1) Character Control (Karakterin hareket etmesi için)

2) Grab (Herhangi bir nesneyi tutmak için)

3) BlueprintPure (BP’ye akatarma yaparken kullanmak için)

4) BlueprintImplementableEvent (BP’ye aktarma yaparken kullanmak için)

5) Movable Platform (Herhangi bir nesneyi hareket ettirmek için ve bulmacada kullanmak için)

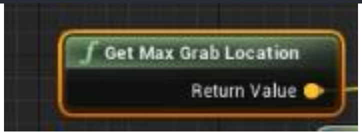
6) Spawner (Bir nesneyi belirli aralıkarla yaratmak için)

7) Project Tile (Tank mermisi gibi tek parça atılan mermiler için)

8) Gun (FPS oyunlarında olduğu gibi silah sistemini kullanmak için)

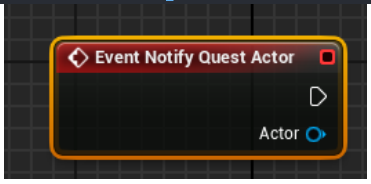
Blueprint pure: No node function in bp

```
UFUNCTION(BlueprintCallable, BlueprintPure)
FVector GetMaxGrabLocation() const;
```



BlueprintImplementableEvent: It create custom function.

```
UFUNCTION(BlueprintImplementableEvent)
void NotifyQuestActor(AActor* Actor);
```



Character Control

Character oluşturduktan sonra .h file dosyasına:

```
class UGrabber;
protected:
    UFUNCTION(BlueprintImplementableEvent, BlueprintPure)
    UGrabber* GetGrabber() const;
Public:
UGrabber* GetGrabber = nullptr;

private:
    void Forward(float AxisValue);
    void Right(float AxisValue);
    void Grab();
    void Release();
```

ekledikten sonra .cpp dosyasına:

```
#include "Components/InputComponent.h"
#include "Grabber.h"
#include "GameFramework/CharacterMovementComponent.h"

// Called to bind functionality to input
void AFirstPersonCharacter::SetupPlayerInputComponent(UInputComponent* PlayerInputComponent)
{
    Super::SetupPlayerInputComponent(PlayerInputComponent);
// key mapping all functions.
    PlayerInputComponent->BindAxis(TEXT("Forward"), this, &AFirstPersonCharacter::Forward);
    PlayerInputComponent->BindAxis(TEXT("Right"), this, &AFirstPersonCharacter::Right);
    PlayerInputComponent->BindAxis(TEXT("LookUp"), this, &APawn::AddControllerPitchInput);
    PlayerInputComponent->BindAxis(TEXT("LookRight"), this, &APawn::AddControllerYawInput);
    PlayerInputComponent->BindAction(TEXT("Jump"), EInputEvent::IE_Pressed, this, &ACharacter
::Jump);
    PlayerInputComponent->BindAction(TEXT("Grab"), EInputEvent::IE_Pressed, this, &AFirstPers
onCharacter::Grab);
    PlayerInputComponent->BindAction(TEXT("Grab"), EInputEvent::IE_Released, this, &AFirstPer
sonCharacter::Release);
}

void AFirstPersonCharacter::Forward(float AxisValue)
{
// movement for forward or backward.
    GetCharacterMovement()->AddInputVector(GetActorForwardVector() * AxisValue);
}

void AFirstPersonCharacter::Right(float AxisValue)
{
// movement for right or left.
    GetCharacterMovement()->AddInputVector(GetActorRightVector() * AxisValue);
}

void AFirstPersonCharacter::Grab()
{
// calling grab component in grab function.
    GetGrabber->Grab();
}

void AFirstPersonCharacter::Release()
{
// calling grab component in release function.
    GetGrabber->Release();
}
```

Grab

Scene Component oluşturuyoruz ve eğer bir objenin taşınabilir olmasını istiyorsak objeyi moveable, simulate physics true olarak işaretliyoruz.

Grabber.h:

```
#include "CoreMinimal.h"

#include "FirstPersonCharacter.h"
#include "Components/SceneComponent.h"
#include "PhysicsEngine/PhysicsHandleComponent.h"
#include "Grabber.generated.h"

public:
   // Sets default values for this component's properties
   UGrabber();

   // Called every frame
   virtual void TickComponent(float DeltaTime, ELevelTick TickType,
                                FActorComponentTickFunction* ThisTickFunction) override;

   void Grab();
   void Release();

private:
   void Grabbed();

   FVector GetMaxGrabLocation() const;
   FVector GetHoldLocation() const;

   UFUNCTION(BlueprintCallable, BlueprintPure)
   UPhysicsHandleComponent* GetPhysicsComponent() const;
   FHitResult GetFirstPhysicsBodyInReach() const;

   UPROPERTY(EditAnywhere, BlueprintReadOnly, meta = (AllowPrivateAccess = "true"))
   float MaxGrabDistance = 100;

   UPROPERTY(EditAnywhere, BlueprintReadOnly, meta = (AllowPrivateAccess = "true"))
   float HoldDistance = 100;

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

   UFUNCTION(BlueprintCallable, BlueprintImplementableEvent)
   void NotifyQuestActor(AActor* Actor);

   AFirstPersonCharacter* FirstPersonCharacter = nullptr;
```

Grabber.cpp:

```
#include "Grabber.h"
#include "FirstPersonCharacter.h"
#include "Engine/World.h"
#include "GameFramework/Actor.h"

#define OUT

// Sets default values for this component's properties
UGrabber::UGrabber()
{
   // Set this component to be initialized when the game starts, and to be ticked every frame.
You can turn these features
   // off to improve performance if you don't need them.
   PrimaryComponentTick.bCanEverTick = true;
}

// Called when the game starts
void UGrabber::BeginPlay()
{
   Super::BeginPlay();

   FirstPersonCharacter = Cast<AFirstPersonCharacter>(GetOwner());
   if (!FirstPersonCharacter) { return; }
   FirstPersonCharacter->GetGrabber = this;
   GetPhysicsComponent();
}

// Called every frame
void UGrabber::TickComponent(float DeltaTime, ELevelTick TickType,
FActorComponentTickFunction* ThisTickFunction)
{
   Super::TickComponent(DeltaTime, TickType, ThisTickFunction);
   Grabbed();
}

FVector UGrabber::GetMaxGrabLocation() const
{
   FVector PlayerViewPointLocation;
   FRotator PlayerViewPointRotation;

   GetWorld()->GetFirstPlayerController()->GetPlayerViewPoint(
      OUT PlayerViewPointLocation,
      OUT PlayerViewPointRotation
   );

   return PlayerViewPointLocation + PlayerViewPointRotation.Vector() * MaxGrabDistance;
}

FVector UGrabber::GetHoldLocation() const
{
   FVector PlayerViewPointLocation;
   FRotator PlayerViewPointRotation;

   GetWorld()->GetFirstPlayerController()->GetPlayerViewPoint(
      OUT PlayerViewPointLocation,
      OUT PlayerViewPointRotation
   );

   return PlayerViewPointLocation + PlayerViewPointRotation.Vector() * HoldDistance;
}

UPhysicsHandleComponent* UGrabber::GetPhysicsComponent() const
{
   return GetOwner()->FindComponentByClass<UPhysicsHandleComponent>();
}

void UGrabber::Grab()
{
   FHitResult HitResult = GetFirstPhysicsBodyInReach();
   AActor* HitActor = HitResult.GetActor();
   UPrimitiveComponent* HitComponent = HitResult.GetComponent();
   if (HitActor && HitComponent)
   {
      if (!GetPhysicsComponent()) { return; }
      //HitComponent->SetSimulatePhysics(true);
      GetPhysicsComponent()->GrabComponentAtLocationWithRotation(
         HitComponent,
         NAME_None,
         HitComponent->GetCenterOfMass(),
         FRotator()
      );
      NotifyQuestActor(HitActor);
   }
}

void UGrabber::Grabbed()
{
   if (!GetPhysicsComponent()) { return; }
   if (GetPhysicsComponent()->GrabbedComponent)
   {
      GetPhysicsComponent()->SetTargetLocation(GetMaxGrabLocation());
   }
}
```

```
void UGrabber::Release()
{
   if (!GetPhysicsComponent()) { return; }
   GetPhysicsComponent()->ReleaseComponent();
}

FHitResult UGrabber::GetFirstPhysicsBodyInReach() const
{
   FHitResult Hit;
   FCollisionQueryParams TraceParams(FName(TEXT("")), false, GetOwner());

   GetWorld()->LineTraceSingleByObjectType(
      OUT Hit,
      GetHoldLocation(),
      GetMaxGrabLocation(),
      FCollisionObjectQueryParams(ECollisionChannel::ECC_PhysicsBody),
      TraceParams
   );
   return Hit;
}
```

Movable Platform

İlk önce hangi nesnenin hareket edeceğini belirlemek için actor component oluşturup onu neseneye ekliyoruz. Hemen ardından nesnenin parenti olacak trigger volume ekliyoruz eğer platform harekete başladığında ses gelmesini istiyorsak, nesnenin detaylar panelinden add component diyerek audio component ekliyoruz. Audio component de detaylar kısmından auto active özelliğini kapatıyoruz (oyun başlar başlamaz çalmaması için).

ActorMovement.h:

```
#include "Components/ActorComponent.h"
#include "Engine/TriggerVolume.h"

UENUM()
enum EPlatformDirection
{
   DirectionX UMETA(Display = "X direction"),
   DirectionY UMETA(Display = "Y direction"),
   DirectionZ UMETA(Display = "Z direction")
};


class CONTRA_API UActorMovement : public UActorComponent
{
   GENERATED_BODY()

private:
   UPROPERTY(EditAnywhere)
   ATriggerVolume* PressPlate = nullptr;
   UPROPERTY(EditAnywhere)
   UAudioComponent* AudioComponent = nullptr;
   UPROPERTY(EditAnywhere)
   AActor* ActorThatOpen;

   UPROPERTY(EditAnywhere, BlueprintReadOnly, Category= "Direction", meta =
(AllowPrivateAccess = "true"))
   TEnumAsByte<EPlatformDirection> Direction;
   float DirectionValue;
   FVector Backward;
   FVector Forward;

   float TransporterForwardLast = 0.0f;
   UPROPERTY(EditAnywhere)
   float TransporterForwardSpeed = 200.0f;
   UPROPERTY(EditAnywhere)
   float TransportDelay = 5.0f;
   float TransporterInitial;
   float TransporterCurrent;
   UPROPERTY(EditAnywhere)
   float TransporterTarget = 3000;

public:
   // Sets default values for this component's properties
   UActorMovement();
   // Called every frame
   virtual void TickComponent(float DeltaTime, ELevelTick TickType,
                              FActorComponentTickFunction* ThisTickFunction) override;

   void FindPressPlate();
   void FindAudioComponent();
   void DirectionChoice();
   void ForwardTransporter(float DeltaTime);
   void BackwardTransporter(float DeltaTime);
   void TransporterLogic(float DeltaTime);

   bool ForwardSound = false;
   bool BackwardSound = true;
```

Transfer.cpp:

```
#include "ActorMovement.h"
#include "Components/AudioComponent.h"
#include "Engine/World.h"
#include "GameFramework/Actor.h"
#include "GameFramework/PlayerController.h"

#define OUT

// Called when the game starts
void UActorMovement::BeginPlay()
{
   Super::BeginPlay();
   // Getting your direction choice.
   DirectionChoice();
   // Setting initial value for target.
   TransporterInitial = DirectionValue;
   TransporterCurrent = TransporterInitial;
   TransporterTarget = TransporterInitial + TransporterTarget;
   // for the null pointer
   FindPressPlate();
   FindAudioComponent();
   // it is answering to "who can start this movement ?"
   ActorThatOpen = GetWorld()->GetFirstPlayerController()->GetPawn();
}

// Called every frame
void UActorMovement::TickComponent(float DeltaTime, ELevelTick TickType,
FActorComponentTickFunction* ThisTickFunction)
{
   Super::TickComponent(DeltaTime, TickType, ThisTickFunction);

   TransporterLogic(DeltaTime);
```

```
}

void UActorMovement::DirectionChoice()
{
   // Getting your choice and setting value for target.
   switch (Direction)
   {
   case DirectionX:
      {
         DirectionValue = GetOwner()->GetActorLocation().X;
         //Forward
         Forward = GetOwner()->GetActorLocation();
         Forward.X = TransporterCurrent;
         //Backward
         Backward = GetOwner()->GetActorLocation();
         Backward.X = TransporterCurrent;
         break;
      }
   case DirectionY:
      {
         DirectionValue = GetOwner()->GetActorLocation().Y;
         //Forward
         Forward = GetOwner()->GetActorLocation();
         Forward.Y = TransporterCurrent;
         //Backward
         Backward = GetOwner()->GetActorLocation();
         Backward.Y = TransporterCurrent;
         break;
      }
   case DirectionZ:
      {
         DirectionValue = GetOwner()->GetActorLocation().Z;
         //Forward
         Forward = GetOwner()->GetActorLocation();
         Forward.Z = TransporterCurrent;
         //Backward
         Backward = GetOwner()->GetActorLocation();
         Backward.Z = TransporterCurrent;
         break;
      }
   default:
      DirectionValue = GetOwner()->GetActorLocation().Z;
      //Forward
      Forward = GetOwner()->GetActorLocation();
      Forward.Z = TransporterCurrent;
      //Backward
      Backward = GetOwner()->GetActorLocation();
      Backward.Z = TransporterCurrent;
      break;
   }
}

void UActorMovement::TransporterLogic(float DeltaTime)
{
   if (PressPlate && PressPlate->IsOverlappingActor(ActorThatOpen))
   {
      if (TransporterCurrent != TransporterTarget)
      {
         // when it was moving, it is updating location.
         ForwardTransporter(DeltaTime);
         // it is updating last value to return.
         TransporterForwardLast = GetWorld()->GetTimeSeconds();
      }
```

```cpp
	}
	else if (PressPlate && !PressPlate->IsOverlappingActor(ActorThatOpen))
	{
		// returns after delay
		if (GetWorld()->GetTimeSeconds() - TransporterForwardLast > TransportDelay)
		{
			BackwardTransporter(DeltaTime);
		}
	}
}

void UActorMovement::ForwardTransporter(float DeltaTime)
{
	// Calculating movement speed with direction.
	TransporterCurrent = FMath::FInterpConstantTo(TransporterCurrent, TransporterTarget, 
DeltaTime,
	                                              TransporterForwardSpeed);
	DirectionChoice();
	GetOwner()->SetActorLocation(Forward);

	BackwardSound = false;
	// to play sound while moving
	if (!AudioComponent) { return; }
	if (!ForwardSound)
	{
		AudioComponent->Play();
		ForwardSound = true;
	}
}

void UActorMovement::BackwardTransporter(float DeltaTime)
{
	// Calculating movement speed with direction.
	TransporterCurrent = FMath::FInterpConstantTo(TransporterCurrent, TransporterInitial,
	                                              DeltaTime,
	                                              TransporterForwardSpeed);
	DirectionChoice();
	GetOwner()->SetActorLocation(Backward);
	// to play sound while moving
	ForwardSound = false;
	if (!AudioComponent) { return; }
	if (!BackwardSound)
	{
		AudioComponent->Play();
		BackwardSound = true;
	}
}

void UActorMovement::FindAudioComponent()
{
	// it is picking to audio.
	// Note: Don't forget add audio component on details.
	AudioComponent = GetOwner()->FindComponentByClass<UAudioComponent>();
	// for the null pointer.
	if (!AudioComponent)
	{
		UE_LOG(LogTemp, Warning, TEXT("%s Audio Component bulunamadı"), 
*GetOwner()->GetName());
	}
}

void UActorMovement::FindPressPlate()
{
    // for the null pointer.
    if (!PressPlate)
    {
        UE_LOG(LogTemp, Warning, TEXT("PressPlate not found!!"));
    }
}
}
```

Spawner

Spawner adında bir actor oluşturuyoruz.

Spawner.h:

```
private:
    void Spawn();
    void CalculateSpawn();

    FTimerHandle SpawnTimer;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Spawn Settings", meta =
(AllowPrivateAccess = "true"))
    ;
    FVector SpawnerLocation;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Spawn Settings", meta =
(AllowPrivateAccess = "true"))
    ;
    FRotator SpawnerRotation;


    // Spawn Object
    UPROPERTY(EditAnywhere, Category = "Spawner Settings")
    TSubclassOf<AActor> ActorToSpawn;

    // Spawner Settings
    UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Spawner Settings", meta =
(AllowPrivateAccess = "true"))
    float SpawnSpeed = 0.5f;
    UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Spawner Settings", meta =
(AllowPrivateAccess = "true"))
    float SpawnRepeat = 1.0f;
    // Spawn Location and Rotation
    UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Spawner Settings", meta =
(AllowPrivateAccess = "true"))
    float SpawnMinZ = -30.0f;
    UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Spawner Settings", meta =
(AllowPrivateAccess = "true"))
    float SpawnMaxZ = 300.0f;
```

Spawner.cpp

```
#include "Engine/World.h"
#include "GameFramework/Actor.h"
#include "TimerManager.h"

void ASpawner::BeginPlay()
{
   Super::BeginPlay();
   // Settings for spawn time etc.
   GetWorldTimerManager().SetTimer(SpawnTimer, this, &ASpawner::Spawn, SpawnRepeat, true,
SpawnSpeed);
}

void ASpawner::Spawn()
{
   CalculateSpawn();
   /*
   if you want to spawn other cpp file and reach value use this:
   APipeActor* SpawnedActor = GetWorld()->SpawnActor<APipeActor>(ActorToSpawn,
SpawnerLocation, SpawnerRotation);
   if (SpawnedActor)
   {
      SpawnedActor->Spawner = this;
   }
   */
   GetWorld()->SpawnActor<AActor>(ActorToSpawn, SpawnerLocation, SpawnerRotation);
}

void ASpawner::CalculateSpawn()
{
   // it is spawning between minZ - maxZ value.
   UE_LOG(LogTemp, Warning, TEXT("Spawned"));
   SpawnerLocation = GetActorLocation();
   SpawnerLocation.Z = GetActorLocation().Z - FMath::RandRange(SpawnMinZ, SpawnMaxZ);
   SpawnerRotation = GetActorRotation();
}
```

Project Tile

Projecttile için ilk önce actor oluşturuyoruz. Aşşağıdaki gibi project tile oluşturduktan sonra bunu silaha veya karaktere ekliyoruz.

ProjecttileBase.h

```
class UProjectileMovementComponent;
private:
    // COMPONENTS
    UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Components", meta = (AllowPriva
teAccess = "true"))
    UProjectileMovementComponent* ProjectileMovement = nullptr;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Move", meta = (AllowPrivateAccess
 = "true"))
    float MovementSpeed = 1300.0f;

    UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Components", meta = (AllowPriva
teAccess = "true"))
    UStaticMeshComponent* ProjectileMesh = nullptr;
    UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Components", meta = (AllowPrivateAccess = "true"))
    UParticleSystemComponent* ParticleTrail = nullptr;

    // VARIABLES
    UPROPERTY(EditDefaultsOnly, Category = "Damage")
    TSubclassOf<UDamageType> DamageType;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Damage", meta = (AllowPrivateAccess = "true"))
    float Damage = 50.0f;
    // Projecttile effect(behind smoke)
    UPROPERTY(EditAnywhere, Category= "Effect")
    UParticleSystem* HitParticle = nullptr;
    // Audio
    UPROPERTY(EditAnywhere, Category = "Effect")
    USoundBase* HitSound;
    UPROPERTY(EditAnywhere, Category = "Effect")
    USoundBase* LaunchSound;
    UPROPERTY(EditAnywhere, Category = "Effects")
    TSubclassOf<UCameraShake> HitShake;

    // FUNCTION
    UFUNCTION()
    void OnHit(UPrimitiveComponent* HitComponent, AActor* OtherActor, UPrimitiveComponent* OtherComponent, FVector NormalImpulse, const FHitResult& Hit);
```

Projecttile.cpp

```
#include "ProjectileBase.h"
#include "Components/StaticMeshComponent.h"
#include "GameFramework/ProjectileMovementComponent.h"
#include "Kismet/GameplayStatics.h"
#include "Particles/ParticleSystemComponent.h"

// Sets default values
AProjectileBase::AProjectileBase()
{
    PrimaryActorTick.bCanEverTick = false;

    ProjectileMesh = CreateDefaultSubobject<UStaticMeshComponent>(TEXT("Projectile Mesh"));
    ProjectileMesh->OnComponentHit.AddDynamic(this, &AProjectileBase::OnHit);
    RootComponent = ProjectileMesh;

    ProjectileMovement = CreateDefaultSubobject<UProjectileMovementComponent>(TEXT("Projectile Movement"));
    ProjectileMovement->InitialSpeed = MovementSpeed;
    ProjectileMovement->MaxSpeed = MovementSpeed;

    ParticleTrail = CreateDefaultSubobject<UParticleSystemComponent>(TEXT("Particle Trail"));
    ParticleTrail->SetupAttachment(RootComponent);

    InitialLifeSpan = 3.0f;
}

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

    UGameplayStatics::PlaySoundAtLocation(this,LaunchSound,GetActorLocation());

}


void AProjectileBase::OnHit(UPrimitiveComponent* HitComponent, AActor* OtherActor, UPrimitive
Component* OtherComponent, FVector NormalImpulse, const FHitResult& Hit)
{
    AActor* MyOwner = GetOwner();
    if (!MyOwner) { return; }

    if (OtherActor && OtherActor != this && OtherActor != MyOwner)
    {
        UGameplayStatics::ApplyDamage(
            OtherActor,
            Damage,
            MyOwner->GetInstigatorController(),
            this,
            DamageType
            );
        // When hitted object, It will create hitparticle and audio.
        UGameplayStatics::SpawnEmitterAtLocation(this, HitParticle, GetActorLocation());
        UGameplayStatics::PlaySoundAtLocation(this, HitSound, GetActorLocation());
        GetWorld()->GetFirstPlayerController()->ClientPlayCameraShake(HitShake);
        Destroy();
    }
}
```

Gun

İlk önce actor oluşturuyoruz ve bunu ana karakterimize ekliyoruz.

Gun.h

```
public:
    void PullTrigger();

    UPROPERTY(EditAnywhere, Category = "Attack", meta = (AllowPrivateAccess = "true"))
        int DefaultAmmo;
    int Ammo;
    UPROPERTY(EditAnywhere, Category = "Attack", meta = (AllowPrivateAccess = "true"))
        float FireRate = 0.5f;
    UPROPERTY(EditAnywhere, Category = "Attack", meta = (AllowPrivateAccess = "true"))
        float FireRepeat = 1.0f;

private:
    UPROPERTY(VisibleAnywhere)
        USceneComponent* Root;
    UPROPERTY(VisibleAnywhere)
        USkeletalMeshComponent* Mesh;
    UPROPERTY(EditAnywhere, Category = "Effect", meta = (AllowPrivateAccess = "true"))
        UParticleSystem* MuzzleFlash;
    UPROPERTY(EditAnywhere, Category = "Effect", meta = (AllowPrivateAccess = "true"))
        UParticleSystem* ImpactEffect;
    UPROPERTY(EditAnywhere, Category = "Effect", meta = (AllowPrivateAccess = "true"))
        USoundBase* MuzzleSound;
    UPROPERTY(EditAnywhere, Category = "Effect", meta = (AllowPrivateAccess = "true"))
        USoundBase* ImpactSound;
    UPROPERTY(EditAnywhere, Category = "Attack", meta = (AllowPrivateAccess = "true"))
        float MaxRange = 1000;
    UPROPERTY(EditAnywhere)
        float Damage = 10;

    bool GunTrace(FHitResult &Hit, FVector &ShotDirection);

    AController* GetOwnerController() const;
```

Gun.cpp

```
// Fill out your copyright notice in the Description page of Project Settings.


#include "Gun.h"
#include "Components/SkeletalMeshComponent.h"
#include "Kismet/GameplayStatics.h"
#include "DrawDebugHelpers.h"

// Sets default values
AGun::AGun()
{
    // Set this actor to call Tick() every frame.  You can turn this off to improve performan
ce if you don't need it.
    PrimaryActorTick.bCanEverTick = true;

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

    Mesh = CreateDefaultSubobject<USkeletalMeshComponent>(TEXT("Mesh"));
    Mesh->SetupAttachment(Root);
```

```
}

void AGun::PullTrigger()
{
    // Effect Spawn
    UGameplayStatics::SpawnEmitterAttached(MuzzleFlash, Mesh, TEXT("MuzzleFlashSocket"));
    UGameplayStatics::SpawnSoundAttached(MuzzleSound, Mesh, TEXT("MuzzleFlashSocket"));

    FHitResult Hit;
    FVector ShotDirection;
    bool bSuccess = GunTrace(Hit, ShotDirection);
    // if distance possible create particle and LineTraceSingle at location
    if (bSuccess)
    {
        UGameplayStatics::SpawnEmitterAtLocation(
            GetWorld(),
            ImpactEffect,
            Hit.Location,
            ShotDirection.Rotation());
        UGameplayStatics::SpawnSoundAtLocation(
            GetWorld(),
            ImpactSound,
            Hit.Location,
            ShotDirection.Rotation()
        );

        AActor* HitActor = Hit.GetActor();
        if (HitActor != nullptr)
        {
            FPointDamageEvent DamageEvent(Damage, Hit, ShotDirection, nullptr);
            AController* OwnerController = GetOwnerController();
            HitActor->TakeDamage(Damage, DamageEvent, OwnerController, this);
        }
    }
}


bool AGun::GunTrace(FHitResult& Hit, FVector& ShotDirection)
{
    AController* OwnerController = GetOwnerController();
    if (!OwnerController) { return false; }

    FVector Location;
    FRotator Rotation;

    OwnerController->GetPlayerViewPoint(Location, Rotation);
    ShotDirection = -Rotation.Vector();
    // We are calculating PlayerViewPoint between Wall distance
    FVector End = Location + Rotation.Vector() * MaxRange;
    // We are ignoring own character
    FCollisionQueryParams Params;
    Params.AddIgnoredActor(this);
    Params.AddIgnoredActor(GetOwner());
    return GetWorld()->LineTraceSingleByChannel(
        Hit, Location,
        End,
        ECollisionChannel::ECC_GameTraceChannel1,
        Params);
}

AController* AGun::GetOwnerController() const
{
    APawn* OwnerPawn = Cast<APawn>(GetOwner());
    if (OwnerPawn == nullptr) { return nullptr; }

    return OwnerPawn->GetController();
}
```