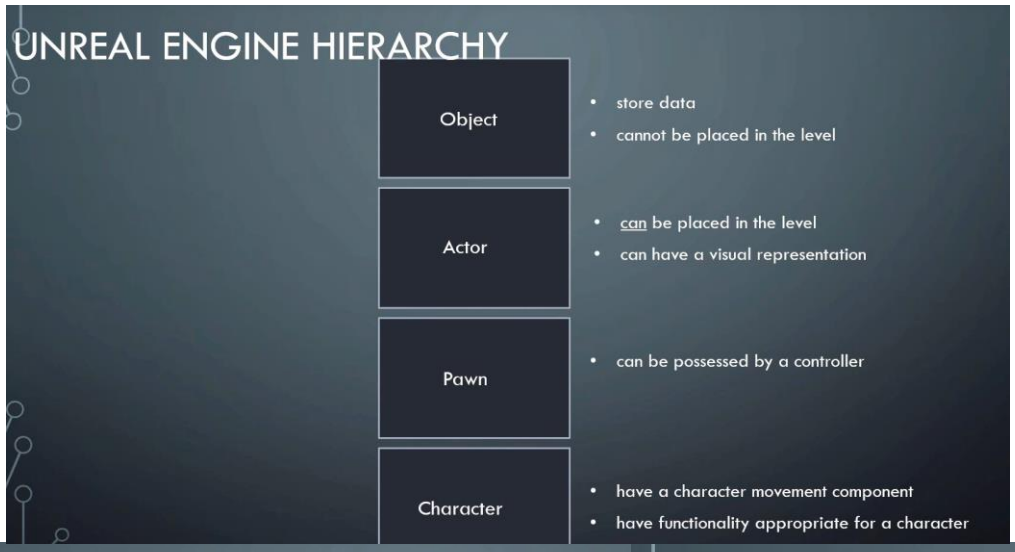
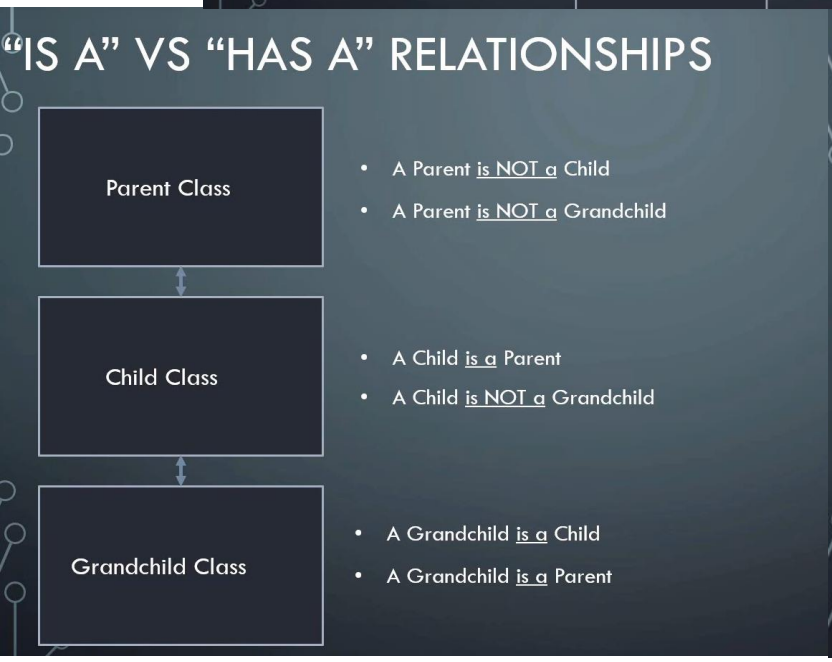
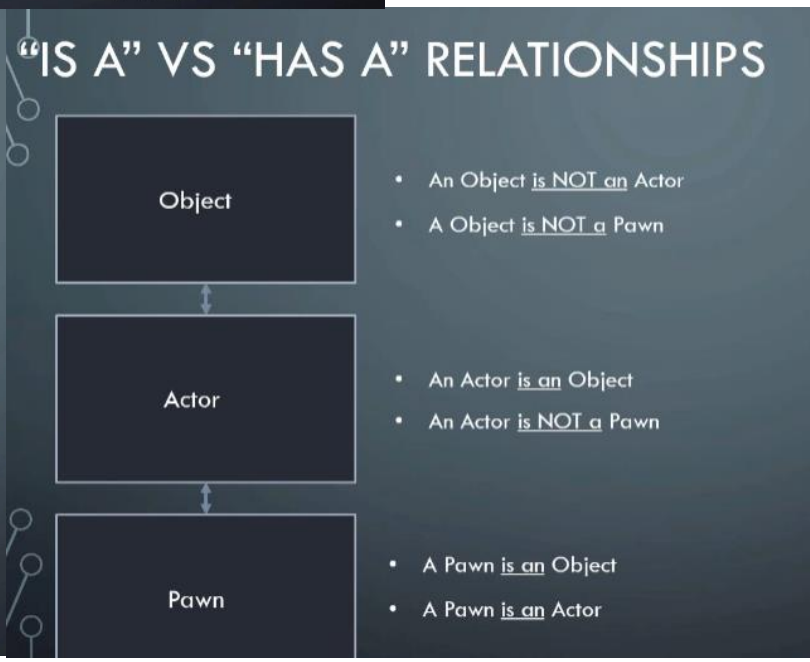
```
PickupWidget->SetVisibility(false);
```
C++ fundament







Int a = 10, b=10;

Int *pointer = &a; pointer 有自己的 address，reference 没有自己的 address

Int &ref = a;

*pointer = b; pointer 指向的 address 的值会变成 b ref = b; ref reference 的 a 的值会变 b

pointer = &b; pointer 指向 b，a 值不变 cout<<pointer<<*pointer;

whatever is inside the address of the pointer will become b

address value

Macro

- Fragment of code which has been given a name
- Whenever the name of macro is used, it is replaced by the content of that macro

UCLASS()

- Part of Unreal Reflection System
- Makes engine aware of a class below

Virtual

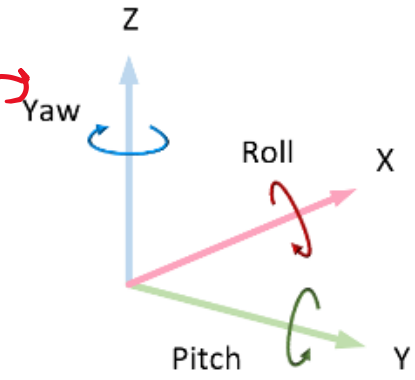
- The function/method can be rewrite in child classes

Tick

- Is called every single frame

Roll Pitch Yaw(X Y Z)

- Pitch 是围绕 x 轴旋转，也叫做俯仰角
- Yaw 是围绕 y 轴旋转，也叫做偏航角
- Roll 是围绕 z 轴旋转，也叫做翻滚角
- 以上是右手坐标系而 UE4 是左手坐标系
- 所以 x→y, y→z, z→y
- Pitch Yaw Roll (Frotator)



Delete class

13. Deleting Classes.mp4

UE_log (similar to console log)

```
UE_LOG(LogTemp, Warning, TEXT("BeginPlay() called!"));

int myInt{ 42 };
UE_LOG(LogTemp, Warning, TEXT("int myInt: %d"), myInt);

float myFloat{ 3.14159f };
UE_LOG(LogTemp, Warning, TEXT("float myFloat: %f"), myFloat);

double myDouble{ 0.000235 };
UE_LOG(LogTemp, Warning, TEXT("double myDouble: %lf"), myDouble);

char myChar{ 'j' };
UE_LOG(LogTemp, Warning, TEXT("char myChar: %c"), myChar);

wchar_t wideChar{ L'j' };
UE_LOG(LogTemp, Warning, TEXT("wchar_t wideChar: %lc"), wideChar);

bool myBool{ true };
UE_LOG(LogTemp, Warning, TEXT("bool myBool: %d"), myBool);
```

Content Browser | Output Log

Filters | Search Log

```
LogInit: FAudioDevice initialized.
LogAudio: Display: Audio Device (ID: 10) registered with world 'DefaultMap'.
LogLoad: Game class is 'ShooterGameModeBaseBP_C'
LogWorld: Bringing World /Game/_Game/UEDPIE_0_DefaultMap.DefaultMap up for play (max tick rate 0) at 2021.10.15-19.48
LogWorld: Bringing up level for play took: 0.000441
LogOnline: OSS: Creating online subsystem instance for: :Context_13
LogTemp: Warning: BeginPlay() called!
LogTemp: Warning: int myInt: 42
LogTemp: Warning: float myFloat: 3.141590
LogTemp: Warning: double myDouble: 0.000235
LogTemp: Warning: char myChar: j
LogTemp: Warning: wchar_t wideChar: j
LogTemp: Warning: bool myBool: 1
PIE: Server logged in
PIE: Play in editor total start time 0.101 seconds.
```

Cmd | Enter Console Command

```
FString myString{ TEXT("My String!!!") };
UE_LOG(LogTemp, Warning, TEXT("FString myString: %s"), *myString);
```

这个build in function 可以获得这个object的名字

这是个object 需要*才能换成char array

```
LogTemp: Warning: Name of instance: MyShooterCharacterBP_C_0
UE_LOG(LogTemp, Warning, TEXT("Name of instance: %s"), *GetName());
```

Spring Arm Component

ShooterCharacter.h*

```
private:

    class USpringArmComponent* CameraBoom;

public:

    FORCEINLINE USpringArmComponent* GetCameraBoom() const { return CameraBoom; }
```

因为这个function很短，为了optimize compile，所有的这个function都会被replace成这个

Forward Declaration: A forward declaration **tells the compiler about the existence of an entity before actually defining the entity**.

Getters & Setters: -They are used to access and set the values of private members of classes respectively.

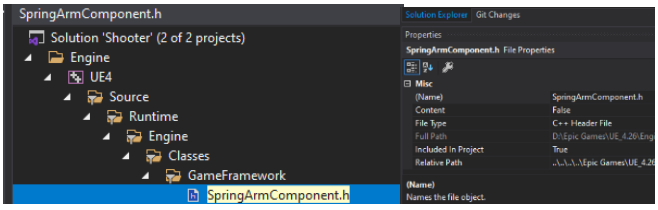
+ *Getter* is commonly known as accessor function. Getter is a special function that allows us to access the data members of the class(even private and protected members).

+ *Setter* is commonly known as the mutator function. It is a special function that allows us to set(assign) values to a data member of an object of a class. It can also change the previous value assigned to a data member. However, the data members of an object of the class are usually assigned values by creating a special function called the constructor. Usually, setters are used along with conditions which are checked before assigning the value of data member.

简单来说getter function能让你access某个class里面的data(even private and protected members)，setter是能让你更改data

ShooterCharacter.cpp*

```
AShooterCharacter::AShooterCharacter()
{
    // Set this character to call Tick() every frame.  You can turn this off to improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;

    CameraBoom = CreateDefaultSubobject<USpringArmComponent>(TEXT("CameraBoom"));
    CameraBoom->SetupAttachment(RootComponent);
}
```



D:\Epic Games\UE_4.26\Engine\Source\Runtime\Engine\Classes\GameFramework\SpringArmComponent.h

#include “GameFramework/SpringArmComponent.h”

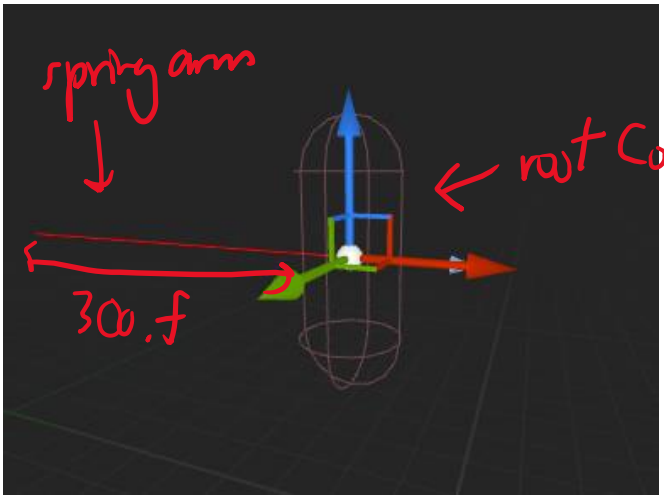
也可以去docs.unrealengine.com 找path

#Final code(camera boom)

```
private:
    /* Camera boom positioning the camera behind the character */
    UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = Camera, meta = (AllowPrivateAccess = "true"))
    class USpringArmComponent* CameraBoom;

public:
    /* Returns CameraBoom subobject */
    FORCEINLINE USpringArmComponent* GetCameraBoom() const { return CameraBoom; }

};
```

```
// Create a camera boom (pulls in towards the character if there is a collision)
CameraBoom = CreateDefaultSubobject<USpringArmComponent>(TEXT("CameraBoom"));
CameraBoom->SetupAttachment(RootComponent);
CameraBoom->TargetArmLength = 300.f; // The camera follow at this distance behind the character
CameraBoom->bUsePawnControlRotation = true; //Rotate the arm based on the controller
```

#Final result(camera boom)



Follow Camera

```
/* Camera that follows the character */
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = Camera, meta = (AllowPrivateAccess = "true"))
class UCameraComponent* FollowCamera;
/* Returns FollowCamera subobject */
FORCEINLINE UCameraComponent* GetFollowCamera() const { return FollowCamera;  }
```

```
#include "Camera/CameraComponent.h"
```

```
// Create a follow camera
FollowCamera = CreateAbstractDefaultSubobject<UCameraComponent>(TEXT("FollowCamera"));
FollowCamera->SetupAttachment()
```

ctrl+shift+spacebar

```
void SetupAttachment(USceneComponent *InParent, FName InSocketName = NAME_None)
Initializes desired Attach Parent and SocketName to be attached to when the component is registered.Generally intended to be called from its Owning Actor's constructor and should be preferred over AttachToComponent whena component is not registered.
InParent: Parent to attach to.
```

```
FollowCamera->SetupAttachment(CameraBoom, USpringArmComponent::SocketName)
```

```
static const FName USpringArmComponent::SocketName
* The name of the socket at the end of the spring arm (looking back towards the spring arm origin)
Search Online
```

USpringArmComponent::SocketName
是springarm的尾端的socket的名字

```
FollowCamera->bUsePawnControlRotation = false;
```

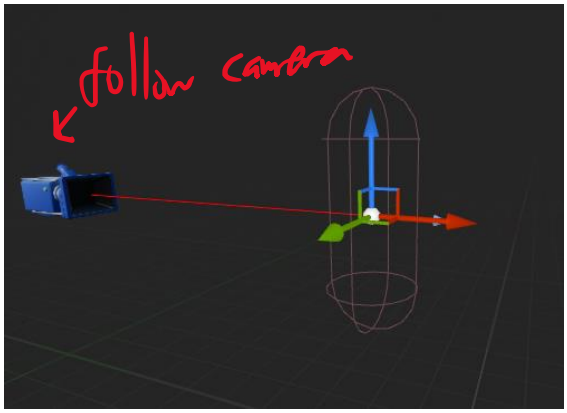
不要camera跟着pawn转动，只要他attach在springarm尾端

#Final code(follow camera)

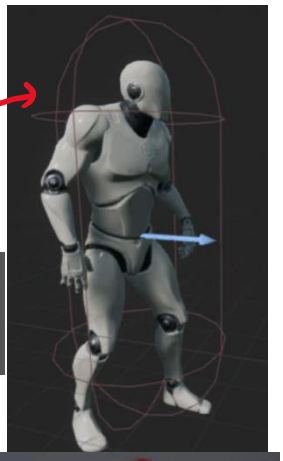
```
private:
    /* Camera boom positioning the camera behind the character */
    UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = Camera, meta = (AllowPrivateAccess = "true"))
    class USpringArmComponent* CameraBoom;

    /* Camera that follows the character */
    UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = Camera, meta = (AllowPrivateAccess = "true"))
    class UCameraComponent* FollowCamera;

public:
    /* Returns CameraBoom subobject */
    FORCEINLINE USpringArmComponent* GetCameraBoom() const { return CameraBoom; }

    /* Returns FollowCamera subobject */
    FORCEINLINE UCameraComponent* GetFollowCamera() const { return FollowCamera;  }
};
```

```
// Create a follow camera
FollowCamera = CreateAbstractDefaultSubobject<UCameraComponent>(TEXT("FollowCamera"));
FollowCamera->SetupAttachment(CameraBoom, USpringArmComponent::SocketName); //Attach camera to end of boom
FollowCamera->bUsePawnControlRotation = false; //Camera does not rotate relative to arm
```

#Final result(follow camera)



Controller and input



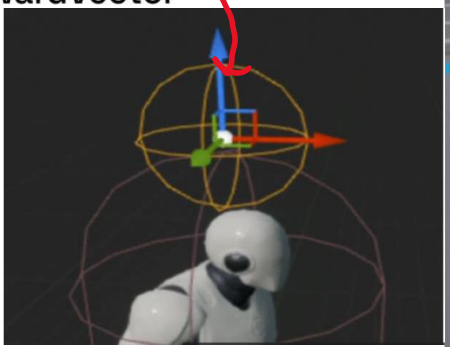
Capsule

It's the capsule that actually collides with objects in the world, because if most objects collided, 记穿量太大

The Controller Class

The Controller is Also Invisible

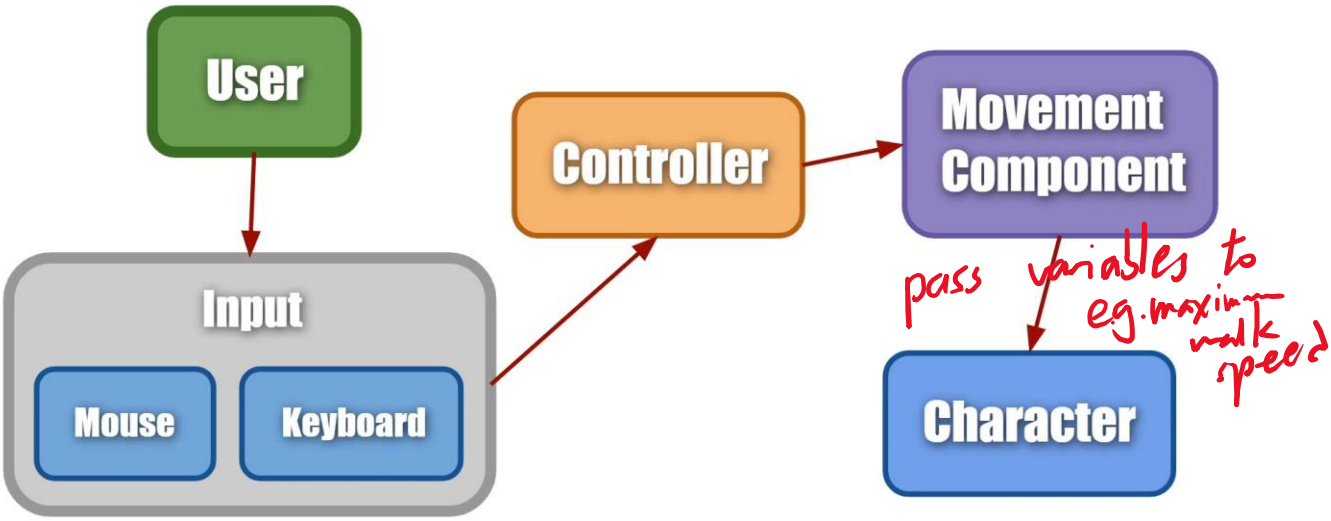
- An invisible entity
- It has an orientation (rotation)
- There is no GetControllerForwardVector
- But we can calculate it!



-Controller只是一个object然后associate with the character
-有function可以get或set controller 的rotation

Possession

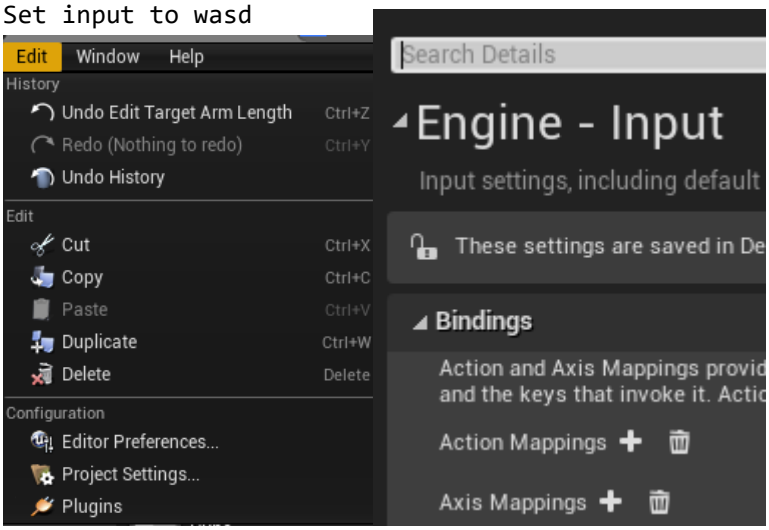
When a controller possesses a character, it controls how that character moves.



-Movement component takes input data and performs calculations to makes sure that before the character is moved, it moves in the right way, it obeys law such as gravity, cannot move through walls or other objects

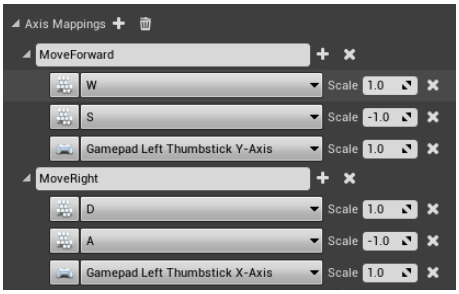
Move Froward and Right

Set input to wasd



Action Mappings:
You pressed a key and the function is called single time

Axis Mappings:
Continuous receive data about the key pressed every frame

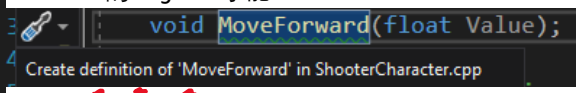


因为前后的movement的logic都是一样唯一差的是input的data，向前为+则向后为-，向右为+则向左为-

Writing function for movement

```
protected:
    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

    /* Called for forwards/backwards input */
    void MoveForward(float Value);

    /* Called for side to side input */
    void MoveRight(float Value);
```

在protected的地方因为这个function只有这个class的object或者inherited这个class的object才能call



快速create definition

```
id AShooterCharacter::MoveForward(float Value)

 if ((Controller != nullptr) && (Value != 0.0f))
 {
    //find out which way is forward
    const FRotator Rotation{ Controller->GetControlRotation() };
    const FRotator YawRotation{ 0,Rotation.Yaw,0 };

 }
const FVector Direction{ FRotationMatrix{YawRotation}.GetUnitAxis(EAxis::X) };
AddMovementInput(Direction, Value);
```

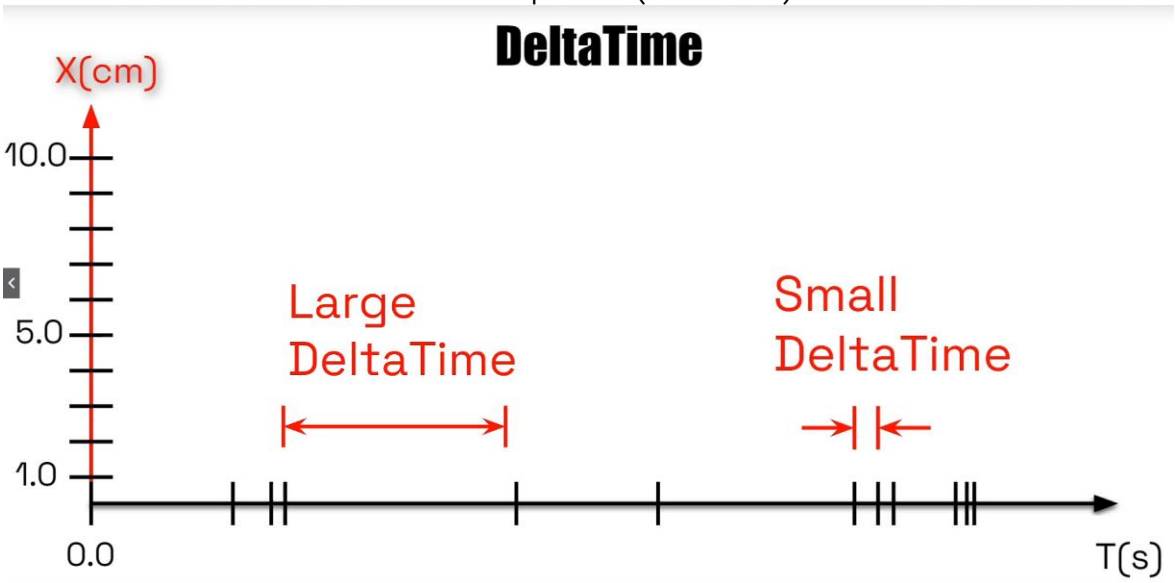
pitch Yaw roll 只要 Yaw，其他放0.

```
void AShooterCharacter::MoveForward(float Value)
{
    if ((Controller != nullptr) && (Value != 0.0f))
    {
        //find out which way is forward
        const FRotator Rotation{ Controller->GetControlRotation() };
        const FRotator YawRotation{ 0,Rotation.Yaw,0 };

        const FVector Direction{ FRotationMatrix{YawRotation}.GetUnitAxis(EAxis::X) };
        AddMovementInput(Direction, Value);
    }
}

void AShooterCharacter::MoveRight(float Value)
{
    if ((Controller != nullptr) && (Value != 0.0f))
    {
```

Binding the key to the function

DeltaTime

DeltaTime: The time between Frames
Frame: A single image updated to the screen
Frame Rate: Number of frames updated per second(FPS)
Tick: Synonymous with Frame
**Frame times are not constant and sporadic(scattered)




```
void Tick(float DeltaTime)
{
  Move(1.f);    WRONG!
}
```

```
void Tick(float DeltaTime)
{
  Move(1.f * DeltaTime);
}
```

$$1.0 \frac{cm}{\cancel{sec}} \times DT \frac{\cancel{sec}}{frame} = \frac{cm}{frame}$$

假设1s要移动1cm
因为deltatime不是constant，所以每个frame之间的时间可能不一样
2second per frame:
每个frame就要走1*2=2cm

1second per frame:
每个frame就要走1*1=1cm，两个frame也就是两秒后会走2cm也就和2second per frame一样

Turn At Rate

```
/* Base turn rate, in degree/sec. Other scaling may affect final turn rate*/
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = Camera, meta = (AllowPrivateAccess = "true"))
float BaseTurnRate;

/* Base look up/down rate, in degree/sec. Other scaling may affect final turn rate*/
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = Camera, meta = (AllowPrivateAccess = "true"))
float BaseLookUpRate;

/*
Called via input to turn at a given rate.
@param Rate This is a normalized rate,i.e. 1.0 means 100% of desired turn rate
*/
void TurnAtRate(float Rate);

/*
Called via input to look up/down at a given rate.
@param Rate This is a normalized rate,i.e. 1.0 means 100% of desired turn rate
*/
void LookUpAtRate(float Rate);
```

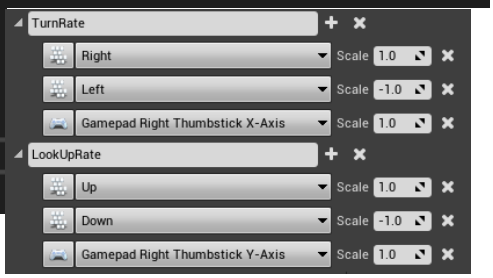
← from 0 ~ 1 means 0% to 100% of the base rate

```
void AShooterCharacter::TurnAtRate(float Rate)
{
    // calculate delta for this frame from the rate information
    AddControllerYawInput(Rate * BaseTurnRate * GetWorld()->GetDeltaSeconds()); //degree/second *second/frame = degree/frame

}
```

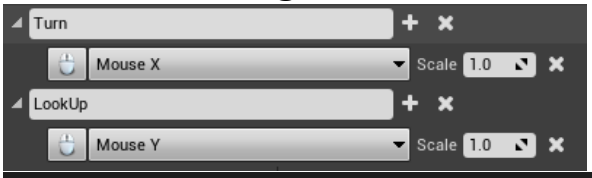
Delta Time

```
void AShooterCharacter::LookUpAtRate(float Rate)
{
    AddControllerPitchInput(Rate * BaseLookUpRate * GetWorld()->GetDeltaSeconds()); //degree/second *second/frame = degree/frame
}
```

```
// Sets default values
AShooterCharacter::AShooterCharacter():
    BaseTurnRate(45.f),
    BaseLookUpRate(45.f)
{
```



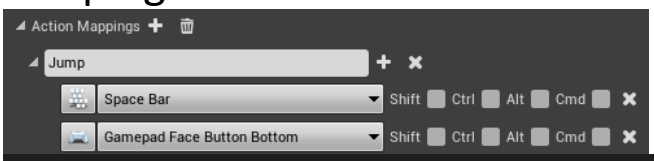
Mouse Turning



Build in function

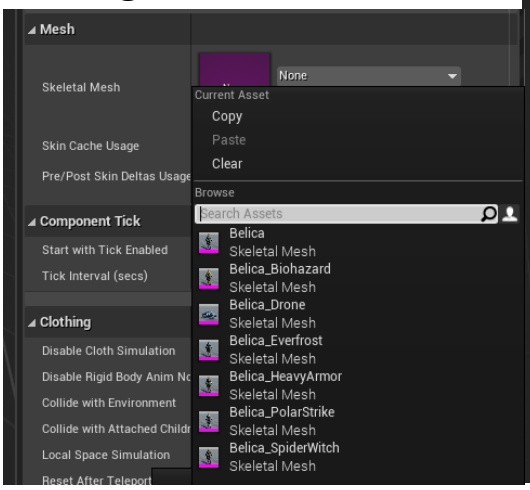
```
PlayerInputComponent->BindAxis("Turn", this, &APawn::AddControllerYawInput);
PlayerInputComponent->BindAxis("LookUp", this, &APawn::AddControllerPitchInput);
```

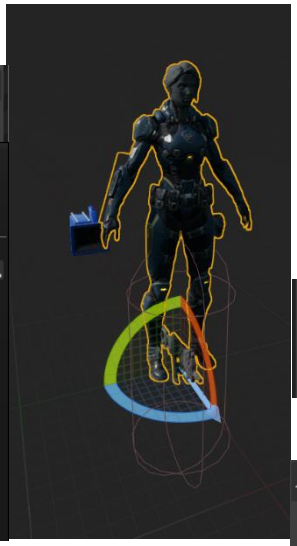
Jumping




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

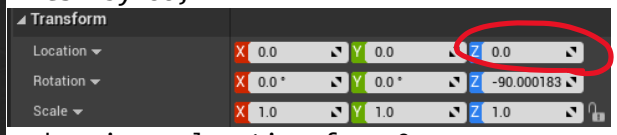
Adding a Mesh





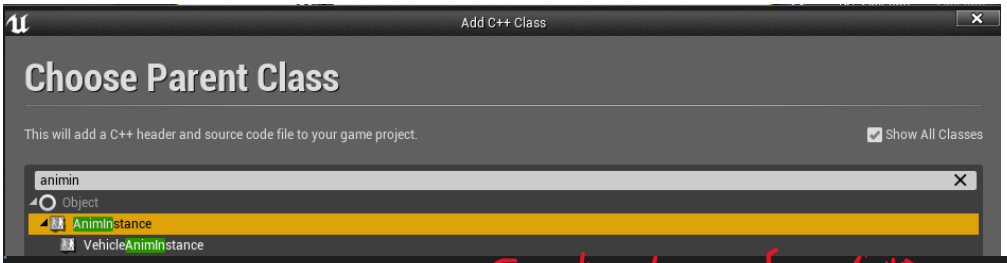
Press E to rotate

As capsule half height is 88, we can drop down the mesh by 88,



changing z location from 0 to -88

The Animation Instance




```
UFUNCTION(BlueprintCallable)
void UpdateAnimationProperties(float DeltaTime);

virtual void NativeInitializeAnimation() override;

private:
    UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = Movement, meta = (AllowPrivateAccess = "true"))
    class AShooterCharacter* ShooterCharacter;

    /* The speed of the character */
    UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = Movement, meta = (AllowPrivateAccess = "true"))
    float Speed;

    /* Wheather or not the character is in the air*/
    UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = Movement, meta = (AllowPrivateAccess = "true"))
    bool bIsInAir;

    /* Wheather or not the character is moving */
    UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = Movement, meta = (AllowPrivateAccess = "true"))
    bool bIsInAccelerating;
```

Create two functions update animation

can call the function on BP

sort of BeginPlay, constructor as animinstance dont have those

```
void UShooterAnimInstance::NativeInitializeAnimation()
{
    ShooterCharacter = Cast<AShooterCharacter>(TryGetPawnOwner());
}
```

This return a reference to the Pawn class that owns the animation instance im running, and although it is a pawn, we need to cast it to AShooterCharacter

```
void UShooterAnimInstance::UpdateAnimationProperties(float DeltaTime)
{
    if (ShooterCharacter == nullptr)
    {
        ShooterCharacter = Cast<AShooterCharacter>(TryGetPawnOwner());
    }
    if (ShooterCharacter)
    {
        // Get the lateral speed of the character from velocity
        FVector Velocity{ ShooterCharacter->GetVelocity() };
        Velocity.Z = 0;
        Speed = Velocity.Size();

        // Is the character in the air?
        bIsInAir = ShooterCharacter->GetCharacterMovement()->IsFalling();

        // Is the character accelerating?
        if (ShooterCharacter->GetCharacterMovement()->GetCurrentAcceleration().Size() > 0.f)
        {
            bIsAccelerating = true;
        }
        else
        {
            bIsAccelerating = false;
        }
    }
}
```

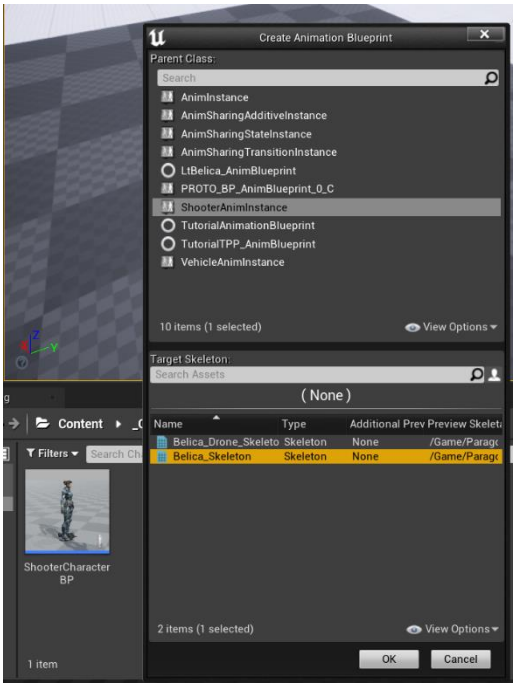
check if ShooterCharacter is null

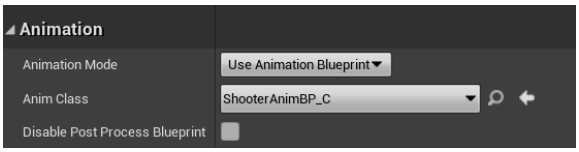
remove Z value as falling/jumping doesn't affect

return characterMovement and from it, call this function that returns bool

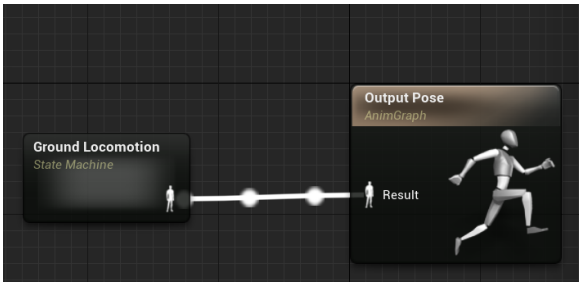
check if Character is standing still or accelerating

Create a animation blueprint base on the aminstance

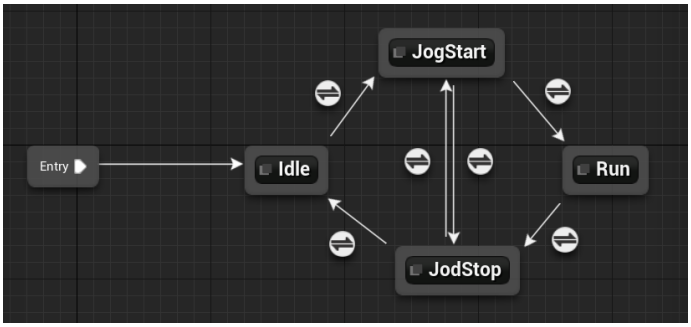




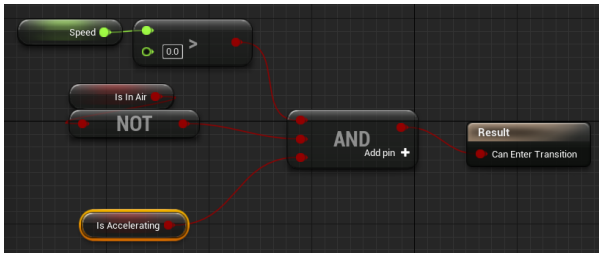
Assign the mesh to the animation blueprint



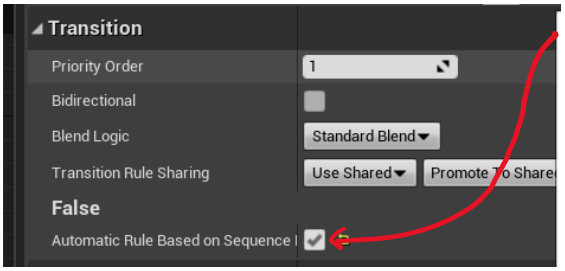
In Ground Locomotion



JogStart



JogStop



Automatic blend two states' animation

```
// Don't rotate when the controller rotates. Let the controller only affect the camera.
bUseControllerRotationPitch = false;
bUseControllerRotationYaw = false;
bUseControllerRotationYaw = false;
```

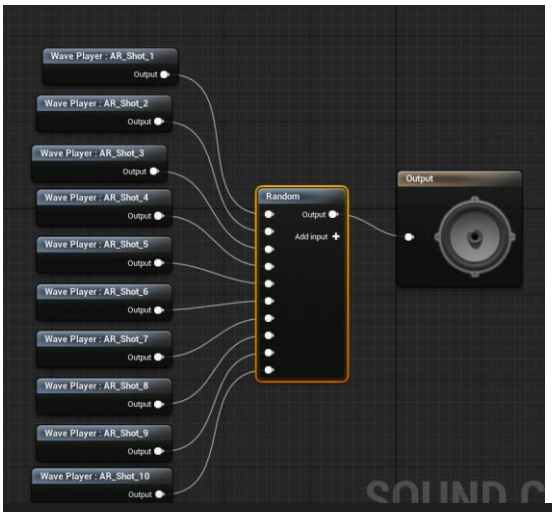
```
// Don't rotate when the controller rotates. Let the controller only affect the camera.
bUseControllerRotationPitch = false;
bUseControllerRotationYaw = false;
bUseControllerRotationYaw = false;

// Configure character movement
GetCharacterMovement()->bOrientRotationToMovement = true; //Character moves in th direction of input...
GetCharacterMovement()->RotationRate = FRotator(0.f, 540.f, 0.f);// ... at this rotation rate
GetCharacterMovement()->JumpZVelocity = 600.f;
GetCharacterMovement()->AirControl = 0.2f;
```

← Character will rotate toward direction of movement

Fire Weapon

```
/* Called when the Fire Button is pressed */
void FireWeapon();
```

Fire Sound




```
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Combat, meta = (AllowPrivateAccess = "true"))
class USoundCue* FireSound;
```

```
void PlaySound2D(const UObject *WorldContextObject, USoundBase *Sound,
```

```
void AShooterCharacter::FireWeapon()
{
    if (FireSound)
    {
        UGameplayStatics::PlaySound2D(this, FireSound);
    }
}
```

as other parameter will be default vale.

USoundCue is derived by USoundBase

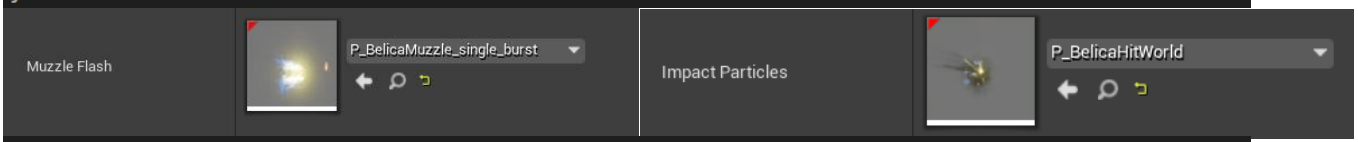
Particle system

```
/* Flash spawned at BarrelScoket*/
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Combat, meta = (AllowPrivateAccess = "true"))
class UParticleSystem* MuzzleFlash;
```

return Mesh, and from the Mesh, get the socket by name

```
const USkeletalMeshSocket* BarrelSocket = GetMesh()->GetSocketByName("BarrelSocket");
if (BarrelSocket) ← if not null
{
    const FTransform SocketTransform = BarrelSocket->GetSocketTransform(GetMesh());

    if (MuzzleFlash)
    {
        UGameplayStatics::SpawnEmitterAtLocation(GetWorld(), MuzzleFlash, SocketTransform);
    }
}
```

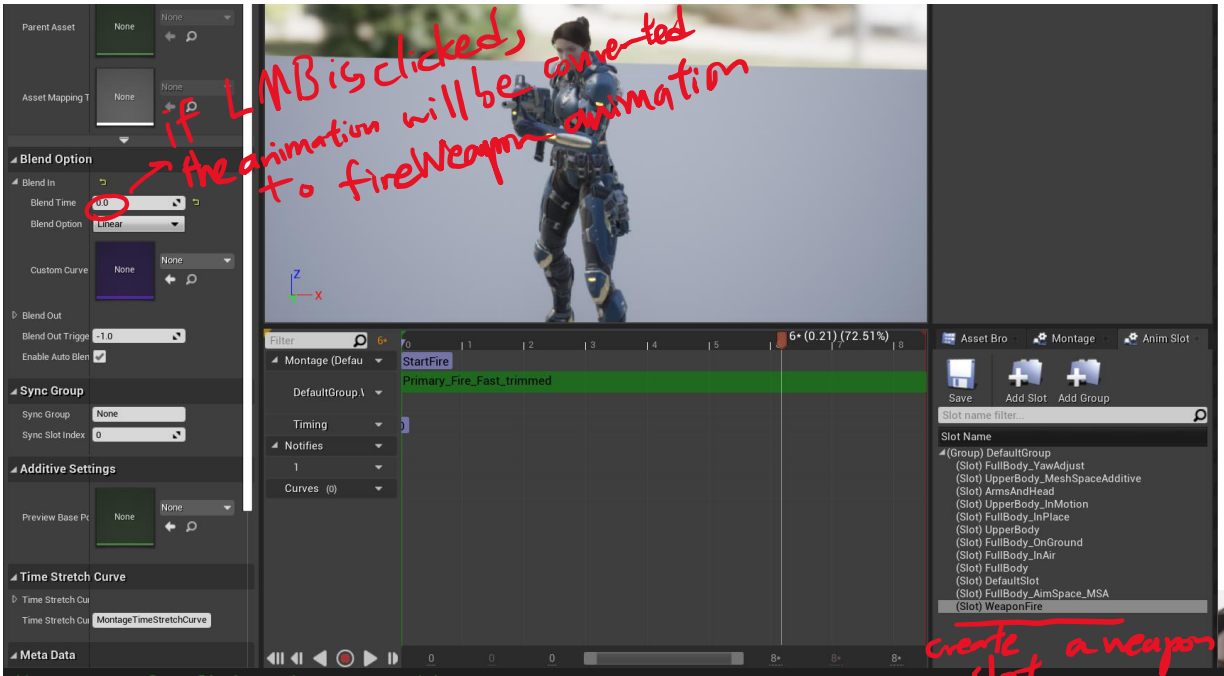
From Barrel Socket, Get Socket Transform from the Mesh and store inside the Socket Transform




```
/* Particles spawned upon bullet impact */
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Combat, meta = (AllowPrivateAccess = "true"))
UParticleSystem* ImpactParticles;
```

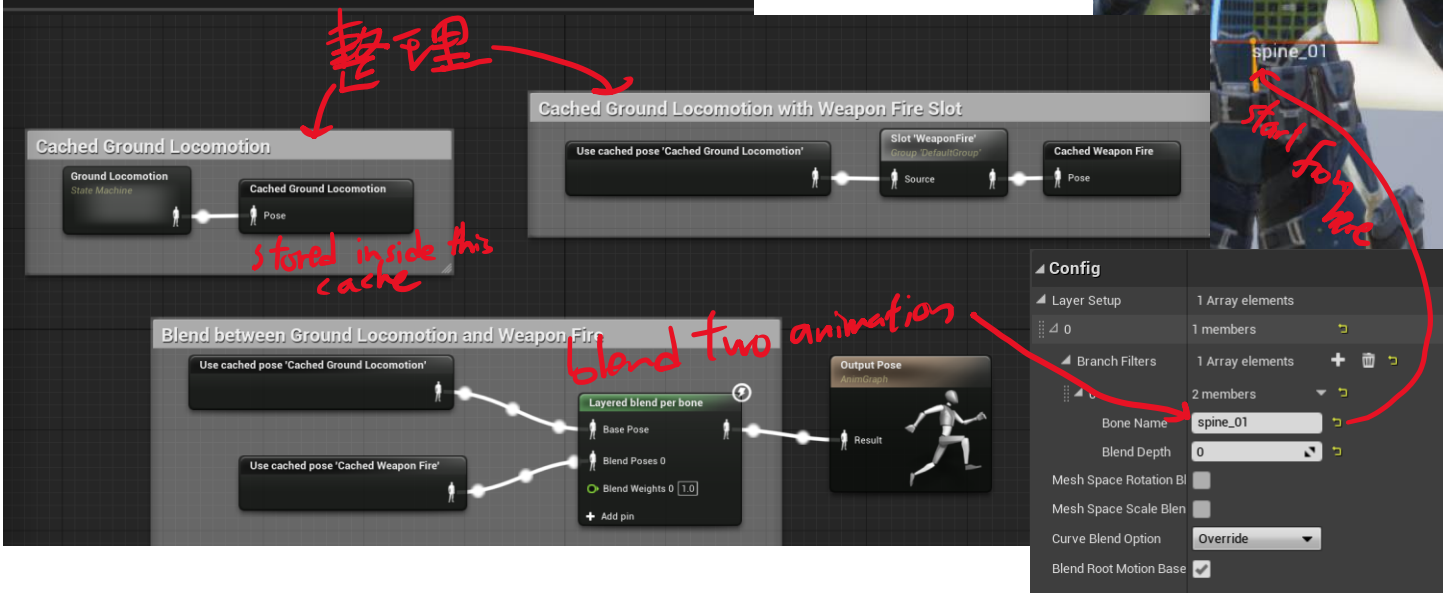
```
if (FireHit.bBlockingHit)
{
    DrawDebugLine(GetWorld(), Start, End, FColor::Red, false, 2.f);
    DrawDebugPoint(GetWorld(), FireHit.Location, 5.f, FColor::Red, false, 2.f);

    if (ImpactParticles)
    {
        UGameplayStatics::SpawnEmitterAtLocation(GetWorld(), ImpactParticles, FireHit.Location);
    }
}
```

Recoil animation




```
/* Montage for firing the weapon */
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Combat, meta = (AllowPrivateAccess = "true"))
class UAnimMontage* HipFireMontage;
```

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



```
FHitResult FireHit;
const FVector Start{ SocketTransform.GetLocation() };
const FQuat Rotation{ SocketTransform.GetRotation() };
const FVector RotationAxis{ Rotation.GetAxisX() };
const FVector End{ Start + RotationAxis * 50'000.f };

GetWorld()->LineTraceSingleByChannel(FireHit, Start, End, ECollisionChannel::ECC_Visibility);
```

draw a line

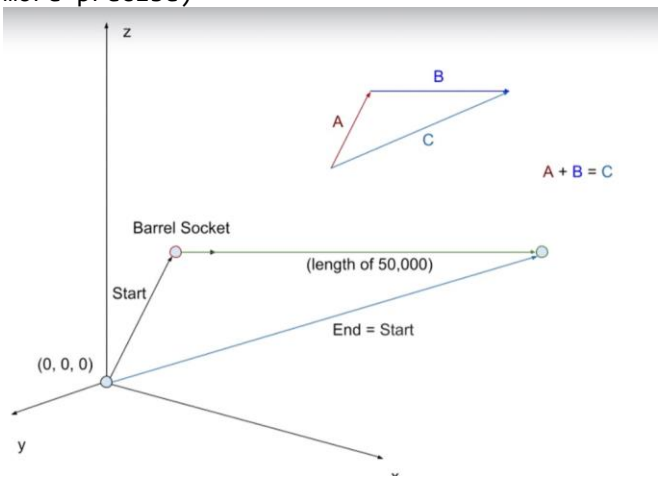
trace will register objects that are set to block the Visibility collision channel

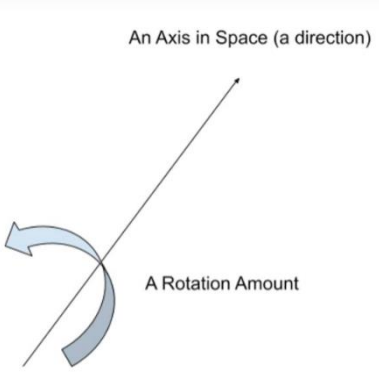
```
bool UWorld::LineTraceSingleByChannel(FHitResult &OutHit,
```

pass by reference so that the HitResult can be changed is there is a collision occur. LineTraceSingleByChannel fills in FHitResult with important information about the line trace, such as the trace hit location.

Quaternions

-Store information about the rotation about an axis in space(similar to FRotator but more precise)





Draw a debug line

```
#include "DrawDebugHelpers.h"
```

```
if (FireHit.bBlockingHit)
{
    DrawDebugLine(GetWorld(), Start, End, FColor::Red, false, 2.f);
    DrawDebugPoint(GetWorld(), FireHit.Location, 5.f, FColor::Red, false, 2.f);
}
```

Beam Particles

```
#include "particles/ParticleSystemComponent.h"
```

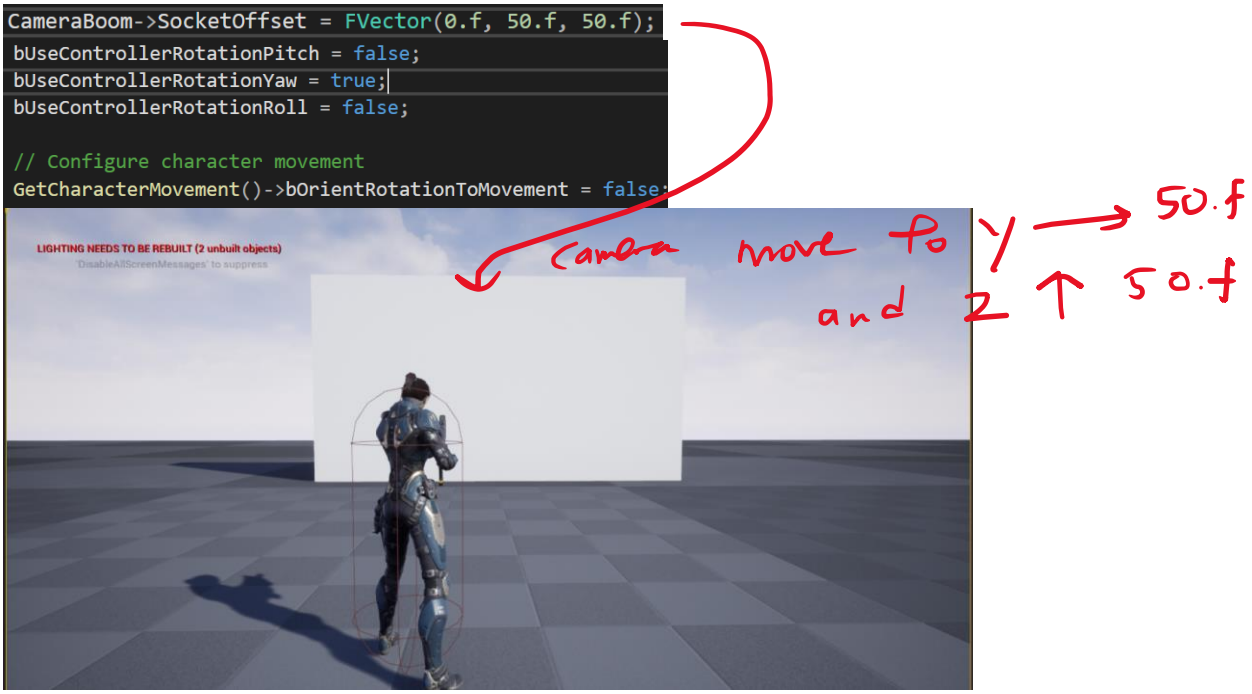
```
/* Smoke trail for bullets */
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Combat, meta = (AllowPrivateAccess = "true"))
UParticleSystem* BeamParticles;
```

```
FVector BeamEndPoint{ End };
```

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

Socket offset

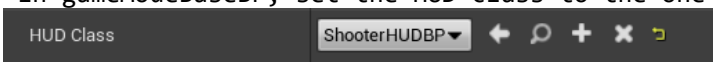
```
CameraBoom->SocketOffset = FVector(0.f, 50.f, 50.f);
bUseControllerRotationPitch = false;
bUseControllerRotationYaw = true;
bUseControllerRotationRoll = false;

// Configure character movement
GetCharacterMovement()->bOrientRotationToMovement = false;
```



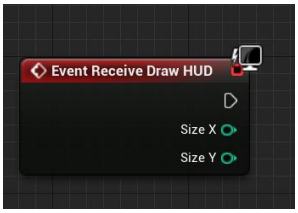
HUD(Head-up-display)

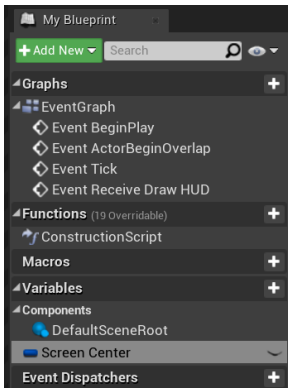
-create a HUD blueprint

-in gameModeBaseBP, set the HUD class to the one you created and named

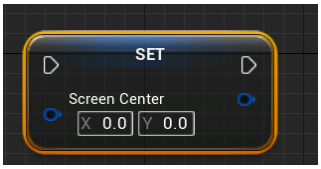


-in ShooterHUDBP, create this node and a 2D vector that store the location of the center of the screen

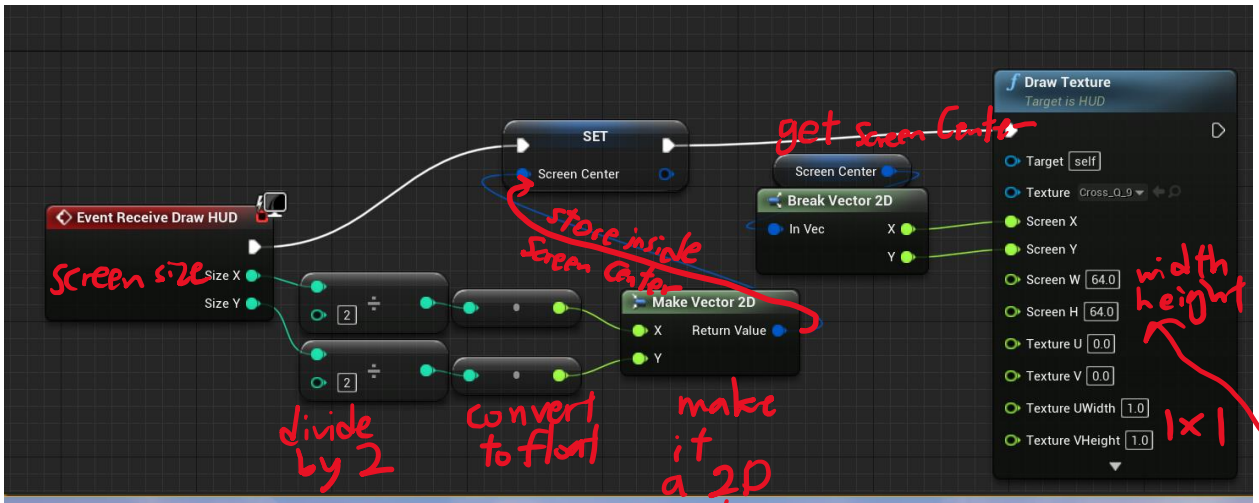


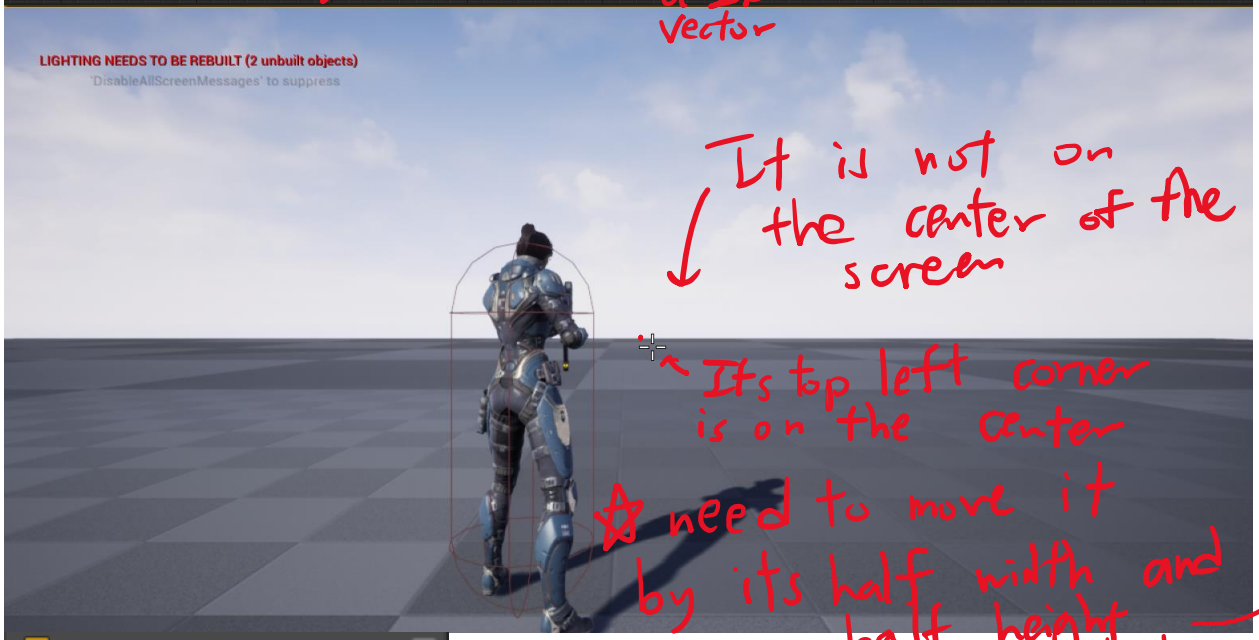


-Drag out the variable and choose set

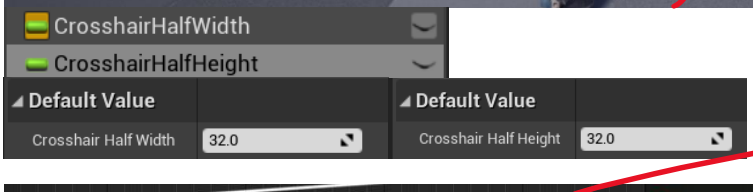


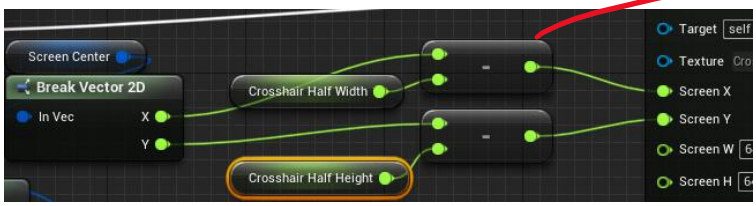
-next page...





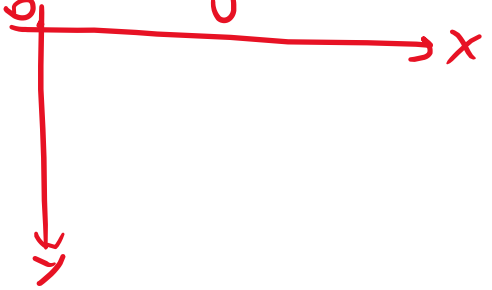
which are 32

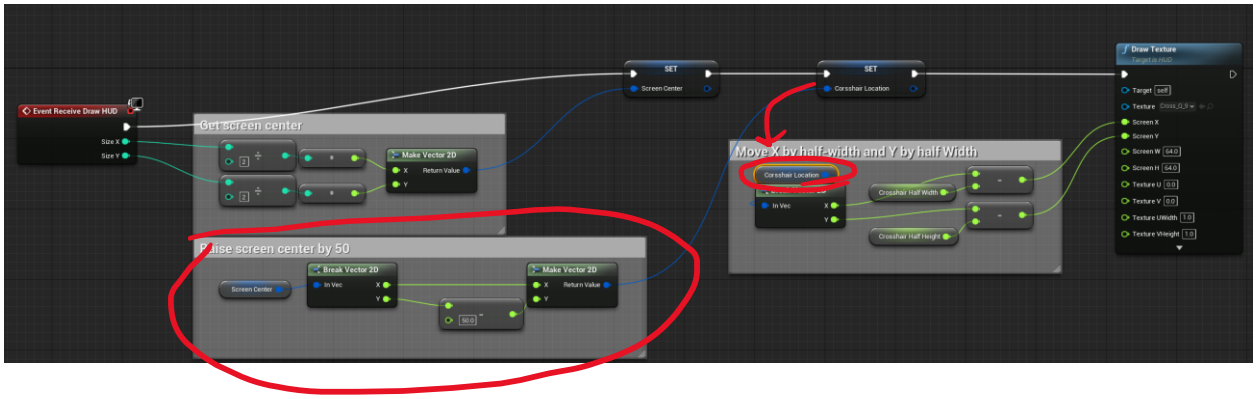




float minus float

Unreal Engin Vector 2D





Crosshair

```
// Get current size of the viewport
FVector2D ViewportSize;
if (GEngine && GEngine->GameViewport)
{
    GEngine->GameViewport->GetViewportSize(ViewportSize);
}
//Get world position and direction of crosshairs
bool bScreenToWorld = UGameplayStatics::DeprojectScreenToWorld(UGameplayStatics::GetPlayerController(this, 0),
    CrosshairLocation, CrosshairWorldPosition, CrosshairWorldDirection);
```

```
// Get screen space location of crosshairs
FVector2D CrosshairLocation(ViewportSize.X / 2.f, ViewportSize.Y / 2.f);
CrosshairLocation.Y -= 50.f;
FVector CrosshairWorldPosition;
FVector CrosshairWorldDirection;
```

const FVector2D &ScreenPosition, FVector &WorldPosition, FVector &WorldDirection)

-DeprojectScreeToWorld will project a 2D item to a 3D space and get its position, if it is success and it will return a true and the pass in parameter will store informations

-pass by reference means the crosshairWorldPosition and crosshairWorldDirection will store important informations about the location and direction

```
    if (bScreenToWorld)//was deprojection successful?
    {
        FHitResult ScreenTraceHit;
        const FVector Start{ CrosshairWorldPosition };
        const FVector End{ CrosshairWorldPosition + CrosshairWorldDirection * 50'000.f };

        // Set beam end point to line trace end point
        FVector BeamEndPoint{ End };

        // Trace outward from crosshairs world location
        GetWorld()->LineTraceSingleByChannel(ScreenTraceHit, Start, End, ECollisionChannel::ECC_Visibility);
        if (ScreenTraceHit.bBlockingHit)// was there a trace hit?
        {
            // Beam end point is now trace hit location
            BeamEndPoint = ScreenTraceHit.Location;
            if (ImpactParticles)
            {
                UGameplayStatics::SpawnEmitterAtLocation(GetWorld(), ImpactParticles, ScreenTraceHit.Location);
            }
        }
        if (BeamParticles)
        {
            UParticleSystemComponent* Beam = UGameplayStatics::SpawnEmitterAtLocation(GetWorld(), BeamParticles, SocketTransform);
            if (Beam)
            {
                Beam->SetVectorParameter(FName("Target"), BeamEndPoint);
            }

        }
```

Problem: if there are object between aiming point and gun barrel, the beam particle will go through the first object, it is because we are tracing from the crosshair location to the world location ,

fixing at next page

Second trace, tracing from the gun barrel

```
// Perform a second trace, this time from the gun barrrel
FHitResult WeaponTraceHit;
const FVector WeaponTraceStart{ SocketTransform.GetLocation() };
const FVector WeaponTraceEnd{ BeamEndPoint };
GetWorld()->LineTraceSingleByChannel(WeaponTraceHit, WeaponTraceStart, WeaponTraceEnd, ECollisionChannel::
if (WeaponTraceHit.bBlockingHit)// object between barrel and BeamEndPoint?
{
    BeamEndPoint = WeaponTraceHit.Location;
}
// Spawn impact particles after updating BeamEndPoint
if (ImpactParticles)
{
    UGameplayStatics::SpawnEmitterAtLocation(GetWorld(), ImpactParticles, BeamEndPoint);
}
```

if there are object between, the the endPoint will be updated

Refactoring the code

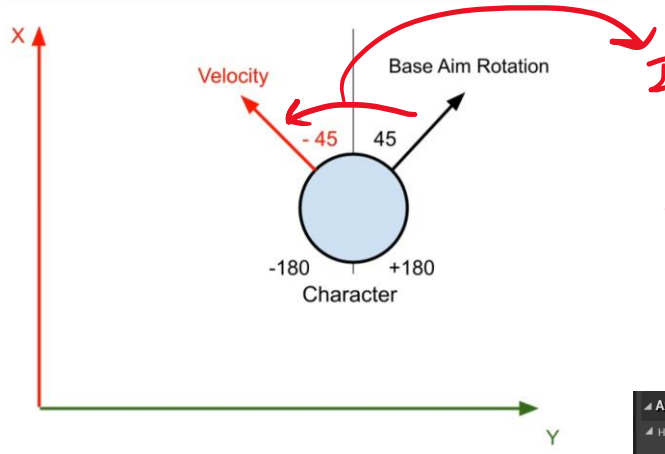
-store the find beam end point into a function

```
bool GetBeamEndLocation(const FVector& MuzzleSocketLocation, FVector& OutBeamLocation);
```

```
bool AShooterCharacter::GetBeamEndLocation(const FVector& MuzzleSocketLocation, FVector& OutBeamLocation)
{
    // Get current size of the viewport
    FVector2D ViewportSize;
    if (GEngine && GEngine->GameViewport)
    {
        GEngine->GameViewport->GetViewportSize(ViewportSize);
    }
    // Get screen space location of crosshairs
    FVector2D CrosshairLocation(ViewportSize.X / 2.f, ViewportSize.Y / 2.f);
    CrosshairLocation.Y -= 50.f;
    FVector CrosshairWorldPosition;
    FVector CrosshairWorldDirection;
    //Get world position and direction of crosshairs
    bool bScreenToWorld = UGameplayStatics::DeprojectScreenToWorld(UGameplayStatics::GetPlayerController(this, 0),
        CrosshairLocation, CrosshairWorldPosition, CrosshairWorldDirection);
    if (bScreenToWorld)//was deprojection successful?
    {
        FHitResult ScreenTraceHit;
        const FVector Start{ CrosshairWorldPosition };
        const FVector End{ CrosshairWorldPosition + CrosshairWorldDirection * 50'000.f };

        OutBeamLocation = End;

        // Trace outward from crosshairs world location
        GetWorld()->LineTraceSingleByChannel(ScreenTraceHit, Start, End, ECollisionChannel::ECC_Visibility);
        if (ScreenTraceHit.bBlockingHit)// was there a trace hit?
        {
            // Beam end point is now trace hit location
            OutBeamLocation = ScreenTraceHit.Location;
        }

        // Perform a second trace, this time from the gun barrrel
        FHitResult WeaponTraceHit;
        const FVector WeaponTraceStart{ MuzzleSocketLocation };
        const FVector WeaponTraceEnd{ OutBeamLocation };
        GetWorld()->LineTraceSingleByChannel(WeaponTraceHit, WeaponTraceStart, WeaponTraceEnd, ECollisionChannel::ECC_Visibility);
        if (WeaponTraceHit.bBlockingHit)// object between barrel and BeamEndPoint?
        {
            OutBeamLocation = WeaponTraceHit.Location;
        }
        return true;
    }
    return false;
}
```

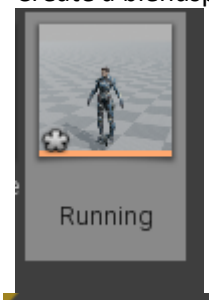
```
FVector BeamEnd;
bool bBeamEnd = GetBeamEndLocation(SocketTransform.GetLocation(), BeamEnd);
if (bBeamEnd)
{
    if (ImpactParticles)
    {
        UGameplayStatics::SpawnEmitterAtLocation(GetWorld(), ImpactParticles, BeamEnd);
    }

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

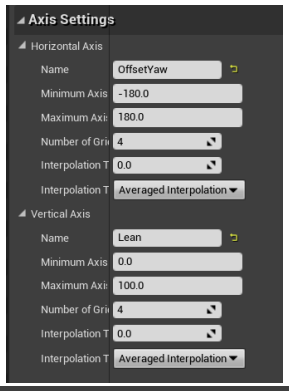
Strafing

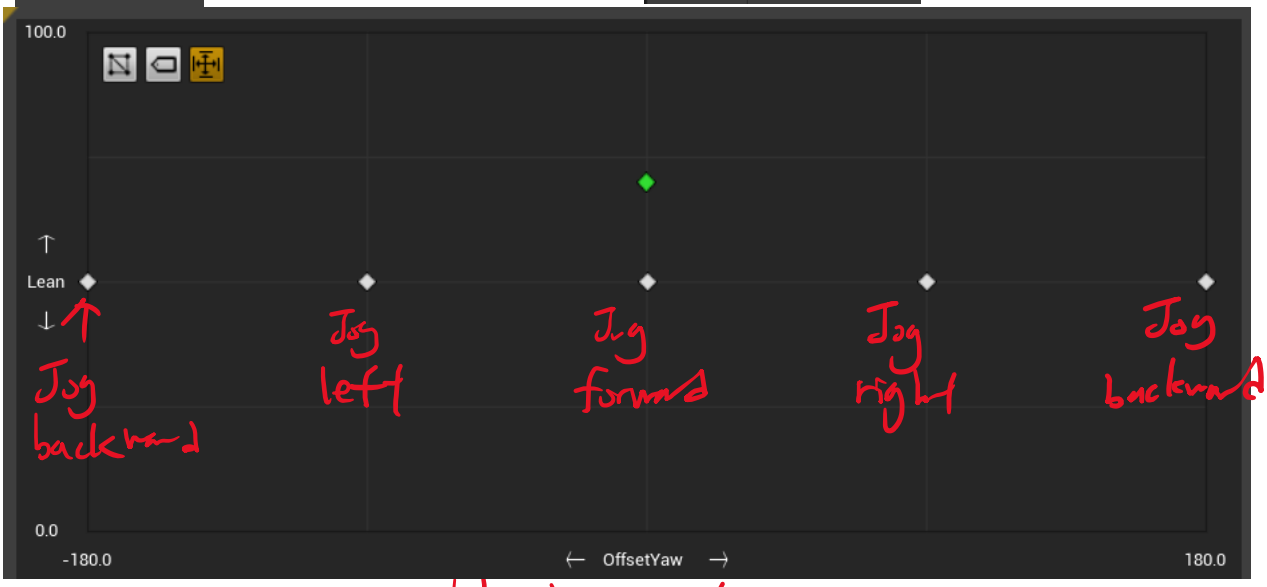


If can get this different, then we can play stafing animation

Create a blendspace(not 1D)







blend animation between -180 and 180

In aminstance

```
/* Offset yaw used for strafing */
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Movement, meta = (AllowPrivateAccess = "true"))
float MovementOffsetYaw;
```

```
#include "Kismet/KismetMathLibrary.h"
```

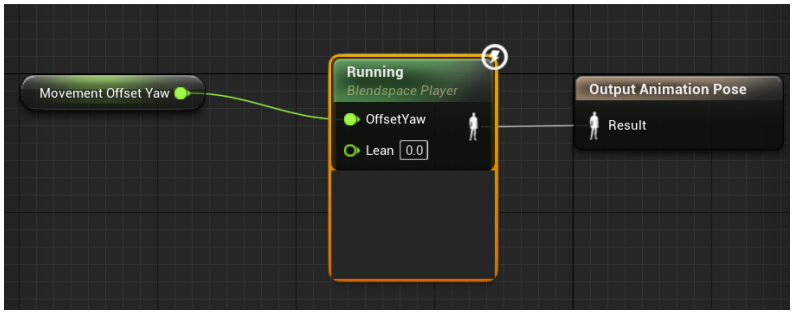
```
FRotator AimRotation = ShooterCharacter->GetBaseAimRotation();
FRotator MovementRotation = UKismetMathLibrary::MakeRotFromX(ShooterCharacter->GetVelocity());

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

做差

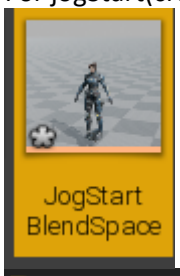
Movement Offset yaw is the difference in yaw between the character's velocity direction and the aim rotation.

At animation blueprint

In run



For jogStart(create blendspace 1D)



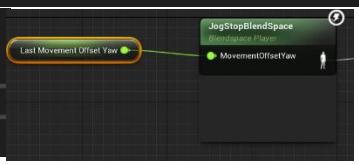
For stopping

-因为stop的时候我们的velocity是0，没有的和aim的方向做差（因为我们拿velocity的方向和aim的方向做差）

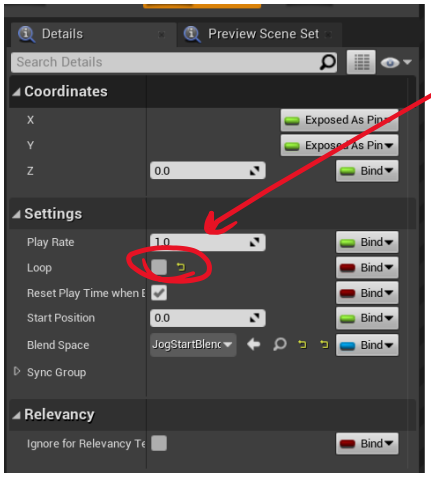
-所以我们要把velocity0之前一瞬间的方向记录起来

```
/* Offset yay the fram before we stopped moving*/
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Movement, meta = (AllowPrivateAccess = "t
float LastMovementOffsetYaw
```

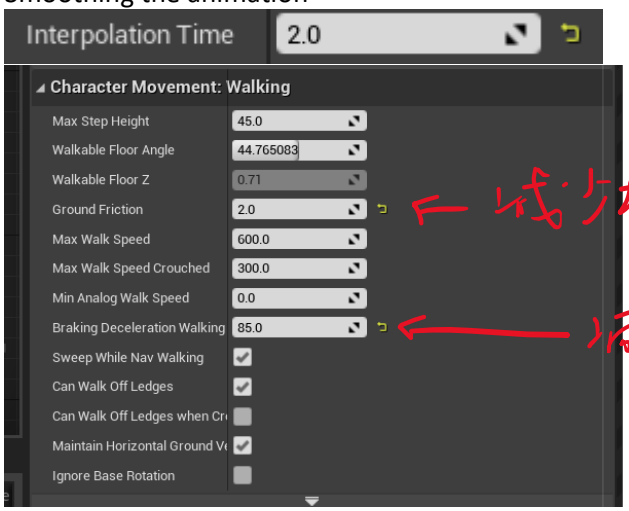
```
if (ShooterCharacter->GetVelocity().Size() > 0.f)
{
    LastMovementOffsetYaw = MovementOffsetYaw;
}
```



-如果那个animation结束后又播多一次，uncheck loop



Smoothing the animation



减少地面摩擦 让它滑一段距离

减少它的 加速度

Aiming(zoom in)

```
PlayerInputComponent->BindAction("AimingButton", IE_Pressed, this, &AShooterCharacter::AimingButtonPressed);
PlayerInputComponent->BindAction("AimingButton", IE_Released, this, &AShooterCharacter::AimingButtonReleased);
/* True when aiming */
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = Combat, meta = (AllowPrivateAccess = "true"))
bool bAiming;

/* Default camera field of view value */
float CameraDefaultFOV;

/* Field of view value for when zoomed in */
float CameraZoomedFOV;

/* Current field of view this frame*/
float CameraCurrentFOV;

/* Interp speed for zooming when aiming*/
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Combat, meta = (AllowPrivateAccess = "true"))
float ZoomInterpSpeed;
```

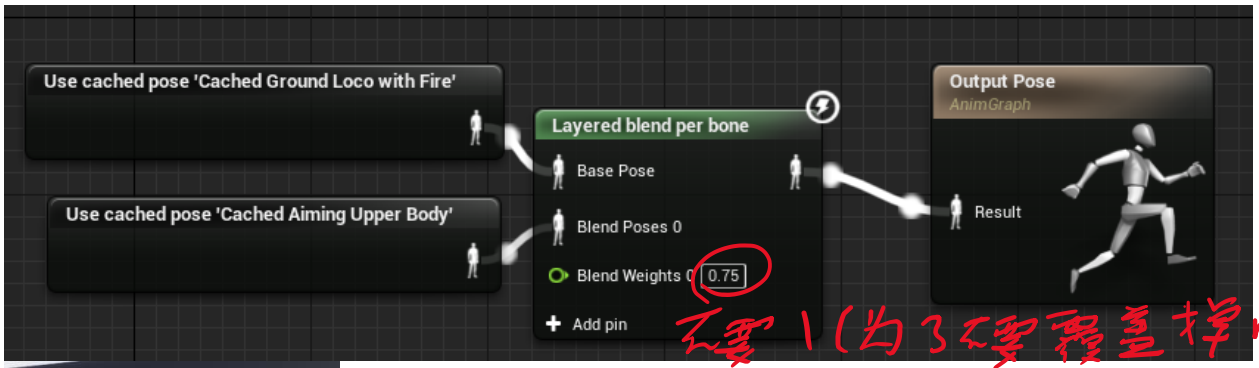
```
bAiming(false),
CameraDefaultFOV(0.f), // set in BeginPlay
CameraZoomedFOV(35.f),
CameraCurrentFOV(0.f),
ZoomInterpSpeed(20.f)
```

```
// Called when the game starts or when spawned
void AShooterCharacter::BeginPlay()
{
    Super::BeginPlay();

    if (FollowCamera)
    {
        CameraDefaultFOV = GetFollowCamera()->FieldOfView;
        CameraCurrentFOV = CameraDefaultFOV;
    }
}
```

```
void AShooterCharacter::AimingButtonPressed()
{
    bAiming = true;
}

void AShooterCharacter::AimingButtonReleased()
{
    bAiming = false;
}
```

```
// Called every frame
void AShooterCharacter::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);

    // Set current camera field of view
    if (bAiming)
    {
        // Interpolate to zoomed FOV
        CameraCurrentFOV = FMath::FInterpTo(CameraCurrentFOV, CameraZoomedFOV, DeltaTime, ZoomInterpSpeed);
    }
    else
    {
        // Interpolate to default FOV
        CameraCurrentFOV = FMath::FInterpTo(CameraCurrentFOV, CameraDefaultFOV, DeltaTime, ZoomInterpSpeed);
    }
    GetFollowCamera()->SetFieldOfView(CameraCurrentFOV);
}
```

Aiming animation



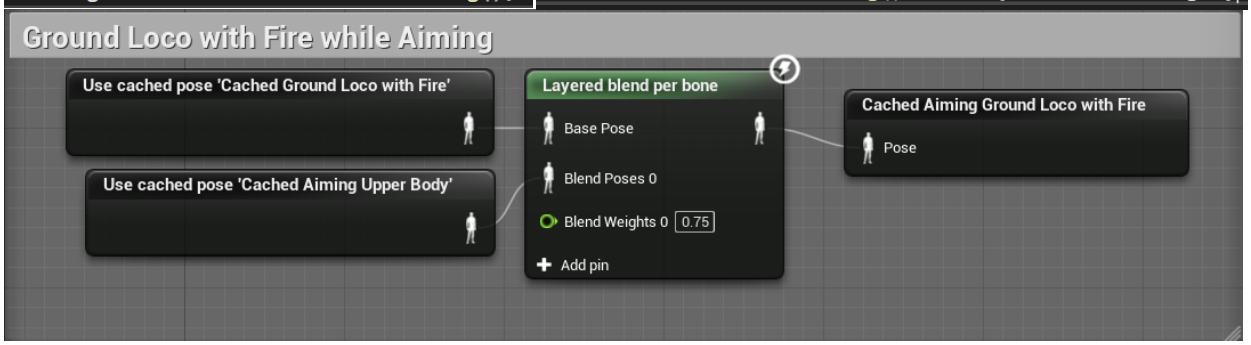
要1（为了要覆盖掉recoil）

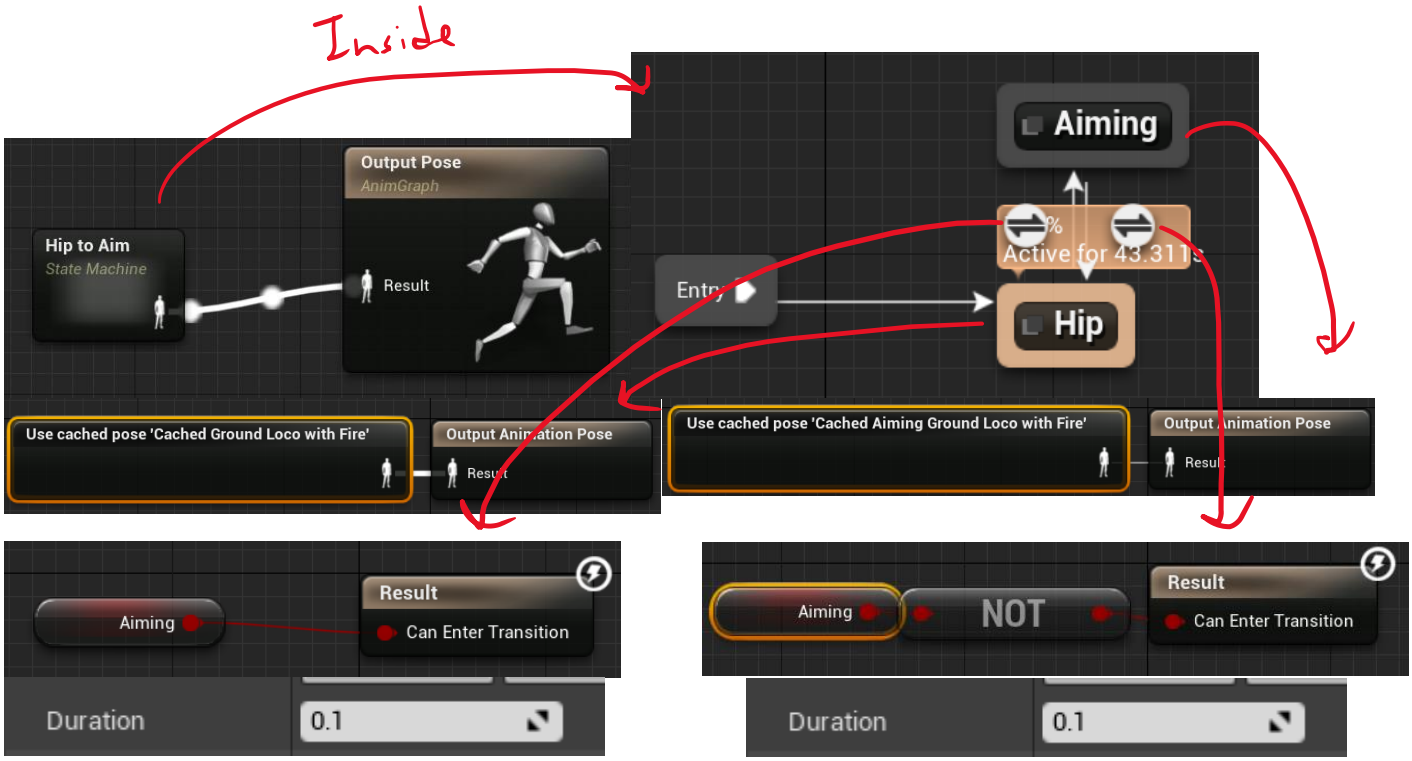
← 现在把重在aim需要fix，

In aminstance

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

bAiming = ShooterCharacter->GetAiming();  FORCEINLINE bool GetAiming() const { return bAiming; }
```

animinstance → ← shooter characte





sensitivity when aiming(for console)

```
/* Turn rate when not aiming */
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = Camera, meta = (AllowPrivateAccess = "true"))
float HipTurnRate;

/* Look up rate when not aiming */
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = Camera, meta = (AllowPrivateAccess = "true"))
float HipLookUpRate;

/* Look up rate when aiming */
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = Camera, meta = (AllowPrivateAccess = "true"))
float AimingLookUpRate;

/* Turn rate when aiming */
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = Camera, meta = (AllowPrivateAccess = "true"))
float AimingTurnRate;
```

```
// Turn rates for aiming/not aiming
HipTurnRate(90.f),
HipLookUpRate(90.f),
AimingTurnRate(20.f),
AimingLookUpRate(20.f),
void AShooterCharacter::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);

    // Handle interpolation for zoom when aiming
    CameraInterpZoom(DeltaTime);
    SetLookRates();

}
```

```
void AShooterCharacter::SetLookRates()
{
    if (bAiming)
    {
        BaseTurnRate = AimingTurnRate;
        BaseLookUpRate = AimingLookUpRate;
    }
    else
    {
        BaseTurnRate = HipTurnRate;
        BaseLookUpRate = HipLookUpRate;
    }
}
```

sensitivity when aiming(for mouse)

```
/*
Rotate controller based on mouse X movement
@param Rate This input value for mouse movement
*/
void Turn(float Value);

/*
    Rotate controller based on mouse Y movement
    @param Rate This input value for mouse movement
*/
void LookUp(float Value);
```

```
PlayerInputComponent->BindAxis("Turn", this, &AShooterCharacter::Turn);
PlayerInputComponent->BindAxis("LookUp", this, &AShooterCharacter::LookUp);
```

```
/* Scale factor for mouse sensitivity. Turn rate when not aiming. */
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category = Camera, meta = (AllowPrivateAccess = "true"), meta = (ClampMin = "0.0", ClampMax = "1.0", UIMin = "0.0", UIMax = "1.0"))
float MouseHipTurnRate;

/* Scale factor for mouse sensitivity. Look up rate when not aiming. */
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category = Camera, meta = (AllowPrivateAccess = "true"), meta = (ClampMin = "0.0", ClampMax = "1.0", UIMin = "0.0", UIMax = "1.0"))
float MouseHipLookUpRate;

/* Scale factor from mouse sensitivity. Turn rate when aiming. */
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category = Camera, meta = (AllowPrivateAccess = "true"), meta = (ClampMin = "0.0", ClampMax = "1.0", UIMin = "0.0", UIMax = "1.0"))
float MouseAimingTurnRate;

/* Scale factor for mouse sensitivity. Look up rate when aiming. */
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category = Camera, meta = (AllowPrivateAccess = "true"), meta = (ClampMin = "0.0", ClampMax = "1.0", UIMin = "0.0", UIMax = "1.0"))
float MouseAimingLookUpRate;
```

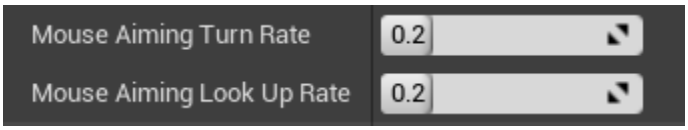
clap for actual value

clamp for slider

這兩個改成 0～1以外的數字

```
// Mouse look sensitivity scale factors
MouseHipTurnRate(1.0f),
MouseHipLookUpRate(1.0f),
MouseAimingTurnRate(0.2f),
MouseAimingLookUpRate(0.2f),
```

```
void AShooterCharacter::Turn(float Value)
{
    float TurnScaleFactor{};
    if (bAiming)
    {
        TurnScaleFactor = MouseAimingTurnRate;
    }
    else
    {
        TurnScaleFactor = MouseHipTurnRate;
    }
    AddControllerYawInput(Value * TurnScaleFactor);
}

void AShooterCharacter::LookUp(float Value)
{
    float LookUpScaleFactor{};
    if (bAiming)
    {
        LookUpScaleFactor = MouseAimingLookUpRate;
    }
    else
    {
        LookUpScaleFactor = MouseHipLookUpRate;
    }
    AddControllerPitchInput(Value * LookUpScaleFactor);
```

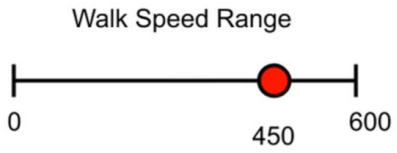


Dynamic crosshair

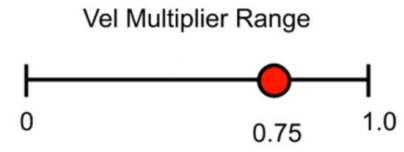
```
/* Determine the spread of the crosshairs */
UPROPERTY(VisibleAnywhere, BlueprintReadonly, Category = Crosshairs, meta = (AllowPrivateAccess = "true"))
float CrosshairSpreadMultiplier;

/* Velocity component for crosshiars spread*/
UPROPERTY(VisibleAnywhere, BlueprintReadonly, Category = Crosshairs, meta = (AllowPrivateAccess = "true"))
float CrosshairVelocityFactor;

/* In air component for crosshiars spread*/
UPROPERTY(VisibleAnywhere, BlueprintReadonly, Category = Crosshairs, meta = (AllowPrivateAccess = "true"))
float CrosshairInAirFactor;

/* Aim component for crosshiars spread*/
UPROPERTY(VisibleAnywhere, BlueprintReadonly, Category = Crosshairs, meta = (AllowPrivateAccess = "true"))
float CrosshairAimFactor;

/* Shooting component for crosshiars spread*/
UPROPERTY(VisibleAnywhere, BlueprintReadonly, Category = Crosshairs, meta = (AllowPrivateAccess = "true"))
float CrosshairShootingFactor;
```

```
void CalculateCrosshairSpread(float DeltaTime);
```

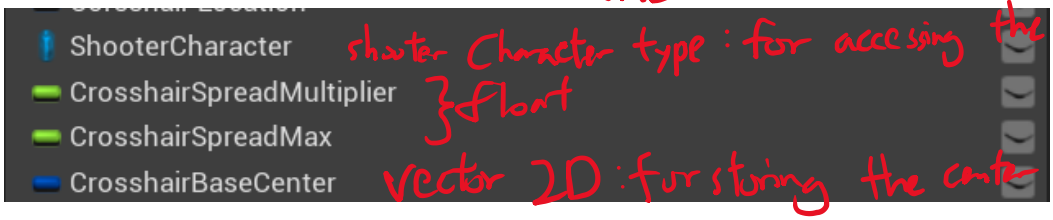
```
void AShooterCharacter::CalculateCrosshairSpread(float DeltaTime)
{
    FVector2D WalkSpeedRange{ 0.f, 600.f };
    FVector2D VelocityMultiplierRange{ 0.f, 1.f };
    FVector Velocity{ GetVelocity() };
    Velocity.Z = 0.f;
    CrosshairVelocityFactor = FMath::GetMappedRangeValueClamped(WalkSpeedRange, VelocityMultiplierRange, Velocity.Size());

    CrosshairSpreadMultiplier = 0.5f + CrosshairVelocityFactor;
}
```



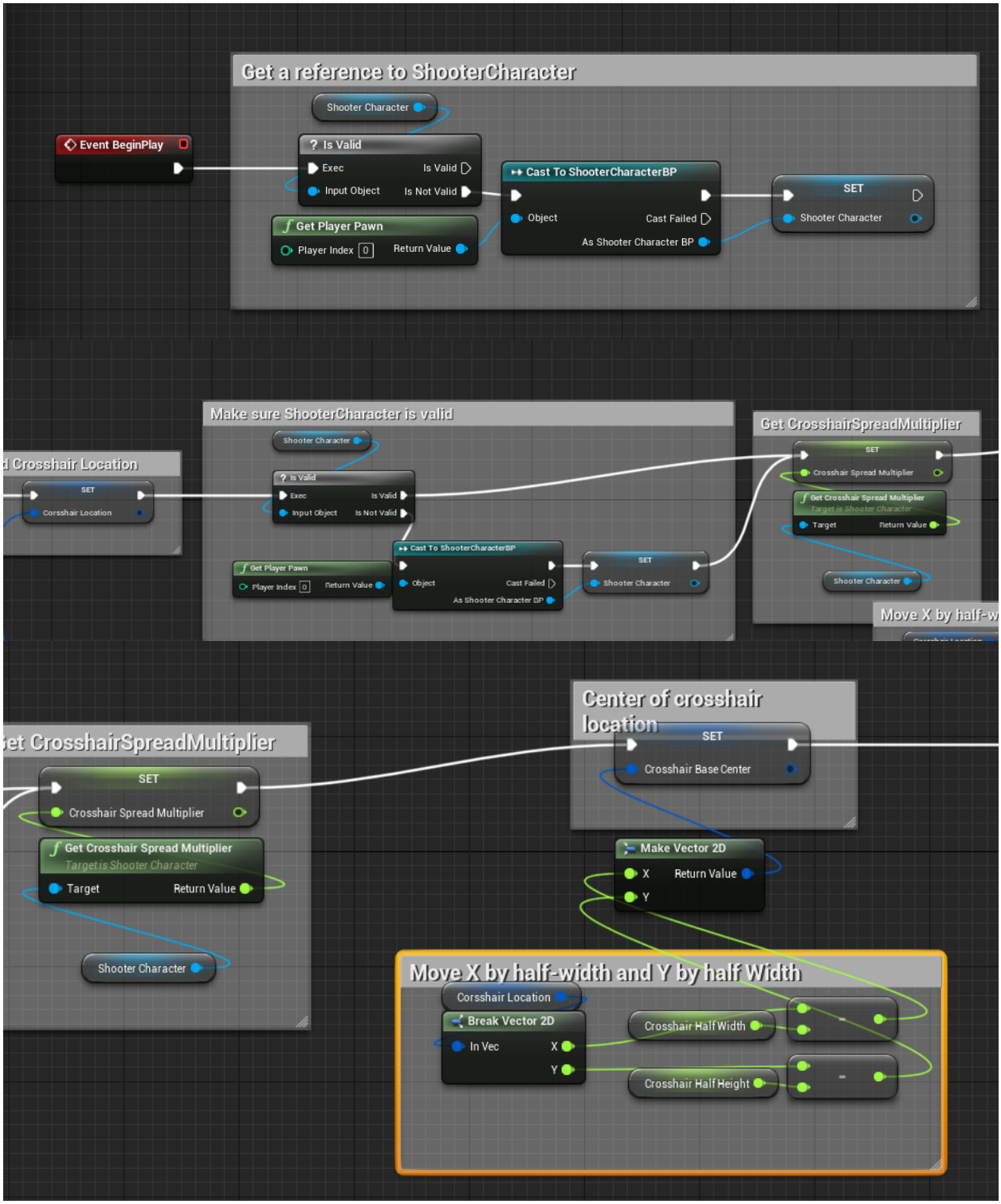
把 0~600 ← max walk speed
↓变成
0~1

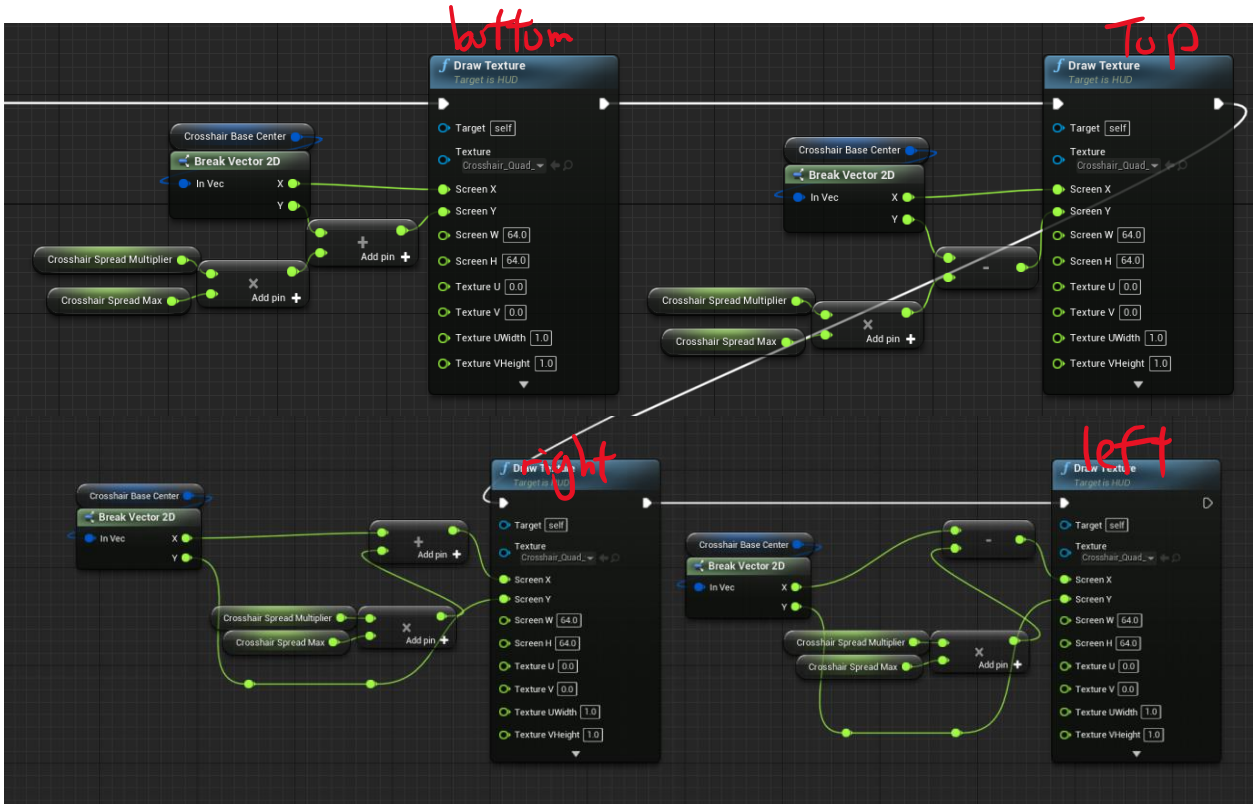


Create these four variable



shooter Character type : for accessing the
3 float
vector 2D : for storing the center location
spread Multiplier


Get a reference to ShooterCharacter
Shooter Character
Event BeginPlay
Is Valid
Exec
Is Valid
Input Object
Is Not Valid
Cast To ShooterCharacterBP
Object
Cast Failed
As Shooter Character BP
SET
Shooter Character
Get Player Pawn
Player Index 0
Return Value
Make sure ShooterCharacter is valid
Get CrosshairSpreadMultiplier
Crosshair Location
Corsshair Location
Crosshair Spread Multiplier
Get Crosshair Spread Multiplier
Target is Shooter Character
Target
Return Value
Move X by half-w
Center of crosshair location
Crosshair Base Center
Make Vector 2D
X
Y
Move X by half-width and Y by half Width
Break Vector 2D
In Vec
Crosshair Half Width
Crosshair Half Height



In air, shooting and aiming factor

```
// Calculate crosshair in air factor
if (GetCharacterMovement()->IsFalling())// is in air?
{
    // Spread the crosshairs slowly while in air
    CrosshairInAirFactor = FMath::FInterpTo(CrosshairInAirFactor, 2.25f, DeltaTime, 2.25f);
}
else// Character is on the ground
{
    // Shrink the crosshairs rapidly while on the ground
    CrosshairInAirFactor = FMath::FInterpTo(CrosshairInAirFactor, 0.f, DeltaTime, 30.f);
}

// Calculate crosshair aim factor
if (bAiming) // Are we aiming?
{
    // Shrink crosshairs a small amount very quickly
    CrosshairAimFactor = FMath::FInterpTo(CrosshairAimFactor, 0.4f, DeltaTime, 30.f);
}
else// Character is on the ground
{
    // Shrink crosshairs a small amount very quickly
    CrosshairAimFactor = FMath::FInterpTo(CrosshairAimFactor, 0.f, DeltaTime, 30.f);
}

// True 0.05 second after firing
if (bFiringBullet)
{
    CrosshairShootingFactor = FMath::FInterpTo(CrosshairShootingFactor, 0.3f, DeltaTime, 60.f);
}
else
{
    CrosshairShootingFactor = FMath::FInterpTo(CrosshairShootingFactor, 0.f, DeltaTime, 60.f);
}
```

```
CrosshairSpreadMultiplier = 0.3f + CrosshairVelocityFactor + CrosshairInAirFactor - CrosshairAimFactor + CrosshairShootingFactor;
```

For shooting factor, we need a timer dat is called when firing bullet and stop after 0.5 sec and some variables

```
float ShootTimeDuration;

bool bFiringBullet;

FTimerHandle CrosshairShootTimer;
```

```
// Bullet fire timer variables
ShootTimeDuration(0.05f),
bFiringBullet(false)
```

Call the function at FireWeapon() function

```
// Start bullet fire timer for crosshairs
StartCrosshairBulletFire();
```

```
void StartCrosshairBulletFire();

UFUNCTION()
void FinishCrosshairBulletFire();
```

```
void AShooterCharacter::StartCrosshairBulletFire()
{
    bFiringBullet = true;

    GetWorldTimerManager().SetTimer(CrosshairShootTimer, this,&AShooterCharacter::FinishCrosshairBulletFire, ShootTimeDuration);
}

void AShooterCharacter::FinishCrosshairBulletFire()
{
    bFiringBullet = false;
}
```

set a timer

call this function when time's up

Automanic fire

```
/* Left mouse button or right console trigger pressed */
bool bFireButtonPressed;

/* True when we can fire. False when waiting for the timer */
bool bShouldFire;

/* Rate of automatic gun fire*/
float AutomaticFireRate;

/* Seta timer between gunshots*/
FTimerHandle AutoFireTimer;
```

```
// Automatic fire variables
AutomaticFireRate(0.1f),
bShouldFire(true),
bFireButtonPressed(false)
```

```
void FireButtonPressed();
void FireButtonReleased();
void StartFireTimer();
UFUNCTION()
void AutoFireReset();
```

```
PlayerInputComponent->BindAction("FireButton", IE_Pressed, this, &AShooterCharacter::FireButtonPressed);
PlayerInputComponent->BindAction("FireButton", IE_Released, this, &AShooterCharacter::FireButtonReleased);
```

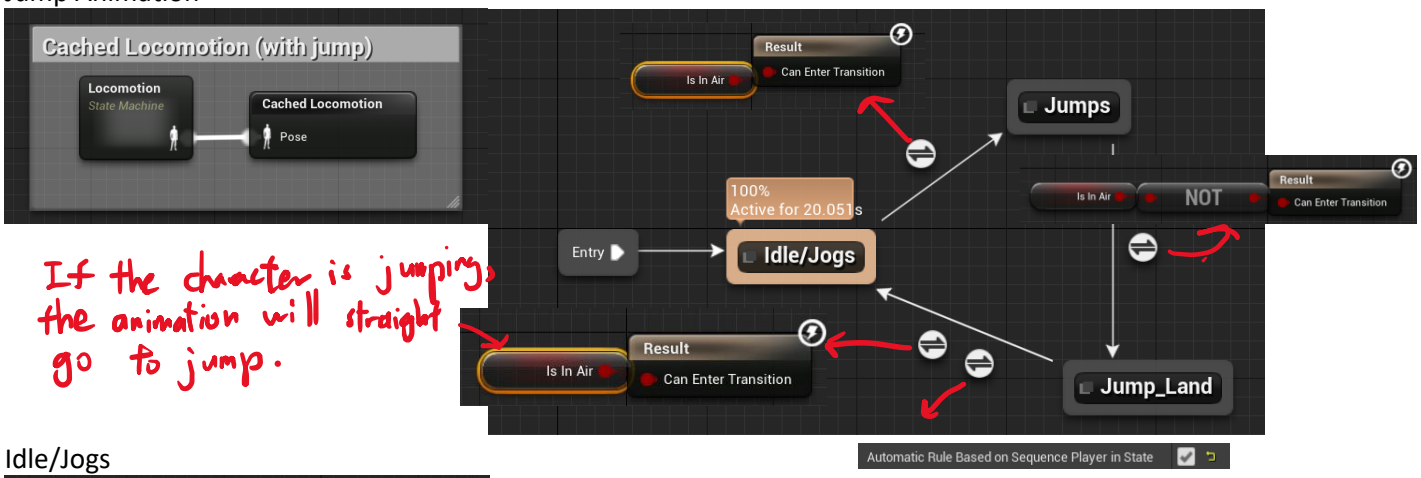
```
void AShooterCharacter::FireButtonPressed()
{
    bFireButtonPressed = true;
    StartFireTimer();
}

void AShooterCharacter::FireButtonReleased()
{
    bFireButtonPressed = false;
}

void AShooterCharacter::StartFireTimer()
{
    if (bShouldFire)
    {
        FireWeapon();
        bShouldFire = false;
        GetWorldTimerManager().SetTimer(AutoFireTimer, this, &AShooterCharacter::AutoFireReset, AutomaticFireRate);
    }
}

void AShooterCharacter::AutoFireReset()
{
    bShouldFire = true;
    if (bFireButtonPressed)
    {
        StartFireTimer();
    }
}
```

Mouse button pressed →set `bFireButtonPressed to true and call StartFireTimer()`
→`if ShouldFire→fire weapon and set Should fire to false`
→ `set timer for duration of AutomacticFireRate(0.1sec)→times up and call AutoFireReset`
→set ShouldFire to true, and now the weapon can be fired again→ is FireButtonPressed is true
→back to call StartFireTimer()

功能

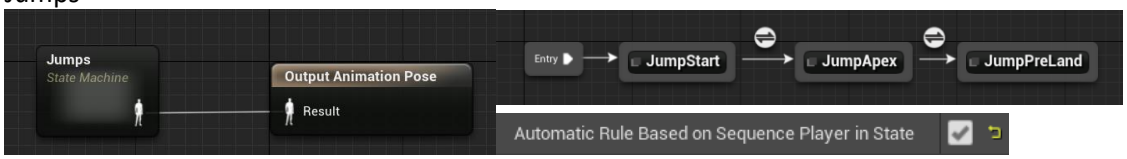
- AutomaticFireRate:两次发射之间的时长
- bShouldFire: 让枪在发射后的0.1秒才能继续发射
- StartFireTimer：发射后开始计时，时间到才让set bShouldFire 成true
- AutoFireReset：发射了 0.1 秒后会被呼叫然后set bShouldFire 成true 顺便检查是否还按着发射健，如果是就呼叫StartFireTimer

Jump Animation

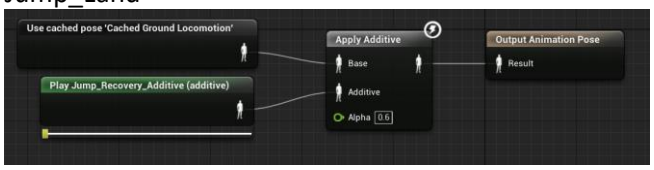


Idle/Jogs

Jumps

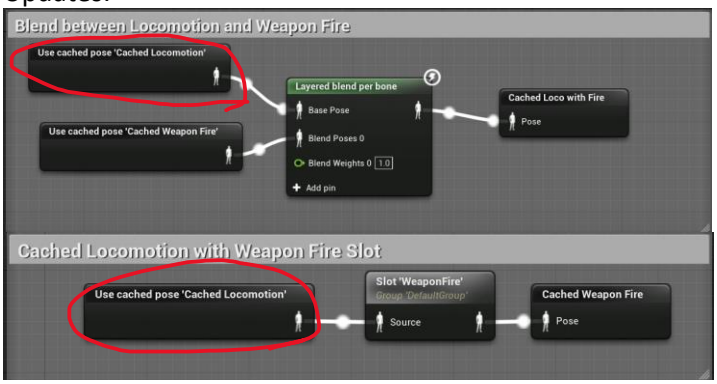


Jump_Land

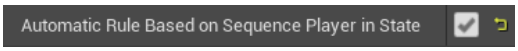


-now we created a state machine which contains all jumping animation, and ground locomotion, hence we need to update those node which still using ground locomotion and change it to the new cached locomotion

Updates:

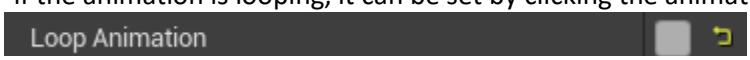


-and remember to set those jumping animation to this

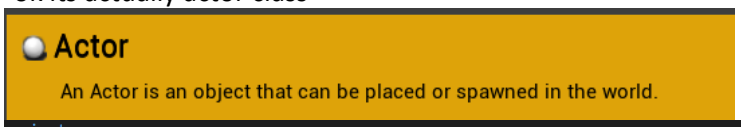


For automatic change state if the animation is finish.

-if the animation is looping, it can be set by clicking the animation node



Item class(base class for weapon class and etc)
-ok its actually actor class

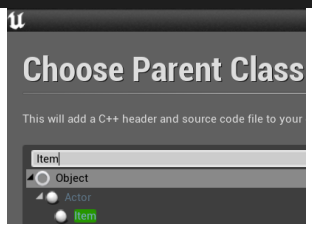



```
private:
    //  Skeletal Mesh for the item
    UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = " Item Properties", meta = (AllowPrivateAccess = "t
    USkeletalMeshComponent* ItemMesh;

    // Line trace collides with box to show HUD widgets
    UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = " Item Properties", meta = (AllowPrivateAccess = "true
    class UBoxComponent* CollisionBox;
```

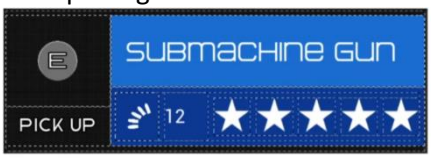
```
ItemMesh = CreateDefaultSubobject<USkeletalMeshComponent>(TEXT("ItemMesh"));
SetRootComponent(ItemMesh);

CollisionBox = CreateDefaultSubobject<UBoxComponent>(TEXT("CollisionBox"));
CollisionBox->SetupAttachment(ItemMesh);
```

```
#include "Components/WidgetComponent.h"
PickupWidget = CreateDefaultSubobject<UWidgetComponent>(TEXT("PickupWidget"));
PickupWidget->SetupAttachment(RootComponent);
```

Weapon Class
-create weapon class derived from item class



Pickup Widget



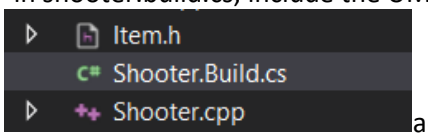
- Use UMG - Unreal Motion Graphics
- Create a Widget Blueprint

-Display above item
-Show which key to press(E)
-an item icon(ammo.etc)
-an item amount(ammo count)
-item rarity(number of stars,color)

-in item.h

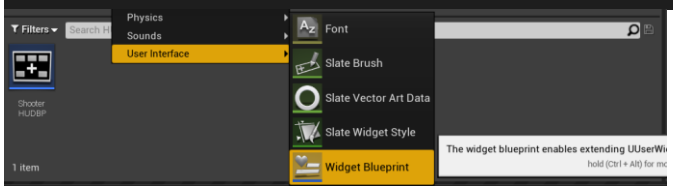
```
// Popup widget for when the player looks at the item
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = " Item Properties", meta = (AllowPrivateAccess = "true
class UWidgetComponent* PickupWidget;
```

-in shooter.build.cs, include the UMG module



add ,"UMG"

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



-Widget 的制作过程就不纪录了太麻烦了

-in Base Weapon Blueprint, select pickup widget

In item.cpp

```cpp
void AItem::BeginPlay()
{
    Super::BeginPlay();

    // Hide Pickup Widget
    PickupWidget->SetVisibility(false);
}
```

-now we need to write a function that detect the crosshair when hover over the weapon, and we need to do it in shootercharacter, similar to the beam trace

```cpp
// Line trace  for items under the crosshairs
bool TraceUnderCrosshairs(FHitResult& OutHitResult);
```

```cpp
bool AShooterCharacter::TraceUnderCrosshairs(FHitResult& OutHitResult)
{
    // Get Viewport Size
    FVector2D ViewportSize;
    if (GEngine && GEngine->GameViewport)
    {
        GEngine->GameViewport->GetViewportSize(ViewportSize);
    }
    // Get screen space location of crosshairs
    FVector2D CrosshairLocation(ViewportSize.X / 2.f, ViewportSize.Y / 2.f);
    FVector CrosshairWorldPosition;
    FVector CrosshairWorldDirection;

    //Get world position and direction of crosshairs
    bool bScreenToWorld = UGameplayStatics::DeprojectScreenToWorld(UGameplayStatics::GetPlayerController(this, 0),
        CrosshairLocation, CrosshairWorldPosition, CrosshairWorldDirection);
    if (bScreenToWorld)
    {
        // Trace from Crosshair world location outward
        const FVector Start{CrosshairWorldPosition};
        const FVector End{ Start + CrosshairWorldDirection * 50'000.f };
        GetWorld()->LineTraceSingleByChannel(OutHitResult, Start, End, ECollisionChannel::ECC_Visibility);
        if (OutHitResult.bBlockingHit)
        {
            return true;
        }
    }
    return false;
}
```

```cpp
void AShooterCharacter::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);

    // Handle interpolation for zoom when aiming
    CameraInterpZoom(DeltaTime);
    // Change look sensitivity based on aiming
    SetLookRates();
    // Caculate crosshair spread multiplier
    CalculateCrosshairSpread(DeltaTime);

    FHitResult ItemTraceResult;
    TraceUnderCrosshairs(ItemTraceResult);
    if (ItemTraceResult.bBlockingHit)
    {
        AItem* HitItem = Cast<AItem>(ItemTraceResult.Actor);
        if (HitItem && HitItem->GetPickupWidget())
        {
            // Show Item's Pickup Widget
            HitItem->GetPickupWidget()->SetVisibility(true);
        }
    }
}
```

checking valid

-call the function every tick in shootercharacter.cpp
-if it hit something, cast the item into AItem class
-to set the pickup widget visibility to true, we construct a getter function in item.h which return the PickupWidget
-and we access the item and set the widget visibility to true

```cpp
FORCEINLINE UWidgetComponent* GetPickupWidget() const { return PickupWidget; }
```

-the collision box in weapon item can affect the hit result, hence it need to be change to invisible and not response to LineTraceSingleByChannel

```cpp
CollisionBox->SetCollisionResponseToAllChannels(ECollisionResponse::ECR_Ignore);
CollisionBox->SetCollisionResponseToChannel(ECollisionChannel::ECC_Visibility, ECollisionResponse::ECR_Block);
```

-this function is very similar to GetBeamEndLocation which do line trace for the weapon fire, we can actually use the TraceUnderCrosshairs in GetBeamEndLocation for first trace
-因为在GetBeamEndLocation，如果没有打到东西，beam的终点会是 的地方，所以为了再利用我们写的TraceUnderCrosshairs在GetBeamEndLocation里面， 我们需要多一个parameter纪录beam的终点，并不是写给我们的widget的

```cpp
bool AShooterCharacter::TraceUnderCrosshairs(FHitResult& OutHitResult, FVector& OutHitLocation)
{
const FVector End{ Start + CrosshairWorldDirection * 50'000.f };
OutHitLocation = End;
GetWorld()->LineTraceSingleByChannel(OutHitResult, Start, End, ECollisionChannel::ECC_Visibility);
if (OutHitResult.bBlockingHit)
{
    OutHitLocation = OutHitResult.Location;
    return true;
}
```

如果没有打中，就会by default 在End

如果有打中，就会在那物件的地方

```cpp
bool AShooterCharacter::GetBeamEndLocation(const FVector& MuzzleSocketLocation, FVector& OutBeamLocation)
{
    // Check for crosshair trace hit
    FHitResult CrosshairHitResult;
    bool bCrosshairHit = TraceUnderCrosshairs(CrosshairHitResult, OutBeamLocation);

    if (bCrosshairHit)
    {
        // Tentative beam location - still need to trace from gun
        OutBeamLocation = CrosshairHitResult.Location;
    }
    else // no crosshair trace hit
    {
        //  OutBeamLocation is the End location for the line trace
    }
    // Perform a second trace, this time from the gun barrrel
    FHitResult WeaponTraceHit;
    const FVector WeaponTraceStart{ MuzzleSocketLocation };
    const FVector StartToEnd{ OutBeamLocation - MuzzleSocketLocation };
    const FVector WeaponTraceEnd{ OutBeamLocation + StartToEnd*1.25f};
    GetWorld()->LineTraceSingleByChannel(WeaponTraceHit, WeaponTraceStart, WeaponTraceEnd, ECollisionChannel::ECC_Visibility);
    if (WeaponTraceHit.bBlockingHit)// object between barrel and BeamEndPoint?
    {
        OutBeamLocation = WeaponTraceHit.Location;
        return true;
    }
    return false;
}
```

现在GetBeamEndLocation比较干净了

-now when we hover over the weapon the widget visibility will be true but it not going to be set back to false, so we need to fix it and even we are far from the object, the widget will still show up

-in order to display the widget only we are close enough, we can give our weapon a sphere component, when character is overlap with the sphere, it can perform line tracing

-in item.h(#include sphereComponent.h)

```
// Enable item tracing when overlapped
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = " Item Properties", meta = (AllowPrivateAccess = "true"))
class USphereComponent* AreaSphere;
```

```
AreaSphere = CreateDefaultSubobject<USphereComponent>(TEXT("AreaSphere"));
AreaSphere->SetupAttachment(RootComponent);
```

```
// Call when overlapping AreaSphere
UFUNCTION()
void OnSphereOverlap(
    UPrimitiveComponent* OverlappedComponent,
    AActor* OtherActor,
    UPrimitiveComponent* OtherComp,
    int32 OtherBodyIndex,
    bool bFromSweep,
    const FHitResult& SweepResult);
// Call when End overlapping AreaSphere
UFUNCTION()
void OnSphereEndOverlap(UPrimitiveComponent* OverlappedComponent,
    AActor* OtherActor,
    UPrimitiveComponent* OtherComp,
    int32 OtherBodyIndex);
// Hide Pickup Widget
PickupWidget->SetVisibility(false);

// Setup overlap for AreaSphere
AreaSphere->OnComponentBeginOverlap.AddDynamic(this, &AItem::OnSphereOverlap);
AreaSphere->OnComponentEndOverlap.AddDynamic(this, &AItem::OnSphereEndOverlap);
```

```
void AItem::OnSphereOverlap(UPrimitiveComponent* OverlappedComponent, AActor* Other
{
    if (OtherActor)
    {
        AShooterCharacter* ShooterCharacter = Cast<AShooterCharacter>(OtherActor);
        if (ShooterCharacter)
        {
            ShooterCharacter->IncrementOverlappedItemCount(1);
        }
    }
}
void AItem::OnSphereEndOverlap(UPrimitiveComponent* OverlappedComponent, AActor* Ot
{
    if (OtherActor)
    {
        AShooterCharacter* ShooterCharacter = Cast<AShooterCharacter>(OtherActor);
        if (ShooterCharacter)
        {
            ShooterCharacter->IncrementOverlappedItemCount(-1);
        }
    }
}
```

这两个function里面的parameter会有很多overlap的information

给AreaSphere进行Overlap的function其中要pass in的2个parameter的第二个function很特别

OtherActor就是和这个物体overlap的物体，把它cast成shooterchar，如果成功，那么代表Character有和物体overlap，进行这个function，overlap的物体为0一overlap结束则减一，这两个function的功能在下面

-shooter character.h

```
// True if we should trace every frame for items
bool bShouldTraceForItem;
// Number of overlapped AItem
int8 OverlappedItemCount;
// Line trace  for items under the crosshairs
bool TraceUnderCrosshairs(FHitResult& OutHitResult, FVector& OutHitLocation);

// Trace for items if OverlappedItemCount > 0
void TraceForItems();
FORCEINLINE int8 GetOverlappedItemCount() const { return OverlappedItemCount; }

// Add/Subtracts to/from OverlappedItemCount and updates bShouldTraceForItems
void IncrementOverlappedItemCount(int8 Amount);
```

-shooter character.cpp (construtor)

```
// Item trace variables
bShouldTraceForItem(false)
```

```
void AShooterCharacter::IncrementOverlappedItemCount(int8 Amount)
{
    if (OverlappedItemCount + Amount <= 0)
    {
        OverlappedItemCount = 0;
        bShouldTraceForItems = false;
    }
    else
    {
        OverlappedItemCount += Amount;
        bShouldTraceForItems = true;
    }
}
```

接着上面的，如果这个function 被called，OverlappedItemCount（重叠的物体数量）就会加上pass in的数字，+1 or -1，如果-OverlappedItemCount 少过0，就让他变零（没有负个物体），bShouldTraceForItems 会是false（不会进行line trace）

-相反OverlappedItemCount大于0则要检擦crosshair是否有在物体上（line trace）

```
void AShooterCharacter::TraceForItems()
{
    if (bShouldTraceForItems)
    {
        FHitResult ItemTraceResult;
        FVector HitLocation;
        TraceUnderCrosshairs(ItemTraceResult, HitLocation);
        if (ItemTraceResult.bBlockingHit)
        {
            AItem* HitItem = Cast<AItem>(ItemTraceResult.Actor);
            if (HitItem && HitItem->GetPickupWidget())
            {
                // Show Item's Pickup Widget
                HitItem->GetPickupWidget()->SetVisibility(true);
            }
        }
    }
}
```

-如果line trace有hit到物体，就尝试把那个物体cast成item 如果casting 成功，那么久把那个item里面的widget 的visibility调成true（如果hit到的不是AItem 那么casting就会失败）

-把这个function 放进tick里面每秒检擦是否要做line tracing

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

    // Check OverlappedItemCount, then trace for items
    TraceForItems();
```

-set visibility to false

My solution:

```
void AItem::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);
    PickupWidget->SetVisibility(false);
}
```

Instructor solution:

```
// The AItem we hit last frame
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = Items, meta = (AllowPrivateAccess = "t
class AItem* TraceHitItemLastFrame;
```

```
if (bShouldTraceForItems)
{
    FHitResult ItemTraceResult;
    FVector HitLocation;
    TraceUnderCrosshairs(ItemTraceResult, HitLocation);
    if (ItemTraceResult.bBlockingHit)
    {
        AItem* HitItem = Cast<AItem>(ItemTraceResult.Actor);
        if (HitItem && HitItem->GetPickupWidget())
        {
            // Show Item's Pickup Widget
            HitItem->GetPickupWidget()->SetVisibility(true);
        }

        // We hit an AItem last frame
        if (TraceHitItemLastFrame)
        {
            if (HitItem!= TraceHitItemLastFrame)
            {
                // We are hitting a different AItem this frame from last frame
                // Or AItem is null
                TraceHitItemLastFrame->GetPickupWidget()->SetVisibility(false);
            }
        }

        // Store a reference to HitItem for next frame
        TraceHitItemLastFrame = HitItem;
    }
}
else if (TraceHitItemLastFrame)
{
    // No longer overlapping any items,
    // Item last frame should not show widget
    TraceHitItemLastFrame->GetPickupWidget()->SetVisibility(false);
}
```

为了下一frame把这frame的物体存起来

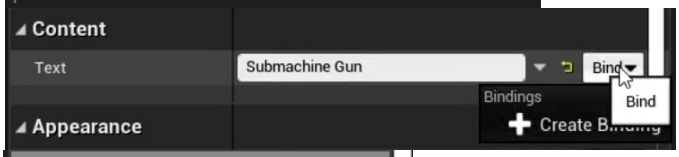
如果现在的对着的物体和上一个物体不一样，上一个的visibility就要false

或者上一个物体不是null（存在）而这一frame并没有trace到任何物体，上一个frame的物体visibility也要false

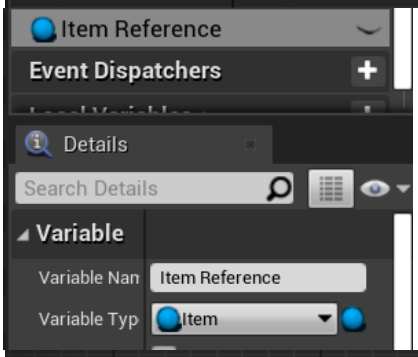
-Bind property of the widget

```
// The name which appears on the Pickup Widget
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = " Item Properties", meta = (AllowPrivateAccess = "true"
FString ItemName;
```

```
AItem::AItem() :
    ItemName(FString("Default"))
```

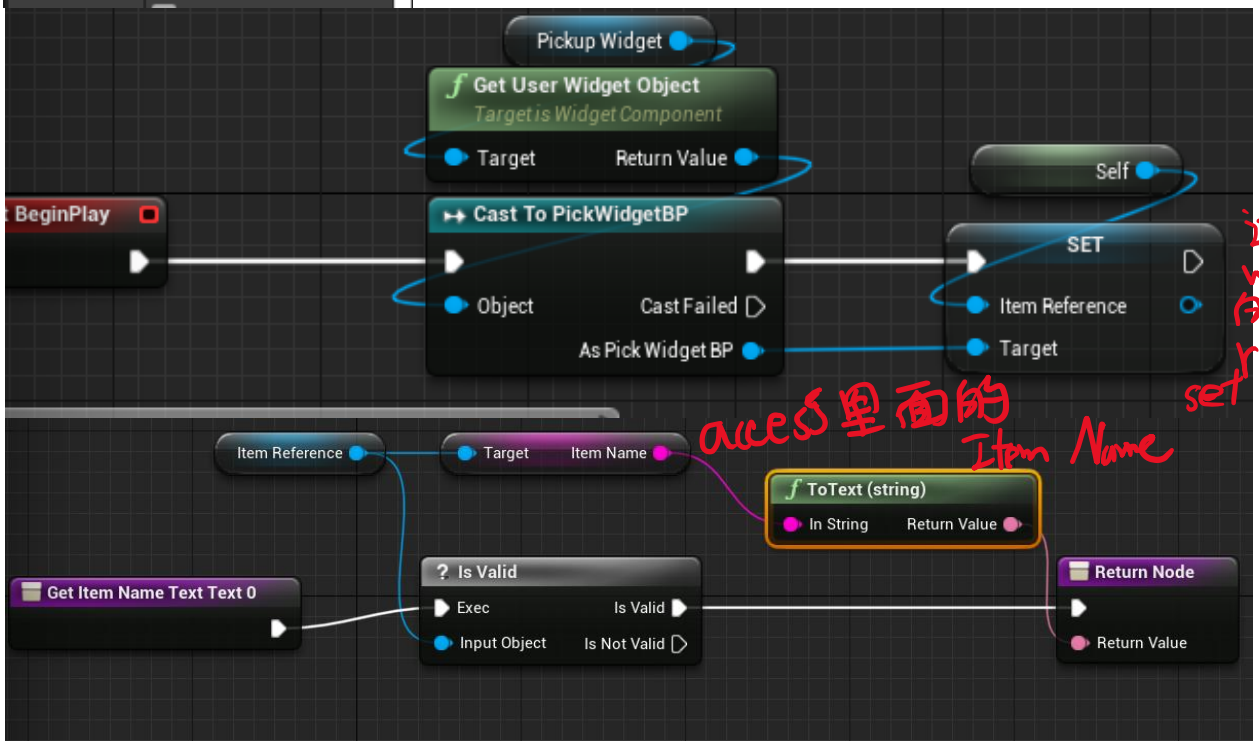


Create binding

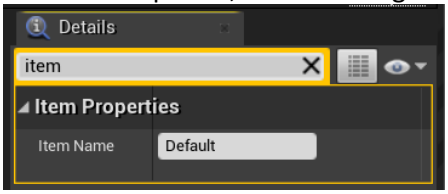


Create an item reference to get the item and to access the item name inside the item reference

-in base weapon bp

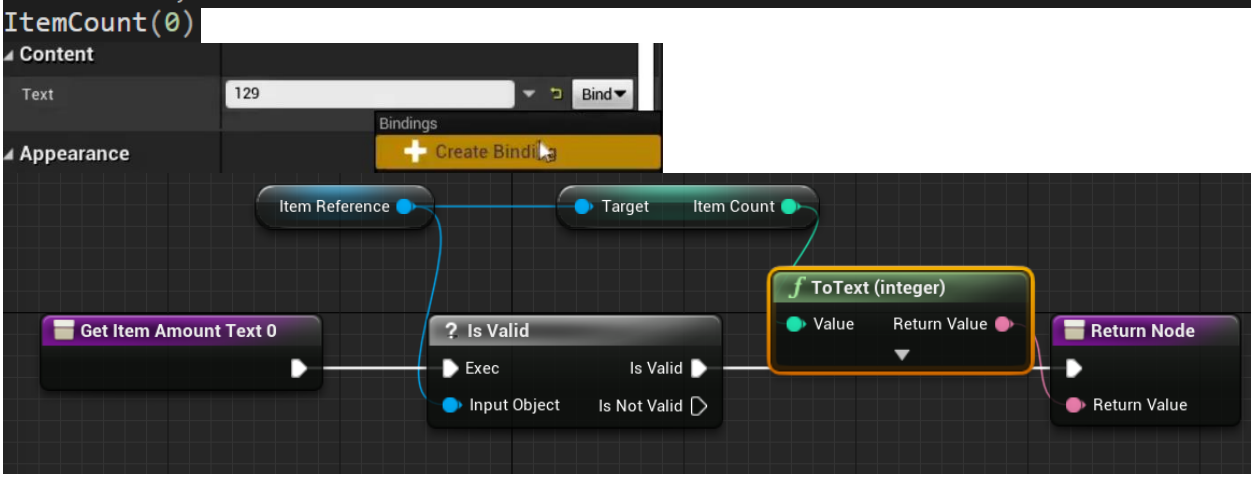


-in base weaponBP, we can change the item name cuz it's inherited from item



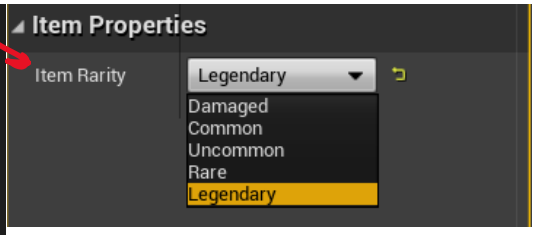
-Amount

```
// Item count(ammo,etc.)
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Item Properties", meta = (AllowPrivateAccess = "true"))
int32 ItemCount;
```



-Rarity

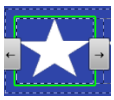
```
UENUM(BlueprintType)
enum class EItemRarity : uint8
{
    EIR_Damaged UMETA(DisplayName = "Damaged"),
    EIR_Common UMETA(DisplayName = "Common"),
    EIR_Uncommon UMETA(DisplayName = "Uncommon"),
    EIR_Rare UMETA(DisplayName = "Rare"),
    EIR_Legendary UMETA(DisplayName = "Legendary"),

    EIR_MAX UMETA(DisplayName = "DefultMAX")
};
```

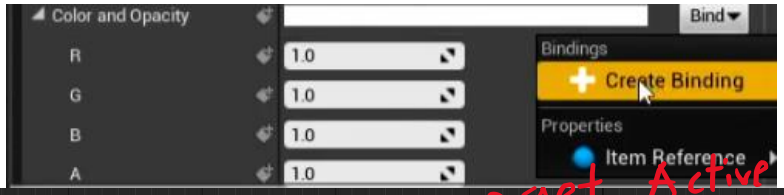



```
// Item rarity - determines number of stars in Pickup Widget
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Item Properties", meta = (AllowPrivateAccess = "true"))
EItemRarity ItemRarity;

UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Item Properties", meta = (AllowPrivateAccess = "true
TArray<bool> ActiveStars;
```

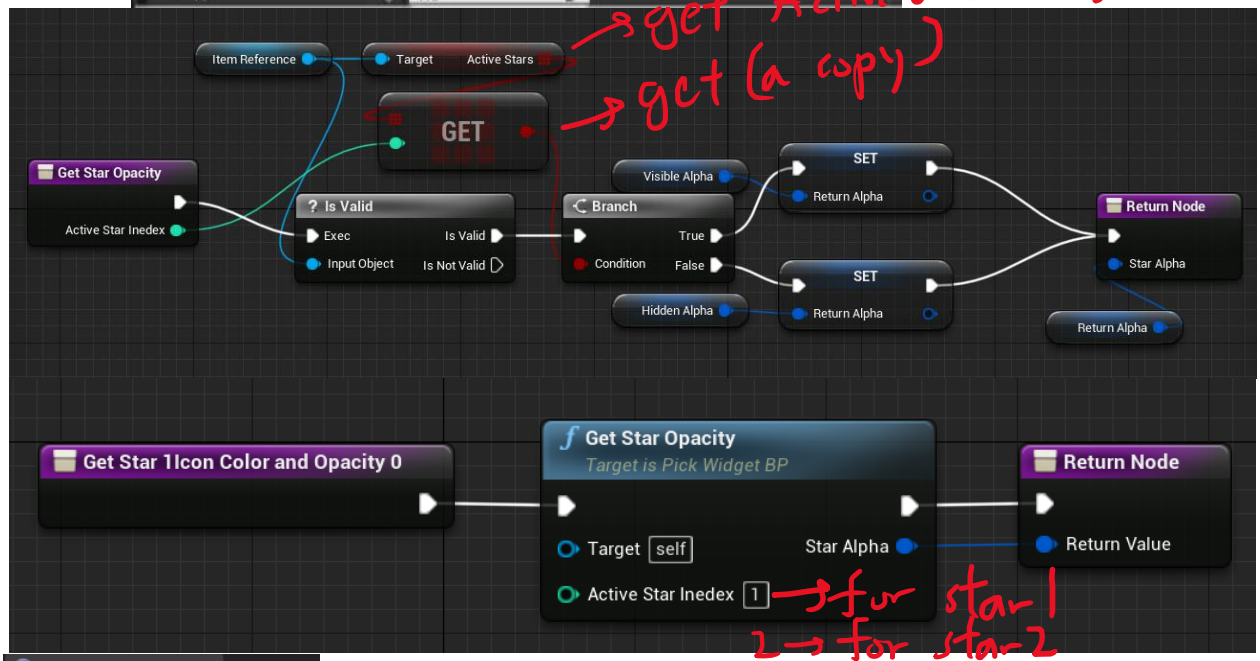
```
void AItem::SetActiveStars()
{
    // The 0 element isn't used
    for (int32 i = 0; i <= 5; i++)
    {
        ActiveStars.Add(false);
    }

    switch (ItemRarity)
    {
    case EItemRarity::EIR_Damaged:
        ActiveStars[1] = true;
        break;
    case EItemRarity::EIR_Common:
        ActiveStars[1] = true;
        ActiveStars[2] = true;
        break;
    case EItemRarity::EIR_Uncommon:
        ActiveStars[1] = true;
        ActiveStars[2] = true;
        ActiveStars[3] = true;
        break;
    case EItemRarity::EIR_Rare:
        ActiveStars[1] = true;
        ActiveStars[2] = true;
        ActiveStars[3] = true;
        ActiveStars[4] = true;
        break;
    case EItemRarity::EIR_Legendary:
        ActiveStars[1] = true;
        ActiveStars[2] = true;
        ActiveStars[3] = true;
        ActiveStars[4] = true;
        ActiveStars[5] = true;
        break;
    }
}
```

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

    // Hide Pickup Widget
    if (PickupWidget)
    {
    PickupWidget->SetVisibility(false);
    }
    // Set ActiveStars array based on Item Rarity
    SetActiveStars();
```



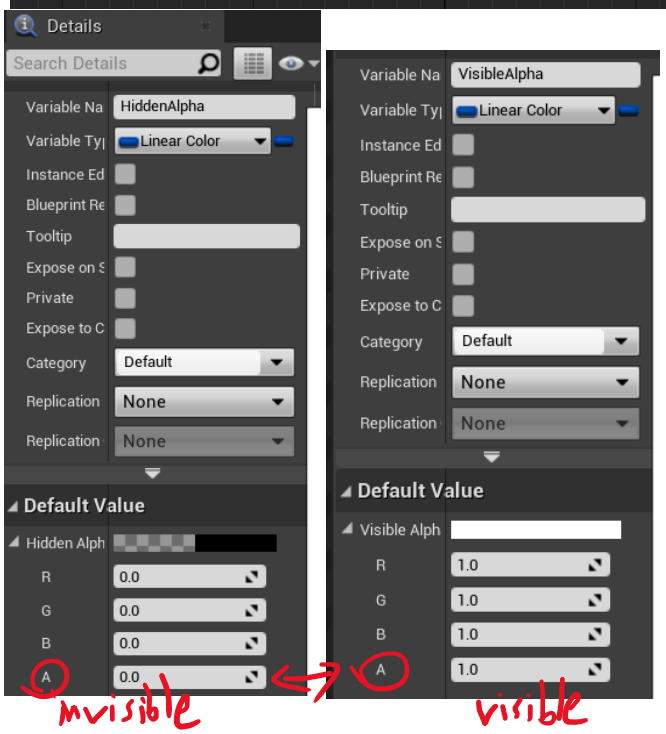


get Active Stars Array



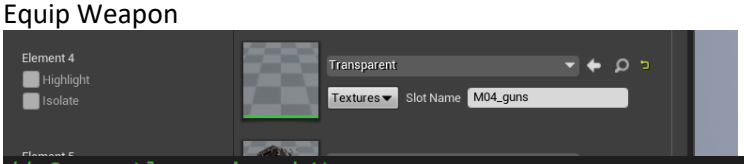
get (a copy)

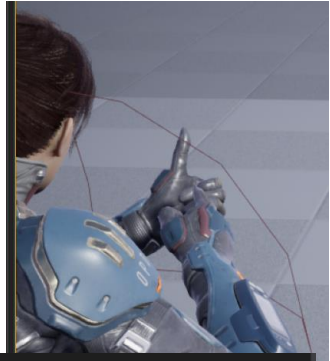
1 → for star 1
2 → for star 2
so on and so forth



invisible ⟷ visible

Equip Weapon

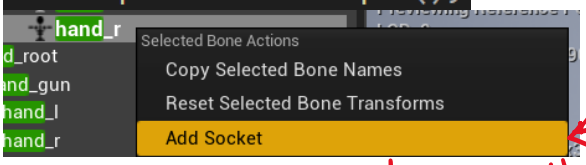




```
// Currently equipped Weapon
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = Combat, meta = (AllowPrivateAccess = "true"))
class AWeapon* EquippedWeapon;
// Set this in Blueprints for the default Weapon class
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category = Combat, meta = (AllowPrivateAccess = "true"))
TSubclassOf<AWeapon> DefaultWeaponClass;
```

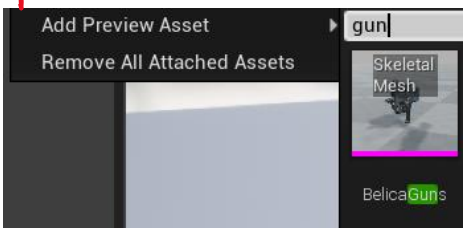
TSubclassOf<AWeapon>可以让这个viariables 拥有blueprint（aka UClass）

```
void SpawnDefaultWeapon();
```

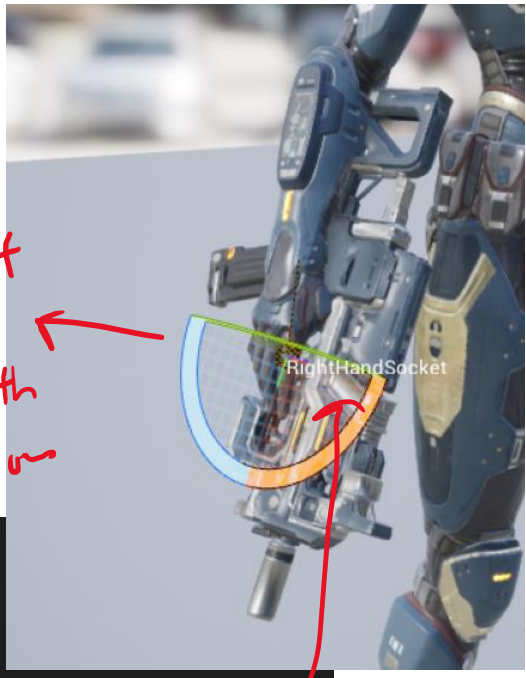


Add a socket on hand

preview the mesh on the socket



position the socket currently so the weapon will spawn with correct position




```
void AShooterCharacter::SpawnDefaultWeapon()
{
    // Check the TSubclassOf variable
    if (DefaultWeaponClass)
    {
        // Spawn the Weapon
        AWeapon* DefaultWeapon = GetWorld()->SpawnActor<AWeapon>(DefaultWeaponClass);
        // Get the Hand Socket
        const USkeletalMeshSocket* HandSocket = GetMesh()->GetSocketByName(FName("RightHandSocket"));

        if (HandSocket)
        {
            // Attach the Weapon to the hand socket RightHandSocket
            HandSocket->AttachActor(DefaultWeapon, GetMesh());
        }

        // Set EquippedWeapon to the newly spawned Weapon
        EquippedWeapon = DefaultWeapon;
    }
}
void AShooterCharacter::BeginPlay()
{
    Super::BeginPlay();

    // Spawn the default weapon and attach it to the mesh
    SpawnDefaultWeapon();
```

```
// Takes a weapon and attaches it to the mesh
void EquipWeapon(AWeapon* WeaponToEquip);
```

```
#include "Components/BoxComponent.h"
#include "Components/SphereComponent.h"
```

```
void AShooterCharacter::EquipWeapon(AWeapon* WeaponToEquip)
{
    if (WeaponToEquip)
    {
        // Set AreaSphere to ignore all Collision Channels
        WeaponToEquip->GetAreaSphere()->SetCollisionResponseToAllChannels(ECollisionResponse::ECR_Ignore);
        // Set CollisionBox to ignore all Collision Channels
        WeaponToEquip->GetCollisionBox()->SetCollisionResponseToAllChannels(ECollisionResponse::ECR_Ignore);

        // Get the Hand Socket
        const USkeletalMeshSocket* HandSocket = GetMesh()->GetSocketByName(FName("RightHandSocket"));
        if (HandSocket)
        {
            // Attach the Weapon to the hand socket RightHandSocket
            HandSocket->AttachActor(WeaponToEquip, GetMesh());
        }
        // Set EquippedWeapon to the newly spawned Weapon
        EquippedWeapon = WeaponToEquip;
    }
}
```

取消 collision

this is from the old SpawnDefaultWeapon

```
AWeapon* AShooterCharacter::SpawnDefaultWeapon()
{
    // Check the TSubclassOf variable
    if (DefaultWeaponClass)
    {
        // Spawn and return the Weapon
        return GetWorld()->SpawnActor<AWeapon>(DefaultWeaponClass);
    }
    return nullptr;
}
```

← 改成只create一个weapon

改成在Begin Play equip SpawnDefaultWeapon return的weapon object

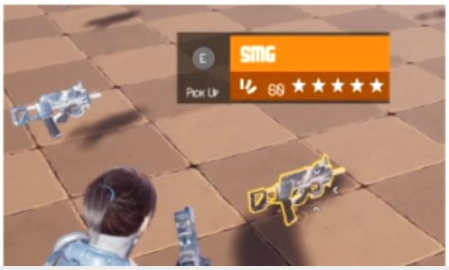
```
// Spawn the default weapon and equip it
EquipWeapon(SpawnDefaultWeapon());
```

Item States - Equipped

- Held in hand
- Attached to Socket
- Collision turned off
- EquippedWeapon variable holds a reference to it

there will only be one EquippedWeapon

Item States - Pickup



- Sitting on the ground
- AreaSphere and Collision Box are active
- Not attached to anything

Item States - EquipInterping

- We are picking up the Item
- It flies up to our face!
- Collision should be off

Item States - PickedUp

- We are holding the item, but it's not equipped
- Sitting in the inventory
- Collision should be off
- Visibility should be off

Item States - Falling

- We are dropping the item
- Hasn't yet hit the ground
- Should block the floor
- Area Sphere and Collision Box should be off

-in item.h

```
UENUM(BlueprintType)
enum class EItemState : uint8
{
    EIS_Pickup UMETA(DisplayName = "Pickup"),
    EIS_EquipInterping UMETA(DisplayName = "EquipInterping"),
    EIS_PickedUp UMETA(DisplayName = "PickedUp"),
    EIS_Equipped UMETA(DisplayName = "Equipped"),
    EIS_Falling UMETA(DisplayName = "Falling"),
    EIS_MAX UMETA(DisplayName = "DefultMAX")
};
// State of the Item
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Item Properties", meta = (AllowPrivateAccess = "tru
EItemState ItemState;
```

`ItemState(EItemState::EIS_Pickup)` in constructor, set default value

```
FORCEINLINE EItemState GetItemState() const { return ItemState; }
FORCEINLINE void SetItemState(EItemState State) { ItemState = State; }
```

-in shooterCharacter.cpp, equipweapon function

```
// Set EquippedWeapon to the newly spawned Weapon
EquippedWeapon = WeaponToEquip;
WeaponToEquip->SetItemState(EItemState::EIS_Equipped);
```

```
void AItem::SetItemProperties(EItemState State)
{
    switch (State)
    {
case EItemState::EIS_Pickup:
    // Set Mesh properties
    ItemMesh->SetSimulatePhysics(false);
    ItemMesh->SetEnableGravity(false);
    ItemMesh->SetVisibility(true);
    ItemMesh->SetCollisionResponseToAllChannels(ECollisionResponse::ECR_Ignore);
    ItemMesh->SetCollisionEnabled(ECollisionEnabled::NoCollision);
    // Set AreaPhere properties
    AreaSphere->SetCollisionResponseToAllChannels(ECollisionResponse::ECR_Overlap);
    AreaSphere->SetCollisionEnabled(ECollisionEnabled::QueryOnly);
    // Set CollisionBox properties
    CollisionBox->SetCollisionResponseToAllChannels(ECollisionResponse::ECR_Ignore);
    CollisionBox->SetCollisionResponseToChannel(ECollisionChannel::ECC_Visibility, ECollisionResponse::ECR_Block);
    CollisionBox->SetCollisionEnabled(ECollisionEnabled::QueryAndPhysics);
    break;
case EItemState::EIS_Equipped:
    PickupWidget->SetVisibility(false);
    // Set Mesh properties
    ItemMesh->SetSimulatePhysics(false);
    ItemMesh->SetEnableGravity(false);
    ItemMesh->SetVisibility(true);
    ItemMesh->SetCollisionResponseToAllChannels(ECollisionResponse::ECR_Ignore);
    ItemMesh->SetCollisionEnabled(ECollisionEnabled::NoCollision);
    // Set AreaPhere properties
    AreaSphere->SetCollisionResponseToAllChannels(ECollisionResponse::ECR_Ignore);
    AreaSphere->SetCollisionEnabled(ECollisionEnabled::NoCollision);
    // Set CollisionBox properties
    CollisionBox->SetCollisionResponseToAllChannels(ECollisionResponse::ECR_Ignore);
    CollisionBox->SetCollisionEnabled(ECollisionEnabled::NoCollision);
    break;
```

ignore all channel except for visibility, so it will block visibility

```cpp
case EItemState::EIS_Falling:
    // Set Mesh properties
    ItemMesh->SetSimulatePhysics(true);
    ItemMesh->SetVisibility(true);
    ItemMesh->SetEnableGravity(true);
    ItemMesh->SetCollisionResponseToAllChannels(ECollisionResponse::ECR_Ignore);
    ItemMesh->SetCollisionEnabled(ECollisionEnabled::QueryAndPhysics);
    ItemMesh->SetCollisionResponseToChannel(ECollisionChannel::ECC_WorldStatic, ECollisionResponse::ECR_Block);
    // Set AreaPhere properties
    AreaSphere->SetCollisionResponseToAllChannels(ECollisionResponse::ECR_Ignore);
    AreaSphere->SetCollisionEnabled(ECollisionEnabled::NoCollision);
    // Set CollisionBox properties
    CollisionBox->SetCollisionResponseToAllChannels(ECollisionResponse::ECR_Ignore);
    CollisionBox->SetCollisionEnabled(ECollisionEnabled::NoCollision);
    break;
}
```

-In begin play

```cpp
// Set Item properties based on Item State
SetItemProperties(ItemState);
```

-in shooterCharacter.cpp, remove all this cuz it's already set in the setItemProperties

```cpp
void AShooterCharacter::EquipWeapon(AWeapon* WeaponToEquip)
{
    if (WeaponToEquip)
    {
        // Set AreaSphere to ignore all Collision Channels
        WeaponToEquip->GetAreaSphere()->SetCollisionResponseToAllChannels(ECollisionResponse::ECR_Ignore);
        // Set CollisionBox to ignore all Collision Channels
        WeaponToEquip->GetCollisionBox()->SetCollisionResponseToAllChannels(ECollisionResponse::ECR_Ignore);
```

-in item.h

We can change the set item state function

```cpp
void SetItemState(EItemState State);
```

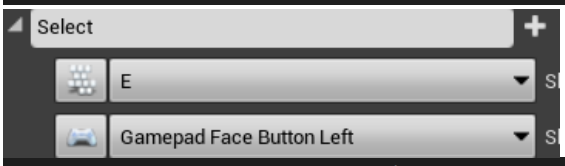
```cpp
void AItem::SetItemState(EItemState State)
{
    ItemState = State;
    SetItemProperties(State);
}
```

Drop weapon

-in item.h

```
FORCEINLINE USkeletalMeshComponent* GetItemMesh() const { return ItemMesh; }
```

```
void AShooterCharacter::DropWeapon()
{
    if (EquippedWeapon)
    {
        FDetachmentTransformRules DetachmentTransformRules(EDetachmentRule::KeepWorld, true);
        EquippedWeapon->GetItemMesh()->DetachFromComponent(DetachmentTransformRules);

        EquippedWeapon->SetItemState(EItemState::EIS_Falling);
    }
}
```



Action mapping

```
void SelectButtonPressed();
void SelectButtonReleased();
```

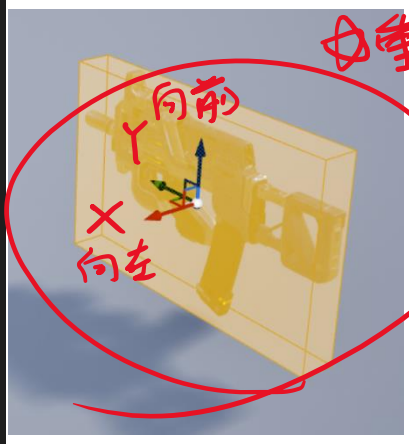
```
PlayerInputComponent->BindAction("Select", IE_Pressed, this, &AShooterCharacter::SelectButtonPressed);
PlayerInputComponent->BindAction("Select", IE_Released, this, &AShooterCharacter::SelectButtonReleased);
```

```
void AShooterCharacter::SelectButtonPressed()
{
    DropWeapon();
}
```

Throw weapon

-in weapon.h

```h
class SHOOTER_API AWeapon : public AItem
{
    GENERATED_BODY()

public:
    AWeapon();

    virtual void Tick(float DeltaTime) override;
protected:
    void StopFalling();
private:
    FTimerHandle ThrowWeaponTimer;
    float ThrowWeaponTime;
    bool bFalling;
public:
    // Add an impulse to the weapon
    void ThrowWeapon();

};
```



z

```cpp
void AWeapon::ThrowWeapon()
{
    FRotator MeshRotation{ 0.f, GetItemMesh()->GetComponentRotation().Yaw, 0.f };
    GetItemMesh()->SetWorldRotation(MeshRotation, false, nullptr, ETeleportType::TeleportPhysics);

    const FVector MeshForward{ GetItemMesh()->GetForwardVector() };
    const FVector MeshRight{ GetItemMesh()->GetRightVector() };
    // Direction in which we throw the Weapon
    FVector ImpulseDirection = MeshRight.RotateAngleAxis(-20.f, MeshForward);

    float RandomRotation{ FMath::FRandRange(0.f,30.f) };
    ImpulseDirection = ImpulseDirection.RotateAngleAxis(RandomRotation, FVector(0.f, 0.f, 1.f));
    ImpulseDirection *= 20'000.f;
    GetItemMesh()->AddImpulse(ImpulseDirection);

    bFalling = true;
    GetWorldTimerManager().SetTimer(ThrowWeaponTimer, this, &AWeapon::StopFalling, ThrowWeaponTime);
}
```

```cpp
AWeapon::AWeapon():
    ThrowWeaponTime(1.5f),
    bFalling(false)
{
    PrimaryActorTick.bCanEverTick = true;
}

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

    // Keep the Weapon upright
    /*if (GetItemState() == EItemState::EIS_Falling && bFalling)
    {
        const FRotator MeshRotation{ 0.f, GetItemMesh()->GetComponentRotation().Yaw, 0.f };
        GetItemMesh()->SetWorldRotation(MeshRotation, false, nullptr, ETeleportType::TeleportPhysics);
    }*/
}
```

```cpp
void AWeapon::StopFalling()
{
    bFalling = false;
    SetItemState(EItemState::EIS_Pickup);
}
```

Swap Weapon
-in shooterCharacter.h

```cpp
//The item currently hit by our trace in TraceForItem (could be null)
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = Combat, meta = (AllowPrivateAccess = "true"))
AItem* TraceHitItem;
// Drop currently equipped Weapon and Equips TraceHitItem
void SwapWeapon(AWeapon* WeaponToSwap);
    if (TraceHitItem)
    {
        auto TraceHitWeapon = Cast<AWeapon>(TraceHitItem);
        SwapWeapon(TraceHitWeapon);
    }

}

void AShooterCharacter::SelectButtonReleased()
{

}

void AShooterCharacter::SwapWeapon(AWeapon* WeaponToSwap)
{
    DropWeapon();
    EquipWeapon(WeaponToSwap);
    TraceHitItem = nullptr;
    TraceHitItemLastFrame = nullptr;
}
void AShooterCharacter::TraceForItems()
{
    if (bShouldTraceForItems)
    {
        FHitResult ItemTraceResult;
        FVector HitLocation;
        TraceUnderCrosshairs(ItemTraceResult, HitLocation);
        if (ItemTraceResult.bBlockingHit)
        {
            TraceHitItem = Cast<AItem>(ItemTraceResult.Actor);
            if (TraceHitItem && TraceHitItem->GetPickupWidget())
            {
                // Show Item's Pickup Widget
                TraceHitItem->GetPickupWidget()->SetVisibility(true);
            }

            // We hit an AItem last frame
            if (TraceHitItemLastFrame)
            {
                if (TraceHitItem != TraceHitItemLastFrame)
                {
                    // We are hitting a different AItem this frame from last frame
                    // Or AItem is null
                    TraceHitItemLastFrame->GetPickupWidget()->SetVisibility(false);
                }
            }

            // Store a reference to HitItem for next frame
            TraceHitItemLastFrame = TraceHitItem;
        }
    }
```