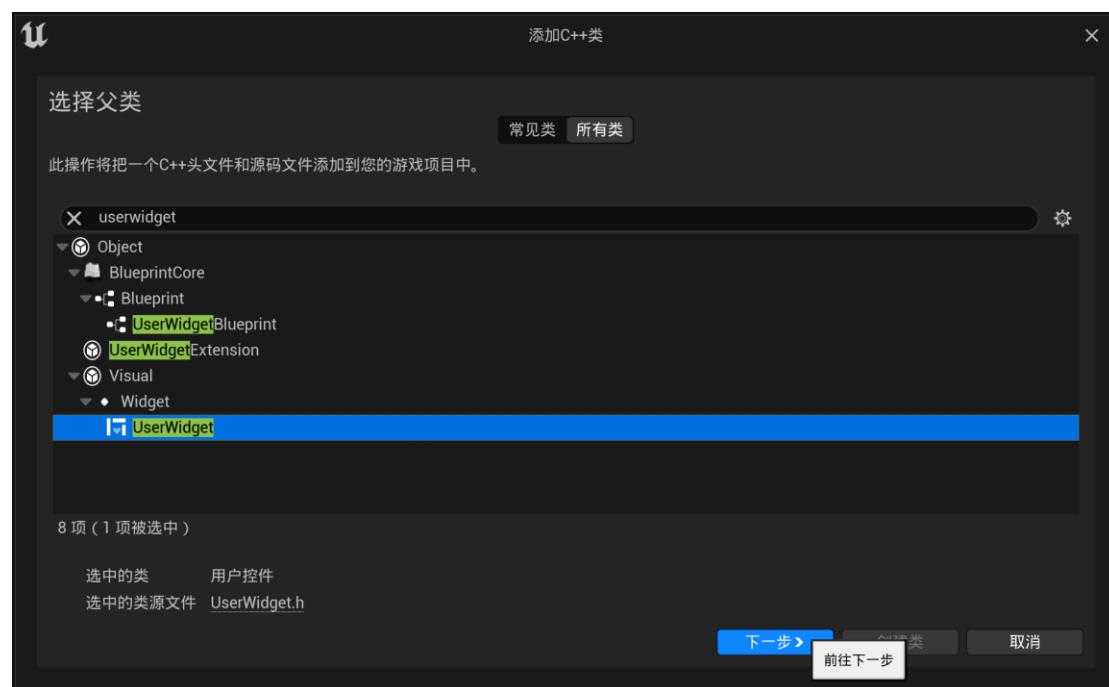
ENetRole::ROLE_AutonomousProxy 自己控制的角色。

ENetRole::ROLE_None

GetRemoteRole()在客户端调用的时候返回的是服务器上的该角色的 ENetRole::ROLE_Authority，而在服务端调用的时候返回的是 ENetRole::ROLE_AutonomousProxy 和 ENetRole::ROLE_SimulateProxy

GetLocalRole() 在客户端调用的时候返回的是 ENetRole::ROLE_AutonomousProxy 和 ENetRole::ROLE_SimulateProxy，而在服务端调用时，返回的是 ENetRole::ROLE_Authority

创建一个 UserWidget 的子类：



.h 中：

`public:`

`void SetDisplayText(FString TextToDisplay);`

`UFUNCTION(BlueprintCallable)`

`void ShowPlayerNetRole(APawn* InPawn);`

`public:`

```

UPROPERTY(meta = (BindWidget))
class UTextBlock* DisplayText;

protected:
    virtual void NativeDestruct() override;

.cpp 中:
#include "Components/TextBlock.h"

void UOverheadWidget::SetDisplayText(FString TextToDisplay)
{
    if (DisplayText)
    {
        DisplayText->SetText(FText::FromString(TextToDisplay));
    }
}

void UOverheadWidget::ShowPlayerNetRole(APawn* InPawn)
{
    ENetRole RemoteRole = InPawn->GetRemoteRole();
    FString Role;
    switch (RemoteRole)
    {
        case ENetRole::ROLE_Authority:
            Role = FString("Authority");
            break;
        case ENetRole::ROLE_AutonomousProxy:
            Role = FString("AutonomousProxy");
            break;
        case ENetRole::ROLE_SimulatedProxy:
            Role = FString("SimulatedProxy");
            break;
        case ENetRole::ROLE_None:
            Role = FString("None");
            break;
    }
    FString RemoteRoleString = FString::Printf(TEXT("Remote Role: %s"), *Role);
    SetDisplayText(RemoteRoleString);
}

void UOverheadWidget::NativeDestruct()
{
    RemoveFromParent();
}

```

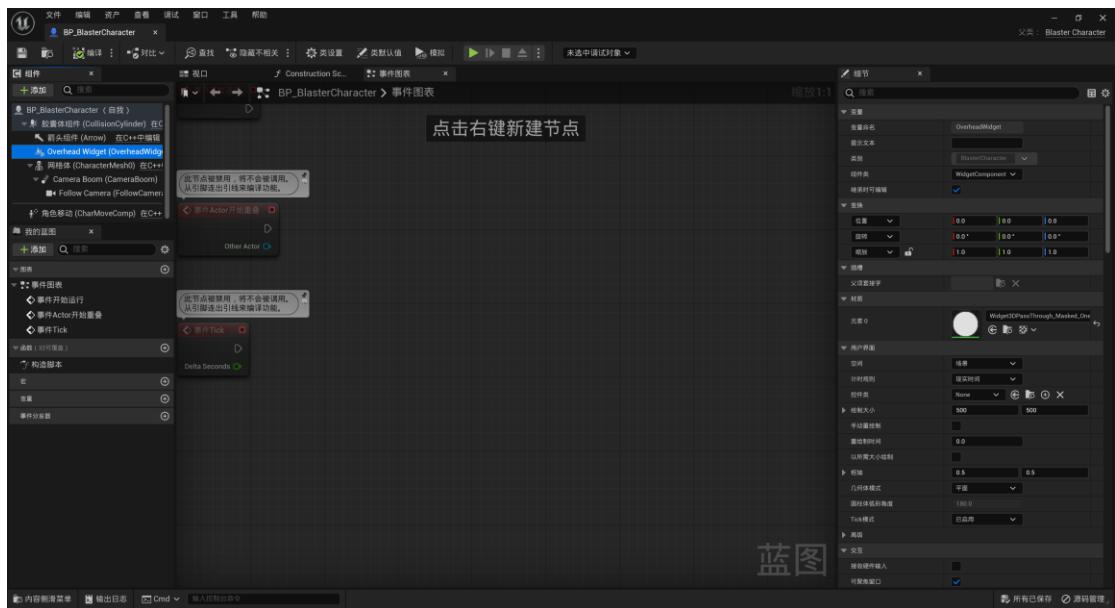
```

    Super::NativeDestruct();
}

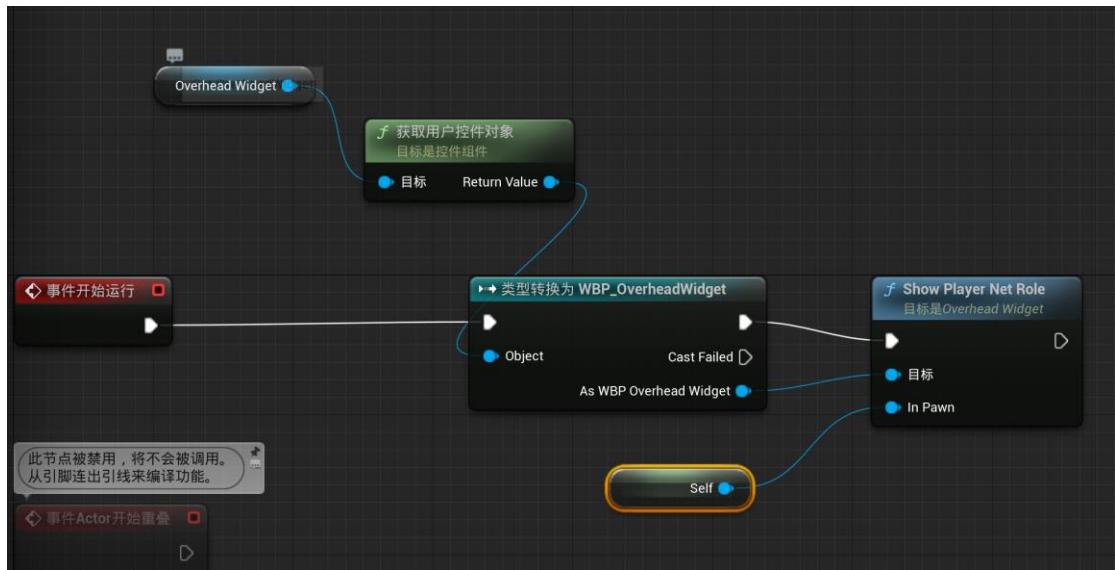
```

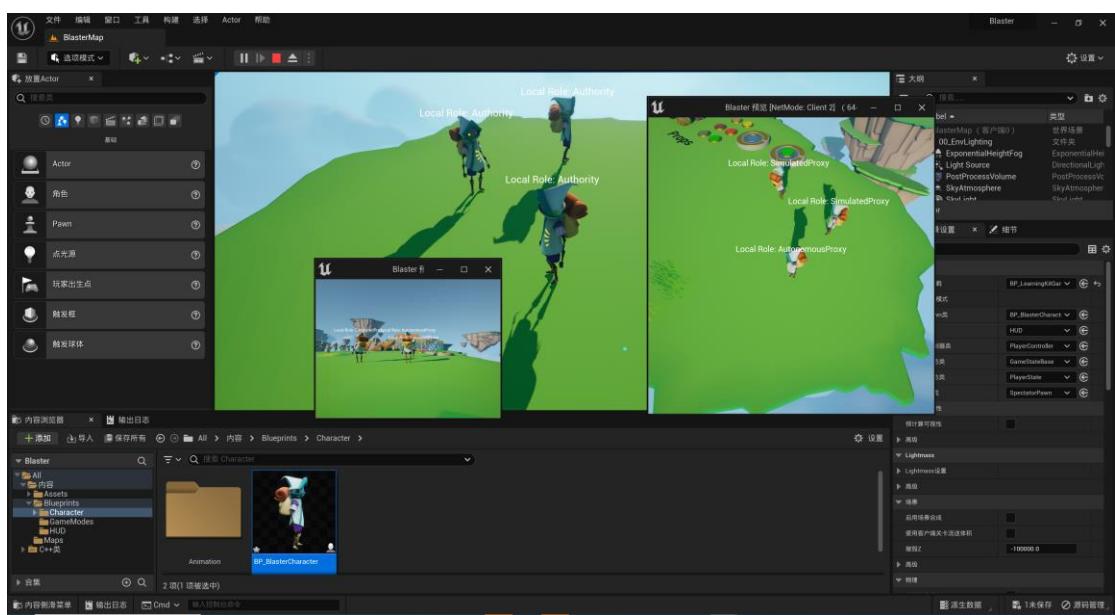
生成蓝图类。

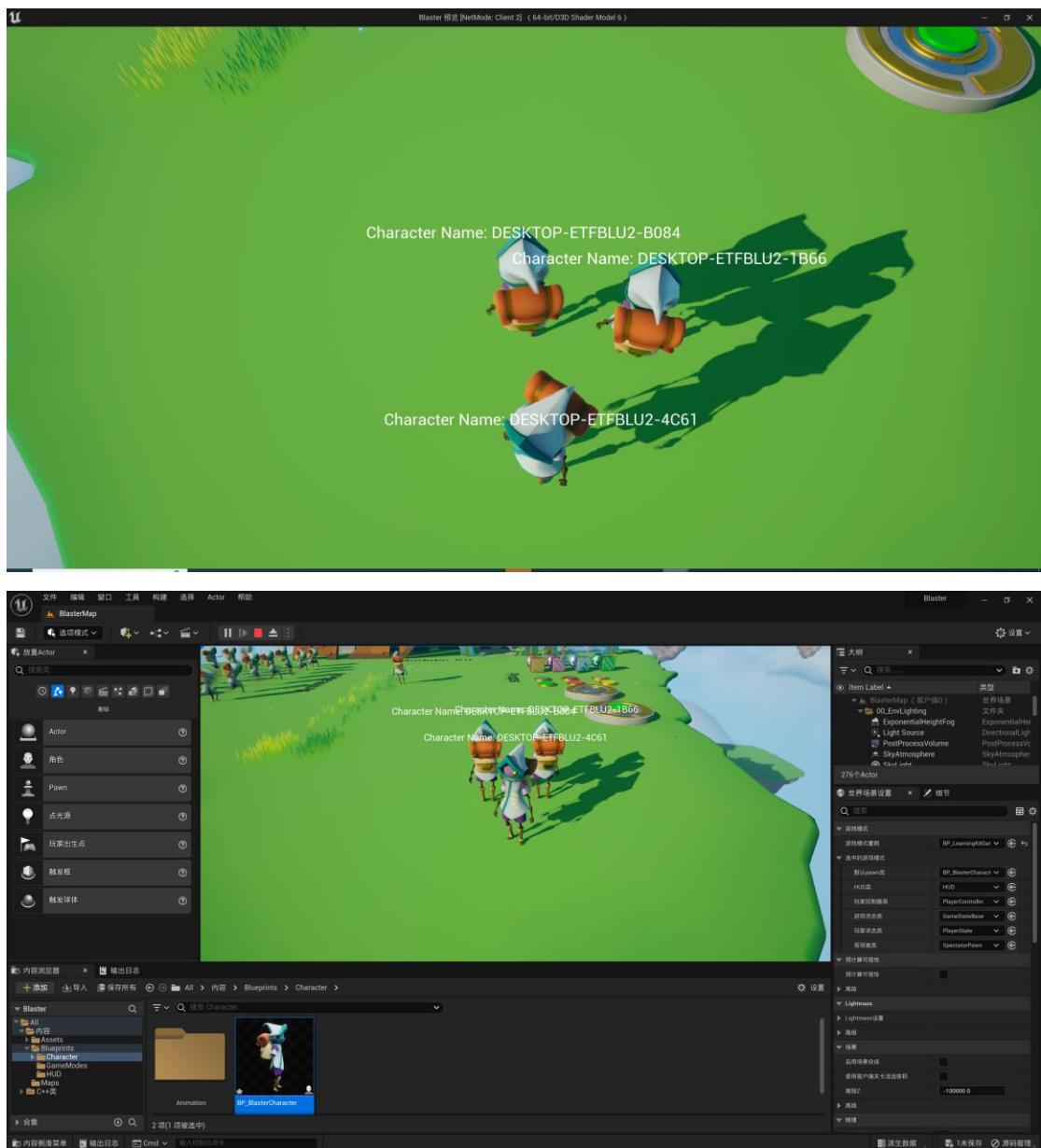
在角色中添加一个 widget 的成员，并在角色蓝图中选择刚才的蓝图类作为 overheadwidget 的控件类型：



连接蓝图：







进阶选项

显示各个角色控制者的名字在头上的 overhead widget 中，经过尝试，当修改函数后：实现的想法很简单，名字无法显示出来是因为在调用函数的时候 PlayerState 还没创建出来，那么我们设置一个定时器，一段时间后再调用这个函数即可。添加如下变量：

```
FTimerHandle UpdateTimer;
void StartUpdateTimer(APawn* InPawn);
void UpdateTimerFinished();
APawn* LocalPawn;
```

实现如下：

```
void UOverheadWidget::UpdatePlayerName(APawn* InPawn)
{
```

```

LocalPawn = InPawn;
APlayerState* PlayerState = InPawn->GetPlayerState();
FString Name;
if (PlayerState)
{
    Name = PlayerState->GetPlayerName();
}
else
{
    StartUpdateTimer(InPawn);
}
FString CharacterNameString = FString::Printf(TEXT("Character Name: %s"), *Name);
SetDisplayText(CharacterNameString);
}

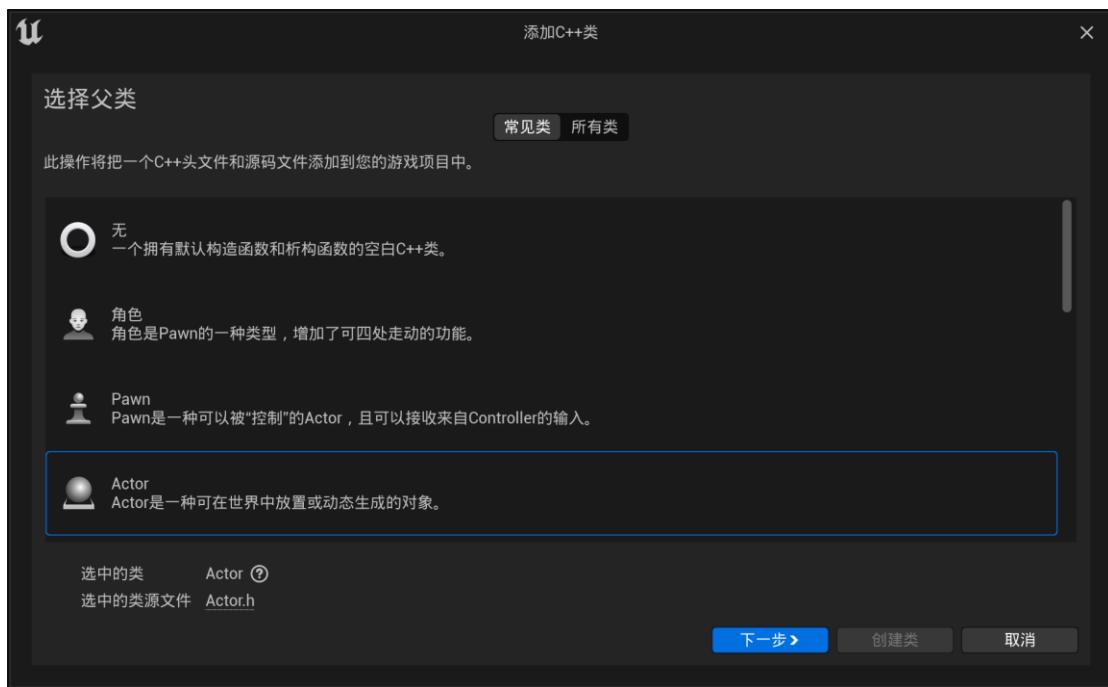
void UOverheadWidget::StartUpdateTimer(APawn* InPawn)
{
    InPawn->GetWorldTimerManager().SetTimer(
        UpdateTimer,
        this,
        &UOverheadWidget::UpdateTimerFinished,
        2. f
    );
}

void UOverheadWidget::UpdateTimerFinished()
{
    UpdatePlayerName(LocalPawn);
}

```

武器

036 武器类



新建一个 Actor 类

因为我们的资产是骨骼网格体，一个拾取范围的碰撞，以及一个状态的枚举，所以我们需要添加一些成员变量：

```
private:  
    UPROPERTY(VisibleAnywhere, Category = "Weapon Properties")  
    USkeletalMeshComponent* WeaponMesh;
```

```
    UPROPERTY(VisibleAnywhere, Category = "Weapon Properties")  
    class USphereComponent* AreaSphere;
```

```
    UPROPERTY(VisibleAnywhere)  
    EWeaponState WeaponState;
```

武器状态的枚举：

```
UENUM(BlueprintType)  
enum class EWeaponState : uint8  
{  
    EWS_Initial UMETA(DisplayName = "Initial State"),  
    EWS_Equipped UMETA(DisplayName = "Equipped"),  
    EWS_Dropped UMETA(DisplayName = "Dropped"),  
  
    EWS_MAX UMETA(DisplayName = "DefaultMAX"),
```

```
};
```

.cpp 中我们做如下设置,

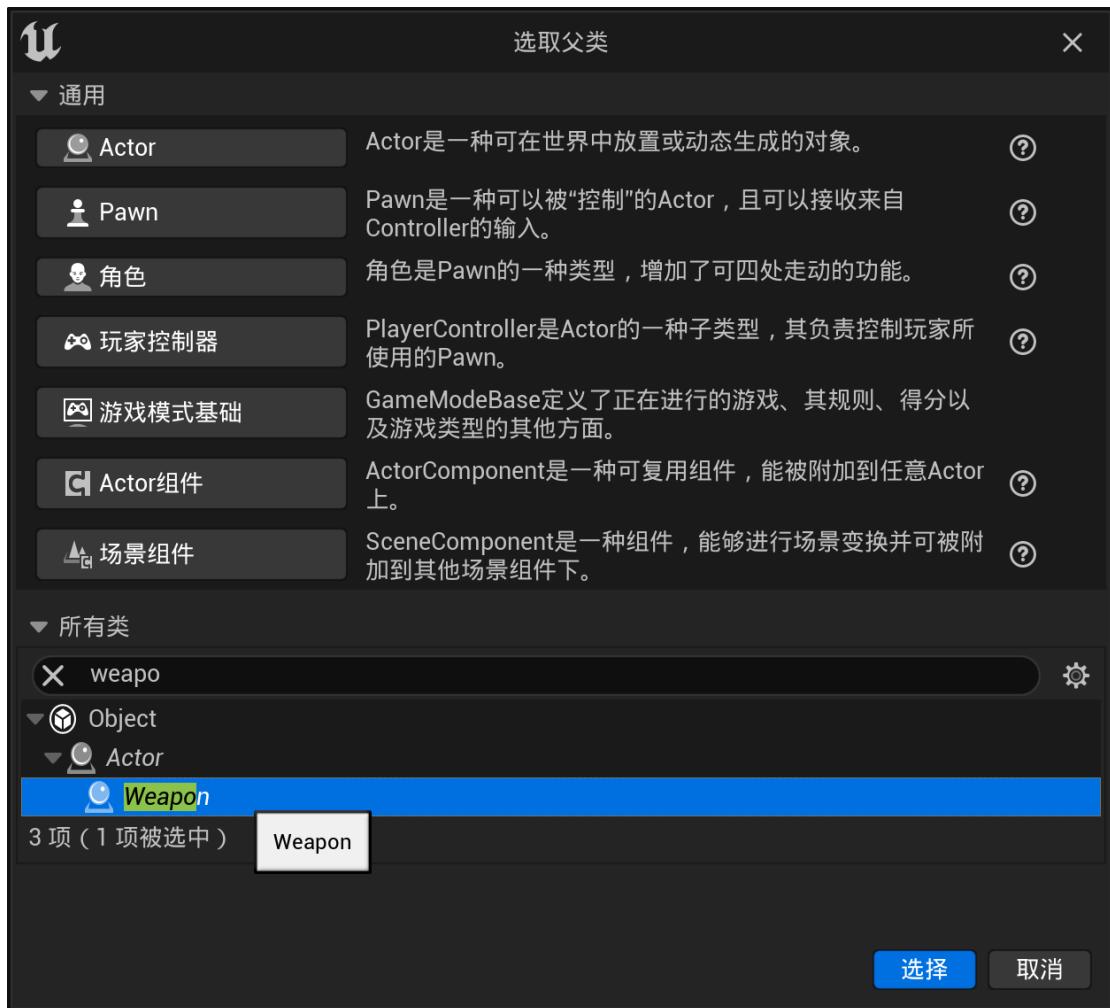
```
// Sets default values
AWeapon::AWeapon()
{
    // Set this actor to call Tick() every frame. You can turn this off to improve
    // performance if you don't need it.
    PrimaryActorTick.bCanEverTick = false; // 关闭tick, 之后有需要可以打开
    bReplicates = true; // 开启复制

    WeaponMesh = CreateDefaultSubobject<USkeletalMeshComponent>(TEXT("WeaponMesh"));
    WeaponMesh->SetupAttachment(RootComponent);
    SetRootComponent(WeaponMesh);

    WeaponMesh->SetCollisionResponseToAllChannels(ECollisionResponse::ECR_Block);
    WeaponMesh->SetCollisionResponseToChannel(ECollisionChannel::ECC_Pawn, ECollisionRes-
    ponse::ECR_Ignore);
    WeaponMesh->SetCollisionEnabled(ECollisionEnabled::NoCollision);

    AreaSphere = CreateDefaultSubobject <USphereComponent>(TEXT("AreaSphere"));
    AreaSphere->SetupAttachment(RootComponent);
    AreaSphere->SetCollisionResponseToAllChannels(ECollisionResponse::ECR_Ignore);
    AreaSphere->SetCollisionEnabled(ECollisionEnabled::NoCollision);
}

void AWeapon::BeginPlay()
{
    Super::BeginPlay();
    // 只在服务器上检测是否和武器有重叠、所以在服务端打开碰撞检测。
    if (HasAuthority()) //HasAuthority() 等于 GetLocalRole() ==
        ENetRole::ROLE_Authority
    {
        AreaSphere->SetCollisionEnabled(ECollisionEnabled::QueryAndPhysics);
        WeaponMesh->SetCollisionResponseToChannel(ECollisionChannel::ECC_Pawn,
            ECollisionResponse::ECR_Overlap);
    }
}
```



编译后打开 UE，创建一个 Weapon 的蓝图类。

选择一个枪的骨骼拖进蓝图，在游戏中防止一个蓝图我们可以看到



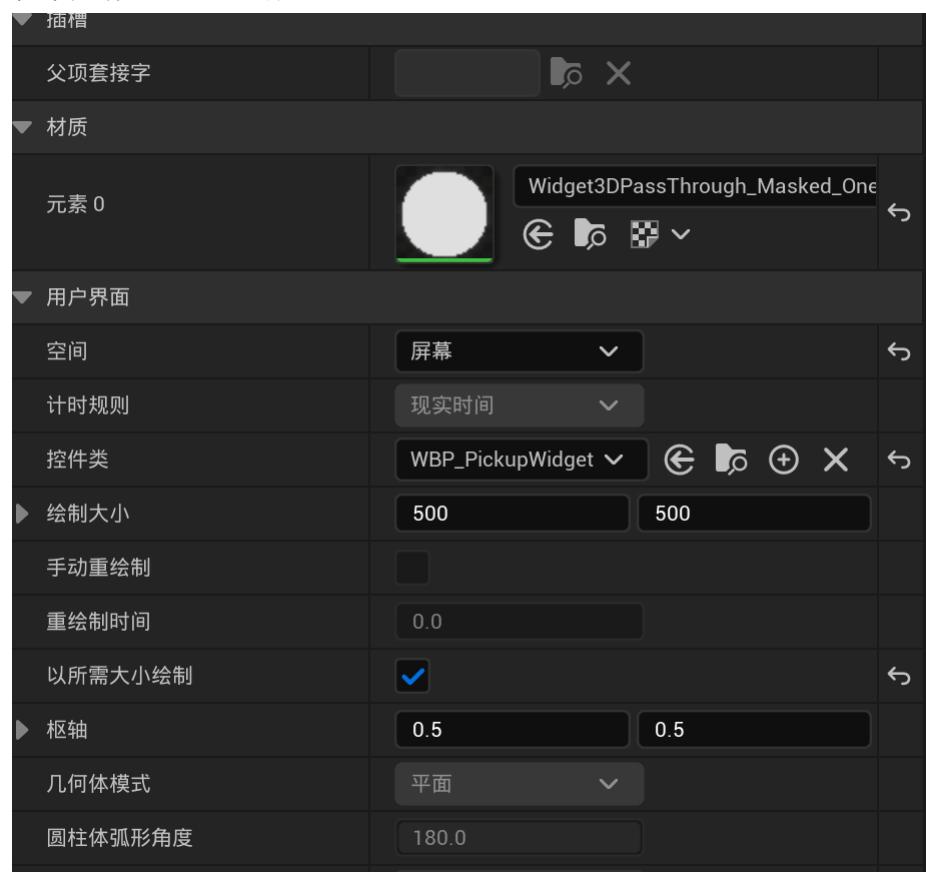
037 拾取控件

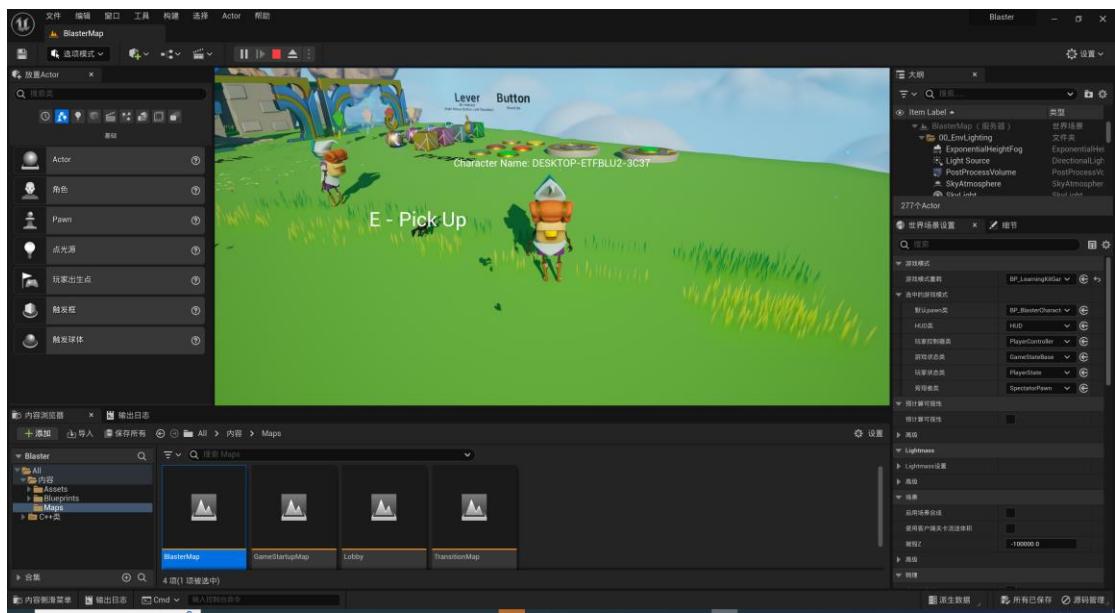
给武器添加一个类似 overhead widget 的控件。

给武器添加碰撞的回调函数：

```
void AWeapon::OnSphereOverlap(UPrimitiveComponent* OverlappedComponent, AActor* OtherActor, UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromSweep, const FHitResult& SweepResult)
{
    ABlasterCharacter* BlasterCharacter = Cast<ABlasterCharacter>(OtherActor);
    if (BlasterCharacter && PickupWidget)
    {
        PickupWidget->SetVisibility(true);
    }
}
```

在服务端绑定回调函数。





但是我们发现并没有成功的将文本复制到客户端上，下一节将学习如何复制其到客户端。

038 变量复制

首先我们在角色类中添加一个变量以及一个函数：

```
UPROPERTY(ReplicatedUsing = OnRep_OverLappingWeapon)
class AWeapon* OverlappingWeapon;
```

```
UFUNCTION()
void OnRep_OverLappingWeapon(AWeapon* LastWeapon);
```

OverlappingWeapon 变量代表的是现在重叠着的武器对象，这个变量我们设置了他会在改变时进行复制，我们限定了他只会复制给拥有它的角色，在.cpp 中实现这个复制的注册：

```
void ABlasterCharacter::GetLifetimeReplicatedProps(TArray<FLifetimeProperty>&
OutLifetimeProps) const
{
    Super::GetLifetimeReplicatedProps(OutLifetimeProps);
```

```
    DOREPLIFETIME_CONDITION(ABlasterCharacter, OverlappingWeapon, COND_OwnerOnly);
}
```

因为这个变量是从服务端复制到的客户端，所以服务端中的那个角色的OverlappingWeapon 变量也会和该客户端相同，这时候会发生服务端也看到了客户端上显示的 widget 的情况。

为了解决这个问题 PEP 复制通知函数：OnRep_OverLappingWeapon 就派上了用场。这个函数被设置在变量 OverlappingWeapon 复制的时候进行调用，因为服务端中变量的修改不是通过复制完成的而是直接的修改，所以服务端中不会调用这个函数，因此这个函数可以帮助我们实现只在客户端完成的操作。

OnRep_OverLappingWeapon 可以设置一个参数，他传递的是发生改变之前的变量的值，这样可以方便于我们关闭之前打开的 widget。除此之外我们需要另外实现一个服务端的显示 widget 的逻辑。

```
void ABlasterCharacter::SetOverlappingWeapon(AWeapon* Weapon)
{
    if (OverlappingWeapon)
    {
        OverlappingWeapon->ShowPickupWidget(false);
    }
    OverlappingWeapon = Weapon;
    if (IsLocallyControlled())
    {
        if (OverlappingWeapon)
        {
            OverlappingWeapon->ShowPickupWidget(true);
        }
    }
}
```

```

void ABlasterCharacter::OnRep_OverlappingWeapon(AWeapon* LastWeapon)
{
    if (OverlappingWeapon)
    {
        OverlappingWeapon->ShowPickupWidget(true);
    }
    if (LastWeapon)
    {
        LastWeapon->ShowPickupWidget(false);
    }
}

```

在角色.cpp 中我们需要实现重叠以及结束重叠的回调函数，以及回调函数的注册。

```

void AWeapon::OnSphereOverlap(UPrimitiveComponent* OverlappedComponent, AActor*
OtherActor, UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromSweep,
const FHitResult& SweepResult)
{
    ABlasterCharacter* BlasterCharacter = Cast<ABlasterCharacter>(OtherActor);
    if (BlasterCharacter)
    {
        BlasterCharacter->SetOverlappingWeapon(this);
    }
}

void AWeapon::OnSphereEndOverlap(UPrimitiveComponent* OverlappedComponent, AActor*
OtherActor, UPrimitiveComponent* OtherComp, int32 OtherBodyIndex)
{
    ABlasterCharacter* BlasterCharacter = Cast<ABlasterCharacter>(OtherActor);
    if (BlasterCharacter)
    {
        BlasterCharacter->SetOverlappingWeapon(nullptr);
    }
}

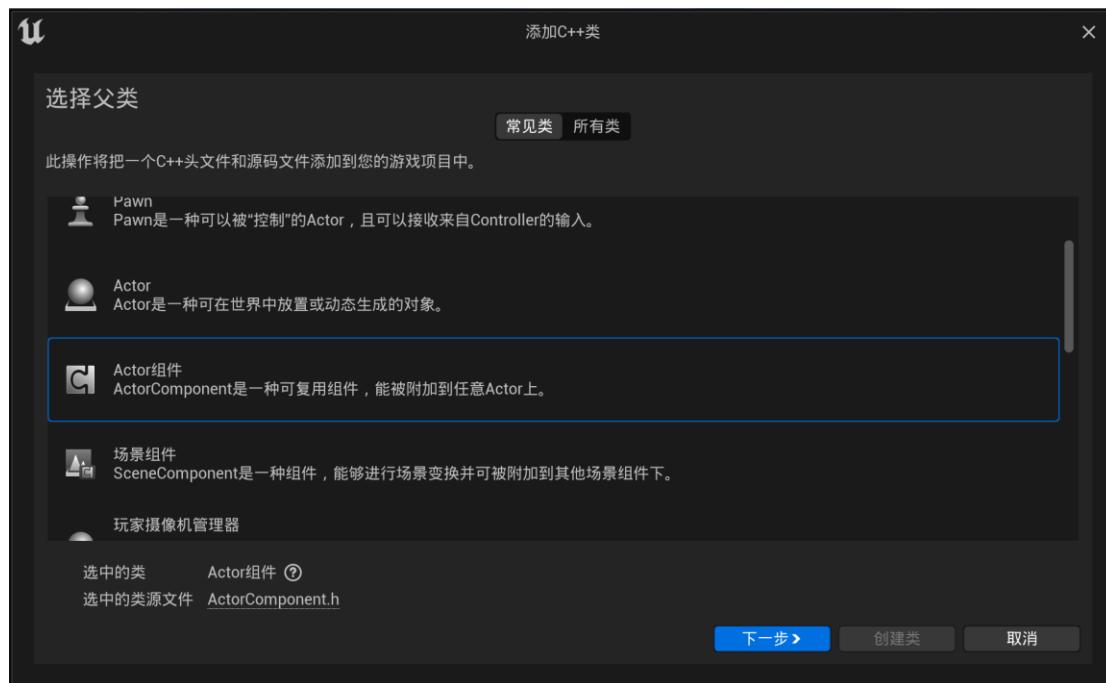
```

需要注意的是我们将客户端的逻辑和服务器的逻辑分开处理的方法。

039 装备武器

由于我们需要实现按下按键后装备武器，现在项目设置中添加一个输入 E 为 Equip。

创建一个 actor 组件类



在.h 文件中添加:

```
private:  
    class ABlasterCharacter* Character;  
    AWeapon* EquippedWeapon;
```

此外，由于在角色类中会调用这些变量，所以添加一个友元

```
public:  
    friend class ABlasterCharacter;
```

.cpp 中:

```
UCombatComponent::UCombatComponent()  
{  
    PrimaryComponentTick.bCanEverTick = false;  
}
```

```
void UCombatComponent::EquipWeapon(AWeapon* WeaponToEquip)  
{  
    if (Character == nullptr || WeaponToEquip == nullptr)  
    {  
        return;  
    }
```

```

EquippedWeapon = WeaponToEquip;
EquippedWeapon->SetWeaponState(EWeaponState::EWS_Equipped);
const USkeletalMeshSocket* HandSocket =
Character->GetMesh()->GetSocketByName(FName("RightHandSocket"));
if (HandSocket)
{
    HandSocket->AttachActor(EquippedWeapon, Character->GetMesh());
}
EquippedWeapon->SetOwner(Character);
EquippedWeapon->ShowPickupWidget(false);
}

```

在角色类中，我们需要绑定 E 键的回调函数，并初始化 Combat 类：

```

Combat = CreateDefaultSubobject<UCombatComponent>(TEXT("CombatComponent"));
Combat->SetIsReplicated(true);

```

因为我们想要尽早的绑定 combat 组件，所以我们使用重载函数：

```

virtual void PostInitializeComponents() override;
void ABlasterCharacter::PostInitializeComponents()
{
    Super::PostInitializeComponents();
    if (Combat)
    {
        Combat->Character = this;
    }
}

```

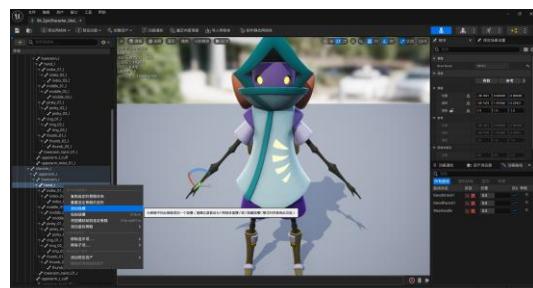
E 键的回调函数中，我们希望调用装备函数的过程只发生在服务器：

```

void ABlasterCharacter::EquipButtonPressed()
{
    if (Combat && HasAuthority())
    {
        Combat->EquipWeapon(OverlappingWeapon);
    }
}

```

为角色的手添加一个插槽并做适当的调整。



由于现在我们只有在服务端才能触发 equip 所以我们服务端可以拿起武器，但是客户端不行。

040 远程过程调用 (RPC)

我们可以通过 RPC 让服务器发出指令并在客户端执行。

RPC 可以分为可靠 RPC 和不可靠 RPC。

- 可靠 RPC 会保证执行。
- 不可靠 RPC 可能会被丢弃。

我们从客户端向服务器发送信息时，信息会被打包，这些包体是在网络间传送的基本单元，这些包有可能被丢弃，可靠 RPC 会返回接受到包体的确认信息，会发生重传。

创建一个可靠 RPC，用于让客户端装备武器：

```
UFUNCTION(Server, Reliable)
```

```
void ServerEquipButtonPressed();
```

在.cpp 中，我们需要在函数的后面加上_Implementation，来告诉 UE 当 RPC 调用的时候，应该做什么。

```
void ABlasterCharacter::ServerEquipButtonPressed_Implementation()
```

```
{  
    if (Combat)  
    {  
        Combat->EquipWeapon(OverlappingWeapon);  
    }  
}
```

原本的按钮回调函数我们做如下修改：

```
void ABlasterCharacter::EquipButtonPressed()  
{  
    if (Combat)  
    {  
        if (HasAuthority())  
        {  
            Combat->EquipWeapon(OverlappingWeapon);  
        }  
        else  
        {  
            ServerEquipButtonPressed();  
        }  
    }  
}
```

当我们是服务器时，我们直接调用 equip 函数，而不是服务器时，我们可以调用 RPC，让服务器帮我们执行 ServerEquipButtonPressed_Implementation()。

编译并运行，我们会发现虽然客户端可以拿起武器了，但是武器上的 widget 却没有消失。

为了解决这个问题，我们要做一个变量复制

```
UPROPERTY(ReplicatedUsing = OnRep_WeaponState, VisibleAnywhere, Category = "Weapon  
Properties")
```

```
EWeaponState WeaponState;
```

```
UFUNCTION()
void OnRep_WeaponState();
```

在武器.cpp 文件中，添加头文件：

```
#include "Net/UnrealNetwork.h"
```

重载变量复制函数：

```
void AWeapon::GetLifetimeReplicatedProps(TArray<FLifetimeProperty>& OutLifetimeProps)
{
    Super::GetLifetimeReplicatedProps(OutLifetimeProps);

    DOREPLIFETIME(AWeapon, WeaponState);
}
```

再将 PEP 复制通知函数完成：

```
void AWeapon::OnRep_WeaponState()
{
    switch (WeaponState)
    {
        case EWeaponState::EWS_Equipped:
            ShowPickupWidget(false);
            break;
    }
}
```

注意到在 Combat 的 cpp 中有这样一个语句：

```
EquippedWeapon->SetOwner(Character);
```

查看 SetOwner 的定义我们可以看到一个 Owner 变量，查看他的申明：

```
UPROPERTY(ReplicatedUsing=OnRep_Owner)
TObjectPtr<AActor> Owner;
```

可以看到 Owner 变量不仅复制了，甚至还有 PEP 复制通知函数。我们可以用这个 OnRep_Owner 函数实现我们想要的功能。查看这个函数的定义，他是一个虚函数，所以我们可以重写他。

```
UFUNCTION()
virtual void OnRep_Owner();
```

Owner 变量既然已经复制了，那么我们设置 Owner 的语句执行之后结果会被复制到客户端，所以这个语句没有问题。

最后是 EquippedWeapon->ShowPickupWidget(false)；语句，由于我们在 OnRep_WeaponState() 中执行了隐藏操作，所以在客户端也不再能看到 widget。

为了关闭枪的重叠检查，我们可以增加一个获取碰撞体的接口，以便从外部修改：

```
FORCEINLINE USphereComponent* GetAreaSphere() const { return AreaSphere; }
```

但是一个更好的方法是直接吧 SetWeaponState 从内联函数变成一个普通成员函数，然后直接调用他。由于是在服务端，在这里关闭重叠即可。

```
void AWeapon::SetWeaponState(EWeaponState State)
{
    WeaponState = State;
    switch (WeaponState)
```

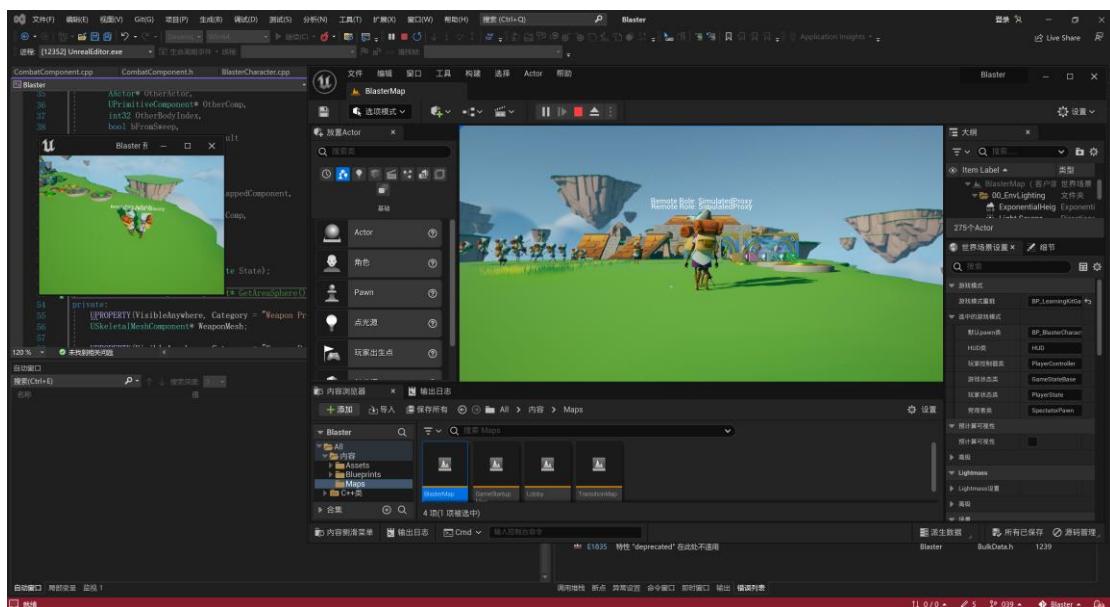
```

    }

    case EWeaponState::EWS_Equipped:
        ShowPickupWidget(false);
        AreaSphere->SetCollisionEnabled(ECollisionEnabled::NoCollision);
        break;
    }
}

```

修改完成后我们可以看到重叠已经不再会 widget 出现了。



041 装备武器的动画姿势

为了实现装备武器的动画姿势，我们需要在动画蓝图中添加一个新的变量：

```
UPROPERTY(BlueprintReadOnly, Category = Movement, meta = (AllowPrivateAccess = "true"))
bool bWeaponEquipped;
```

由于我们动画蓝图中已经有了角色的成员变量，所以我们考虑从角色成员变量中获取 Combat 组件，然后通过 Combat 组件的 EquippedWeapon 值是否为空来返回我们所需的布尔变量。

```
bWeaponEquipped = BlasterCharacter->IsWeaponEquipped();
```

但是还有一个问题，那就是我们只在服务器端调用了装备武器的函数：

```
void ABlasterCharacter::EquipButtonPressed()
{
    if (Combat)
    {
        if (HasAuthority())
        {
            Combat->EquipWeapon(OverlappingWeapon);
        }
        else
        {
            ServerEquipButtonPressed();
        }
    }
}

void ABlasterCharacter::ServerEquipButtonPressed_Implementation()
{
    if (Combat)
    {
        Combat->EquipWeapon(OverlappingWeapon);
    }
}
```

当按下 E 键时，如果是服务器，会直接执行 EquipWeapon，而如果是客户端，则会通过 RPC 调用在服务器执行 EquipWeapon。

在 EquipWeapon 函数中：

```
void UCombatComponent::EquipWeapon(AWeapon* WeaponToEquip)
{
    if (Character == nullptr || WeaponToEquip == nullptr)
    {
        return;
    }
```

```

EquippedWeapon = WeaponToEquip;
EquippedWeapon->SetWeaponState(EWeaponState::EWS_Equipped);
const USkeletalMeshSocket* HandSocket =
Character->GetMesh()->GetSocketByName(FName("RighthandSocket"));
if (HandSocket)
{
    HandSocket->AttachActor(EquippedWeapon, Character->GetMesh());
}
EquippedWeapon->SetOwner(Character);
}

```

我们是通过 SetEquipState，的时候触发了装备状态的变化，然后客户端的装备状态通过复制发生改变，并调用 PEP 复制通知函数实现了 widget 的隐藏。但是整个过程中我们的 EquippedWeapon 变量只有在服务器上是被修改了的，而在客户端上是没有修改的，因此我们需要去吧 Combat 组件中的 EquippedWeapon 变量设置为复制变量：

依旧是这个函数：

```

virtual void TickComponent(float DeltaTime, ELevelTick TickType,
FActorComponentTickFunction* ThisTickFunction) override;
.cpp 中添加头文件和函数体
#include "Net/UnrealNetwork.h"
void UCombatComponent::GetLifetimeReplicatedProps(TArray<FLifetimeProperty>&
OutLifetimeProps) const
{
    Super::GetLifetimeReplicatedProps(OutLifetimeProps);

    DOREPLIFETIME(UCombatComponent, EquippedWeapon);
}

```

这样就能够通过复制了的变量通知客户端的蓝图了。

042 下蹲

在项目中添加一个输入：

```
PlayerInputComponent->BindAction("Crouch", IE_Pressed, this,  
&ABlasterCharacter::CrouchButtonPressed);
```

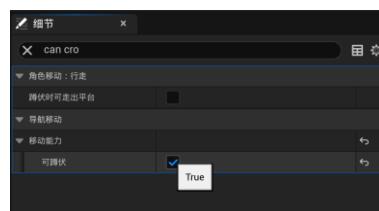
下蹲是一个比较复杂的操作，包括胶囊体大小的改变，移动速度的改变等，好在运动组件为我们提供了这个操作。在回调函数中直接调用运动组件自带的下蹲操作：

```
void ABlasterCharacter::CrouchButtonPressed()  
{  
    if (bIsCrouched)  
    {  
        UnCrouch();  
    }  
    else  
    {  
        Crouch();  
    }  
}
```

Crouch()的定义可以直接查看源码，我们可以发现，他会修改一个运动组件中的变量 IsCrouched，并且这个变量会通过 PEP 复制通知到客户端，所以我们直接在蓝图中查询这个 IsCrouched 的值即可。

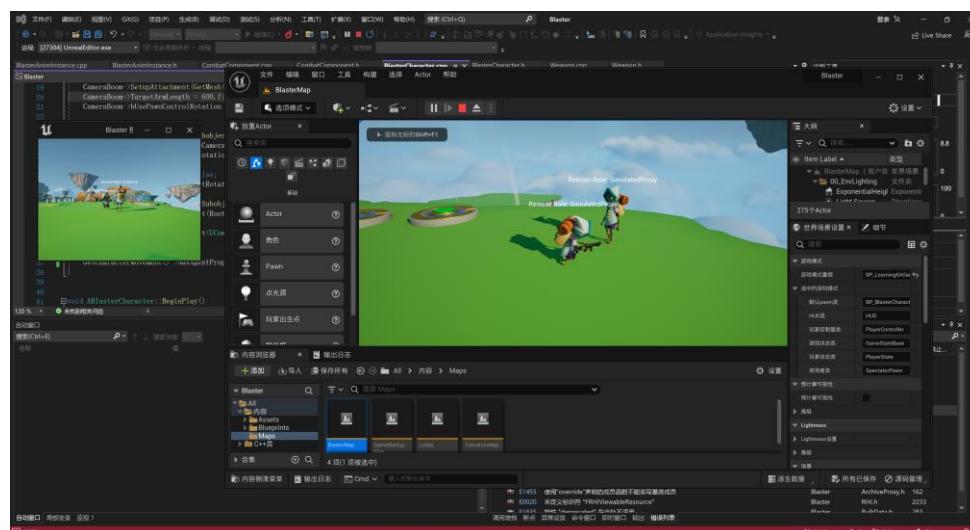
bIsCrouched = BlasterCharacter->bIsCrouched;

需要注意的是我们需要设置可蹲伏，可移动过角色蓝图中的移动组件设置：



或者在构造函数中直接通过代码设置：

```
GetCharacterMovement()->NavAgentProps.bCanCrouch = true;
```



043 瞄准（本地调用弱化延迟感受）

注册两个输入：

```
PlayerInputComponent->BindAction("Aim", IE_Pressed, this,  
&ABlasterCharacter::AimButtonPressed);  
PlayerInputComponent->BindAction("Aim", IE_Released, this,  
&ABlasterCharacter::AimButtonReleased);
```

在 Combat 组件中添加一个 bool 变量 bAiming，并在角色中提供获取这个变量的接口：

```
bool ABlasterCharacter::IsAiming()  
{  
    return (Combat && Combat->bAiming);  
}
```

在动画蓝图中添加一个 bool 变量：

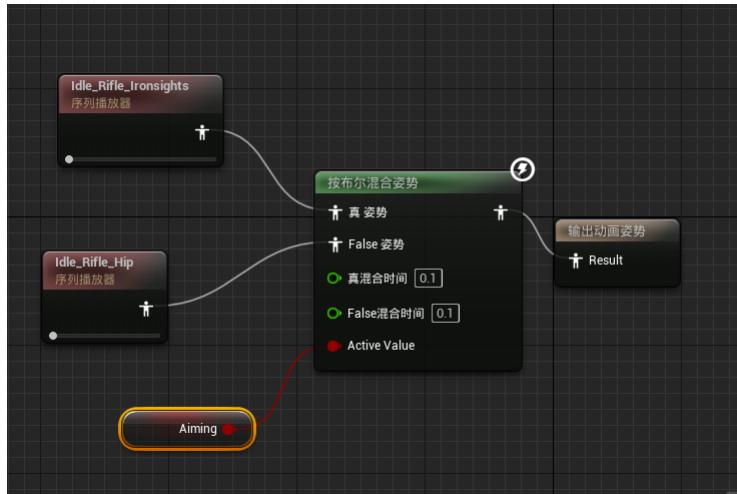
```
UPROPERTY(BlueprintReadOnly, Category = Movement, meta = (AllowPrivateAccess = "true"))  
bool bAiming;
```

并在蓝图更新函数中设置这个变量：

```
bAiming = BlasterCharacter->IsAiming();
```

完成我们的回调函数：

```
void ABlasterCharacter::AimButtonPressed()  
{  
    if (Combat)  
    {  
        Combat->bAiming = true;  
    }  
}  
void ABlasterCharacter::AimButtonReleased()  
{  
    if (Combat)  
    {  
        Combat->bAiming = false;  
    }  
}
```



打开 UE 的动画蓝图，我们在持有武器的 idle 状态中添加一个混合节点。

打开 CrouchIdle 的状态节点，同样添加一个 blend 节点：

然后我们发现只有自己能够看到瞄准的动画，这是因为我们的 bAiming 并没有复制，添加变量复制：

.h 中

UPROPERTY(Replicated)

bool bAiming;

.cpp 中

DOREPLIFETIME(UCombatComponent, bAiming);

之后运行我们会发现当服务器瞄准的时候客户端可以看到瞄准的动画，但是当客户端瞄准的时候，服务器和其他客户端却没有看到动画，这是因为变量复制是单向的，只会从服务器复制到客户端。我们已经学过怎么解决这个问题，那就是使用 RPC。

Combat 中添加两个函数：

`void SetAiming(bool bIsAiming);`

`UFUNCTION(Server, Reliable)`

`void ServerSetAiming(bool bIsAiming);`

和我们之前学过的 PEP 复制通知函数不同，PEP 复制通知函数不能随意传递参数，其输入参数只能是变量复制生效前的原变量，而 RPC 不同，RPC 是可以传递参数的。

`void UCombatComponent::SetAiming(bool bIsAiming)`

{

 bAiming = bIsAiming;

 ServerSetAiming(bIsAiming);

}

`void UCombatComponent::ServerSetAiming_Implementation(bool bIsAiming)`

{

 bAiming = bIsAiming;

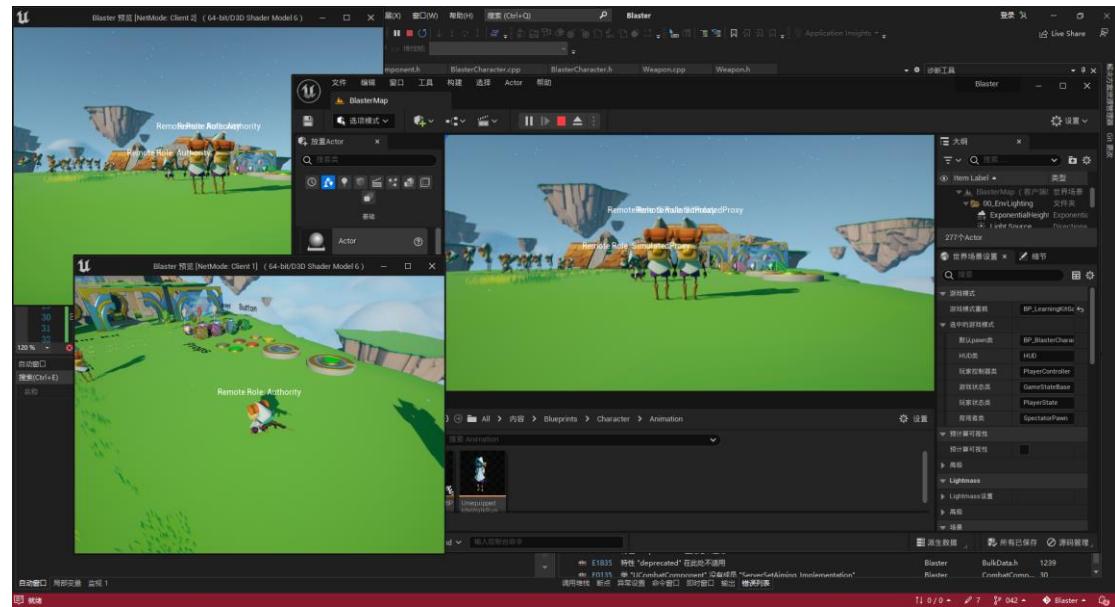
}

我们按照上面的代码实现 RPC 的调用，RPC 的调用规则可以参考：

<https://docs.unrealengine.com/5.1/zh-CN/rpcs-in-unreal-engine/>

当我们从客户端调用 RPC 的时候，会在服务器上运行，这是后变量改变，然后再复制给每个客户端，当我们在服务端调用的时候，同样会在服务器上运行，然后复制给每个客户端。

但是我们仍然保留了在客户端本地设置变量的代码，这是因为从客户端调用 RPC 到服务端执行函数，再到变量复制到客户端是需要延迟的，我们直接在客户端修改变量可以弱化玩家对于延迟的感受。



044 跑步混合空间



创建许多倾斜动画，设置如图的混合空间：

045 倾斜与扫射（利用旋转差值优化动作）

为动画蓝图设置两个新的变量：

```
UPROPERTY(BlueprintReadOnly, Category = Movement, meta = (AllowPrivateAccess = "true"))
float YawOffset;
UPROPERTY(BlueprintReadOnly, Category = Movement, meta = (AllowPrivateAccess = "true"))
float Lean;
```

我们将会在蓝图更新函数中设置这两个值，但是我们要怎么进行设置呢？

```
FRotator AimRotation = BlasterCharacter->GetBaseAimRotation();
UE_LOG(LogTemp, Warning, TEXT("AimRotation Yaw %f: "), AimRotation.Yaw);
```

角色中有一个 GetBaseAimRotation() 函数，我们看看这个函数的作用：

我们发现这是一个向左转减少，向右转增加的值，并且是针对世界坐标的。

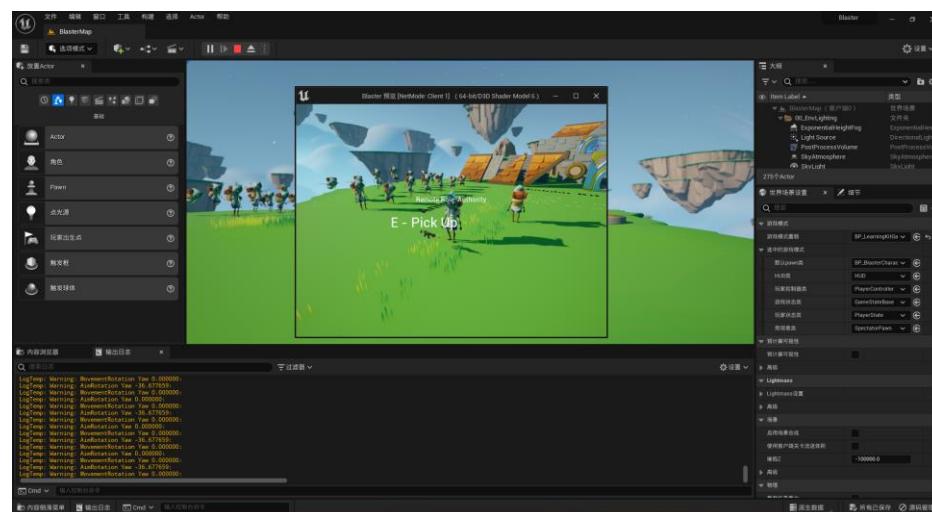
我们再来看一个函数的作用：

```
FRotator MovementRotation =
UKismetMathLibrary::MakeRotFromX(BlasterCharacter->GetVelocity());
UE_LOG(LogTemp, Warning, TEXT("MovementRotation Yaw %f: "), MovementRotation.Yaw);
```

可以发现这个函数的作用是返回我们速度的方向

那么这两个函数在客户端表现如何呢？

```
FRotator AimRotation = BlasterCharacter->GetBaseAimRotation();
FRotator MovementRotation =
UKismetMathLibrary::MakeRotFromX(BlasterCharacter->GetVelocity());
if (!BlasterCharacter->HasAuthority())
{
    UE_LOG(LogTemp, Warning, TEXT("AimRotation Yaw %f: "), AimRotation.Yaw);
    UE_LOG(LogTemp, Warning, TEXT("MovementRotation Yaw %f: "), MovementRotation.Yaw);
}
```

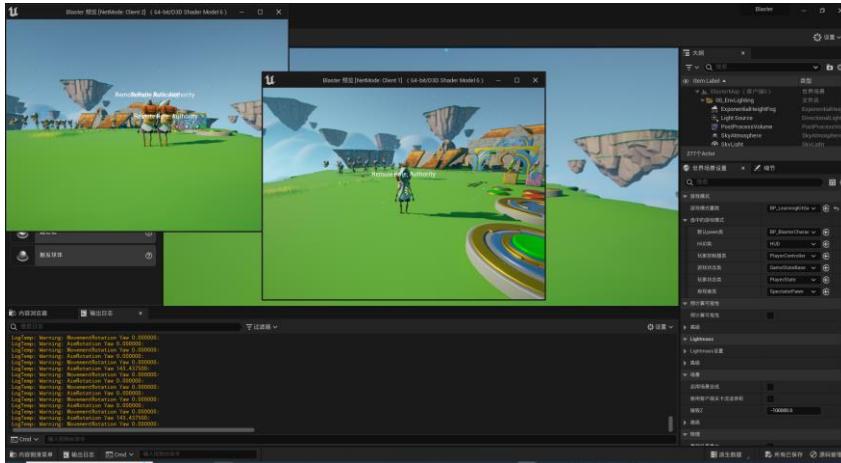


我们可以发现在客户端对自己控制的角色这两个函数是没有问题的。

那么其他客户端能看到这些数据吗：

```
FRotator AimRotation = BlasterCharacter->GetBaseAimRotation();  
FRotator MovementRotation =  
UKismetMathLibrary::MakeRotFromX(BlasterCharacter->GetVelocity());  
if (!BlasterCharacter->HasAuthority() && !BlasterCharacter->IsLocallyControlled())  
{  
    UE_LOG(LogTemp, Warning, TEXT("AimRotation Yaw %f: "), AimRotation.Yaw);  
    UE_LOG(LogTemp, Warning, TEXT("MovementRotation Yaw %f: "), MovementRotation.Yaw);  
}
```

增加一个判断，现在我们只会打印 SimulateProxy 的角色的数据了。



可以看到，数据依然是没有问题的。所以我们可以直接使用这些数据给混合空间进行设置。

更改好状态蓝图后，我们希望拿起武器之后，任务始终面向我们看的方向：

```
void UCombatComponent::EquipWeapon(AWeapon* WeaponToEquip)  
{  
    if (Character == nullptr || WeaponToEquip == nullptr)  
    {  
        return;  
    }  
  
    EquippedWeapon = WeaponToEquip;  
    EquippedWeapon->SetWeaponState(EWeaponState::EWS_Equipped);  
    const USkeletalMeshSocket* HandSocket =  
Character->GetMesh()->GetSocketByName(FName("RightHandSocket"));  
    if (HandSocket)  
    {  
        HandSocket->AttachActor(EquippedWeapon, Character->GetMesh());  
    }  
    EquippedWeapon->SetOwner(Character);  
    Character->GetCharacterMovement()->bOrientRotationToMovement = false;  
    Character->bUseControllerRotationYaw = true;  
}
```

所接下来设置混合空间所用的值

```
YawOffset =  
UKismetMathLibrary::NormalizedDeltaRotator(MovementRotation, AimRotation).Yaw;
```

所以我们在 combat 类中添加两条语句，关闭使运动朝向旋转，开启用控制器旋转 Yaw。
但是这个函数只在服务端调用，我们需要利用 PEP 复制通知实现在客户端的设置。

```
UFUNCTION()  
void OnRep_EquippedWeapon();  
  
UPROPERTY(ReplicatedUsing = OnRep_EquippedWeapon)  
AWeapon* EquippedWeapon;  
  
void UCombatComponent::OnRep_EquippedWeapon()  
{  
    if (EquippedWeapon && Character)  
    {  
        Character->GetCharacterMovement()->bOrientRotationToMovement = false;  
        Character->bUseControllerRotationYaw = true;  
    }  
}
```

Lean的部分按照如下代码实现：

```
CharacterRotationLastFrame = CharacterRotation;  
CharacterRotation = BlasterCharacter->GetActorRotation();  
const FRotator Delta =  
UKismetMathLibrary::NormalizedDeltaRotator(CharacterRotation,  
CharacterRotationLastFrame);  
const float Target = Delta.Yaw / DeltaTime;  
const float Interp = FMath::FInterpTo(Lean, Target, DeltaTime, 6.f);  
Lean = FMath::Clamp(Interp, -90.f, 90.f);
```

由于GetBaseAimRotation() 函数返回的值已经被复制了，所以在里我们不用再担心复制的问题。

但是实际运行之后我们发现，动画的左右切换太过僵硬，为了让左右移动的动画切换变得更加丝滑，我们引入如下代码：

```
FRotator DeltaRot = UKismetMathLibrary::NormalizedDeltaRotator(MovementRotation,  
AimRotation);  
DeltaRotation = FMath::RInterpTo(DeltaRotation, DeltaRot, DeltaTime, 6.f);  
YawOffset = DeltaRotation.Yaw;
```

这是因为在混合空间中直接设置混合时间会导致我们从180度到-180度之间的变化时，由于插值的原因会将中间的动画都播放，发生抖动，而使用UE自带的旋转插值可以避免这个问题。

046 静立与跳跃

调整动画蓝图

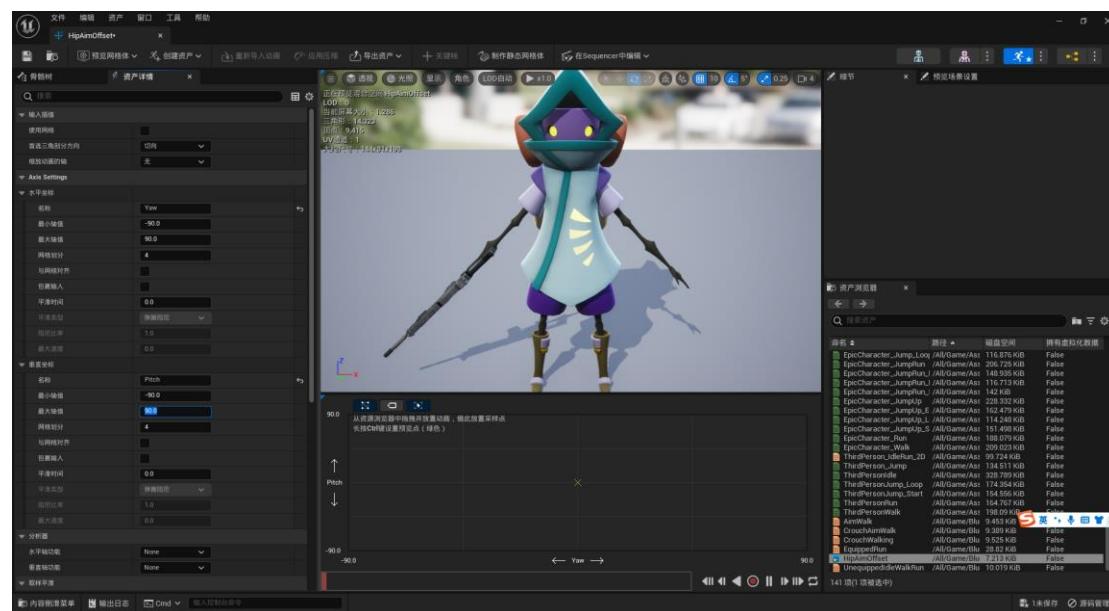
047 蹲走

调整动画蓝图

048 瞄准移动

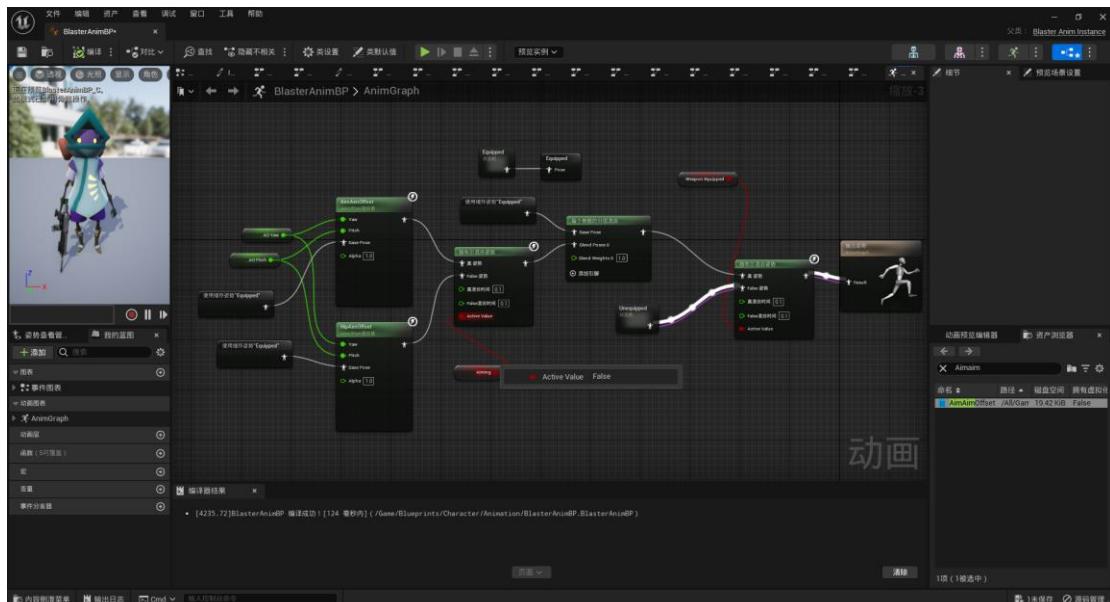
以调整蓝图为主

049 瞄准偏移



创建两个瞄准偏移

050 应用瞄准偏移



我们需要重新连接我们的动画蓝图

在此之前我们要给我们的瞄准空间设置两个变量 AO_Yaw 和 AO_Pitch。

我们可以在角色类中设置，然后再蓝图类中直接获取过来进行更新。

我们希望当我们静止时，小范围移动鼠标不会发生转身，因为这样会导致脚底滑步，但是跳起来的时候或者跑动中能够时刻保持转身。

为了实现这个想法，在 tick 函数中调用以下函数：

```
void ABlasterCharacter::AimOffset(float DeltaTime)
{
    if (!Combat)
    {
        return;
    }
    if (Combat && Combat->EquippedWeapon == nullptr)
    {
        return;
    }

    FVector Velocity = GetVelocity();
    Velocity.Z = 0.f;
    float Speed = Velocity.Size();

    bool bIsInAir = GetCharacterMovement()->IsFalling();

    // standing still, not jumping
    if (Speed == 0.f && !bIsInAir)
    {
```

```

FRotator CurrentAimRotation = FRotator(0. f, GetBaseAimRotation().Yaw, 0. f);
FRotator DeltaAimRotation =
UKismetMathLibrary::NormalizedDeltaRotator(CurrentAimRotation, StartingAimRotation);
AO_Yaw = DeltaAimRotation.Yaw;
bUseControllerRotationYaw = false;
}

// Running or Jumping
if (Speed > 0. f || bIsInAir)
{
    StartingAimRotation = FRotator(0. f, GetBaseAimRotation().Yaw, 0. f);
    AO_Yaw = 0. f;
    bUseControllerRotationYaw = true;
}

AO_Pitch = GetBaseAimRotation().Pitch;
}

```

在动画蓝图类中，我们设置两个变量，并且给他赋值：

```

UPROPERTY(BlueprintReadOnly, Category = Movement, meta = (AllowPrivateAccess = "true"))
float AO_Yaw;
UPROPERTY(BlueprintReadOnly, Category = Movement, meta = (AllowPrivateAccess = "true"))
float AO_Pitch;

AO_Yaw = BlasterCharacter->GetAO_Yaw();
AO_Pitch = BlasterCharacter->GetAO_Pitch();

```

完成以上代码后我们发现只在单人测试的话没有问题，多人测试的时候会出现问题，为什么呢？

Awesome Job, I'll see you soon.

051 多人模式下的 Pitch (UE 压缩包体带来的问题)

在角色中增加一行代码查看到底 pitch 的值发生了什么

```
UE_LOG(LogTemp, Warning, TEXT("AO_Pitch: %f"), AO_Pitch);
```

在单人模式下我们可以看到 Pitch 是一个-89 到 89 之间的值

添加一个条件，用于查看客户端的情况。

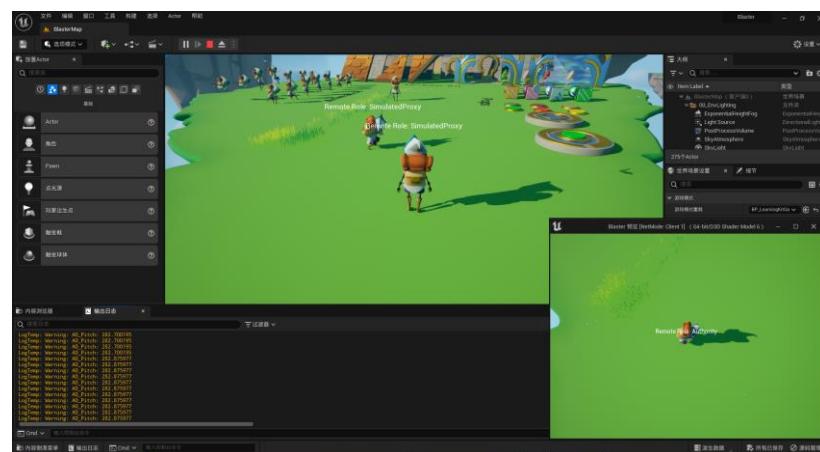
```
if (!HasAuthority() && IsLocallyControlled())
{
    UE_LOG(LogTemp, Warning, TEXT("AO_Pitch: %f"), AO_Pitch);
}
```

我们发现客户端上看自己的 Pitch 也没有问题。

但是我们在服务端上看到的 Pitch 是有问题的，我们改变条件，打印服务端的数据：

```
if (HasAuthority() && !IsLocallyControlled())
{
    UE_LOG(LogTemp, Warning, TEXT("AO_Pitch: %f"), AO_Pitch);
}
```

可以看到朝上看的时候没有问题，但是朝下的时候，Pitch 的值变成了 270-359 之间的值：



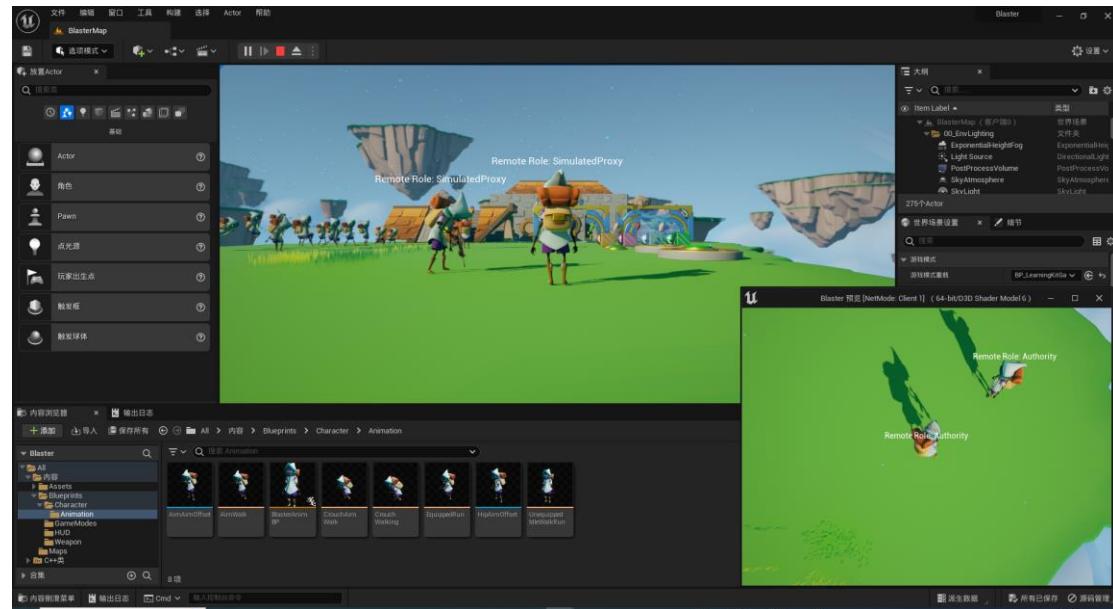
这个原因是因为在进行网络传的时候，UE 为了压缩包体的体积，会把旋转压缩到 5bytes，我们在用旋转的时候可能会有负值，但是 UE 在传递包体的时候会统一压缩成正值。

```
Character...ponent.cpp 旭 x Character...ponent.h 旭 BlasterAnimInstance.cpp 旭 BlasterAnimInstance.h 旭 CombatComponent.cpp 旭 CombatComponent.h 旭
UE5
11925 }
11926
11927     // Lower frequency for standing still and not rotating camera
11928     if (Acceleration.IsZero() && Velocity.IsZero() && ClientData->LastAckedMove.IsValid() && ClientData->LastAckedMove->IsMa
11929     {
11930         NetMoveDelta = FMath::Max(GameNetworkManager->ClientNetSendMoveDeltaTimeStationary, BNetMoveDelta);
11931     }
11932 }
11933
11934     return NetMoveDelta;
11935 }
11936
11937 Ebool FSavedMove_Character::IsMatchingStartControlRotation(const APlayerController* PC) const
11938 {
11939     return PC ? StartControlRotation.Equals(PC->GetControlRotation(), CharacterMovementCVars::NetStationaryRotationTolerance) :
11940 }
11941
11942 void FSavedMove_Character::GetPackedAngles(uint32& YawAndPitchPack, uint8& RollPack) const
11943 {
11944     // Compress rotation down to 5 bytes
11945     YawAndPitchPack = UCharacterMovementComponent::PackYawAndPitchTo32(SavedControlRotation.Yaw, SavedControlRotation.Pitch);
11946     RollPack = FRotator::CompressAxisToByte(Angle, SavedControlRotation.Roll);
11947 }
```

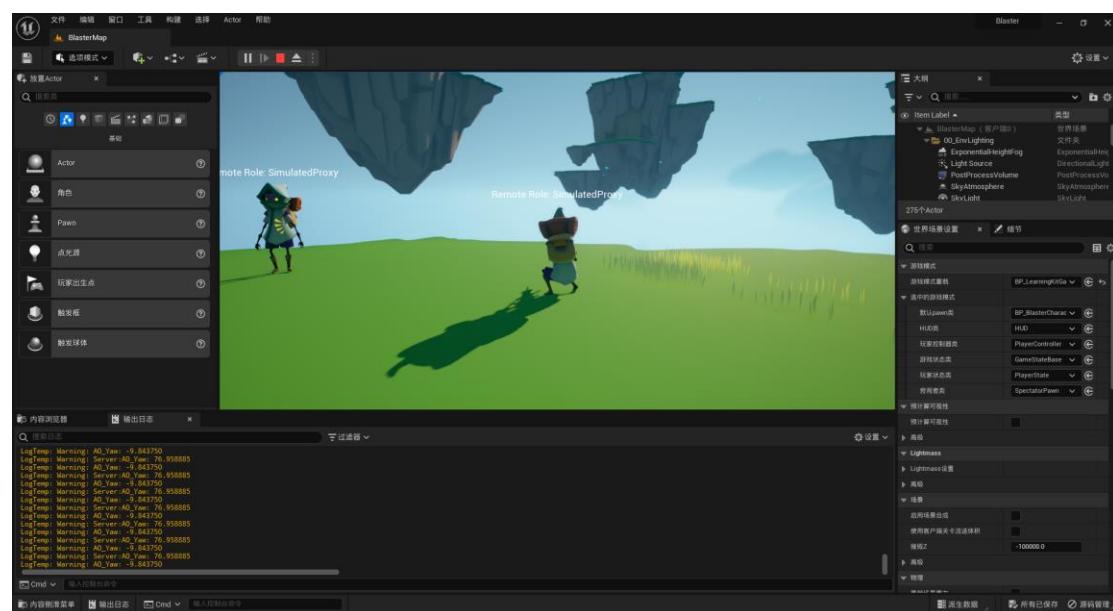
知道了原因我们就能对应的解决这个问题：

```
A0_Pitch = GetBaseAimRotation().Pitch;  
if (A0_Pitch > 90.0f && !IsLocallyControlled())  
{  
    // map pitch from [270, 360) to [-90, 0)  
    FVector2D InRange(270.0f, 360.0f);  
    FVector2D OutRange(-90.0f, 0.0f);  
    A0_Pitch = FMath::GetMappedRangeValueClampedInRange(InRange, OutRange, A0_Pitch);  
}
```

编译代码，可以看到之前的问题已经解决了：



但是还有问题是我们在客户端是看不到服务端的 Yaw 的变化的，



虽然能看到值应该不是复制的问题，但我还是尝试使用复制 AO_Yaw 的办法，发现在客户

端，即使是复制了 AO_Yaw，也会被矫正，出现抖动的情况，验证了我们的想法并不是复制导致的问题，这让我感觉是不是 bUseControllerRotationYaw 的设置出现了问题，尝试添加代码验证：

```
if (!HasAuthority() && !IsLocallyControlled())
{
    if (bUseControllerRotationYaw)
    {
        UE_LOG(LogTemp, Warning, TEXT("True"));
    }
    else
    {
        UE_LOG(LogTemp, Warning, TEXT("False"));
    }
}
```

然而也不是这个问题。

思考了一下，或许是因为这个函数是从控制器获取的玩家的视角的值，但是客户端并没有其他玩家的控制器，因为服务端有所有玩家的控制器应该需要在服务端执行计算然后将结果进行复制。

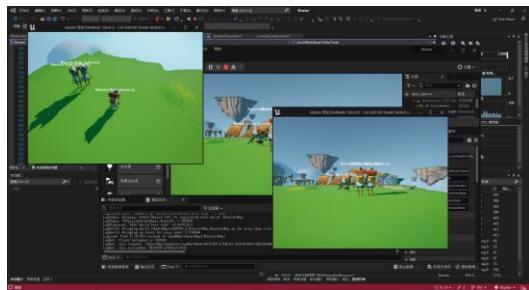
这里尝试如下方案：

```
if (!HasAuthority())
{
    if (IsLocallyControlled())
    {
        // standing still, not jumping
        if (Speed == 0.f && !bIsInAir)
        {
            FRotator CurrentAimRotation = FRotator(0.f, GetBaseAimRotation().Yaw,
0.f);
            FRotator DeltaAimRotation =
UKismetMathLibrary::NormalizedDeltaRotator(CurrentAimRotation, StartingAimRotation);
            AO_Yaw = DeltaAimRotation.Yaw;
            bUseControllerRotationYaw = false;
        }
        // Running or Jumping
        if (Speed > 0.f || bIsInAir)
        {
            StartingAimRotation = FRotator(0.f, GetBaseAimRotation().Yaw, 0.f);
            AO_Yaw = 0.f;
            bUseControllerRotationYaw = true;
        }
    }
    else
    {
        // standing still, not jumping
```

```

        if (Speed == 0. f && !bIsInAir)
        {
            bUseControllerRotationYaw = false;
        }
        // Running or Jumping
        if (Speed > 0. f || bIsInAir)
        {
            bUseControllerRotationYaw = true;
        }
    }
}
else
{
    // standing still, not jumping
    if (Speed == 0. f && !bIsInAir)
    {
        FRotator CurrentAimRotation = FRotator(0. f, GetBaseAimRotation().Yaw,
0. f);
        FRotator DeltaAimRotation =
UKismetMathLibrary::NormalizedDeltaRotator(CurrentAimRotation, StartingAimRotation);
        AO_Yaw = DeltaAimRotation.Yaw;
        bUseControllerRotationYaw = false;
    }
    // Running or Jumping
    if (Speed > 0. f || bIsInAir)
    {
        StartingAimRotation = FRotator(0. f, GetBaseAimRotation().Yaw, 0. f);
        AO_Yaw = 0. f;
        bUseControllerRotationYaw = true;
    }
}

```



成功解决问题。但是为了配合教程，先不自己做调整，之后教程不解决的话再用此方案。

052 使用我们的瞄准偏移

配置蓝图

053 FABRIK IK

FABRIK IK

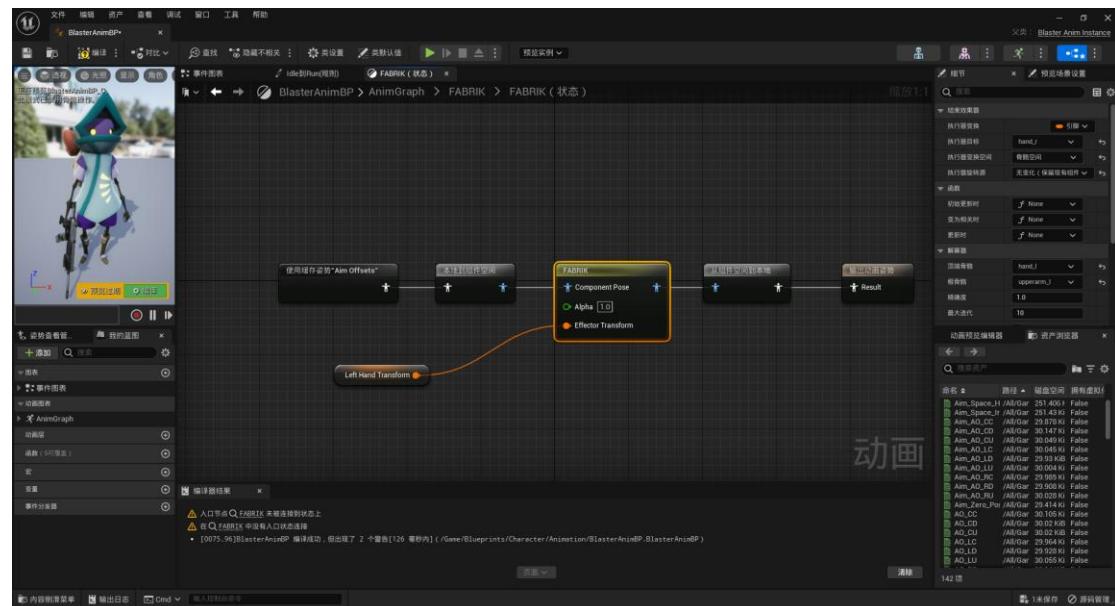
Featured snippet from the web

FABRIK.

- Forward And Backward Reaching Inverse Kinematics (FABRIK) is an iterative approach for solving inverse kinematics (IK) problem.
- IK is a method of determining the joint parameters (the Degrees of Freedoms or DoFs) that provide a desired position for each of the end-effectors.

通过一下函数为我们的蓝图设置一个变量，让他能够知道武器上 LeftHandSocket 的位置：

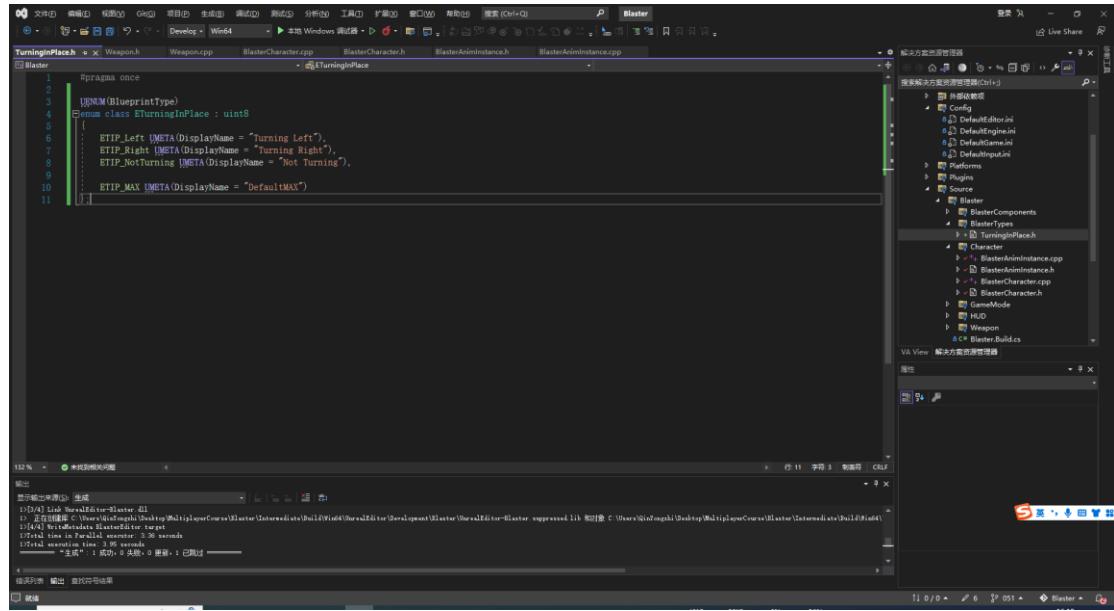
```
if (bWeaponEquipped && EquippedWeapon && EquippedWeapon->GetWeaponMesh() &&
BlasterCharacter->GetMesh())
{
    LeftHandTransform =
    EquippedWeapon->GetWeaponMesh()->GetSocketTransform(FName("LeftHandSocket"), ERelativeTr
ansformSpace::RTS_World);
    FVector OutPosition;
    FRotator OutRotation;
    BlasterCharacter->GetMesh()->TransformToBoneSpace(FName("hand_r"),
    LeftHandTransform.GetLocation(),
    FRotator::ZeroRotator,
    OutPosition,
    OutRotation);
    LeftHandTransform.SetLocation(OutPosition);
    LeftHandTransform.SetRotation(FQuat(OutRotation));
}
```



在动画蓝图中调用 FABRIK 算法

054 原地转身

添加一个枚举类，这次我们用一个新的头文件：



修改角色.cpp 设置枚举变量：

```
if (Speed == 0.0f && !bIsInAir)
{
    FRotator CurrentAimRotation = FRotator(0.0f, GetBaseAimRotation().Yaw, 0.0f);
    FRotator DeltaAimRotation =
        UKismetMathLibrary::NormalizedDeltaRotator(CurrentAimRotation, StartingAimRotation);
    A0_Yaw = DeltaAimRotation.Yaw;
    bUseControllerRotationYaw = false;
    TurnInPlace(DeltaTime);
}

// Running or Jumping
if (Speed > 0.0f || bIsInAir)
{
    StartingAimRotation = FRotator(0.0f, GetBaseAimRotation().Yaw, 0.0f);
    A0_Yaw = 0.0f;
    bUseControllerRotationYaw = true;
    TurningInPlace = ETurningInPlace::ETIP_NotTurning;
}

A0_Pitch = GetBaseAimRotation().Pitch;
if (A0_Pitch > 90.0f && !IsLocallyControlled())
{
    // map pitch from [270, 360) to [-90, 0)
    FVector2D InRange(270.0f, 360.0f);
    FVector2D OutRange(-90.0f, 0.0f);
```

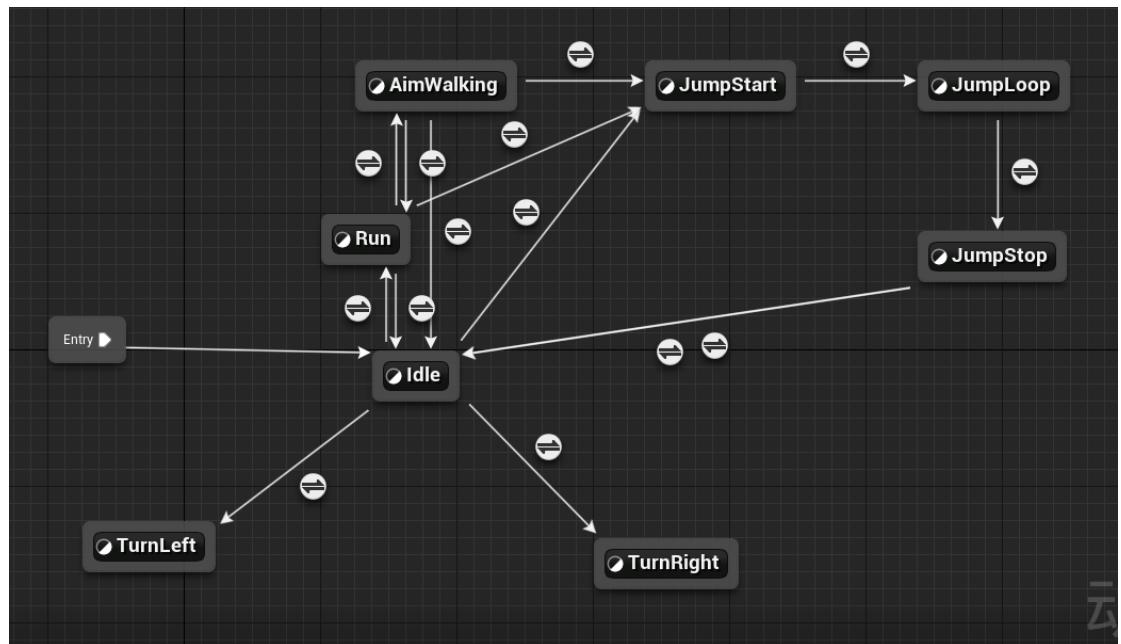
```

AO_Pitch = FMath::GetMappedRangeValueClamped(InRange, OutRange, AO_Pitch);
}

void ABlasterCharacter::TurnInPlace(float DeltaTime)
{
    if (AO_Yaw > 90. f)
    {
        TurningInPlace = ETurningInPlace::ETIP_Right;
    }
    else if (AO_Yaw < -90. f)
    {
        TurningInPlace = ETurningInPlace::ETIP_Left;
    }
}

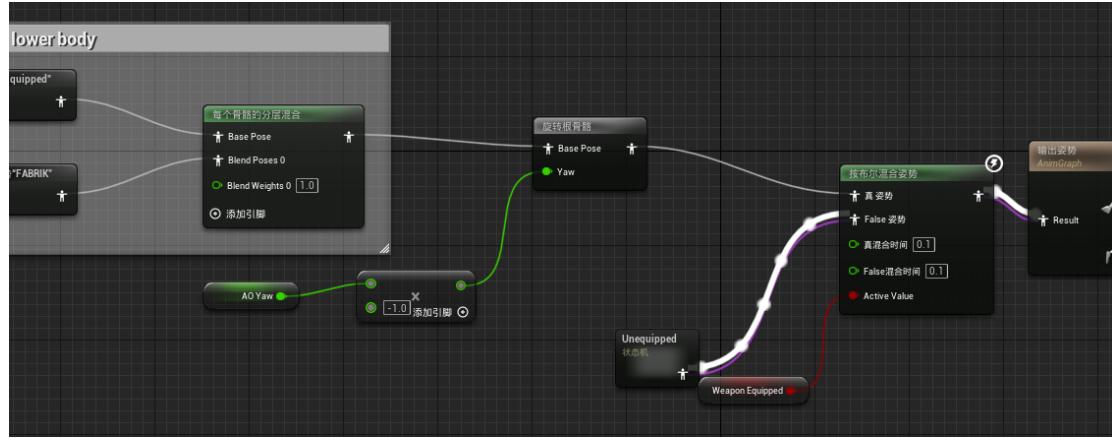
```

从蓝图读取该变量，然后再动画蓝图中进行使用。



055 旋转根骨骼

我们之前设置了在站立不动的时候关闭用控制器 Yaw 控制旋转，现在我们打开他，相应的，为了解决滑步的问题，我们在动画蓝图添加一个节点，抵消掉静止时的旋转。



修改之前的函数：

```
void ABlasterCharacter::AimOffset(float DeltaTime)
{
    if (!Combat)
    {
        return;
    }

    if (Combat && Combat->EquippedWeapon == nullptr)
    {
        return;
    }

    FVector Velocity = GetVelocity();
    Velocity.Z = 0.f;
    float Speed = Velocity.Size();

    bool bIsInAir = GetCharacterMovement()->IsFalling();
    if (Speed == 0.f && !bIsInAir)
    {
        FRotator CurrentAimRotation = FRotator(0.f, GetBaseAimRotation().Yaw, 0.f);
        FRotator DeltaAimRotation =
            UKismetMathLibrary::NormalizedDeltaRotator(CurrentAimRotation, StartingAimRotation);
        AO_Yaw = DeltaAimRotation.Yaw;
        if (TurningInPlace == ETurningInPlace::ETIP_NotTurning)
        {
            InterpAO_Yaw = AO_Yaw;
        }
        bUseControllerRotationYaw = true;
    }
}
```

```

        TurnInPlace(DeltaTime);
    }

    // Running or Jumping
    if (Speed > 0.f || bIsInAir)
    {
        StartingAimRotation = FRotator(0.f, GetBaseAimRotation().Yaw, 0.f);
        A0_Yaw = 0.f;
        bUseControllerRotationYaw = true;
        TurningInPlace = ETurningInPlace::ETIP_NotTurning;
    }

    A0_Pitch = GetBaseAimRotation().Pitch;
    if (A0_Pitch > 90.f && !IsLocallyControlled())
    {
        // map pitch from [270, 360) to [-90, 0)
        FVector2D InRange(270.f, 360.f);
        FVector2D OutRange(-90.f, 0.f);
        A0_Pitch = FMath::GetMappedRangeValueClamped(InRange, OutRange, A0_Pitch);
    }
}

void ABlasterCharacter::TurnInPlace(float DeltaTime)
{
    if (A0_Yaw > 75.f)
    {
        TurningInPlace = ETurningInPlace::ETIP_Right;
    }
    else if (A0_Yaw < -75.f)
    {
        TurningInPlace = ETurningInPlace::ETIP_Left;
    }
    if (TurningInPlace != ETurningInPlace::ETIP_NotTurning)
    {
        InterpAO_Yaw = FMath::FInterpTo(InterpAO_Yaw, 0, DeltaTime, 4.f);
        A0_Yaw = InterpAO_Yaw;
        if (FMath::Abs(A0_Yaw) < 10.f)
        {
            TurningInPlace = ETurningInPlace::ETIP_NotTurning;
            StartingAimRotation = FRotator(0.f, GetBaseAimRotation().Yaw, 0.f);
        }
    }
}

```

当旋转超过一定范围的时候，进行转身，并播放动画。

056 网络更新频率



在角色类中也可以设置，通常射击游戏使用 66 和 33：

```
NetUpdateFrequency = 66. f;
```

```
MinNetUpdateFrequency = 33. f;
```

在 DefaultEngine.ini 中我们可以设置服务器最大帧率：

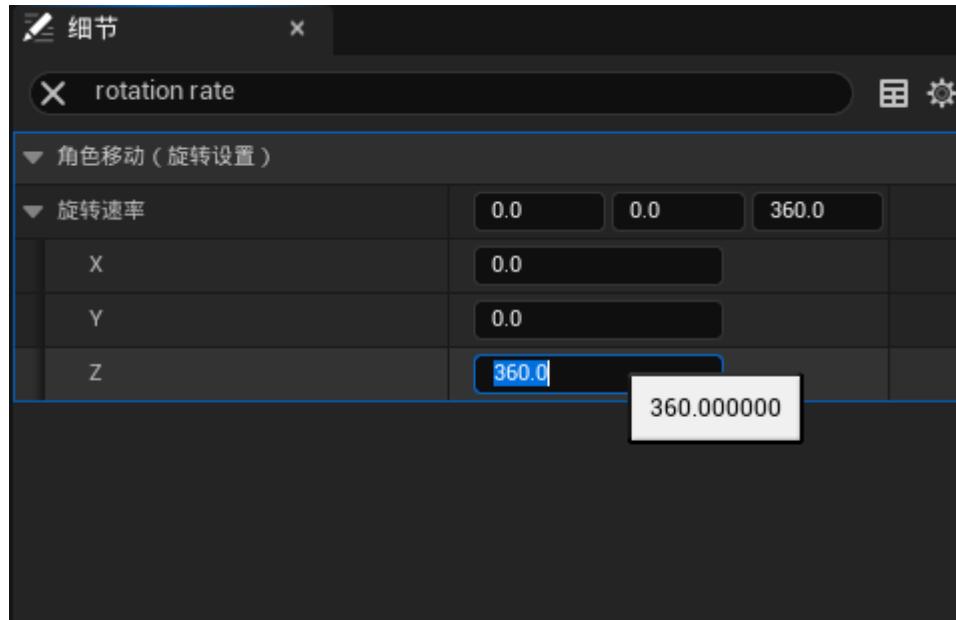
```
[/Script/OnlineSubsystemUtils. IpNetDriver]
```

```
NetServerMaxTickRate = 60
```

057 无装备下的蹲动作

在动画蓝图中，未装备的状态中使用自带的 CrouchWalk 的混合空间。

此外我们发现转身速率有些慢：



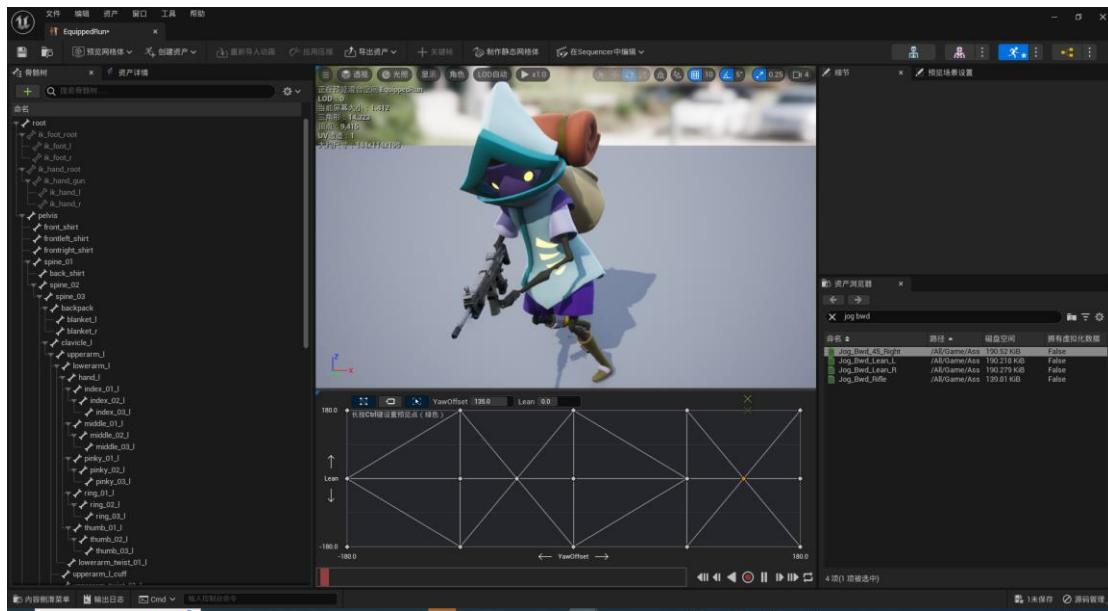
在旋转速率中有控制我们角色 Yaw 方向旋转的值，也可以用 C++ 设置：

```
GetCharacterMovement() -> RotationRate = FRotator(0.f, 0.f, 850.f);
```

最后，我们在蹲下时，space 会无效，我们可以重写 Jump 函数，并更改输入绑定，让我们在蹲下状态按空格键时可以站起来。

```
void ABlasterCharacter::Jump()
{
    if (bIsCrouched)
    {
        UnCrouch();
    }
    else
    {
        Super::Jump();
    }
}
```

058 旋转跑步动画



我们发现之前向左前方移动和向右后方移动的动画很奇怪, 我们直接用向前和向后的动画根节点旋转 45 度, spine 再旋转 45 度回来实现。

059 脚步和跳跃声

添加同步信号用于动画之前的同步, 利用通知添加脚步声, 起跳声和落地声。

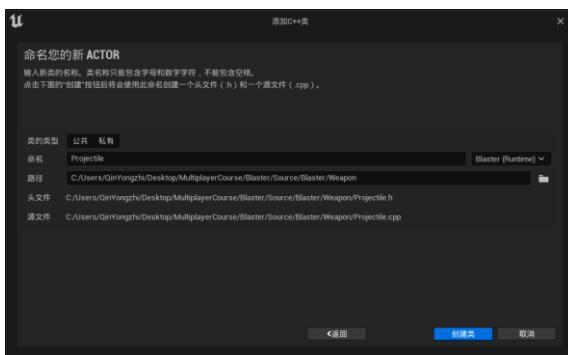
武器开火

060 发射武器类 (projectile weapon class)

创建一个 Weapon 类的子类:



发射类武器需要有一个发射物:



在构造函数中设置 Projectile 的碰撞通道:

```
AProjectile::AProjectile()
{
    PrimaryActorTick.bCanEverTick = true;

    CollisionBox = CreateDefaultSubobject<UBoxComponent>(TEXT("CollisionBox"));
    SetRootComponent(CollisionBox);
    CollisionBox->SetCollisionObjectType(ECollisionChannel::ECC_WorldDynamic);
    CollisionBox->SetCollisionEnabled(ECollisionEnabled::QueryAndPhysics);
    CollisionBox->SetCollisionResponseToAllChannels(ECollisionResponse::ECR_Ignore);
    CollisionBox->SetCollisionResponseToChannel(ECollisionChannel::ECC_Visibility,
        ECollisionResponse::ECR_Block);
    CollisionBox->SetCollisionResponseToChannel(ECollisionChannel::ECC_WorldStatic,
        ECollisionResponse::ECR_Block);
}
```

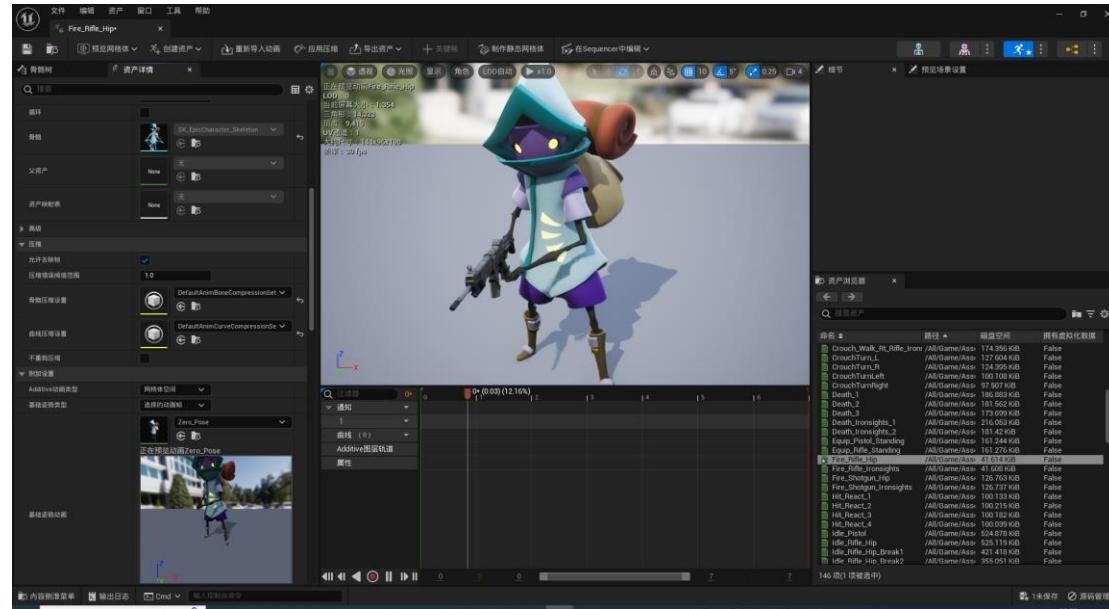
061 开火蒙太奇

添加一个输入，开火：

```
void ABlasterCharacter::FireButtonPressed()
{
    if (Combat)
    {
        Combat->FireButtonPressed(true);
    }
}

void ABlasterCharacter::FireButtonReleased()
{
    if (Combat)
    {
        Combat->FireButtonPressed(false);
    }
}
```

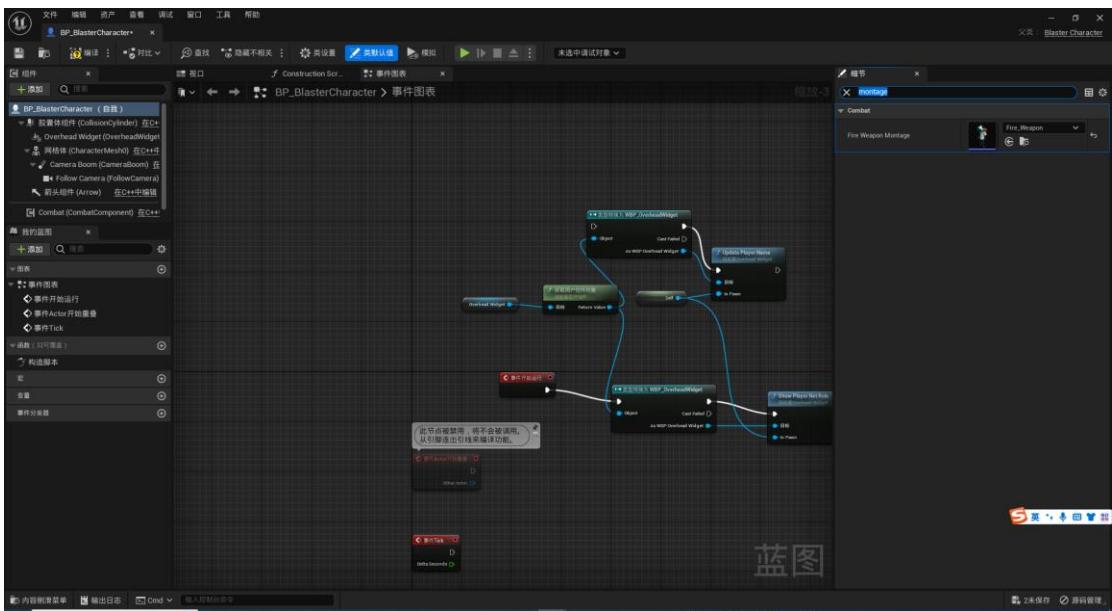
将开火动画设置为 Additive 动画，然后制作蒙太奇：



给角色类新添加一个蒙太奇成员：

```
UPROPERTY(EditAnywhere, Category = Combat)
class UAnimMontage* FireWeaponMontage;
```

设置蒙太奇成员：



设置一个用于播放蒙太奇动画的成员函数：

```
void ABlasterCharacter::PlayFireMontage(bool bAiming)
{
    if (nullptr == Combat || nullptr == Combat->EquippedWeapon)
    {
        return;
    }

    UAnimInstance* AnimInstance = GetMesh()->GetAnimInstance();
    if (AnimInstance && FireWeaponMontage)
    {
        AnimInstance->Montage_Play(FireWeaponMontage);
        FName SectionName;
        SectionName = bAiming ? FName("RifleAim") : FName("RifleHip");
        AnimInstance->Montage_JumpToSection(SectionName);
    }
}
```

在 Combat 中调用播放开火动画的成员函数：

```
void UCombatComponent::FireButtonPressed(bool bPressed)
{
    bFrieButtonPressed = bPressed;
    if (Character && bFrieButtonPressed)
    {
        Character->PlayFireMontage(bAiming);
    }
}
```

062 开火效果

为我们的 Weapon 类添加一个动画成员：

```
UPROPERTY(EditAnywhere, Category = "Weapon Properties")
```

```
class UAnimationAsset* FireAnimation;
```

设置开火函数：

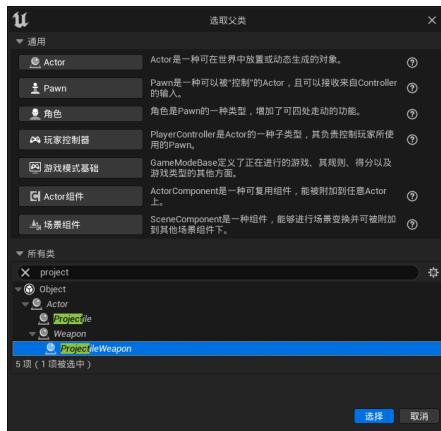
```
void AWeapon::Fire()
{
    if (FireAnimation)
    {
        WeaponMesh->PlayAnimation(FireAnimation, false);
    }
}
```

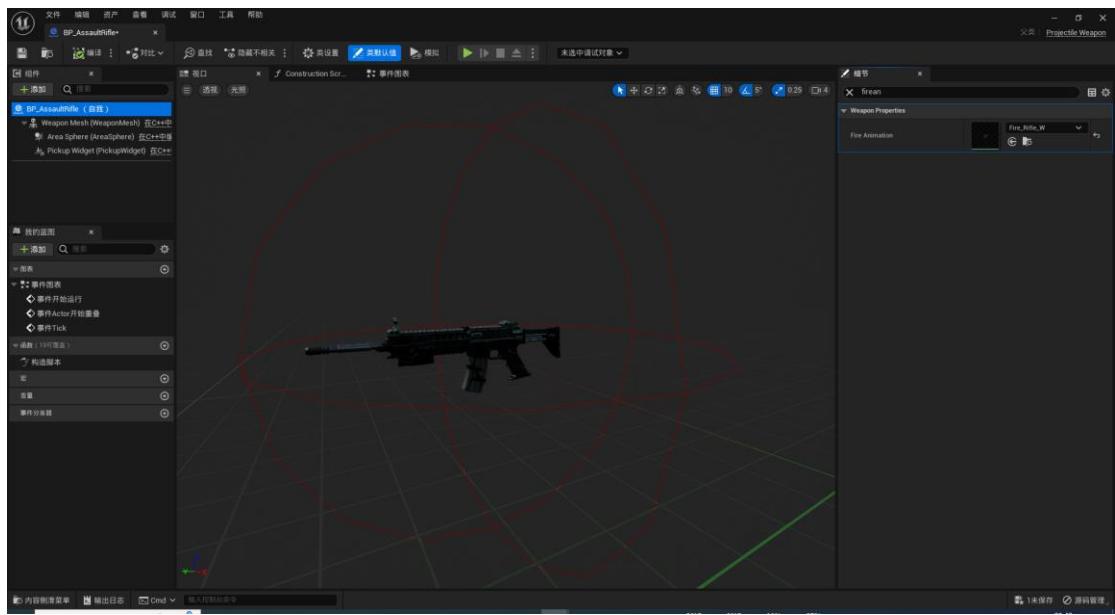
修改之前的开火按钮按下的函数：

```
void UCombatComponent::FireButtonPressed(bool bPressed)
{
    bFrieButtonPressed = bPressed;
    if (nullptr == EquippedWeapon)
    {
        return;
    }

    if (Character && bFrieButtonPressed)
    {
        Character->PlayFireMontage(bAiming);
        EquippedWeapon->Fire();
    }
}
```

创建一个 ProjectileWeapon 类：





设置开火动画，现在我们可以看到开火的效果了。

但是需要注意的是，开火函数并没有复制，是在本地进行的，所以我们需要做一些修改，这些修改将在下一节中进行。

063 多人模式下的开火

我们会用 RPC 来实现多人模式下的开火，我们之前说过的服务器 RPC 会让 RPC 在服务器执行。

在 Combat.h 中创建一个 ServerRPC

```
UFUNCTION(Server, Reliable)
void ServerFire();
```

那么我们再在服务端进行给一个开火的多播，就可以让所有客户端看到这个动画：

```
UFUNCTION(NetMulticast, Reliable)
void MulticastFire();
```

利用三个函数实现开火的多播：

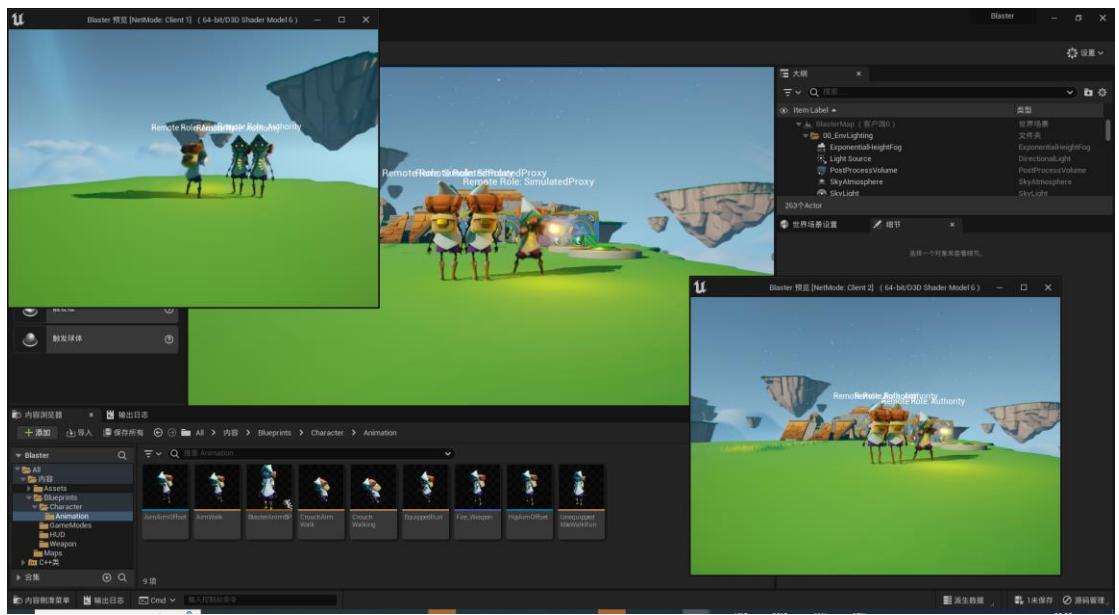
```
void UCombatComponent::FireButtonPressed(bool bPressed)
{
    bFrieButtonPressed = bPressed;

    if (bFrieButtonPressed)
    {
        ServerFire();
    }
}

void UCombatComponent::ServerFire_Implementation()
{
    MulticastFire();
}

void UCombatComponent::MulticastFire_Implementation()
{
    if (nullptr == EquippedWeapon)
    {
        return;
    }

    if (Character)
    {
        Character->PlayFireMontage(bAiming);
        EquippedWeapon->Fire();
    }
}
```



064 准星上的目标

我们创建一个函数在 Combat 中，用于返回我们视角下准星锁看到的目标，为此我们需要用到射线检测，并且需要使用函数将摄像机投影回世界坐标，并获取其起点和方向用于射线检测。

之后我们会在 tick 函数中调用这个函数。

```
void UCombatComponent::TraceUnderCrosshairs(FHitResult& TraceHitResult)
{
    FVector2D ViewportSize;
    if (GEngine && GEngine->GameViewport)
    {
        GEngine->GameViewport->GetViewportSize(ViewportSize);
    }

    FVector2D CrosshairLocation(ViewportSize.X / 2.f, ViewportSize.Y / 2.f);
    FVector CrosshairWorldPosition;
    FVector CrosshairWorldDirection;
    bool bScreenToWorld = UGameplayStatics::DeprojectScreenToWorld(
        UGameplayStatics::GetPlayerController(this, 0),
        CrosshairLocation,
        CrosshairWorldPosition,
        CrosshairWorldDirection
    );

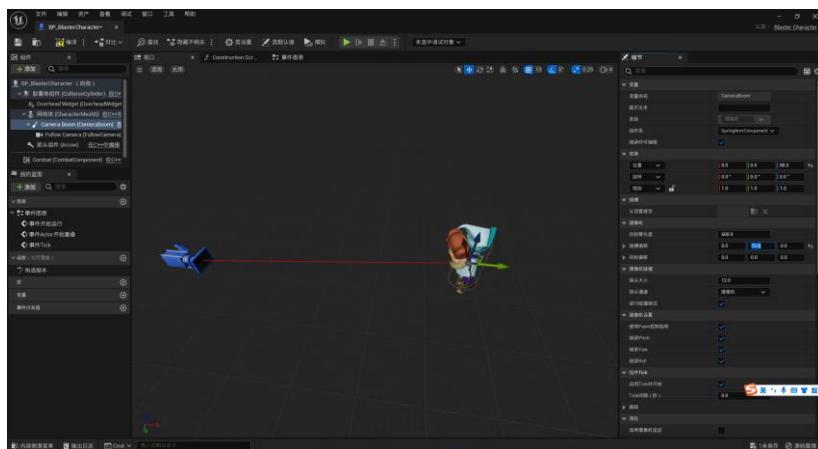
    if (bScreenToWorld)
    {
        FVector Start = CrosshairWorldPosition;
        FVector End = Start + CrosshairWorldDirection * TRACE_LENGTH;

        GetWorld()->LineTraceSingleByChannel(
            TraceHitResult,
            Start,
            End,
            ECollisionChannel::ECC_Visibility
        );
        if (!TraceHitResult.bBlockingHit)
        {
            TraceHitResult.ImpactPoint = End;
        }
        else
        {
            DrawDebugSphere(
                GetWorld(),
                TraceHitResult.ImpactPoint,
```

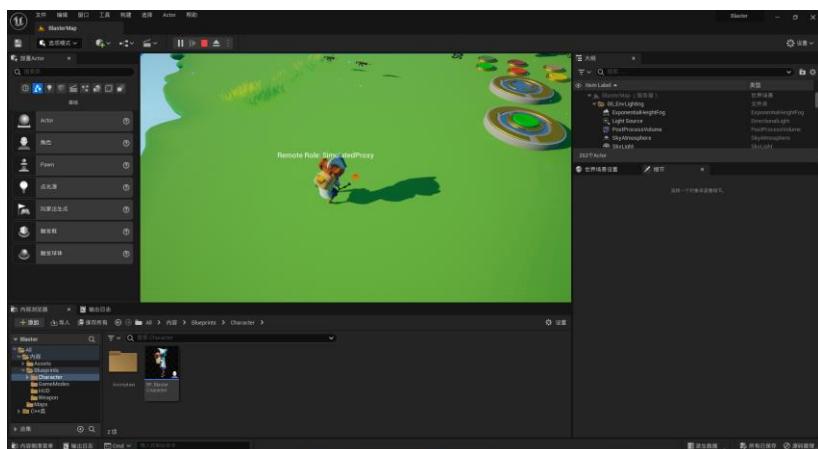
```

12. f,
12,
FColor::Red
);
}
}
}

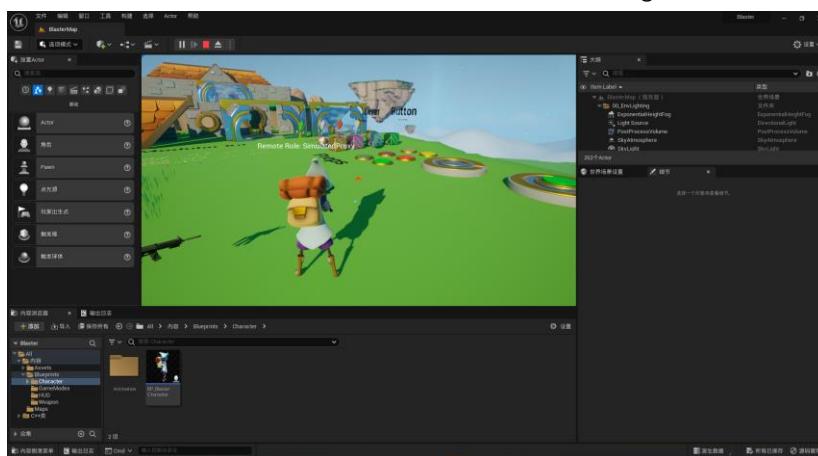
```



由于摄像机设置在了角色正后方，我们将摄像机进行一个小偏移。



可以看到屏幕中间出现了射线检测命中目标的 Debug 结果。

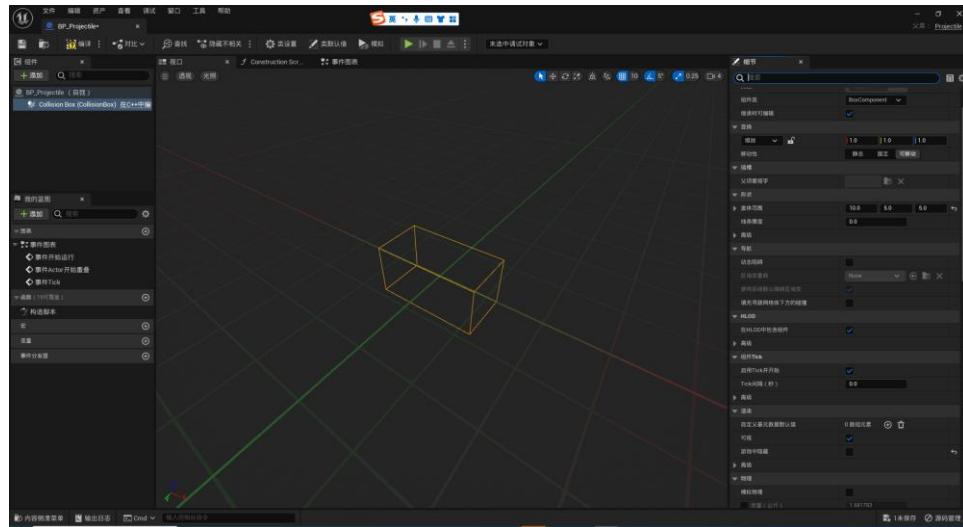


065 生成投射物

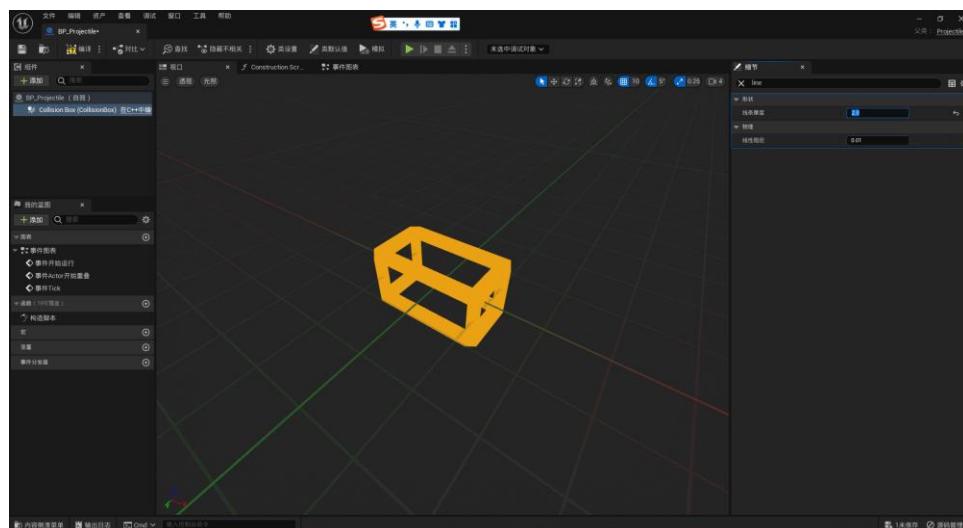
现在投射武器类中添加一个成员：

```
private:  
    UPROPERTY(EditAnywhere)  
    TSubclassOf<class AProjectile> ProjectileClass;
```

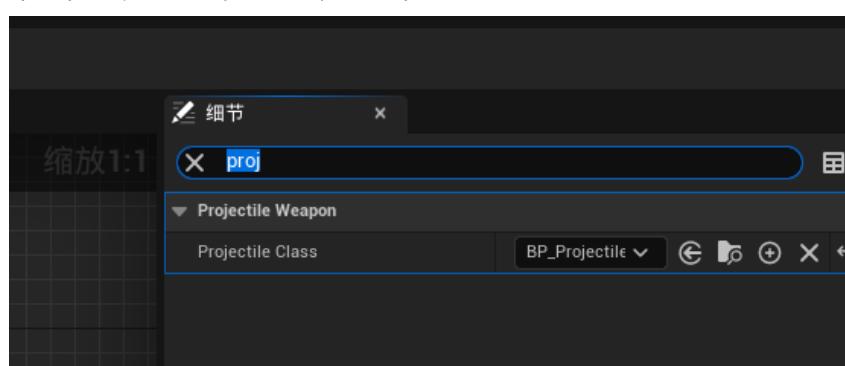
用于设置我们投掷类的类型。



创建一个 projectile 的子类，设置大小和可见性，以及线条厚度。



来到投掷类武器的蓝图中，设置投掷物：



将 Weapon.h 中的 Fire 函数设为虚函数，并添加一个输入：

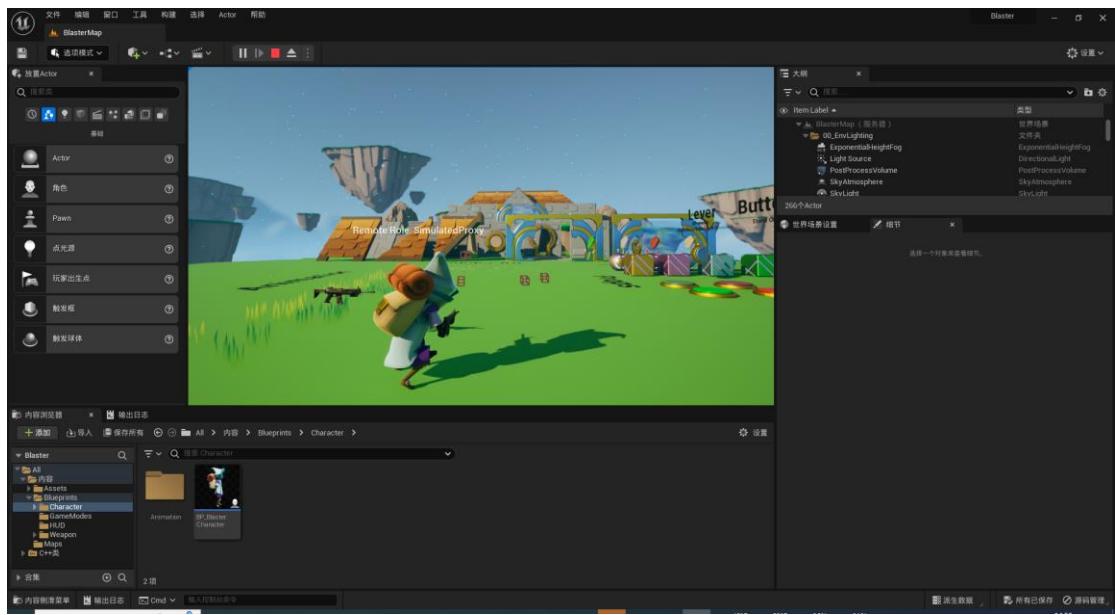
```
virtual void Fire(const FVector& HitTarget);
```

在 WeaponProjectile 中重写这个函数：

在枪口的 socket 位置生成子弹，并且通过枪口位置以及屏幕中心射线检测返回的结果设置生成的 Actor 的方向，用来解决第三人称下，准星和位置不是实际瞄准位置的问题。

```
void AProjectileWeapon::Fire(const FVector& HitTarget)
{
    Super::Fire(HitTarget);
    APawn* InstigatorPawn = Cast<APawn>(GetOwner());
    const auto MuzzleFlashSocket =
        GetWeaponMesh()->GetSocketByName(FName("MuzzleFlash"));
    if (MuzzleFlashSocket)
    {
        FTransform SocketTransform =
            MuzzleFlashSocket->GetSocketTransform(GetWeaponMesh());
        // From muzzle flash socket to hit location from TraceUnderCrosshairs
        FVector ToTarget = HitTarget - SocketTransform.GetLocation();
        FRotator TargetRotation = ToTarget.Rotation();
        if (ProjectileClass && InstigatorPawn)
        {
            FActorSpawnParameters SpawnParams;
            SpawnParams.Owner = GetOwner();
            SpawnParams.Instigator = InstigatorPawn;
            UWorld* World = GetWorld();
            if (World)
            {
                World->SpawnActor<AProjectile>(
                    ProjectileClass,
                    SocketTransform.GetLocation(),
                    TargetRotation,
                    SpawnParams
                );
            }
        }
    }
}
```

需要注意的是，在 Combat.cpp 中我们有几处调用了 Fire 函数的地方，所以我们需要先设置一个 FVector 成员变量用于接受射线检测返回的位置，然后传给 Fire 函数。



然后我们就实现了子弹。

066 发射物移动组件

在 projectile 类中创建一个移动组件来实现子弹的移动，这个组件是 UE 自带的组件。

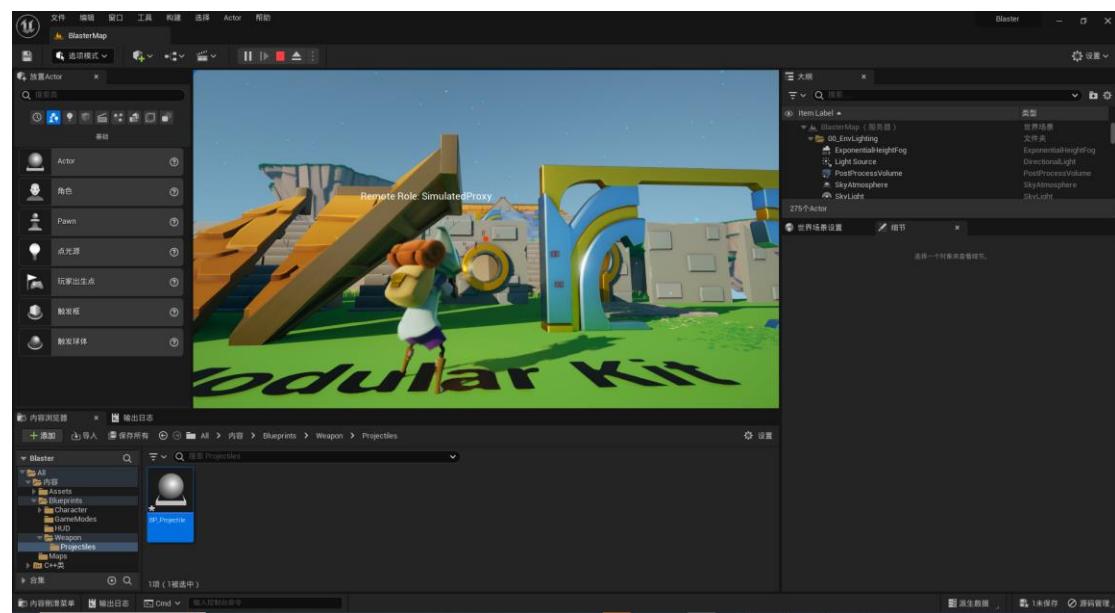
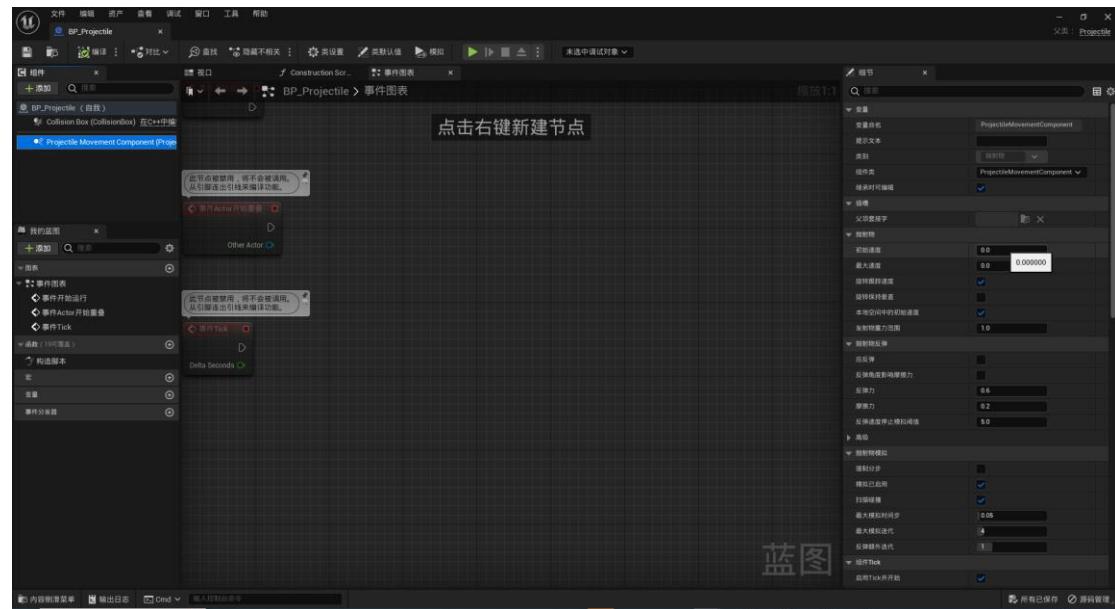
UPROPERTY(VisibleAnywhere)

```
class UProjectileMovementComponent* ProjectileMovementComponent;
```

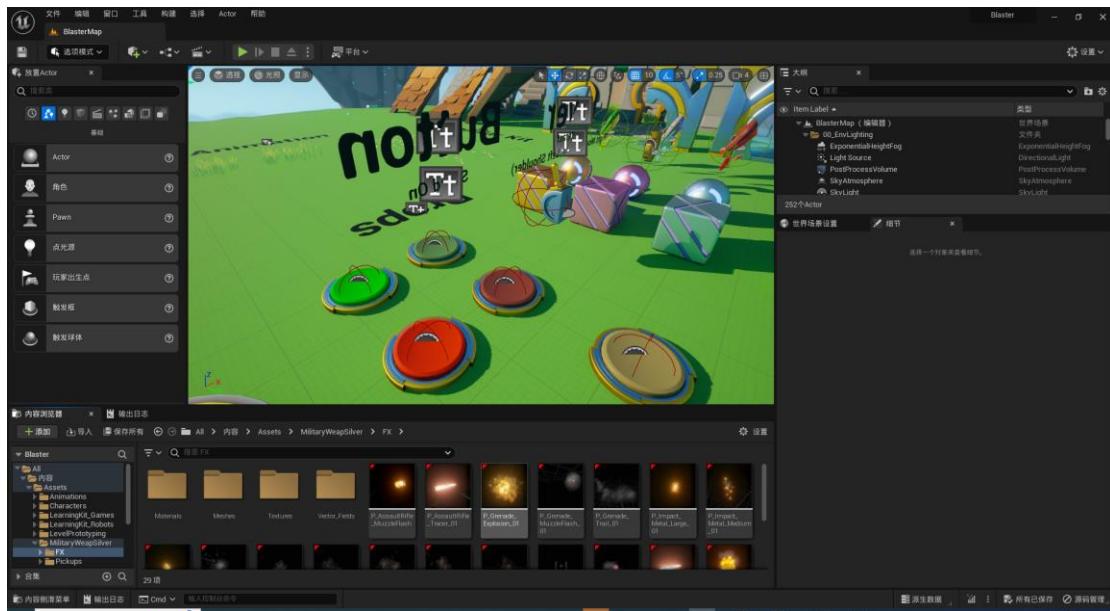
```
#include "GameFramework/ProjectileMovementComponent.h"
```

```
ProjectileMovementComponent  
CreateDefaultSubobject<UProjectileMovementComponent>(TEXT("ProjectileMovementComponent"));  
ProjectileMovementComponent->bRotationFollowsVelocity = true;
```

在 projectile 蓝图中我们可以设置子弹的速度：



067 投射物示踪



如下位置有我们所需的效果。

在 Projectile.h 中创建两个成员：

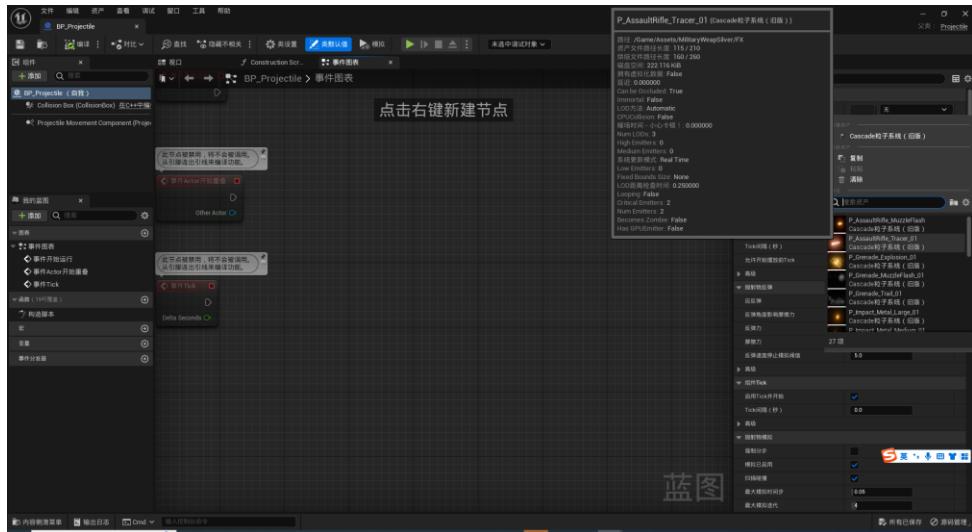
```
UPROPERTY(EditAnywhere)
class UParticleSystem* Tracer;
```

```
class UParticleSystemComponent* TracerComponent;
```

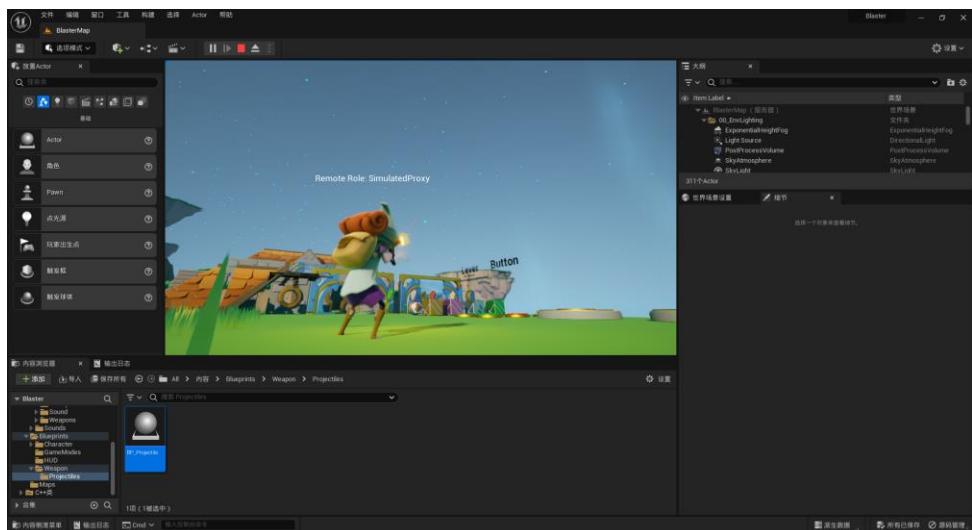
在.cpp 的 beginPlay 中生成发射器：

```
#include "Kismet/GameplayStatics.h"
#include "Particles/ParticleSystemComponent.h"
#include "Particles/ParticleSystem.h"
void AProjectile::BeginPlay()
{
    Super::BeginPlay();

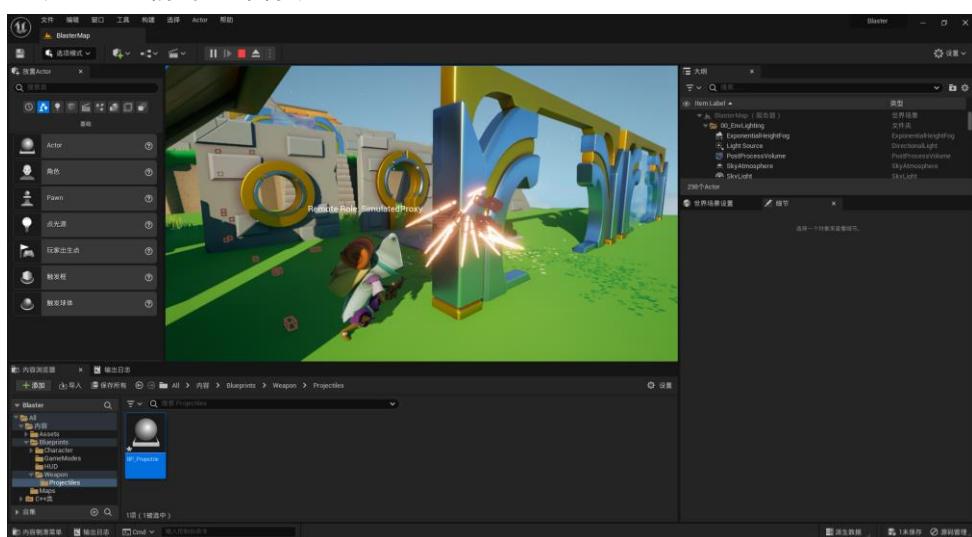
    if (Tracer)
    {
        TracerComponent = UGameplayStatics::SpawnEmitterAttached(
            Tracer,
            CollisionBox,
            FName(),
            GetActorLocation(),
            GetActorRotation(),
            EAttachLocation::KeepWorldPosition
        );
    }
}
```



打开蓝图设置子弹轨迹特效



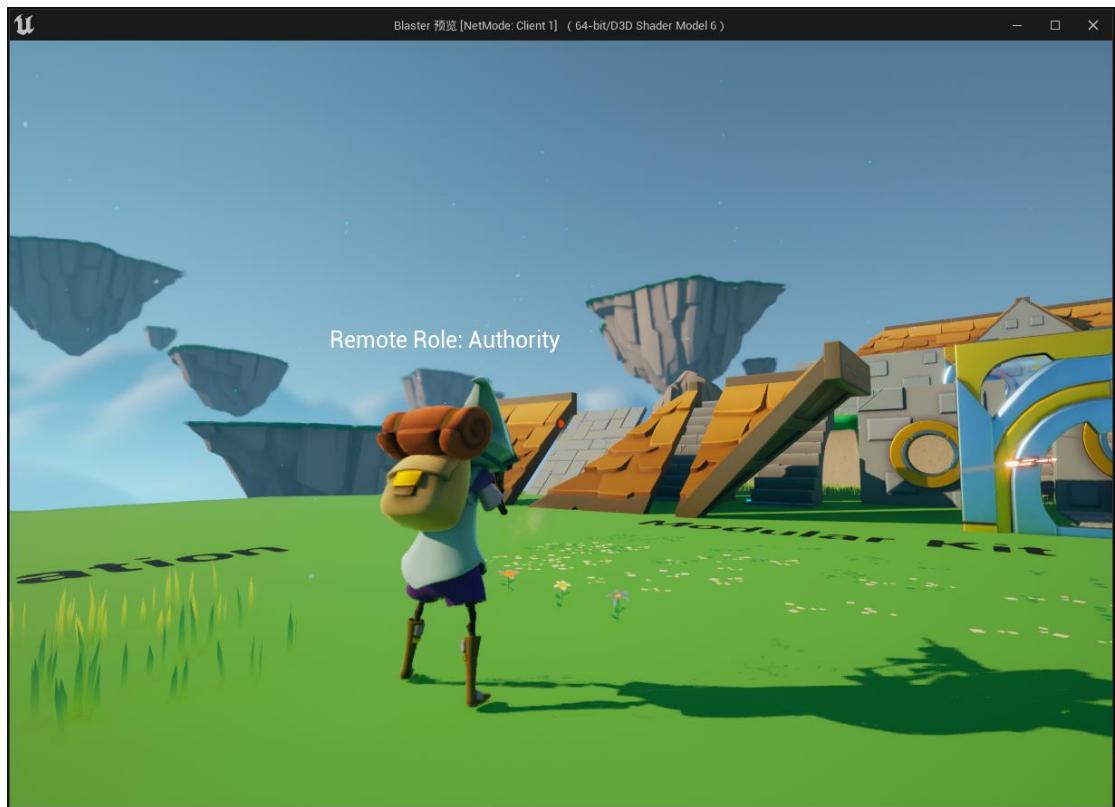
可以看到我们的子弹特效了。



但是我们只能在服务器上看到这个特效。我们现在需要在 Projectile.cpp 构造函数中添加一行：

```
bReplicates = true;
```

开启子弹的复制，如果不复制，每个子弹都是客户端自己生成的，且相互独立，如果开启复制，则子弹是由服务器生成的，服务器拥有权威。



但是我们可以看到客户端上的子弹并不能正确的飞向目标，这个问题将在下一节解决。

068 复制瞄准目标

我们可以通过在之前的 RPC 函数中传递参数来解决客户端不知道命中的位置在哪的问题，由于我们只需要在开火时知道这个值，所以我们可以将他不放在 tick 中进行。

之前的中间变量 HitTarget 也可以删除。此外 UE 在传递 FVector 时为了减小包体使用了一个优化的数据结构。

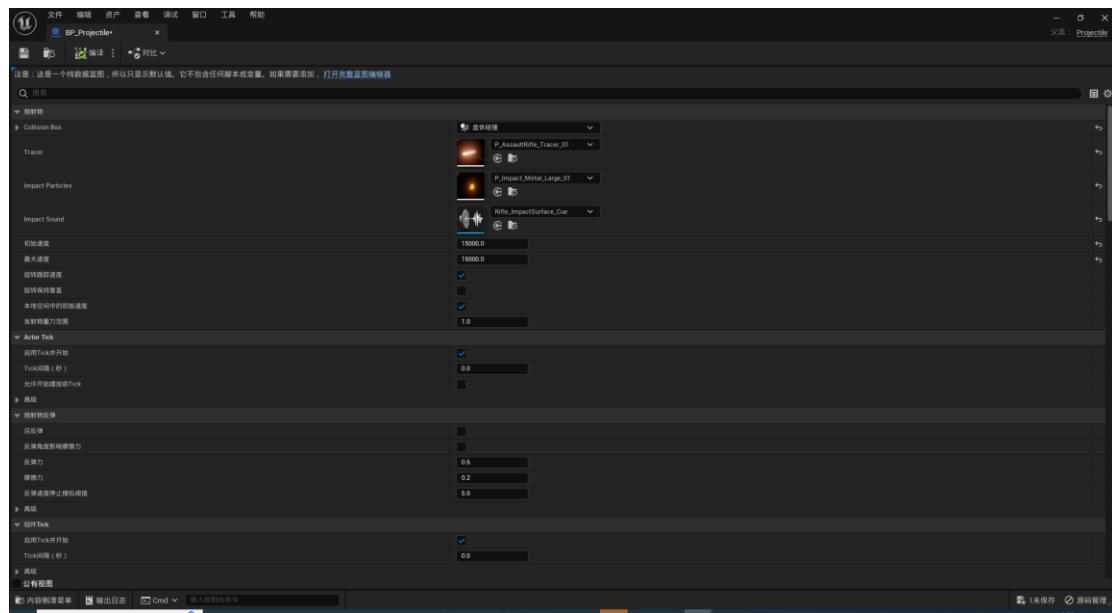
```
/**  
 *  FVector_NetQuantize  
 *  
 *  0 decimal place of precision.  
 *  Up to 20 bits per component.  
 *  Valid range: 2^20 = +/- 1,048,576  
 *  
 *  Note: this is the historical UE format for vector net serialization  
 */  
  
void UCombatComponent::ServerFire_Implementation(const FVector_NetQuantize&  
TraceHitTarget)  
{  
    MulticastFire(TraceHitTarget);  
}  
  
void UCombatComponent::MulticastFire_Implementation(const FVector_NetQuantize&  
TraceHitTarget)  
{  
    if (nullptr == EquippedWeapon)  
    {  
        return;  
    }  
    if (Character)  
    {  
        Character->PlayFireMontage(bAiming);  
        EquippedWeapon->Fire(TraceHitTarget);  
    }  
}
```

069 投掷物命中事件

给投掷物添加添加两个成员：
命中声音和命中效果，并在蓝图中修改。

```
UPROPERTY(EditAnywhere)
UParticleSystem* ImpactParticles;

UPROPERTY(EditAnywhere)
class USoundCue* ImpactSound;
```



添加一个命中事件，重写 `Destroyed()` 函数，不直接在命中的回调函数中播放声音和效果是因为回调函数只会在服务端执行：

```
void AProjectile::OnHit(UPrimitiveComponent* HitComp, AActor* OtherActor,
UPrimitiveComponent* OtherComp, FVector NormalImpulse, const FHitResult& Hit)
{
    Destroy();
}

void AProjectile::Destroyed()
{
    Super::Destroyed();

    if (ImpactParticles)
    {
        UGameplayStatics::SpawnEmitterAtLocation(GetWorld(), ImpactParticles,
GetActorTransform());
    }
    if (ImpactSound)
```

```
{  
    UGameplayStatics::PlaySoundAtLocation(this, ImpactSound, GetActorLocation());  
}  
}
```

070 子弹壳

添加一个子弹壳类，在武器中添加一个该类的成员。

修改开火的函数：

```
void AWeapon::Fire(const FVector& HitTarget)
{
    if (FireAnimation)
    {
        WeaponMesh->PlayAnimation(FireAnimation, false);
    }
    if (CasingClass)
    {
        const auto AmmoEjectSocket = WeaponMesh->GetSocketByName(FName("AmmoEject"));
        if (AmmoEjectSocket)
        {
            FTransform SocketTransform =
AmmoEjectSocket->GetSocketTransform(WeaponMesh);

            UWorld* World = GetWorld();
            if (World)
            {
                World->SpawnActor<ACasing>(
                    CasingClass,
                    SocketTransform.GetLocation(),
                    SocketTransform.GetRotation().Rotator()
                );
            }
        }
    }
}
```

071 弹壳的物理

子弹壳构造函数中设置：

```
ACasing::ACasing()
{
    PrimaryActorTick.bCanEverTick = false;

    CasingMesh = CreateDefaultSubobject<UStaticMeshComponent>(TEXT("CasingMesh"));
    SetRootComponent(CasingMesh);
    CasingMesh->SetCollisionResponseToChannel(ECollisionChannel::ECC_Camera,
        ECollisionResponse::ECR_Ignore);
    CasingMesh->SetSimulatePhysics(true);
    CasingMesh->SetEnableGravity(true);
    CasingMesh->SetNotifyRigidBodyCollision(true);
    ShellEjectionImpulse = 10. f;
}
```

添加弹壳的音效。

选做

给弹壳添加旋转

武器瞄准机制

072-073 Blaster 的 HUD 和玩家控制器

给武器类添加 5 个准星的纹理类:

```
/**  
 * Textures for the weapon crosshairs  
 */  
UPROPERTY(EditAnywhere, Category = "Crosshairs")  
class UTexture2D* CrosshairsCenter;  
  
UPROPERTY(EditAnywhere, Category = "Crosshairs")  
UTexture2D* CrosshairsLeft;  
  
UPROPERTY(EditAnywhere, Category = "Crosshairs")  
UTexture2D* CrosshairsRight;  
  
UPROPERTY(EditAnywhere, Category = "Crosshairs")  
UTexture2D* CrosshairsTop;  
  
UPROPERTY(EditAnywhere, Category = "Crosshairs")  
UTexture2D* CrosshairsBottom;
```

在 BlasterHUD 类中重写 DrawHUD 函数，绘制准星。

我们使用一个结构体来记录准星的数据:

```
USTRUCT(BlueprintType)  
struct FHUDPackage  
{  
    GENERATED_BODY()  
public:  
    class UTexture2D* CrosshairsCenter;  
    UTexture2D* CrosshairsLeft;  
    UTexture2D* CrosshairsRight;  
    UTexture2D* CrosshairsTop;  
    UTexture2D* CrosshairsBottom;  
    float CrosshairSpread;  
};
```

绘制过程中使用:

```
void ABlasterHUD::DrawHUD()  
{  
    Super::DrawHUD();
```

```

FVector2D ViewportSize;
if (GEngine)
{
    GEngine->GameViewport->GetViewportSize(ViewportSize);
    const FVector2D ViewportCenter(ViewportSize.X / 2.f, ViewportSize.Y / 2.f);

    float SpreadScaled = CrosshairSpreadMax * HUDPackage.CrosshairSpread;

    if (HUDPackage.CrosshairsCenter)
    {
        FVector2D Spread(0.f, 0.f);
        DrawCrosshair(HUDPackage.CrosshairsCenter, ViewportCenter, Spread);
    }
    if (HUDPackage.CrosshairsLeft)
    {
        FVector2D Spread(-SpreadScaled, 0.f);
        DrawCrosshair(HUDPackage.CrosshairsLeft, ViewportCenter, Spread);
    }
    if (HUDPackage.CrosshairsRight)
    {
        FVector2D Spread(SpreadScaled, 0.f);
        DrawCrosshair(HUDPackage.CrosshairsRight, ViewportCenter, Spread);
    }
    if (HUDPackage.CrosshairsTop)
    {
        FVector2D Spread(0.f, -SpreadScaled);
        DrawCrosshair(HUDPackage.CrosshairsTop, ViewportCenter, Spread);
    }
    if (HUDPackage.CrosshairsBottom)
    {
        FVector2D Spread(0.f, SpreadScaled);
        DrawCrosshair(HUDPackage.CrosshairsBottom, ViewportCenter, Spread);
    }
}

void ABlasterHUD::DrawCrosshair(UTexture2D* Texture, FVector2D ViewportCenter, FVector2D
Spread)
{
    const float TextureWidth = Texture->GetSizeX();
    const float TextureHeight = Texture->GetSizeY();
    const FVector2D TextureDrawPoint(
        ViewportCenter.X - (TextureWidth / 2.f) + Spread.X,

```

```

        ViewportCenter.Y - (TextureHeight / 2. f) + Spread.Y
    );

    DrawTexture(
        Texture,
        TextureDrawPoint.X,
        TextureDrawPoint.Y,
        TextureWidth,
        TextureHeight,
        0. f,
        0. f,
        1. f,
        1. f,
        FLinearColor::White
    );
}

```

为此我们需要设置 FHUDPackage，我们在 combat 类中实现，在 combat 的 tick 中调用：

```

void UCombatComponent::SetHUCrosshairs(float DeltaTime)
{
    if (Character == nullptr || Character->Controller == nullptr)
    {
        return;
    }

    Controller = Controller == nullptr ?
        Cast<ABlasterPlayerController>(Character->Controller) : Controller;
    if (Controller)
    {
        HUD = HUD == nullptr ? Cast<ABlasterHUD>(Controller->GetHUD()) : HUD;
        if (HUD)
        {
            FHUDPackage HUDPackage;
            if (EquippedWeapon)
            {
                HUDPackage.CrosshairsCenter = EquippedWeapon->CrosshairsCenter;
                HUDPackage.CrosshairsTop = EquippedWeapon->CrosshairsTop;
                HUDPackage.CrosshairsLeft = EquippedWeapon->CrosshairsLeft;
                HUDPackage.CrosshairsRight = EquippedWeapon->CrosshairsRight;
                HUDPackage.CrosshairsBottom = EquippedWeapon->CrosshairsBottom;
            }
            else
            {
                HUDPackage.CrosshairsCenter = nullptr;
            }
        }
    }
}

```

```

        HUDPackage.CrosshairsTop = nullptr;
        HUDPackage.CrosshairsLeft = nullptr;
        HUDPackage.CrosshairsRight = nullptr;
        HUDPackage.CrosshairsBottom = nullptr;
    }

    // Calculate crosshair spread

    // Clamp speed [0, MaxWalkSpeed] -> [0, 1]
    FVector2D WalkSpeedRange(0. f,
Character->GetCharacterMovement()->MaxWalkSpeed);
    FVector2D VelocityMultiplierRange(0. f, 1. f);
    FVector Velocity = Character->GetVelocity();
    Velocity.Z = 0. f;
    float Speed = Velocity.Size();
    CrosshairVelocityFactor =
FMath::GetMappedRangeValueClamped(WalkSpeedRange, VelocityMultiplierRange, Speed);

    if (Character->GetCharacterMovement()->IsFalling())
    {
        CrosshairInAirFactor = FMath::FInterpTo(CrosshairInAirFactor,
2.25f, DeltaTime, 2.25f);
    }
    else
    {
        CrosshairInAirFactor = FMath::FInterpTo(CrosshairInAirFactor, 0. f,
DeltaTime, 30. f);
    }

    HUDPackage.CrosshairSpread = CrosshairVelocityFactor +
CrosshairInAirFactor;

    HUD->SetHUDPackage(HUDPackage);
}
}
}

```

从而实现了动态准星的绘制，后续我们添加功能时可以进一步添加相应的影响准星扩散的 Factor。

074 正确的武器旋转

为了显示当前武器旋转和目标的不一致，我们在动画蓝图类的末尾添加如下 Debug 语句：

```
FTransform MuzzleTipTransform =
EquippedWeapon->GetWeaponMesh()->GetSocketTransform(FName("MuzzleFlash"),
ERelativeTransformSpace::RTS_World);
FVector
MuzzleX(FRotationMatrix(MuzzleTipTransform.GetRotation().Rotator()).GetUnitAxis(EAxis::
X));
DrawDebugLine(GetWorld(), MuzzleTipTransform.GetLocation(),
MuzzleTipTransform.GetLocation() + MuzzleX * 1000. f, FColor::Red);
DrawDebugLine(GetWorld(), MuzzleTipTransform.GetLocation(),
BlasterCharacter->GetHitTarget(), FColor::Orange);
```

在 BlasterCharacter 中添加：

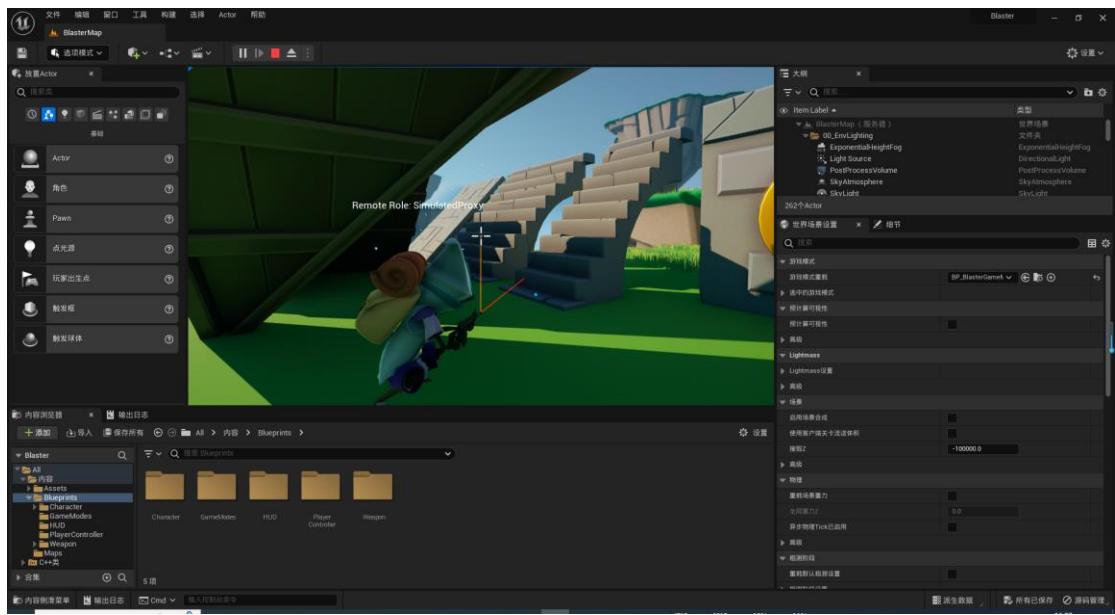
```
FVector ABlasterCharacter::GetHitTarget() const
{
    if (Combat == nullptr)
    {
        return FVector();
    }
    return Combat->HitTarget;
}
```

在 Combat 中，我们需要：

添加一个变量：HitTarget

在 Tick 中：

```
if (Character && Character->IsLocallyControlled())
{
    FHitResult HitResult;
    TraceUnderCrosshairs(HitResult);
    HitTarget = HitResult.ImpactPoint;
}
```



可以很明显的看到，枪口所指向的位置并不是我们所瞄准的位置。

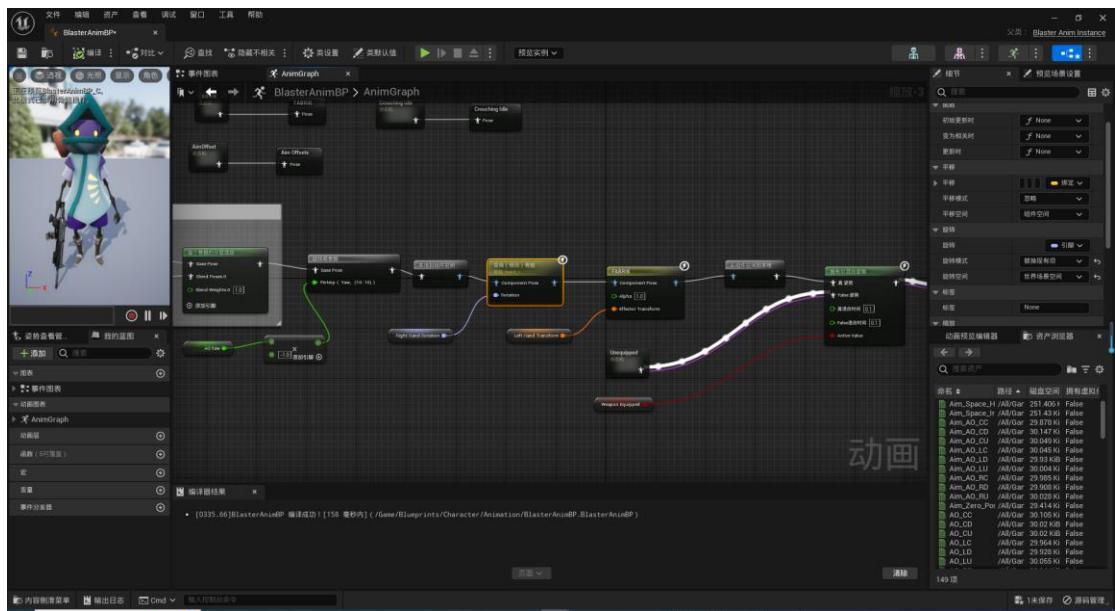
我们现在要做的就是将枪口旋转到方向与瞄准的位置一致。

添加一个变量在动画类中：

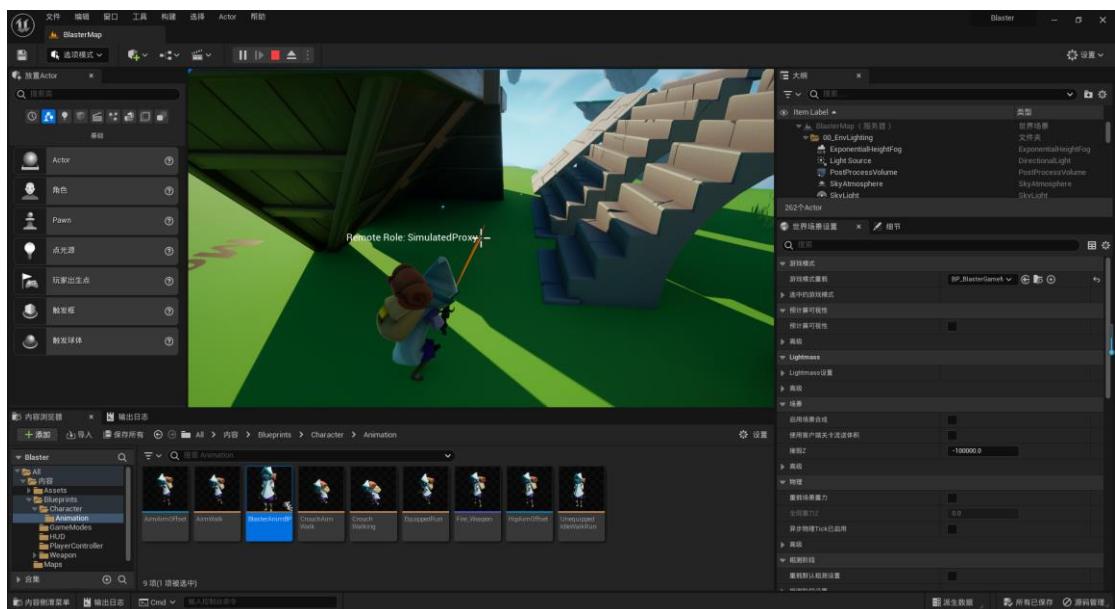
```
UPROPERTY(BlueprintReadOnly, Category = Movement, meta = (AllowPrivateAccess = "true"))
FRotator RightHandRotation;
```

动画蓝图更新中：

```
if (BlasterCharacter->IsLocallyControlled())
{
    FTransform RightHandTransform =
        EquippedWeapon->GetWeaponMesh()->GetSocketTransform(FName("Hand_R"),
        ERelativeTransformSpace::RTS_World);
    RightHandRotation =
        UKismetMathLibrary::FindLookAtRotation(RightHandTransform.GetLocation(),
        RightHandTransform.GetLocation() + (RightHandTransform.GetLocation()
        - BlasterCharacter->GetHitTarget()));
}
```



在动画蓝图中旋转根骨骼之后进行旋转



我们发现虽然大约对齐了瞄准的位置，但是我们之前所用的 FABRIK 失效了，这是因为我们再一开始就是用了 FABRIK，所以我们需要吧之前的 Use cached pose FABRIK 替换正 Use cached pose AimOffset，然后再完成骨骼旋转后再添加 FABRIK

最后由于这个对齐过程每帧都会进行，所以我们没有必要在每台机器上都这样做，我们只需要对控制着的机器进行对齐，其他机器就用 Aimoffset 就好。

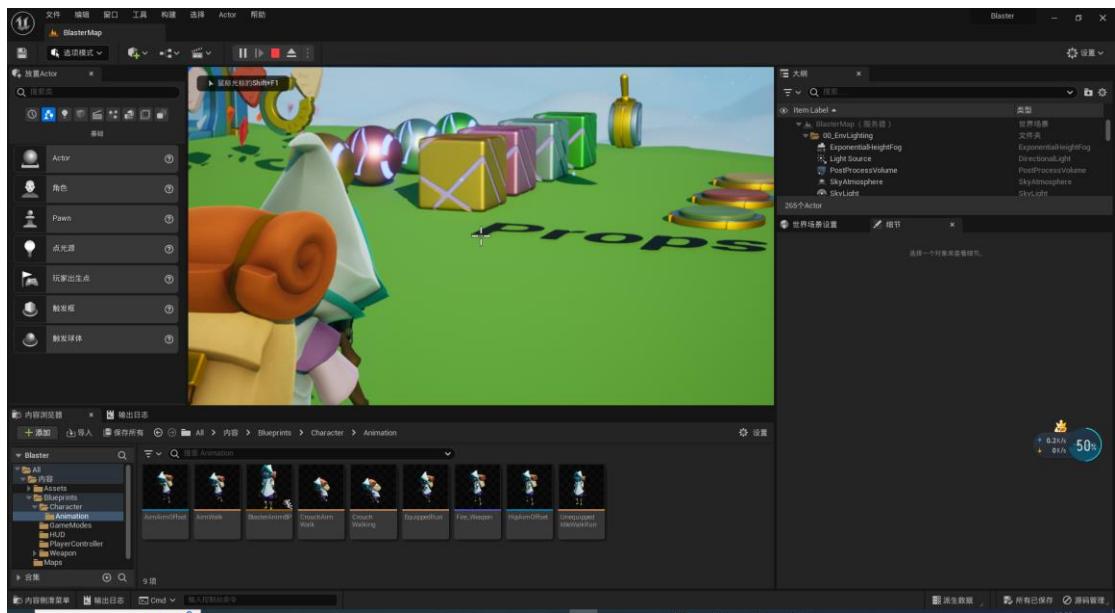
075 变焦瞄准 Zoom Aiming

在 Weapon 中设置瞄准变焦的参数：

```
/**  
 * Zoomed FOV while aiming  
 */  
  
UPROPERTY(EditAnywhere)  
float ZoomedFOV = 30.f;  
  
UPROPERTY(EditAnywhere)  
float ZoomInterpSpeed = 20.f;
```

在 Combat 的 tick 函数中实现瞄准变焦：

```
void UCombatComponent::InterpFOV(float DeltaTime)  
{  
    if (EquippedWeapon == nullptr)  
    {  
        return;  
    }  
  
    if (bAiming)  
    {  
        CurrentFOV = FMath::FInterpTo(CurrentFOV, EquippedWeapon->GetZoomedFOV(),  
        DeltaTime, EquippedWeapon->GetZoomInterpSpeed());  
    }  
    else  
    {  
        CurrentFOV = FMath::FInterpTo(CurrentFOV, DefaultFOV, DeltaTime,  
        ZoomInterpSpeed);  
    }  
  
    if (Character && Character->GetFollowCamera())  
    {  
        Character->GetFollowCamera()->SetFieldOfView(CurrentFOV);  
    }  
}
```



为了应对出现变焦后模糊的情况，我们可以把光圈设置到最大，把焦距设置得足够远。

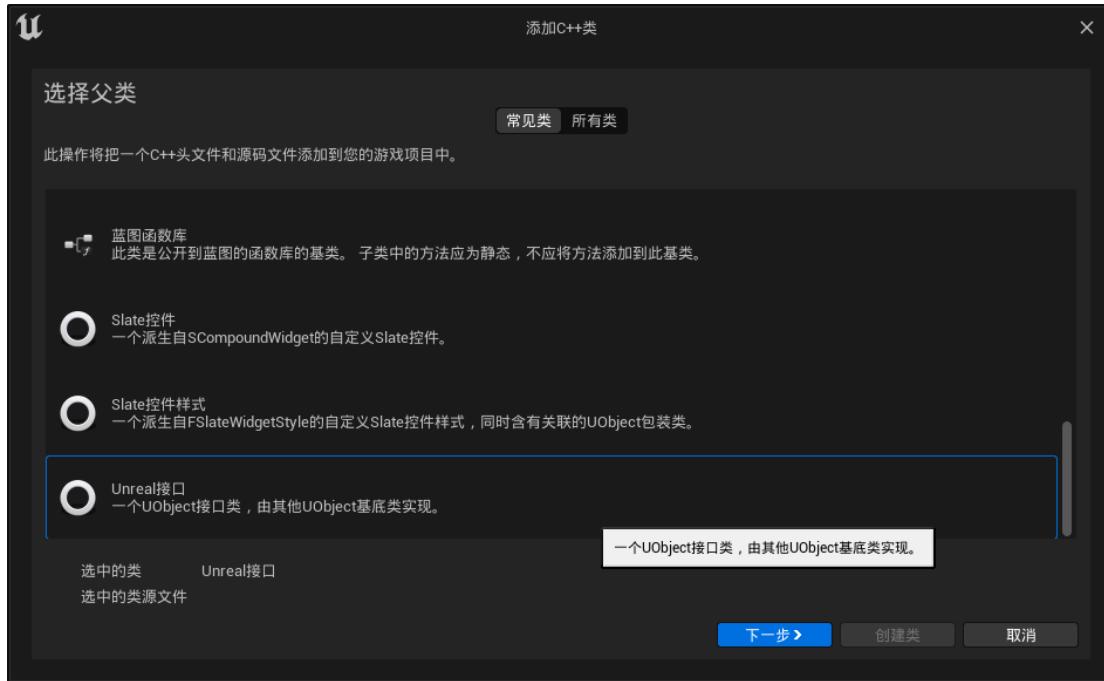


076 瞄准时缩小准星

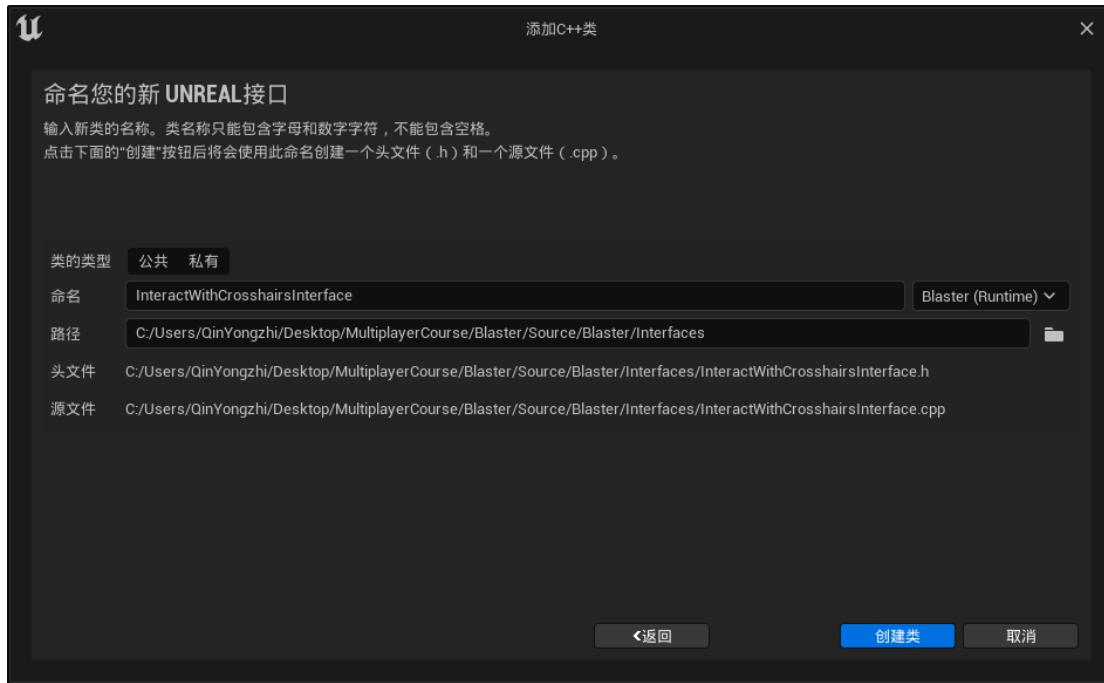
添加两个影响因子：一个在射击时增加，并且不断往回插值，另一个在按下瞄准时增加，松开时减少即可。

077 改变准星颜色

我们角色的网格体碰撞 ignore 了通道 visibility, 所以不会返回射线检测的结果。



创建一个 Unreal interface,



来到 BlasterCharacter.h 中，修改 class 的继承，增加一个接口的继承：

```
#include "Blaster/Interfaces/InteractWithCrosshairsInterface.h"
UCLASS()
class BLASTER_API ABlasterCharacter : public ACharacter, public IInteractWithCrosshairsInterface
```

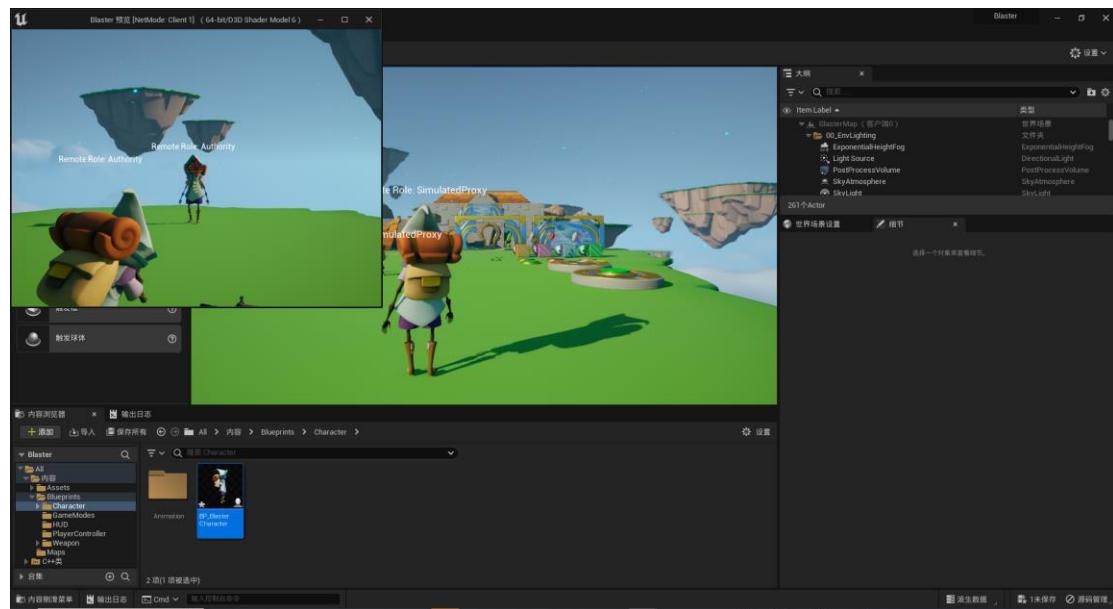
修改 HUD 中的结构体，增加一个颜色变量

```
USTRUCT(BlueprintType)
struct FHUDPackage
{
    GENERATED_BODY()

public:
    class UTexture2D* CrosshairsCenter;
    UTexture2D* CrosshairsLeft;
    UTexture2D* CrosshairsRight;
    UTexture2D* CrosshairsTop;
    UTexture2D* CrosshairsBottom;
    float CrosshairSpread;
    FLinearColor CrosshairsColor;
};
```

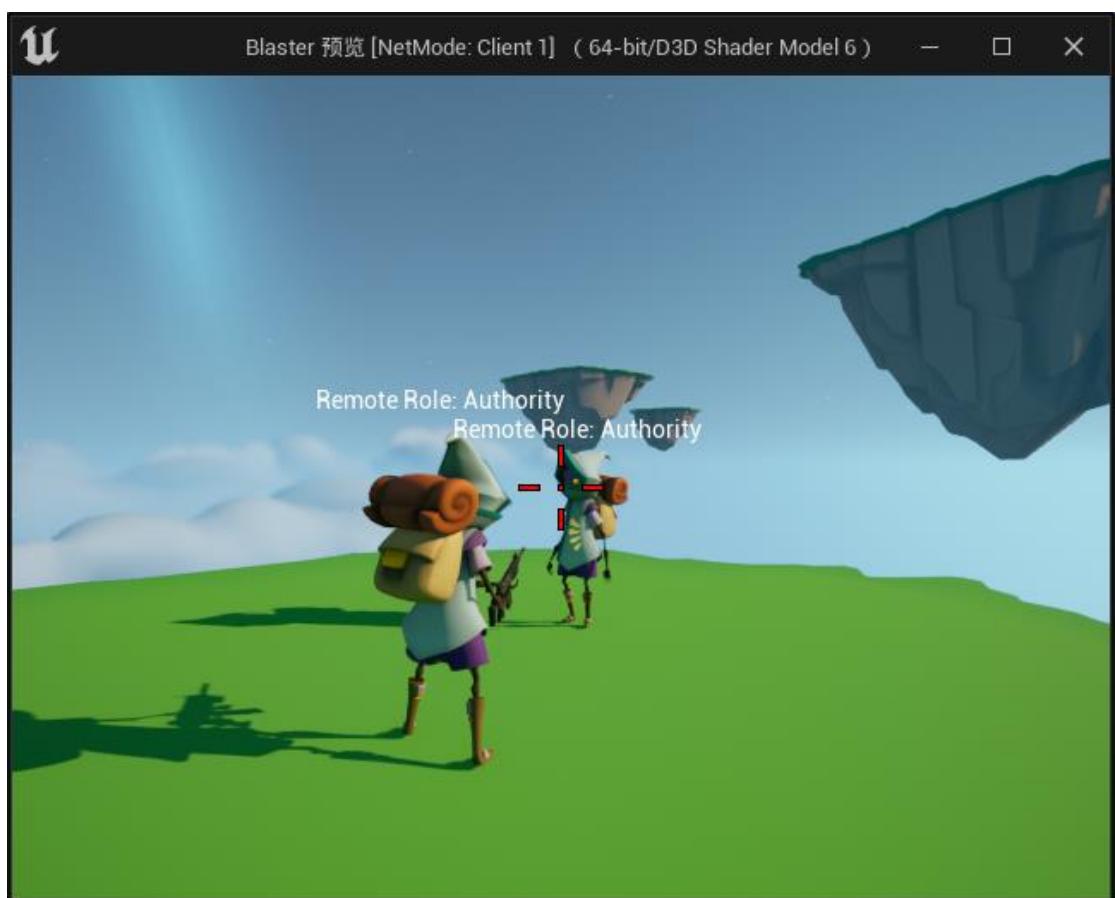
在 `UCombatComponent::TraceUnderCrosshairs` 中对命中的目标执行接口，判断是否需要变色，然后修改 `HUDPackage` 的准星颜色值。

注意：要检查角色蓝图中所用的角色是否拥有物理资产，不然无法成功检测。



还有一个问题是，我们的枪由于会指向我们的目标，造成枪的吸附过于明显，这个问题可以用一个简单的插值结局：

```
FRotator LookAtRotation =
UKismetMathLibrary::FindLookAtRotation(RightHandTransform.GetLocation(), RightHandTransform.GetLocation() + (RightHandTransform.GetLocation() -
BlasterCharacter->GetHitTarget()));
RightHandRotation = FMath::RInterpTo(RightHandRotation, LookAtRotation,
DeltaTime, 30. f);
```

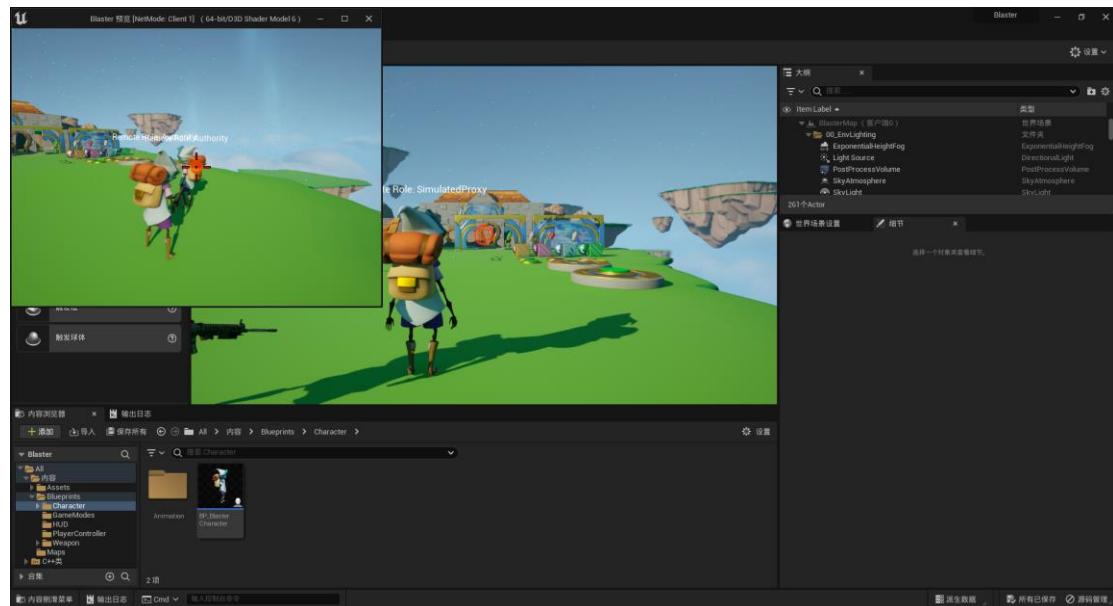


078 增加射线的起点

我们现在的射线检测是从摄像机发出的，这会导致我们射击的目标变成身后的目标，以及会检测到我们自己，此外我们还需要当摄像机离我们很近的时候隐藏掉摄像机。

按照如下代码，我们在我们身前 100 个单位的位置作为起点进行射线检测，看看结果怎样：

```
if (Character)
{
    float DistanceToCharacter = (Character->GetActorLocation() - Start).Size();
    Start += CrosshairWorldDirection * (DistanceToCharacter + 100. f);
    DrawDebugSphere(GetWorld(), Start, 16. f, 12, FColor::Red, false);
}
```



100 个单位看起来还可以。

接下来，在相机离角色太近时，我们隐藏相机，在角色的 Tick 中调用以下函数：

```
void ABlasterCharacter::HideCameraIfCharacterClose()
{
    if (!IsLocallyControlled())
    {
        return;
    }

    if ((FollowCamera->GetComponentLocation() - GetActorLocation()).Size() <
        CameraThreshold)
    {
        GetMesh()->SetVisibility(false);
        if (Combat && Combat->EquippedWeapon &&
            Combat->EquippedWeapon->GetWeaponMesh())
    }
}
```

```
{  
    Combat->EquippedWeapon->GetWeaponMesh()->bOwnerNoSee = true;  
}  
}  
else  
{  
    GetMesh()->SetVisibility(true);  
    if (Combat && Combat->EquippedWeapon &&  
Combat->EquippedWeapon->GetWeaponMesh())  
    {  
        Combat->EquippedWeapon->GetWeaponMesh()->bOwnerNoSee = false;  
    }  
}  
}
```

079 命中角色



将受击动画制作成 additive 动画。

将动画制作成蒙太奇。

打开 VS，为我们的角色添加一个蒙太奇成员：

```
UPROPERTY(EditAnywhere, Category = Combat)
class UAnimMontage* HitReactMontage;
```

并添加一个播放蒙太奇的函数：

```
void ABlasterCharacter::PlayHitReactMontage()
{
    if (nullptr == Combat || nullptr == Combat->EquippedWeapon)
    {
        return;
    }

    UAnimInstance* AnimInstance = GetMesh()->GetAnimInstance();
    if (AnimInstance && HitReactMontage)
    {
        AnimInstance->Montage_Play(HitReactMontage);
        FName SectionName("FromFront");
        AnimInstance->Montage_JumpToSection(SectionName);
    }
}
```

在 Projectile 类中，我们修改 OnHit 函数：

```
void AProjectile::OnHit(UPrimitiveComponent* HitComp, AActor* OtherActor,
```

```

UPrimitiveComponent* OtherComp, FVector NormalImpulse, const FHitResult& Hit)
{
    ABlasterCharacter* BlasterCharacter = Cast<ABlasterCharacter>(OtherActor);
    if (BlasterCharacter)
    {
        BlasterCharacter->PlayHitReactMontage();
    }
    Destroy();
}

```

并且设置 Projectile 于 Pawn 的碰撞，并且在蓝图中进行 double check:

```

CollisionBox->SetCollisionResponseToChannel(ECollisionChannel::ECC_Pawn,
ECollisionResponse::ECR_Block);

```

最后打开动画插槽，添加我们之前设置好的插槽即可。



运行游戏，我们发现受击动画只在服务端能够看到，这是因为在 Projectile 中：

```

if (HasAuthority())
{
    CollisionBox->OnComponentHit.AddDynamic(this, &APiece::OnHit);
}

```

为了让所有人都能看到蒙太奇动画，我们可以在角色中将这个播放蒙太奇的函数变成一个多播 RPC。

```

UFUNCTION(NetMulticast, Unreliable)
void MulticastHit();

```

蒙太奇并不影响游戏逻辑，所以我们可以选择不可靠 RPC。

```

void ABlasterCharacter::MulticastHit_Implementation()
{
    PlayHitReactMontage();
}

```

但是现在还有一个问题，那就是我们的胶囊体的碰撞通道也是 pawn，所以我们的子弹实际上也会达到 pawn 上，这不利于我们之后做爆头的判断，所以我们可以给 Mesh 单独设置一个碰撞通道解决这个问题。

打开项目设置搜索 Object channels:



为了方便理解，我们在 Blaster.h 中添加一个宏定义：

```
#define ECC_SkeletalMesh ECollisionChannel::ECC_GameTraceChannel1
```

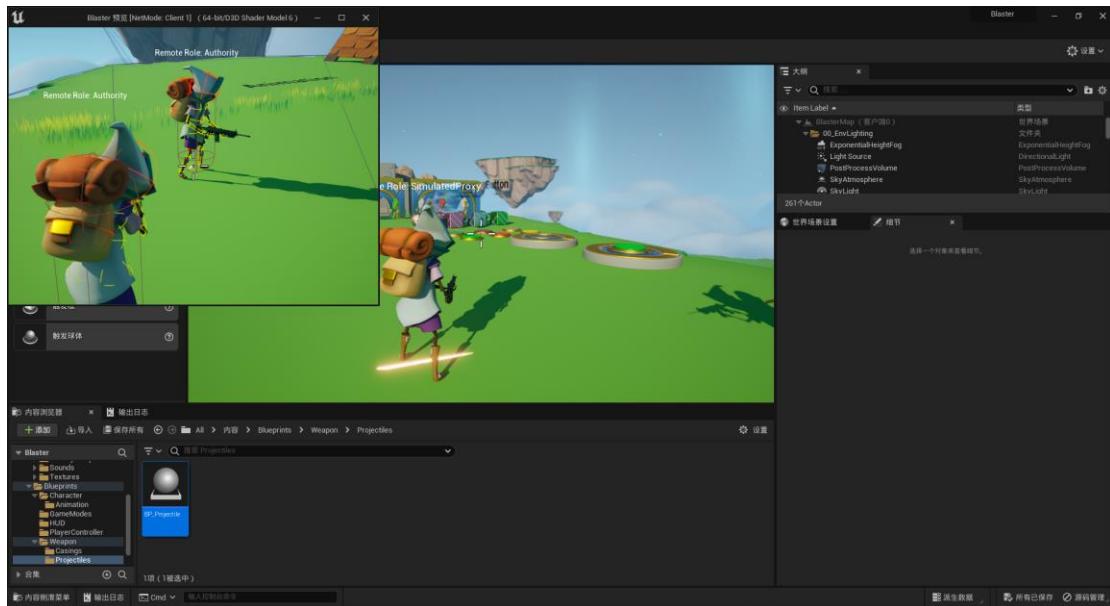
之前的 Projectile 的语句可以改成如下：

```
CollisionBox->SetCollisionResponseToChannel(ECC_SkeletalMesh,  
ECollisionResponse::ECR_Block);
```

在角色.cpp 中，设置碰撞通道：

```
GetMesh()->SetCollisionObjectType(ECC_SkeletalMesh);
```

重新编译，然后再编辑器中设置 Mesh 的碰撞，并且关闭 Projectile 蓝图中的与 pawn 的碰撞，可以看到子弹不再和胶囊体产生碰撞了：



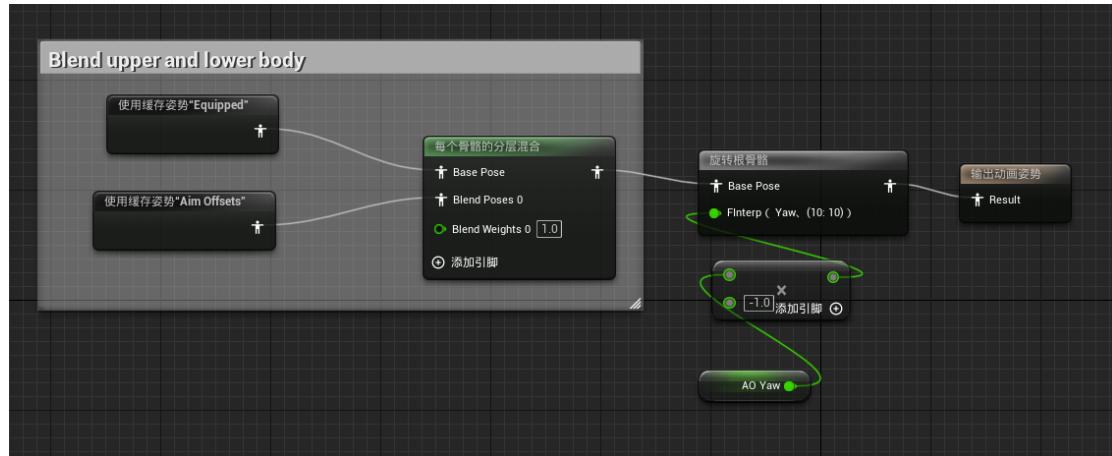
080 代理角色的平滑旋转

OnRep_ReplicatedMovement 函数：

这个函数在我们的角色 Movement 复制的时候会被调用。

我们会用他来使代理角色原地旋转变得平滑。

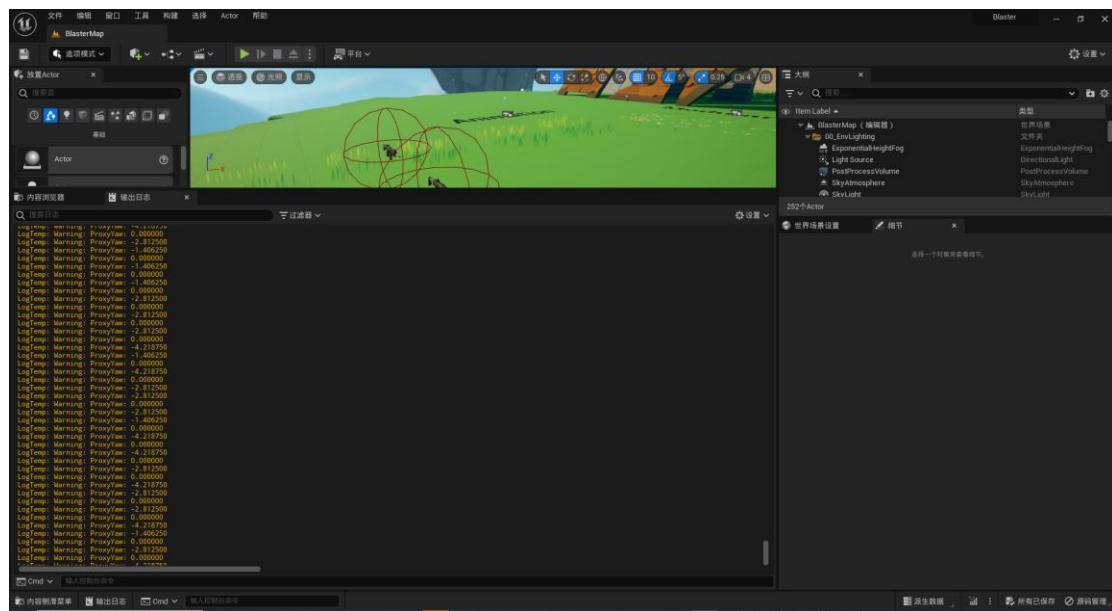
我们自己控制的角色原地旋转不会出现抖动，但是代理角色会发生抖动，这是因为：



我们是在动画蓝图中设置的旋转骨骼来抵消掉我们的旋转，但是动画蓝图并不是每帧更新的

我们的转身动画是在动画蓝图中通过获取是否需要转身的枚举类来实现的，原地转身的根本目的是为了避免原地转身时脚下的漂移，教程这里使用的方法是直接在转身的时候播放转身动画，这样一来虽然避免了漂移问题，但是会导致客户端看到的不是服务器中看到的情况，所以我本人对这种做法持保留意见。但是其实无所谓，我们已经学习了那么多的方法，之后觉得不合适就再改回去就好了，况且这一节的一些知识还是很不错的。

首先是 OnRep_ReplicatedMovement 函数的使用，如果我们直接在 tick 函数中进行更新：会出现如下结果：



可以看到旋转过程中有断续的 0 和数据，这是因为网络传递的速度比不上 tick 的速度，所以会导致这种问题，我们在 rep 中调用我们所需的转身函数，就可以避免这个问题，但是如果我们一直没有 rep 就不会调用这个函数了，所以我们在 tick 中进行一个计时，超过某个时间之后我们就自己调用一个 rep 函数即可。

选做项

1. 让准星在瞄准向另一个玩家的时候轻微收缩。
2. 射击命中角色时不生成粒子效果，否则生成粒子效果

首先我们的 OnHit 函数只在服务端产生，所以一个方法是设置一个 bool 变量并复制，但是由于子弹很多，每个子弹都复制一个变量的话对于带宽消耗不小，所以考虑在 Destroyed 函数中调用一个小范围的射线检测，查看是否击中了角色，把服务器的压力转移到客户端？

但是效果好像始终一般，考虑到子弹移动都已经复制了，所以没必要解约这一点点流量。如果要完美实现的话，还是考虑用了一个多播 RPC 调用取代了之前直接用 destroyed 函数在全部客户端上执行：

```
void AProjectile::MulticastHitEffect_Implementation(bool bHitted, const
FVector_NetQuantize& HitPoint)
{
    if (!bHitted)
    {
        if (ImpactParticles)
        {
            UGameplayStatics::SpawnEmitterAtLocation(GetWorld(), ImpactParticles,
HitPoint);
        }
        if (ImpactSound)
        {
            UGameplayStatics::PlaySoundAtLocation(this, ImpactSound, HitPoint);
        }
    }
    else
    {
        if (HitSound)
        {
            UGameplayStatics::PlaySoundAtLocation(this, HitSound, HitPoint);
        }
    }
}
Destroy();
```

081 自动开火（定时器用法）

给我们的武器添加两个变量：

```
/**  
 * Automatic fire  
 */  
UPROPERTY(EditAnywhere, Category = Combat)  
float FireDelay = .15f;  
  
UPROPERTY(EditAnywhere, Category = Combat)  
bool bAutomatic = true;
```

在 Combat 中使用一个计时器，为此我们需要两个函数，和一个判断是否可以开火的 bool 变量：

```
FTimerHandle FireTimer;  
bool bCanFire = true;  
void StartFireTimer();  
void FireTimerFinished();
```

计时器使用方法如下：

```
void UCombatComponent::StartFireTimer()  
{  
    if (EquippedWeapon == nullptr || Character == nullptr)  
    {  
        return;  
    }  
  
    Character->GetWorldTimerManager().SetTimer(  
        FireTimer,  
        this,  
        &UCombatComponent::FireTimerFinished,  
        EquippedWeapon->GetFireDelay()  
    );  
}
```

函数 FireTimerFinished() 作为回调函数。

```
void UCombatComponent::FireTimerFinished()  
{  
    bCanFire = true;  
    if (EquippedWeapon && bFireButtonPressed && EquippedWeapon->IsAutomatic())  
    {  
        Fire();  
    }  
}
```

082 测试游戏



完善了之前的一些选做项，并且测试了目前为止的游戏，Awesome Job!

生命值与玩家状态

083 游戏框架

再继续往下之前我们需要了解 UE 多人游戏的框架：

- GameMode：只存在于客户端，在客户端试图连接 GameMode 只会获得空指针。
- GameState：同时存在于服务器和客户端，服务器中的数据可以在 GameState 中复制到客户端。
- PlayerState：同时存在于服务器和客户端。
- PlayController：服务器拥有所有玩家控制器，客户端只拥有自己的控制器。
- Pawn：同时存在于服务器和客户端。
- HUD/Widgets：只存在于拥有的客户端。

每个种类又有其各自的职能：

GameMode

1. 设置了游戏的各种默认类，比如：Pawn, PlayerController, HUD。

在多人游戏中，GameMode 还有一些其他用处：

2. 维持游戏规则：
 - a) 玩家淘汰
 - b) 玩家重生
3. 保存 MatchState
 - a) Warmup Time
 - b) 游戏时间

GameState

1. 用来保存游戏状态
2. 最高分玩家
3. 队伍的队长
4. 队伍分数
5. 保存一个存有所有玩家状态的数组 TArray

PlayerState

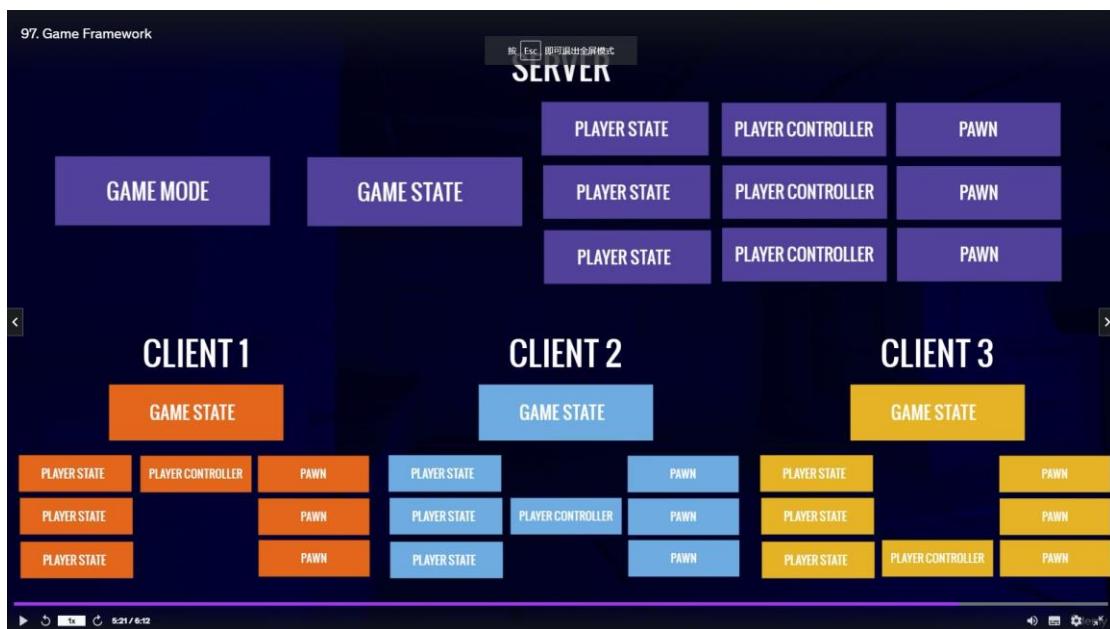
用来记录单独一个玩家的状态

- a) 分数
- b) 击败数
- c) 携带的弹药

d) 队伍归属

PlayerController

1. 连接 HUD
 - a) 显示信息
 - b) 生命值
 - c) 分数
 - d) 击败
 - e) 弹药

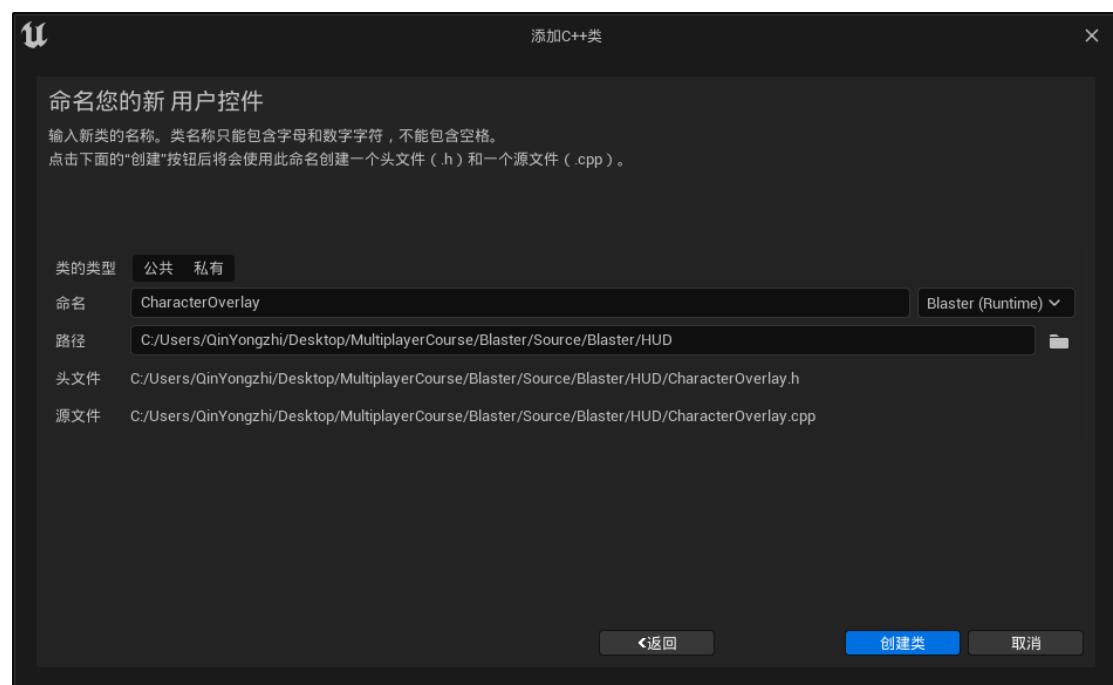


084 生命值

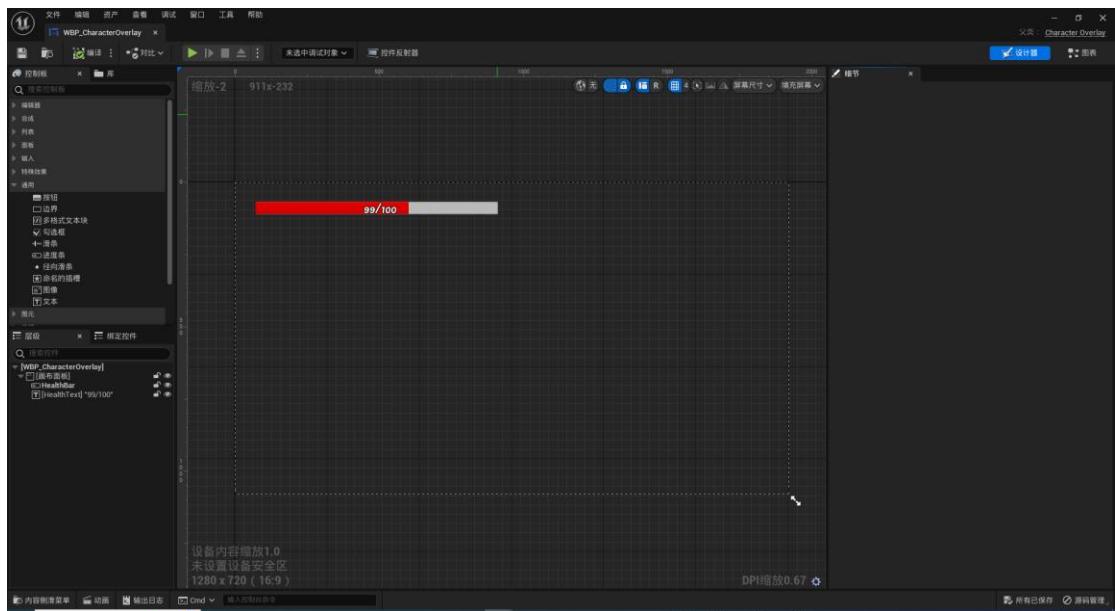
接下来我们要为角色添加生命值，我们会把生命值添加在角色中，为什么不添加在 PlayerState 中呢，因为生命值需要及时的更新，而角色的网络同步频率要高于 PlayerState，这会造成延迟。

在角色类中添加：

```
/**  
 * Player health  
 */  
UPROPERTY(EditAnywhere, Category = "Player Stats")  
float MaxHealth = 100.f;  
  
UPROPERTY(ReplicatedUsing = OnRep_Health, Category = "Player Stats",  
VisibleAnywhere)  
float Health = 100.f;  
  
UFUNCTION()  
void OnRep_Health();
```



创建一个新的用户控件蓝图，添加一个进度条以及一个文本框：

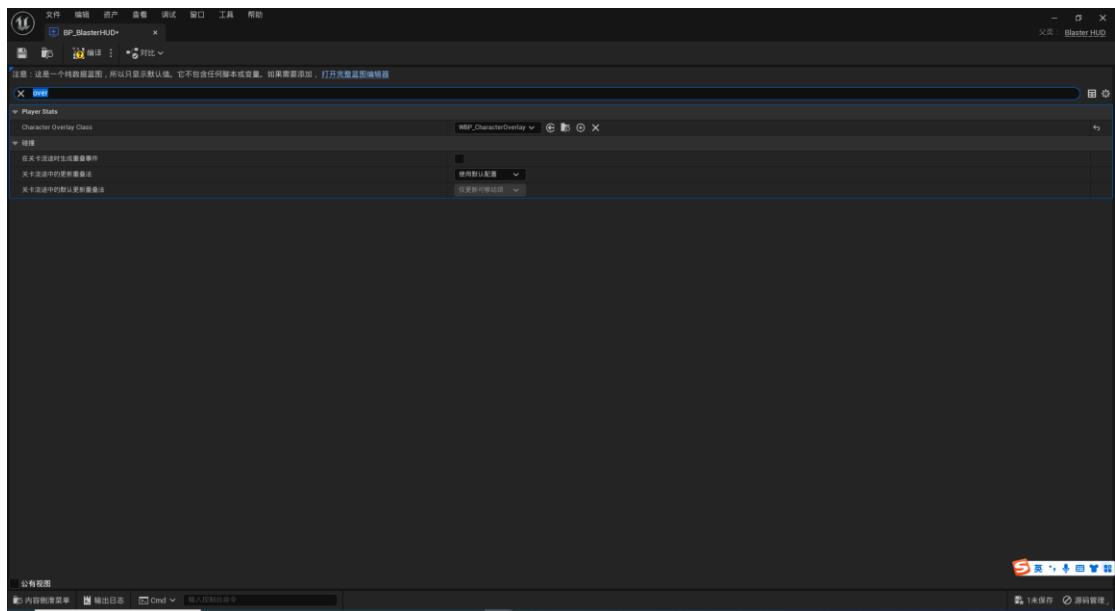


创建一个新的 C++ 用户控件类，在 BlasterHUD 中添加一个该类的变量，并重写 BeginPlay 函数用来设置该变量：

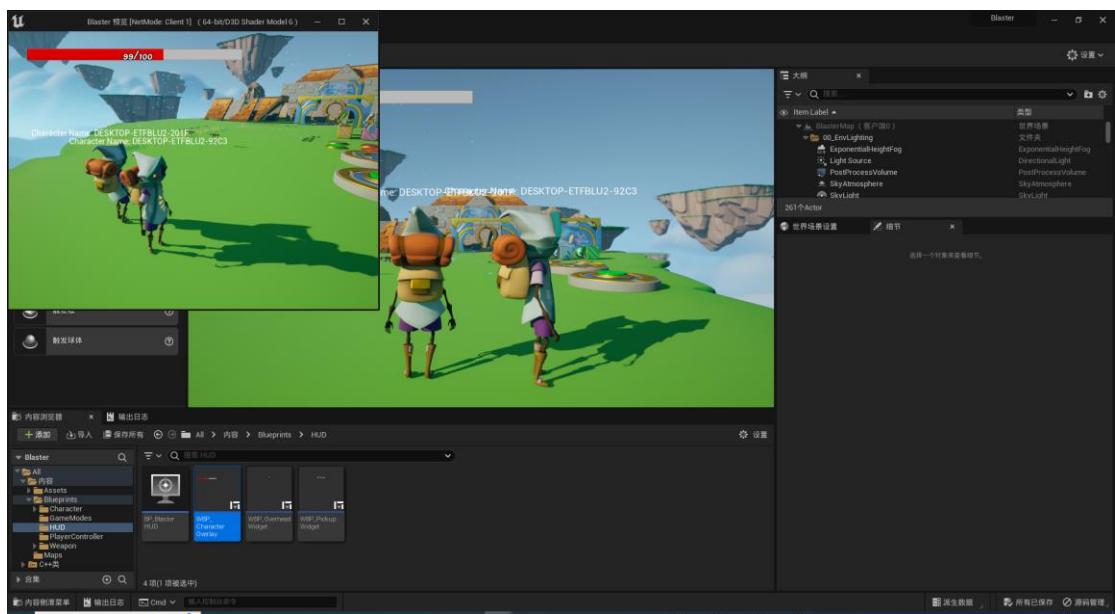
```
void ABlasterHUD::BeginPlay()
{
    Super::BeginPlay();

    AddCharacterOverlay();
}

void ABlasterHUD::AddCharacterOverlay()
{
    APlayerController* PlayerController = GetOwningPlayerController();
    if (PlayerController && CharacterOverlayClass)
    {
        CharacterOverlay = CreateWidget<UCharacterOverlay>(PlayerController,
CharacterOverlayClass);
        CharacterOverlay->AddToViewport();
    }
}
```



编译工程，在 BlasterCharacter 蓝图中设置变量。



085 在 HUD 中设置生命值

在 PlayerController 中实现如下代码：

```
#include "BlasterPlayerController.h"
#include "Blaster/HUD/BlasterHUD.h"
#include "Blaster/HUD/CharacterOverlay.h"
#include "Components/ProgressBar.h"
#include "Components/TextBlock.h"

void ABlasterPlayerController::BeginPlay()
{
    Super::BeginPlay();

    BlasterHUD = Cast<ABlasterHUD>(GetHUD());
}

void ABlasterPlayerController::SetHUDHealth(float Health, float MaxHealth)
{
    BlasterHUD = BlasterHUD == nullptr ? BlasterHUD = Cast<ABlasterHUD>(GetHUD()) : BlasterHUD;

    bool bHUDValid = BlasterHUD &&
        BlasterHUD->CharacterOverlay &&
        BlasterHUD->CharacterOverlay->HealthBar &&
        BlasterHUD->CharacterOverlay->HealthText;
    if (bHUDValid)
    {
        const float HealthPercent = Health / MaxHealth;
        BlasterHUD->CharacterOverlay->HealthBar->SetPercent(HealthPercent);

        FString HealthText = FString::Printf(TEXT("%d/%d"), FMath::CeilToInt(Health),
        FMath::CeilToInt(MaxHealth));

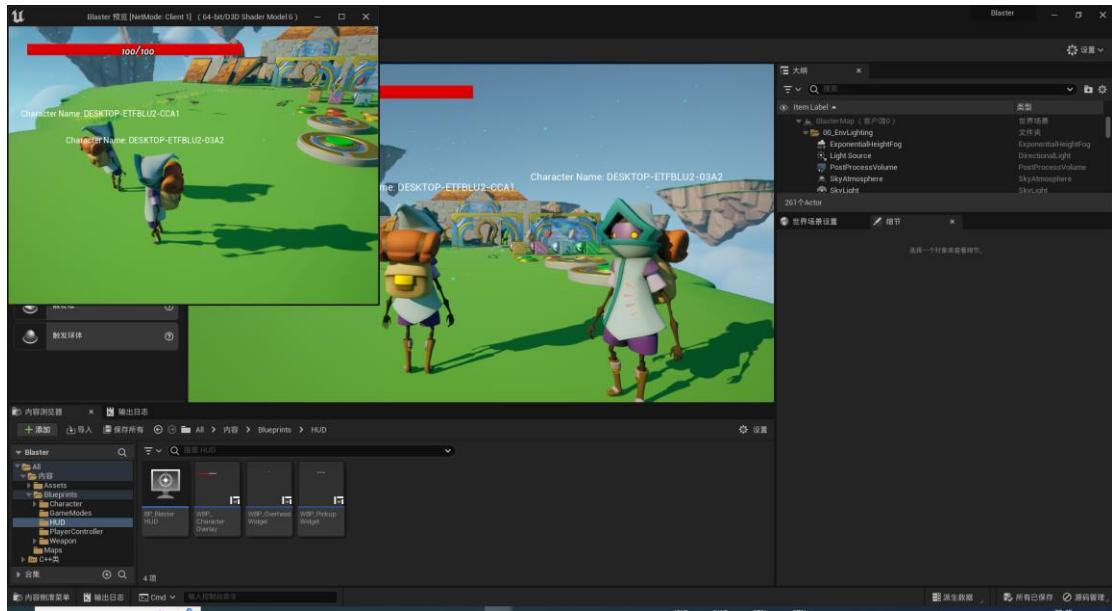
        BlasterHUD->CharacterOverlay->HealthText->SetText(FText::FromString(HealthText));
    }
}
```

在角色.cpp 中调用 setHUD 的函数。

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

    BlasterPlayerController = Cast<ABlasterPlayerController>(Controller);
    if (BlasterPlayerController)
```

```
{
    BlasterPlayerController->SetHUDHealth(Health, MaxHealth);
}
}
```



086 伤害



创建一个 Projectile 的子类

给 Projectile 添加一个成员 Damage

重写 ProjectileBullet 中的 OnHit 函数，由于之前已经有了宏标记 UFUNCTION，这里不需要再添加该宏标记。

```
void AProjectileBullet::OnHit(UPrimitiveComponent* HitComp, AActor* OtherActor,
UPrimitiveComponent* OtherComp, FVector NormalImpulse, const FHitResult& Hit)
{
    ACharacter* OwnerCharacter = Cast<ACharacter>(GetOwner());
    if (OwnerCharacter)
    {
        AController* OwnerController = OwnerCharacter->Controller;
        if (OwnerController)
        {
            UGameplayStatics::ApplyDamage(OtherActor, Damage, OwnerController, this,
UDamageType::StaticClass());
        }
    }

    Super::OnHit(HitComp, OtherActor, OtherComp, NormalImpulse, Hit);
}
```

为了实现伤害事件，我们需要在角色类中添加一个伤害事件的回调函数：

```
void ReceiveDamage(AActor* DamagedActor, float Damage, const UDamageType* DamageType,
class AController* InstigatorController, AActor* DamageCauser);
```

修改角色的 BeginPlay：

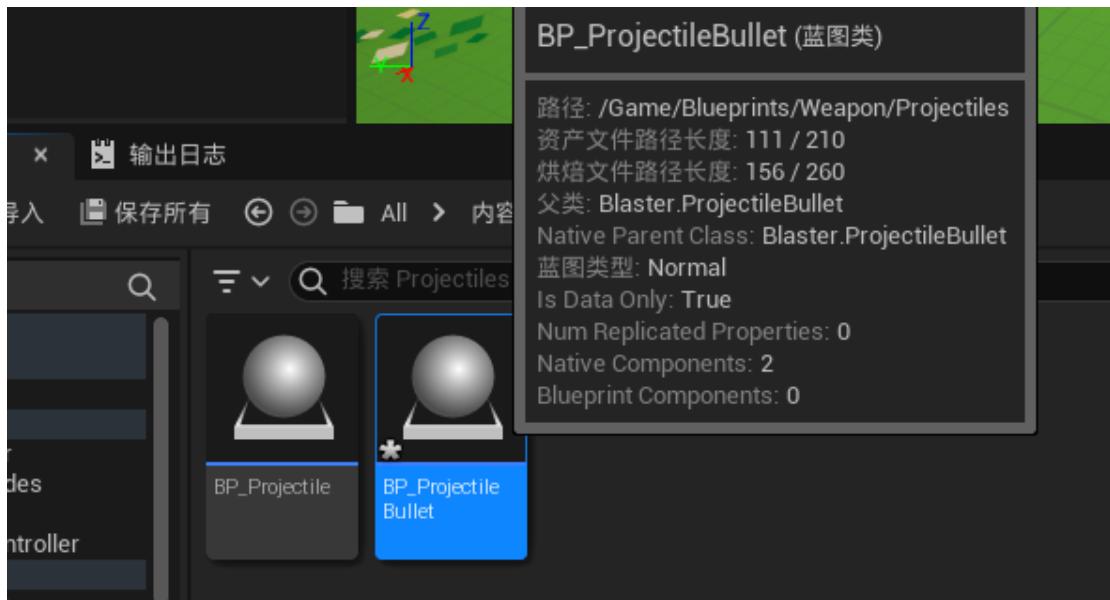
```
void ABlasterCharacter::BeginPlay()
{
    Super::BeginPlay();

    BlasterPlayerController = Cast<ABlasterPlayerController>(Controller);
    if (BlasterPlayerController)
    {
        BlasterPlayerController->SetHUDHealth(Health, MaxHealth);
    }
    if (HasAuthority)
    {
        OnTakeAnyDamage.AddDynamic(this, &ABlasterCharacter::ReceiveDamage);
    }
}
```

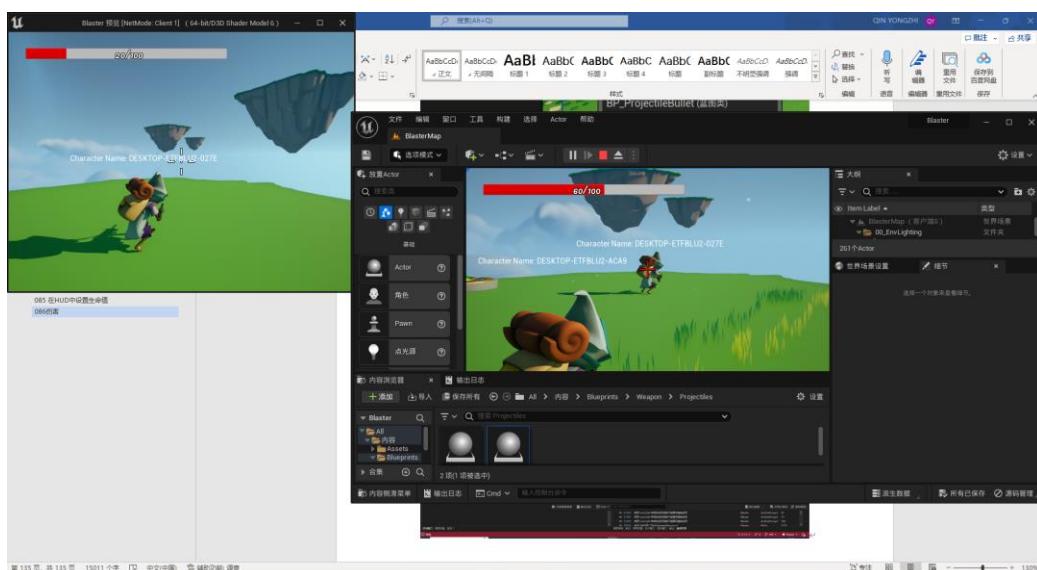
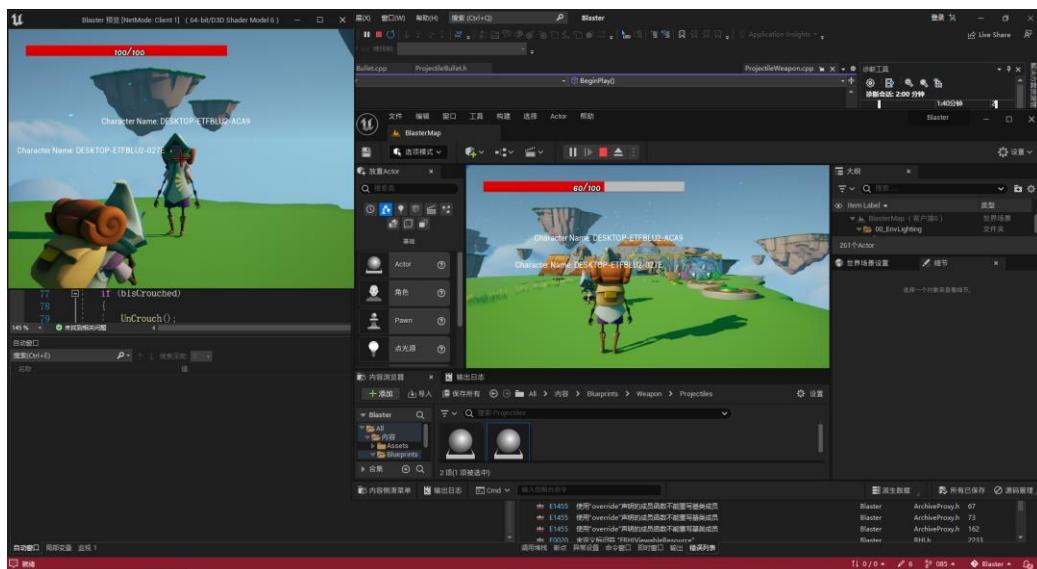
由于我们可以在收到伤害的时候利用 REP 通知播放受击的蒙太奇，所以我们可以移除之前的 RPC 多播函数，REP 通知的效率要高于 RPC。

```
void ABlasterCharacter::ReceiveDamage(AActor* DamagedActor, float Damage, const
UDamageType* DamageType, AController* InstigatorController, AActor* DamageCauser)
{
    Health = FMath::Clamp(Health - Damage, 0.0f, MaxHealth);
    UpdateHUDHealth();
    PlayHitReactMontage();
}

void ABlasterCharacter::OnRep_Health()
{
    UpdateHUDHealth();
    PlayHitReactMontage();
}
```



创建一个新的 BP_ProjectileBullet，设置速度，效果和声音。

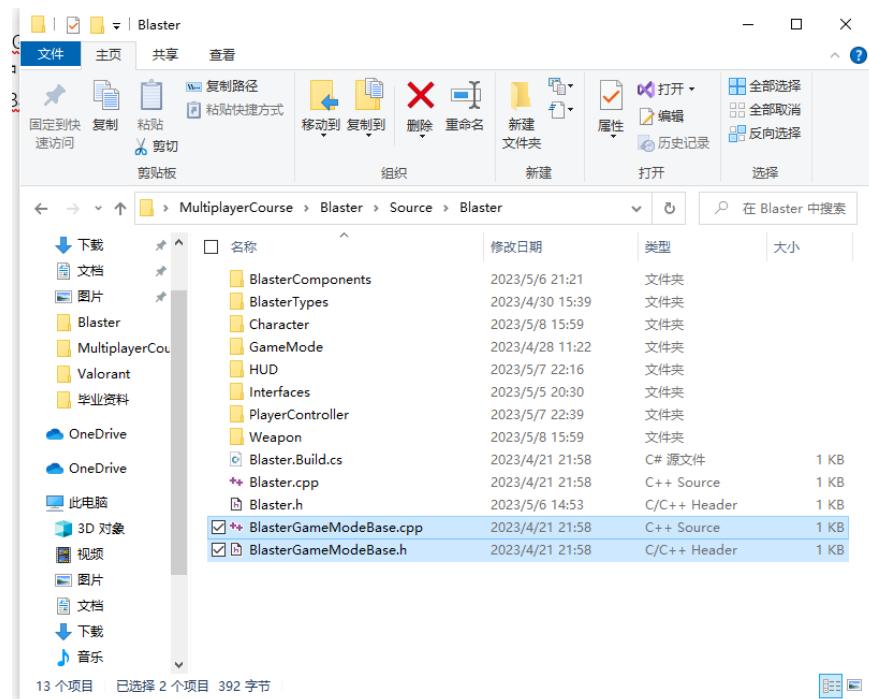


087 Blaster 游戏模式

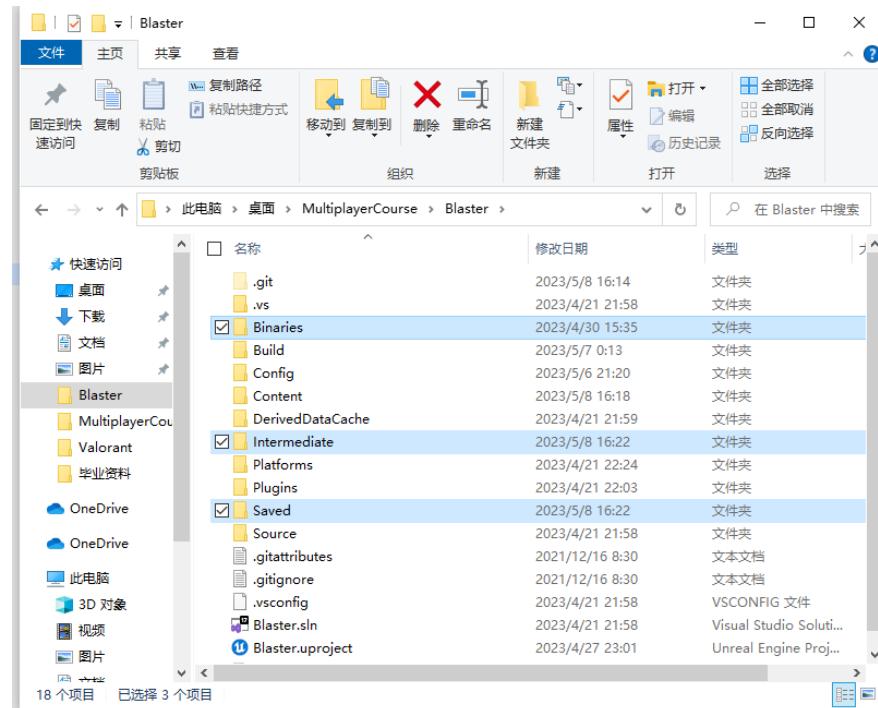
删除 C++ 类：

我们现在打算删除掉 BlasterGameModeBase 这个类。

1. 关闭 VS，打开 Blaster 中的 Source 文件夹
2. 选择 BlasterGameModeBase 的两个文件，进行删除。



3. 删 除三个文件夹



创建一个我们自己的 GameMode 类，注意不是 GameModeBase。

在角色中添加一个用于淘汰的函数：

```
void ABlasterCharacter::Elim()
```

更新角色的受到伤害的函数：

```
void ABlasterCharacter::ReceiveDamage(AActor* DamagedActor, float Damage, const UDamageType* DamageType, AController* InstigatorController, AActor* DamageCauser)
{
    Health = FMath::Clamp(Health - Damage, 0.0f, MaxHealth);
    UpdateHUDHealth();
    PlayHitReactMontage();

    if (Health == 0.0f)
    {
        ABlasterGameMode* BlasterGameMode =
GetWorld() -> GetAuthGameMode<ABlasterGameMode>();
        if (BlasterGameMode)
        {
            BlasterPlayerController = BlasterPlayerController == nullptr ?
Cast<ABlasterPlayerController>(Controller) : BlasterPlayerController;
            ABlasterPlayerController* AttackerController =
Cast<ABlasterPlayerController>(InstigatorController);
            BlasterGameMode->PlayerEliminated(this, BlasterPlayerController,
AttackerController);
        }
    }
}
```

重新创建一个 BP_BlastGameMode，并设置地图的游戏模式重载。

088 淘汰动画

创建一个蒙太奇，并添加到角色类中

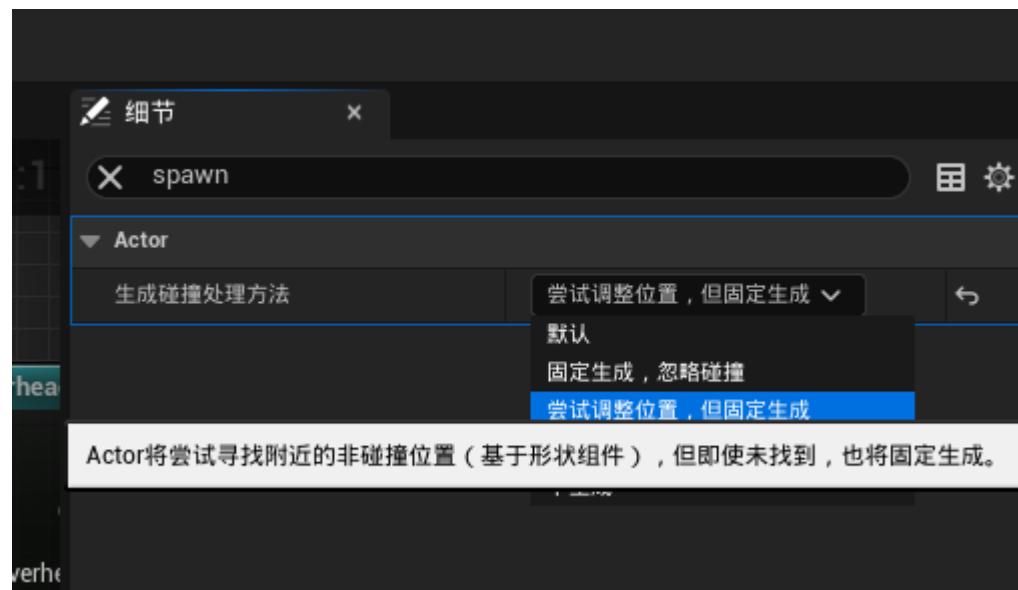
```
UPROPERTY(EditAnywhere, Category = Combat)
UAnimMontage* ElimMontage;
```

在 Elim 函数中播放蒙太奇动画，在 GameMode 的淘汰函数中，调用 Elim:

我们发现 Elim 的蒙太奇只在服务端能够调用，所以我们考虑用一个多播 RPC 来实现在所有端的淘汰动画播放。

089 重生

设置角色的生成碰撞处理方法：



C++ 中的设置：

```
SpawnCollisionHandlingMethod = ESpawnActorCollisionHandlingMethod::AdjustIfPossibleButAlwaysSpawn;
```

在角色中添加一些淘汰相关的变量：

```
FTimerHandle ElimTimer;
```

```
UPROPERTY(EditDefaultsOnly)
float ElimDelay = 3.f;
```

```
void ElimTiometerFinished();
```

添加一个函数 Elim：

```
void ABlasterCharacter::Elim()
{
```

```

        MulticastElim();
        GetWorldTimerManager().SetTimer(
            ElimTimer,
            this,
            &ABlasterCharacter::ElimTiomerFinished,
            ElimDelay
        );
    }
}

```

```
void ABlasterCharacter::ElimTiomerFinished()
```

```
{
    ABlasterGameMode* BlasterGameMode =
    GetWorld()->GetAuthGameMode<ABlasterGameMode>();
    if (BlasterGameMode)
    {
        BlasterGameMode->RequestRespawn(this, Controller);
    }
}
```

他会调用 GameMode 中的重生请求，重生请求中先要解除 Character 的控制，然后销毁 Character，然后再随机选择一个重生点重生角色：

```

void ABlasterGameMode::RequestRespawn(ACharacter* ElimmedCharacter, AController*
ElimmedController)
{
    if (ElimmedCharacter)
    {
        ElimmedCharacter->Reset();
        ElimmedCharacter->Destroy();
    }
    if (ElimmedController)
    {
        TArray<AActor*> PlayerStarts;
        UGameplayStatics::GetAllActorsOfClass(this, APlayerStart::StaticClass(),
PlayerStarts);
        int32 Selection = FMath::RandRange(0, PlayerStarts.Num() - 1);
        RestartPlayerAtPlayerStart(ElimmedController, PlayerStarts[Selection]);
    }
}

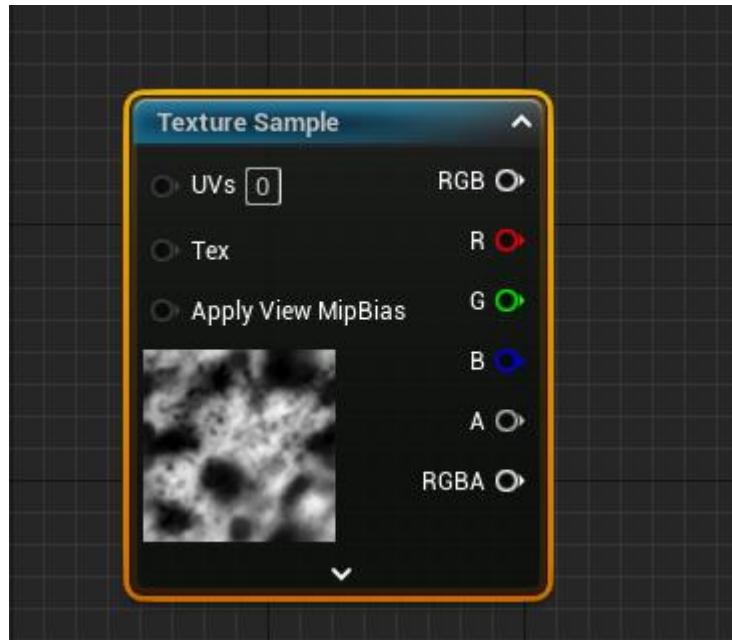
```

选做

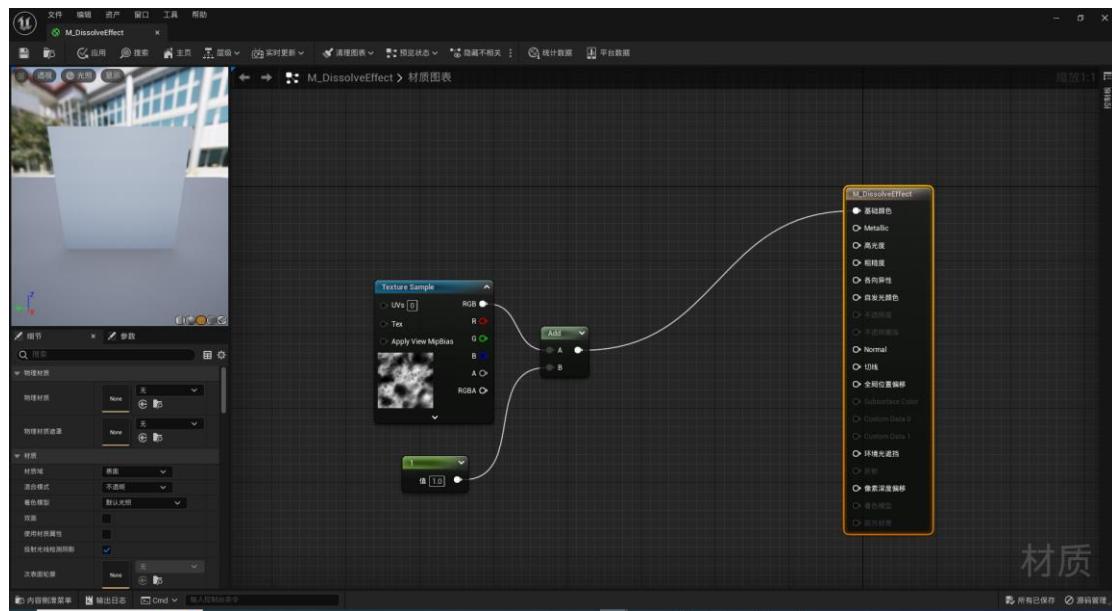
让角色重生再离其他角色最远的重生点。

090 溶解材料

创建一个新的材质，命名为 M_DissolveEffect。

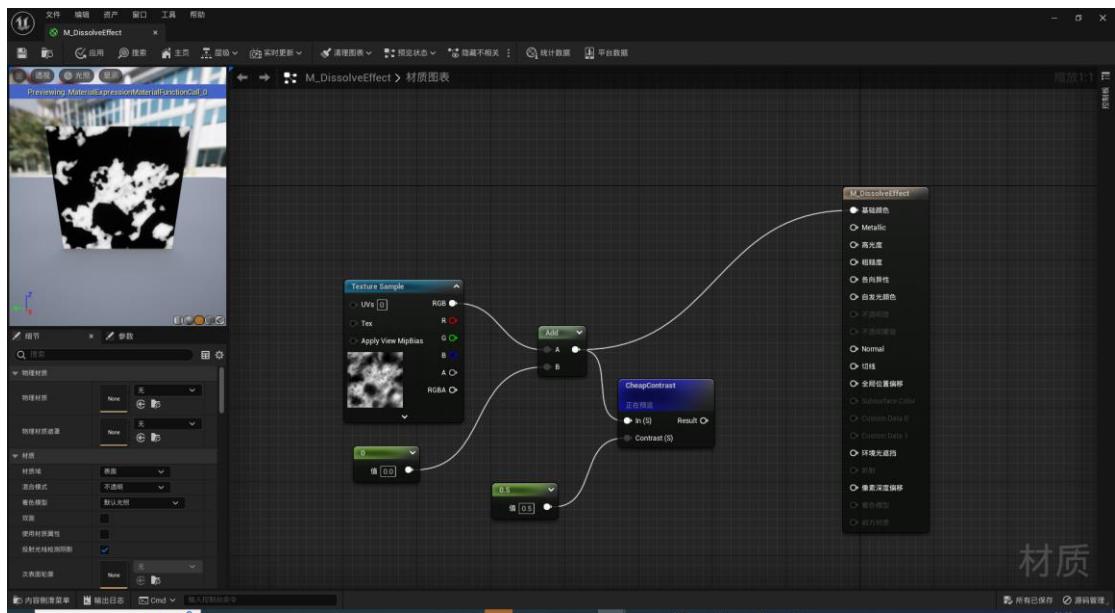


创建一个纹理采样，并且将 texture 设置为引擎自带噪声 T_Noise01。

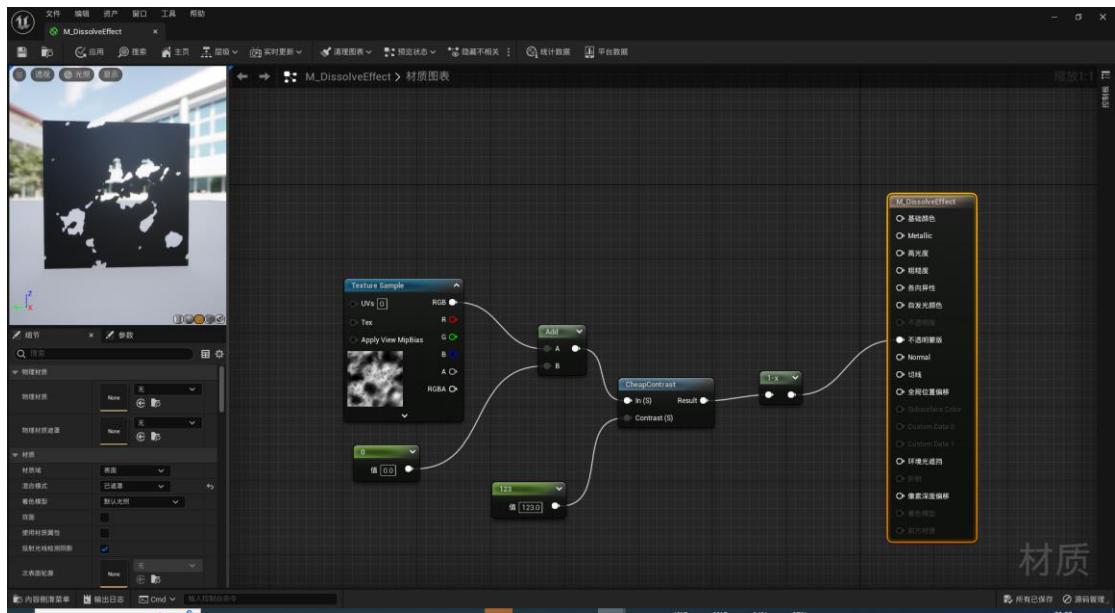


给噪声全都加上一个值，这样的话所有的值都会大于等于 1，纹理会被 clamp 在 0-1 之间，所以就会全变成 1。

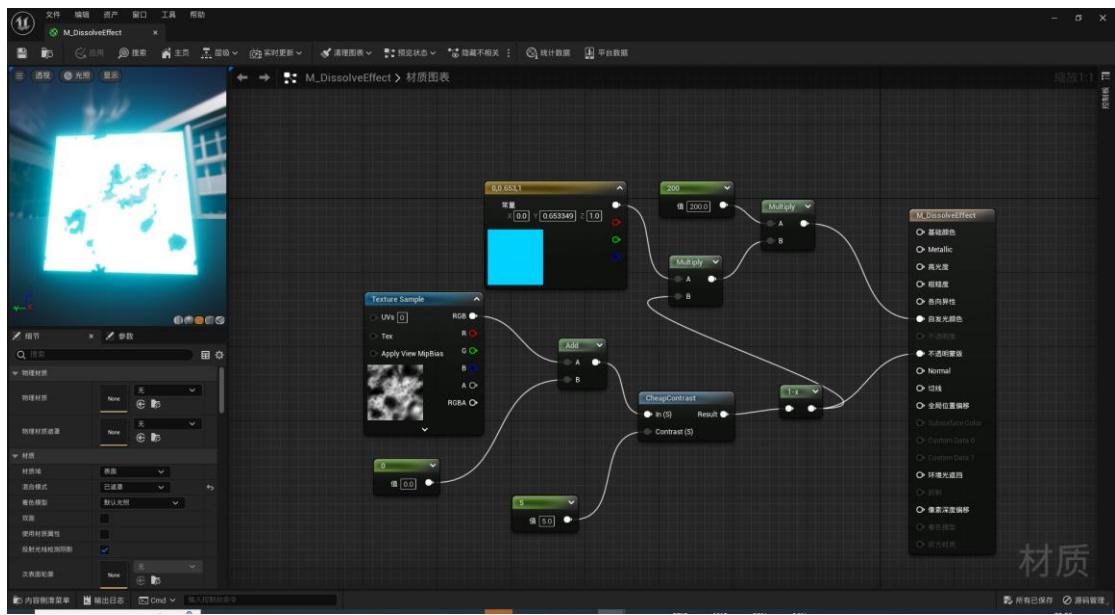
此外这个噪声纹理边界非常的模糊，我们可以用一个节点来锐化他的边界：



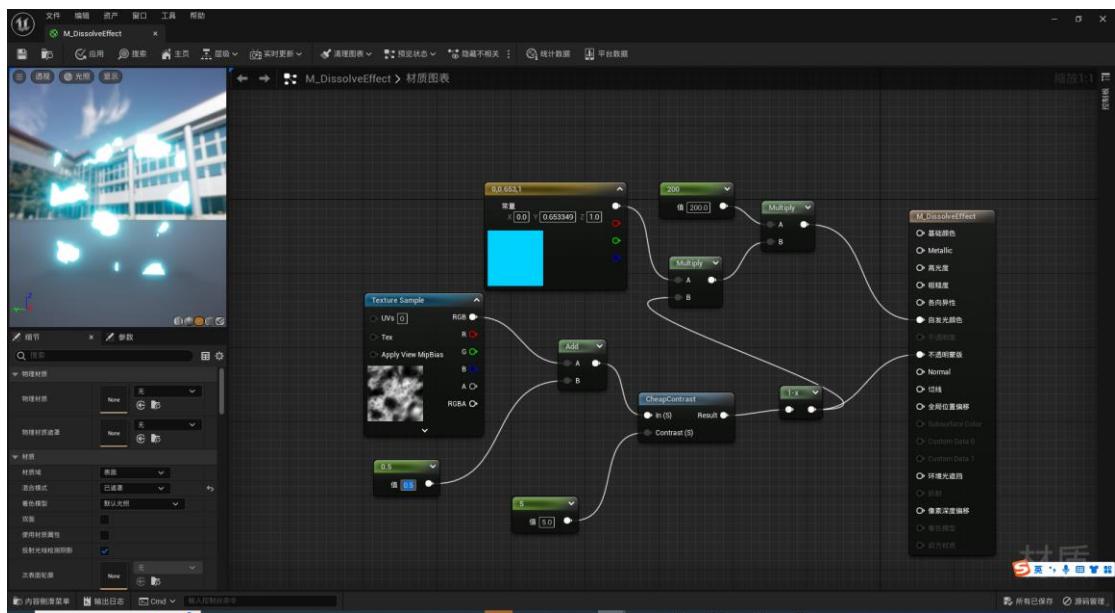
设置一个比较大的值，从而得到一个锐利的边界，之后再将材质设置为已遮罩，从而得到如下效果：



我们可以把它设置为自发光材料：

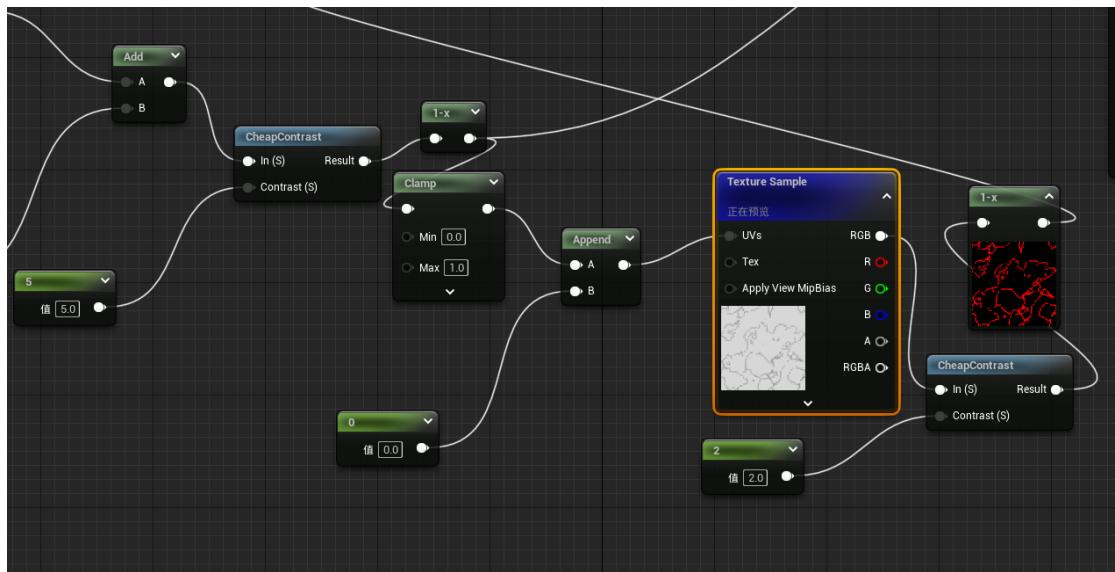


当我们改变左下角的值时，整个材质也会发生变化：



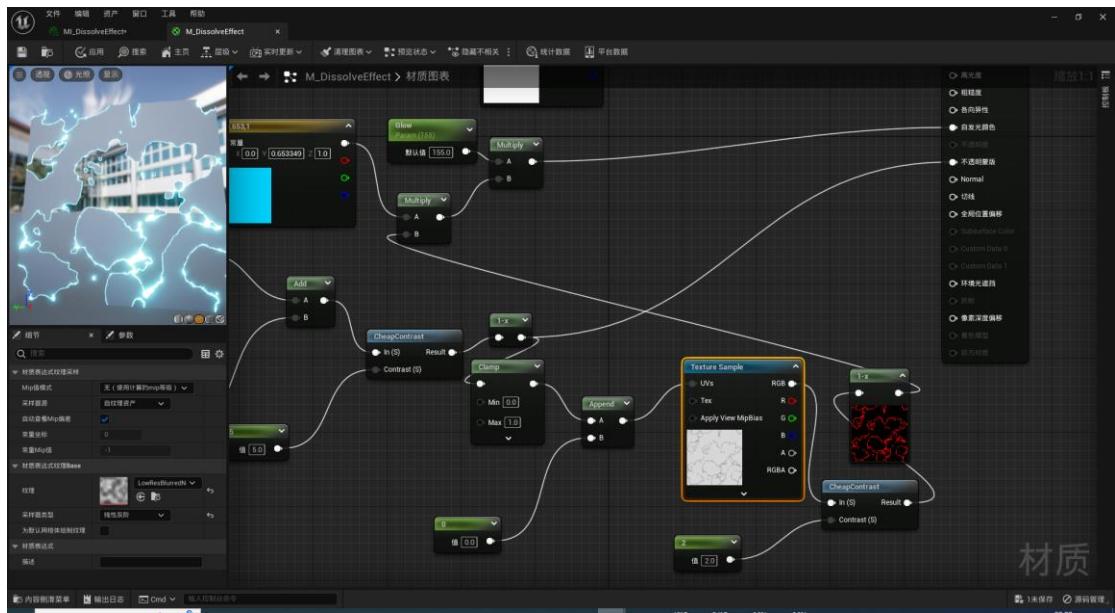
我们可以把这个数字提升为变量，命名为 Dissolve。发光的强度也可以提升为变量 Glow。

保存材质并创建材质实例。

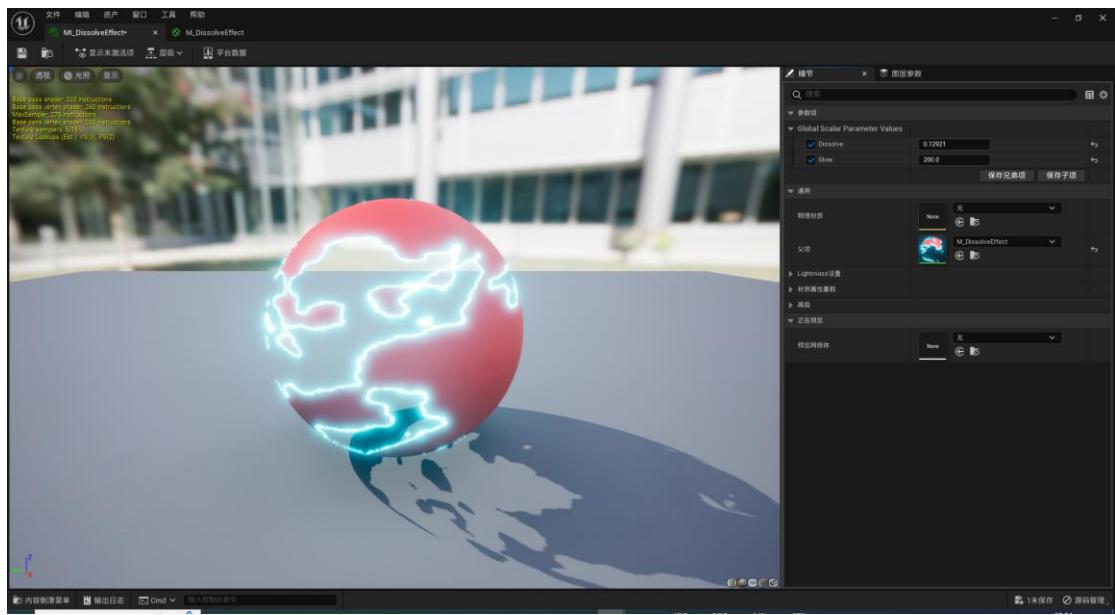


通过如下连接可以得到噪声的边界。原因应该是由于选取的第二个噪声在 UV(0,0)和(1,0)两个点之前时，均为白色，所以在过渡区域，也就是边界才会有值出现。

然后我们锐化这个结果，得到一个边界，让边界发光，可以得到以下效果：

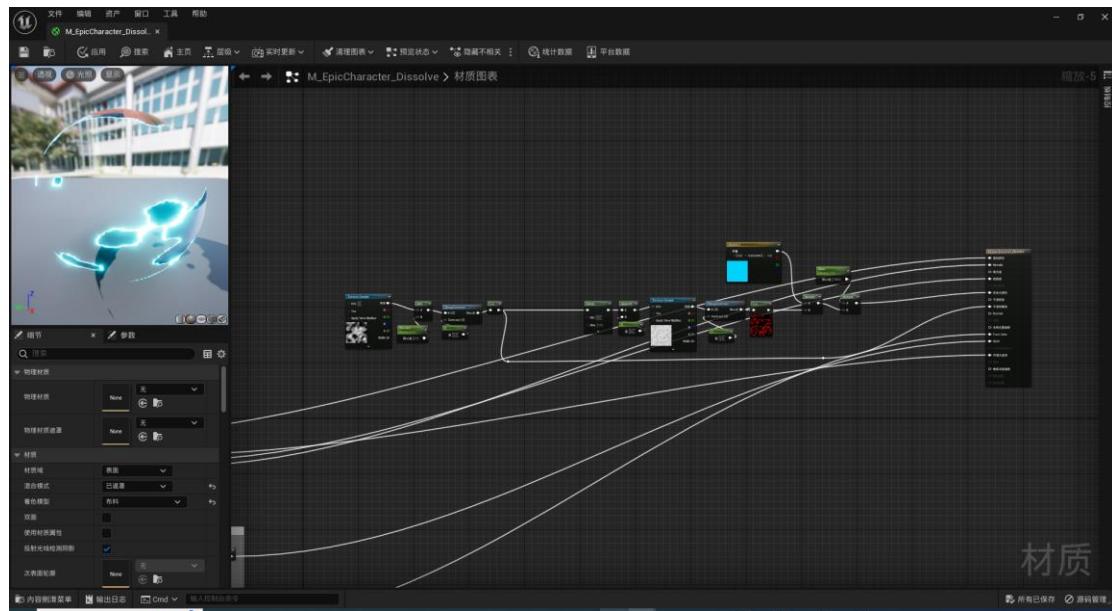


来到材质实例，拖动 Dissolve，可以看到不错的效果：

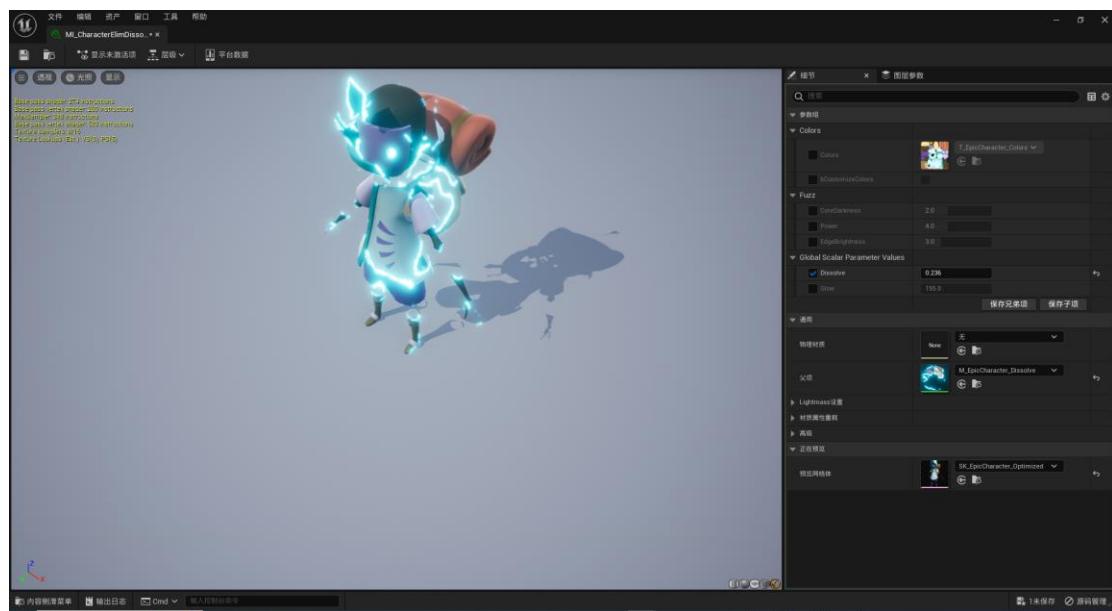


091 溶解角色

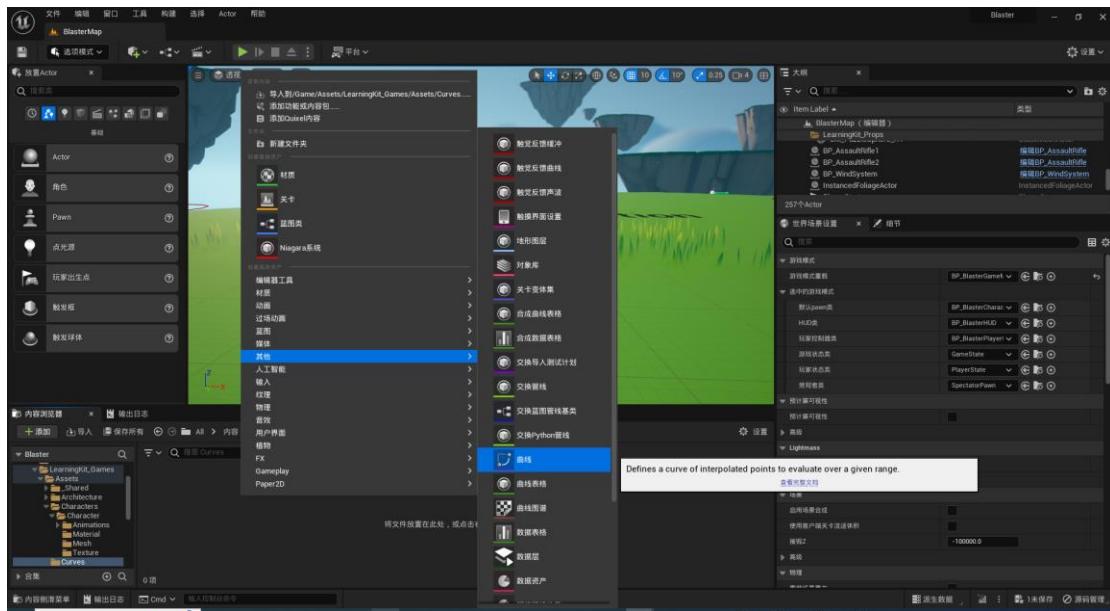
找到我们角色中的材质，添加我们之前的节点：



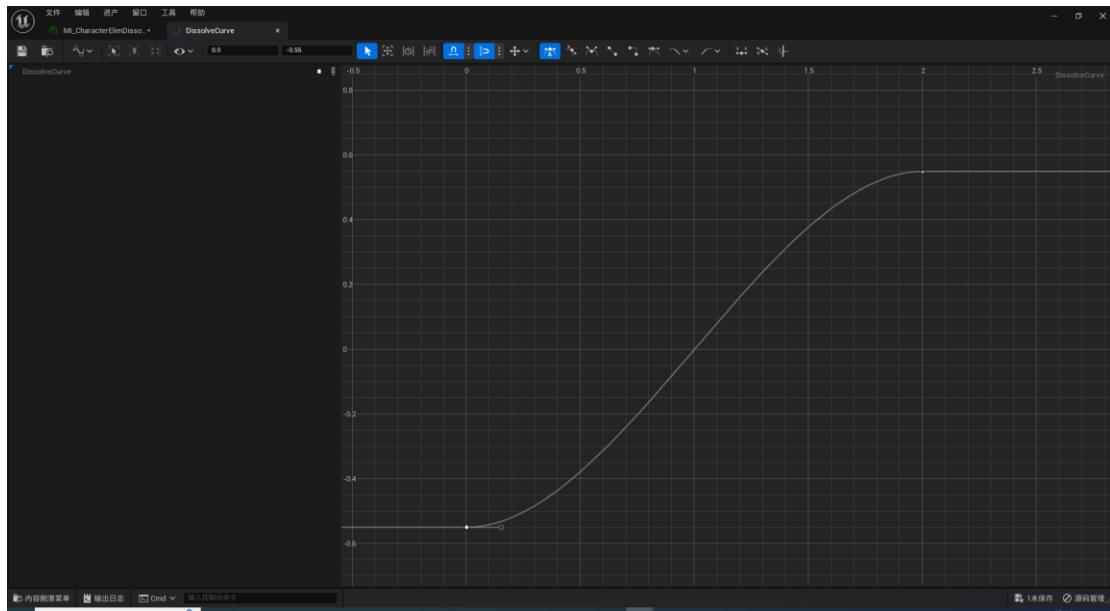
创建一个材质实例，可以看到：



092 溶解曲线



创建一个曲线，创建一个 FloatCurve:



在角色类中，添加如下变量：

```
/**  
 * Dissolve effect  
 */  
UPROPERTY(VisibleAnywhere)  
UTimelineComponent* DissolveTimeline;  
FOnTimelineFloat DissolveTrack;  
  
UPROPERTY(EditAnywhere)  
UCurveFloat* DissolveCurve;
```

```

UFUNCTION()
void UpdateDissolveMaterial(float DissolveValue);
void StartDissolve();

// Dynamic instace that we can change at runtime
UPROPERTY(VisibleAnywhere, Category = Elim)
UMaterialInstanceDynamic* DynamicDissolveMaterialInstance;

// Material instace set on the Blueprint, used with the dynamic material instance
UPROPERTY(EditAnywhere, Category = Elim)
UMaterialInstance* DissolveMaterialInstance;

UPROPERTY(VisibleAnywhere)
UTimelineComponent* DissolveTimeline;
FOnTimelineFloat DissolveTrack;

16    /** Signature of function to handle timeline float track */
17    DECLARE_DYNAMIC_DELEGATE_OneParam( FOnTimelineFloat, float, Output );
18    /** Signature of function to handle timeline vector track */
19    DECLARE_DYNAMIC_DELEGATE_OneParam( FOnTimelineVector, FVector, Output );

```

可以看到 FOnTimelineFloat 是一个单参数的动态委托。

我们为他绑定回调函数 UpdateDissolveMaterial，由于我们需要在淘汰的多播函数中调用我们的材质溶解，所以实现代码如下：

```

void ABlasterCharacter::UpdateDissolveMaterial(float DissolveValue)
{
    if (DynamicDissolveMaterialInstance)
    {
        DynamicDissolveMaterialInstance->SetScalarParameterValue(TEXT("Dissolve"),
DissolveValue);
    }
}

void ABlasterCharacter::StartDissolve()
{
    DissolveTrack.BindDynamic(this, &ABlasterCharacter::UpdateDissolveMaterial);
    if (DissolveCurve && DissolveTimeline)
    {
        DissolveTimeline->AddInterpFloat(DissolveCurve, DissolveTrack);
        DissolveTimeline->Play();
    }
}

void ABlasterCharacter::MulticastElim_Implementation()
{
    bElimmed = true;
}

```

```

PlayElimMontage();

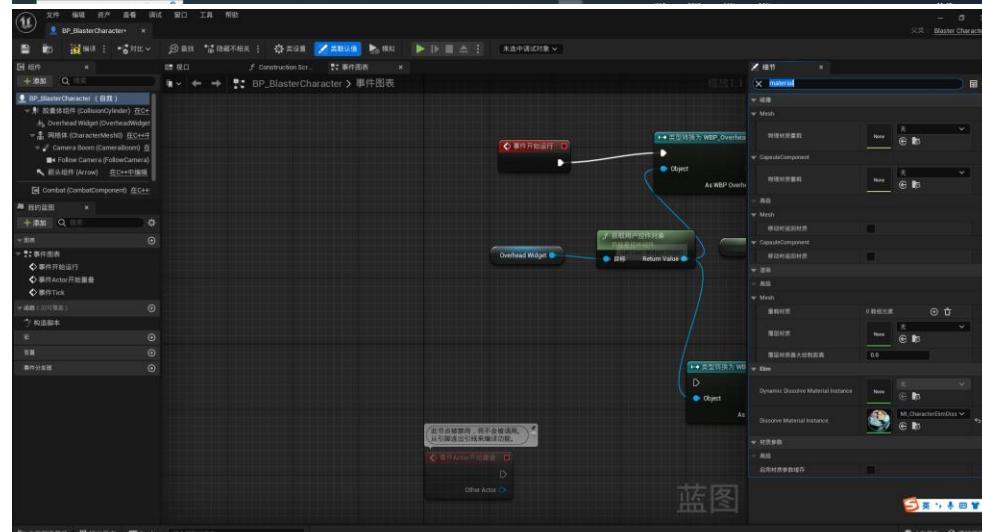
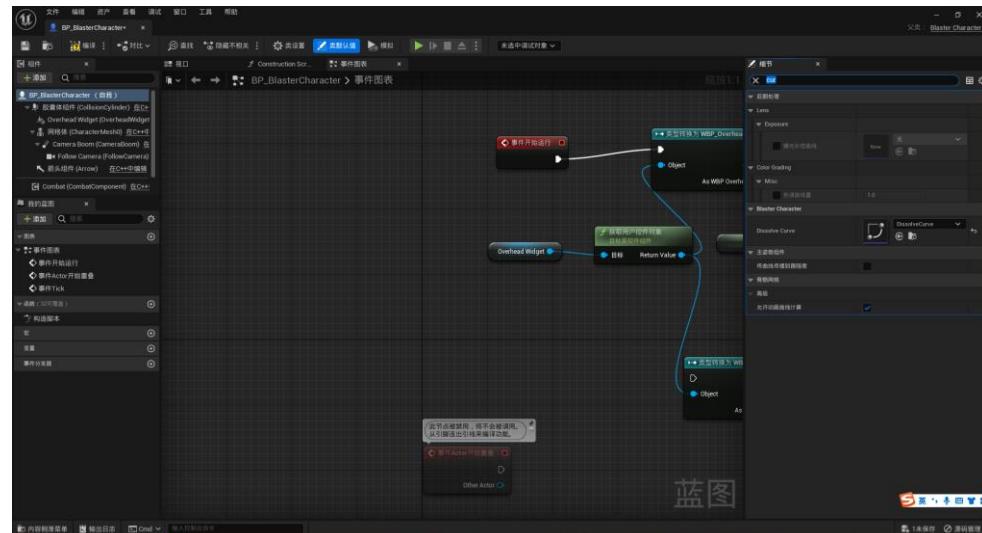
if (DissolveMaterialInstance)
{
    DynamicDissolveMaterialInstance =
        UMaterialInstanceDynamic::Create(DissolveMaterialInstance, this);
    GetMesh()->SetMaterial(0, DynamicDissolveMaterialInstance);
    DynamicDissolveMaterialInstance->SetScalarParameterValue(TEXT("Dissolve"), -0.55f);
    DynamicDissolveMaterialInstance->SetScalarParameterValue(TEXT("Glow"), 200.f);
}

StartDissolve();
}

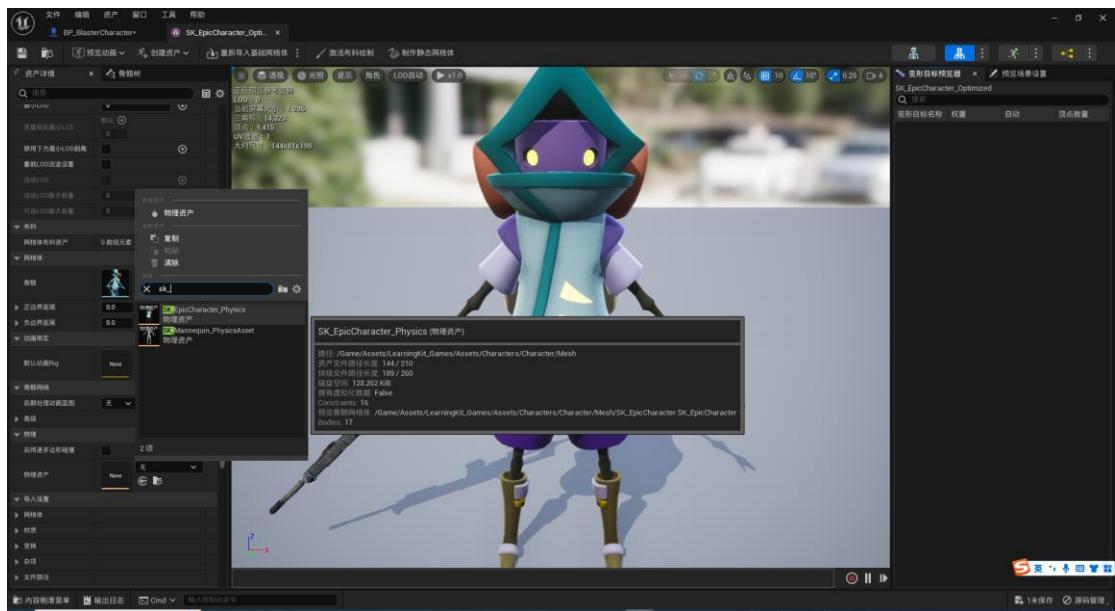
```

值得注意的是，我们需要在构造函数中设置我们的组件变量：

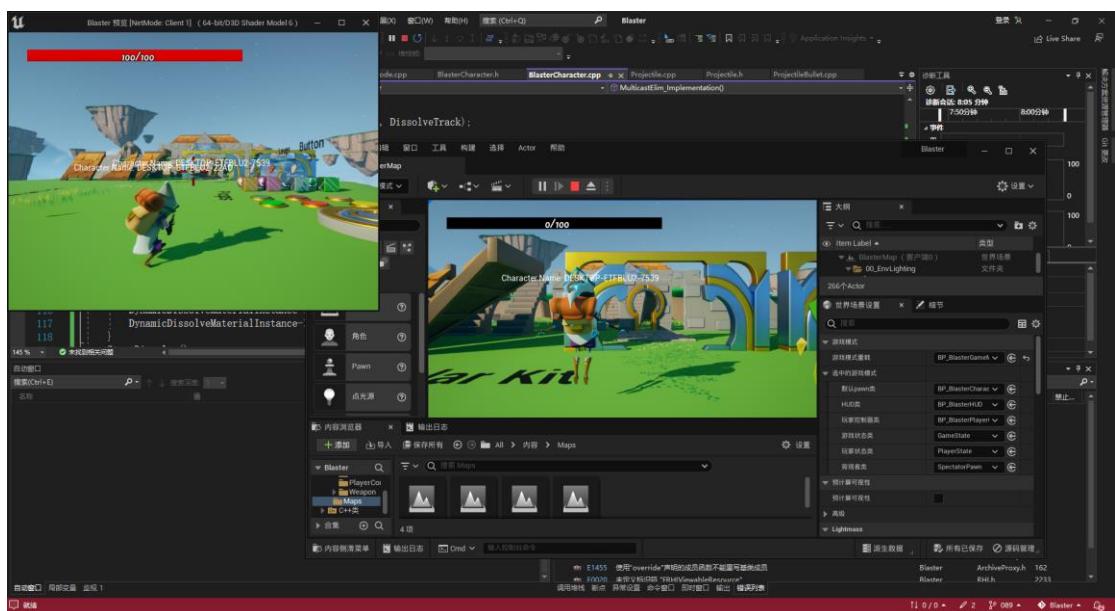
```
DissolveTimeline = CreateDefaultSubobject<UTimelineComponent>(TEXT("DissolveTimeline"));
```



设置角色蓝图的属性中的曲线，以及材质。



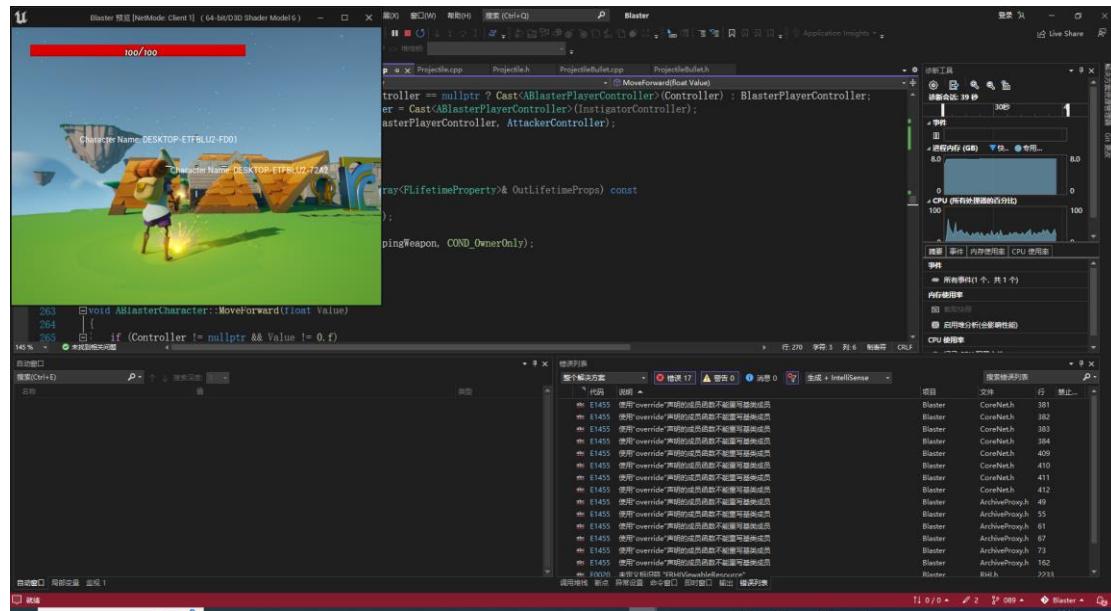
还要对我们的 optimize 角色设置物理资产，不然会出现无法命中的问题。



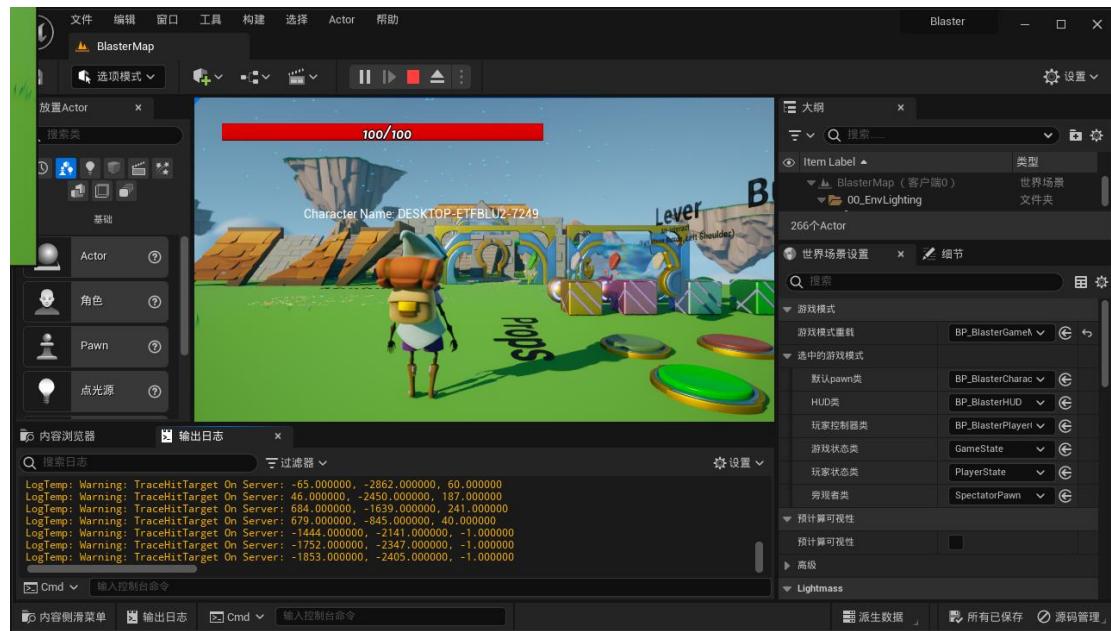
ISSUE

射击起点错误

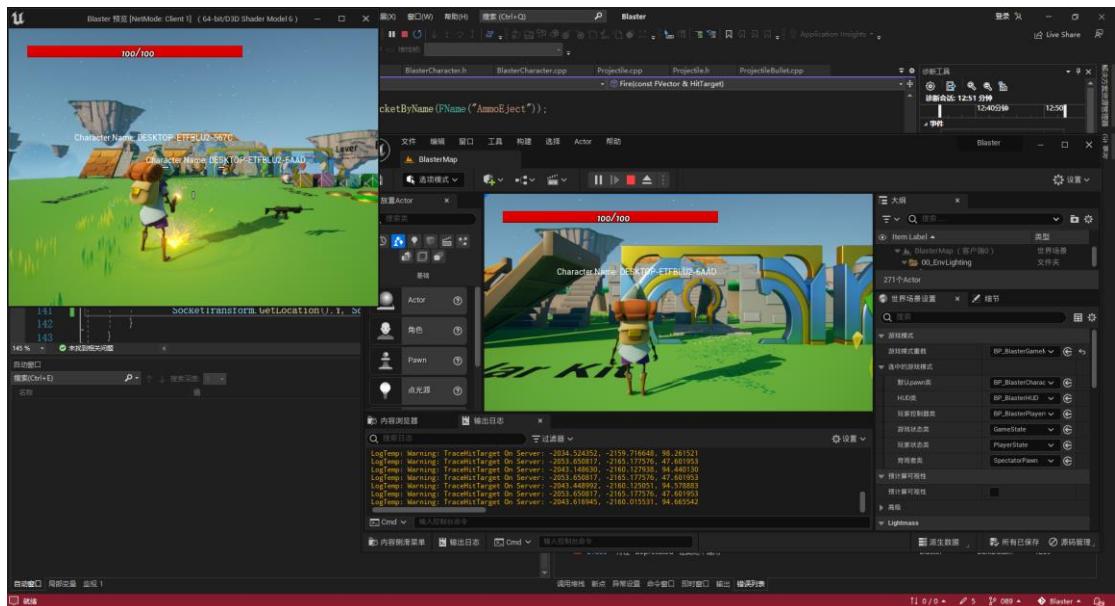
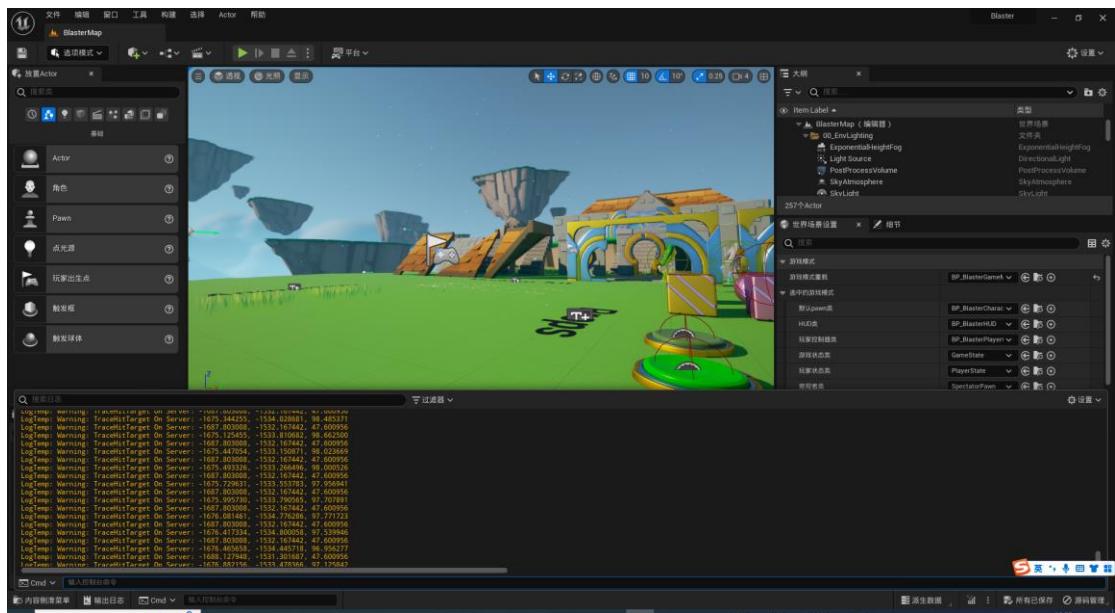
射击的时候会出现好似的子弹垂直向下的情况。



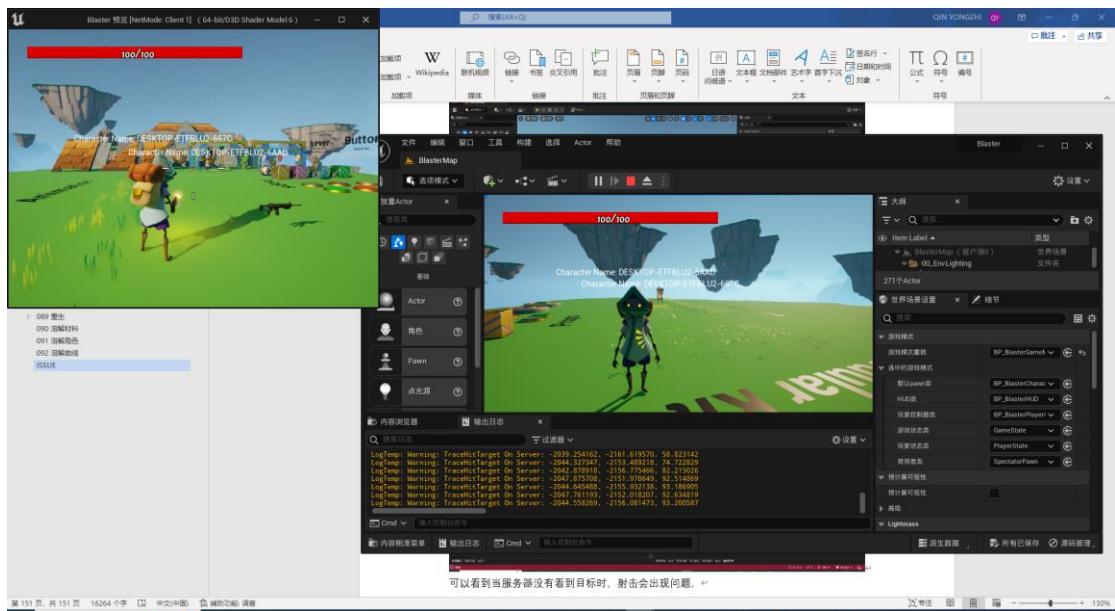
可以看到服务端接收到的瞄准方向没有问题：



经过调查可以发现，子弹生成的位置出现了问题，子弹生成在了地下的位置：



可以看到当服务器没有看到目标时，射击会出现问题，而当服务器看向目标之后，或者目标来到服务器附近时，目标就能正确的射击。



猜测原因可能是因为 UE 动画蓝图的更新在服务器不关注目标的时候并没有正确的更新，所以目标甚至是以 A-pose 在持枪射击，所以不能够正确的击中目标。

之后如果教程没有纠正，可以尝试在调用 RPC 生成子弹的时候传入 Socket 的位置，以确保正确生成子弹。（见 126）

服务端血量不能正确更新

服务端重生之后 HUD 显示的生命值为 0。

093 在淘汰时禁用移动

我们希望在角色淘汰的时候移动以及碰撞都被禁用。

我们可以使用如下语句在淘汰的多播函数中：

```
// Disable Character movement
GetCharacterMovement() -> DisableMovement();
GetCharacterMovement() -> StopMovementImmediately();
if (BlasterPlayerController)
{
    DisableInput(BlasterPlayerController);
}
// Disable Collision
GetCapsuleComponent() -> SetCollisionEnabled(ECollisionEnabled::NoCollision);
GetMesh() -> SetCollisionEnabled(ECollisionEnabled::NoCollision);
```

此外，我们希望淘汰的时候武器会掉落下来，我们现在 Weapon 中添加代码：

```
void AWeapon::Dropped()
{
    SetWeaponState(EWeaponState::EWS_Dropped);
    FDetachmentTransformRules DetachRules(EDetachmentRule::KeepWorld, true);
    WeaponMesh->DetachFromComponent(DetachRules);
    SetOwner(nullptr);
}
```

由于我们之前没有设置 WeaponState 变为 Dropped 时该做的事，我们现在进行设置：

```
void AWeapon::SetWeaponState(EWeaponState State)
{
    WeaponState = State;
    switch (WeaponState)
    {
        case EWeaponState::EWS_Equipped:
            ShowPickupWidget(false);
            AreaSphere->SetCollisionEnabled(ECollisionEnabled::NoCollision);
            WeaponMesh->SetSimulatePhysics(false);
            WeaponMesh->SetEnableGravity(false);
            WeaponMesh->SetCollisionEnabled(ECollisionEnabled::NoCollision);
            break;
        case EWeaponState::EWS_Dropped:
            if (HasAuthority())
            {
                AreaSphere->SetCollisionEnabled(ECollisionEnabled::QueryOnly);
            }
            WeaponMesh->SetSimulatePhysics(true);
            WeaponMesh->SetEnableGravity(true);
            WeaponMesh->SetCollisionEnabled(ECollisionEnabled::QueryAndPhysics);
```

```

        break;
    }
}

void AWeapon::OnRep_WeaponState()
{
    switch (WeaponState)
    {
    case EWeaponState::EWS_Equipped:
        ShowPickupWidget(false);
        WeaponMesh->SetSimulatePhysics(false);
        WeaponMesh->SetEnableGravity(false);
        WeaponMesh->SetCollisionEnabled(ECollisionEnabled::NoCollision);
        break;
    case EWeaponState::EWS_Dropped:
        WeaponMesh->SetSimulatePhysics(true);
        WeaponMesh->SetEnableGravity(true);
        WeaponMesh->SetCollisionEnabled(ECollisionEnabled::QueryAndPhysics);
        break;
    }
}

```

回到 Combat 中，我们之前直接在服务端将武器绑定到了 socket 上，但是由于我们现在设置了物理模拟，在关闭物理模拟之前，不能进行 Attach 操作，而由于网络的原因，可能造成我们的 RPC 执行顺序改变，所以我们在客户端也要进行一次绑定。

```

void UCombatComponent::EquipWeapon(AWeapon* WeaponToEquip)
{
    if (Character == nullptr || WeaponToEquip == nullptr)
    {
        return;
    }

    EquippedWeapon = WeaponToEquip;
    EquippedWeapon->SetWeaponState(EWeaponState::EWS_Equipped);
    const USkeletalMeshSocket* HandSocket =
Character->GetMesh()->GetSocketByName(FName("RightHandSocket"));
    if (HandSocket)
    {
        HandSocket->AttachActor(EquippedWeapon, Character->GetMesh());
    }
    EquippedWeapon->SetOwner(Character);
    Character->GetCharacterMovement()->bOrientRotationToMovement = false;
    Character->bUseControllerRotationYaw = true;
}

```

```

void UCombatComponent::OnRep_EquippedWeapon()
{
    if (EquippedWeapon && Character)
    {
        EquippedWeapon->SetWeaponState(EWeaponState::EWS_Equipped);
        const USkeletalMeshSocket* HandSocket =
Character->GetMesh()->GetSocketByName(FName("RightHandSocket"));
        if (HandSocket)
        {
            HandSocket->AttachActor(EquippedWeapon, Character->GetMesh());
        }
        Character->GetCharacterMovement()->bOrientRotationToMovement = false;
        Character->bUseControllerRotationYaw = true;
    }
}

void ABlasterCharacter::EquipButtonPressed()
{
    if (Combat)
    {
        if (HasAuthority())
        {
            Combat->EquipWeapon(OverlappingWeapon);
        }
        else
        {
            ServerEquipButtonPressed();
        }
    }
}

void ABlasterCharacter::ServerEquipButtonPressed_Implementation()
{
    if (Combat)
    {
        Combat->EquipWeapon(OverlappingWeapon);
    }
}

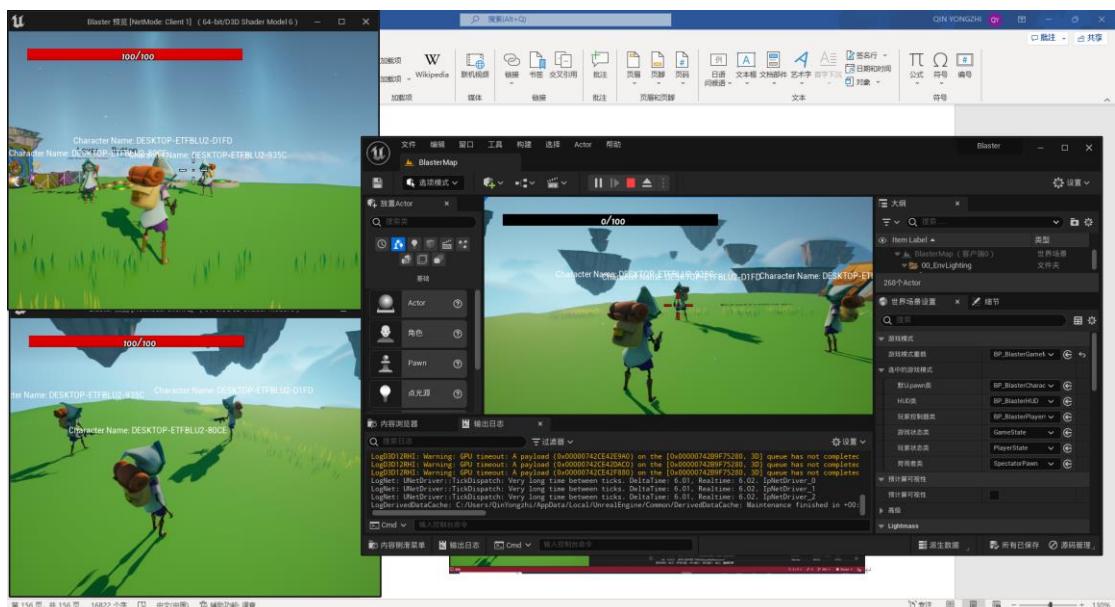
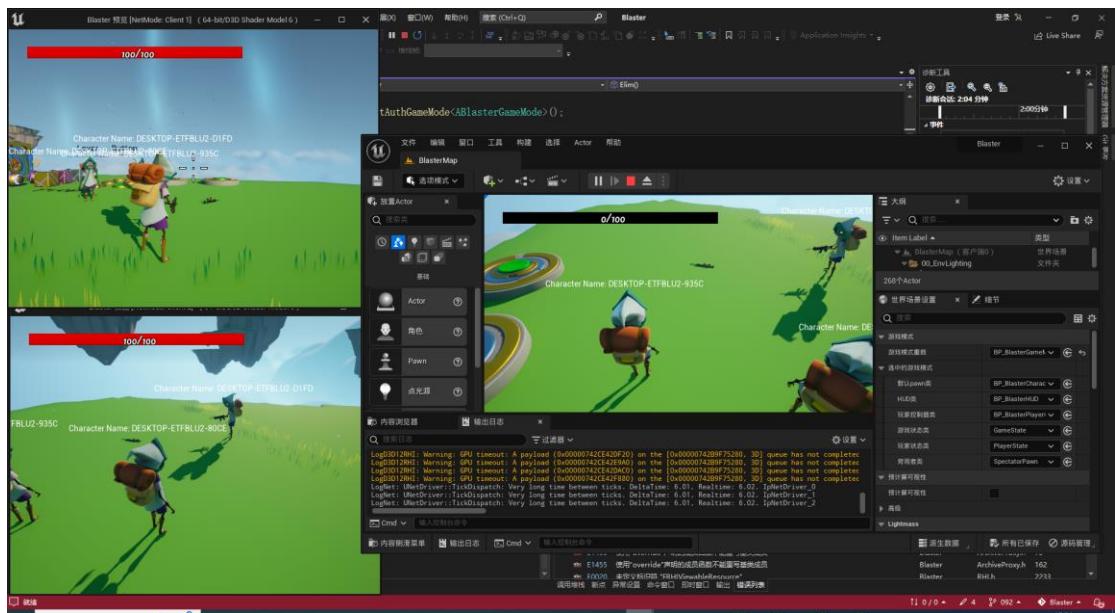
```

最后我们在角色的 Elim() 中添加如下代码即可：

```

if (Combat && Combat->EquippedWeapon)
{
    Combat->EquippedWeapon->Dropped();
}

```



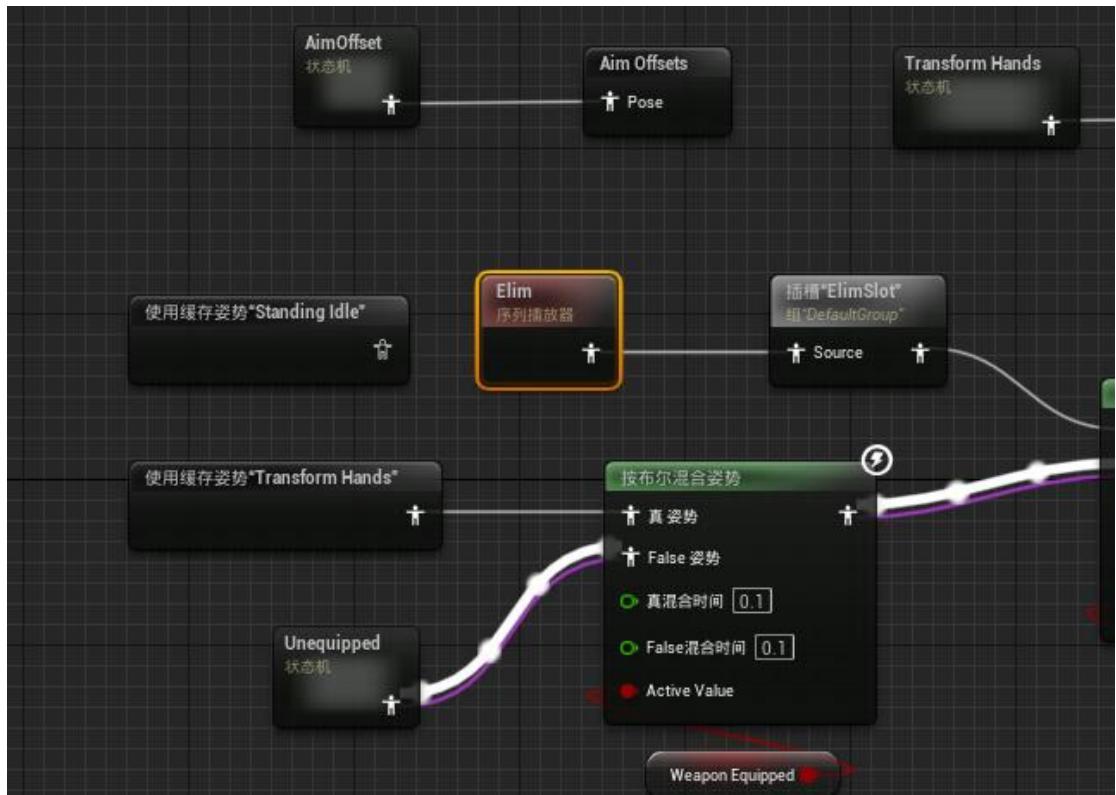
可以看到武器能被成功捡起。

但是我们可以看到两个问题：第一是由于在客户端也开启了物理模拟，不可避免的会导致在服务端和客户端物理模拟的不一致。

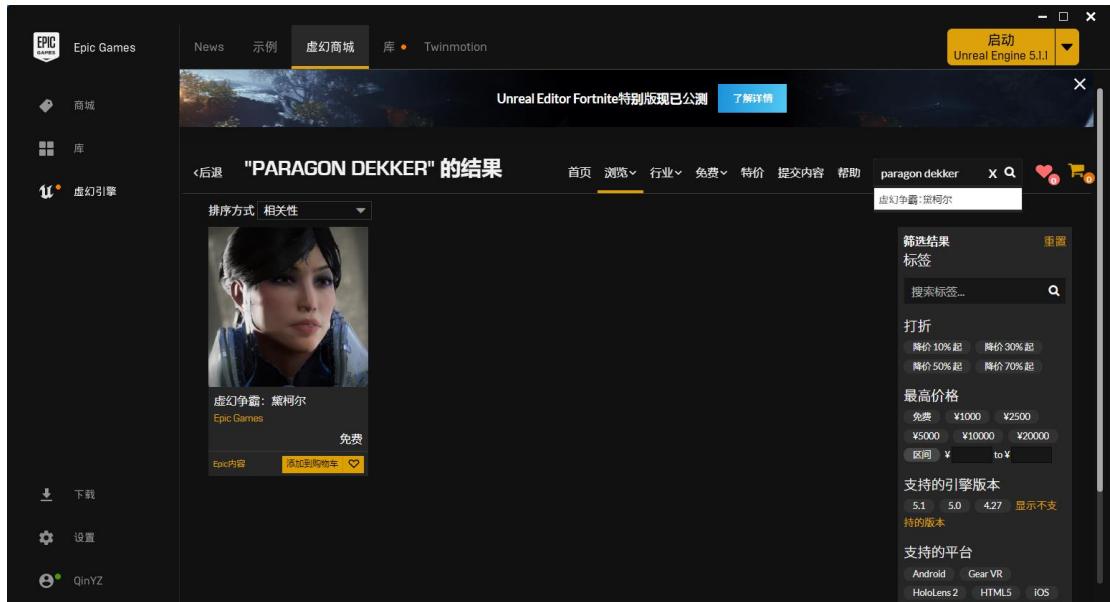
这个问题想要解决的话需要在服务端进行物理模拟，然后复制运动到客户端，这样做对网络的消耗增加，如果一定范围内的误差可以接受的话，可以直接在客户端模拟。

第二个问题是，在装备武器的情况下被击败不会播放淘汰动画。

此问题的解决办法是把动画蓝图进行修改，用 Elim 的动画代替之前的 Standing Idle 的状态。



094 淘汰机器人特效



搜索资源 paragon dekker，将他添加到之前的学习工具包的工程：



将其中的一个粒子效果迁移到 Blaster。

下载一个机器人的声音，并转换成 WAV 格式，导入文件。

在角色中添加以下成员变量：

```
/**  
 * Elim bot  
 */  
UPROPERTY(EditAnywhere)  
UParticleSystem* ElimBotEffect;
```

```

UPROPERTY(VisibleAnywhere)
UParticleSystemComponent* ElimBotComponent;

UPROPERTY(EditAnywhere)
class USoundCue* ElimBotSound;

```

在淘汰的多播函数中，添加如下代码：

```

// Spawn Elim Bot
if (ElimBotEffect)
{
    FVector ElimBotSpawnPoint(GetActorLocation().X, GetActorLocation().Y,
GetActorLocation().Z + 200. f);
    ElimBotComponent = UGameplayStatics::SpawnEmitterAtLocation(
        GetWorld(),
        ElimBotEffect,
        ElimBotSpawnPoint,
        GetActorRotation()
    );
}

if (ElimBotSound)
{
    UGameplayStatics::SpawnSoundAtLocation(
        this,
        ElimBotSound,
        GetActorLocation()
    );
}

```

最后我们要再 ElimTimerFinished 中销毁掉产生的机器人：

```

if (ElimBotComponent)
{
    ElimBotComponent->DestroyComponent();
}

```

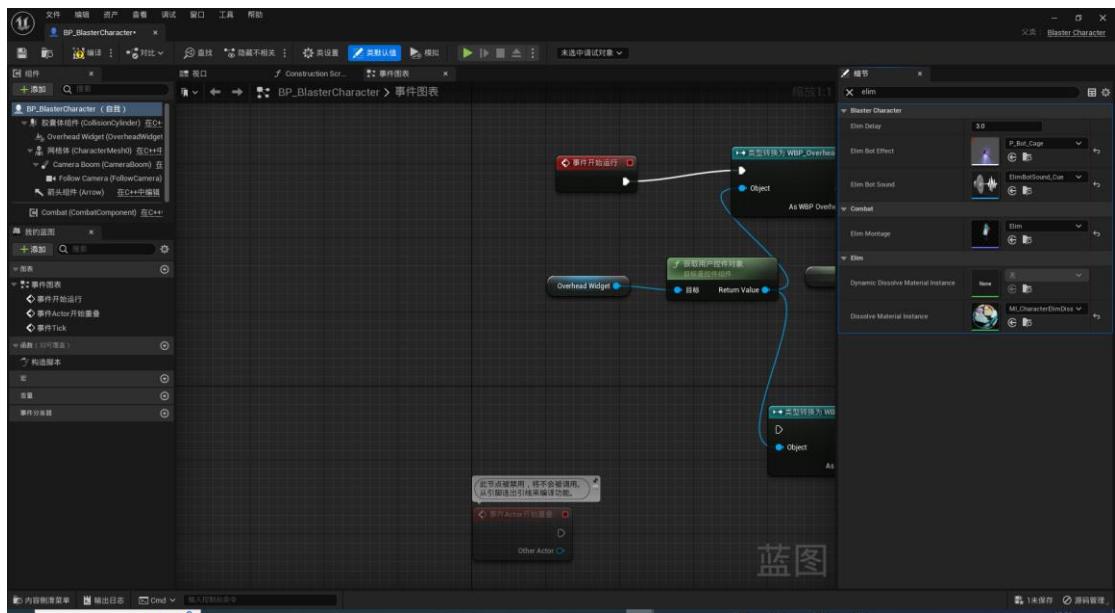
为了在所有客户端能够看见，我们把它放在 Destroyed()中执行。

```

void ABlasterCharacter::Destroyed()
{
    Super::Destroyed();

    if (ElimBotComponent)
    {
        ElimBotComponent->DestroyComponent();
    }
}

```



在蓝图中设置成员变量。

095 OnPossess 函数

我们之前服务端在被淘汰复活之后，血条无法回复，这是因为我们在 BeginPlay 中调用了更新HUD的函数，但是这个函数会尝试将角色的控制器转换成 ABlasterPlayerController 类型，如果失败就不会进行更新。APlayerController 中的 OnPossess 函数可以帮助我们解决这个问题：

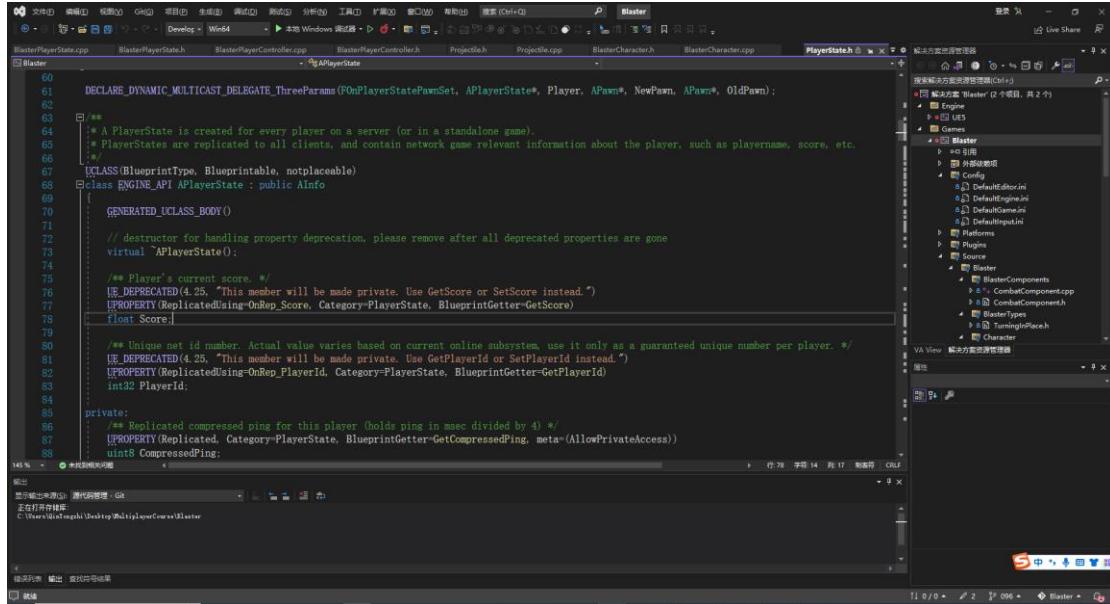
```
virtual void OnPossess(APawn* InPawn) override;

void ABlasterPlayerController::OnPossess(APawn* InPawn)
{
    Super::OnPossess(InPawn);

    ABlasterCharacter* BlasterCharacter = Cast<ABlasterCharacter>(InPawn);
    if (BlasterCharacter)
    {
        SetHUDHealth(BlasterCharacter->GetHealth(), BlasterCharacter->GetMaxHealth());
    }
}
```

096 Blaster 玩家状态

创建一个 C++ 的 PlayerState 类：



The screenshot shows the Unreal Engine 4 Editor interface. The code editor displays the `PlayerState.h` header file for the `Blaster` project. The header file defines a class `APlayerState` with various properties and methods, including `Score` and `PlayerId`. The Project Browser on the right shows the `Blaster` project structure, including source files like `BlasterCharacter.cpp` and `BlasterCharacter.h`, and configuration files like `DefaultEngine.ini` and `DefaultGame.ini`.

```
60 DECLARE_DYNAMIC_MULTICAST_DELEGATE_ThreeParams(FOnPlayerStatePowerSet, APlayerState*, Player, APawn*, NewPawn, APawn*, OldPawn);
61
62 /**
63  * A PlayerState is created for every player on a server (or in a standalone game).
64  * PlayerStates are replicated to all clients, and contain network game relevant information about the player, such as playername, score, etc.
65  */
66 UCLASS(BlueprintType, Blueprintable, notplaceable)
67 class ENGINE_API APlayerState : public AInfo
68 {
69     GENERATED_UCLASS_BODY()
70
71     // destructor for handling property deprecation. please remove after all deprecated properties are gone
72     virtual ~APlayerState();
73
74     /** Player's current score. */
75     UPROPERTY(ReplicatedUsing=OnRep_Score, Category=PlayerState, BlueprintGetter=GetScore)
76     float Score;
77
78     /** Unique net id number. Actual value varies based on current online subsystem, use it only as a guaranteed unique number per player. */
79     UPROPERTY(Replicated, Category=PlayerState, BlueprintGetter=GetPlayerId, meta=(AllowPrivateAccess))
80     int32 PlayerId;
81
82     /** Replicated compressed ping for this player (holds ping in msec divided by 4) */
83     UPROPERTY(Replicated, Category=PlayerState, BlueprintGetter=GetCompressedPing, meta=(AllowPrivateAccess))
84     uint8 CompressedPing;
85
86     /**
87      * @warning This member will be made private. Use GetScore or SetScore instead.
88      */
89     UPROPERTY(ReplicatedUsing=OnRep_PlayerId, Category=PlayerState, BlueprintGetter=GetPlayerId)
90     int32 PlayerId;
```

查看基类，可以看到其中有着分数一项。

CharacterOverlay 中添加变量：

```
UPROPERTY(meta = (BindWidget))
UTextBlock* ScoreAmount;
```

PlayerController 中添加函数：

```
void ABlasterPlayerController::SetHUDScore(float Score)
{
    BlasterHUD = BlasterHUD == nullptr ? BlasterHUD = Cast<ABlasterHUD>(GetHUD()) : BlasterHUD;

    bool bHUDValid = BlasterHUD &&
        BlasterHUD->CharacterOverlay &&
        BlasterHUD->CharacterOverlay->ScoreAmount;
    if (bHUDValid)
    {
        FString ScoreText = FString::Printf(TEXT("%d"), FMath::FloorToInt(Score));

        BlasterHUD->CharacterOverlay->ScoreAmount->SetText(FText::FromString(ScoreText));
    }
}
```

PlayerState 中我们要注意，REP 通知只会在客户端生效，所以要注意服务端的 HUD 设置不要被忘记：

```
void ABlasterPlayerState::AddToScore(float ScoreAmount)
{
    Score += ScoreAmount;
```

```

Character = Character == nullptr ? Cast<ABlasterCharacter>(GetPawn()) : Character;
if (Character)
{
    Controller = Controller == nullptr ?
Cast<ABlasterPlayerController>(Character->Controller) : Controller;
    if (Controller)
    {
        Controller->SetHUDScore(Score);
    }
}
}

void ABlasterPlayerState::OnRep_Score()
{
Super::OnRep_Score();

Character = Character == nullptr ? Cast<ABlasterCharacter>(GetPawn()) : Character;
if (Character)
{
    Controller = Controller == nullptr ?
Cast<ABlasterPlayerController>(Character->Controller) : Controller;
    if (Controller)
    {
        Controller->SetHUDScore(Score);
    }
}
}

```

最后，我们在 BlasterGameMode 中调用函数，增加分数：

```

void ABlasterGameMode::PlayerEliminated(ABlasterCharacter* ElimmedCharacter,
ABlasterPlayerController* VictimComtroller, ABlasterPlayerController*
AttackerController)
{
    ABlasterPlayerState* AttackerPlayerState = AttackerController ?
Cast<ABlasterPlayerState>(AttackerController->PlayerState) : nullptr;
    ABlasterPlayerState* VictimPlayerState = VictimComtroller ?
Cast<ABlasterPlayerState>(VictimComtroller->PlayerState) : nullptr;

    if (AttackerPlayerState && AttackerPlayerState != VictimPlayerState)
    {
        AttackerPlayerState->AddToScore(1. f);
    }

    if (ElimmedCharacter)

```

```

    {
        ElimmedCharacter->Elim();
    }
}

```

为我们的 PlayerState 类创建一个蓝图，并且在 GameMode 中设置使用该蓝图。

我们注意到在初始的时候，分数并没有更新，但是我们在这里不能直接将他用在 BeginPlay 中，正如我们前面的选做项中遇到的问题，PlayerState 在游戏的第一帧并没有初始化，所以我们需要之后再初始化，在当时我们使用了一个计时器，每隔 2 秒钟进行一次更新，而这里，教程给出了类似的方法，他在 Tick 中进行判断，第一次判断后保存一个成员变量，后续就不再进行更新。

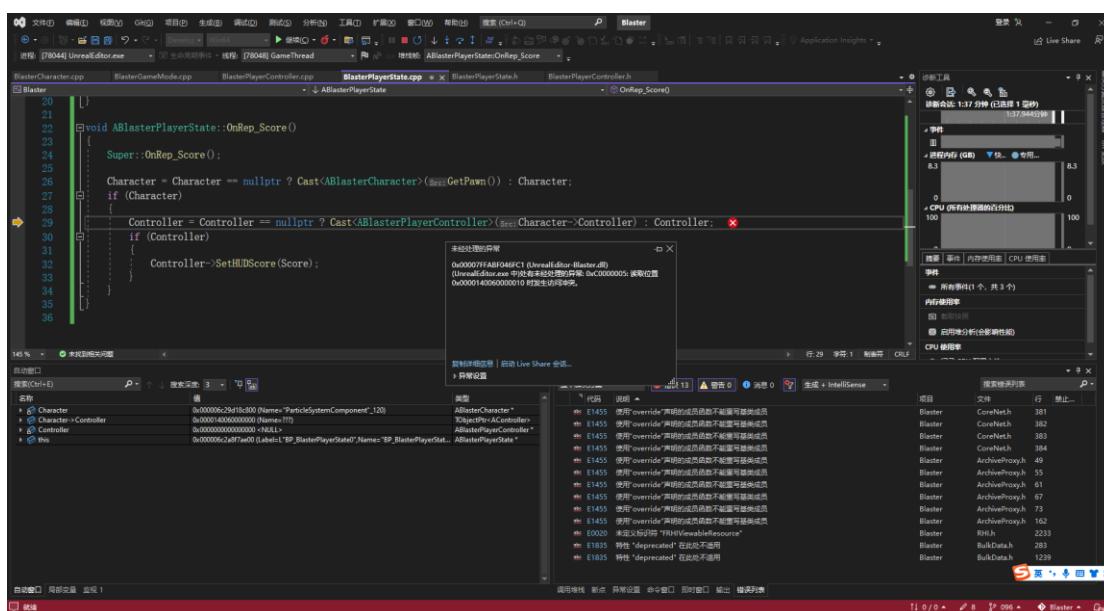
在角色的 Tick 中使用如下函数初始化 HUD 中的分数：

```

void ABlasterCharacter::PollInit()
{
    if (BlasterPlayerState == nullptr)
    {
        BlasterPlayerState = GetPlayerState<ABlasterPlayerState>();
        if (BlasterPlayerState)
        {
            BlasterPlayerState->AddToScore(0. f);
        }
    }
}

```

ISSUE



测试过程中居然还会出现 BUG！在 097 课中给出了解决方案，出现问题的原因是野指针可以通过 if 的判断，在之后的调用中产生内存错误，一个方法是给成员变量初始化为 nullptr，在 UE 中我们还可以给引擎自带类型的成员变量添加 UPROPERTY() 的宏标记来解决。

097 失败数

失败数和分数的设置类似，添加方法几乎一样，但是要注意的是，分数是 PlayerState 自带的变量，而失败数需要我们自己添加复制和复制通知：

要重写注册复制的函数：

```
void ABlasterPlayerState::GetLifetimeReplicatedProps(TArray<FLifetimeProperty>& OutLifetimeProps) const
{
    Super::GetLifetimeReplicatedProps(OutLifetimeProps);

    DOREPLIFETIME(ABlasterPlayerState, Defeats);
}

UPROPERTY(ReplicatedUsing = OnRep_Defeats)
int32 Defeats;
```

Defeats 我们采用整数 int32，因为 Score 系统选择了 float 所以我们在 Score 时沿用了 float，Defeats 是我们自己定义的量，所以可以选择 int32。

选做项

在 HUD 中显示击败信息

弹药

099 武器弹药

我们先把在武器弹药上要做的事情列举出来：

- 我们想给武器添加弹药以及弹夹容量 2 个变量 (Weapon)
- 控制器中添加一个设置 HUD 中子弹的函数 (Controller)
- 开火后减少一发子弹, Fire 中调用 SpendRound(), 并显示在 HUD 上, 我们需要在 Weapon 中连接控制器, 并进行控件的设置, 为了获取控制器, 我们需要一个 Character 变量以及一个 Controller 变量, 为了实现在所有客户端上的减少, 我们需要把子弹 Ammo 变量提升为复制的变量, 并提供复制通知 (Weapon)
- 捡起武器后, 在 UI 界面显示武器中的弹药容量, 需要在 HUD 中添加一个控件, 由于在 Actor 中 Owner 变量是一个复制变量, 并且拥有可以重载的复制通知, 我们在复制通知中调用 Weapon 中设置 HUD 的函数。这里不能使用 WeaponState 复制的原因是我们不知道 WeaponState 的复制和 Owner 的复制谁先进行, 而设置 HUD 需要用到获取 Owner (CharacterOverlay)
- 最后捡起武器后需要丢掉原有的武器, 我们在 combat 中 Equip 中进行判断, 如果原来有武器则丢掉之前的武器。

```
void AWeapon::SetHUDAmmo()
{
    BlasterOwnerCharacter = BlasterOwnerCharacter == nullptr ?
        Cast<ABlasterCharacter>(GetOwner()) : BlasterOwnerCharacter;
    if (BlasterOwnerCharacter)
    {
        BlasterOwnerController = BlasterOwnerController == nullptr ?
            Cast<ABlasterPlayerController>(BlasterOwnerCharacter->Controller) :
        BlasterOwnerController;
        if (BlasterOwnerController)
        {
            BlasterOwnerController->SetHUDWeaponAmmo(Ammo);
        }
    }
}

void AWeapon::SpendRound()
{
    --Ammo;
    SetHUDAmmo();
}

void AWeapon::OnRep_Ammo()
{
```

```

        SetHUDAmmo() ;
    }

void AWeapon::OnRep_Owner()
{
    Super::OnRep_Owner();
    if (Owner == nullptr)
    {
        BlasterOwnerCharacter = nullptr;
        BlasterOwnerController = nullptr;
    }
    else
    {
        SetHUDAmmo();
    }
}

void AWeapon::Dropped()
{
    SetWeaponState(EWeaponState::EWS_Dropped);
    FDetachmentTransformRules DetachRules(EDetachmentRule::KeepWorld, true);
    WeaponMesh->DetachFromComponent(DetachRules);
    SetOwner(nullptr);
    BlasterOwnerCharacter = nullptr;
    BlasterOwnerController = nullptr;
}

void ABlasterPlayerController::SetHUDWeaponAmmo(int32 Ammo)
{
    BlasterHUD = BlasterHUD == nullptr ? BlasterHUD = Cast<ABlasterHUD>(GetHUD()) : BlasterHUD;
    bool bHUDValid = BlasterHUD &&
        BlasterHUD->CharacterOverlay &&
        BlasterHUD->CharacterOverlay->WeaponAmmoAmount;
    if (bHUDValid)
    {
        FString AmmoText = FString::Printf(TEXT("%d"), Ammo);

        BlasterHUD->CharacterOverlay->WeaponAmmoAmount->SetText(FText::FromString(AmmoText));
    }
}

```

100 检查能否开火

为 Component 创建一个检查能否开火的函数：

```
bool UCombatComponent::CanFire()
{
    if (EquippedWeapon == nullptr)
    {
        return false;
    }

    if (EquippedWeapon->IsEmpty())
    {
        return false;
    }

    return bCanFire;
}
```

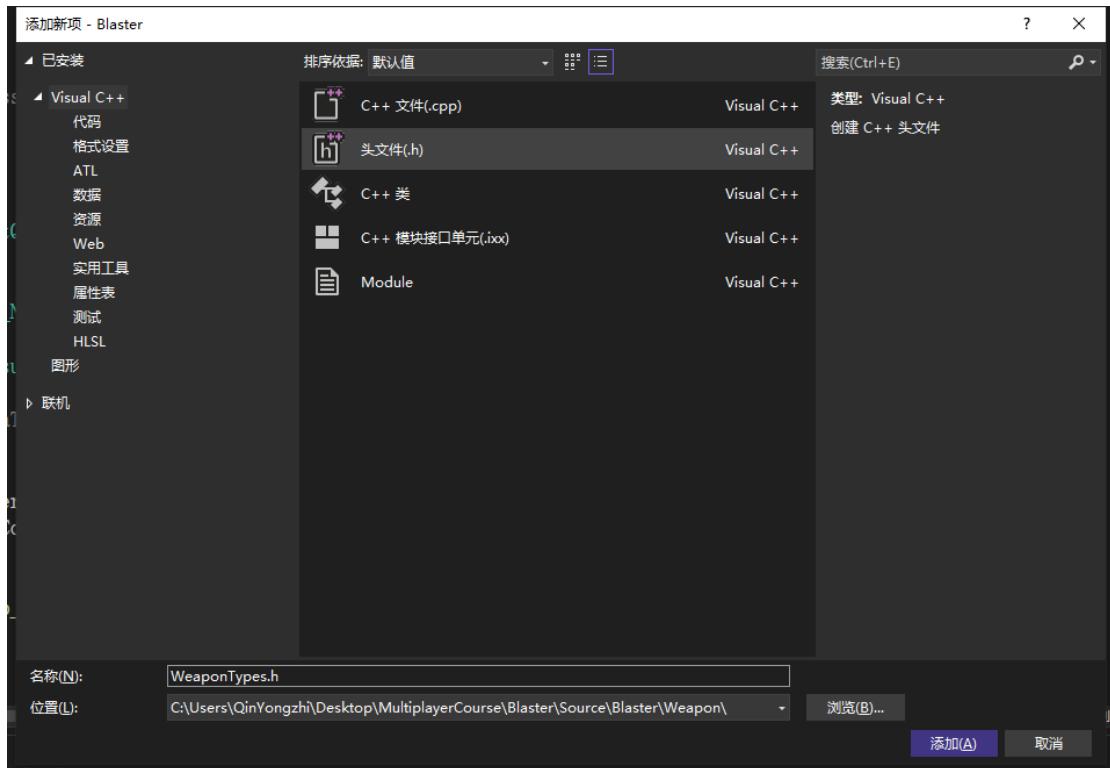
在 Weapon 中，比起—Ammo，我们可以钳制 Ammo 的值：

```
void AWeapon::SpendRound()
{
    Ammo = FMath::Clamp(Ammo - 1, 0, MagCapacity);
    SetHUDAmmo();
}
```

此外再提供一个判断是否还有子弹的接口：

```
bool AWeapon::IsEmpty()
{
    return Ammo <= 0;
}
```

101-102 携带弹药



为我们之后不同的武器添加不同的类：

我们打算把携带的弹药添加到 CombatComponent 中，为此，我们添加一系列变量和函数：

```
// Carried ammo for the currently-equipped weapon
UPROPERTY(ReplicatedUsing = OnRep_CarriedAmmo)
int32 CarriedAmmo;

UFUNCTION()
void OnRep_CarriedAmmo();

TMap<EWeaponType, int32> CarriedAmmoMap;

UPROPERTY(EditAnywhere)
int32 StartingARAMmo = 30;

void InitializeCarriedAmmo();
```

同样，在 Controller 中，添加一个设置 HUD 的函数：

```
void ABlasterPlayerController::SetHUDCarriedAmmo(int32 Ammo)
{
    BlasterHUD = BlasterHUD == nullptr ? BlasterHUD = Cast<ABlasterHUD>(GetHUD()) :
    BlasterHUD;
    bool bHUDValid = BlasterHUD &&
```

```
BlasterHUD->CharacterOverlay &&
BlasterHUD->CharacterOverlay->CarriedAmmoAmount;
if (bHUDValid)
{
    FString AmmoText = FString::Printf(TEXT("%d"), Ammo);

    BlasterHUD->CharacterOverlay->CarriedAmmoAmount->SetText(FText::FromString(AmmoText
)));
}
}
```

在 EquipWeapon 中进行设置：

```
Controller = Controller == nullptr ?
Cast<ABlasterPlayerController>(Character->Controller) : Controller;

if (CarriedAmmoMap.Contains(EquippedWeapon->GetWeaponType()))
{
    CarriedAmmo = CarriedAmmoMap[EquippedWeapon->GetWeaponType()];
}

if (Controller)
{
    Controller->SetHUDCarriedAmmo(CarriedAmmo);
}
```

103-104 换弹

新建一个枚举类 CombatState， 用于判断是否在换弹。

```
UENUM(BlueprintType)
enum class ECombatState : uint8
{
    ECS_Unoccupied UMETA(DisplayName = "Unoccupied"),
    ECS_Reloading UMETA(DisplayName = "Reloading"),
    ECS_Max UMETA(DisplayName = "DefaultMax")
};
```

在 Combat 中新建一个 Reload 函数， 和一个服务器 RPC， 添加一个 CombatState 变量，并设置为可复制，添加复制通知，由于要在动画的事件蓝图中调用，函数要用宏申明：

```
UFUNCTION(BlueprintCallable)
void FinishReloading();
```

具体实现如下：

```
void UCombatComponent::Reload()
{
    if (CarriedAmmo == 0)
    {
        return;
    }

    if (CombatState == ECombatState::ECS_Reloading)
    {
        return;
    }

    if (EquippedWeapon)
    {
        if (EquippedWeapon->GetAmmo() == EquippedWeapon->GetMagCapacity())
        {
            return;
        }
        ServerReload();
    }
}

void UCombatComponent::ServerReload_Implementation()
{
    if (Character == nullptr)
    {
        return;
    }
```

```

}

CombatState = ECombatState::ECS_Reloading;
HandleReload();
}

void UCombatComponent::OnRep_CombatState()
{
    switch (CombatState)
    {
        case ECombatState::ECS_Reloading:
            HandleReload();
            break;
    }
}

void UCombatComponent::FinishReloading()
{
    if (Character == nullptr)
    {
        return;
    }

    if (Character->HasAuthority())
    {
        CombatState = ECombatState::ECS_Unoccupied;
    }
}

void UCombatComponent::HandleReload()
{
    Character->PlayReloadMontage();
}

```

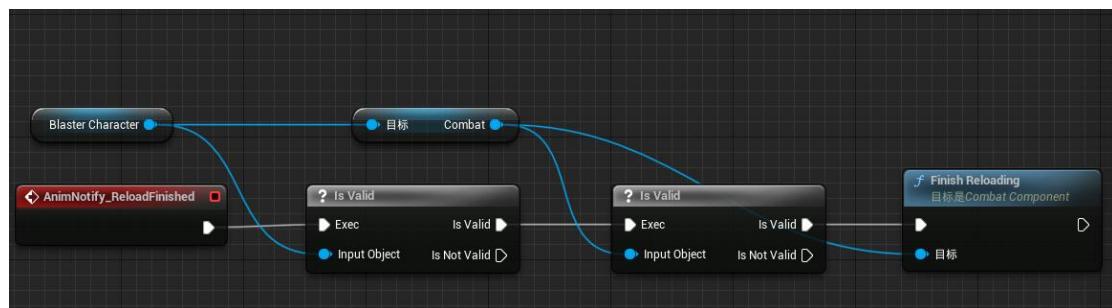
在角色类中，添加一个新的蒙太奇成员，此外由于要在蓝图中获取 Combat，所以需要修改 Combat 的宏定义：

```

UPROPERTY(VisibleAnywhere, BlueprintReadOnly, meta = (AllowPrivateAccess = "true"))
class UCombatComponent* Combat;

```

新建一个换弹的蒙太奇，并添加一个动画通知，在动画蓝图的事件蓝图中调用：



105 允许武器开火

我们还要解决两个问题，一个是在换弹期间开火的问题，另一个是换弹结束后回复开火。

首先要解决在换弹期间开火的问题很简单，我们可以在之前的 CanFire 函数中添加一个判断：

```
bool UCombatComponent::CanFire()
{
    if (EquippedWeapon == nullptr)
    {
        return false;
    }

    if (EquippedWeapon->IsEmpty())
    {
        return false;
    }

    return bCanFire && CombatState == ECombatState::ECS_Unoccupied;
}
```

要在换弹结束后能开火，我们可以在 FinsihReload 中添加一个判断，如果换弹结束后 bFireButtonPressed，则调用开火函数，但是为了防止播放开火动画中断掉换弹蒙太奇（指令重排？），导致不能够顺利的利用通知设置 CombatState，我们在多播函数中添加判断 CombatState == ECombatState::ECS_Unoccupied：

```
void UCombatComponent::MulticastFire_Implementation(const FVector_NetQuantize&
TraceHitTarget)
{
    if (nullptr == EquippedWeapon)
    {
        return;
    }

    if (Character && CombatState == ECombatState::ECS_Unoccupied)
    {
        Character->PlayFireMontage(bAiming);
        EquippedWeapon->Fire(TraceHitTarget);
    }
}
```

最后我们可以在复制通知中 case ECombatState::ECS_Unoccupied 进行 Fire 的调用，使得客户端能够顺利开火。

106 更新弹药

我们用如下函数计算装填弹药的数量：

```
int32 UCombatComponent::AmountToReload()
{
    if (EquippedWeapon == nullptr)
    {
        return 0;
    }

    int32 RoomInMag = EquippedWeapon->GetMagCapacity() - EquippedWeapon->GetAmmo();
    if (CarriedAmmoMap.Contains(EquippedWeapon->GetWeaponType()))
    {
        int32 AmountCarried = CarriedAmmoMap[EquippedWeapon->GetWeaponType()];
        int32 Least = FMath::Min(RoomInMag, AmountCarried);
        return FMath::Clamp(RoomInMag, 0, Least);
    }

    return 0;
}
```

我们用如下函数更新子弹和备弹的数值，由于 OnRep 的存在，结果会被复制到客户端 HUD 上：

```
void UCombatComponent::UpdateAmmoValues()
{
    if (Character == nullptr || EquippedWeapon == nullptr)
    {
        return;
    }

    int32 ReloadAmount = AmountToReload();
    if (CarriedAmmoMap.Contains(EquippedWeapon->GetWeaponType()))
    {
        CarriedAmmoMap[EquippedWeapon->GetWeaponType()] -= ReloadAmount;
        CarriedAmmo = CarriedAmmoMap[EquippedWeapon->GetWeaponType()];
    }

    Controller = Controller == nullptr ?
        Cast<ABlasterPlayerController>(Character->Controller) : Controller;
    if (Controller)
    {
        Controller->SetHUDCarriedAmmo(CarriedAmmo);
    }

    EquippedWeapon->AddAmmo(ReloadAmount);
}
```

我们在 finishReloading 中调用更新弹药数值的函数:

```
void UCombatComponent::FinishReloading()
{
    if (Character == nullptr)
    {
        return;
    }
    if (Character->HasAuthority())
    {
        CombatState = ECombatState::ECS_Unoccupied;
        UpdateAmmoValues();
    }
    if (bFireButtonPressed)
    {
        Fire();
    }
}
```

107 换弹效果

为我们的动画蓝图新添加两个变量：

```
bUseAimOffsets = BlasterCharacter->GetCombatState() != ECombatState::ECS_Reloading;  
bTransformRightHand = BlasterCharacter->GetCombatState() != ECombatState::ECS_Reloading;
```

用来控制我们关闭瞄准时的动作偏差。

在 Weapon 中创建一个 SoundCue，用来在 Combat 中播放装备武器的音效，注意要在 EquipWeapon 和 OnRep_EquippedWeapon 两个地方调用才能保证客户端和服务端都能听到。

在蒙太奇动画中添加音效通知。

108 自动装填

在 EquippWeapon 和 FireTimerFinished 的末尾调用 Reload 函数。

选做项

Optional Challenge: Show the Weapon Type in the HUD for the Equipped Weapon

When equipping a weapon, show the weapon type in the HUD, so we know what the Ammo and Carried Ammo values are.

Share your work in the 😊 | share-your-work channel in the Druid Mechanics Discord!

匹配状态

109 游戏计时器

我们打算为游戏做一个计时器，并显示在 HUD 上，首先我们需要一个 CharacterOverlay 的 TextBlock：

```
UPROPERTY(meta = (BindWidget))
UTextBlock* MatchCountdownText;
```

在 Controller 中需要一个函数来设置它：

```
void ABlasterPlayerController::SetHUDMatchCountdown(float CountdownTime)
{
    BlasterHUD = BlasterHUD == nullptr ? BlasterHUD = Cast<ABlasterHUD>(GetHUD()) : BlasterHUD;
    bool bHUDValid = BlasterHUD &&
        BlasterHUD->CharacterOverlay &&
        BlasterHUD->CharacterOverlay->MatchCountdownText;
    if (bHUDValid)
    {
        int32 Minutes = FMath::FloorToInt(CountdownTime / 60.0f);
        int32 Seconds = FMath::FloorToInt(CountdownTime - Minutes * 60.0f);

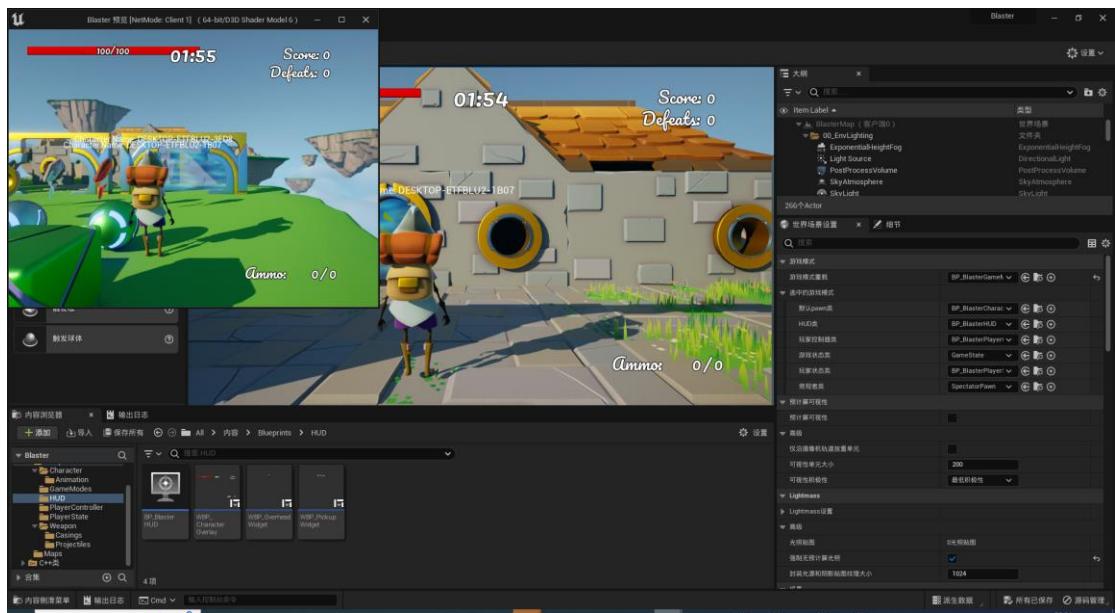
        FString CountdownText = FString::Printf(TEXT("%02d:%02d"), Minutes, Seconds);

        BlasterHUD->CharacterOverlay->MatchCountdownText->SetText(FText::FromString(CountdownText));
    }
}
```

我们设置一个函数在 Tick 中更新时间，我们希望时间每隔一秒才更新一次：

```
void ABlasterPlayerController::SetHudTime()
{
    uint32 SecondsLeft = FMath::CeilToInt(MatchTime - GetWorld()->GetTimeSeconds());
    if (CountdownInt != SecondsLeft)
    {
        SetHUDMatchCountdown(MatchTime - GetWorld()->GetTimeSeconds());
    }
    CountdownInt = SecondsLeft;
}
```

一个简单的计时器就做好了，但是一个问题就是这个计时器是基于客户端本身的，所以会出现服务器和客户端时间不同步的情况：



110 同步客户端和服务器时间

我们首先要知道怎么样同步客户端的时间，客户端向服务器发送请求查询时间，服务器向客户端传回服务器当前的时间，这个过程中存在两个延迟，一个是客户端发送向服务端的，一个是服务端返回客户端的。

虽然两个延迟的长度是不一样的，但是我们假设他们的延迟相同，用返回的服务器时间，加上整个 RTT 的一半就可以得到客户端收到时间时，服务器的时间。

为此我们需要两个 RPC 函数：

```
// Requests the current server time, passing in the client's time when the request  
was sent  
UFUNCTION(Server, Reliable)  
void ServerRequestServerTime(float TimeOfClientRequest);  
  
// Report the current server time to the client in response to  
ServerRequestServerTime  
UFUNCTION(Client, Reliable)  
void ClientReportServerTime(float TimeOfClientRequest, float  
TimeServerReceivedClientRequest);  
  
void ABlasterPlayerController::ServerRequestServerTime_Implementation(float  
TimeOfClientRequest)  
{  
    float ServerTimeOfReceipt = GetWorld()->GetTimeSeconds();  
    ClientReportServerTime(TimeOfClientRequest, ServerTimeOfReceipt);  
}  
  
void ABlasterPlayerController::ClientReportServerTime_Implementation(float  
TimeOfClientRequest, float TimeServerReceivedClientRequest)  
{  
    float RoundTripTime = GetWorld()->GetTimeSeconds() - TimeOfClientRequest;  
    float CurrentServerTime = TimeServerReceivedClientRequest + (0.5f * RoundTripTime);  
    ClientServerDelta = CurrentServerTime - GetWorld()->GetTimeSeconds();  
}
```

此外我们希望这个延迟初始化的时间越早越好，所以我们可以重写函数 ReceivedPlayer，来尽早的初始化，然后再提供一个获取服务器时间的函数：

```
float ABlasterPlayerController::GetServerTime()  
{  
    if (HasAuthority())  
    {  
        return GetWorld()->GetTimeSeconds();  
    }
```

```

        else
        {
            return GetWorld()->GetTimeSeconds() + ClientServerDelta;
        }
    }

void ABlasterPlayerController::ReceivedPlayer()
{
    Super::ReceivedPlayer();
    if (IsLocalController())
    {
        ServerRequestServerTime(GetWorld()->GetTimeSeconds());
    }
}

```

最后我们在 Tick 函数中每隔一段时间更新一次服务器时间，以免出现时间的漂移：

```

void ABlasterPlayerController::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);

    SetHUDTime();

    CheckTimeSync(DeltaTime);
}

void ABlasterPlayerController::CheckTimeSync(float DeltaTime)
{
    TimeSyncRunningTime += DeltaTime;
    if (IsLocalController() && TimeSyncRunningTime >= TimeSyncFrequency)
    {
        ServerRequestServerTime(GetWorld()->GetTimeSeconds());
        TimeSyncRunningTime = 0.0f;
    }
}

```

之前的 HUD 时间更新的函数也需要进行相应的更新：

```

void ABlasterPlayerController::SetHUDMatchCountdown(float CountdownTime)
{
    BlasterHUD = BlasterHUD == nullptr ? BlasterHUD = Cast<ABlasterHUD>(GetHUD()) :
    BlasterHUD;
    bool bHUDValid = BlasterHUD &&
        BlasterHUD->CharacterOverlay &&
        BlasterHUD->CharacterOverlay->MatchCountdownText;
}

```

```
if (bHUDValid)
{
    int32 Minutes = FMath::FloorToInt(CountdownTime / 60. f);
    int32 Seconds = FMath::FloorToInt(CountdownTime - Minutes * 60. f);

    FString CountdownText = FString::Printf(TEXT("%02d:%02d"), Minutes, Seconds);

    BlasterHUD->CharacterOverlay->MatchCountdownText->SetText(FText::FromString(CountdownText));
}
}
```

111 匹配状态

AGameModeBase:

- 默认类
- 生成玩家 Pawn
- 重启玩家
- 重启游戏

AGameMode

- 匹配状态
- 处理匹配状态
- 自定义匹配状态

GameMode 中有一个 namespace MatchState:

- 进入地图(EnteringMap)
- 等待开始(WaitingToStart)
- 进行中(InProgress)
- WaitingPostMatch
- 离开地图(LeavingMap)
- 终止(Aborted)

HasMatchStarted()返回我们是否在进行中。

HasMatchEnded()游戏是否结束

GetMatchState()

SetMatchState()

OnMatchStateSet()

StartMatch()

只能在 WaitingToStart 和 WaitingPostMatch 之间添加新状态。

我们要为我们的游戏创建三个新的状态：

WarmupTime, MatchTime 和 CooldownTime。

一个变量：

LevelStartingTime 用于从主菜单转移到 BlasterMap。

GameMode 中有一个变量 bDelayedStart，当被设置为 true 时，MatchState 将停留在 WaitingToStart 状态，知道我们调用 StartMatch()。

```

#include "CoreMinimal.h"
#include "UObject/ObjectMacros.h"
#include "Templates/SubclassOf.h"
#include "GameFramework/GameModeBase.h"
#include "GameMode.generated.h"

class APlayerState;
class ULocalMessage;
class UNetDriver;

/** Possible state of the current match, where a match is all the gameplay that happens on a single map */
namespace MatchState
{
    extern ENGINE_API const FName EnteringMap;           // We are entering this map, actors are not yet ticking
    extern ENGINE_API const FName WaitingToStart;        // Actors are ticking, but the match has not yet started
    extern ENGINE_API const FName InProgress;            // Normal gameplay is occurring. Specific games will have their own state machine inside
    extern ENGINE_API const FName WaitingPostMatch;       // Match has ended so we aren't accepting new players, but actors are still ticking
    extern ENGINE_API const FName LeavingMap;            // We are transitioning out of the map to another location
    extern ENGINE_API const FName Aborted;                // Match has failed due to network issues or other problems, cannot continue
}

// If a game needs to add additional states, you may need to override HasMatchStarted and HasMatchEnded to deal with the new states
// Do not add any states before WaitingToStart or after WaitingPostMatch

/** GameModeBase is a template class that handles the logic for a game mode */

```

```

GameMode.cpp BlasterGameMode.h
UE5 AGameMode
OnMatchStateSet()

348     }
349
350     void AGameMode::OnMatchStateSet()
351     {
352         FGameModeEvents::OnGameModeMatchStateSetEvent().Broadcast(MatchState);
353         // Call change callbacks
354         if (MatchState == MatchState::WaitingToStart)
355         {
356             HandleMatchIsWaitingToStart();
357         }
358         else if (MatchState == MatchState::InProgress)
359         {
360             HandleMatchHasStarted();
361         }
362         else if (MatchState == MatchState::WaitingPostMatch)
363         {
364             HandleMatchHasEnded();
365         }
366         else if (MatchState == MatchState::LeavingMap)
367         {
368             HandleLeavingMap();
369         }
370         else if (MatchState == MatchState::Aborted)
371         {
372             HandleMatchAborted();
373         }
374     }

```

利用如下代码实现在开启游戏后等待一段 WarmupTime 时间，然后才会生成角色以供玩家控制，在此之前玩家只能使用默认 pawn：

```

ABlasterGameMode::ABlasterGameMode()
{
    bDelayedStart = true;
}

void ABlasterGameMode::BeginPlay()
{
    Super::BeginPlay();

    LevelStartingTime = GetWorld()->GetTimeSeconds();
}

void ABlasterGameMode::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);
}

```

```
if (MatchState == MatchState::WaitingToStart)
{
    CountdownTime = WarmupTime - GetWorld()->GetTimeSeconds() + LevelStartingTime;
    if (CountdownTime <= 0. f)
    {
        StartMatch();
    }
}
```

首先在构造函数中打开bDelayedStart，然后在Beginplay中获取当前游戏时间，最后在Tick中计算时间，到时间后调用StartMatch函数。

112 OnMatchStateSet

我们希望在一开始使用默认Pawn的时候不显示UI界面，所以我们打算先将界面隐藏：这需要我们在PlayerController中获取GameState的状态，我们在PlayerController中设置一个复制变量：FName GameState，并且设置一个public的函数来设置它，在GameMode中我们在OnMatchStateSet调用这个函数设置PlayerController的变量，再利用复制通知在客户端执行。

```
void ABlasterGameMode::OnMatchStateSet()
{
    Super::OnMatchStateSet();

    for (FConstPlayerControllerIterator It = GetWorld()->GetPlayerControllerIterator();
It; It++)
    {
        ABlasterPlayerController* BlasterPlayer = Cast<ABlasterPlayerController>(*It);
        if (BlasterPlayer)
        {
            BlasterPlayer->OnMatchStateSet(MatchState);
        }
    }
}

void ABlasterPlayerController::OnMatchStateSet(FName State)
{
    MatchState = State;

    if (MatchState == MatchState::InProgress)
    {
        BlasterHUD = BlasterHUD == nullptr ? BlasterHUD =
Cast<ABlasterHUD>(GetHUD()) : BlasterHUD;
        if (BlasterHUD)
        {
            BlasterHUD->AddCharacterOverlay();
        }
    }
}

void ABlasterPlayerController::OnRep_MatchState()
{
    if (MatchState == MatchState::InProgress)
    {
        BlasterHUD = BlasterHUD == nullptr ? BlasterHUD =
Cast<ABlasterHUD>(GetHUD()) : BlasterHUD;
        if (BlasterHUD)
```

```

    {
        BlasterHUD->AddCharacterOverlay();
    }
}

```

我们将AddCharacterOverlay()改成public函数，并且不再在BeginPlay()中调用他，而是在MatchState == MatchState::InProgress之后才调用，我们发现，虽然这样在初始的时候隐藏了UI但是在开始的时候会出现生命值没有在UI上正确显示的问题，这是因为我们在SetHUDHealth的时候需要判断CharacterOverlay是否存在，为了解决这个问题，我们可以把在没有初始化之前SetHUDHealth,Score以及Defeats等值先记录下来，然后再在Tick中调用如下函数解决初始化更新的问题：

```

void ABlasterPlayerController::PollInit()
{
    if (CharacterOverlay == nullptr)
    {
        if (BlasterHUD && BlasterHUD->CharacterOverlay)
        {
            CharacterOverlay = BlasterHUD->CharacterOverlay;
            if (CharacterOverlay)
            {
                SetHUDHealth(HUDHealth, HUDMaxHealth);
                SetHUDScore(HUDScore);
                SetHUDDefeats(HUDDefeats);
            }
        }
    }
}

```

113 预热计时器

添加一个新的 UserWidget 命名为 Announcement。

在 BlasterHUD 中类似 CharacterOverlay 声明 2 个变量和一个初始化函数：

```
UPROPERTY(EditAnywhere, Category = "Announcements")
TSubclassOf<class UUserWidget> AnnouncementClass;

UPROPERTY()
class UAnnouncement* Announcement;
void AddAnnouncement();
```

在 Controller 中，在初始化阶段调用 AddAnnouncement 来初始化 UI。

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

    BlasterHUD = Cast<ABlasterHUD>(GetHUD());
    if (BlasterHUD)
    {
        BlasterHUD->AddAnnouncement();
    }
}
```

当游戏开始后，隐藏此 UI 而不是删除，以便之后结束游戏后再次使用：

```
void ABlasterPlayerController::OnMatchStateSet(FName State)
{
    MatchState = State;

    if (MatchState == MatchState::WaitingToStart)
    {

    }

    if (MatchState == MatchState::InProgress)
    {
        HandleMatchHasStarted();
    }
}

void ABlasterPlayerController::OnRep_MatchState()
{
    if (MatchState == MatchState::InProgress)
    {
        HandleMatchHasStarted();
    }
}
```

```
}

void ABlasterPlayerController::HandleMatchHasStarted()
{
    BlasterHUD = BlasterHUD == nullptr ? BlasterHUD = Cast<ABlasterHUD>(GetHUD()) :
    BlasterHUD;
    if (BlasterHUD)
    {
        BlasterHUD->AddCharacterOverlay();
        if (BlasterHUD->Announcement)
        {
            BlasterHUD->Announcement->SetVisibility(ESlateVisibility::Hidden);
        }
    }
}
```

114 更新预热时间

之前为了方便测试我们直接把 MatchTime 创建在了 Controller 中，现在我们可以把他更改到 GameMode 中，然后再从 GameMode 中使用 RPC 获取。

类似于在 CharacterOverlay 中更新时间，我们也需要创建一系列函数为 Announcement 更新时间：

```
void ABlasterPlayerController::SetHUDAnnouncementCountdown(float CountdownTime)
{
    BlasterHUD = BlasterHUD == nullptr ? BlasterHUD = Cast<ABlasterHUD>(GetHUD()) :
    BlasterHUD;
    bool bHUDValid = BlasterHUD &&
        BlasterHUD->Announcement &&
        BlasterHUD->Announcement->WarmupTime;
    if (bHUDValid)
    {
        int32 Minutes = FMath::FloorToInt(CountdownTime / 60.f);
        int32 Seconds = FMath::FloorToInt(CountdownTime - Minutes * 60.f);

        FString CountdownText = FString::Printf(TEXT("%02d:%02d"), Minutes, Seconds);

        BlasterHUD->Announcement->WarmupTime->SetText(FText::FromString(CountdownText));
    }
}
```

为了获取时间，我们需要两个 RPC 来进行操作，此外我们不再需要在 BeginPlay 中创建 Announcement 界面，二是在 RPC 之后查询状态后再决定是否创建：

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

    BlasterHUD = Cast<ABlasterHUD>(GetHUD());

    ServerCheckMatchState();
}

void ABlasterPlayerController::ServerCheckMatchState_Implementation()
{
    ABlasterGameMode* GameMode =
    Cast<ABlasterGameMode>(UGameplayStatics::GetGameMode(this));
    if (GameMode)
    {
        WarmupTime = GameMode->WarmupTime;
        MatchTime = GameMode->MatchTime;
        LevelStartingTime = GameMode->LevelStartingTime;
```

```

MatchState = GameMode->GetMatchState();
ClientJoinMidgame(MatchState, WarmupTime, MatchTime, LevelStartingTime);

if (BlasterHUD && MatchState == MatchState::WaitingToStart)
{
    BlasterHUD->AddAnnouncement();
}

void ABlasterPlayerController::ClientJoinMidgame_Implementation(FName StateOfMatch,
float Warmup, float Match, float StartingTime)
{
    WarmupTime = Warmup;
    MatchTime = Match;
    LevelStartingTime = StartingTime;
    MatchState = StateOfMatch;
    OnMatchStateSet(MatchState);
    if (BlasterHUD && MatchState == MatchState::WaitingToStart)
    {
        BlasterHUD->AddAnnouncement();
    }
}

```

我们之前创建了一个获取服务器时间的函数，我们可以利用它方便的计算出客户端的时间，所以我们只需要简单的修改 SetHUDTime 函数：

```

void ABlasterPlayerController::SetHUDTime()
{
    float TimeLeft = 0.f;
    if (MatchState == MatchState::WaitingToStart)
    {
        TimeLeft = WarmupTime - GetServerTime() + LevelStartingTime;
    }
    else if (MatchState == MatchState::InProgress)
    {
        TimeLeft = WarmupTime + MatchTime - GetServerTime() + LevelStartingTime;
    }

    uint32 SecondsLeft = FMath::CeilToInt(TimeLeft);

    if (CountdownInt != SecondsLeft)
    {
        if (MatchState == MatchState::WaitingToStart)
        {

```

```
        SetHUDAnnouncementCountdown(SecondsLeft);
    }
    if (MatchState == MatchState::InProgress)
    {
        SetHUDMatchCountdown(SecondsLeft);
    }
}
CountdownInt = SecondsLeft;
}
```

115 自定义匹配状态

首先为我们的 MatchState 添加一个新的状态

.h 文件中

```
namespace MatchState
{
    extern BLASTER_API const FName Cooldown; // Match duration has been reached.

    Display winner and begin cooldown timer.
}
```

.cpp 文件中

```
namespace MatchState
{
    const FName Cooldown = FName("Cooldown");
}
```

由于我们暂时不需要再 GameMode 中进行其他操作，只需要设置 GameState 转换的时机：

```
void ABlasterGameMode::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);

    if (MatchState == MatchState::WaitingToStart)
    {
        CountdownTime = WarmupTime - GetWorld()->GetTimeSeconds() + LevelStartingTime;
        if (CountdownTime <= 0. f)
        {
            StartMatch();
        }
    }
    else if (MatchState == MatchState::InProgress)
    {
        CountdownTime = WarmupTime + MatchTime - GetWorld()->GetTimeSeconds() +
        LevelStartingTime;
        if (CountdownTime <= 0. f)
        {
            SetMatchState(MatchState::Cooldown);
        }
    }
}
```

来到 Controller，我们要更改设置 GameState 的函数和他的复制函数：

```
void ABlasterPlayerController::OnMatchStateSet(FName State)
{
    MatchState = State;
```

```

    if (MatchState == MatchState::InProgress)
    {
        HandleMatchHasStarted();
    }
    else if (MatchState == MatchState::Cooldown)
    {
        HandleColldown();
    }
}

void ABlasterPlayerController::OnRep_MatchState()
{
    if (MatchState == MatchState::InProgress)
    {
        HandleMatchHasStarted();
    }
    else if (MatchState == MatchState::Cooldown)
    {
        HandleColldown();
    }
}

```

我们希望游戏结束之后，有一个 CooldownTime 来显示对局信息，所以我们将之前的 UI 隐藏，然后显示 Announcement 的 UI。

```

void ABlasterPlayerController::HandleColldown()
{
    BlasterHUD = BlasterHUD == nullptr ? BlasterHUD = Cast<ABlasterHUD>(GetHUD()) :
    BlasterHUD;
    if (BlasterHUD)
    {
        if (BlasterHUD->CharacterOverlay)
        {
            BlasterHUD->CharacterOverlay->SetVisibility(ESlateVisibility::Hidden);
        }
        //if (BlasterHUD->Announcement == nullptr)
        //{
        //    BlasterHUD->AddCharacterOverlay();
        //}
        //BlasterHUD->Announcement->SetVisibility(ESlateVisibility::Visible);
    }
}

```

116 Cooldown Announcement

完成 HandleCooldown 的函数:

```
void ABlasterPlayerController::HandleCooldown()
{
    BlasterHUD = BlasterHUD == nullptr ? BlasterHUD = Cast<ABlasterHUD>(GetHUD()) : BlasterHUD;
    if (BlasterHUD)
    {
        if (BlasterHUD->CharacterOverlay)
        {
            BlasterHUD->CharacterOverlay->SetVisibility(ESlateVisibility::Hidden);
        }
        if (BlasterHUD->Announcement == nullptr)
        {
            BlasterHUD->AddCharacterOverlay();
        }

        bool bHUDValid = BlasterHUD->Announcement &&
            BlasterHUD->Announcement->InfoText &&
            BlasterHUD->Announcement->AnnouncementText;
        if (bHUDValid)
        {
            BlasterHUD->Announcement->SetVisibility(ESlateVisibility::Visible);
            FString AnnouncementText("New Match Starts In:");
            BlasterHUD->Announcement->AnnouncementText->SetText(FText::FromString(AnnouncementText));
            BlasterHUD->Announcement->InfoText->SetText(FText());
        }
    }
}
```

完成 SetUpTime 的函数:

```
void ABlasterPlayerController::SetHUDTime()
{
    float TimeLeft = 0.f;
    if (MatchState == MatchState::WaitingToStart)
    {
        TimeLeft = WarmupTime - GetServerTime() + LevelStartingTime;
    }
    else if (MatchState == MatchState::InProgress)
    {
        TimeLeft = WarmupTime + MatchTime - GetServerTime() + LevelStartingTime;
    }
}
```

```

else if (MatchState == MatchState::Cooldown)
{
    TimeLeft = WarmupTime + MatchTime + CooldownTime - GetServerTime() +
LevelStartingTime;
}
uint32 SecondsLeft = FMath::CeilToInt(TimeLeft);

if (HasAuthority())
{
    BlasterGameMode = BlasterGameMode == nullptr ?
Cast<ABlasterGameMode>(UGameplayStatics::GetGameMode(this)) : BlasterGameMode;
    if (BlasterGameMode)
    {
        SecondsLeft = FMath::CeilToInt(BlasterGameMode->GetCountdownTime() +
LevelStartingTime);
    }
}

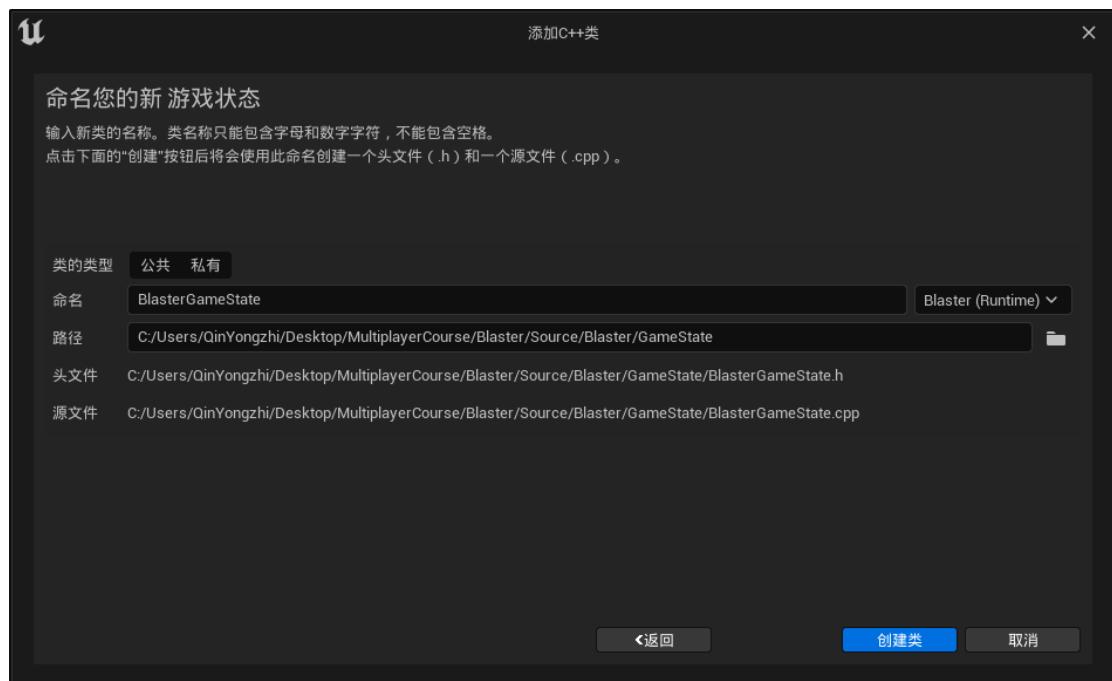
if (CountdownInt != SecondsLeft)
{
    if (MatchState == MatchState::WaitingToStart || MatchState ==
MatchState::Cooldown)
    {
        SetHUDAnnouncementCountdown(TimeLeft);
    }
    if (MatchState == MatchState::InProgress)
    {
        SetHUDMatchCountdown(TimeLeft);
    }
}
CountdownInt = SecondsLeft;
}

```

117 重启游戏

在角色中设置一个变量来禁用游戏内的 Gameplay 相关的操作，在 GameMode 中设置了当 Cooldown 结束后调用 RestartGame()。

118 Blaster Game state



创建一个 GameState 类。

在 GameState 中我们追踪分数最高的玩家：

```
void ABlasterGameState::GetLifetimeReplicatedProps(TArray<FLifetimeProperty>& OutLifetimeProps) const
{
    Super::GetLifetimeReplicatedProps(OutLifetimeProps);

    DOREPLIFETIME(ABlasterGameState, TopScoringPlayers);
}

void ABlasterGameState::UpdateTopScore(ABlasterPlayerState* ScoringPlayer)
{
    if (TopScoringPlayers.Num() == 0)
    {
        TopScoringPlayers.Add(ScoringPlayer);
        TopScore = ScoringPlayer->GetScore();
    }
    else if (TopScore == ScoringPlayer->GetScore())
    {
        TopScore = ScoringPlayer->GetScore();
        TopScoringPlayers.Add(ScoringPlayer);
    }
}
```

```

    {
        TopScoringPlayers.AddUnique(ScoringPlayer);
    }
    else if (TopScore < ScoringPlayer->GetScore())
    {
        TopScoringPlayers.Empty();
        TopScoringPlayers.AddUnique(ScoringPlayer);
        TopScore = ScoringPlayer->GetScore();
    }
}

```

在 GamePlayerController 中我们在 cooldown 中显示获胜玩家:

```

void ABlasterPlayerController::HandleCooldown()
{
    BlasterHUD = BlasterHUD == nullptr ? BlasterHUD = Cast<ABlasterHUD>(GetHUD()) : BlasterHUD;
    if (BlasterHUD)
    {
        if (BlasterHUD->CharacterOverlay)
        {
            BlasterHUD->CharacterOverlay->SetVisibility(EVisibility::Hidden);
        }
        if (BlasterHUD->Announcement == nullptr)
        {
            BlasterHUD->AddCharacterOverlay();
        }

        bool bHUDValid = BlasterHUD->Announcement &&
                        BlasterHUD->Announcement->InfoText &&
                        BlasterHUD->Announcement->AnnouncementText;
        if (bHUDValid)
        {
            BlasterHUD->Announcement->SetVisibility(EVisibility::Visible);
            FString AnnouncementText("New Match Starts In:");
            BlasterHUD->Announcement->AnnouncementText->SetText(FText::FromString(AnnouncementText));
        }

        ABlasterGameState* BlasterGameState =
        Cast<ABlasterGameState>(UGameplayStatics::GetGameState(this));
        ABlasterPlayerState* BlasterPlayerState =
        GetPlayerState<ABlasterPlayerState>();
        if (BlasterGameState && BlasterPlayerState)
        {
    }
}

```

```

TArray<ABlasterPlayerState*> TopPlayers =
BlasterGameState->TopScoringPlayers;
    FString InfoTextString;
    if (TopPlayers.Num() == 0)
    {
        InfoTextString = FString("This is no winner.");
    }
    else if (TopPlayers.Num() == 1 && TopPlayers[0] ==
BlasterPlayerState)
    {
        InfoTextString = FString("You are the winner!");
    }
    else if (TopPlayers.Num() == 1)
    {
        InfoTextString = FString::Printf(TEXT("winner: \n%s."),
*TopPlayers[0]->GetPlayerName());
    }
    else if (TopPlayers.Num() > 1)
    {
        InfoTextString = FString("Players tied for the win:\n");
        for (auto TiedPlayer : TopPlayers)
        {
            InfoTextString.Append(FString::Printf(TEXT("%s\n"),
*TiedPlayer->GetPlayerName())));
        }
    }
    BlasterHUD->Announcement->InfoText->SetText(FText::FromString(InfoTextString));
}
}

ABlasterCharacter* BlasterCharacter = Cast<ABlasterCharacter>(GetPawn());
if (BlasterCharacter && BlasterCharacter->GetCombatComponent())
{
    BlasterCharacter->bDisableGameplay = true;
    BlasterCharacter->GetCombatComponent()->FireButtonPressed(false);
}
}

```

选做项

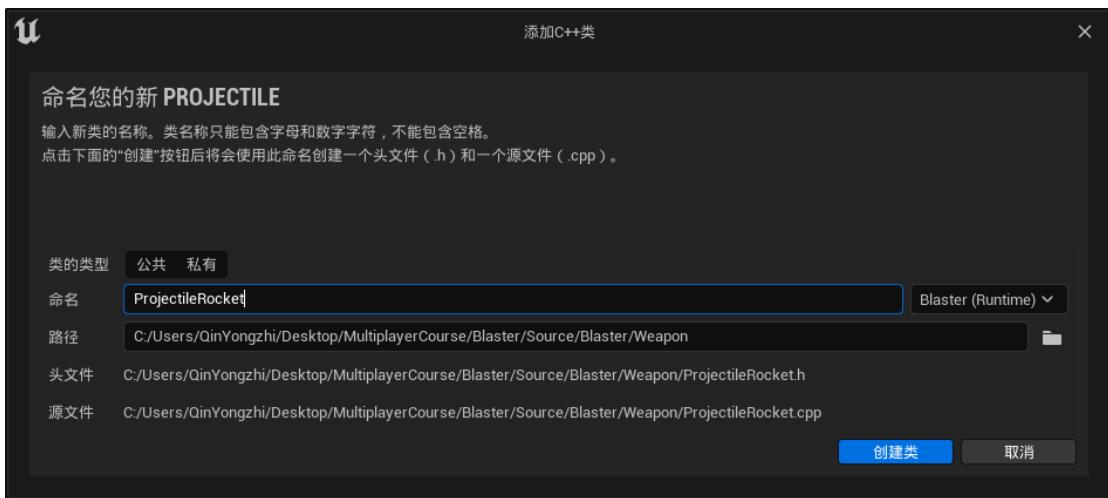
Optional Challenge: Blinking Countdown Text

To give the players a sense of urgency, when there is only 30 seconds left in the match, make the countdown text red and blinking.

Share your solution in the 😊 | share-your-work channel in the Druid Mechanics Discord!

各种武器

119 火箭投射物



```
AProjectileRocket::AProjectileRocket()
{
    RocketMesh = CreateDefaultSubobject<UStaticMeshComponent>(TEXT("RocketMesh"));
    RocketMesh->SetupAttachment(RootComponent);
    RocketMesh->SetCollisionEnabled(ECollisionEnabled::NoCollision);
}

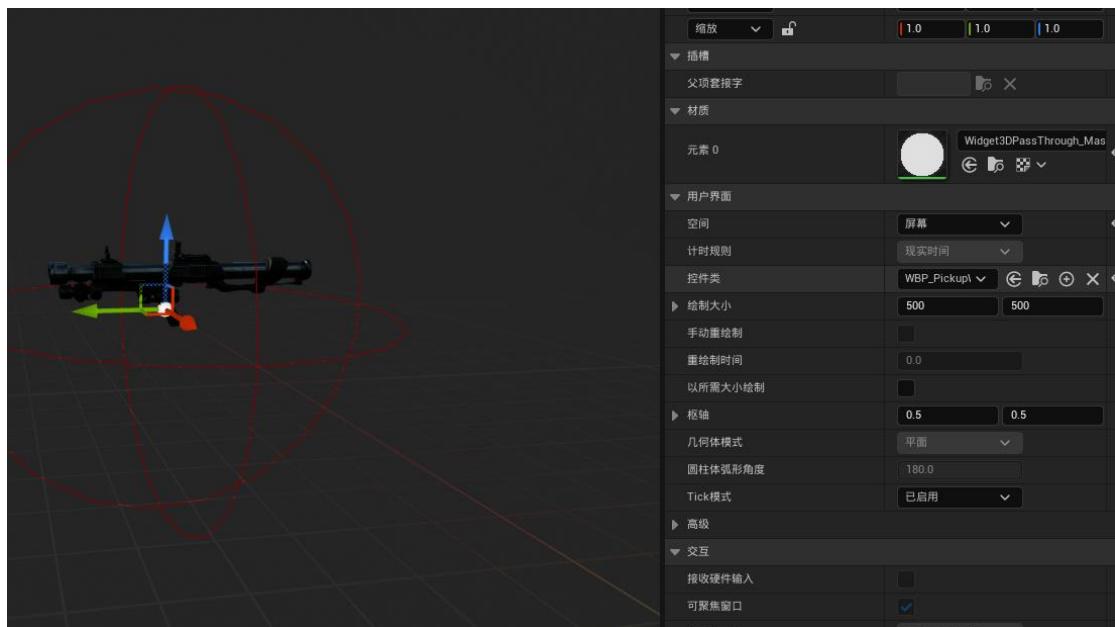
void AProjectileRocket::OnHit(UPrimitiveComponent* HitComp, AActor* OtherActor,
    UPrimitiveComponent* OtherComp, FVector NormalImpulse, const FHitResult& Hit)
{
    APawn* FiringPawn = GetInstigator();
    if (FiringPawn)
    {
        AController* FiringController = FiringPawn->GetController();
        if (FiringController)
        {
            UGameplayStatics::ApplyRadialDamageWithFalloff(
                this,
                Damage,
                MinimumDamage,
                GetActorLocation(),
                DamageInnerRadius,
                DamageOuterRadius,
                1. f,
                UDamageType::StaticClass(),
                TArray<AActor*>(),
                0.0f
            );
        }
    }
}
```

```

        this,
        FiringController
    );
}

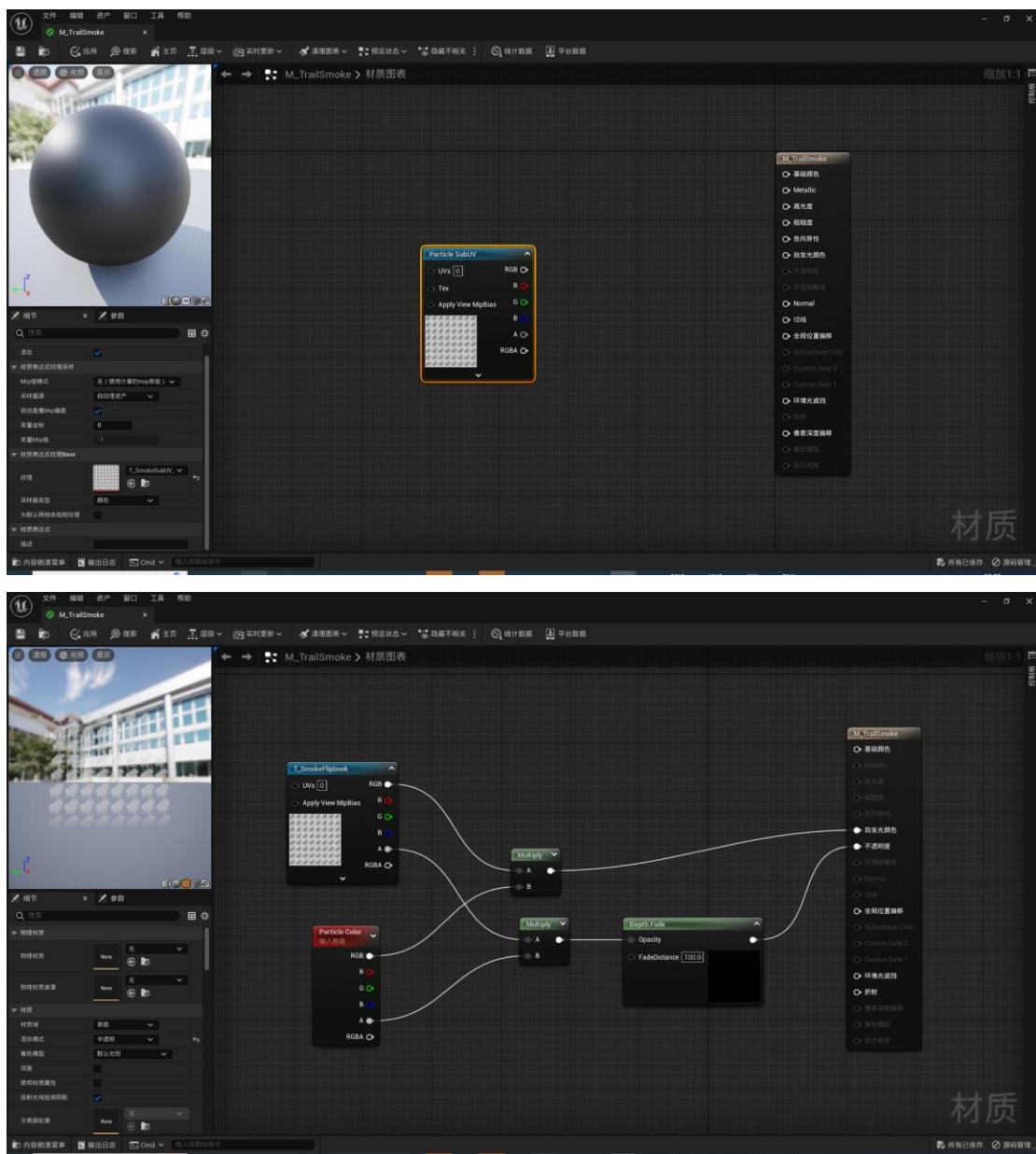
Super::OnHit(HitComp, OtherActor, OtherComp, NormalImpulse, Hit);
}

```



创建一个 RocketLauncher，并进行一系列设置，最后在 WeaponType 中添加一个新的枚举量，以及在 Combat 中做相应的更新即可。

120 火箭尾迹





为发射器选择一个起始点



新建发射器

从一个模板或行为发射器（无继承关系）或从一个父项（继承关系）新建一个发射器

拷贝现有发射器

从项目内容中拷贝一个现有的发射器

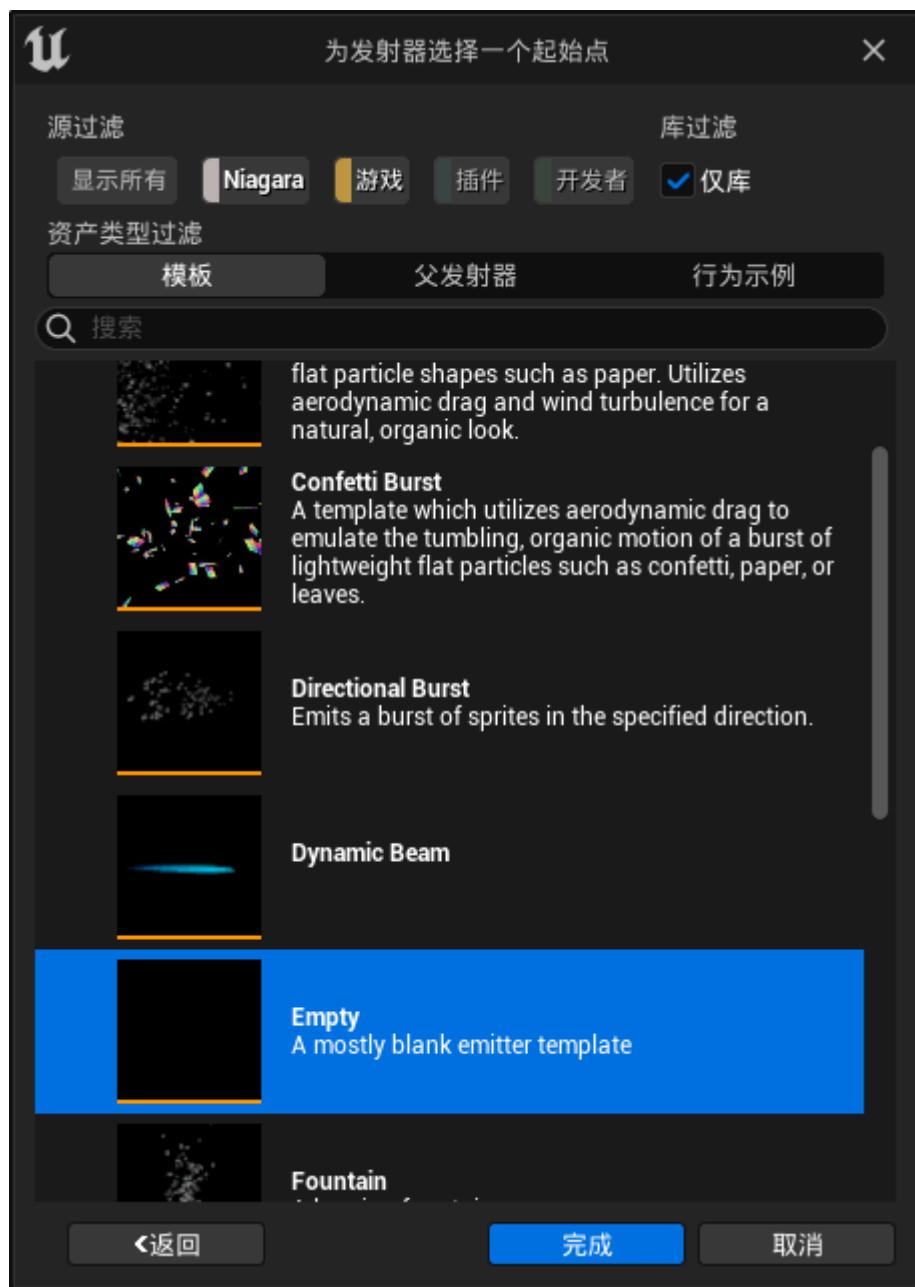
创建一个空发射器

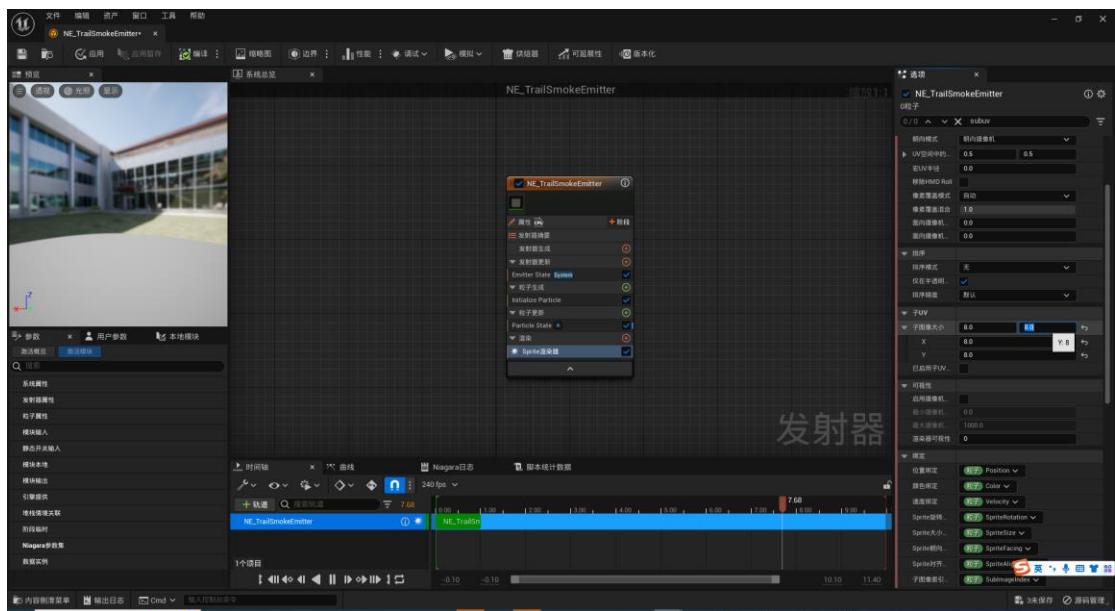
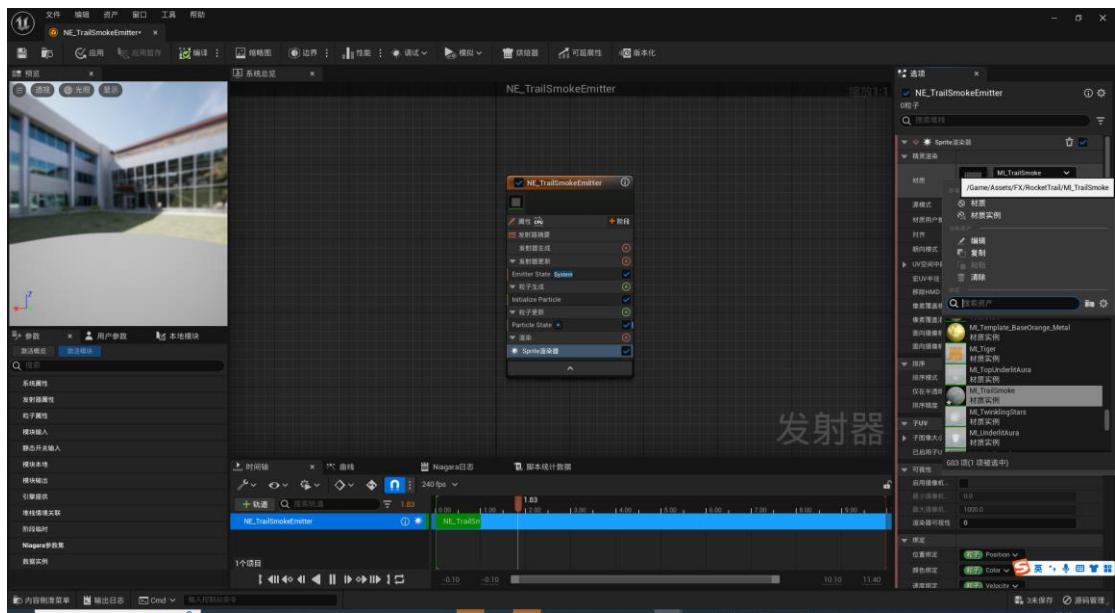
创建一个不含模块或渲染器的空白发射器

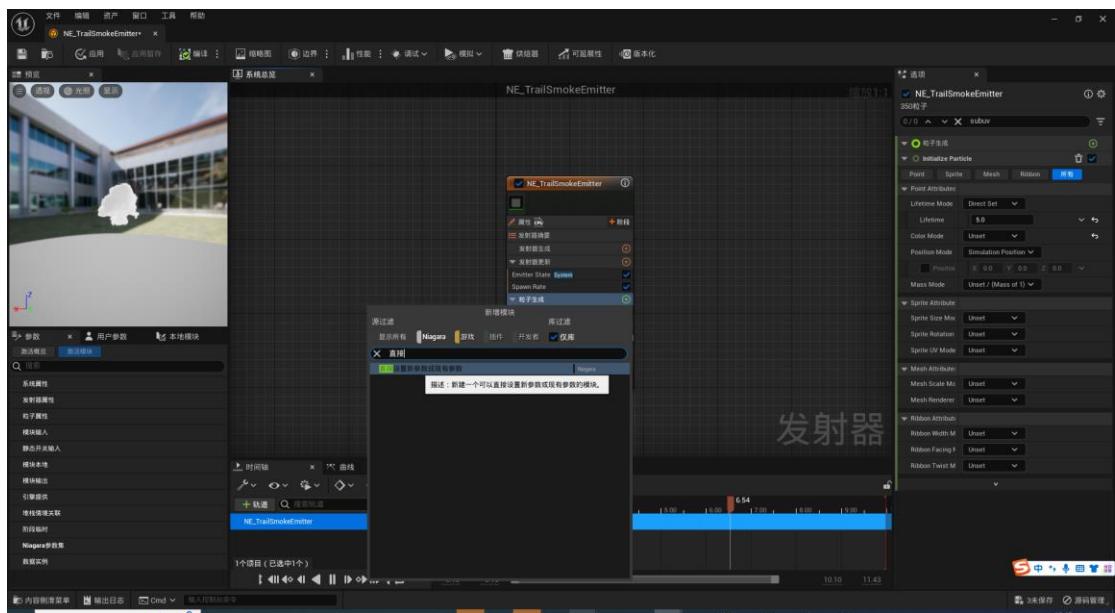
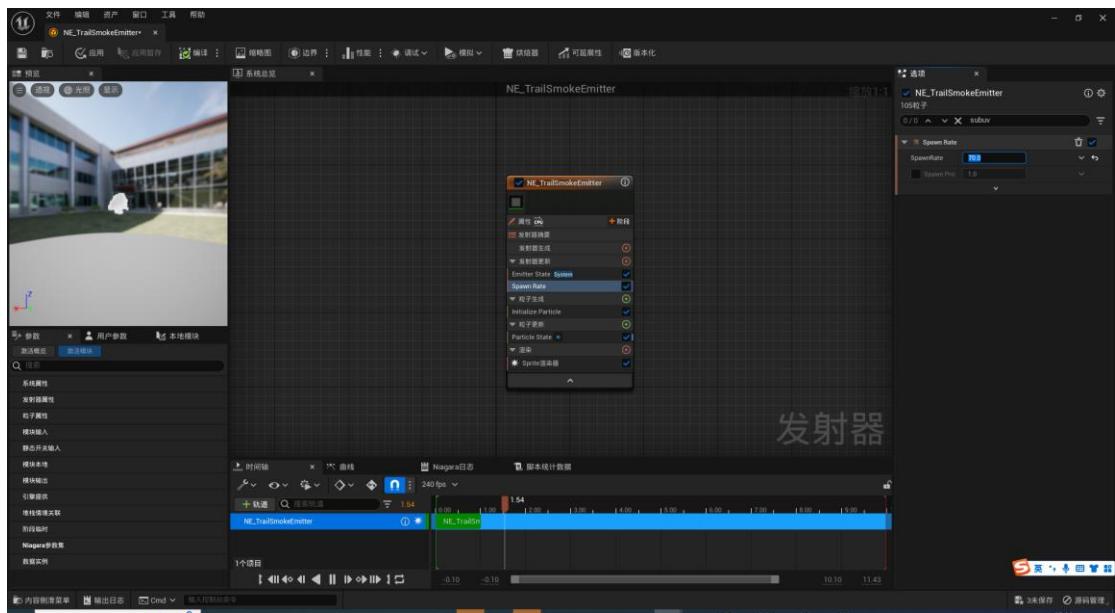
下一步 >

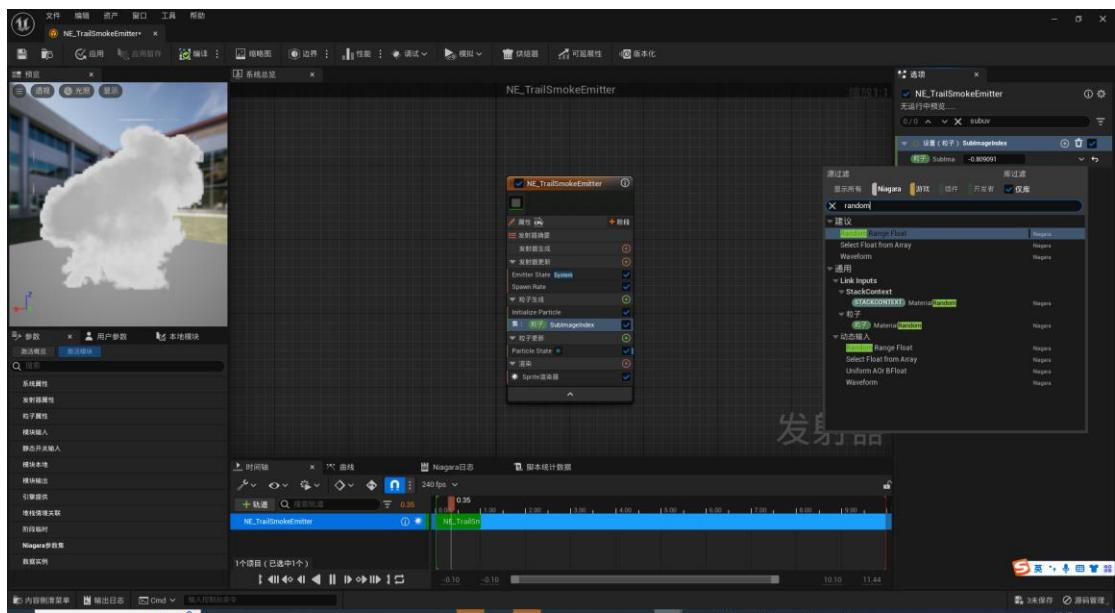
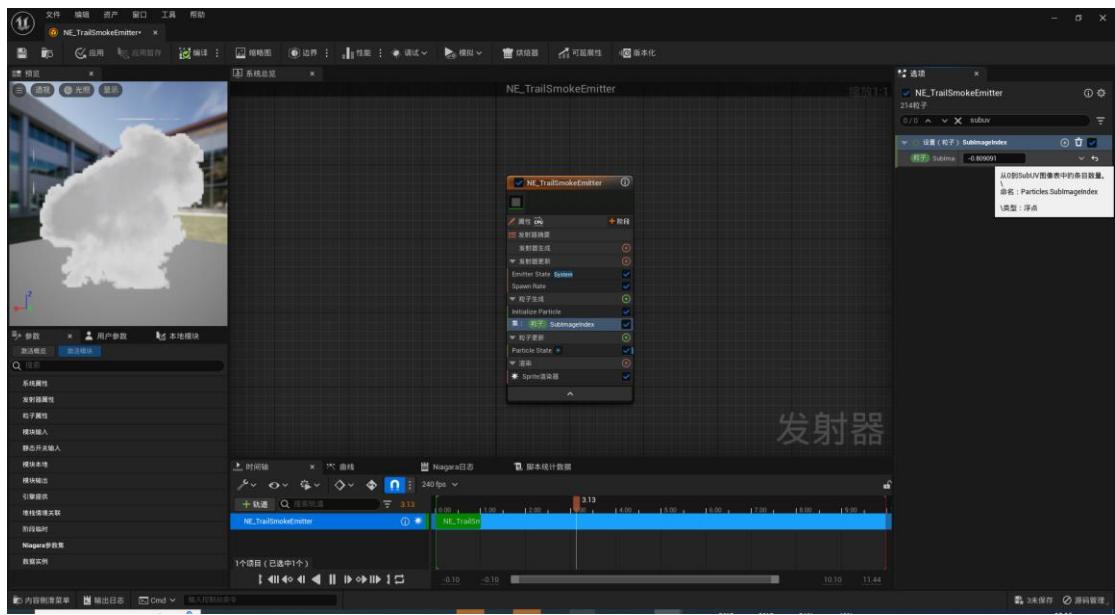
完成

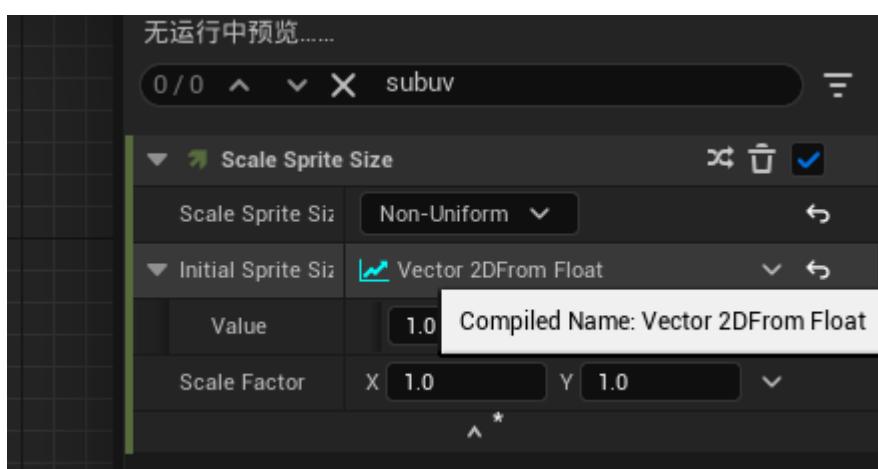
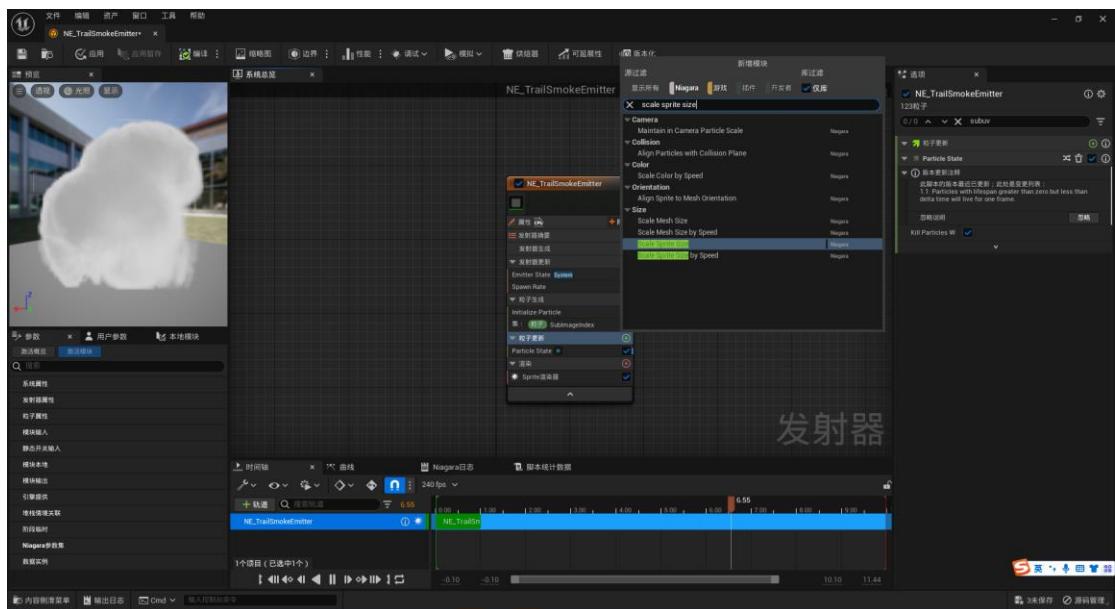
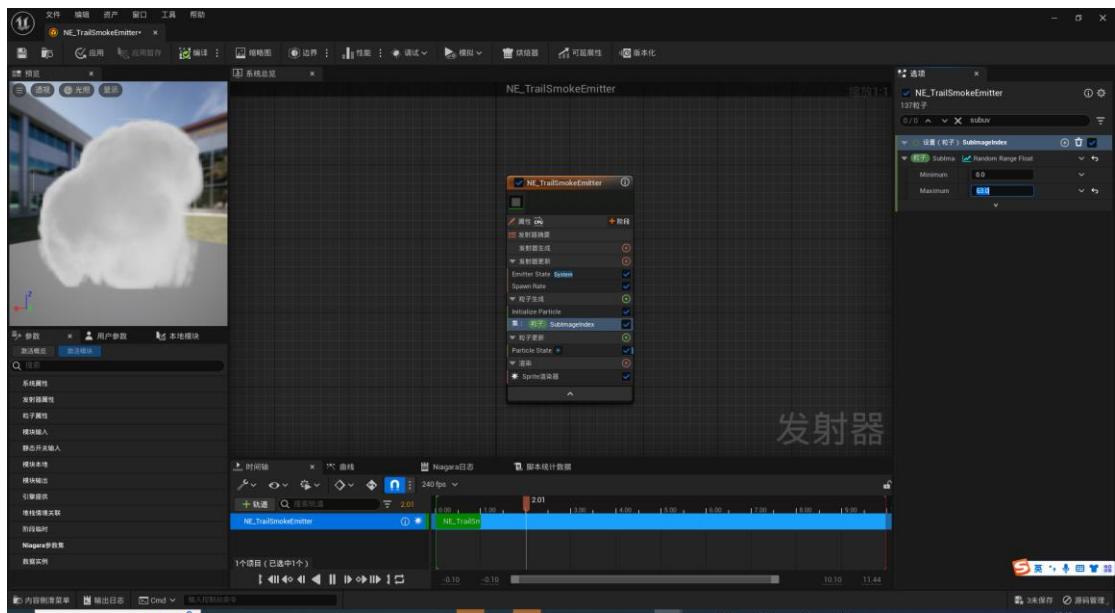
取消

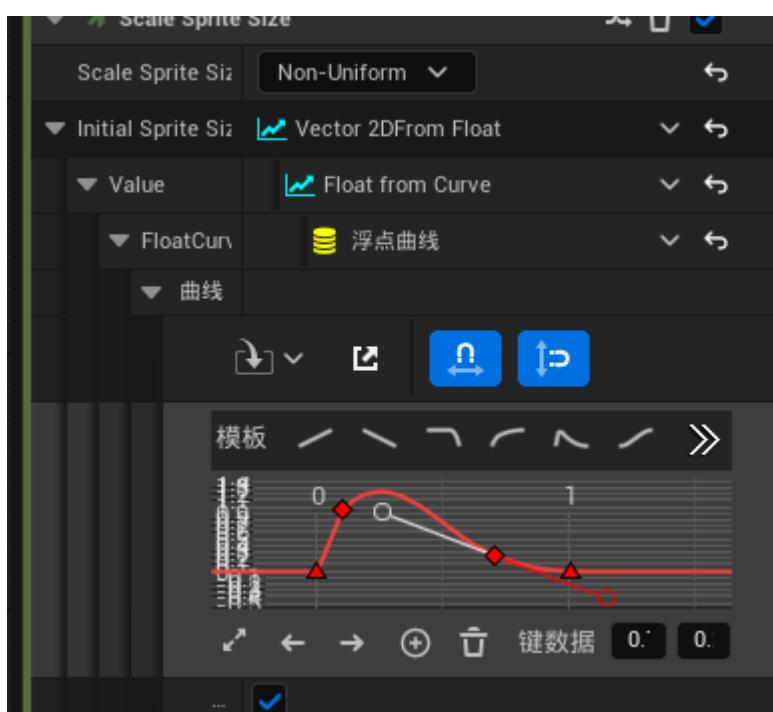
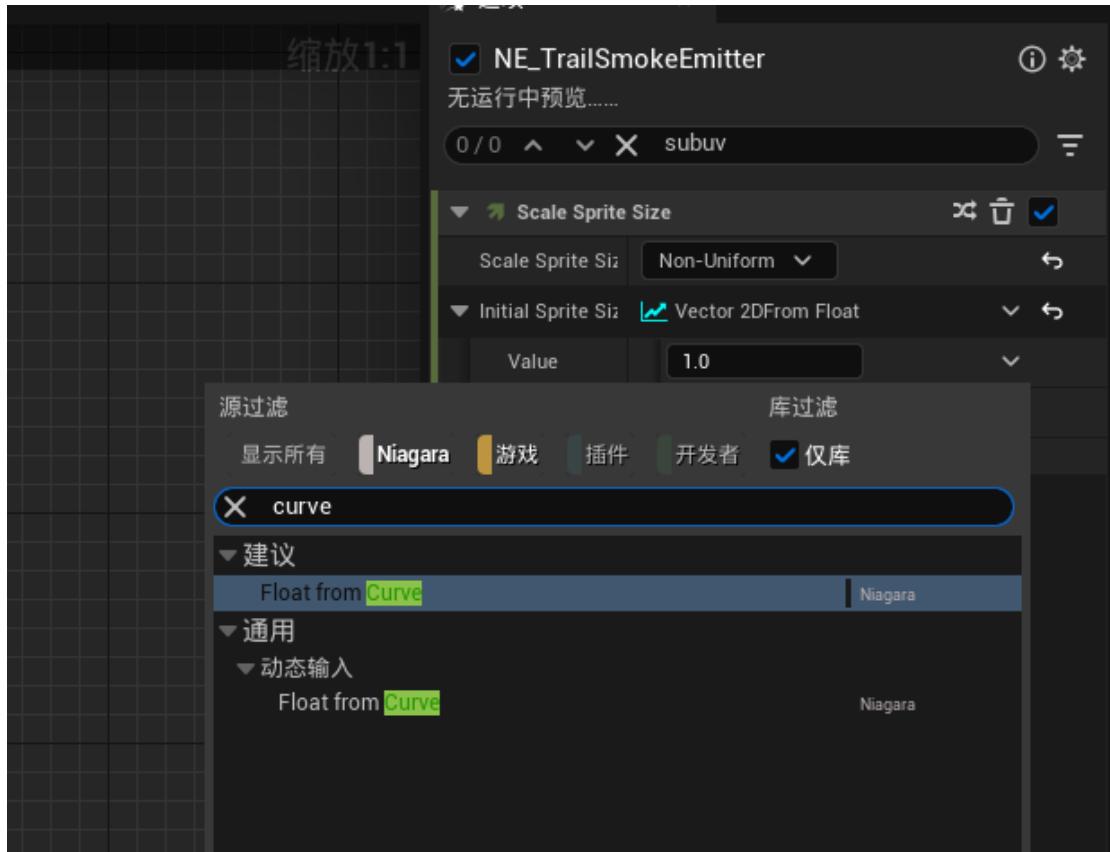


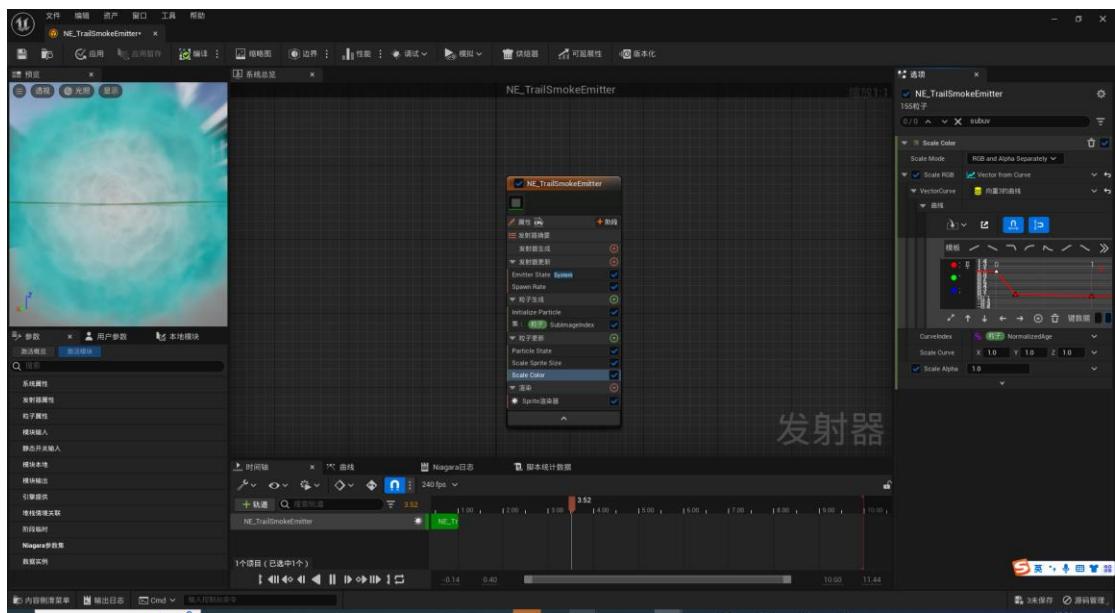
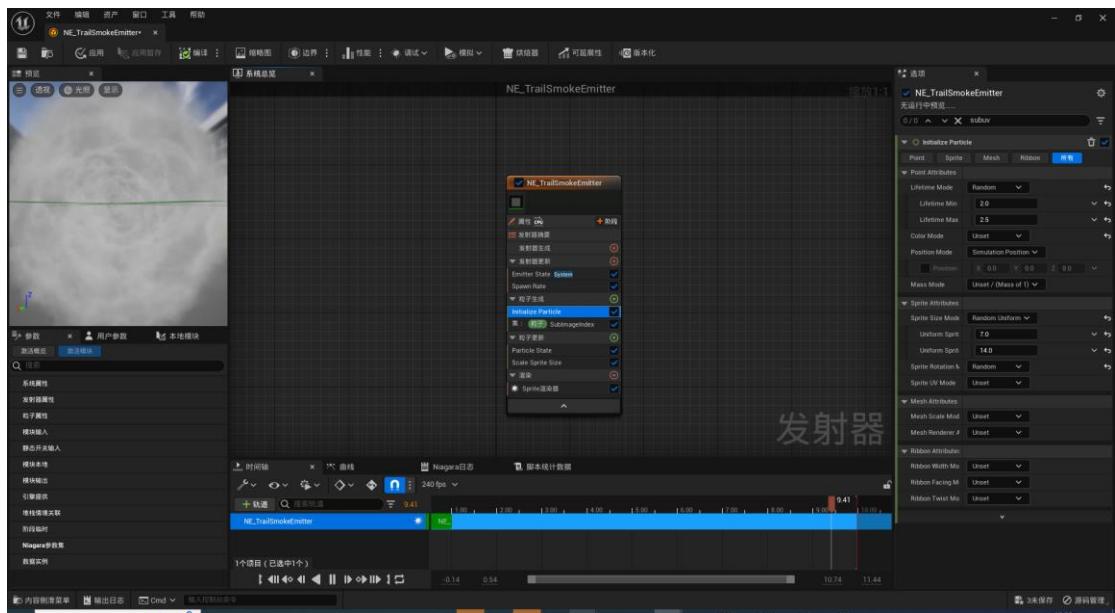


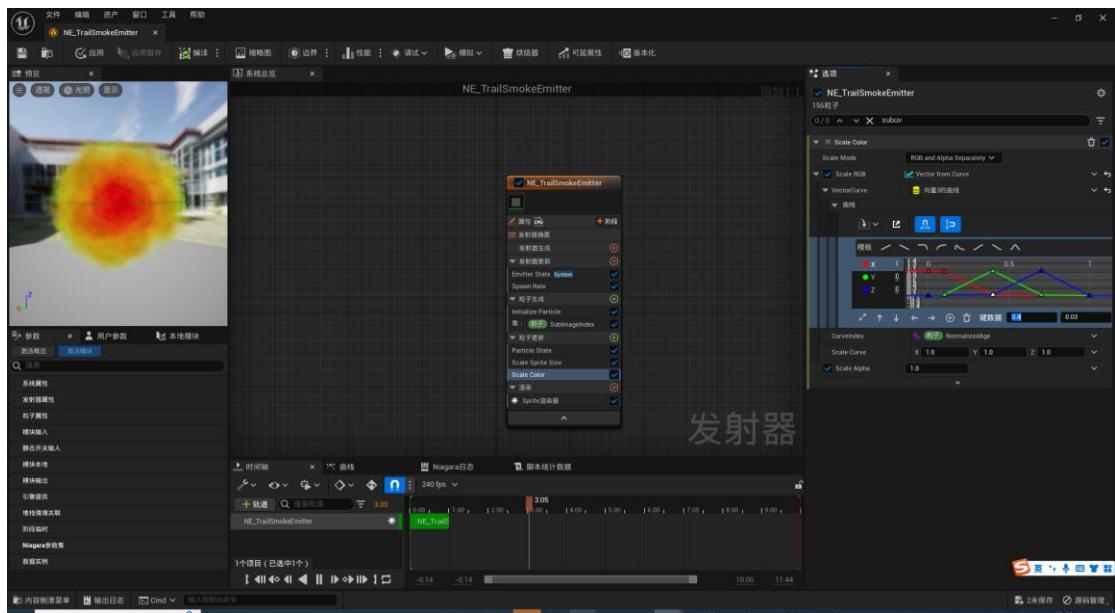














为系统选择一个起始点



来自所选发射器的新系统

选择发射器（有继承关系）和发射器模板/行为示例（无继承关系）的混合

从模板或行为示例新建系统

新系统将派生自系统模板或行为示例

拷贝现有系统

从项目内容中拷贝一个现有系统，并维持包含发射器的所有继承关系

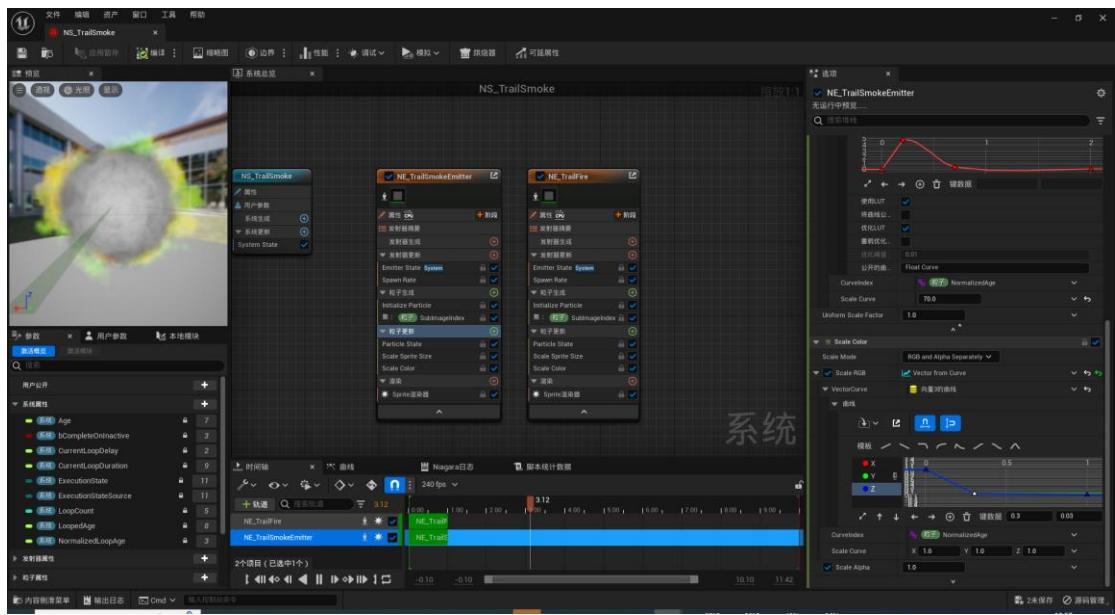
创建空白系统

创建一个不含发射器或发射器模板的空白系统

下一步 >

完成

取消



121 生成火箭尾迹

在 Blaster.Build.cs 中添加“Niagara”依赖。

由于之前实现了选做项击中不同物体时发出不同声音，所以这里实现方法和教程不同：

在 Projectile 基类中添加了 Onhit 事件之后的触发器，在设置的触发时间为 0 时直接 Destroy，而不为 0 时延迟触发 Destroy，以便火箭炮的尾气能够不要瞬间消失。

添加了一个虚函数，在多播函数 ImpactEffect 中能够添加其他类所需的效果，此外也避免了需要在子类中去使用私有变量 CollisionBox 关闭碰撞的操作：

```
void AProjectile::MulticastHitEffect_Implementation(bool bHitted, const
FVector_NetQuantize& HitPoint)
{
    if (!bHitted)
    {
        if (ImpactParticles)
        {
            UGameplayStatics::SpawnEmitterAtLocation(GetWorld(), ImpactParticles,
HitPoint);
        }
        if (ImpactSound)
        {
            UGameplayStatics::PlaySoundAtLocation(this, ImpactSound, HitPoint);
        }
    }
    else
    {
        if (HitParticles)
        {
            UGameplayStatics::SpawnEmitterAtLocation(GetWorld(), HitParticles,
HitPoint);
        }
        if (HitSound)
        {
            UGameplayStatics::PlaySoundAtLocation(this, HitSound, HitPoint);
        }
    }
}

OtherImpactEffects();

if (DestroyTime == 0.f)
{
    Destroy();
}
```

```

        }

    else
    {
        GetWorldTimerManager().SetTimer(
            DestroyTimer,
            this,
            &ThisClass::DestroyTimerFinished,
            DestroyTime
        );
    }
}

void AProjectile::OtherImpactEffects()
{
    if (CollisionBox)
    {
        CollisionBox->SetCollisionEnabled(ECollisionEnabled::NoCollision);
    }
}

```

在火箭投射物中，额外添加了隐藏 mesh 的，关闭声音播放以及关闭尾气的效果：

```

AProjectileRocket::AProjectileRocket()
{
    RocketMesh = CreateDefaultSubobject<UStaticMeshComponent>(TEXT("RocketMesh"));
    RocketMesh->SetupAttachment(RootComponent);
    RocketMesh->SetCollisionEnabled(ECollisionEnabled::NoCollision);
}

void AProjectileRocket::BeginPlay()
{
    Super::BeginPlay();

    if (TrailSystem)
    {
        TrailSystemComponent = UNiagaraFunctionLibrary::SpawnSystemAttached(
            TrailSystem,
            GetRootComponent(),
            FName(),
            GetActorLocation(),
            GetActorRotation(),
            EAttachLocation::KeepWorldPosition,
            false
        );
    }
}

```

```

    if (ProjectileLoop && LoopingSoundAttenuation)
    {
        ProjectileLoopComponent = UGameplayStatics::SpawnSoundAttached(
            ProjectileLoop,
            GetRootComponent(),
            FName(),
            GetActorLocation(),
            EAttachLocation::KeepWorldPosition,
            false,
            1. f,
            1. f,
            0. f,
            LoopingSoundAttenuation,
            (USoundConcurrency*) nullptr,
            false
        );
    }
}

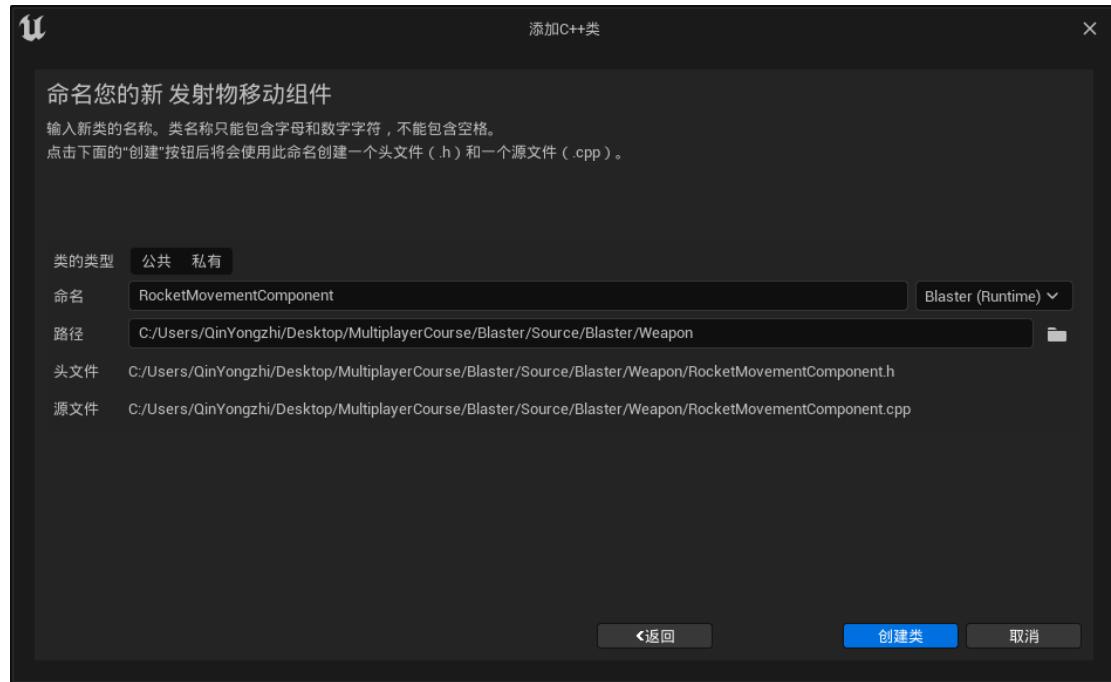
void AProjectileRocket::OtherImpactEffects()
{
    Super::OtherImpactEffects();

    if (RocketMesh)
    {
        RocketMesh->SetVisibility(false);
    }
    if (TrailSystemComponent && TrailSystemComponent->GetSystemInstance())
    {
        TrailSystemComponent->GetSystemInstance()->Deactivate();
    }
    if (ProjectileLoopComponent && ProjectileLoopComponent->IsPlaying())
    {
        ProjectileLoopComponent->Stop();
    }
}

```

122 火箭移动组件

火箭炮有可能会和我们自己发生碰撞，导致直接在我们身上爆炸，一个简单的解决办法是将 socket 往前移动，另一个方法是现在 OnHit 中检查是否碰撞到了自己，如果是，则直接 return，但是会带来一些问题，火箭会停在原地，这是因为 ProjectileMovementComponent 导致的，我们可以创建一个新的 ProjectileMovementComponent：



里面控制碰撞的函数是 HandleBlockingHit 和 HandleImpact。

我们重写两个函数，在碰撞后还能继续飞行，知道我们的 OnHit 函数将其销毁：

```
URocketMovementComponent::EHandleBlockingHitResult
URocketMovementComponent::HandleBlockingHit(const FHitResult& Hit, float TimeTick,
const FVector& MoveDelta, float& SubTickTimeRemaining)
{
    Super::HandleBlockingHit(Hit, TimeTick, MoveDelta, SubTickTimeRemaining);
    return EHandleBlockingHitResult::AdvanceNextSubstep;
}

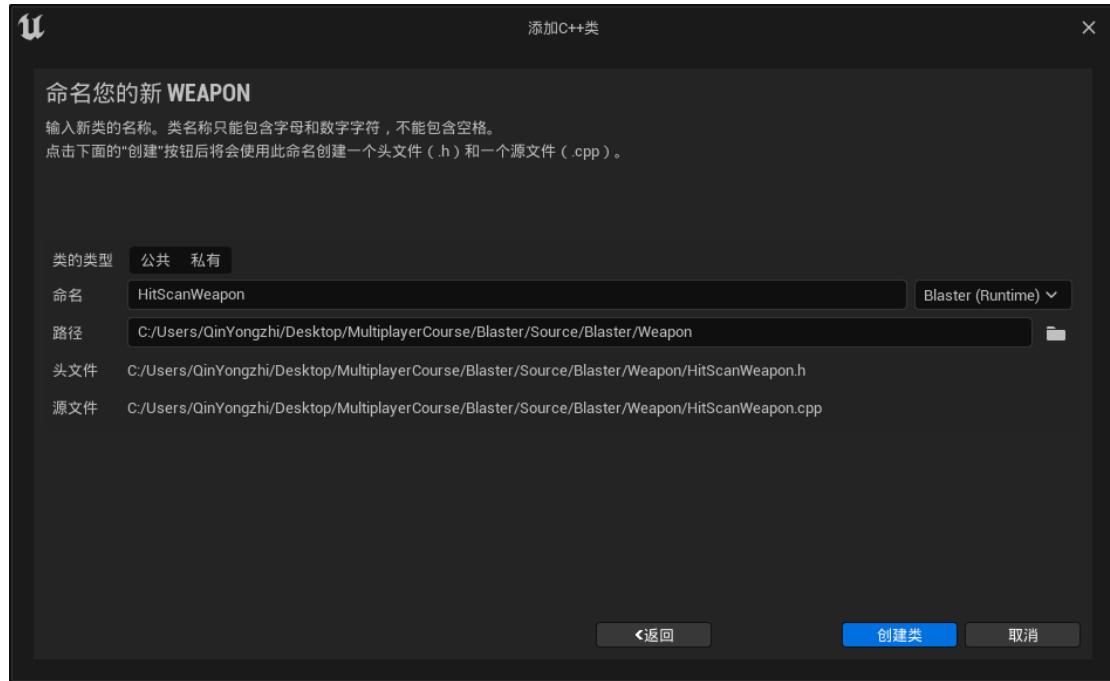
void URocketMovementComponent::HandleImpact(const FHitResult& Hit, float TimeSlice,
const FVector& MoveDelta)
{
    // Rockets should not stop; only explode when their CollisionBox detects a hit.
}
```

此外为了武器掉落后位置的正确，我们打开了武器位置的复制。

```
356     /**
357      * Handle blocking hit during simulation update. Checks that simulation remains valid after collision.
358      * If simulating then calls HandleImpact(), and returns EHandleHitWallResult::Deflect by default to enable multi-bounce and sliding support there.
359      * If no longer simulating then returns EHandleHitWallResult::Abort, which aborts attempts at further simulation.
360      */
361
362     void HandleBlockingHit(const FHitResult & Hit, float TimeTick, const FVector& MoveDelta, float& SubTickTimeRemaining)
363     {
364         if (Hit.bBlockingHit)
365         {
366             // Blocking hit occurred.
367             // Time delta of last move that resulted in the blocking hit.
368             // Movement delta for the current sub-step.
369             // How much time to continue simulating in the current sub-step, which may change as a result of this function.
370             // Initial default value is: TimeTick * (1.f - Hit.Time)
371
372             // Return result indicating how simulation should proceed.
373             // @see EHandleHitWallResult, HandleImpact()
374
375         virtual EHandleHitWallResult HandleBlockingHit(const FHitResult & Hit, float TimeTick, const FVector& MoveDelta, float& SubTickTimeRemaining) override;
376
377         /**
378          * Applies bounce logic if enabled to affect velocity upon impact (using ComputeBounceResult()),
379          * or stops the projectile if bounces are not enabled or velocity is below BounceVelocityStopSimulatingThreshold.
380          * Triggers applicable events (OnProjectileBounce).
381         */
382
383         virtual void HandleImpact(const FHitResult & Hit, float TimeSlice=0.f, const FVector& MoveDelta = FVector::ZeroVector) override;
384
385         /**
386          * Handle a blocking hit after HandleBlockingHit() returns a result indicating that deflection occurred.
387          * Default implementation increments NumBounces, checks conditions that could indicate a slide, and calls HandleSliding() if necessary.
388         */
389
390         void HandleBlockingHit(const FHitResult & Hit, float TimeTick, const FVector& MoveDelta, float& SubTickTimeRemaining)
391         {
392             if (Hit.bBlockingHit)
393             {
394                 // Blocking hit occurred. May be changed to indicate the last hit result of further movement.
395                 // Velocity at the start of the simulation update sub-step. Current Velocity may differ (as a result of a bounce).
396             }
397         }
398     }
399 }
```

123 扫描类武器

创建一个基于 Weapon 的新武器 C++：



回看我们的 Weapon 类，里面并没有太多的代码，只有最基本的，所以我们可以继承并重写新的 Fire 在 HitScanWeapon 中。

添加几个变量：

```
UPROPERTY(EditAnywhere)
float Damage = 20.f;

UPROPERTY(EditAnywhere)
class UParticleSystem* ImpactParticles;
```

重写 Fire 函数，添加一个射线检测用来射击：

```
void AHitScanWeapon::Fire(const FVector& HitTarget)
{
    Super::Fire(HitTarget);

    APawn* OwnerPawn = Cast<APawn>(GetOwner());
    if (OwnerPawn == nullptr)
    {
        return;
    }
    AController* InstigatorController = OwnerPawn->GetController();

    const auto MuzzleFlashSocket =
```

```

GetWeaponMesh()->GetSocketByName(FName("MuzzleFlash"));
    if (MuzzleFlashSocket && InstigatorController)
    {
        FTransform SocketTransform =
MuzzleFlashSocket->GetSocketTransform(GetWeaponMesh());
        FVector Start = SocketTransform.GetLocation();
        FVector End = Start + (HitTarget - Start) * 1.25f;

        FHitResult FireHit;
        UWorld* World = GetWorld();
        if (World)
        {
            World->LineTraceSingleByChannel(
                FireHit,
                Start,
                End, ECollisionChannel::ECC_Visibility
            );
            if (FireHit.bBlockingHit)
            {
                ABlasterCharacter* BlasterCharacter =
Cast<ABlasterCharacter>(FireHit.GetActor());
                if (BlasterCharacter)
                {
                    if (HasAuthority())
                    {
                        UGameplayStatics::ApplyDamage(
                            BlasterCharacter,
                            Damage,
                            InstigatorController,
                            this,
                            UDamageType::StaticClass()
                        );
                    }
                }
            }
            if (ImpactParticles)
            {
                UGameplayStatics::SpawnEmitterAtLocation(
                    World,
                    ImpactParticles,
                    FireHit.ImpactPoint,
                    FireHit.ImpactNormal.Rotation()
                );
            }
        }
    }
}

```

```
    }  
}  
  
}
```

最后和其他武器一样，把一些设置设置一下，以及 Character 中和 Combat 中的设置设置好即可。

124 射线特效

添加一个 Particle 来设置子弹的轨迹：

```
UPROPERTY(EditAnywhere)
UParticleSystem* BeamParticles;
```

修正 Fire 函数：

```
void AHitScanWeapon::Fire(const FVector& HitTarget)
{
    Super::Fire(HitTarget);

    APawn* OwnerPawn = Cast<APawn>(GetOwner());
    if (OwnerPawn == nullptr)
    {
        return;
    }
    AController* InstigatorController = OwnerPawn->GetController();

    const auto MuzzleFlashSocket =
        GetWeaponMesh()->GetSocketByName(FName("MuzzleFlash"));
    if (MuzzleFlashSocket)
    {
        FTransform SocketTransform =
            MuzzleFlashSocket->GetSocketTransform(GetWeaponMesh());
        FVector Start = SocketTransform.GetLocation();
        FVector End = Start + (HitTarget - Start) * 1.25f;

        FHitResult FireHit;
        UWorld* World = GetWorld();
        if (World)
        {
            World->LineTraceSingleByChannel(
                FireHit,
                Start,
                End, ECollisionChannel::ECC_Visibility
            );
            FVector BeamEnd = End;
            if (FireHit.bBlockingHit)
            {
                ABlasterCharacter* BlasterCharacter =
                    Cast<ABlasterCharacter>(FireHit.GetActor());

                if (BlasterCharacter && HasAuthority() && InstigatorController)
                {

```

```

    UGameplayStatics::ApplyDamage(
        BlasterCharacter,
        Damage,
        InstigatorController,
        this,
        UDamageType::StaticClass()
    );
}

if (ImpactParticles)
{
    UGameplayStatics::SpawnEmitterAtLocation(
        World,
        ImpactParticles,
        FireHit.ImpactPoint,
        FireHit.ImpactNormal.Rotation()
    );
}
BeamEnd = FireHit.ImpactPoint;
}

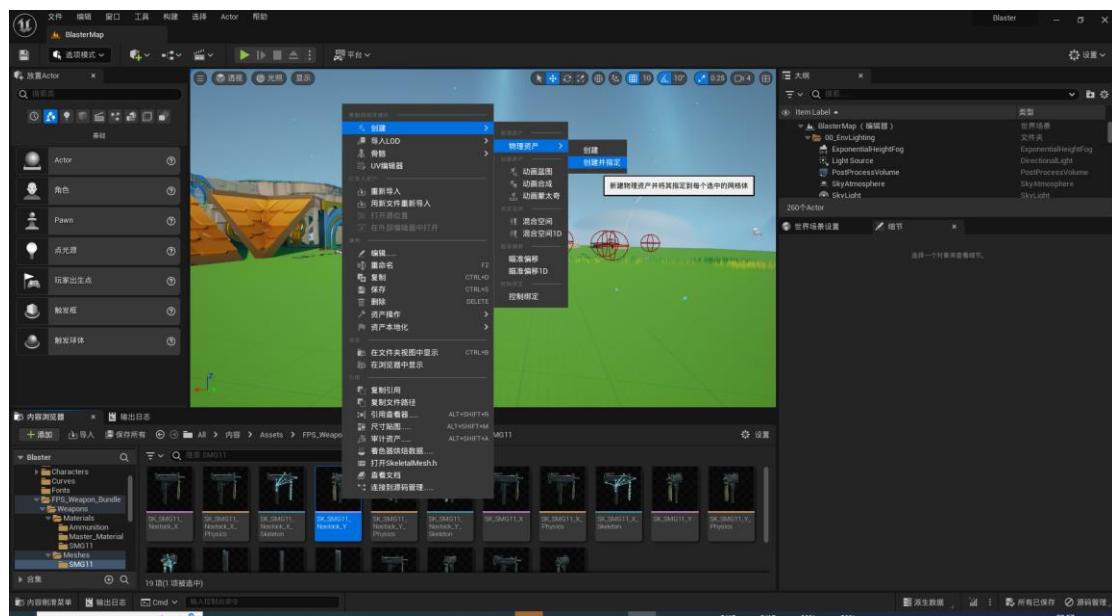
if (BeamParticles)
{
    UParticleSystemComponent* Beam =
        UGameplayStatics::SpawnEmitterAtLocation(
            World,
            BeamParticles,
            SocketTransform
        );
    if (Beam)
    {
        Beam->SetVectorParameter(FName("Target"), BeamEnd);
    }
}
}
}
}

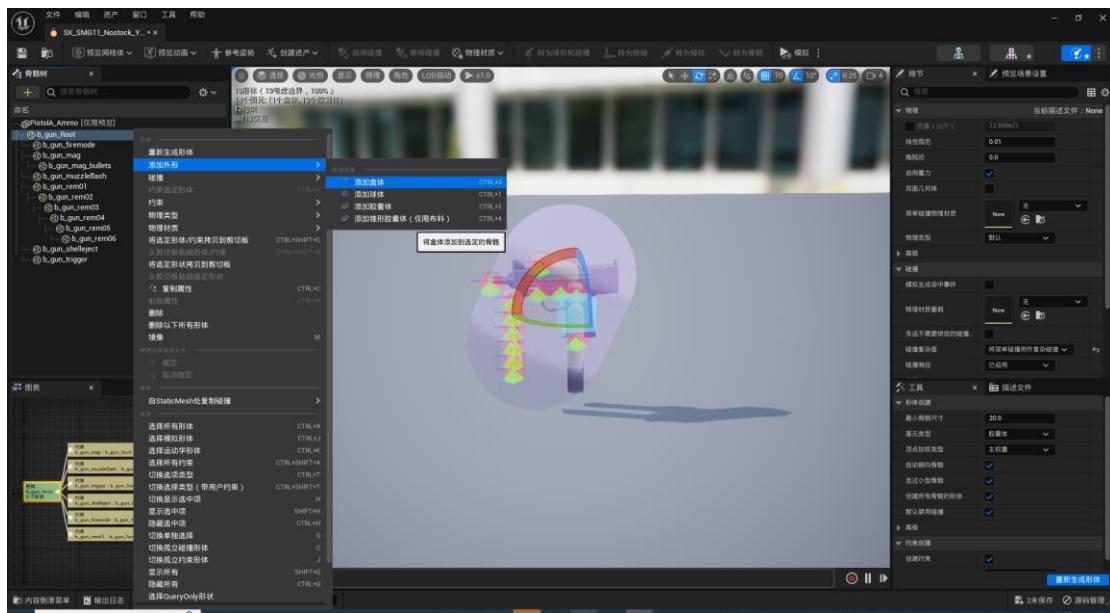
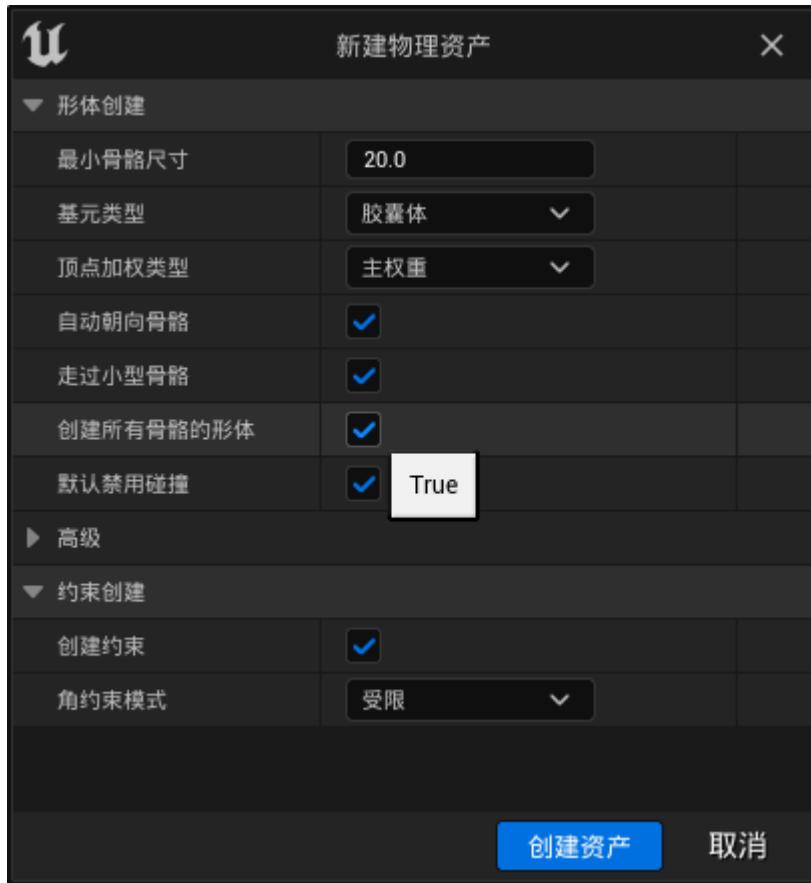
```

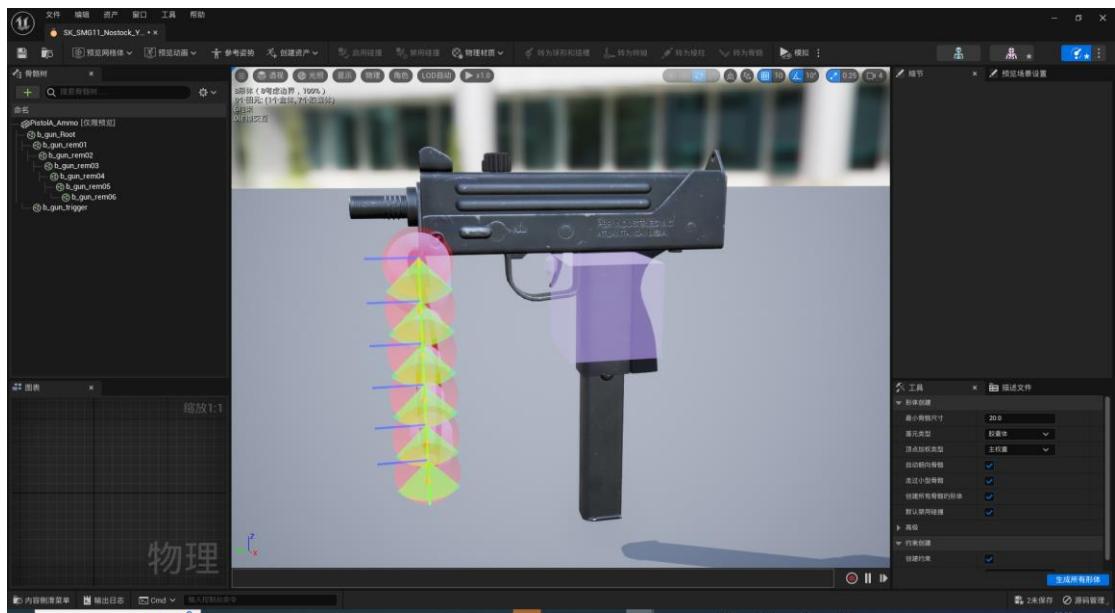
125 冲锋枪

创建一个冲锋枪类并添加一系列属性和资产。

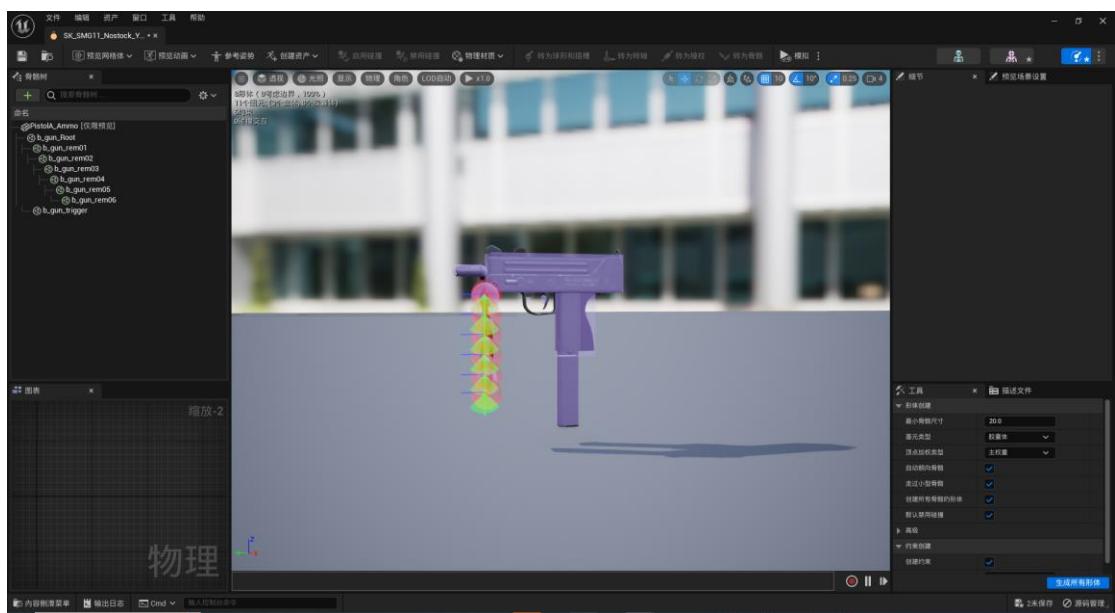
126 冲锋枪带应用物理效果

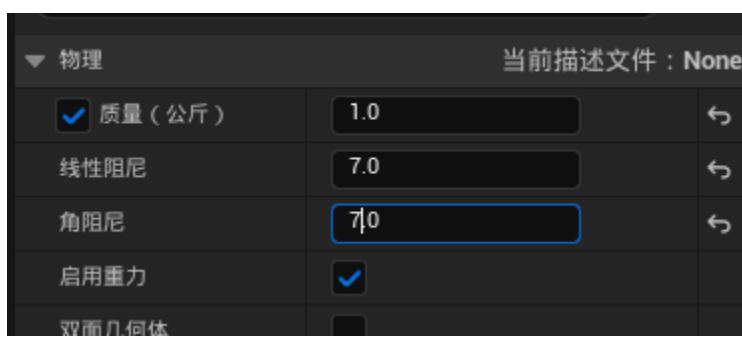
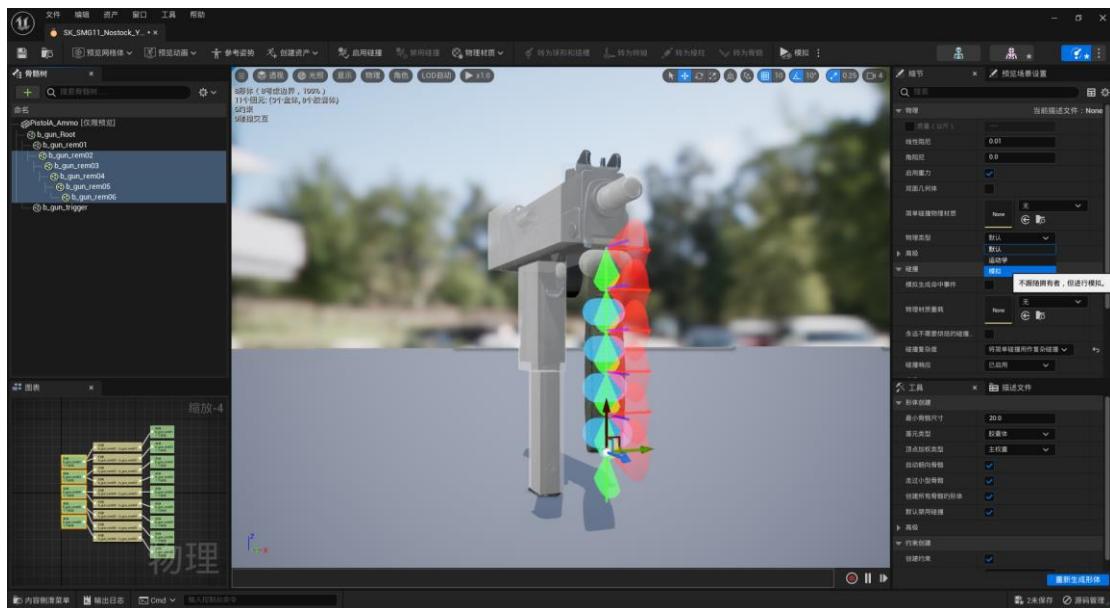






删除多余胶囊体。





在 Combat 中的 EquipWeapon 和 WeaponState 的复制函数中使用如下代码，开启物理模拟，并且丢弃的时候关闭：

```

switch (WeaponState)
{
    case EWeaponState::EWS_Equipped:
        ShowPickupWidget(false);
        AreaSphere->SetCollisionEnabled(ECollisionEnabled::NoCollision);
        WeaponMesh->SetSimulatePhysics(false);
        WeaponMesh->SetEnableGravity(false);
        WeaponMesh->SetCollisionEnabled(ECollisionEnabled::NoCollision);
        if (WeaponType == EWeaponType::EWT_SubmachineGun)
        {
            WeaponMesh->SetCollisionEnabled(ECollisionEnabled::QueryAndPhysics);
            WeaponMesh->SetEnableGravity(true);

            WeaponMesh->SetCollisionResponseToAllChannels(ECollisionResponse::ECR_Ignore);
        }
        break;
    case EWeaponState::EWS_Dropped:
        if (HasAuthority())

```

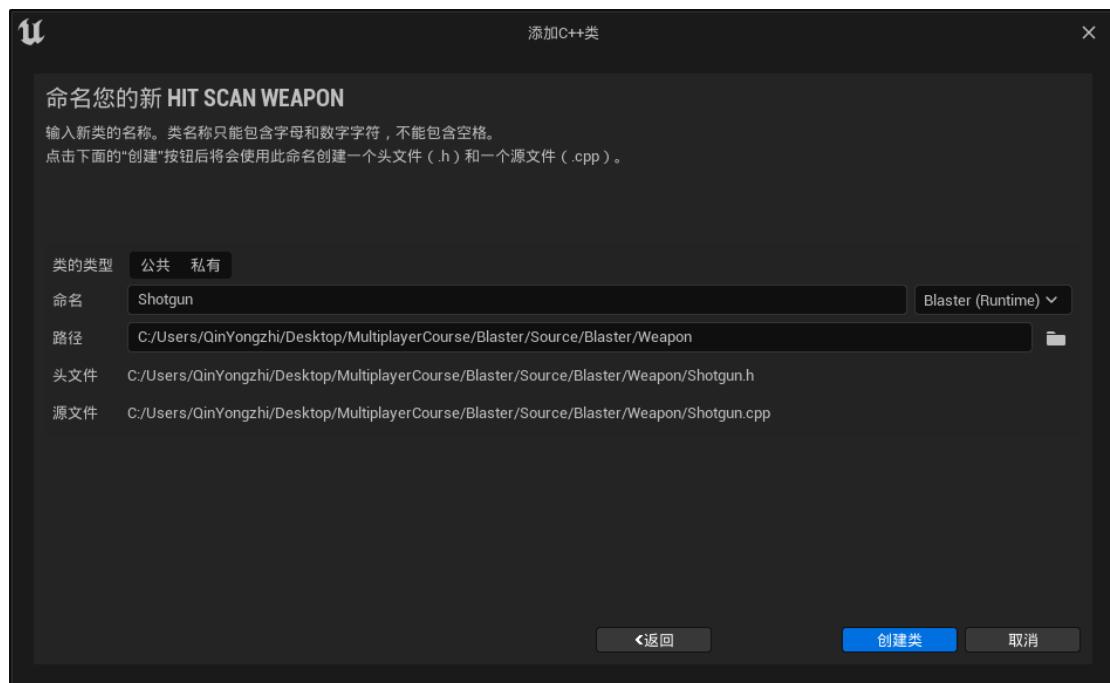
```
{  
    AreaSphere->SetCollisionEnabled(ECollisionEnabled::QueryOnly);  
}  
WeaponMesh->SetSimulatePhysics(true);  
WeaponMesh->SetEnableGravity(true);  
WeaponMesh->SetCollisionEnabled(ECollisionEnabled::QueryAndPhysics);  
WeaponMesh->SetCollisionResponseToAllChannels(ECollisionResponse::ECR_Block);  
WeaponMesh->SetCollisionResponseToChannel(ECollisionChannel::ECC_Pawn,  
ECollisionResponse::ECR_Ignore);  
WeaponMesh->SetCollisionResponseToChannel(ECollisionChannel::ECC_Camera,  
ECollisionResponse::ECR_Ignore);  
    break;  
}
```

修复

在本章由于长时间没有修复之前服务端和客户端 MuzzleFlash 的 Socket 位置不一致的问题，我直接修改了开火的逻辑，在 RPC 请求的时候直接传入当前客户端的 Socket 位置，以确保开火的顺利进行，如果后续有其他的解决方案我会再更新。

修复问题导致了如果装备的武器没有名为 MuzzleFlash 的 Socket 将无法再进行开火。

127 霰弹枪



创建一个 HitScanGun 的子类，重写 Fire 函数，但是不使用 Super::Fire()，而是直接使用 AWeapon::Fire() 播放效果。

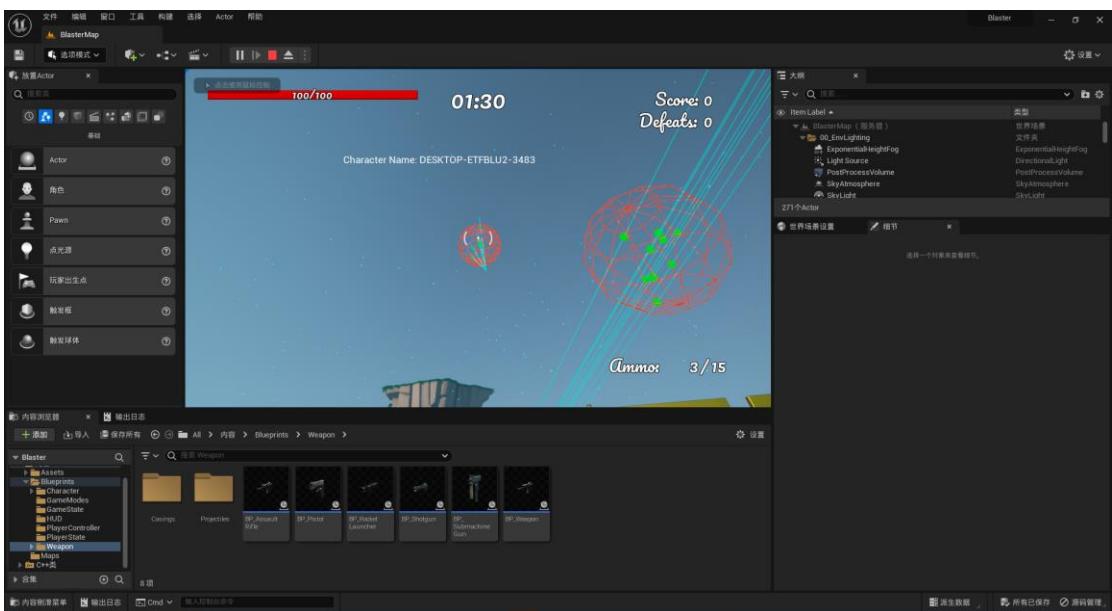
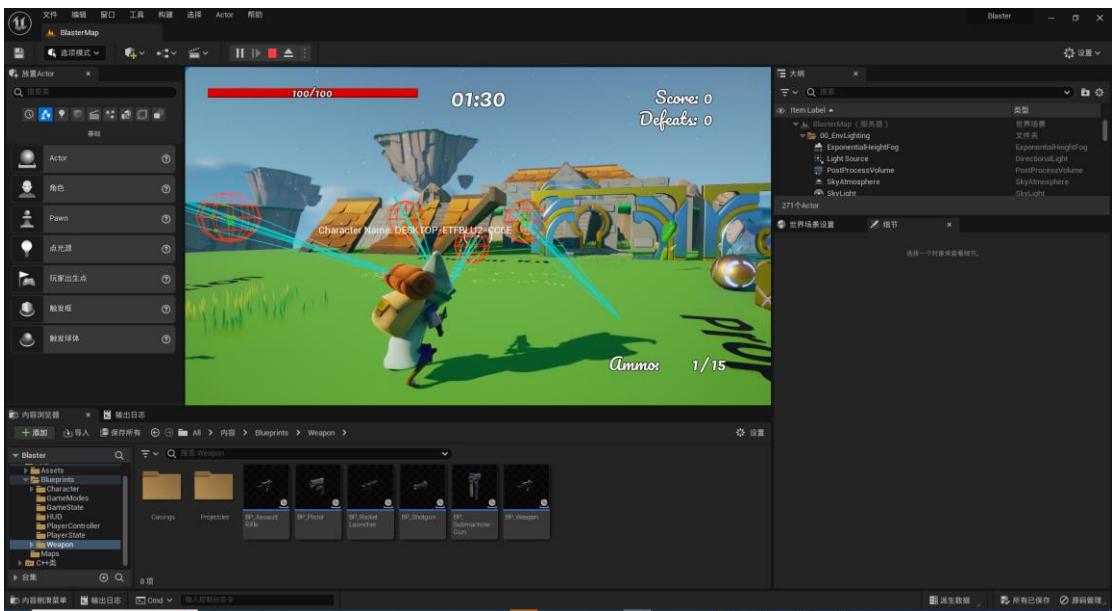
在 HitScanWeapon 中添加一个实现散射的函数：

```
FVector AHitScanWeapon::TraceEndWithScatter(const FVector& TraceStart, const FVector&
HitTarget)
{
    FVector ToTargetNormalized = (HitTarget - TraceStart).GetSafeNormal();
    FVector SphereCenter = TraceStart + ToTargetNormalized * DistanceToSphere;
    FVector RandVec = UKismetMathLibrary::RandomUnitVector() * FMath::FRandRange(0.f,
SphereRadius);
    FVector EndLoc = SphereCenter + RandVec;
    FVector ToEndLoc = EndLoc - TraceStart;

    DrawDebugSphere(GetWorld(), SphereCenter, SphereRadius, 12, FColor::Red, true);
    DrawDebugSphere(GetWorld(), EndLoc, 4.f, 12, FColor::Green, true);
    DrawDebugLine(GetWorld(),
        TraceStart,
        FVector(TraceStart + ToEndLoc * TRACE_LENGTH / ToEndLoc.Size()),
        FColor::Cyan,
        true
    );

    return FVector(TraceStart + ToEndLoc * TRACE_LENGTH / ToEndLoc.Size());
}
```

实现的效果如下：



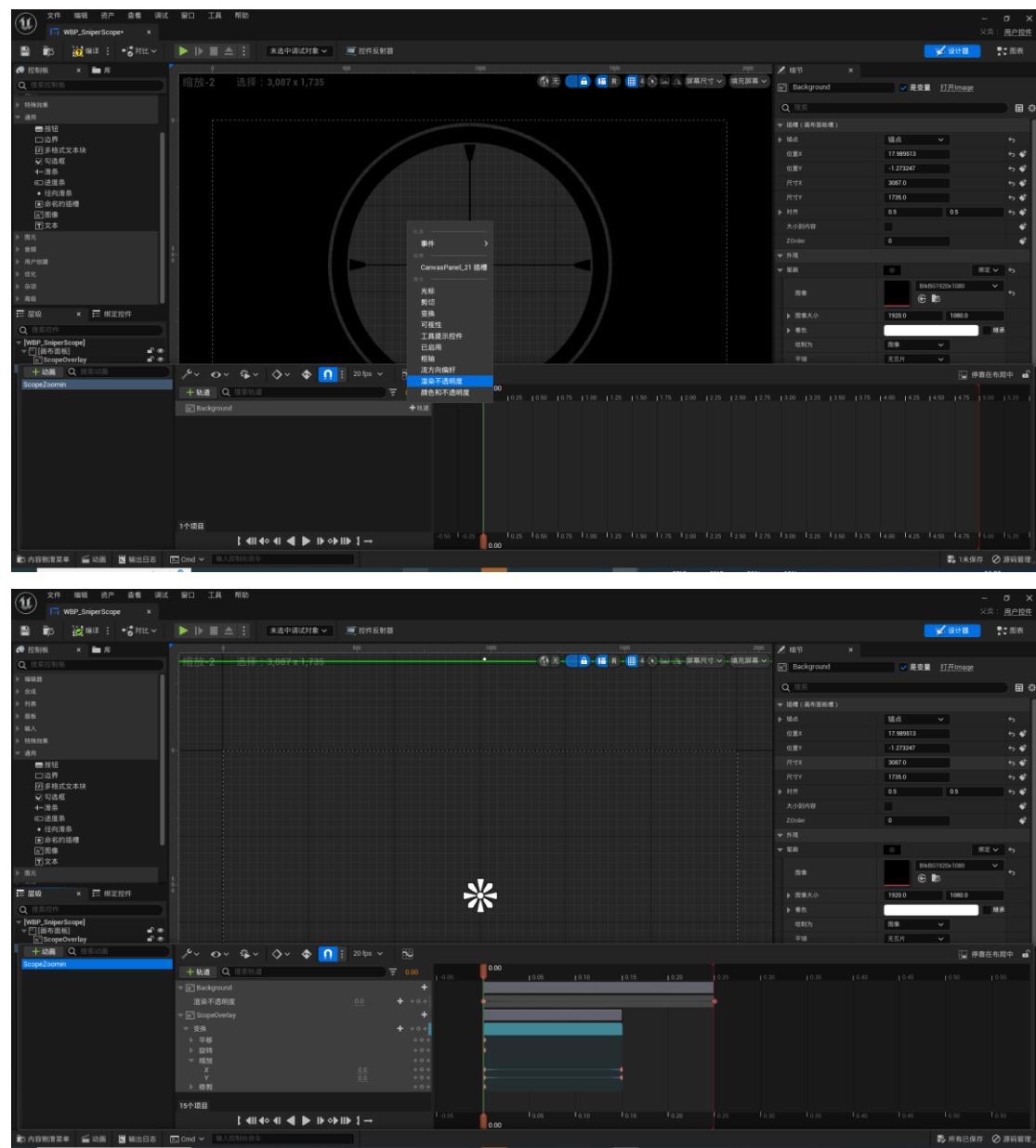
128 武器散射

修改了之前的代码结构

129 狙击枪

创建一把狙击枪。

130 狙击瞄准镜（蓝图可执行事件）



创建一个蓝图控件，并添加瞄准镜的淡入动画。

在 Character 中添加一个蓝图可执行事件：

```
UFUNCTION(BlueprintImplementableEvent)
void ShowSniperScopeWidget(bool bShowScope);
```

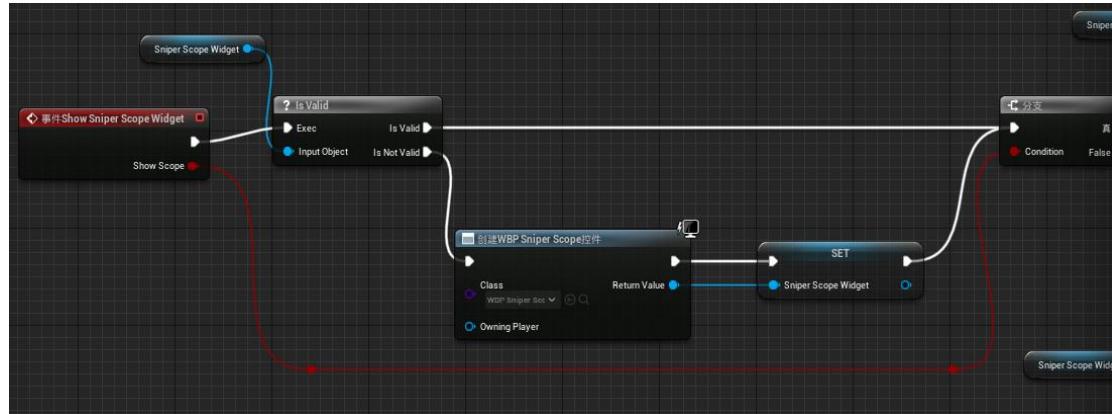
修改 SetAiming 函数：

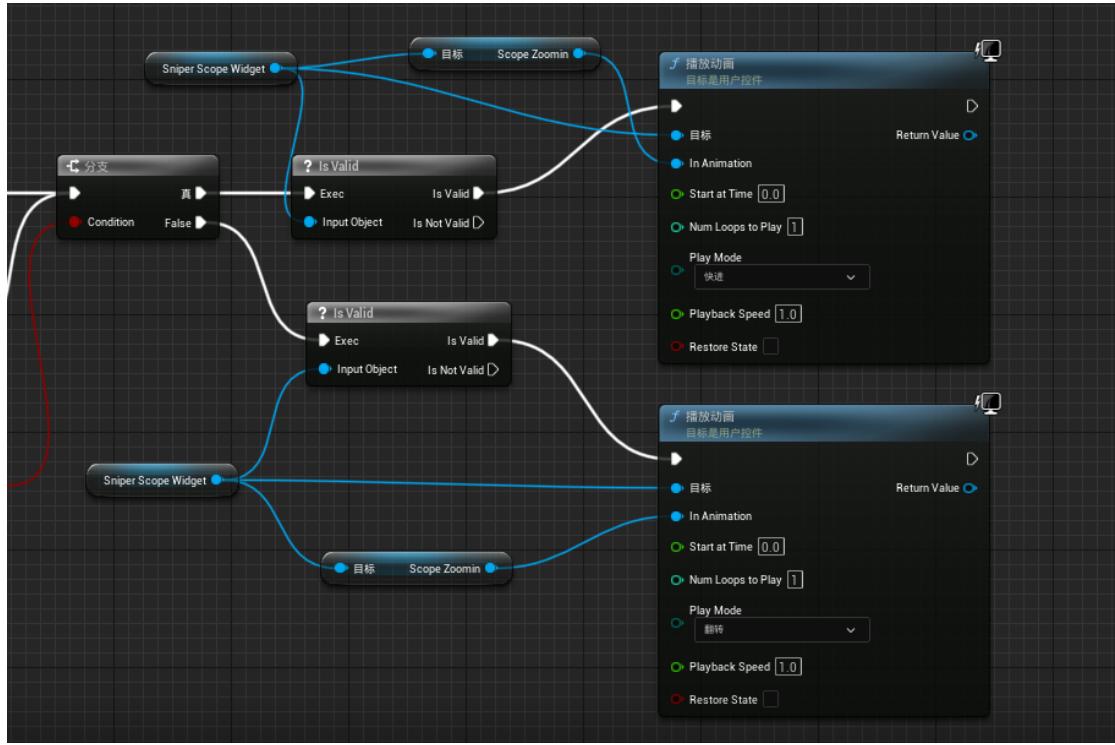
```
void UCombatComponent::SetAiming(bool bIsAiming)
{
    if (Character == nullptr || EquippedWeapon == nullptr)
    {
        return;
    }

    bAiming = bIsAiming;
    ServerSetAiming(bIsAiming);
    if (Character)
    {
        Character->GetCharacterMovement()->MaxWalkSpeed = bIsAiming ? AimWalkSpeed : BaseWalkSpeed;
    }

    if (Character->IsLocallyControlled() && EquippedWeapon->GetWeaponType() ==
EWeaponType::EWT_SniperRifle)
    {
        Character->ShowSniperScopeWidget(bIsAiming);
    }
}
```

在蓝图中我们可以实现这个函数：

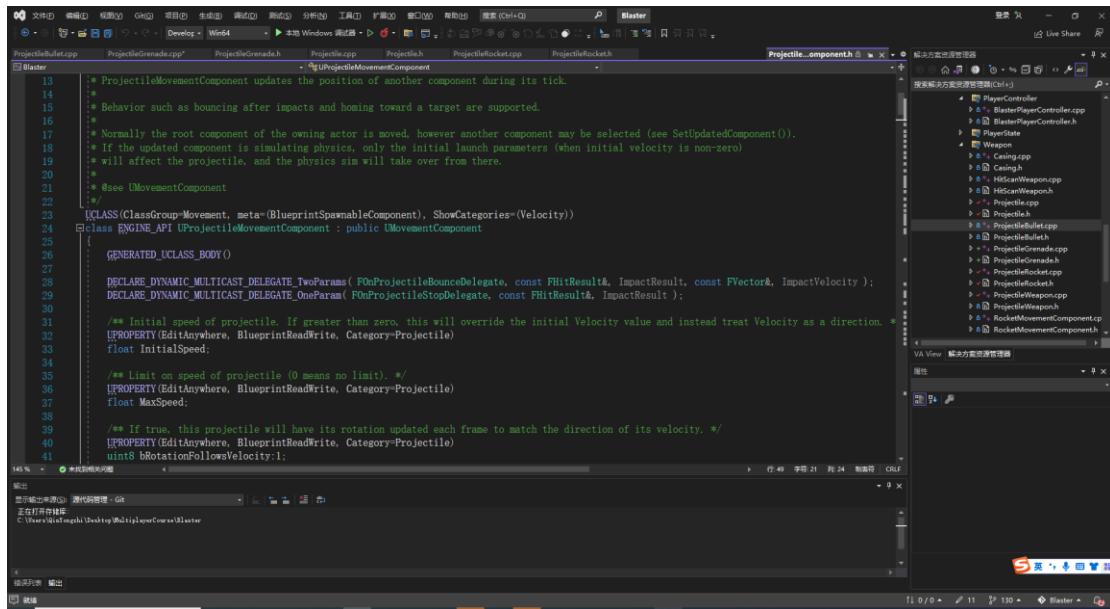




131 榴弹发射器

创建一个榴弹发射器。

132 榴弹投射物



The screenshot shows the Unreal Engine 4 Editor interface with the code editor open. The file is `ProjectileMovementComponent.h` located in the `Blaster` project. The code defines a class `UProjectileMovementComponent` that inherits from `UMovementComponent`. It includes declarations for delegates `FOnProjectileBounceDelegate` and `FOnProjectileStopDelegate`, properties for initial speed (`InitialSpeed`), maximum speed (`MaxSpeed`), and rotation follow velocity (`bRotationFollowsVelocity`), and a boolean property `bUseInitialVelocity`. The code also contains comments explaining the behavior of the component, such as handling impacts and homing towards targets.

```
13 // ProjectileMovementComponent updates the position of another component during its tick.
14 // Behavior such as bouncing after impacts and homing toward a target are supported.
15 //
16 // Normally the root component of the owning actor is moved, however another component may be selected (see SetUpdatedComponent()).
17 // If the updated component is simulating physics, only the initial launch parameters (when initial velocity is non-zero)
18 // will affect the projectile, and the physics sim will take over from there.
19 //
20 // @see UMovementComponent
21 //
22 UCLASS(ClassGroup=Movement, meta=(BlueprintSpawnableComponent), ShowCategories=(Velocity))
23 DECLARE_DYNAMIC_MULTICAST_DELEGATE_TwoParams( FOnProjectileBounceDelegate, const FHitResult, ImpactResult, const FVector&, ImpactVelocity );
24 DECLARE_DYNAMIC_MULTICAST_DELEGATE_OneParam( FOnProjectileStopDelegate, const FHitResult, ImpactResult );
25
26 GENERATED_BODY()
27
28 // Initial speed of projectile. If greater than zero, this will override the initial Velocity value and instead treat Velocity as a direction.
29 UPROPERTY(EditAnywhere, BlueprintReadWrite, Category=Projectile)
30 float InitialSpeed;
31
32 // Limit on speed of projectile (0 means no limit). */
33 UPROPERTY(EditAnywhere, BlueprintReadWrite, Category=Projectile)
34 float MaxSpeed;
35
36 /** If true, this projectile will have its rotation updated each frame to match the direction of its velocity. */
37 UPROPERTY(EditAnywhere, BlueprintReadWrite, Category=Projectile)
38 uint8 bRotationFollowsVelocity:1;
39
40
41
```

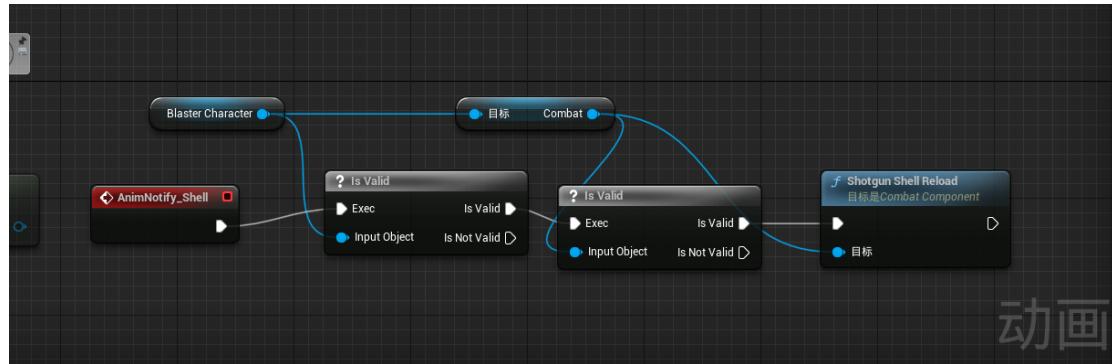
可以看到 `projectilemovementcomponent` 中有一个弹跳的委托, 我们可以在这个委托上添加弹跳的声音, 重写 `BeginPlay` 函数, 不需要绑定 `Onhit` 函数, 直接开始 `Destroy` 的计时器, 重写 `Destroy` 函数应用伤害。

133 装弹动画

为各个武器创建蒙太奇，并在 Character 中设置。

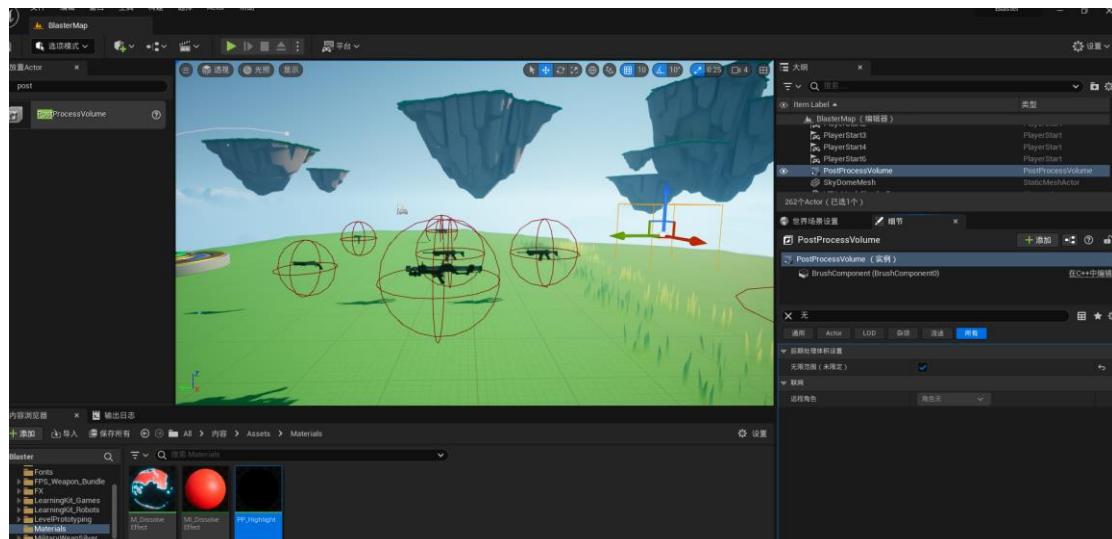
134 霰弹枪装填

霰弹枪装填需要在每次装弹后调用一个蒙太奇通知，让子弹加一，而且要检查是否装满，如果装满则调到蒙太奇结尾。

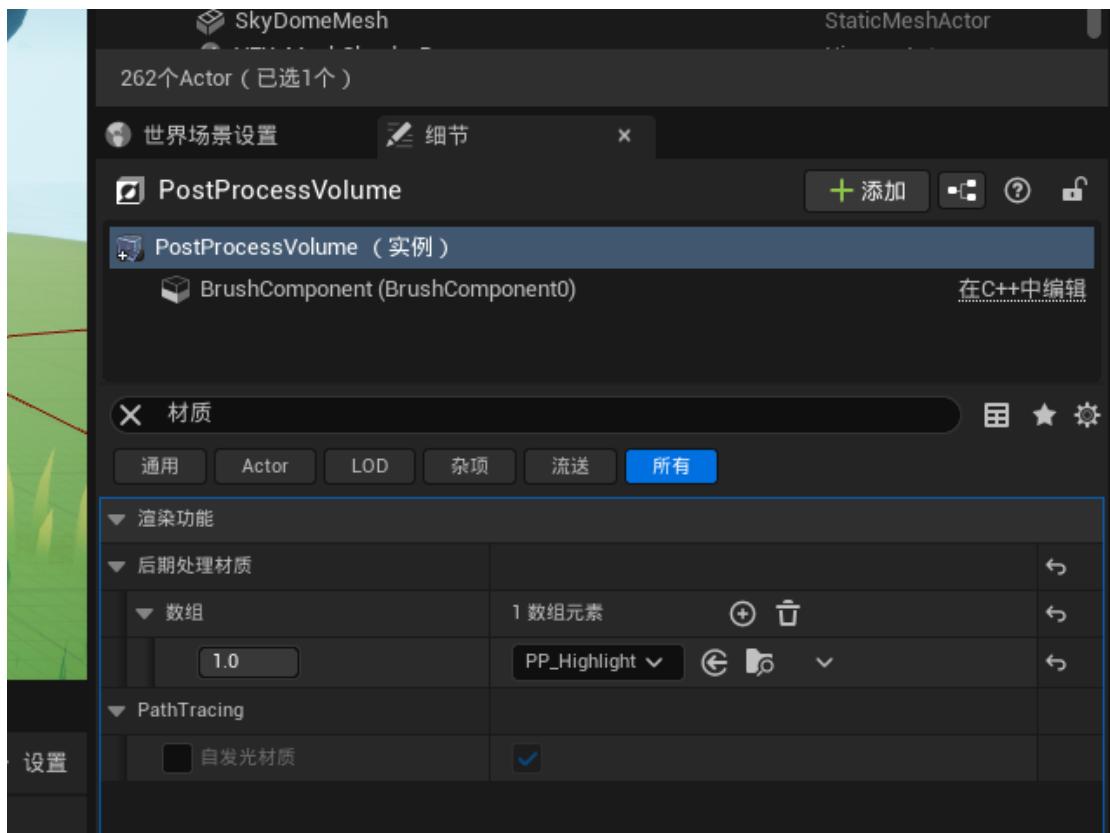


创建一个蓝图可调用函数，并在动画蓝图中通过通知调用。

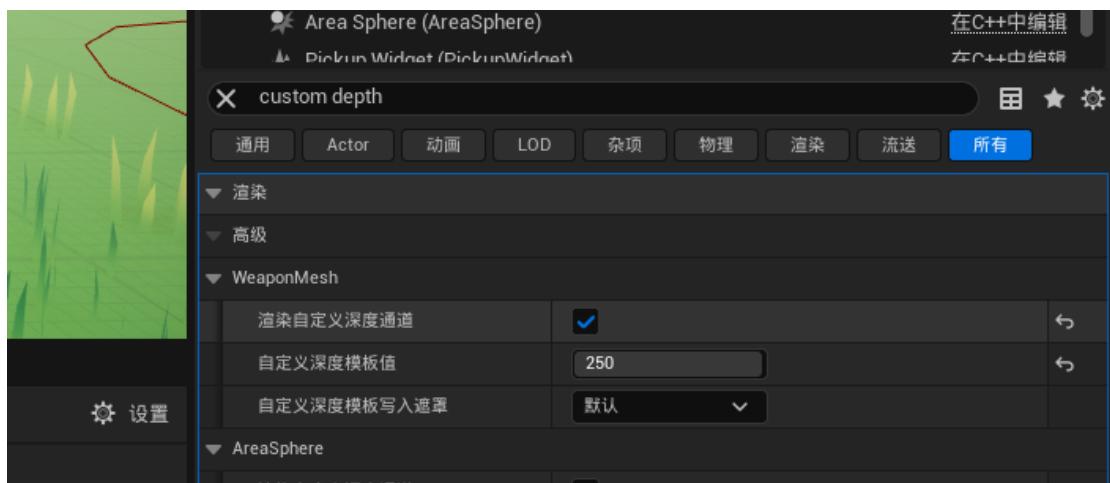
135 武器外轮廓效果



在世界中放置一个后期处理体积



添加一个后期处理材质。



选择渲染自定义深度通道，设置自定义模板值 250 可以看到一个紫色的轮廓。

使用如下代码可以在 C++ 中实现控制：

```
void AWeapon::EnableCustomDepth(bool bEnable)
{
    if (WeaponMesh)
    {
        WeaponMesh->SetRenderCustomDepth(bEnable);
    }
}
```

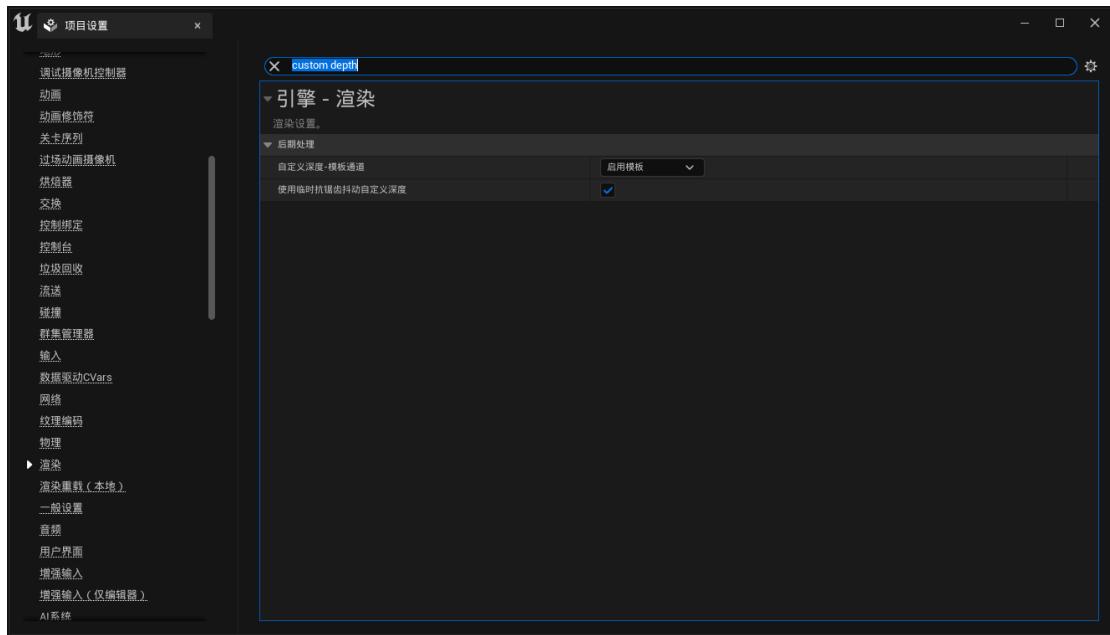
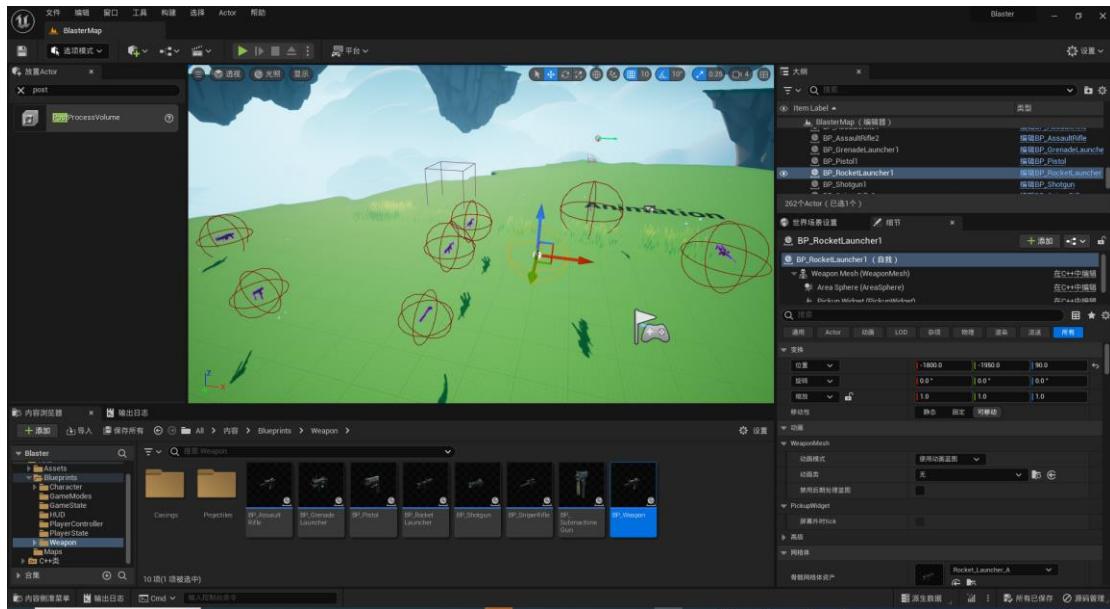
在 Weapon 的构造函数中添加如下语句：

```
WeaponMesh->SetCustomDepthStencilValue(CUSTOM_DEPTH_PURPLE);
```

```
WeaponMesh->MarkRenderStateDirty();
EnableCustomDepth(true);
```

```
#define CUSTOM_DEPTH_PURPLE 250
#define CUSTOM_DEPTH_BLUE 251
#define CUSTOM_DEPTH_TAN 252
```

现在，所有武器都添加上了紫色轮廓：



ISSUE

`void ABlasterPlayerController::HandleColldown()` 中，在中途加入游戏时调用创建 widget 的函数无法创建出 Announcement 的 widget。

136 丢手雷时的武器附着点

使用两个函数，在丢手雷的时候将武器附着到左手，在 finish 函数中添加回右手。

```
void UCombatComponent::AttachActorToRightHand(AActor* ActorToAttach)
{
    if (Character == nullptr || Character->GetMesh() == nullptr || ActorToAttach ==
        nullptr)
    {
        return;
    }

    const USkeletalMeshSocket* HandSocket =
Character->GetMesh()->GetSocketByName(FName("RightHandSocket"));

    if (HandSocket)
    {
        HandSocket->AttachActor(ActorToAttach, Character->GetMesh());
    }
}

void UCombatComponent::AttachActorToLeftHand(AActor* ActorToAttach)
{
    if (Character == nullptr || Character->GetMesh() == nullptr || ActorToAttach ==
        nullptr || EquippedWeapon == nullptr)
    {
        return;
    }

    bool bUsePistolSocket =
        EquippedWeapon->GetWeaponType() == EWeaponType::EWT_Pistol ||
        EquippedWeapon->GetWeaponType() == EWeaponType::EWT_SubmachineGun;
    FName SocketName = bUsePistolSocket ? FName("PistolSocket") :
        FName("LeftHandSocket");
    const USkeletalMeshSocket* HandSocket =
Character->GetMesh()->GetSocketByName(SocketName);

    if (HandSocket)
    {
        HandSocket->AttachActor(ActorToAttach, Character->GetMesh());
    }
}
```

137 手雷资产

添加一个手雷资产，并为其创建 Socket。

138 投掷手雷

在蒙太奇冲设置一个通知，用于投掷手雷时隐藏手雷。

139 生成手雷

按照类似榴弹发射器开火的方式生成手雷即可，但是 HitTarget 并没有更新到服务端，所以还需要做进一步的调整。

140 多人模式中的手雷

之前的手雷在服务端由于 hitTarget 没有更新所以不能正确的扔出，我们可以简单的通过一个 RPC 实现我们想要的结果。

141 手雷 HUD

按部就班的添加手雷的 HUD，并在 controller 中更新。

修复了一个问题：人物淘汰后会受到伤害，从而被淘汰多次，可以通过在 ReceiveDamage 中添加红色代码修复：

```
void ABlasterCharacter::ReceiveDamage(AActor* DamagedActor, float Damage, const UDamageType* DamageType, AController* InstigatorController, AActor* DamageCauser)
{
    if (bElimmed)
    {
        return;
    }
    Health = FMath::Clamp(Health - Damage, 0.0f, MaxHealth);
    UpdateHUDHealth();
    PlayHitReactMontage();

    if (Health == 0.0f)
    {
        ABlasterGameMode* BlasterGameMode =
        GetWorld()->GetAuthGameMode<ABlasterGameMode>();
        if (BlasterGameMode)
        {
            BlasterGameMode->HandleCharacterDeath(DamagedActor);
        }
    }
}
```

```
BlasterPlayerController = BlasterPlayerController == nullptr ?  
Cast<ABlasterPlayerController>(Controller) : BlasterPlayerController;  
ABlasterPlayerController* AttackerController =  
Cast<ABlasterPlayerController>(InstigatorController);  
BlasterGameMode->PlayerEliminated(this, BlasterPlayerController,  
AttackerController);  
}  
}  
}
```

这张内容很长，而且重复度很高，所以没有耐心一点一点写笔记了，直接看源码可能也更方便。这张过后课程内容已经结束了四分之三，学了 20 天了，终于快结束了。

选做项

Optional Challenge: Create a new Weapon!

We've created lots of different weapon types. Try creating a new weapon type! Be creative!

We'd love to see your custom weapon. Show it off in the 😍 | share-your-work channel in the Druid Mechanics Discord!

Q

是否 ThisClass 在子类中会动态调用子类的函数。

拾取物

142 拾取物类

说实话学到这个地方，直接看代码也知道在做什么了，所以不废话直接上代码：

```
APickup::APickup()
{
    PrimaryActorTick.bCanEverTick = true;
    bReplicates = true;

    RootComponent = CreateDefaultSubobject<USceneComponent>(TEXT("Root"));

    OverlapSphere = CreateDefaultSubobject<USphereComponent>(TEXT("OverlapSphere"));
    OverlapSphere->SetupAttachment(RootComponent);
    OverlapSphere->SetSphereRadius(150. f);
    OverlapSphere->SetCollisionEnabled(ECollisionEnabled::QueryOnly);
    OverlapSphere->SetCollisionResponseToAllChannels(ECollisionResponse::ECR_Ignore);
    OverlapSphere->SetCollisionResponseToChannel(ECollisionChannel::ECC_Pawn,
ECollisionResponse::ECR_Overlap);

    PickMesh = CreateDefaultSubobject<UStaticMeshComponent>(TEXT("Pickup Mesh"));
    PickMesh->SetupAttachment(OverlapSphere);
    PickMesh->SetCollisionEnabled(ECollisionEnabled::NoCollision);
}

void APickup::BeginPlay()
{
    Super::BeginPlay();

    if (HasAuthority())
    {
        OverlapSphere->OnComponentBeginOverlap.AddDynamic(this,
&APickup::OnSphereOverlap);
    }
}

void APickup::OnSphereOverlap(UPrimitiveComponent* OverlappedComponent, AActor*
OtherActor, UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromSweep,
const FHitResult& SweepResult)
{
}
```

```
void APickup::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);
}

void APickup::Destroyed()
{
    Super::Destroyed();

    if (PickupSound)
    {
        UGameplayStatics::PlaySoundAtLocation(
            this,
            PickupSound,
            GetActorLocation()
        );
    }
}
```

143 弹药拾取

创建一个 AmmoPickup 类，为每种弹药创建蓝图：

```
APickup::APickup()
{
    PrimaryActorTick.bCanEverTick = true;
    bReplicates = true;

    RootComponent = CreateDefaultSubobject<USceneComponent>(TEXT("Root"));

    OverlapSphere = CreateDefaultSubobject<USphereComponent>(TEXT("OverlapSphere"));
    OverlapSphere->SetupAttachment(RootComponent);
    OverlapSphere->SetSphereRadius(150. f);
    OverlapSphere->SetCollisionEnabled(ECollisionEnabled::QueryOnly);
    OverlapSphere->SetCollisionResponseToAllChannels(ECollisionResponse::ECR_Ignore);
    OverlapSphere->SetCollisionResponseToChannel(ECollisionChannel::ECC_Pawn,
ECollisionResponse::ECR_Overlap);
    OverlapSphere->AddLocalOffset(FVector(0. f, 0. f, 85. f));

    PickupMesh = CreateDefaultSubobject<UStaticMeshComponent>(TEXT("Pickup Mesh"));
    PickupMesh->SetupAttachment(OverlapSphere);
    PickupMesh->SetCollisionEnabled(ECollisionEnabled::NoCollision);
    PickupMesh->SetRelativeScale3D(FVector(5. f, 5. f, 5. f));
    PickupMesh->SetRenderCustomDepth(true);
    PickupMesh->SetCustomDepthStencilValue(CUSTOM_DEPTH_PURPLE);
}

void APickup::BeginPlay()
{
    Super::BeginPlay();

    if (HasAuthority())
    {
        OverlapSphere->OnComponentBeginOverlap.AddDynamic(this,
&APickup::OnSphereOverlap);
    }
}

void APickup::OnSphereOverlap(UPrimitiveComponent* OverlappedComponent, AActor*
OtherActor, UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromSweep,
const FHitResult& SweepResult)
{
}
```

```

void APickup::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);

    if (PickupMesh)
    {
        PickupMesh->AddLocalRotation(FRotator(0.f, BaseTurnRate * DeltaTime, 0.f));
    }
}

void APickup::Destroyed()
{
    Super::Destroyed();

    if (PickupSound)
    {
        UGameplayStatics::PlaySoundAtLocation(
            this,
            PickupSound,
            GetActorLocation()
        );
    }
}

```

在 Combat 中需要一个函数来执行弹药装填:

```

void UCombatComponent::PickupAmmo(EWeaponType WeaponType, int32 AmmoAmount)
{
    if (CarriedAmmoMap.Contains(WeaponType) && MaxCarriedAmmoMap.Contains(WeaponType))
    {
        CarriedAmmoMap[WeaponType] = FMath::Clamp(CarriedAmmoMap[WeaponType] +
            AmmoAmount, 0, MaxCarriedAmmoMap[WeaponType]);
        UpdateCarriedAmmo();
    }

    if (EquippedWeapon && EquippedWeapon->IsEmpty() && EquippedWeapon->GetWeaponType() ==
        WeaponType)
    {
        Reload();
    }
}

```

144 BUFF 组件

添加一个 BUFFcomponent 在角色中：

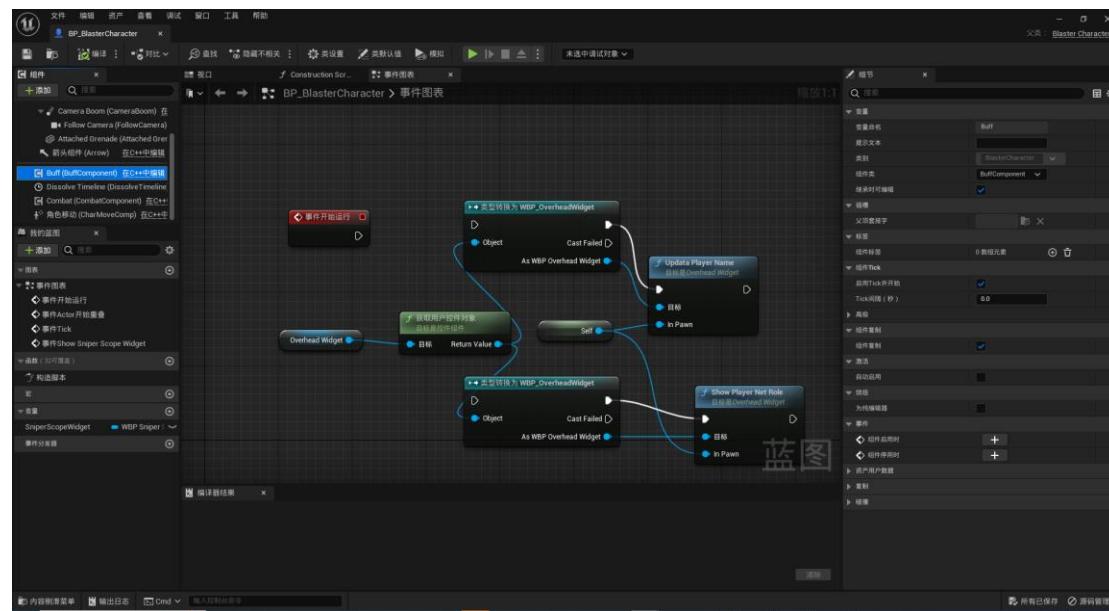
```
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, meta = (AllowPrivateAccess = "true"))
class UBuffComponent* Buff;
```

在构造函数中设置：

```
Buff = CreateDefaultSubobject<UBuffComponent>(TEXT("BuffComponent"));
Buff->SetIsReplicated(true);
```

在 Character.cpp 中添加初始化：

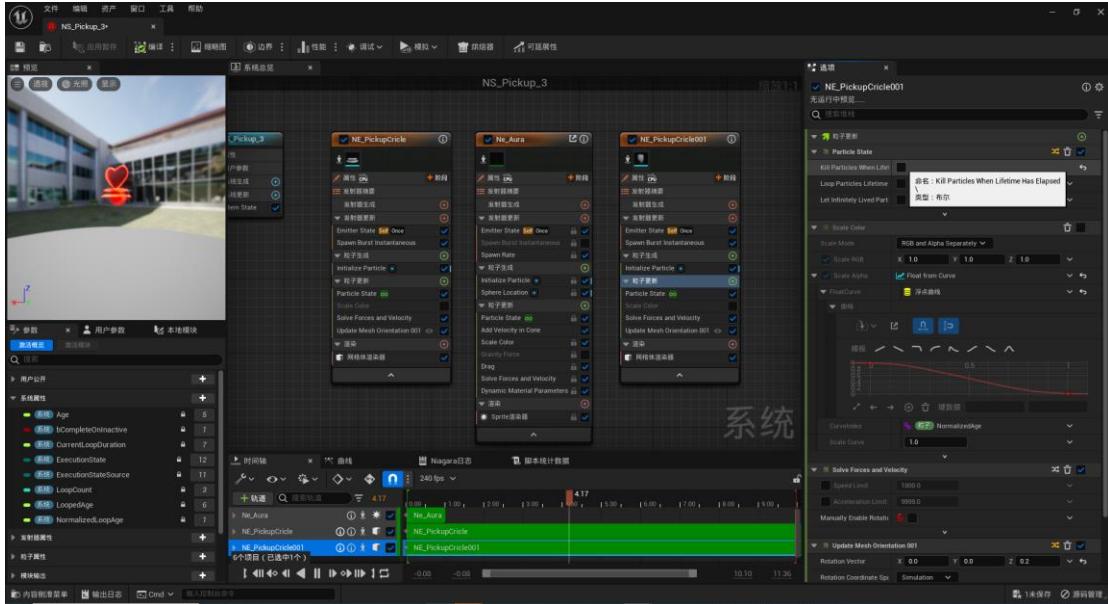
```
void ABlasterCharacter::PostInitializeComponents()
{
    Super::PostInitializeComponents();
    if (Combat)
    {
        Combat->Character = this;
    }
    if (Buff)
    {
        Buff->Character = this;
    }
}
```



在蓝图中我们已经可以看到我们的 buff 组件了。

145 拾取回血

可以在商城中搜索 Basic Pickups VFX Set 来获取资源。



取消掉每个发射器的 kill particle when lifetime elapsed.

为 HealthPickup 添加相应的效果功能。

```
AHealthPickup::AHealthPickup()
{
    bReplicates = true;
    PickupEffectComponent =
        CreateDefaultSubobject<UNiagaraComponent>(TEXT("PickupEffectComponent"));
    PickupEffectComponent->SetupAttachment(RootComponent);
}

void AHealthPickup::OnSphereOverlap(UPrimitiveComponent* OverlappedComponent, AActor*
OtherActor, UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromSweep,
const FHitResult& SweepResult)
{
    Super::OnSphereOverlap(OverlappedComponent, OtherActor, OtherComp, OtherBodyIndex,
    bFromSweep, SweepResult);

    ABlasterCharacter* BlasterCharacter = Cast<ABlasterCharacter>(OtherActor);
    if (BlasterCharacter)
    {

        Destroy();
    }
}
```

```
void AHealthPickup::Destroyed()
{
    if (PickupEffect)
    {
        UNiagaraFunctionLibrary::SpawnSystemAtLocation(
            this,
            PickupEffect,
            GetActorLocation(),
            GetActorRotation()
        );
    }

    Super::Destroyed();
}
```

回复功能将在下一节实现。

146 治疗角色（定时器设置有参回调函数）

在本章中使用了和教程中不一样的实现方式，教程中在重复获取血包时，是通过增加治疗总量，延长治疗时间从而实现的治疗，我觉得实际游戏中一般是通过叠加状态实现的恢复效果，所以采取了叠加状态的方式。

在重复获得 buff 时增加治疗速度，并设置一个定时器，在定时器的回调函数中进行治疗速度的减少。

为了实现这个效果，在网上查找发放后，发现需要在定时器中调用委托 TimerDelegate，并且传入参数，具体实现如下：

```
UBuffComponent::UBuffComponent()
{
    PrimaryComponentTick.bCanEverTick = true;

}

void UBuffComponent::BeginPlay()
{
    Super::BeginPlay();

}

void UBuffComponent::TickComponent(float DeltaTime, ELevelTick TickType,
FActorComponentTickFunction* ThisTickFunction)
{
    Super::TickComponent(DeltaTime, TickType, ThisTickFunction);
    HealRampUp(DeltaTime);
}

void UBuffComponent::Heal(float HealAmount, float HealingTime)
{
    bHealing = true;
    float Rate = HealAmount / HealingTime;
    HealingRate += Rate;
    AmountToHeal += HealAmount;

    FTimerHandle TimerHandle;
    GetWorld()->GetTimerManager().SetTimer(
        TimerHandle,
        FTimerDelegate::CreateUObject(this, &UBuffComponent::HealingTimerFinished,
        Rate),
        HealingTime,
```

```

        false
    );
}

void UBuffComponent::HealRampUp(float DeltaTime)
{
    if (!bHealing || Character == nullptr || Character->IsElimmed())
    {
        return;
    }

    const float HealThisFrame = FMath::Min(HealingRate * DeltaTime, AmountToHeal);
    Character->SetHealth(FMath::Clamp(Character->GetHealth() + HealThisFrame, 0.f,
Character->GetMaxHealth()));
    Character->UpdateHUDHealth();
    AmountToHeal -= HealThisFrame;

    if (AmountToHeal <= 0.f || Character->GetHealth() >= Character->GetMaxHealth())
    {
        bHealing = false;
        AmountToHeal = 0;
    }
}

void UBuffComponent::HealingTimerFinished(float Rate)
{
    HealingRate -= Rate;
}

```

147 速度 Buff

设置一个速度 buff，用计时器删除效果，在设置的时候需要注意不能光在服务端设置，需要进行多播 RPC，同时在服务端和客户端设置 MovementComponent 的速度。

148 跳跃 Buff

可以进入 Interface and item sounds 下载资源。

类似速度 BUFF，创建一个跳跃增加的 buff。

149 护甲值

操作和生命值完全一样。

150 更新护甲

和生命值一样。

151 拾取护甲

和生命值一样。

152 拾取物生成点

用到了一个 AActor 的委托：

FActorDestroyedSignature

.h 文件中，利用了一个数组来存储将要放入的 APickup 子类：

```
UPROPERTY(EditAnywhere)
TArray<TSubclassOf<class APickup>> PickupClasses;
```

.cpp 文件中，主要调用了 AActor 的委托，在 Pickup 销毁时启动定时器：

```
void APickupSpawnPoint::SpawnPickup()
{
    int32 NumPickupClasses = PickupClasses.Num();
    if (NumPickupClasses > 0)
    {
        int32 Selection = FMath::RandRange(0, NumPickupClasses - 1);
        SpawnerPickup = GetWorld()->SpawnActor<APickup>(
            PickupClasses[Selection],
            GetActorTransform()
        );

        if (HasAuthority() && SpawnerPickup)
        {
            SpawnerPickup->OnDestroyed.AddDynamic(this,
&APickupSpawnPoint::StartSpawnPickupTimer);
        }
    }
}
```

此外还需要注意，当我们一直站在生成点上时，出现的物体立刻被 destroy，导致还没来得及绑定重置计时器的函数，所以我们在 Pickup 类中添加一个延迟，在 Beginplay 之后一小段时间再调用重叠的回调函数。

153 为关卡添加物资生成点

摆放关卡中的物件

154 生成默认武器

为角色添加一个默认武器，此外在初始化过程中我并不能像教程中一样成功初始化服务端的 HUD，所以我直接设置了一个定时器，在一小段时间后再更新 HUD。

155 副武器

在 Combat 中添加一个副武器成员变量，和主武器类似。

在 EquipWeapon 中进行检测，有主武器而没有副武器时，添加武器到副武器插槽。

教程中在添加副武器后会设置显示轮廓，我觉得不是很好就没有做这一环节。

156 交换武器

简化了 WeaponState 相关的函数，方便我们拓展一个新状态 EquippedSecondary，新增了一个输入时间 SwapWeapons 以及相应的函数和 RPC，实现武器的交换。

选做项

Optional Challenge: Custom Pickup!

Create a custom pickup! It can do anything you like. Be creative!

Show off your new pickup in the 😊 | share-your-work channel in the Druid Mechanics Discord!

网络延迟补偿

158 网络延迟补偿的概念

首先要知道，当网络延迟过高的时候，无论使用什么延迟补偿的技术，都不能获得好的体验。但当延迟可以控制的时候，我们需要使用延迟补偿技术，从而获得更好的游戏体验。

其次，每种延迟补偿的技术都有其优缺点，使用哪种技术取决于开发团队在哪里做出妥协。在射击游戏中，当延迟超过 50ms 之后，玩家就能够明显的感觉到延迟的存在。

最后，游戏的哪些部分需要做延迟补偿？

重要的部分，当涉及到游戏逻辑相关的部分我们通常需要延迟补偿，而在表现上的部分，我们通常不使用延迟补偿。

常见的服务器补偿方法有：

内插值

当客户端收到服务器传来的位置信息时，可以和上一次传来的位置信息做内插值，这样可以使得移动的过程看起来更加平滑。

但是也会导致延迟的进一步加重。这种延迟可能对射击游戏不利。例如你射击的敌人位置可能由于内插值更加靠后，这样的话，你的射击更有可能无法命中目标。

外插值

如果你知道对手在往哪个方向跑，你可以假设他们会继续朝着同一方向和客户端跑。你可以在等待下一次更新的同时继续向前移动这个角色。如果对手没有改变路线，这会非常有效，保持相同的速度。

但是在射击游戏中，玩家通常不会以可预测的方式移动，这会导致使用外插值的结果比没有使用外插值的结果更加差。

UE 中橡皮筋

UE 使用了内插值和外插值结合的方式来进行移动，如果你的 ping 非常高，角色移动组件将使用你的速度来推断你在其他机器上的位置，使你的角色在他们的机器上移动更平滑。

如果需要进行校正，则角色移动组件会使用内插值平滑地进行校正。

如果服务器和客户端的位置太不同步，角色移动组件将传送你的角色回到正确的位置。这被称为橡皮筋。

服务器倒带 (server side rewind)

可以提高高 ping 玩家的体验，但是会损害其他玩家体验。
因此有些游戏会有限制，不让延迟过高的人使用服务器倒带。

159 高延迟警告

在 CharacterOverlay 中添加两个变量：

```
UPROPERTY(meta = (BindWidget))
class UImage* HighPingImage;

UPROPERTY(meta = (BindWidgetAnim), Transient)
UWidgetAnimation* HighPingAnimation;
```

在 Controller 中：

```
void ABlasterPlayerController::HighPingWarning()
{
    BlasterHUD = BlasterHUD == nullptr ? BlasterHUD = Cast<ABlasterHUD>(GetHUD()) : BlasterHUD;
    bool bHUDValid = BlasterHUD &&
        BlasterHUD->CharacterOverlay &&
        BlasterHUD->CharacterOverlay->HighPingImage &&
        BlasterHUD->CharacterOverlay->HighPingAnimation;
    if (bHUDValid)
    {
        BlasterHUD->CharacterOverlay->HighPingImage->SetOpacity(1. f);
        BlasterHUD->CharacterOverlay->PlayAnimation(
            BlasterHUD->CharacterOverlay->HighPingAnimation,
            0. f,
            5
        );
    }
}

void ABlasterPlayerController::StopHighPingWarning()
{
    BlasterHUD = BlasterHUD == nullptr ? BlasterHUD = Cast<ABlasterHUD>(GetHUD()) : BlasterHUD;
    bool bHUDValid = BlasterHUD &&
        BlasterHUD->CharacterOverlay &&
        BlasterHUD->CharacterOverlay->HighPingImage &&
        BlasterHUD->CharacterOverlay->HighPingAnimation;
    if (bHUDValid)
    {
        BlasterHUD->CharacterOverlay->HighPingImage->SetOpacity(0. f);
        if
            (BlasterHUD->CharacterOverlay->IsAnimationPlaying(BlasterHUD->CharacterOverlay->HighPingAnimation))
        {

```

```

        BlasterHUD->CharacterOverlay->StopAnimation(BlasterHUD->CharacterOverlay->HighPingA
imation);
    }
}
}

```

在 Tick 函数中我们没 20 秒检查 Ping 值，如果 Ping 过高则播放 UI 中的图标闪烁动画，这里教程中使用了一个 PlayerState->GetPing() 函数，这个函数只能获取压缩过后的 Ping 值，Ping 被除了 4，如果想要获取精确的 Ping 可以将我们之前做时间对齐的时候从客户端发送的时间戳的时间差提升为变量获取：

```

float HighPingRunningTime = 0.f;
UPROPERTY(EditAnywhere)
float HighPingDuration = 5.f;
UPROPERTY(EditAnywhere)
float PingAnimationRunningTime = 0.f;
UPROPERTY(EditAnywhere)
float CheckPingFrequency = 20.f;
UPROPERTY(EditAnywhere)
float HighPingThreshold = 50.f;

/** Gets the literal value of the compressed Ping value (Ping = PingInMS / 4). */
UFUNCTION(BlueprintGetter)
uint8 GetCompressedPing() const
{
    return CompressedPing;
}

UE_DEPRECATED(5.0, "Use GetPingInMilliseconds() or GetCompressedPing() instead")
uint8 GetPing() const { return GetCompressedPing(); }

```

```

void ABlasterPlayerController::CheckPing(float DeltaTime)
{
    HighPingRunningTime += DeltaTime;
    if (HighPingRunningTime > CheckPingFrequency)
    {
        HighPingRunningTime = 0.f;

        PlayerState = PlayerState == nullptr ? GetPlayerState<APlayerState>() :
PlayerState;
        if (PlayerState)
        {
            if (PlayerState->GetPing() * 4 > HighPingThreshold) // ping is compressed;
it's actually ping / 4
            {
                HighPingWarning();
                PingAnimationRunningTime = 0.f;
            }
        }
    }
}

```

```

    }

    bool bHighPingAnimationPlaying =
        BlasterHUD &&
        BlasterHUD->CharacterOverlay &&
        BlasterHUD->CharacterOverlay->HighPingAnimation &&

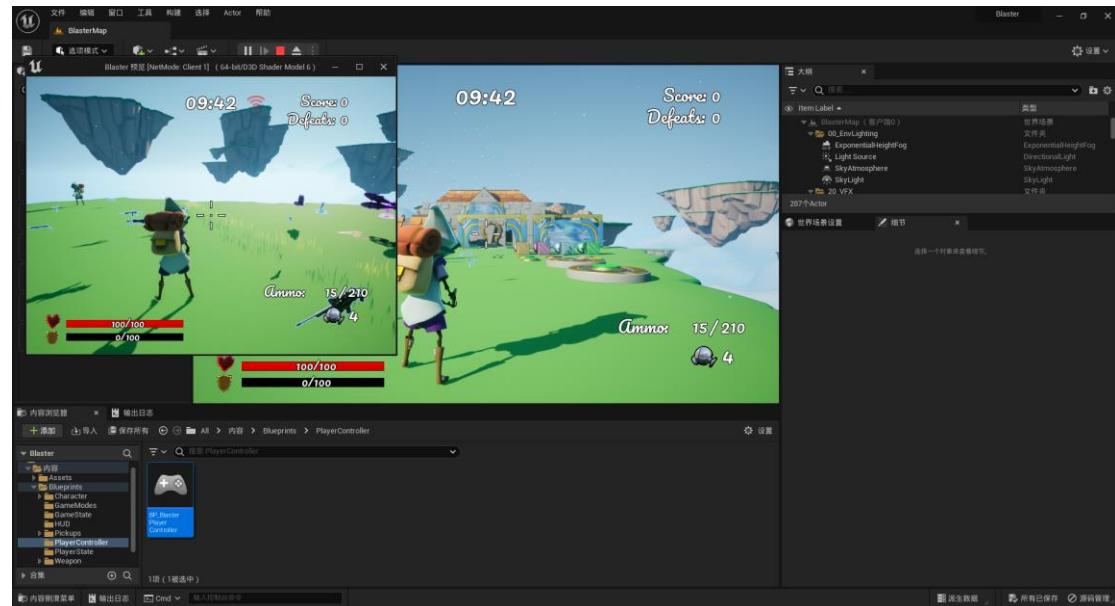
        BlasterHUD->CharacterOverlay->IsAnimationPlaying(BlasterHUD->CharacterOverlay->High
PingAnimation);
    if (bHighPingAnimationPlaying)
    {
        PingAnimationRunningTime += DeltaTime;
        if (PingAnimationRunningTime > HighPingDuration)
        {
            StopHighPingWarning();
        }
    }
}

```

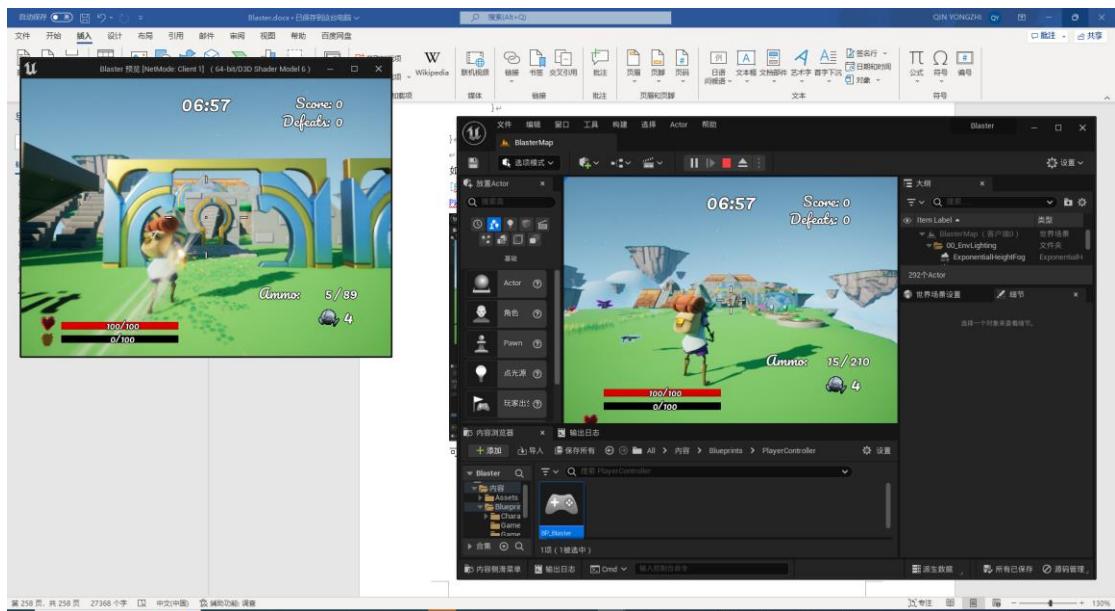
如何测试有 Ping 的情况呢，打开 DefaultEngine.ini，用如下代码模拟 ping：

[PacketSimulationSettings]

PktLag = 100



可以看到我们现在已经是一个高 Ping 环境了。



已经可以看到一些高 Ping 导致的问题，在客户端上向前移动看到开火会击中自己，子弹从自己背后射出，这也和我们射击的实现方式有关系，之前为了解决 Socket 位置不对的关系，我们会将 RPC 中上传开枪的位置，从而获得服务端和客户端一致的效果。

160 本地开火效果

由于我们之前的代码，伤害相关的逻辑都在后续的代码中进行了 HasAuthority 的检查，我们可以直接在 Combat 中将 multicast 的代码修改到每个客户端执行，但是需要注意的是，在 HitScan 武器的散射中我们使用了随机数，这里会导致服务端和客户端的不一致，我们之后会修复这个问题。

```
void UCombatComponent::Fire()
{
    if (CanFire())
    {
        if (EquippedWeapon)
        {
            const auto MuzzleFlashSocket =
                EquippedWeapon->GetWeaponMesh()->GetSocketByName(FName("MuzzleFlash"));
            if (MuzzleFlashSocket)
            {
                FTransform SocketTransform =
                    MuzzleFlashSocket->GetSocketTransform(EquippedWeapon->GetWeaponMesh());
                CrosshairShootingFactor +=

                EquippedWeapon->GetCrosshairShootingFactor();
                CrosshairShootingFactor =
                    UKismetMathLibrary::FMin(EquippedWeapon->GetCrosshairShootingMaxFactor(),
                    CrosshairShootingFactor);

                ServerFire(SocketTransform.GetLocation(), HitTarget);
                LocalFire(SocketTransform.GetLocation(), HitTarget);
                StartFireTimer();
            }
        }
    }
}

void UCombatComponent::ServerFire_Implementation(const FVector_NetQuantize&
    SocketLocation, const FVector_NetQuantize& TraceHitTarget)
{
    MulticastFire(SocketLocation, TraceHitTarget);
}

void UCombatComponent::MulticastFire_Implementation(const FVector_NetQuantize&
    SocketLocation, const FVector_NetQuantize& TraceHitTarget)
{
    if (Character && Character->IsLocallyControlled())
    {
        return;
    }
}
```

```

        LocalFire(SocketLocation, TraceHitTarget);
    }

void UCombatComponent::LocalFire(const FVector_NetQuantize& SocketLocation, const
FVector_NetQuantize& TraceHitTarget)
{
    if (nullptr == EquippedWeapon)
    {
        return;
    }

    if (Character && CombatState == ECombatState::ECS_Reloading &&
EquippedWeapon->GetWeaponType() == EWeaponType::EWT_Shotgun)
    {
        Character->PlayFireMontage(bAiming);
        EquippedWeapon->Fire(SocketLocation, TraceHitTarget);
        CombatState = ECombatState::ECS_Unoccupied;
        return;
    }

    if (Character && CombatState == ECombatState::ECS_Unoccupied)
    {
        Character->PlayFireMontage(bAiming);
        EquippedWeapon->Fire(SocketLocation, TraceHitTarget);
    }
}

```

此外，我们还注意到，开出一枪后立刻换子弹会导致无法回到 Unoccupied 状态，这是由于我们没有在 ServerReload 中做检查导致的。

```

void UCombatComponent::ServerReload_Implementation()
{
    if (Character == nullptr || EquippedWeapon == nullptr)
    {
        return;
    }

    if (CarriedAmmo == 0 || EquippedWeapon->IsFull())
    {
        return;
    }

    if (CombatState != ECombatState::ECS_Unoccupied)
    {
        return;
    }

    CombatState = ECombatState::ECS_Reloading;
}

```

```
    HandleReload();  
}
```

最后我们注意到瞄准动作会在快速按下右键后触发两次，这是由于我们在客户端虽然退出了瞄准状态，但是由于复制的变量 bAiming 刚从服务器返回，我们会被设置成秒准状态，然后又返回普通的状态。

161 在本地的控件展示

在 Weapon 中，不再只设置在服务端生成 overlap 事件：

```
void AWeapon::BeginPlay()
{
    Super::BeginPlay();
    if (PickupWidget)
    {
        PickupWidget->SetVisibility(false);
    }

    AreaSphere->SetCollisionEnabled(ECollisionEnabled::QueryAndPhysics);
    AreaSphere->SetCollisionResponseToChannel(ECollisionChannel::ECC_Pawn,
ECollisionResponse::ECR_Overlap);
    AreaSphere->OnComponentBeginOverlap.AddDynamic(this, &AWeapon::OnSphereOverlap);
    AreaSphere->OnComponentEndOverlap.AddDynamic(this, &AWeapon::OnSphereEndOverlap);
}
```

后续拾起武器的操作只能在服务器生效，所以这样改动并不会导致大的问题。

162 复制扩散

在 Weapon 中添加一个枚举类，并添加一个该类型的变量：

```
UENUM(BlueprintType)
enum class EFireType :uint8
{
    EFT_HitScan UMETA(DisplayName = "Hit Scan Weapon"),
    EFT_Projectile UMETA(DisplayName = "Projectile Weapon"),
    EFT_Shotgun UMETA(DisplayName = "Shotgun"),
    EFT_MAX UMETA(DisplayName = "DefaultMAX"),
};
```

我们在 Combat 的 Fire 中对每种武器种类单独处理，并且在计算散射后的开火后的目标直接传到服务端，为了方便，我们把 Scatter 相关的函数移动到了 Weapon 类中，这样我们甚至可以在 ProjectileWeapon 中调用他。

```
void UCombatComponent::Fire()
{
    if (CanFire())
    {
        if (EquippedWeapon)
        {
            switch (EquippedWeapon->FireType)
            {
                case EFireType::EFT_Projectile:
                    FireProjectileWeapon();
                    break;
                case EFireType::EFT_HitScan:
                    FireHitScanWeapon();
                    break;
                case EFireType::EFT_Shotgun:
                    FireShotgun();
                    break;
            }
            CrosshairShootingFactor += EquippedWeapon->GetCrosshairShootingFactor();
            CrosshairShootingFactor =
UKismetMathLibrary::FMin(EquippedWeapon->GetCrosshairShootingMaxFactor(),
CrosshairShootingFactor);
            StartFireTimer();
        }
    }
}

void UCombatComponent::FireProjectileWeapon()
```

```

{
    const auto MuzzleFlashSocket =
EquippedWeapon->GetWeaponMesh()->GetSocketByName(FName("MuzzleFlash"));

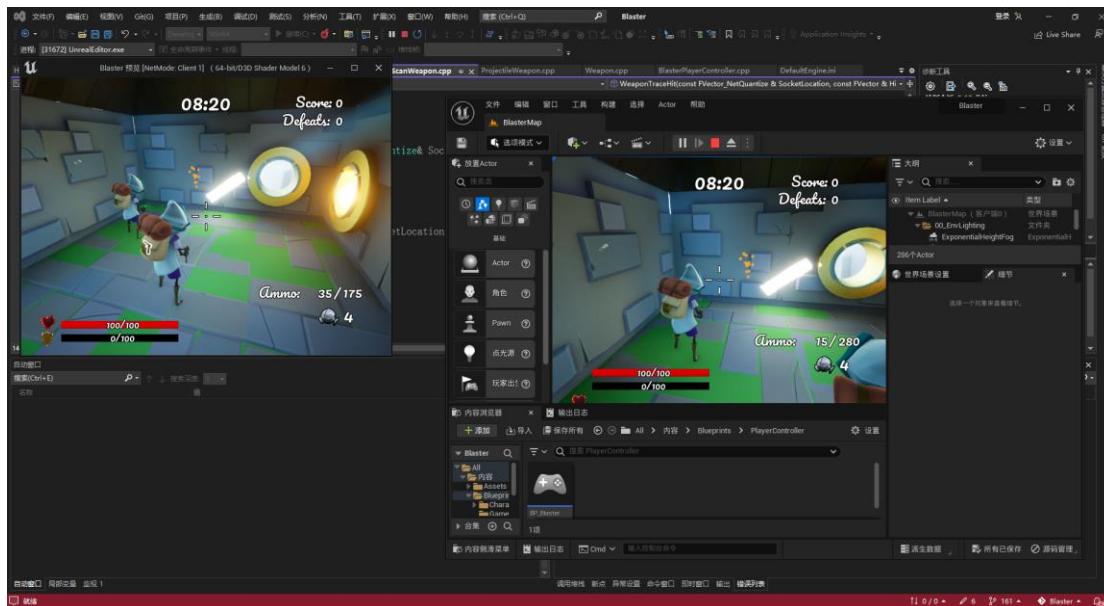
    if (MuzzleFlashSocket)
    {
        FTransform SocketTransform =
MuzzleFlashSocket->GetSocketTransform(EquippedWeapon->GetWeaponMesh());
        ServerFire(SocketTransform.GetLocation(), HitTarget);
        LocalFire(SocketTransform.GetLocation(), HitTarget);
    }
}

void UCombatComponent::FireHitScanWeapon()
{
    const auto MuzzleFlashSocket =
EquippedWeapon->GetWeaponMesh()->GetSocketByName(FName("MuzzleFlash"));

    if (MuzzleFlashSocket)
    {
        FTransform SocketTransform =
MuzzleFlashSocket->GetSocketTransform(EquippedWeapon->GetWeaponMesh());
        HitTarget = EquippedWeapon->bUseScatter ?
EquippedWeapon->TraceEndWithScatter(SocketTransform.GetLocation(), HitTarget) :
HitTarget;
        ServerFire(SocketTransform.GetLocation(), HitTarget);
        LocalFire(SocketTransform.GetLocation(), HitTarget);
    }
}

void UCombatComponent::FireShotgun()
{
}

```



163-164 复制霰弹枪散射

由于霰弹枪需要连续进行多次射线检测，我们创建一个 TArray 来保存随机散射结果：

```
void UCombatComponent::FireShotgun()
{
    const auto MuzzleFlashSocket =
EquippedWeapon->GetWeaponMesh()->GetSocketByName(FName("MuzzleFlash"));

    if (MuzzleFlashSocket == nullptr)
    {
        return;
    }

    FTransform SocketTransform =
MuzzleFlashSocket->GetSocketTransform(EquippedWeapon->GetWeaponMesh());
    AShotgun* Shotgun = Cast<AShotgun>(EquippedWeapon);
    if (Shotgun)
    {
        TArray<FVector> HitTargets;
        Shotgun->ShotgunTraceEndWithScatter(SocketTransform.GetLocation(), HitTarget,
HitTargets);
    }
}
```

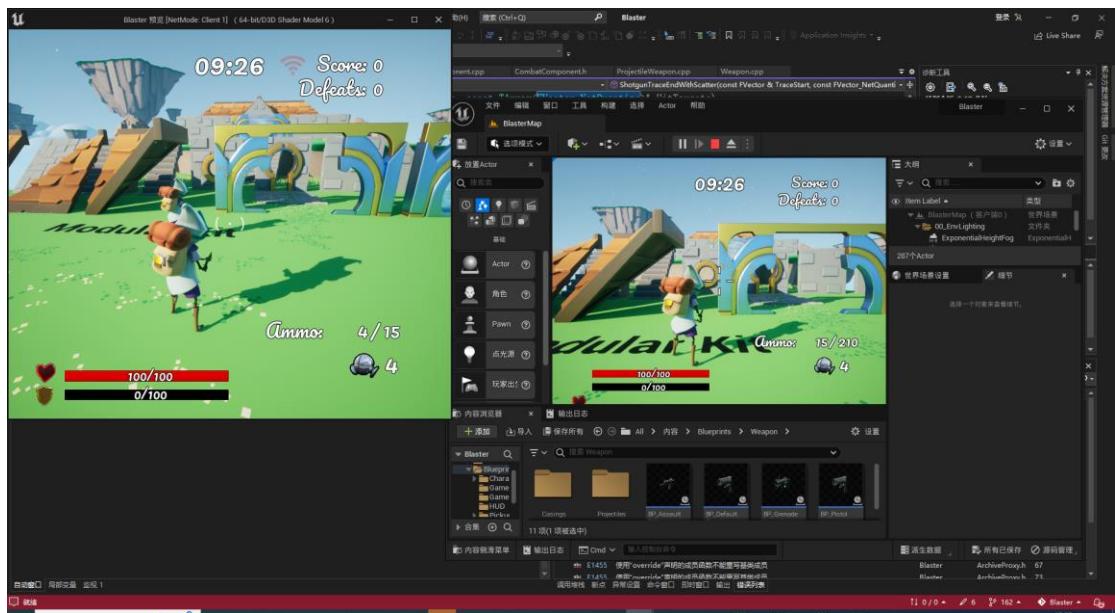
相应的，Shotgun 中我们定义如下函数，来实现霰弹枪的随机散射结果。

```
void AShotgun::ShotgunTraceEndWithScatter(const FVector& TraceStart, const
FVector_NetQuantize& HitTarget, TArray<FVector>& HitTargets)
{
    const FVector ToTargetNormalized = (HitTarget - TraceStart).GetSafeNormal();
    const FVector SphereCenter = TraceStart + ToTargetNormalized * DistanceToSphere;
    for (size_t i = 0; i < NumberOfPellets; i++)
    {
        const FVector RandVec = UKismetMathLibrary::RandomUnitVector() *
FMath::FRandRange(0. f, SphereRadius);
        const FVector EndLoc = SphereCenter + RandVec;
        const FVector ToEndLoc = EndLoc - TraceStart;
        FVector Target = TraceStart + ToEndLoc * TRACE_LENGTH / ToEndLoc.Size();
        HitTargets.Add(Target);
    }
}
```

由于存在一个 Tarray，所以需要使用一个专门的 RPC 来传递这个变量。

我们专门实现一个霰弹枪的 local 开火和 server 开火，以及 multicast 开火函数。

在 shotgun 中，我们重新创建一个 shotgunfire 函数，将输入参数改变为数组。然后再 local 函数中调用这个函数。



我发现 characteroverlay 中有些图标的锚点不对，简单修正即可。

165 客户端预测

相比于普通的移动方式，客户端预测增加了 56 两步，避免了移动过程中的抖动。

1. 客户端移动
2. 发送 RPC (客户端保存下发送的 RPC)
3. 收到服务端的处理 RPC 的响应
4. 修正位置
5. 丢弃收到响应的 RPC
6. 重新按顺序使用没有被丢弃的 RPC

例如，我们在客户端依次走过 1, 2, 3 这三个位置，每个位置向服务器发送一个 RPC，并存储下来，我们走到 3 的时候，收到了服务器传来的响应，我们纠正位置回到 1，并且舍弃 1 号 RPC，接下来我们依次执行我们所存储的 2 和 3 两个 RPC，我们又回到了 3 位置，然后收到服务器传来的 2 号位置的响应，我们回到 2 号位置并丢弃 2 号 RPC，接下来执行 3 号 RPC 又回到 3 号位置，最后我们收到 3 号位置的响应，我们也在 3 号位置，这样实现了一个没有来回抖动的客户端移动。

UE 为我们在角色移动中已经封装了这个功能，但是我们依然可以在其他地方用到这个方法，比如弹药。

166 客户端预测弹药

我们尝试一个更差的网络环境，把延迟设置到 200ms。

我们发现会出现子弹打空之后不自动装弹的问题，这是因为服务端的子弹还没有打光，所以直接被 return 了，为了解决这个问题，我们把 SpendRound 函数不光在服务器上执行，也在客户端执行，但这也会导致客户端会出现子弹数量的抖动，所以就要用到服务端预测的方法，使用两个客户端 RPC 解决这个问题：

```
void AWeapon::SpendRound()
{
    Ammo = FMath::Clamp(Ammo - 1, 0, MagCapacity);
    SetHUDAmmo();
    if (HasAuthority())
    {
        ClientUpdateAmmo(Ammo);
    }
    else
    {
        Sequence++;
    }
    UE_LOG(LogTemp, Warning, TEXT("Sequence : %d"), Sequence);
}

void AWeapon::ClientUpdateAmmo_Implementation(int32 ServerAmmo)
{
```

```

    if (HasAuthority())
    {
        return;
    }
    Sequence--;
    Ammo = ServerAmmo;
    Ammo -= Sequence;
    SetHUDAmmo();
}

void AWeapon::AddAmmo(int32 AmmoToAdd)
{
    Ammo = FMath::Clamp(Ammo + AmmoToAdd, 0, MagCapacity);
    SetHUDAmmo();
    ClientAddAmmo(AmmoToAdd);
}

void AWeapon::ClientAddAmmo_Implementation(int32 AmmoToAdd)
{
    if (HasAuthority())
    {
        return;
    }
    Ammo = FMath::Clamp(Ammo + AmmoToAdd, 0, MagCapacity);
    BlasterOwnerCharacter = BlasterOwnerCharacter == nullptr ?
        Cast<ABlasterCharacter>(GetOwner()) : BlasterOwnerCharacter;
    if (BlasterOwnerCharacter && BlasterOwnerCharacter->GetCombatComponent() &&
        IsFull())
    {
        BlasterOwnerCharacter->GetCombatComponent()->JumpToShotgunEnd();
    }
    SetHUDAmmo();
}

```

但是现在仍然会出现问题，当我们在射击过程中换子弹的时候 Sequence 不会减少，而且换子弹的数量也会错误。还有可能卡住退不出 Reloading 状态。

167 客户端预测瞄准

直接在客户端设置一个变量 bAimButtonPressed 进行 bAiming 的修正：

```
void UCombatComponent::SetAiming(bool bIsAiming)
{
    if (Character == nullptr || EquippedWeapon == nullptr)
    {
        return;
    }

    bAiming = bIsAiming;
    ServerSetAiming(bIsAiming);
    Character->GetCharacterMovement()->MaxWalkSpeed = bIsAiming ? AimWalkSpeed : BaseWalkSpeed;
    if (Character->IsLocallyControlled() && EquippedWeapon->GetWeaponType() ==
EWeaponType::EWT_SniperRifle)
    {
        Character->ShowSniperScopeWidget(bIsAiming);
    }
    if (Character->IsLocallyControlled())
    {
        bAimButtonPressed = bIsAiming;
    }
}

void UCombatComponent::OnRep_Aiming()
{
    if (Character && Character->IsLocallyControlled())
    {
        bAiming = bAimButtonPressed;
    }
}
```

168 客户端预测装填弹药

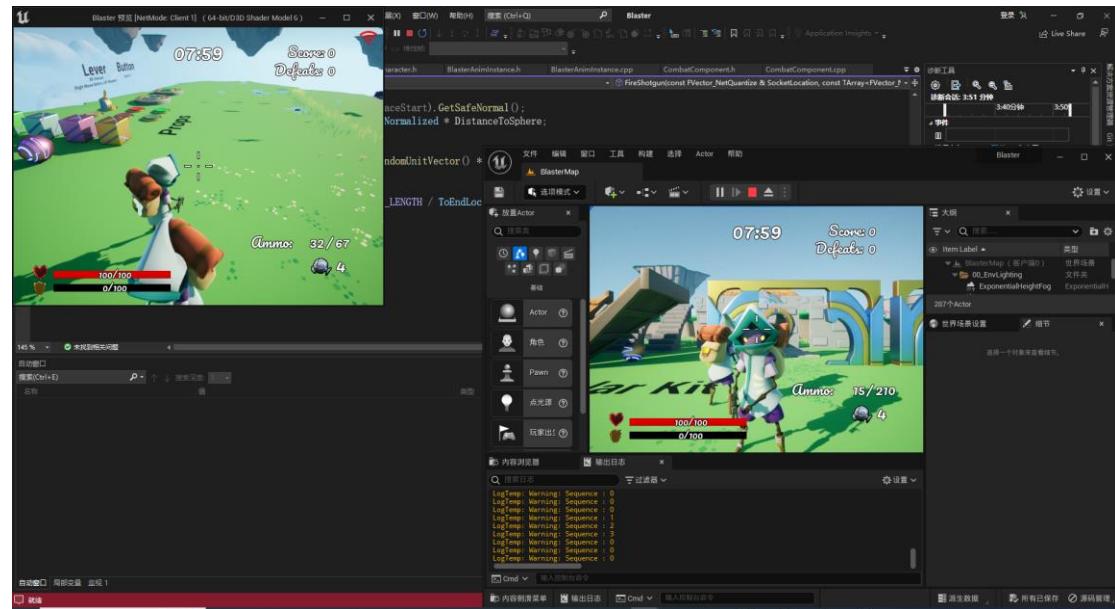
在装填弹药的时候我们需要直接在客户端实现装填的动画，但是我们之前是直接调用了 ServerReload 设置了 CombatState，然后通过复制播放动画。现在由于延迟的存在，这会造成很明显的装填弹药滞后的感觉。

直接在本地播放动画的话，由于还没复制状态，会导致动画被打断，最后无法退出 Reloading 状态。

在本地添加了一个变量，使在换弹的时候不能开火，以及每次调用 reload 之前要判断是否在本地已经进行了换弹，且还没结束。并且由于开始得比服务器早，所以要利用本地变量提前关闭和开启 FABRIK。

BUG

在换弹过程中收到攻击会导致，蒙太奇中断无法完成换弹。其次会在换弹中由于开火中断换弹导致无法退出换弹？又好像又是当远离服务器角色之后会导致的蒙太奇通知没有执行。



169 服务器倒带

170 延迟补偿组件

创建一个 ActorComponent 组件，来作为延迟补偿组件，我们要思考的是，我们需要怎样的一个数据结构来存储某一时刻某个人的姿势？

最简单的使用单个胶囊体，那样结果会很不准确，使用网格体进行检测，那样又会太耗费资源。我们可以采用一系列的包围盒，并记录他们的旋转和位移，以平衡效果和开销。

171 Hit Boxes

在角色中我们按照我们需要添加包围盒的骨骼的名称添加一系列包围盒：

```
/**  
 * Hit Boxes used for server-side rewind  
 */  
UPROPERTY(EditAnywhere)  
class UBoxComponent* head;  
UPROPERTY(EditAnywhere)  
UBoxComponent* pelvis;  
UPROPERTY(EditAnywhere)  
UBoxComponent* spine_02;  
UPROPERTY(EditAnywhere)  
UBoxComponent* spine_03;  
UPROPERTY(EditAnywhere)  
UBoxComponent* upperarm_l;  
UPROPERTY(EditAnywhere)  
UBoxComponent* upperarm_r;  
UPROPERTY(EditAnywhere)  
UBoxComponent* lowerarm_l;  
UPROPERTY(EditAnywhere)  
UBoxComponent* lowerarm_r;  
UPROPERTY(EditAnywhere)  
UBoxComponent* hand_l;  
UPROPERTY(EditAnywhere)  
UBoxComponent* hand_r;  
UPROPERTY(EditAnywhere)  
UBoxComponent* backpack;  
UPROPERTY(EditAnywhere)  
UBoxComponent* blanket;  
UPROPERTY(EditAnywhere)  
UBoxComponent* thigh_l;  
UPROPERTY(EditAnywhere)
```

```

UBoxComponent* thigh_r;
UPROPERTY(EditAnywhere)
UBoxComponent* calf_l;
UPROPERTY(EditAnywhere)
UBoxComponent* calf_r;
UPROPERTY(EditAnywhere)
UBoxComponent* foot_l;
UPROPERTY(EditAnywhere)
UBoxComponent* foot_r;

```

在构造函数中将他们初始化:

```

/**
 * Hit boxes for server-side rewind
 */

head = CreateDefaultSubobject<UBoxComponent>(TEXT("head"));
head->SetupAttachment(GetMesh(), FName("head"));
head->SetCollisionEnabled(ECollisionEnabled::NoCollision);

pelvis = CreateDefaultSubobject<UBoxComponent>(TEXT("pelvis"));
pelvis->SetupAttachment(GetMesh(), FName("pelvis"));
pelvis->SetCollisionEnabled(ECollisionEnabled::NoCollision);

spine_02 = CreateDefaultSubobject<UBoxComponent>(TEXT("spine_02"));
spine_02->SetupAttachment(GetMesh(), FName("spine_02"));
spine_02->SetCollisionEnabled(ECollisionEnabled::NoCollision);

spine_03 = CreateDefaultSubobject<UBoxComponent>(TEXT("spine_03"));
spine_03->SetupAttachment(GetMesh(), FName("spine_03"));
spine_03->SetCollisionEnabled(ECollisionEnabled::NoCollision);

upperarm_l = CreateDefaultSubobject<UBoxComponent>(TEXT("upperarm_l"));
upperarm_l->SetupAttachment(GetMesh(), FName("upperarm_l"));
upperarm_l->SetCollisionEnabled(ECollisionEnabled::NoCollision);

upperarm_r = CreateDefaultSubobject<UBoxComponent>(TEXT("upperarm_r"));
upperarm_r->SetupAttachment(GetMesh(), FName("upperarm_r"));
upperarm_r->SetCollisionEnabled(ECollisionEnabled::NoCollision);

lowerarm_l = CreateDefaultSubobject<UBoxComponent>(TEXT("lowerarm_l"));
lowerarm_l->SetupAttachment(GetMesh(), FName("lowerarm_l"));
lowerarm_l->SetCollisionEnabled(ECollisionEnabled::NoCollision);

lowerarm_r = CreateDefaultSubobject<UBoxComponent>(TEXT("lowerarm_r"));
lowerarm_r->SetupAttachment(GetMesh(), FName("lowerarm_r"));
lowerarm_r->SetCollisionEnabled(ECollisionEnabled::NoCollision);

```

```

hand_l = CreateDefaultSubobject<UBoxComponent>(TEXT("hand_l"));
hand_l->SetupAttachment(GetMesh(), FName("hand_l"));
hand_l->SetCollisionEnabled(ECollisionEnabled::NoCollision);

hand_r = CreateDefaultSubobject<UBoxComponent>(TEXT("hand_r"));
hand_r->SetupAttachment(GetMesh(), FName("hand_r"));
hand_r->SetCollisionEnabled(ECollisionEnabled::NoCollision);

backpack = CreateDefaultSubobject<UBoxComponent>(TEXT("backpack"));
backpack->SetupAttachment(GetMesh(), FName("backpack"));
backpack->SetCollisionEnabled(ECollisionEnabled::NoCollision);

blanket = CreateDefaultSubobject<UBoxComponent>(TEXT("blanket"));
blanket->SetupAttachment(GetMesh(), FName("backpack"));
blanket->SetCollisionEnabled(ECollisionEnabled::NoCollision);

thigh_l = CreateDefaultSubobject<UBoxComponent>(TEXT("thigh_l"));
thigh_l->SetupAttachment(GetMesh(), FName("thigh_l"));
thigh_l->SetCollisionEnabled(ECollisionEnabled::NoCollision);

thigh_r = CreateDefaultSubobject<UBoxComponent>(TEXT("thigh_r"));
thigh_r->SetupAttachment(GetMesh(), FName("thigh_r"));
thigh_r->SetCollisionEnabled(ECollisionEnabled::NoCollision);

calf_l = CreateDefaultSubobject<UBoxComponent>(TEXT("calf_l"));
calf_l->SetupAttachment(GetMesh(), FName("calf_l"));
calf_l->SetCollisionEnabled(ECollisionEnabled::NoCollision);

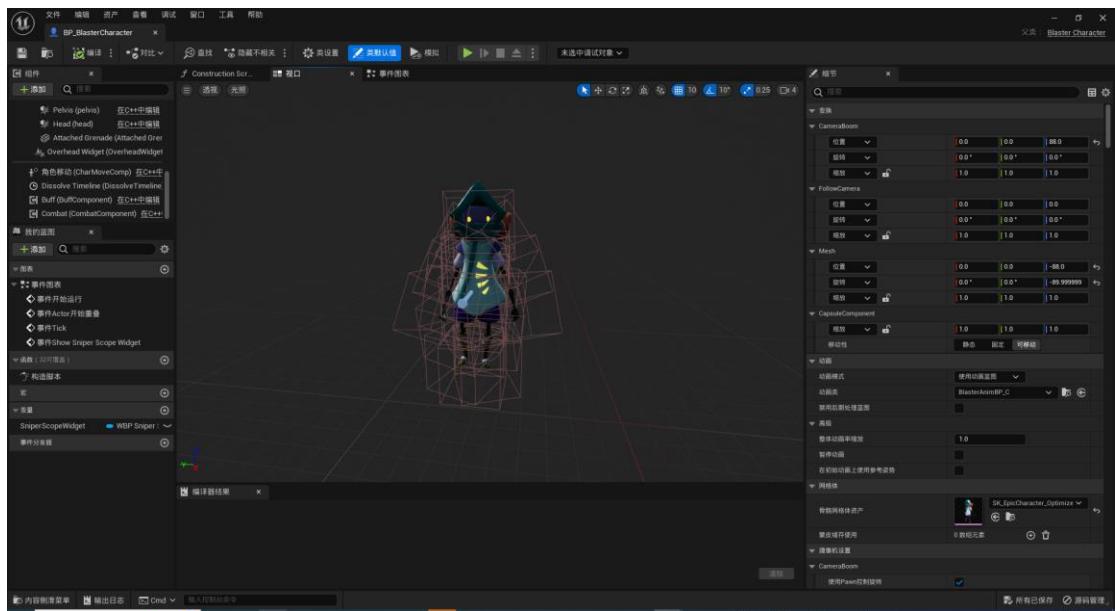
calf_r = CreateDefaultSubobject<UBoxComponent>(TEXT("calf_r"));
calf_r->SetupAttachment(GetMesh(), FName("calf_r"));
calf_r->SetCollisionEnabled(ECollisionEnabled::NoCollision);

foot_l = CreateDefaultSubobject<UBoxComponent>(TEXT("foot_l"));
foot_l->SetupAttachment(GetMesh(), FName("foot_l"));
foot_l->SetCollisionEnabled(ECollisionEnabled::NoCollision);

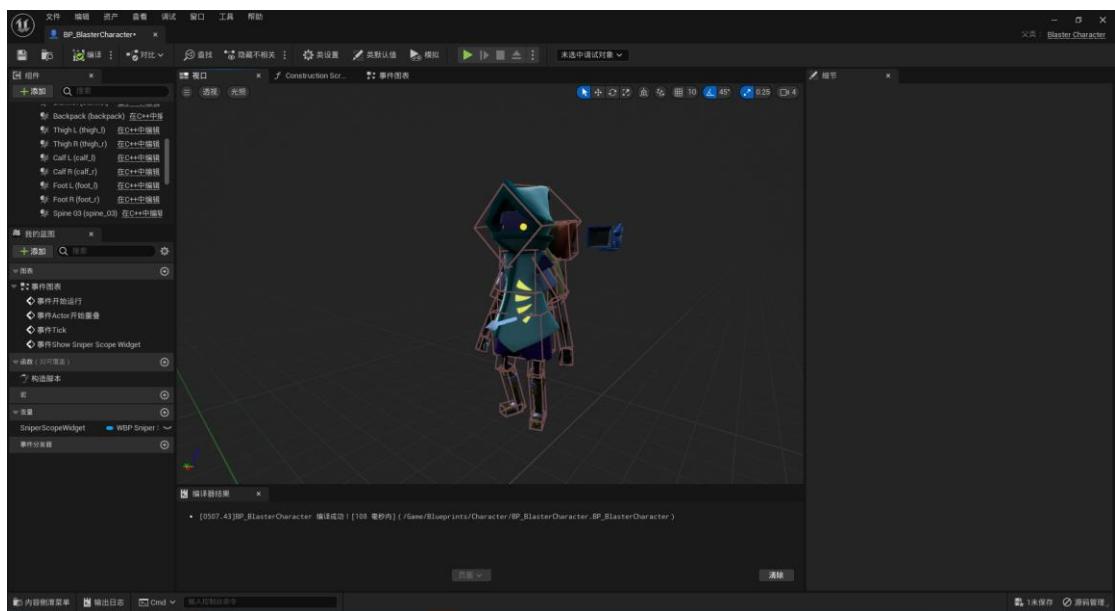
foot_r = CreateDefaultSubobject<UBoxComponent>(TEXT("foot_r"));
foot_r->SetupAttachment(GetMesh(), FName("foot_r"));
foot_r->SetCollisionEnabled(ECollisionEnabled::NoCollision);

```

编译之后再编辑器中编辑哥哥包围盒。



将每个部位调整合适：





172 每帧的包

在 LagCompensation 中我们可以添加两个结构体，来组成我们的包体：

```
USTRUCT(BlueprintType)
struct FFramePackege
{
    GENERATED_BODY()

    UPROPERTY()
    float Time;

    UPROPERTY()
    TMap<FName, FBoxInformation> HitBoxInfo;
};

USTRUCT(BlueprintType)
struct FBoxInformation
{
    GENERATED_BODY()

    UPROPERTY()
    FVector Location;

    UPROPERTY()
    FRotator Rotation;

    UPROPERTY()
    FVector BoxExtent;
};
```

173 保存 FramePackage

为了方便迭代各个盒体，我们在角色中用一个 Map 存储每个盒体：

```
UPROPERTY()
TMap<FName, UBoxComponent*> HitCollisionBoxes;
```

并且在构造函数中按照如下方法初始化它：

```
head = CreateDefaultSubobject<UBoxComponent>(TEXT("head"));
head->SetupAttachment(GetMesh(), FName("head"));
head->SetCollisionEnabled(ECollisionEnabled::NoCollision);
HitCollisionBoxes.Add(FName("head"), head);
```

在延迟补偿组件中，我们添加一个保存包体的函数，此外为了展示包体的信息，我们还添加了一个显示包体的函数，并在 beginplay 中保存一个包体，然后显示他：

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

    FFramePackage Package;
    SaveFramePackage(Package);
    ShowFramePackage(Package, FColor::Orange);
}

void ULagCompensationComponent::SaveFramePackage(FFramePackage& Package)
{
    Character = Character == nullptr ? Cast<ABlasterCharacter>(GetOwner()) : Character;
    if (Character)
    {
        Package.Time = GetWorld()->GetTimeSeconds();
        for (auto& BoxPair : Character->HitCollisionBoxes)
        {
            FBoxInformation BoxInformation;
            BoxInformation.Location = BoxPair.Value->GetComponentLocation();
            BoxInformation.Rotation = BoxPair.Value->GetComponentRotation();
            BoxInformation.BoxExtent = BoxPair.Value->GetScaledBoxExtent();
            Package.HitBoxInfo.Add(BoxPair.Key, BoxInformation);
        }
    }
}

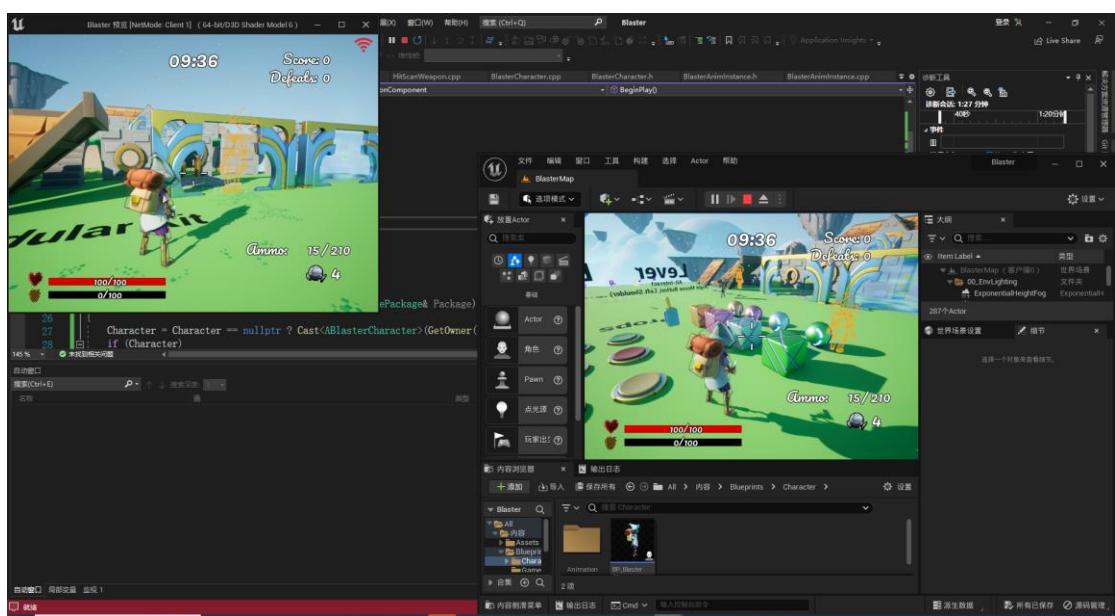
void ULagCompensationComponent::ShowFramePackage(const FFramePackage& Package, const FColor& Color)
{
    for (auto& BoxInfo : Package.HitBoxInfo)
```

```

    }

    DrawDebugBox(
        GetWorld(),
        BoxInfo.Value.Location,
        BoxInfo.Value.BoxExtent,
        FQuat(BoxInfo.Value.Rotation),
        Color,
        true
    );
}

```



174 帧历史

我们用一个双向链表来存储帧的数据:

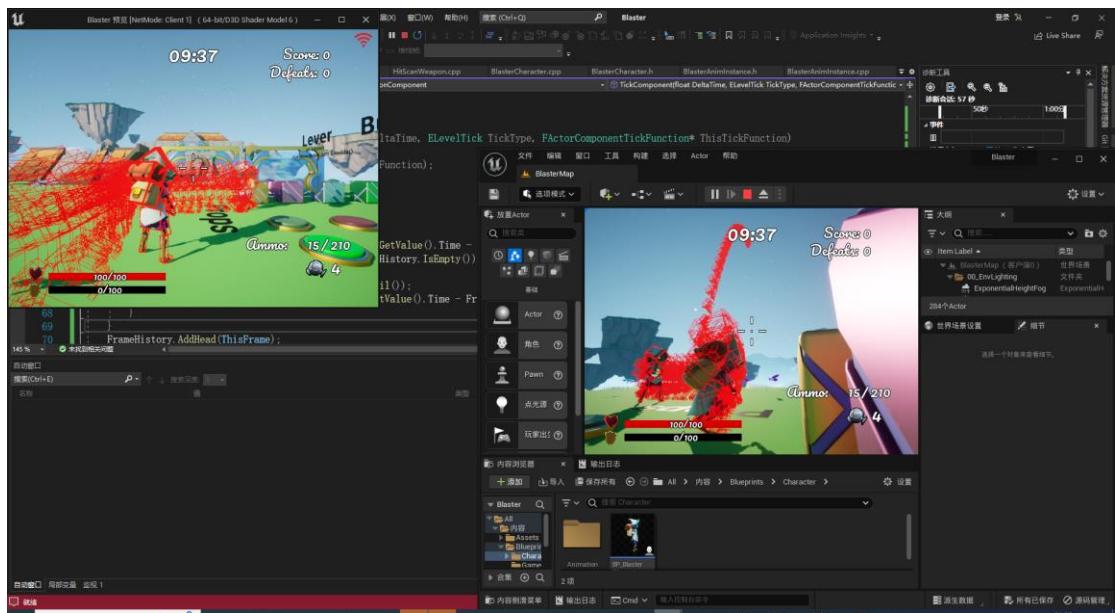
```
UPROPERTY()
class ABlasterPlayerController* Controller;

TDoubleLinkedList<FFramePackage>FrameHistory;
```

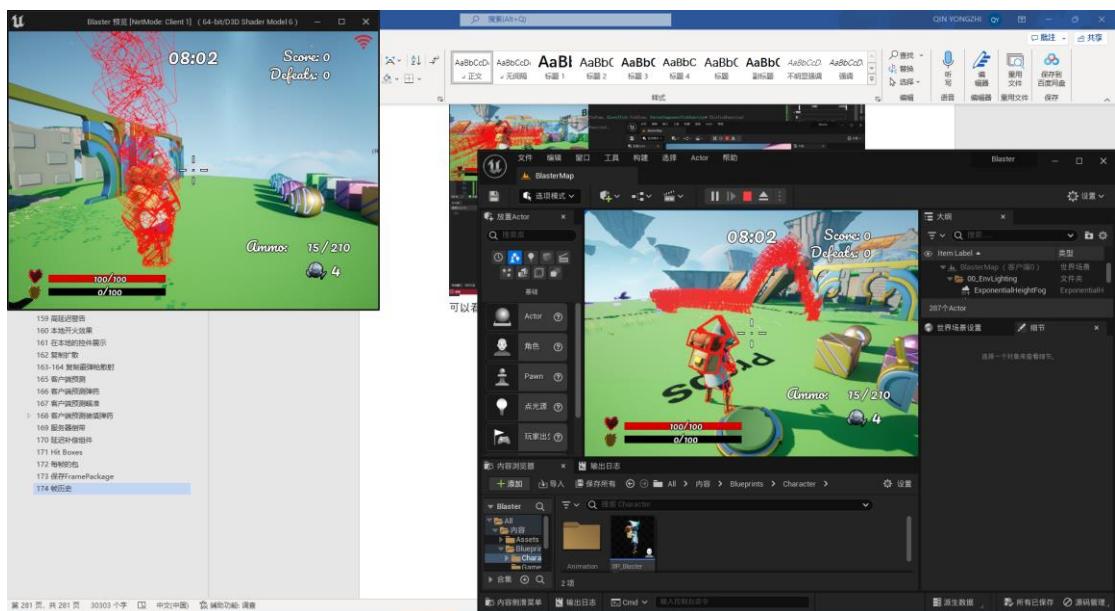
我们在 Tick 函数中添加如下代码来保存每一帧的包，并且 Debug 显示出来:

```
void ULagCompensationComponent::TickComponent(float DeltaTime, ELevelTick TickType,
FActorComponentTickFunction* ThisTickFunction)
{
    Super::TickComponent(DeltaTime, TickType, ThisTickFunction);

    FFramePackage ThisFrame;
    SaveFramePackage(ThisFrame);
    if (FrameHistory.Num() > 1)
    {
        float HistoryLength = FrameHistory.GetHead()->GetValue().Time -
FrameHistory.GetTail()->GetValue().Time;
        while (HistoryLength > MaxRecordTime && !FrameHistory.IsEmpty())
        {
            FrameHistory.RemoveNode(FrameHistory.GetTail());
            HistoryLength = FrameHistory.GetHead()->GetValue().Time -
FrameHistory.GetTail()->GetValue().Time;
        }
    }
    FrameHistory.AddHead(ThisFrame);
    ShowFramePackage(ThisFrame, FColor::Red);
}
```



可以看到我们 4 秒之内的帧都被记录了下来。



175-176 倒带时间&帧的插值

在延迟补偿组件中添加一个服务器倒带函数:

```
void ULagCompensationComponent::ServerSideRewind(ABlasterCharacter* HitCharacter, const FVector_NetQuantize& TraceStart, const FVector_NetQuantize& HitLocation, float HitTime)
{
    bool bReturn =
        HitCharacter == nullptr ||
        HitCharacter->GetLagCompensationComponent() == nullptr ||
        HitCharacter->GetLagCompensationComponent()->FrameHistory.GetHead() == nullptr
    ||
        HitCharacter->GetLagCompensationComponent()->FrameHistory.GetTail() ==
    nullptr;
    if (bReturn)
    {
        return;
    }
    // Frame package that we check to verify a hit
    FFramePackage FrameToCheck;
    bool bShouldInterpolate = true;
    // Frame history of the HitCharacter
    const TDoubleLinkedList<FFramePackage>& History =
        HitCharacter->GetLagCompensationComponent()->FrameHistory;
    const float OldestHistoryTime = History.GetTail()->GetValue().Time;
    const float NewestHistoryTime = History.GetHead()->GetValue().Time;
    if (OldestHistoryTime > HitTime)
    {
        // too far back - too laggy to do SSR
        return;
    }
    if (OldestHistoryTime == HitTime)
    {
        FrameToCheck = History.GetTail()->GetValue();
        bShouldInterpolate = false;
    }
    if (NewestHistoryTime <= HitTime)
    {
        FrameToCheck = History.GetHead()->GetValue();
        bShouldInterpolate = false;
    }

    TDoubleLinkedList<FFramePackage>::TDoubleLinkedListNode* Younger =
        History.GetHead();
    TDoubleLinkedList<FFramePackage>::TDoubleLinkedListNode* Older = Younger;
```

```

while(Older->GetNextNode() && Older->GetValue().Time > HitTime) // is Older still
younger than Hittime?
{
    // March back until: OlderTime < HitTime < YoungerTime
    Older = Older->GetNextNode();
    if (Older->GetValue().Time > HitTime)
    {
        Younger = Older;
    }
}
if(Older->GetValue().Time == HitTime) // highly unlikely, but we found our frame to
check
{
    FrameToCheck = Older->GetValue();
    bShouldInterpolate = false;
}

if (bShouldInterpolate)
{
    // Interpolate between Younger and Older
}
}

```

关于插值的部分，要注意在旋转处使用 RIInterp，关于区别可以看 games104 中的介绍，此外我在这里在 Extend 的部分也用了插值。

```

FFramePackage InterpBetweenFrames(const FFramePackage& OlderFrame, const FFramePackage&
YoungerFrame, float HitTime)
{
    const float Distance = YoungerFrame.Time - OlderFrame.Time;
    const float InterpFraction = FMath::Clamp((HitTime - OlderFrame.Time) / Distance,
0. f, 1. f);

    FFramePackage InterpFramePackage;
    InterpFramePackage.Time = HitTime;

    for (auto& YoungerPair : YoungerFrame.HitBoxInfo)
    {
        const FName& BoxInfoName = YoungerPair.Key;

        const FBoxInformation& OlderBox = OlderFrame.HitBoxInfo[BoxInfoName];
        const FBoxInformation& YoungerBox = YoungerFrame.HitBoxInfo[BoxInfoName];

        FBoxInformation InterpBoxInfo;
        InterpBoxInfo.Location = FMath::VInterpTo(OlderBox.Location,

```

```
YoungerBox.Location, 1.f, InterpFraction);
InterpBoxInfo.Rotation = FMath::RInterpTo(OlderBox.Rotation,
YoungerBox.Rotation, 1.f, InterpFraction);
InterpBoxInfo.BoxExtent = FMath::VInterpTo(OlderBox.BoxExtent,
YoungerBox.BoxExtent, 1.f, InterpFraction);

InterpFramePackage.HitBoxInfo.Add(BoxInfoName, InterpBoxInfo);
}

return InterpFramePackage;
```

最后，暂时我们还没有把插值函数添加到服务器倒带中，我们将在之后进行操作。

177 确认击中

在组件中添加一个确认击中的函数，首先我们要保存下当前帧的状态，然后将盒体移动到我们之前计算好的插值帧位置并开启 visibility 通道的碰撞，同时关闭 character 的 mesh 碰撞，我们可以首先开启 head 的碰撞，检测是否爆头，如果没有爆头，我们可以开启所有碰撞检测是否击中，最后返回结果：

```
USTRUCT(BlueprintType)
struct FServerSideRewindResult
{
    GENERATED_BODY()

    UPROPERTY()
    bool bHitConfirmed;

    UPROPERTY()
    bool bHeadShot;
};
```

在.cpp 中我们添加以下函数：

```
FServerSideRewindResult ULagCompensationComponent::ServerSideRewind(ABlasterCharacter*
HitCharacter, const FVector_NetQuantize& TraceStart, const FVector_NetQuantize&
HitLocation, float HitTime)
{
    bool bReturn =
        HitCharacter == nullptr ||
        HitCharacter->GetLagCompensationComponent() == nullptr ||
        HitCharacter->GetLagCompensationComponent()->FrameHistory.GetHead() == nullptr
        ||
        HitCharacter->GetLagCompensationComponent()->FrameHistory.GetTail() ==
    nullptr;
    if (bReturn)
    {
        return FServerSideRewindResult();
    }
    // Frame package that we check to verify a hit
    FFramePackage FrameToCheck;
    bool bShouldInterpolate = true;
    // Frame history of the HitCharacter
    const TDoubleLinkedList<FFramePackage>& History =
        HitCharacter->GetLagCompensationComponent()->FrameHistory;
    const float OldestHistoryTime = History.GetTail()->GetValue().Time;
    const float NewestHistoryTime = History.GetHead()->GetValue().Time;
    if (OldestHistoryTime > HitTime)
    {
```

```

        // too far back - too laggy to do SSR
        return FServerSideRewindResult();
    }

    if (OldestHistoryTime == HitTime)
    {
        FrameToCheck = History.GetTail()->GetValue();
        bShouldInterpolate = false;
    }

    if (NewestHistoryTime <= HitTime)
    {
        FrameToCheck = History.GetHead()->GetValue();
        bShouldInterpolate = false;
    }

    TDoubleLinkedList<FFramePackage>::TDoubleLinkedListNode* Younger =
History.GetHead();

    TDoubleLinkedList<FFramePackage>::TDoubleLinkedListNode* Older = Younger;
    while(Older->GetNextNode() && Older->GetValue().Time > HitTime) // is Older still
younger than HitTime?
    {

        // March back until: OlderTime < HitTime < YoungerTime
        Older = Older->GetNextNode();
        if (Older->GetValue().Time > HitTime)
        {
            Younger = Older;
        }
    }

    if (Older->GetValue().Time == HitTime) // highly unlikely, but we found our frame to
check
    {
        FrameToCheck = Older->GetValue();
        bShouldInterpolate = false;
    }

    if (bShouldInterpolate)
    {
        // Interpolate between Younger and Older
        FrameToCheck = InterpBetweenFrames(Older->GetValue(), Younger->GetValue(),
HitTime);
    }

    return ConfirmHit(FrameToCheck, HitCharacter, TraceStart, HitLocation);
}

FFramePackage ULagCompensationComponent::InterpBetweenFrames(const FFramePackage&
OlderFrame, const FFramePackage& YoungerFrame, float HitTime)

```

```

{
    const float Distance = YoungerFrame.Time - OlderFrame.Time;
    const float InterpFraction = FMath::Clamp((HitTime - OlderFrame.Time) / Distance,
0. f, 1. f);

    FFramePackage InterpFramePackage;
    InterpFramePackage.Time = HitTime;

    for (auto& YoungerPair : YoungerFrame.HitBoxInfo)
    {
        const FName& BoxInfoName = YoungerPair.Key;

        const FBoxInformation& OlderBox = OlderFrame.HitBoxInfo[BoxInfoName];
        const FBoxInformation& YoungerBox = YoungerFrame.HitBoxInfo[BoxInfoName];

        FBoxInformation InterpBoxInfo;
        InterpBoxInfo.Location = FMath::VInterpTo(OlderBox.Location,
YoungerBox.Location, 1. f, InterpFraction);
        InterpBoxInfo.Rotation = FMath::RInterpTo(OlderBox.Rotation,
YoungerBox.Rotation, 1. f, InterpFraction);
        InterpBoxInfo.BoxExtent = FMath::VInterpTo(OlderBox.BoxExtent,
YoungerBox.BoxExtent, 1. f, InterpFraction);

        InterpFramePackage.HitBoxInfo.Add(BoxInfoName, InterpBoxInfo);
    }

    return InterpFramePackage;
}

FServerSideRewindResult ULagCompensationComponent::ConfirmHit(const FFramePackage&
Package, ABlasterCharacter* HitCharacter, const FVector_NetQuantize& TraceStart, const
FVector_NetQuantize& HitLocaltion)
{
    if (HitCharacter == nullptr)
    {
        return FServerSideRewindResult();
    }

    FFramePackage CurrentFrame;
    CacheBoxPositions(HitCharacter, CurrentFrame);
    MoveBoxes(HitCharacter, Package);
    EnableCharacterMeshCollision(HitCharacter, ECollisionEnabled::NoCollision);

    // Enable collision for the head first
    UBoxComponent* HeadBox = HitCharacter->HitCollisionBoxes[FName("head")];
}

```

```

HeadBox->SetCollisionEnabled(ECollisionEnabled::QueryAndPhysics);
HeadBox->SetCollisionResponseToChannel(ECollisionChannel::ECC_Visibility,
ECollisionResponse::ECR_Block);

FHitResult ConfirmHitResult;
const FVector TraceEnd = TraceStart + (HitLocaltion - TraceStart) * 1.25f;
UWorld* World = GetWorld();
if (World)
{
    World->LineTraceSingleByChannel(
        ConfirmHitResult,
        TraceStart,
        TraceEnd,
        ECollisionChannel::ECC_Visibility
    );
    if (ConfirmHitResult.bBlockingHit)// we hit the head, return ,early
    {
        ResetHitBoxes(HitCharacter, CurrentFrame);
        EnableCharacterMeshCollision(HitCharacter,
ECollisionEnabled::QueryAndPhysics);
        return FServerSideRewindResult{ true, true };
    }
    else // Didn't hit head, check the rest of the boxes
    {
        for (auto& HitBoxPair : HitCharacter->HitCollisionBoxes)
        {
            if (HitBoxPair.Value != nullptr)
            {

                HitBoxPair.Value->SetCollisionEnabled(ECollisionEnabled::QueryAndPhysics);

                HitBoxPair.Value->SetCollisionResponseToChannel(ECollisionChannel::ECC_Visibility,
ECollisionResponse::ECR_Block);
            }
        }
        World->LineTraceSingleByChannel(
            ConfirmHitResult,
            TraceStart,
            TraceEnd,
            ECollisionChannel::ECC_Visibility
        );
        if (ConfirmHitResult.bBlockingHit)// we hit the head, return ,early
        {
            ResetHitBoxes(HitCharacter, CurrentFrame);
        }
    }
}

```

```

        EnableCharacterMeshCollision(HitCharacter,
ECollisionEnabled::QueryAndPhysics);
            return FServerSideRewindResult{ true , false };
        }
    }
}

ResetHitBoxes(HitCharacter, CurrentFrame);
EnableCharacterMeshCollision(HitCharacter, ECollisionEnabled::QueryAndPhysics);
return FServerSideRewindResult{ false, false };
}

void ULagCompensationComponent::CacheBoxPositions(ABlasterCharacter* HitCharacter,
FFramePackage& OutPackage)
{
    if (HitCharacter == nullptr)
    {
        return;
    }

    for (auto& HitBoxPair : HitCharacter->HitCollisionBoxes)
    {
        if (HitBoxPair.Value != nullptr)
        {
            FBoxInformation BoxInfo;
            BoxInfo.Location = HitBoxPair.Value->GetComponentLocation();
            BoxInfo.Rotation = HitBoxPair.Value->GetComponentRotation();
            BoxInfo.BoxExtent = HitBoxPair.Value->GetScaledBoxExtent();
            OutPackage.HitBoxInfo.Add(HitBoxPair.Key, BoxInfo);
        }
    }
}

void ULagCompensationComponent::MoveBoxes(ABlasterCharacter* HitCharacter, const
FFramePackage& Package)
{
    if (HitCharacter == nullptr)
    {
        return;
    }

    for (auto& HitBoxPair : HitCharacter->HitCollisionBoxes)
    {
        if (HitBoxPair.Value != nullptr)
        {

```

```

    HitBoxPair.Value->SetWorldLocation(Package.HitBoxInfo[HitBoxPair.Key].Location);

    HitBoxPair.Value->SetWorldRotation(Package.HitBoxInfo[HitBoxPair.Key].Rotation);

    HitBoxPair.Value->SetBoxExtent(Package.HitBoxInfo[HitBoxPair.Key].BoxExtent);
}

}

void ULagCompensationComponent::ResetHitBoxes(ABlasterCharacter* HitCharacter, const FFramePackage& Package)
{
    if (HitCharacter == nullptr)
    {
        return;
    }

    for (auto& HitBoxPair : HitCharacter->HitCollisionBoxes)
    {
        if (HitBoxPair.Value != nullptr)
        {

            HitBoxPair.Value->SetWorldLocation(Package.HitBoxInfo[HitBoxPair.Key].Location);

            HitBoxPair.Value->SetWorldRotation(Package.HitBoxInfo[HitBoxPair.Key].Rotation);

            HitBoxPair.Value->SetBoxExtent(Package.HitBoxInfo[HitBoxPair.Key].BoxExtent);
            HitBoxPair.Value->SetCollisionEnabled(ECollisionEnabled::NoCollision);
        }
    }
}

void ULagCompensationComponent::EnableCharacterMeshCollision(ABlasterCharacter*
HitCharacter, ECollisionEnabled::Type CollisionEnabled)
{
    if (HitCharacter && HitCharacter->GetMesh())
    {
        HitCharacter->GetMesh()->SetCollisionEnabled(CollisionEnabled);
    }
}

```

现在，最后的问题就是我们什么时候调用服务器倒带函数。

178 分数请求

创建一个服务器 RPC:

```
void ULagCompensationComponent::ServerScoreRequest_Implementation(ABlasterCharacter*
HitCharacter, const FVector_NetQuantize& TraceStart, const FVector_NetQuantize&
HitLocation, float HitTime, AWeapon* DamageCauser)
{
    FServerSideRewindResult Confirm = ServerSideRewind(HitCharacter, TraceStart,
    HitLocation, HitTime);

    if (Character && HitCharacter && Confirm.bHitConfirmed && DamageCauser)
    {
        UGameplayStatics::ApplyDamage(
            HitCharacter,
            DamageCauser->GetDamage(),
            Character->Controller,
            DamageCauser,
            UDamageType::StaticClass()
        );
    }
}
```

在 HitScanWeapon 的 Fire 中，修改 applydamage 相关的代码，在本地命中之后发送击中的请求，触发服务器倒带：

```
void AHitScanWeapon::Fire(const FVector_NetQuantize& SocketLocation, const
FVector_NetQuantize& HitTarget)
{
    Super::Fire(SocketLocation, HitTarget);

    UWorld* World = GetWorld();

    APawn* OwnerPawn = Cast<APawn>(GetOwner());
    if (OwnerPawn == nullptr)
    {
        return;
    }
    AController* InstigatorController = OwnerPawn->GetController();

    FHitResult FireHit;
    WeaponTraceHit(SocketLocation, HitTarget, FireHit);

    ABlasterCharacter* BlasterCharacter = Cast<ABlasterCharacter>(FireHit.GetActor());
    if (BlasterCharacter)
    {
```

```

        if (InstigatorController)
        {
            if (HasAuthority() && !bUseServerSideRewind)
            {
                UGameplayStatics::ApplyDamage(
                    BlasterCharacter,
                    Damage,
                    InstigatorController,
                    this,
                    UDamageType::StaticClass()
                );
            }
            if (!HasAuthority() && bUseServerSideRewind)
            {
                BlasterOwnerCharacter = BlasterOwnerCharacter == nullptr ?
                    Cast<ABlasterCharacter>(OwnerPawn) : BlasterOwnerCharacter;
                BlasterOwnerController = BlasterOwnerController == nullptr ?
                    Cast<ABlasterPlayerController>(InstigatorController) : BlasterOwnerController;
                if (BlasterOwnerCharacter && BlasterOwnerController &&
                    BlasterOwnerCharacter->GetLagCompensationComponent())
                {

                    BlasterOwnerCharacter->GetLagCompensationComponent()->ServerScoreRequest(
                        BlasterCharacter,
                        SocketLocation,
                        FireHit.ImpactPoint,
                        BlasterOwnerController->GetServerTime() -
                    BlasterOwnerController->SingleTripTime,
                        this
                    );
                }
            }
        }
        if (HitSound)
        {
            UGameplayStatics::PlaySoundAtLocation(
                this,
                HitSound,
                FireHit.ImpactPoint
            );
        }
    }
    else
    {

```

```
    if (ImpactSound)
    {
        UGameplayStatics::PlaySoundAtLocation(
            this,
            ImpactSound,
            FireHit.ImpactPoint
        );
    }
    if (ImpactParticles)
    {
        UGameplayStatics::SpawnEmitterAtLocation(
            World,
            ImpactParticles,
            FireHit.ImpactPoint,
            FireHit.ImpactNormal.Rotation()
        );
    }
    if (MuzzleFlash)
    {
        UGameplayStatics::SpawnEmitterAtLocation(
            World,
            MuzzleFlash,
            SocketLocation
        );
    }
    if (FireSound)
    {
        UGameplayStatics::PlaySoundAtLocation(
            this,
            FireSound,
            SocketLocation
        );
    }
}
```

179 霰弹枪的服务器倒带

定义一个数据结构保存 shotgun 射击的结果:

```
USTRUCT(BlueprintType)
struct FShotgunServerSideRewindResult
{
    GENERATED_BODY()

    UPROPERTY()
    TMap<ABlasterCharacter*, int32> HeadShots;

    UPROPERTY()
    TMap<ABlasterCharacter*, int32> BodyShots;
};
```

定义两个函数:

```
/***
 * Shotgun
 ***/
FShotgunServerSideRewindResult ShotgunServerSideRewind(
    const TArray<ABlasterCharacter*>& HitCharacters,
    const FVector_NetQuantize& TraceStart,
    const TArray<FVector_NetQuantize>& HitLocations,
    float HitTime
);
FShotgunServerSideRewindResult ShotgunConfirmHit(
    const TArray<FFramePackage>& Packages,
    const FVector_NetQuantize& TraceStart,
    const TArray<FVector_NetQuantize>& HitLocations
);
```

之后可以向 HitScanWeapon 一样操作。

180 确认霰弹枪击中

添加一个 ShotgunHitConfirm 函数：

```
FShotgunServerSideRewindResult ULagCompensationComponent::ShotgunConfirmHit(const
TArray<FFramePackage>& FramePackages, const FVector_NetQuantize& TraceStart, const
TArray<FVector_NetQuantize>& HitLocations)
{
    FShotgunServerSideRewindResult ShotgunResult;
    for (auto& Frame : FramePackages)
    {
        if (Frame.Character == nullptr)
        {
            return ShotgunResult;
        }
    }

    // Copy frames
    TArray<FFramePackage> CurrentFrames;
    for (auto& Frame : FramePackages)
    {
        FFramePackage CurrentFrame;
        auto HitCharacter = Frame.Character;
        CurrentFrame.Character = Frame.Character;
        CacheBoxPositions(HitCharacter, CurrentFrame);
        MoveBoxes(HitCharacter, Frame);
        EnableCharacterMeshCollision(HitCharacter, ECollisionEnabled::NoCollision);
        CurrentFrames.Add(CurrentFrame);
    }

    // Enable collision for the head first
    for (auto& Frame : FramePackages)
    {
        auto HitCharacter = Frame.Character;
        UBoxComponent* HeadBox = HitCharacter->HitCollisionBoxes[FName("head")];
        HeadBox->SetCollisionEnabled(ECollisionEnabled::QueryAndPhysics);
        HeadBox->SetCollisionResponseToChannel(ECollisionChannel::ECC_Visibility,
ECollisionResponse::ECR_Block);
    }

    UWorld* World = GetWorld();
    // Check for head shots
    for (auto& HitLocation : HitLocations)
    {
        FHitResult ConfirmHitResult;
```

```

const FVector TraceEnd = TraceStart + (HitLocation - TraceStart) * 1.25f;
if (World)
{
    World->LineTraceSingleByChannel(
        ConfirmHitResult,
        TraceStart,
        TraceEnd,
        ECollisionChannel::ECC_Visibility
    );
    ABlasterCharacter* BlasterCharacter =
    Cast<ABlasterCharacter>(ConfirmHitResult.GetActor());
    if (BlasterCharacter)
    {
        if (ShotgunResult.HeadShots.Contains(BlasterCharacter))
        {
            ShotgunResult.HeadShots[BlasterCharacter]++;
        }
        else
        {
            ShotgunResult.HeadShots.Emplace(BlasterCharacter, 1);
        }
    }
}

// Enable collision for all boxes, then disable for head box
for (auto& Frame : FramePackages)
{
    auto HitCharacter = Frame.Character;
    for (auto& HitBoxPair : HitCharacter->HitCollisionBoxes)
    {
        if (HitBoxPair.Value != nullptr)
        {

            HitBoxPair.Value->SetCollisionEnabled(ECollisionEnabled::QueryAndPhysics);

            HitBoxPair.Value->SetCollisionResponseToChannel(ECollisionChannel::ECC_Visibility,
ECollisionResponse::ECR_Block);
        }
    }
    UBoxComponent* HeadBox = HitCharacter->HitCollisionBoxes[FName("head")];
    HeadBox->SetCollisionEnabled(ECollisionEnabled::NoCollision);
}

```

```

// Check for body shots
for (auto& HitLocation : HitLocations)
{
    FHitResult ConfirmHitResult;
    const FVector TraceEnd = TraceStart + (HitLocation - TraceStart) * 1.25f;
    if (World)
    {
        World->LineTraceSingleByChannel(
            ConfirmHitResult,
            TraceStart,
            TraceEnd,
            ECollisionChannel::ECC_Visibility
        );
        ABlasterCharacter* BlasterCharacter =
        Cast<ABlasterCharacter>(ConfirmHitResult.GetActor());
        if (BlasterCharacter)
        {
            if (ShotgunResult.BodyShots.Contains(BlasterCharacter))
            {
                ShotgunResult.BodyShots[BlasterCharacter]++;
            }
            else
            {
                ShotgunResult.BodyShots.Emplace(BlasterCharacter, 1);
            }
        }
    }
}

// Reset hit boxes
for (auto& CurrentFrame : CurrentFrames)
{
    ResetHitBoxes(CurrentFrame.Character, CurrentFrame);
    EnableCharacterMeshCollision(CurrentFrame.Character,
        ECollisionEnabled::QueryAndPhysics);
}

return ShotgunResult;
}

```

ISSUE

这个函数有问题，因为被前面角色挡住的子弹仍然可以击中后面角色的头部，之后如果有空隙来进行修复。

181-182 Shotgun Score Request && Requesting a Shotgun Hit

ISSUE

这一章和之前的扫描武器类似，但是值得注意的是，作者的代码中存在一些错误，我进行了修正：

首先是霰弹枪的 RPC：

```
void ULagCompensationComponent::ShotgunServerScoreRequest_Implementation(const  
TArray<ABlasterCharacter*>& HitCharacters, const FVector_NetQuantize& TraceStart, const  
TArray<FVector_NetQuantize>& HitLocations, float HitTime)  
{  
    FShotgunServerSideRewindResult Confirm = ShotgunServerSideRewind(HitCharacters,  
TraceStart, HitLocations, HitTime);  
    for (auto HitCharacter : HitCharacters)  
    {  
        if (HitCharacter == nullptr || Character->GetEquippedWeapon() == nullptr ||  
Character == nullptr)  
        {  
            continue;  
        }  
        float TotalDamage = 0.f;  
        if (Confirm.HeadShots.Contains(HitCharacter))  
        {  
            float HeadShotDamage = Confirm.HeadShots[HitCharacter] *  
Character->GetEquippedWeapon()->GetDamage();  
            TotalDamage += HeadShotDamage;  
        }  
        if (Confirm.BodyShots.Contains(HitCharacter))  
        {  
            float BodyShotDamage = Confirm.BodyShots[HitCharacter] *  
Character->GetEquippedWeapon()->GetDamage();  
            TotalDamage += BodyShotDamage;  
        }  
        UGameplayStatics::ApplyDamage(  
            HitCharacter,  
            TotalDamage,  
            Character->Controller,  
            Character->GetEquippedWeapon(),  
            UDamageType::StaticClass()  
        );  
    }  
}
```

作者这里把 HitCharacter 的位置和 Character 的位置弄混淆了，计算伤害的时候使用的是受

击者的武器的伤害。

其次在之前 HitScanWeapon 和这次的 ShotgunWeapon 应用伤害的时候，判断的逻辑有一些问题，导致只能在非服务器的玩家拥有的武器才能造成伤害，我也进行了修改。

最后由于之前霰弹枪命中的问题，即使是同一个人也可能会被同一颗子弹造成两次伤害（在两次射线检测中分别命中头和身子）

最后换弹的问题很明显，在使用弹药预测的时候可能出现武器的 Sequence 值不能降到 0 的情况，导致后续一直缺少子弹。UE 的 URO 带来的问题？

183 预测 Projectile 的轨迹

UE 自带了函数用来预测 Projectile 的轨迹：

```
UGameplayStatics::PredictProjectilePath(this, PathParams, PathResult);
```

来到 ProjectileBullet 中输入以下代码，可以在测试中看到子弹轨迹：

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

    FPredictProjectilePathParams PathParams;
    PathParams.bTraceWithChannel = true;
    PathParams.bTraceWithCollision = true;
    PathParams.DrawDebugTime = 5.f;
    PathParams.DrawDebugType = EDrawDebugTrace::ForDuration;
    PathParams.LaunchVelocity = GetActorForwardVector() *
        ProjectileMovementComponent->InitialSpeed;
    PathParams.MaxSimTime = 4.f;
    PathParams.ProjectileRadius = 5.f;
    PathParams.SimFrequency = 30.f;
    PathParams.StartLocation = GetActorLocation();
    PathParams.TraceChannel = ECollisionChannel::ECC_Visibility;
    PathParams.ActorsToIgnore.Add(this);

    FPredictProjectilePathResult PathResult;

    UGameplayStatics::PredictProjectilePath(this, PathParams, PathResult);
}
```



为了在接下来的服务器倒带中使用预测轨迹，我们在 Projectile 中添加一些变量：

```
/**  
 * Used with Server-side Rewind  
 */  
bool bUseServerSideRewind = false;  
FVector_NetQuantize TraceStart;  
FVector_NetQuantize100 InitialVelocity;  
  
UPROPERTY(EditAnywhere)  
float InitialSpeed = 15000.f;
```

接下来我们只需要在 Projectile 碰撞发生的时候进行服务器倒带，获取碰撞时的角色的帧，然后再用 Projectile 所带的信息进行投掷物轨迹预测即可。

184 Post Edit Change Property

我们吧 InitialSpeed 设置为 Projectile 中的一个变量，但是在我们设置他的值的时候，并不能实时的改变 ProjectileMovementComponent 中的速度，为了解决这个问题我们需要用到一个 UE 的函数：

```
#if WITH_EDITOR
void AProjectileBullet::PostEditChangeProperty(FPropertyChangedEvent& Event)
{
    Super::PostEditChangeProperty(Event);

    FNamePropertyName = Event.Property != nullptr ? Event.Property->GetFName() :
    NAME_None;
    if (PropertyName == GET_MEMBER_NAME_CHECKED(AProjectileBullet, InitialSpeed))
    {
        if (ProjectileMovementComponent)
        {
            ProjectileMovementComponent->InitialSpeed = InitialSpeed;
            ProjectileMovementComponent->MaxSpeed = InitialSpeed;
        }
    }
}
#endif
```

这样我们就可在设置 InitialSpeed 的值的时候一起更新 ProjectileMovementComponent 中的速度的值了。

185 本地生成投掷物

首先我们需要在 ProjectileWeapon 中添加另一种子弹，用于在有 SSR 的时候使用，这个子弹和之前普通的子弹区别在于之前的子弹是 Replicates = true 的，而这个子弹需要关闭复制。

复制一个之前的子弹蓝图，关闭复制作为 SSR 的子弹。

接下来需要修改 ProjectileWeapon 的逻辑，用于在不同情况下生成子弹：

```
void AProjectileWeapon::SpawnProjectile(const FVector_NetQuantize& SocketLocation,
const FVector_NetQuantize& HitTarget)
{
    APawn* InstigatorPawn = Cast<APawn>(GetOwner());
    // From muzzle flash socket to hit location from TraceUnderCrosshairs
    FVector ToTarget = HitTarget - SocketLocation;
    FRotator TargetRotation = ToTarget.Rotation();
    UWorld* World = GetWorld();
    if (World)
    {
        FActorSpawnParameters SpawnParams;
        SpawnParams.Owner = GetOwner();
        SpawnParams.Instigator = InstigatorPawn;

        if (InstigatorPawn == nullptr)
        {
            return;
        }

        AProjectile* SpawnerProjectile = nullptr;
        if (bUseServerSideRewind)
        {
            if (InstigatorPawn->HasAuthority()) // server
            {
                if (InstigatorPawn->IsLocallyControlled()) //server, host - use
replicated projectile
            }
            if (ProjectileClass)
            {
                SpawnerProjectile =
World->SpawnActor<AProjectile>(ProjectileClass, SocketLocation, TargetRotation,
SpawnParams);
                SpawnerProjectile->bUseServerSideRewind = false;
                SpawnerProjectile->Damage = Damage;
            }
        }
    }
}
```

```

        else // server, not locally controlled - spawn non-replicated
projectile, no SSR
    {
        if (ServerSideRewindProjectileClass)
        {
            SpawnerProjectile =
World->SpawnActor<AProjectile>(ServerSideRewindProjectileClass, SocketLocation,
TargetRotation, SpawnParams);
            SpawnerProjectile->bUseServerSideRewind = false;
        }
    }
}

else // client, using SSR
{
    if (InstigatorPawn->IsLocallyControlled()) // client, locally
controlled - spawn non-replicated projectile, use SSR
    {
        if (ServerSideRewindProjectileClass)
        {
            SpawnerProjectile =
World->SpawnActor<AProjectile>(ServerSideRewindProjectileClass, SocketLocation,
TargetRotation, SpawnParams);
            SpawnerProjectile->bUseServerSideRewind = true;
            SpawnerProjectile->TraceStart = SocketLocation;
            SpawnerProjectile->InitialVelocity =
SpawnerProjectile->GetActorForwardVector() * SpawnerProjectile->InitialSpeed;
            SpawnerProjectile->Damage = Damage;
        }
    }
    else // client, not locally controlled - spawn non-replicated
projectile, no SSR
    {
        if (ServerSideRewindProjectileClass)
        {
            SpawnerProjectile =
World->SpawnActor<AProjectile>(ServerSideRewindProjectileClass, SocketLocation,
TargetRotation, SpawnParams);
            SpawnerProjectile->bUseServerSideRewind = false;
        }
    }
}
else // Weapon not using SSR
{

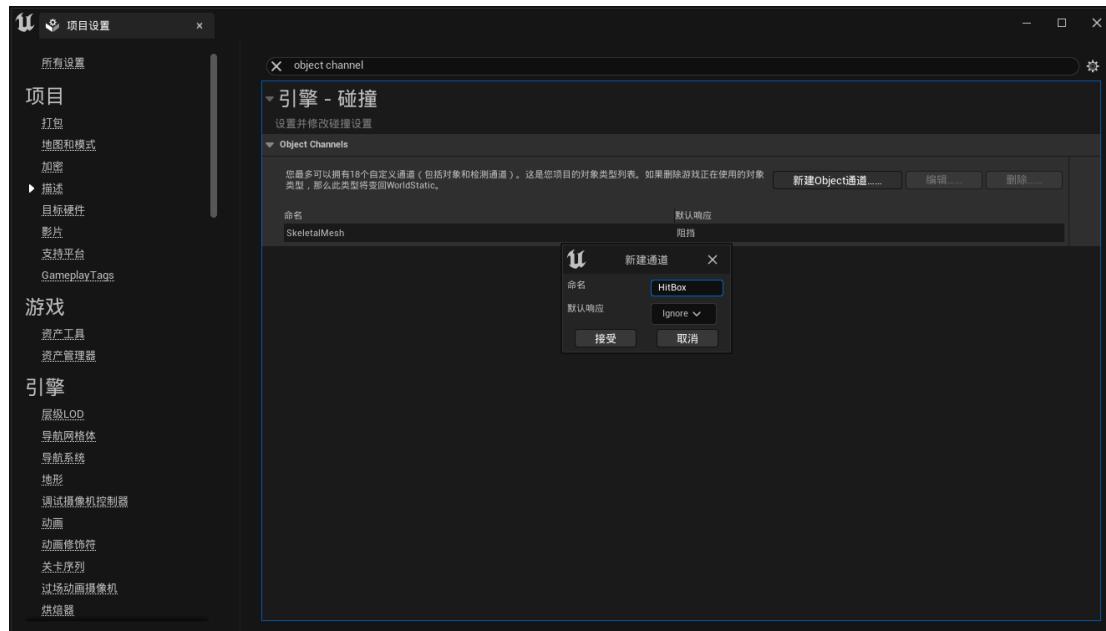
```

```
    if (InstigatorPawn->HasAuthority()) //server, host - use replicated
projectile
    {
        if (ProjectileClass)
        {
            SpawnerProjectile =
World->SpawnActor<AProjectile>(ProjectileClass, SocketLocation, TargetRotation,
SpawnParams);
            SpawnerProjectile->bUseServerSideRewind = false;
            SpawnerProjectile->Damage = Damage;
        }
    }
}
}
```

进行测试可以看到，没有设置 SSR 的武器在开火之后到子弹出现有明显的延迟，而使用了 SSR 的武器则可以立刻生成子弹。

186 HitBox 的碰撞种类

创建一个新的碰撞通道：



在 Character 构造函数的底端添加：

```
for (auto Box : HitCollisionBoxes)
{
    if (Box.Value)
    {
        Box.Value->SetCollisionObjectType(ECC_HitBox);

        Box.Value->SetCollisionResponseToAllChannels(ECollisionResponse::ECR_Ignore);
        Box.Value->SetCollisionResponseToChannel(ECollisionChannel(ECollisionResponse::ECR_Block));
        Box.Value->SetCollisionEnabled(ECollisionEnabled::NoCollision);
    }
}
```

在延迟补偿组件中修改之前设置的 hitscanWeapon 和 shotgun 的射线检测的通道到 HitBox。

187 投掷物服务器倒带

结构和 HitScan 一样，只是把射线检测换成了投掷物路径预测：

```
FServerSideRewindResult ULagCompensationComponent::ProjectileConfirmHit(const
FFramePackage& Package, ABlasterCharacter* HitCharacter, const FVector_NetQuantize&
TraceStart, const FVector_NetQuantize100& InitialVelocity)
{
    if (HitCharacter == nullptr)
    {
        return FServerSideRewindResult();
    }

    FFramePackage CurrentFrame;
    CacheBoxPositions(HitCharacter, CurrentFrame);
    MoveBoxes(HitCharacter, Package);
    EnableCharacterMeshCollision(HitCharacter, ECollisionEnabled::NoCollision);

    // Enable collision for the head first
    UBoxComponent* HeadBox = HitCharacter->HitCollisionBoxes[FName("head")];
    HeadBox->SetCollisionEnabled(ECollisionEnabled::QueryAndPhysics);
    HeadBox->SetCollisionResponseToChannel(ECC_HitBox, ECollisionResponse::ECR_Block);

    FPredictProjectilePathParams PathParams;
    PathParams.bTraceWithChannel = true;
    PathParams.bTraceWithCollision = true;
    PathParams.MaxSimTime = MaxRecordTime;
    PathParams.LaunchVelocity = InitialVelocity;
    PathParams.StartLocation = TraceStart;
    PathParams.SimFrequency = 15.f;
    PathParams.ProjectileRadius = 5.f;
    PathParams.TraceChannel = ECC_HitBox;
    PathParams.ActorsToIgnore.Add(GetOwner());
    PathParams.DrawDebugTime = 5.f;
    PathParams.DrawDebugType = EDrawDebugTrace::ForDuration;
    FPredictProjectilePathResult PathResult;
    UGameplayStatics::PredictProjectilePath(this, PathParams, PathResult);

    if (PathResult.HitResult.bBlockingHit) // hit the head return early
    {
        if (PathResult.HitResult.Component.IsValid())
        {
            UBoxComponent* Box = Cast<UBoxComponent>(PathResult.HitResult.Component);
            if (Box)
            {

```

```

        DrawDebugBox(
            GetWorld(),
            Box->GetComponentLocation(),
            Box->GetScaledBoxExtent(),
            FQuat(Box->GetComponentRotation()),
            FColor::Red,
            true
        );
    }
}

ResetHitBoxes(HitCharacter, CurrentFrame);
EnableCharacterMeshCollision(HitCharacter,
ECollisionEnabled::QueryAndPhysics);
return FServerSideRewindResult{ true, true };
}

else // we didn't hit the head
{
    for (auto& HitBoxPair : HitCharacter->HitCollisionBoxes)
    {
        if (HitBoxPair.Value != nullptr)
        {

            HitBoxPair.Value->SetCollisionEnabled(ECollisionEnabled::QueryAndPhysics);
            HitBoxPair.Value->SetCollisionResponseToChannel(ECC_HitBox,
ECollisionResponse::ECR_Block);
        }
    }

    UGameplayStatics::PredictProjectilePath(this, PathParams, PathResult);
    if (PathResult.HitResult.bBlockingHit)
    {
        if (PathResult.HitResult.Component.IsValid())
        {
            UBoxComponent* Box =
Cast<UBoxComponent>(PathResult.HitResult.Component);
            if (Box)
            {
                DrawDebugBox(
                    GetWorld(),
                    Box->GetComponentLocation(),
                    Box->GetScaledBoxExtent(),
                    FQuat(Box->GetComponentRotation()),
                    FColor::Green,
                    true
                );
            }
        }
    }
}

```

```
        }

    }

    ResetHitBoxes(HitCharacter, CurrentFrame);
    EnableCharacterMeshCollision(HitCharacter,
ECollisionEnabled::QueryAndPhysics);
    return FServerSideRewindResult{ true, false };
}

}

ResetHitBoxes(HitCharacter, CurrentFrame);
EnableCharacterMeshCollision(HitCharacter, ECollisionEnabled::QueryAndPhysics);
return FServerSideRewindResult{ false, false };
}
```

接下来，只需要设置 RPC，并且发送请求调用 RPC 即可。

188 Projectile 分数请求

类似 HitScanWeapon 的结构，实现一个 RPC 函数：

```
void ULagCompensationComponent::ProjectileServerScoreRequest_Implementation(ABlasterCharacter* HitCharacter, const FVector_NetQuantize& TraceStart, const FVector_NetQuantize100& InitialVelocity, float HitTime)
{
    FServerSideRewindResult Confirm = ProjectileServerSideRewind(HitCharacter,
        TraceStart, InitialVelocity, HitTime);

    if (Character && Character->GetEquippedWeapon() && HitCharacter &&
        Confirm.bHitConfirmed)
    {
        UGameplayStatics::ApplyDamage(
            HitCharacter,
            Character->GetEquippedWeapon()->GetDamage(),
            Character->Controller,
            Character->GetEquippedWeapon(),
            UDamageType::StaticClass()
        );
    }
}
```

在 ProjectileBullet 中修改 OnHit 函数，以实现不同情况下的受击伤害和效果使用：

```
void AProjectileBullet::OnHit(UPrimitiveComponent* HitComp, AActor* OtherActor,
    UPrimitiveComponent* OtherComp, FVector NormalImpulse, const FHitResult& Hit)
{
    ABlasterCharacter* OwnerCharacter = Cast<ABlasterCharacter>(GetOwner());
    if (OwnerCharacter)
    {
        ABlasterPlayerController* OwnerController =
        Cast<ABlasterPlayerController>(OwnerCharacter->Controller);
        if (OwnerController)
        {
            if (OwnerCharacter->HasAuthority() && !bUseServerSideRewind)
            {
                UGameplayStatics::ApplyDamage(OtherActor, Damage, OwnerController,
                    this, UDamageType::StaticClass());
                Super::OnHit(HitComp, OtherActor, OtherComp, NormalImpulse, Hit);
                return;
            }
            ABlasterCharacter* HitCharacter = Cast<ABlasterCharacter>(OtherActor);
            if (bUseServerSideRewind && OwnerCharacter->GetLagCompensationComponent()
```

```
&& OwnerCharacter->IsLocallyControlled() && HitCharacter)
    {

        OwnerCharacter->GetLagCompensationComponent () ->ProjectileServerScoreRequest (
            HitCharacter,
            TraceStart,
            InitialVelocity,
            OwnerController->GetServerTime () -
        OwnerController->SingleTripTime
            );
    }
}

Super::OnHit(HitComp, OtherActor, OtherComp, NormalImpulse, Hit);
}
```

BUG

武器在服务端使用后，客户端再使用会出现子弹的 bug。Sequence 不会消去，导致客户端使用时没有子弹甚至负子弹。

189 限制 SSR

启用 SSR 虽然可以让延迟高的玩家获得更好的体验，但是会损害低延迟玩家的体验，所以我们应该限制启用 SSR 的延迟范围。

我们在 Controller 中设置了 PingCheck，我们可以在检查的同时设置武器是否启用 SSR。为此，我们需要把 bUseServerSideRewind 设置为复制变量，在延迟检查的时候在客户端触发了高延迟的话，我们需要一个 ServerRPC 来通知服务器，这时候服务器设置服务器的 Weapon 中的 bUseServerSideRewind 来开启或关闭 SSR。

我们可以用一个代理来设置 Weapon 中的 SSR：

```
DECLARE_DYNAMIC_MULTICAST_DELEGATE_OneParam(FHighPingDelegate, bool, bPingTooHigh);
```

```
void AWeapon::OnPingTooHigh(bool bPingTooHigh)
{
    bUseServerSideRewind = !bPingTooHigh;
}
```

在三个状态切换的函数中进行委托回调函数绑定和解绑，相比于教程的做法，我添加了一个变量用于表示武器原本的 SSR 设置，然后再初始的时候以及丢弃武器和切换到服务器的时候将他设置回原值。

```
void AWeapon::OnEquipped()
{
    ...
    if (BlasterOwnerCharacter && bOriginalUseServerSideRewind)
    {
        BlasterOwnerController = BlasterOwnerController == nullptr ?
            Cast<ABlasterPlayerController>(BlasterOwnerCharacter->Controller) :
            BlasterOwnerController;
        if (BlasterOwnerController && HasAuthority()
            && !BlasterOwnerController->HighPingDelegate.IsBound())
        {
            UE_LOG(LogTemp, Warning, TEXT("Add delegate on Equipped"));
            BlasterOwnerController->HighPingDelegate.AddDynamic(this,
&AWeapon::OnPingTooHigh);
        }
    }
}

void AWeapon::OnEquippedSecondary()
{
    ...
    if (BlasterOwnerCharacter && bOriginalUseServerSideRewind)
    {
```

```

        BlasterOwnerController = BlasterOwnerController == nullptr ?
Cast<ABlasterPlayerController>(BlasterOwnerCharacter->Controller) :
BlasterOwnerController;
        if (BlasterOwnerController && HasAuthority() &&
BlasterOwnerController->HighPingDelegate. IsBound())
{
    UE_LOG(LogTemp, Warning, TEXT("Remove delegate on Equipped Secondary
State"));
    BlasterOwnerController->HighPingDelegate. RemoveDynamic(this,
&AWeapon::OnPingTooHigh);
    bUseServerSideRewind = bOriginalUseServerSideRewind;
}
}

void AWeapon::OnDropped()
{
...
if (BlasterOwnerCharacter && bOriginalUseServerSideRewind)
{
    BlasterOwnerController = BlasterOwnerController == nullptr ?
Cast<ABlasterPlayerController>(BlasterOwnerCharacter->Controller) :
BlasterOwnerController;
    if (BlasterOwnerController && HasAuthority() &&
BlasterOwnerController->HighPingDelegate. IsBound())
{
    UE_LOG(LogTemp, Warning, TEXT("Remove delegate on Dropped"));
    BlasterOwnerController->HighPingDelegate. RemoveDynamic(this,
&AWeapon::OnPingTooHigh);
    bUseServerSideRewind = bOriginalUseServerSideRewind;
}
}
}
}

```

在测试过程中我发现在装备武器时不能触发打印 LOG，原因是在 CombatComponent 中，我们在设置武器的 WeaponState 之后才设置了 Owner，导致无法通过判断从而在装备武器时无法添加委托回调函数。

```

void UCombatComponent::EquipPrimaryWeapon(AWeapon* WeaponToEquip)
{
    DropEquippedWeapon();
    EquippedWeapon = WeaponToEquip;
    EquippedWeapon->SetWeaponState(EWeaponState::EWS_Equipped);
    EquippedWeapon->SetOwner(Character);
    EquippedWeapon->SetWeaponState(EWeaponState::EWS_Equipped);
}

```

```
EquippedWeapon->SetHUDAmmo();  
AttachActorToRightHand(EquippedWeapon);  
UpdateCarriedAmmo();  
PlayEquippedWeaponSound(EquippedWeapon);  
ReloadEmptyWeapon();  
}
```

来到函数中把设置状态移动到最后即可。

190 切枪动画

类似于换弹动画以及换弹动画的本地实现：

现在角色中添加一个动画蒙太奇，蒙太奇中创建两个通知，一个用于设置武器绑定，另一个用于设置切换状态的结束。

设置一个新的 CombatState 用于换弹，在复制通知中调用播放蒙太奇。

为了隐藏延迟，和 Reloading 一样，设置了一个本地变量，用于关闭和开启 FABRIK。

Solve Issue

似乎通过设置蒙太奇通知的属性 Event->Montage Tick Type 为 Branching 可以解决之前上弹后无法触发蒙太奇通知，以及切枪后无法触发蒙太奇通知，导致不能回到 Unoccupied 状态的问题。

191 总结延迟补偿

提出了一些 challenge。

192 作弊与验证

```
UFUNCTION(Server, Reliable, WithValidation)
ServerFire(HitTarget, Damage);

bool ServerFire_Validate(HitTarget, Damage)
{
    if (Damage >= TooMuchDamage)
    {
        return false;    Disconnect the caller!
    }
    return true;    Allow the RPC
}
```

更多多人游戏的特征

193 返回主菜单

创建一个蓝图控件，添加一个按钮。

创建一个 C++ 类，用来实现按钮的功能，由于需要返回主菜单，需要连接我们的插件，所以要在 build.cs 中添加相应的模块。

之后再 Setup 中连接到模块，并且绑定 DestroySession 委托的回调函数，在点击退出后，插件会返回退出是否成功，我们可以据此进行下一部操作。

还要注意的是，客户端和服务器应该调用不同的函数来退出：

服务器调用：GameMode->ReturnToMainMenuHost();

客户端调用：PlayerController->ClientReturnToMainMenuWithTextReason(FText());

```
void UReturnToMainMenu::MenuSetup()
{
    AddToViewport();
    SetVisibility(ESlateVisibility::Visible);
    bIsFocusable = true;

    UWorld* World = GetWorld();
    if (World)
    {
        PlayerController = PlayerController == nullptr ?
World->GetFirstPlayerController(): PlayerController;
        if (PlayerController)
        {
            FInputModeGameAndUI InputModeData;
            InputModeData.SetWidgetToFocus(TakeWidget());
            PlayerController->SetInputMode(InputModeData);
            PlayerController->SetShowMouseCursor(true);
        }
    }

    if (ReturnButton)
    {
        ReturnButton->OnClicked.AddDynamic(this,
&UReturnToMainMenu::ReturnButtonClicked);
    }

    UGameInstance* GameInstance = GetGameInstance();
    if (GameInstance)
```

```

    {

        MultiplayerSessionsSubsystem =
GameInstance->GetSubsystem<UMultiplayerSessionsSubsystem>();

        if (MultiplayerSessionsSubsystem)
        {

            MultiplayerSessionsSubsystem->MultiplayerOnDestroySessionComplete.AddDynamic(this,
&UReturnToMainMenu::OnDestroySession);
        }
    }

}

bool UReturnToMainMenu::Initialize()
{
    if (!Super::Initialize())
    {
        return false;
    }

    return true;
}

void UReturnToMainMenu::OnDestroySession(bool bWasSuccessful)
{
    if (!bWasSuccessful)
    {
        ReturnButton->SetIsEnabled(true);

        return;
    }

    UWorld* World = GetWorld();
    if (World)
    {
        AGameModeBase* GameMode = World->GetAuthGameMode<AGameModeBase>();

        if (GameMode)
        {
            GameMode->ReturnToMainMenuHost();
        }
        else
        {
            PlayerController = PlayerController == nullptr ?
World->GetFirstPlayerController() : PlayerController;

            if (PlayerController)
            {

```

```

        PlayerController->ClientReturnToMainMenuWithTextReason(FText());
    }
}
}

void UReturnToMainMenu::MenuTearDown()
{
    RemoveFromParent();
    UWorld* World = GetWorld();
    if (World)
    {
        PlayerController = PlayerController == nullptr ?
World->GetFirstPlayerController() : PlayerController;
        if (PlayerController)
        {
            FInputModeGameOnly InputModeData;
            PlayerController->SetInputMode(InputModeData);
            PlayerController->SetShowMouseCursor(false);
        }
    }
}

void UReturnToMainMenu::ReturnButtonClicked()
{
    ReturnButton->SetIsEnabled(false);
    if (MultiplayerSessionsSubsystem)
    {
        MultiplayerSessionsSubsystem->DestroySession();
    }
}

```

194 退出游戏

现在我们需要考虑的是，在哪里显示退出游戏的菜单。

首先我们添加一个输入事件，来显示菜单。

如果我们和之前游戏操作相关的输入一样将输入设置在角色中，那么当我们没有角色的时候就不能使用退出功能，所以我们应该把它放在 Controller 中。

Controller 有一个函数可以为我们提供绑定输入的位置：

```
void ABlasterPlayerController::SetupInputComponent()
{
    Super::SetupInputComponent();
    if (InputComponent == nullptr)
    {
        return;
    }
    InputComponent->BindAction("Quit", EInputEvent::IE_Pressed, this,
&ABlasterPlayerController::ShowReturnToMainMenu);
}
```

之后我们就可以在回调函数中实现我们的逻辑了：

```
void ABlasterPlayerController::ShowReturnToMainMenu()
{
    if (ReturnToMainMenuWidget == nullptr)
    {
        return;
    }
    if (ReturnToMainMenu == nullptr)
    {
        ReturnToMainMenu = CreateWidget<UReturnToMainMenu>(this,
ReturnToMainMenuWidget);
    }
    if (ReturnToMainMenu)
    {
        bReturnToMainMenuOpen = !bReturnToMainMenuOpen;
        if (bReturnToMainMenuOpen)
        {
            ReturnToMainMenu->MenuSetup();
        }
        else
        {
            ReturnToMainMenu->MenuTearDown();
        }
    }
}
```

195 退出相关的记录

我们在退出的时候，直接让人物消失的话显得有些僵硬，我们可以播放 Elim 的动画，而且我们在 Elim 中也设置了携带着的武器的掉落，这样我们就不再需要再去重复设置。

为了实现这个功能，我们在角色中添加一个自定义委托，当淘汰的效果结束之后进行广播，在 ReturnToMainMenu 中修改函数，并且添加一个回调函数：

```
void UReturnToMainMenu::ReturnButtonClicked()
{
    ReturnButton->SetIsEnabled(false);
    UWorld* World = GetWorld();
    if (World)
    {
        APlayerController* FirstPlayerController = World->GetFirstPlayerController();
        if (FirstPlayerController)
        {
            ABlasterCharacter* BlasterCharacter =
            Cast<ABlasterCharacter>(FirstPlayerController->GetPawn());
            if (BlasterCharacter)
            {
                BlasterCharacter->ServerLeaveGame();
                BlasterCharacter->OnLeftGame.AddDynamic(this,
&UReturnToMainMenu::OnPlayerLeftGame);
            }
            else
            {
                ReturnButton->SetIsEnabled(true);
            }
        }
    }
}

void UReturnToMainMenu::OnPlayerLeftGame()
{
    if (MultiplayerSessionsSubsystem)
    {
        MultiplayerSessionsSubsystem->DestroySession();
    }
}
```

在角色中，我们还需要一个服务器 RPC：

```
void ABlasterCharacter::ServerLeaveGame_Implementation()
{
    ABlasterGameMode* BlasterGameMode =
```

```

GetWorld()->GetAuthGameMode<ABlasterGameMode>();
BlasterPlayerState = BlasterPlayerState == nullptr ?
GetPlayerState<ABlasterPlayerState>() : BlasterPlayerState;
if (BlasterGameMode && BlasterPlayerState)
{
    BlasterGameMode->PlayerLeftGame(BlasterPlayerState);
}
}

```

在 GameMode 中，由于我们保存了一个第一名的成绩，所以要进行检查，检查完毕后调用 Elim()：

```

void ABusterGameMode::PlayerLeftGame(ABlasterPlayerState* PlayerLeaving)
{
    // TODO call elim, passing in true for bLeftGame
    if (PlayerLeaving == nullptr)
    {
        return;
    }

    ABlasterGameState* BlasterGameState = GetGameState<ABlasterGameState>();
    if (BlasterGameState &&
BlasterGameState->TopScoringPlayers.Contains(PlayerLeaving))
    {
        BlasterGameState->TopScoringPlayers.Remove(PlayerLeaving);
    }
    ABlasterCharacter* CharacterLeaving =
Cast<ABlasterCharacter>(PlayerLeaving->GetPawn());
    if (CharacterLeaving)
    {
        CharacterLeaving->Elim(true);
    }
}

```

Elim()，中有一段代码是请求重新生成角色的，当我们退出后，不应该再重新生成角色，所以我们将 Elim()修改为 Elim(bool bPlayerLeftGame)。这样我们就能在多播中设置角色是否需要重新生成，在 ElimTimer 结束的时候可以相应的取消掉角色的生成。

196 Gaining The Lead

用 Blender 修改一个网格体。

197 Spawning the Crown

创建两个 RPC 在 GameState 中调用，分辨开启和关闭 Crown 的效果。并且在 PollInit 和 MulticastElim 中也进行设置，以实现淘汰后效果的消失，以及重生后效果的出现。

198 淘汰通知

创建一个控件蓝图，添加一个水平框，和一个文本框用于填写淘汰信息，将其添加到 BlasterHUD 中。

发生淘汰时通过 GameMode 向每一个玩家的 Controller 发送客户端 RPC，来显示 HUD。

最后在 BlasterHUD 中添加：

```
void ABlasterHUD::AddElimAnnouncement(FString Attack, FString Victim)
{
    OwningPlayer = OwningPlayer == nullptr ? GetOwningPlayerController() :
    OwningPlayer;
    if (OwningPlayer && ElimAnnouncementClass)
    {
        auto ElimAnnouncementWidget = CreateWidget<UElimAnnouncement>(OwningPlayer,
        ElimAnnouncementClass);
        if (ElimAnnouncementWidget)
        {
            ElimAnnouncementWidget->SetElimAnnouncementText(Attack, Victim);
            ElimAnnouncementWidget->AddToViewport();
        }
    }
}
```

实现功能。

199 动态淘汰通知

创建了一个有参的 Timer 回调函数：

```
void ABlasterHUD::ElimAnnouncementTimerFinished(UElimAnnouncement* MsgToRemove)
{
    if (MsgToRemove)
    {
        MsgToRemove->RemoveFromParent();
    }
}
```

之后为了实现移动，引入三个头文件：

```
#include "Components/HorizontalBox.h"
#include "Blueprint/WidgetLayoutLibrary.h"
#include "Components/CanvasPanelSlot.h"
```

然后用一个 TArray 保存信息，通过如下代码实现信息的移动：

```
void ABlasterHUD::AddElimAnnouncement(FString Attack, FString Victim)
{
    OwningPlayer = OwningPlayer == nullptr ? GetOwningPlayerController() :
    OwningPlayer;
    if (OwningPlayer && ElimAnnouncementClass)
    {
        UElimAnnouncement* ElimAnnouncementWidget =
CreateWidget<UElimAnnouncement>(OwningPlayer, ElimAnnouncementClass);
        if (ElimAnnouncementWidget)
        {
            ElimAnnouncementWidget->SetElimAnnouncementText(Attack, Victim);
            ElimAnnouncementWidget->AddToViewport();

            for (UElimAnnouncement* Msg : ElimMessages)
            {
                if (Msg && Msg->AnnouncementBox)
                {
                    UCanvasPanelSlot* CanvasSlot =
UWidgetLayoutLibrary::SlotAsCanvasSlot(Msg->AnnouncementBox);
                    if (CanvasSlot)
                    {
                        FVector2d Position = CanvasSlot->GetPosition();
                        FVector2D NewPosition(
                            CanvasSlot->GetPosition().X,
                            Position.Y + CanvasSlot->GetSize().Y
                        );
                        CanvasSlot->SetPosition(NewPosition);
                    }
                }
            }
        }
    }
}
```

```
        }

    }

}

ElimMessages.Add(ElimAnnouncementWidget);

FTimerHandle ElimMsgTimer;
FTimerDelegate ElimMsgDelegate;
ElimMsgDelegate.BindUFunction(this,
FName("ElimAnnouncementTimerFinished"), ElimAnnouncementWidget);
GetWorldTimerManager().SetTimer(
    ElimMsgTimer,
    ElimMsgDelegate,
    ElimAnnouncementTime,
    false
);
}

}

}
```

200 爆头

对于非 SSR 的射线武器可以利用射线检测的结果 FireHit.BoneName 来判断是否击中了头部：

```
const bool bHeadShot = FireHit.BoneName.ToString() == FString("head");
```

在霰弹枪中，需要额外添加一个 HeadShotHitMap，来实现目的。

201 投掷物爆头伤害

投掷物爆头伤害和射线检测类似，直接判断 OnHit 中的目标的骨骼即可。

202 服务器倒带中的爆头

在 LagCompensationComponent 中应用相同的判断条件即可。

BUG 修复

修复了一个霰弹枪的 Bug，因为霰弹枪在开火打断 Reloading 蒙太奇后，导致函数没有调用，本地变量 bLocallyReloading 没有重置，导致无法开枪和换弹的问题。

选做项

Optional Challenge: Player Chat!

Use what you learned for our Elim Announcement system to create a player chat system! Add a text box that each player can enter messages into. Display those messages for a short period of time in the Viewport for all players.

One of your fellow students has already done this! Post your solution in the 😊 | share-your-work channel in the Druid Mechanics Discord, and compare your results!

为游戏添加聊天功能。

队伍

203 队伍

添加一个枚举类型，用来区分队伍。

在 PlayerState 中添加该枚举类型的可复制的成员变量。

最后在 GameState 中添加两个数组用来保存两队最高分的成员，之后再添加两个复制变量用来表示两队的分数。

204 组队游戏模式

创建一个新的 GameMode，在新的 GameMode 中我们需要处理三件事：

- 在开始时给每个玩家分配队伍
- 给中途加入的玩家分配队伍
- 将退出的玩家从队伍中移除

因此分别重写三个函数：

```
public:  
    virtual void PostLogin(APlayerController* NewPlayer) override;  
    virtual void Logout(AController* Exiting) override;  
  
protected:  
    virtual void HandleMatchHasStarted() override;
```

在开始时，我们从 GameState 中获取 PlayerArray，遍历数组，给每个角色分配队伍。

在中途加入和退出时，由于直接传递了 Controller，我们可以直接获取 PlayerState 从而实现队伍的设置。

205 队伍颜色

为角色新添加材质，以及溶解材质，用于不同的队伍。

206 设置队伍颜色

在角色中添加一个 SetTeamColor 函数，用来针对不同 team 上颜色。

在 PollInit 中我们获取了 PlayerState，这时候正好可以调用 SetTeamColor 获取颜色。

207 避免友方攻击

这里设置比较简单，在 PlayState 中添加了一个虚函数，用于判断是否应该造成伤害：

```
float ATeamsGameMode::CalculateDamage(AController* Attacker, AController* Victim, float BaseDamage)
{
    ABlasterPlayerState* AttackerPState =
        Attacker->GetPlayerState<ABlasterPlayerState>();
    ABlasterPlayerState* VictimPState = Victim->GetPlayerState<ABlasterPlayerState>();
    if (AttackerPState == nullptr || VictimPState == nullptr)
    {
        return BaseDamage;
    }
    if (AttackerPState == VictimPState)
    {
        return BaseDamage;
    }
    if (AttackerPState->GetTeam() == VictimPState->GetTeam())
    {
        return 0. f;
    }
    return BaseDamage;
}
```

由于 TeamsGameMode 是 BlasterGameMode 的子类，所以直接调用虚函数即可。

208 队伍分数

添加两个文本框在 CharacterOverlay 中，用来显示不同队伍的分数：

在 Controller 中添加四个函数：

```
void HideTeamScores();
void InitTeamScores();
void SetHUDBlueTeamScore(float Score);
void SetHUDRedTeamScore(float Score);
```

再添加一个复制的 bool 变量用于告诉客户端是否需要初始化分数。

在 HandleMatchStart 中设置 Init 或 Hide 即可。

209 更新队伍分数

在 PlayerState 中设置分数之后以及在复制通知函数中，调用 Controller 的设置 HUD 的函数。

之后再 TeamsGameMode 中重写淘汰函数：

```
void ATeamsGameMode::PlayerEliminated(ABlasterCharacter* ElimmedCharacter,
ABlasterPlayerController* VictimComtroller, ABlasterPlayerController*
AttackerController)
{
    Super::PlayerEliminated(ElimmedCharacter, VictimComtroller, AttackerController);

    if (VictimComtroller == AttackerController)
    {
        return;
    }

    ABlasterGameState* BGameState =
Cast<ABlasterGameState>(UGameplayStatics::GetGameState(this));
    ABlasterPlayerState* AttackerPlayerState = AttackerController ?
Cast<ABlasterPlayerState>(AttackerController->PlayerState) : nullptr;
    if (BGameState && AttackerPlayerState)
    {
        if (AttackerPlayerState->GetTeam() == ETeam::ET_BlueTeam)
        {
            BGameState->BlueTeamScores();
        }
        if (AttackerPlayerState->GetTeam() == ETeam::ET_RedTeam)
        {
            BGameState->RedTeamScores();
        }
    }
}
```

从而实现分数的更新。

210 组队模式 Cooldown Announcement

设置了一个新的函数用于显示 TeamsGameMode 下的 Announcement 内容的实现。

选做项

更多的队伍:

Optional Challenge: More Teams!

Add more teams to your game! Think about the changes you would need to make to the code. Is there a better way to make this more efficient?

Show off your work in the 😊 | share-your-work channel in the Druid Mechanics Discord!

夺旗模式

211 夺旗

选择资产添加到项目中

212 拾取旗帜

创建了一个新的武器子类用于装备旗子，创建混合空间并且将其添加到动画蓝图中。

213 拾取旗帜

添加一个插槽，用于装备武器，将 combat 中的 bHoldingFlag 设置为复制变量，在复制通知中设置客户端执行角色蹲下的操作。

214 限制旗手行为

在装备 Flag 时禁用一些行为。

215 掉落旗帜

在被淘汰之后掉落旗帜。

216 Team Flags

给武器添加一个变量，只能拾取对方阵营的旗帜。

217 Team Player Starts

由于我们的队伍信息保存在 PlayerState 中，所以我们要在 PlayerState 初始完毕之后再去遍历出生点，将角色移动到出生点上：

```
void ABlasterCharacter::PollInit()
{
    if (BlasterPlayerState == nullptr)
    {
        BlasterPlayerState = GetPlayerState<ABlasterPlayerState>();
        if (BlasterPlayerState)
        {
            OnPlayerStateInitialized();

            ABlasterGameState* BlasterGameState =
Cast<ABlasterGameState>(UGameplayStatics::GetGameState(this));
            if (BlasterGameState &&
BlasterGameState->TopScoringPlayers.Contains(BlasterPlayerState))
            {
                MulticastGainedTheLead();
            }
        }
    }
}

void ABlasterCharacter::OnPlayerStateInitialized()
{
    BlasterPlayerState->AddToScore(0. f);
    BlasterPlayerState->AddToDefeats(0);
    SetTeamColor(BlasterPlayerState->GetTeam());
    SetSpawnPoint();
}

void ABlasterCharacter::SetSpawnPoint()
{
    if (HasAuthority() && BlasterPlayerState->GetTeam() != ETeam::ET_NoTeam)
    {
        TArray<AActor*> PlayerStarts;
        UGameplayStatics::GetAllActorsOfClass(this, ATeamPlayerStart::StaticClass(),
PlayerStarts);
        TArray<ATeamPlayerStart*> TeamPlayerStarts;
        for (auto Start : PlayerStarts)
        {
            ATeamPlayerStart* TeamStart = Cast<ATeamPlayerStart>(Start);
            if (TeamStart && TeamStart->Team == BlasterPlayerState->GetTeam())
```

```
{  
    TeamPlayerStarts. Add(TeamStart);  
}  
}  
  
if (TeamPlayerStarts. Num() > 0)  
{  
    ATeamPlayerStart* ChosenPlayerStart = TeamPlayerStarts[FMath::RandRange(0,  
TeamPlayerStarts. Num() - 1)];  
    SetActorLocationAndRotation(  
        ChosenPlayerStart->GetActorLocation(),  
        ChosenPlayerStart->GetActorRotation()  
    );  
}  
}  
}
```

BUG

这里设置其实是有问题的，因为只设置了位置和旋转，但是旋转会受到控制器的影响，导致无法正确的转向出生点的位置。

218 夺旗游戏模式

创建一个 FlagZone， 用于触发事件。

继承与子 TeamsGameMode， 两个函数：

```
void ACaptureTheFlagGameMode::PlayerEliminated(ABlasterCharacter* ElimmedCharacter,
ABlasterPlayerController* VictimComtroller, ABlasterPlayerController*
AttackerController)
{
    ABlasterGameMode::PlayerEliminated(ElimmedCharacter, VictimComtroller,
AttackerController);
}

void ACaptureTheFlagGameMode::FlagCaptured(AFlag* Flag, AFlagZone* Zone)
{
    bool bVaildCaptuer = Flag->GetTeam() != Zone->Team;
    ABlasterGameState* BGameState = Cast<ABlasterGameState>(GameState);
    if (BGameState)
    {
        if (Zone->Team == ETeam::ET_BlueTeam)
        {
            BGameState->BlueTeamScores();
        }
        if (Zone->Team == ETeam::ET_RedTeam)
        {
            BGameState->RedTeamScores();
        }
    }
}
```

在 Flag 中由于需要重置 flag， 设置了一个重置函数：

```
void AFlag::ResetFlag()
{
    ABlasterCharacter* FlagBearer = Cast<ABlasterCharacter>(GetOwner());
    if (FlagBearer)
    {
        FlagBearer->SetHoldingTheFlag(false);
        FlagBearer->SetOverlappingWeapon(nullptr);
        FlagBearer->UnCrouch();
    }

    if (!HasAuthority())
    {
        return;
    }
```

```

FDetachmentTransformRules DetachRules(EDetachmentRule::KeepWorld, true);
FlagMesh->DetachFromComponent(DetachRules);
SetWeaponState(EWeaponState::EWS_Initial);
GetAreaSphere()->SetCollisionEnabled(ECollisionEnabled::QueryAndPhysics);
GetAreaSphere()->SetCollisionResponseToChannel(ECollisionChannel::ECC_Pawn,
ECollisionResponse::ECR_Overlap);

SetOwner(nullptr);
BlasterOwnerCharacter = nullptr;
BlasterOwnerController = nullptr;

SetActorTransform(InitialTransform);
}

```

在武器传送回原位置后，我们需要给人物传递一个 nullptr 的 overlap 武器，不然可以再次按 E 将武器隔空取回。

此外角色在武器重置后需要站起来，这个操作需要在客户端上进行 Uncrouch()，所以 overlap 的事件不能仅在服务端注册需要在所有地方注册。

BUG 修复

之前的武器的构造函数中添加了一行代码，会导致武器出现问题，之前的武器瞄准和武器初始位置的问题应该也是由这行代码导致的：

```

AWeapon::AWeapon()
{
    PrimaryActorTick.bCanEverTick = false;
    bReplicates = true;
    SetReplicateMovement(true);

    WeaponMesh = CreateDefaultSubobject<USkeletalMeshComponent>(TEXT("WeaponMesh"));
    WeaponMesh->SetupAttachment(RootComponent);
    SetRootComponent(WeaponMesh);
    ...
}

```

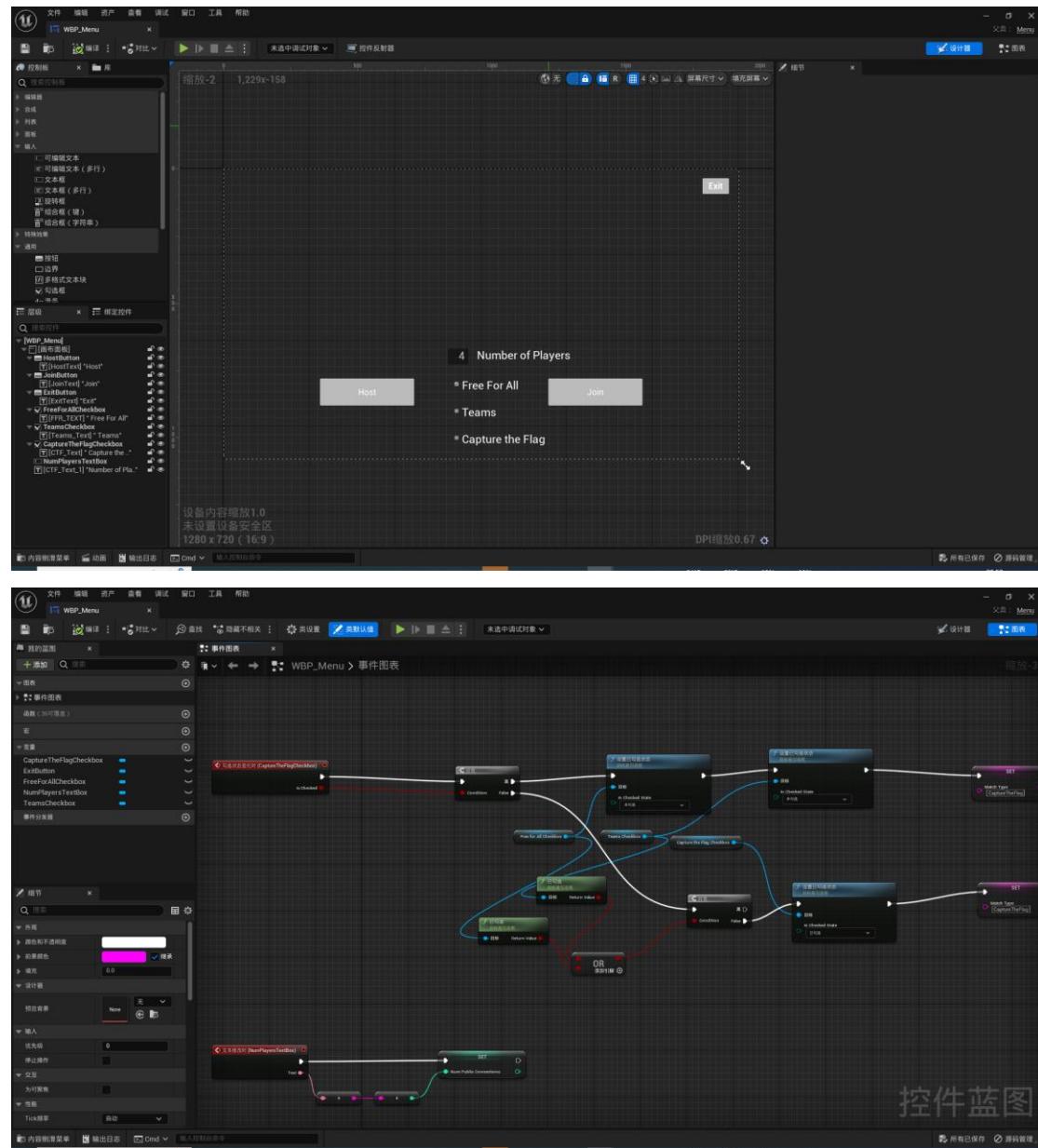
219 选择匹配模式

通过这个宏标记可以在蓝图中编辑变量：

```
UPROPERTY(BlueprintReadWrite, meta = (AllowPrivateAccess = "true"))
int32 NumPublicConnections{4};
```

```
UPROPERTY(BlueprintReadWrite, meta = (AllowPrivateAccess = "true"))
FString MatchType{ TEXT("FreeForAll") };
```

这样，我们可以设置我们在插件 menu 中一些变量，来选择不同的游戏模式和不同的玩家数量：



220 连接我们的子系统

从 LobbyGameMode 中连接子系统，获取变量，来选择我们要进入的地图：

```
void ALobbyGameMode::PostLogin(APlayerController* NewPlayer)
{
    Super::PostLogin(NewPlayer);

    int32 NumberOfPlayers = GameState.Get()>PlayerArray.Num();

    UGameInstance* GameInstance = GetGameInstance();
    if (GameInstance)
    {
        UMultiplayerSessionsSubsystem* SubSystem =
        GameInstance->GetSubsystem<UMultiplayerSessionsSubsystem>();
        check(SubSystem);
        if (NumberOfPlayers == SubSystem->DesiredNumPublicConnections)
        {
            UWorld* World = GetWorld();
            if (World)
            {
                bUseSeamlessTravel = true;

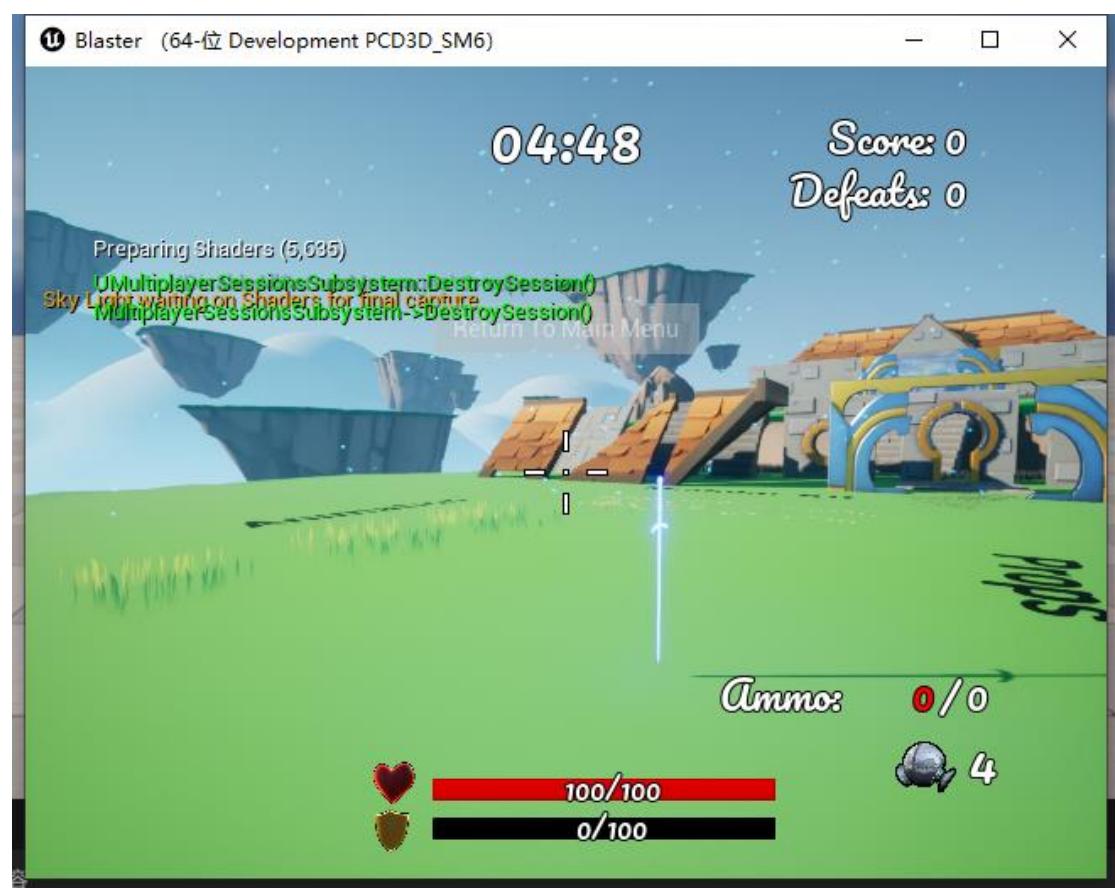
                FString MatchType = SubSystem->DesiredMatchType;
                if (MatchType == "FreeForAll")
                {
                    World->ServerTravel(FString("/Game/Maps/BlasterMap?listen"));
                }
                else if (MatchType == "Teams")
                {
                    World->ServerTravel(FString("/Game/Maps/TeamsMap?listen"));
                }
                else if (MatchType == "CaptureTheFlag")
                {
                    World->ServerTravel(FString("/Game/Maps/CaptureTheFlagMap?listen"));
                }
            }
        }
    }
}
```

221 制作地图

现在游戏所需的功能都已经制作完成了，制作三个模式下的地图，打包游戏~这个教程学了将近一个月，算是初步入门 UE 了。

一些其它问题

MultiplayerSessionsSubsystem 中的代码错误



这个是由于我直接写插件的时候，插件中的错误导致的，少了两个 ‘!’ 符号导致的。

Sequence 无法减少问题

之前在其他角色上射击会导致客户端上 sequence 无法减少，这是因为 clientRPC 导致的。
官网上写的 RPC 的规则如下：

从服务器调用的 RPC

Actor 所有权	未复制	NetMulticast	Server	Client
Client-owned actor	在服务器上运行	在服务器和所有客户端上运行	在服务器上运行	在 actor 的所属客户端上运行
Server-owned actor	在服务器上运行	在服务器和所有客户端上运行	在服务器上运行	在服务器上运行
Unowned actor	在服务器上运行	在服务器和所有客户端上运行	在服务器上运行	在服务器上运行

什么事客户端拥有的 actor 什么是服务端拥有的 actor，有点意味不明。

开始的时候我觉得是每个客户端上都有一个 Weapon 的 Actor, 所以在服务器上跑 ClientRPC 和多播除了服务器之外是一个效果？但是实际使用的时候发现只有当角色持有武器的时候，才能接受到 RPC, 所以其他角色持枪射击的时候收不到 RPC 从而导致 Sequence 无法减少出现 Bug。

因为这个变量只是为了实现本地开火时预测服务器子弹而使用的，所以可以将其只在本地执行：

```
void AWeapon::SpendRound()
{
    Ammo = FMath::Clamp(Ammo - 1, 0, MagCapacity);
    SetHUDAmmo();
    if (HasAuthority())
    {
        ClientUpdateAmmo(Ammo);
    }
    else
    {
        BlasterOwnerCharacter = BlasterOwnerCharacter == nullptr ?
            Cast<ABlasterCharacter>(GetOwner()) : BlasterOwnerCharacter;
        if (BlasterOwnerCharacter->IsLocallyControlled())
        {
            Sequence++;
        }
    }
    UE_LOG(LogTemp, Warning, TEXT("Sequence : %d"), Sequence);
}
```

之后还发现一个问题，就是其他角色换完子弹之后，在客户端拾取起武器时武器的子弹是没有更新的，这会导致捡起其他角色刚上完子弹的武器时，显示的 Ammo 是 0，从而无法开火，换弹也会因为服务器中子弹是满的而无法换弹。

后来发现也是 ClientRPC 的原因：

在换弹之后我们调用了 AddAmmo 函数，函数在调用一个 ClientAmmo 来更新子弹，但是由于此时 Client 没有 Owner，所以只在服务器上运行了函数，客户端没有得到更新，为此将 ClientRPC 替换为多播 RPC 才能正确实现功能：

```
UFUNCTION(Client, Reliable)
void ClientAddAmmo(int32 AmmoToAdd);

UFUNCTION(NetMulticast, Reliable)
void ClientAddAmmo(int32 AmmoToAdd);
```

由此可以猜测 Client-Owner Actor, Server-Owner Actor 以及 Unowned Actor, 这里的 Owner 应该是要实际的 Controller 控制的角色作为 Owner 的物体才可以收到 ClientRPC 的消息。

受击蒙太奇中断换弹蒙太奇



现在的动画蓝图是这个结构，受击和换弹的插槽直接串联，播放一个一定会打断另一个。

可以考虑将受击动画做成一个混合空间，然后用一个 bool 变量和一个 FVector 来控制实现来自不同方向的受击动画，但是这里由于受击动画不会影响游戏逻辑，直接在换弹过程中禁用受击动画。

```
void ABlasterCharacter::ReceiveDamage(AActor* DamagedActor, float Damage, const UDamageType* DamageType, AController* InstigatorController, AActor* DamageCauser)
{
    ...
    if (Damage > 0. f)
    {
        if (Combat && Combat->CombatState == ECombatState::ECS_Unoccupied)
        {
            PlayHitReactMontage();
        }
    }
    ...
}
```

原地旋转

最后是第 80 节提到的原地旋转，我不是很喜欢作者的修改，所以我又改了回去。

子弹击中自己

一个最简单的办法是吧开火的 Socket 挪远一些。

ProjectileBullet 可以在武器中添加：

```
CollisionBox->MoveIgnoreActors. Add(Cast<ABlasterCharacter>(GetOwner()));
```

射线武器，我们点进 LineTracing 的定义可以看到，在最后还有一个默认参数，我们可以选择他，然后添加忽略我们的角色：

```
FCollisionQueryParams TraceParams;  
TraceParams. AddIgnoredActor(Cast<ABlasterCharacter>(GetOwner()));  
bool bHit = World->LineTraceSingleByChannel(  
    OutHit,  
    SocketLocation,  
    End,  
    ECollisionChannel::ECC_Visibility,  
    TraceParams  
) ;
```

在服务器倒带中是不需要再去进行操作的，因为一开始的射线检测不会命中我们，所以服务器不会对武器持有者进行倒带。最后的射线检测通道是特别的通道，由于武器持有者的 HitBox 不是开启目标没有开启碰撞，也会直接忽略掉武器持有者。

出生后方向问题

还有一个问题是出生之后的旋转问题，因为出生的时候是在出生点中随意选取一个点出生，然后设置到 team 的点位上，这会导致出生点的位置虽然没问题，但是旋转却不一样的情况。

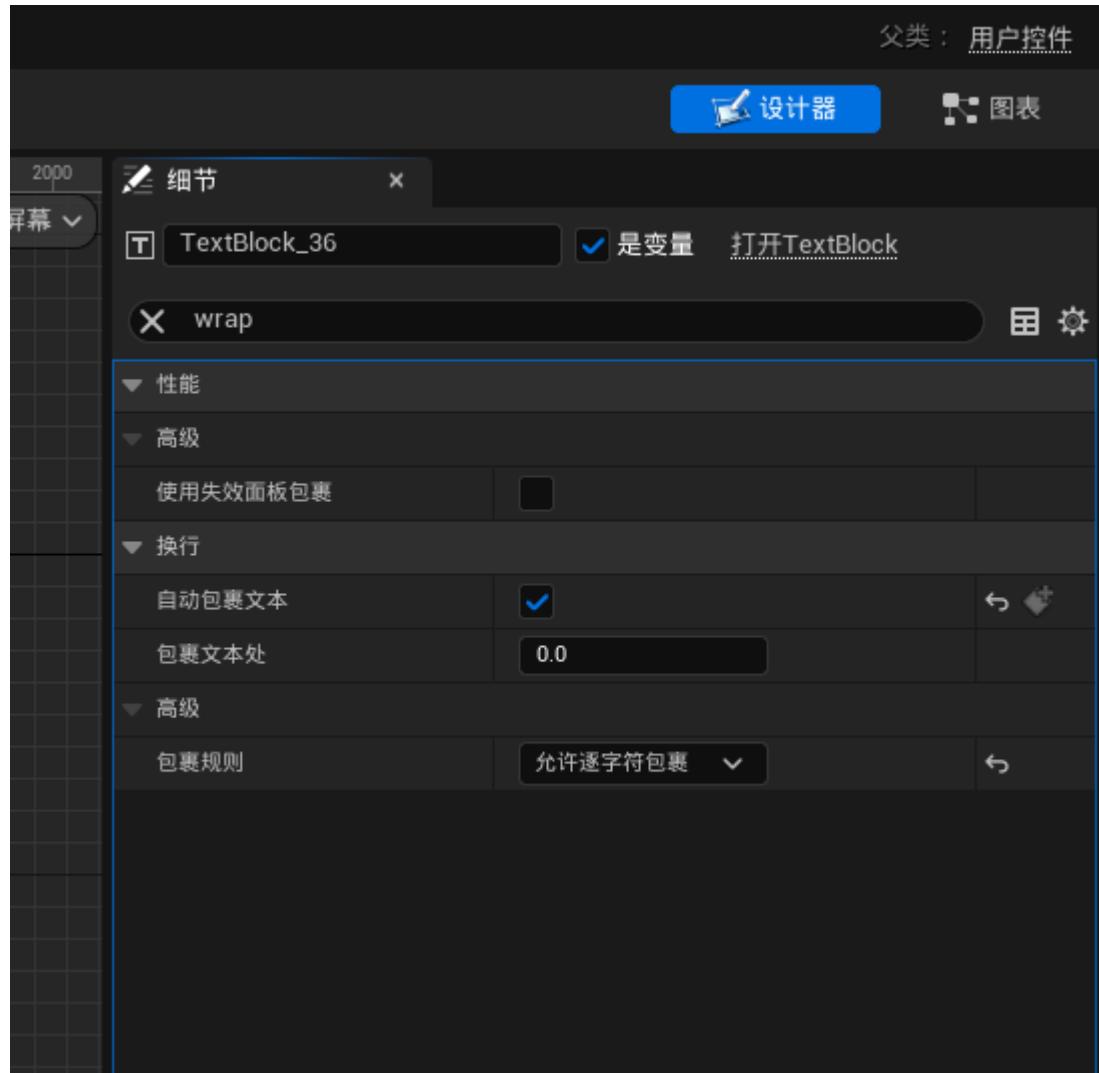
搞了半天，没注意到这个移动是在服务器上进行的，因此，为了在客户端能够实现旋转，我们需要旋转客户端上的控制器，使用一个 ClientRPC 即可，传入出生点：

```
void ABlasterCharacter::ClientSetSpawnRotation_Implementation(AActor* SpawnPoint)
{
    if (Controller)
    {
        Controller->SetControlRotation(SpawnPoint->GetActorRotation());
    }
}
```

聊天框

最后我想给项目加一个聊天框：

研究了很久，终于知道怎么让文本框换行了：勾选自动包裹文本以及包裹规则。



聊天框的实现可以使用 Listview 控件：

在实现多播的时候，简单的想法是直接使用 Controller 一个 serverRPC 调用到服务器，再从服务器调用多播 RPC 实现，但是实际操作之后并不能如愿的实现功能，需要循环所有的 Controller 然后单独发送 ClientRPC，因为每个客户端只有一个 Controller。

使用 ListView 要使用 3 个类：

详细介绍可以看：<https://zhuanlan.zhihu.com/p/127184008>

文章中是使用蓝图实现的，我在实现蓝图之后，用 C++ 重新写了一遍，但是 Entry 类还是保留了使用蓝图实现，因为接口的文档不是很好找，而且今天也比较晚了，以后有空再弄吧。

首先是 ChatWidget 本身：

```
void UChatWidget::NativeConstruct()
{
    Super::NativeConstruct();
    if (ChatInputBox)
    {
        ChatInputBox->OnTextCommitted.AddDynamic(this, &UChatWidget::SendMessage);
    }
}

void UChatWidget::ReceiveMessage(FText Message)
{
    if (ChatListView)
    {
        UIItemData* Item = NewObject<UIItemData>();
        Item->Content = Message;
        ChatListView->AddItem(Item);
        ChatListView->ScrollToBottom();
    }
}

void UChatWidget::SendMessage(const FText& Text, ETextCommit::Type CommitMethod)
{
    Controller = Controller == nullptr ?
        Cast<ABlasterPlayerController>(GetWorld()->GetFirstPlayerController()) : Controller;
    if (Controller == nullptr) return;
    Controller->SendMessage(Text);
    Controller->SetInputMode(FInputModeGameOnly());
    if (!Text.IsEmpty())
    {
        ChatInputBox->SetText(FText());
    }
}
```

其次是所使用的数据类：

```
UCLASS(Blueprintable, BlueprintType)
class BLASTER_API UIItemData : public UObject
{
    GENERATED_BODY()

public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere, Category = "default", meta =
    (ExposeOnSpawn = "true"))
    FText Content;
```

```
UPROPERTY(BlueprintReadWrite, EditAnywhere, Category = "default", meta =  
(ExposeOnSpawn = "true"))  
bool bSelected;  
};
```

瞄准镜

狙击枪开镜之后，会遮挡住 UI，在角色蓝图中设置其优先级为-1 即可解决：

