Lütfen değişiklik yapmak için veya başka yerlerde paylaşmak için iletişime geçiniz. Sorularını veya geliştirmelerini varsa duymaktan memnun olurum.

Github: https://github.com/creosB

Linkedin: https://www.linkedin.com/in/bariseroglu/
Bunları ve daha fazlasını yaptığımız kanal: https://www.twitch.tv/creosb

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) (basit)

7) Spawner v2 (gelişmiş)

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

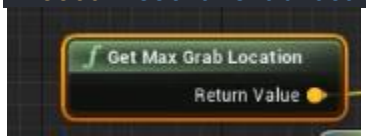
9) Obstacle (Spawner v2 için kullandığım niagara efektli engel)

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

11) Health Box (Alındığında can veren veya silah veren kutular)

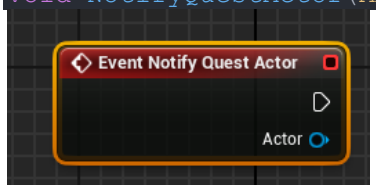
Blueprint pure: No node function in bp

```
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());
    }
}
```

```cpp
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:

```cpp
#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();
      }
```

```
    }
    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

```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

```cpp
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);
```

```cpp
}

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();
}
```

Spawner V2

Pawn oluşturduktan sonra istediğimiz yere spawner'ı sürükleyip bırakıyoruz. Detaylar kısmından istediğimiz örüntüyü oluşturuyoruz bu sayede sürekli olarak o örüntüyü devam ettiriyor.

Özellik:

- Seçtiğiniz yönde(x,y veya z) ve belirlediğiniz aralıkta rastgele oluşturuyor.

- Belirttiğiniz sıklıkta oluşturur.

- Her üyenin farklı şekilde özelleştirilmesini sağlar.

- Trigger'da iseniz belirlediğiniz sırayı takip ederek oluşturmaya devam eder.

- Niagara efektine çarparsanız sağlığınızı düşürür.

Yöntem:

+ Her üyeyi özelleştirmek için structure kullandım.

+ Yön seçmek için enum kullandım.

+ Spawner'ı parent, oluşturulan nesneyi child olarak ayarladım, böylece iletişim kurmasını sağladım.

Örnek kullanımı ve görünüşü:
www.reddit.com/r/unrealengine/comments/o4kfmd/i_tried_to_niagara_electric_ball_with_cpp_if_you/

SpawnerV2.h

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

#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Pawn.h"
#include "Engine/TriggerVolume.h"
#include "FireObstacleSpawner.generated.h"

// Direction Choice
UENUM(BlueprintType)
enum ESpawnDirection
{
    SpawnDirectionX UMETA(Display = "X direction"),
    SpawnDirectionY UMETA(Display = "Y direction"),
    SpawnDirectionZ UMETA(Display = "Z direction")
};

// Choice for every array elements.
USTRUCT(BlueprintType)
struct CONTRA_API FSpawnSettings
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere)
    TEnumAsByte<ESpawnDirection> SpawnDirection;
    // Min-Max value for Spawn Location and Rotation
    UPROPERTY(EditAnywhere)
    float MinSpawnLocation;
    UPROPERTY(EditAnywhere)
    float MaxSpawnLocation;
    // - or + Roll value
    UPROPERTY(EditAnywhere)
    float SpawnRotation;
};

class AContraCharacter;

UCLASS()
class CONTRA_API AFireObstacleSpawner : public APawn
{
    GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    AFireObstacleSpawner();

    virtual void Tick(float DeltaSeconds) override;
```

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

    virtual float TakeDamage(float DamageAmount, FDamageEvent const& DamageEvent, AController
* EventInstigator,
                             AActor* DamageCauser) override;

private:
    // Function
    void ObstacleSpawn(); // creating actor
    void SetSpawnTimer(); // calling timer
    void SpawnChoice(); // defines values.
    void BacktoNormal(); // Slow motion
    bool IsDead() const;

    // Spawner Main Settings
    UPROPERTY(EditDefaultsOnly)
    UStaticMeshComponent* SpawnerMesh = nullptr;
    // Cast Main Player
    AContraCharacter* ContraCharacter = nullptr;
    // Spawn Object
    UPROPERTY(EditAnywhere, Category = "Main Settings")
    TSubclassOf<AActor> ActorToSpawn;
    FTimerHandle SpawnTimer;
    // Info from struct
    UPROPERTY(EditAnywhere, meta = (TitleProperty = "Spawner Settings"))
    TArray<struct FSpawnSettings> SpawnSetting;
    // Spawner temp value for set the actor.
    FVector SpawnerLocation;
    FRotator SpawnerRotation;
    // Trigger Volume for the ActorMovement
    UPROPERTY(EditAnywhere, Category = "Spawner Settings")
    ATriggerVolume* Trigger = nullptr;
    // Spawner Settings
    UPROPERTY(EditAnywhere, Category = "Spawner Settings")
    float SpawnSpeed = 0.5f;
    UPROPERTY(EditAnywhere, Category = "Spawner Settings")
    float SpawnRepeat = 1.0f;
    UPROPERTY(EditAnywhere, Category = "Spawner Settings")
    float SpawnedMovementSpeed = 300.0f;
    UPROPERTY(EditAnywhere, Category = "Spawner Settings")
    float SpawnedDeadTime = 10.0f; // Spawned obstacle destroy time
    int i = 0; // array start value
    bool IsWork = true;

    // Slow Motion
    bool SlowLogic = false;
    FTimerHandle BacktoNormalTimer;
```

```
    float TimerValue = 0.01;
    // Health
    bool ReturnGameModeDead = true;
    UPROPERTY(EditDefaultsOnly)
    float MaxHealth = 100.0f;
    UPROPERTY(VisibleAnywhere)
    float Health;
    // Effect Choice
    UPROPERTY(EditAnywhere, Category = "Effects", meta = (AllowPrivateAccess = "true"))
    UParticleSystem* DeathParticle = nullptr;
    UPROPERTY(EditAnywhere, Category = "Effects")
    USoundBase* DeathSound;
    // for detect attached actor or static meshes
    TArray<AActor*> Attached;
    UPROPERTY(EditAnywhere)
    bool DestroyWAttachment = false;
};
```

SpawnverV2.cpp

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


#include "FireObstacleSpawner.h"
#include "Contra/ContraCharacter.h"
#include "FireObstacle.h"
#include "ActorMovement.h"
#include "ContraGameMode.h"
#include "Engine/World.h"
#include "TimerManager.h"
#include "Kismet/GameplayStatics.h"


// Sets default values
AFireObstacleSpawner::AFireObstacleSpawner()
{
    // 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;

    SpawnerMesh = CreateDefaultSubobject<UStaticMeshComponent>("Spawner Mesh");
    RootComponent = SpawnerMesh;
}

// Called when the game starts or when spawned
void AFireObstacleSpawner::BeginPlay()
{
    Super::BeginPlay();
    // Calling main player.
```

```
    ContraCharacter = Cast<AContraCharacter>(UGameplayStatics::GetPlayerCharacter(this, 0));
    Health = MaxHealth;
}

void AFireObstacleSpawner::Tick(float DeltaSeconds)
{
     // if player goes out of the trigger, timer will stop (spawnner).
    if (Trigger && Trigger->IsOverlappingActor(ContraCharacter) && IsWork)
    {
        SetSpawnTimer();
        IsWork = false;
    }
    else if (Trigger && !Trigger->IsOverlappingActor(ContraCharacter))
    {
        GetWorldTimerManager().PauseTimer(SpawnTimer);
        IsWork = true;
    }
}

// Call Function with Timer
void AFireObstacleSpawner::SetSpawnTimer()
{
    GetWorldTimerManager().UnPauseTimer(SpawnTimer);
    GetWorldTimerManager().SetTimer(SpawnTimer, this, &AFireObstacleSpawner::ObstacleSpawn, S
pawnRepeat, true,
                                    SpawnSpeed);
}

// Spawn Actor
void AFireObstacleSpawner::ObstacleSpawn()
{
    // Repeating array 0 to max and restarting.
    if (i < SpawnSetting.Num())
    {
        SpawnChoice();
        i++;
    }
    else
    {
        i = 0;
        SpawnChoice();
    }
    // It's giving information to child (SpawnedActor)
    AFireObstacle* SpawnedActor = GetWorld()->SpawnActor<AFireObstacle>(ActorToSpawn, Spawner
Location, SpawnerRotation);
    if (SpawnedActor && SpawnedActor->ActorMovement && Trigger)
    {
        SpawnedActor->DeadDelay = SpawnedDeadTime;
        SpawnedActor->ActorMovement->Direction = DirectionX;
```

```
        SpawnedActor->ActorMovement->PressPlate = Trigger;
        SpawnedActor->ActorMovement->TransporterForwardSpeed = SpawnedMovementSpeed;
        SpawnedActor->FireObstacleSpawner = this;
    }
}

// Set Properties
void AFireObstacleSpawner::SpawnChoice()
{
    // It's giving random value to each array elements and repeats every time we call this fu
nction.
    switch (SpawnSetting[i].SpawnDirection)
    {
    case SpawnDirectionZ:
        {
            SpawnerLocation = GetActorLocation();
            SpawnerLocation.Z = GetActorLocation().Z - FMath::RandRange(
                SpawnSetting[i].MinSpawnLocation,
                SpawnSetting[i].MaxSpawnLocation);
            SpawnerRotation = FRotator(0, 0, SpawnSetting[i].SpawnRotation);
            break;
        }
    case SpawnDirectionY:
        {
            SpawnerLocation = GetActorLocation();
            SpawnerLocation.Y = GetActorLocation().Y - FMath::RandRange(
                SpawnSetting[i].MinSpawnLocation,
                SpawnSetting[i].MaxSpawnLocation);
            SpawnerRotation = FRotator(0, 0, SpawnSetting[i].SpawnRotation);
            break;
        }
    case SpawnDirectionX:
        {
            SpawnerLocation = GetActorLocation();
            SpawnerLocation.X = GetActorLocation().X - FMath::RandRange(
                SpawnSetting[i].MinSpawnLocation,
                SpawnSetting[i].MaxSpawnLocation);
            SpawnerRotation = FRotator(0, 0, SpawnSetting[i].SpawnRotation);
            break;
        }
    }
}

float AFireObstacleSpawner::TakeDamage(float DamageAmount, FDamageEvent const& DamageEvent,
                                       AController* EventInstigator, AActor* DamageCauser)
{
    // pawn taking damage and applying health.
    float DamageToApply = Super::TakeDamage(DamageAmount, DamageEvent, EventInstigator, Damag
eCauser);
    DamageToApply = FMath::Min(Health, DamageToApply);
```

```
    Health -= DamageToApply;

    // when health was less than 10, it will set start slow motion value and call function.
    if (Health < 10)
    {
        SlowLogic = true;
        UGameplayStatics::SetGlobalTimeDilation(GetWorld(), TimerValue);
        BacktoNormal();
    }
    // when player was dead, it will spawn particle and sound.
    if (IsDead())
    {
        UGameplayStatics::SpawnEmitterAtLocation(this, DeathParticle, GetActorLocation());
        UGameplayStatics::SpawnSoundAtLocation(this, DeathSound, GetActorLocation());
        // pawn is giving information to gamemode.
        AContraGameMode* GameMode = GetWorld()->GetAuthGameMode<AContraGameMode>();
        if (GameMode != nullptr && ReturnGameModeDead)
        {
            GameMode->PawnKilled(this);
            // returning game mode one! It's protecting sum more score.
            ReturnGameModeDead = false;
        }
        if (Controller != nullptr && Controller->GetPawn() == this)
        {
            Controller->PawnPendingDestroy(this);
            if (Controller != nullptr)
            {
                Controller->UnPossess();
                Controller = nullptr;
            }
        }

        //Destroy();
        //  GetCapsuleComponent()->SetCollisionEnabled(ECollisionEnabled::NoCollision);
    }
    return DamageToApply;
}
// Check Health less than 0
bool AFireObstacleSpawner::IsDead() const
{
    return Health <= 0;
}
// Slow Motion with Destroy
void AFireObstacleSpawner::BacktoNormal()
{
    // setting timer slow motion
    GetWorldTimerManager().SetTimer(BacktoNormalTimer, this, &AFireObstacleSpawner::BacktoNor
mal, 2, true, 0.1f);
    if (SlowLogic)
```

```
    {
        // when player was dead and slow time is enough, it will back to the normal and destr
oy himself.
        if (TimerValue >= 0.5f && Health <= 0)
        {
            TimerValue = 1.0f;
            UGameplayStatics::SetGlobalTimeDilation(this, TimerValue);
            GetWorldTimerManager().ClearTimer(BacktoNormalTimer);
            GetWorldTimerManager().ClearTimer(SpawnTimer);
            // Detect attached actors or static meshes and it will destroy with herself
            if (DestroyWAttachment)
            {
                GetAttachedActors(Attached);
                for(int a = 0; a < Attached.Num(); a++)
                {
                    Attached[a]->Destroy();
                }
            }
            Destroy();
        }
        else
        {
            // if player doesn't dead and slow motion enough, slow time returning normal but
slowly.
            TimerValue += 0.1;
            UGameplayStatics::SetGlobalTimeDilation(this, TimerValue);
        }
    }
}
```

Obstacle

Spawner v2'de istediğimiz nesneyi yaratmak için kullandığımız nesne olması için aktör oluşturuyoruz ve istediğimiz herhangi bir niagara efekti ekliyoruz.

- Karakterin engel arasında olup olmadığını anlamak için line trace kullandım.
- Yaratıldıktan istediğimiz kadar süre sonra yok olması için timer içerisinde DeadDelay kullandım.
- 2 engel arasına gelindiğinde karakterin canını azaltır.
- Görünüşü ve örnek kullanımı: www.reddit.com/r/unrealengine/comments/o97swk/random_obstacle_spawner_with_niagara_effect/

Obstacle.h

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

#pragma once

#include "CoreMinimal.h"
#include "FireObstacleSpawner.h"
#include "GameFramework/Actor.h"
#include "NiagaraComponent.h"
#include "FireObstacle.generated.h"

class UActorMovement;
class AContraCharacter;

UCLASS()
class CONTRA_API AFireObstacle : public AActor
{
    GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    AFireObstacle();

    virtual void Tick(float DeltaSeconds) override;

    // Cast Spawner
    UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "Components")
    AFireObstacleSpawner* FireObstacleSpawner = nullptr;

    // Movement
    UPROPERTY(EditAnywhere)
    UActorMovement* ActorMovement = nullptr;
    // Dead Time
    UPROPERTY(EditAnywhere)
    float DeadDelay = 10.0f;

private:
    void ObstacleHit(); // When hit the obstacle
    void EffectHitStart(); // when It spawned
    void ObstacleDestroy();

    // Main Settings
    USceneComponent* SceneComponent = nullptr;
    UPROPERTY(EditDefaultsOnly, Category= "Effect")
    UStaticMeshComponent* EffectStart = nullptr; // Effect start location
    UPROPERTY(EditDefaultsOnly, Category= "Effect")
    UStaticMeshComponent* EffectEnd = nullptr; // Effect end location
    // Niagara Effect
    UPROPERTY(EditDefaultsOnly, Category= "Effect")
```

```
    UNiagaraComponent* FireEffect = nullptr; // Effect type
    FName EffectName = "User.BeamEnd"; // Effect name
    // Cast Main Player
    AContraCharacter* ContraCharacter = nullptr;

    FTimerHandle DeadTimer;

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

Obstacle.cpp

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


#include "FireObstacle.h"
#include "Components/SceneComponent.h"
#include "Contra/ContraCharacter.h"
#include "Kismet/GameplayStatics.h"
#include "Contra/ActorMovement.h"
#include "Contra/FireObstacleSpawner.h"
#include "Engine/Public/TimerManager.h"


// Sets default values
AFireObstacle::AFireObstacle()
{
    // 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;

    SceneComponent = CreateDefaultSubobject<UStaticMeshComponent>("Obstacle Mesh");
    RootComponent = SceneComponent;

    EffectStart = CreateDefaultSubobject<UStaticMeshComponent>("Effect Start");
    EffectStart->SetupAttachment(SceneComponent);
    EffectEnd = CreateDefaultSubobject<UStaticMeshComponent>("Effect End");
    EffectEnd->SetupAttachment(SceneComponent);

    FireEffect = CreateDefaultSubobject<UNiagaraComponent>("Fire Effect");
    FireEffect->SetupAttachment(EffectEnd);
    // Actor Movement settings on details page.
    ActorMovement = CreateDefaultSubobject<UActorMovement>("FireObstacle Movement");
}

// Called when the game starts or when spawned
void AFireObstacle::BeginPlay()
```

```cpp
{
    Super::BeginPlay();
    // Calling main player.
    ContraCharacter = Cast<AContraCharacter>(UGameplayStatics::GetPlayerCharacter(this, 0));
    FireEffect->SetVectorParameter(EffectName, EffectStart->GetComponentLocation());
    GetWorldTimerManager().SetTimer(DeadTimer, this, &AFireObstacle::ObstacleDestroy, 1.0f, f
alse,
                                    DeadDelay);
}

void AFireObstacle::Tick(float DeltaSeconds)
{
    EffectHitStart();
}

// It is creating LineTrace
void AFireObstacle::EffectHitStart()
{
    // Setting effect location every frame because actor is moving.
    FireEffect->SetVectorParameter(EffectName, EffectStart->GetComponentLocation());

    FHitResult Hit;
    FCollisionQueryParams TraceParams;
    // sending line trace between end-start
    GetWorld()->LineTraceSingleByObjectType(
        OUT Hit,
        EffectStart->GetComponentLocation(),
        EffectEnd->GetComponentLocation(),
        ECC_Pawn,
        TraceParams
    );

    AActor* ActorHit = Hit.GetActor();
    // when hit the pawn and if pawn tag is player, returning true.
    if (ActorHit && ActorHit->ActorHasTag("Player"))
    {
        ObstacleHit();
    }
}

// When hit the obstacle.
void AFireObstacle::ObstacleHit()
{
    // if It called this function, decreasing health value.
    if (ContraCharacter)
    {
        ContraCharacter->Health -= 0.5f;
    }
}
```

```
void AFireObstacle::ObstacleDestroy()
{
    GetWorldTimerManager().ClearTimer(DeadTimer);
    Destroy();
}
```

Health Box

İlk önce aktör oluşturuyoruz ardından detaylar kısmından verilecek can miktarını ve alındığında çıkacak olan sesi seçiyoruz.

- Alındığında can veren, silah veren veya diğer kutuların hepsi bu mantıkladır.

HealthBox.h

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

#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "HealthBox.generated.h"

UCLASS()
class CONTRA_API AHealthBox : public AActor
{
    GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    AHealthBox();

protected:
    // Called when the game starts or when spawned
    virtual void BeginPlay() override;
    UFUNCTION()
    void OnHit(UPrimitiveComponent* HitComponent, AActor* OtherActor, UPrimitiveComponent* Ot
herComponent,
               FVector NormalImpulse, const FHitResult& Hit);

private:
    void FindPointer();
    UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category= "Components",meta = (AllowPrivate
Access = true))
    UStaticMeshComponent* HealthBoxMesh;
    // Effect System
    UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category= "Effects", meta = (AllowPrivateA
ccess = true))
    UParticleSystem* GiveHealthParticle = nullptr;
    UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category= "Effects", meta = (AllowPrivateA
ccess = true))
    USoundBase* GiveHealthSound = nullptr;
```

```
    // Healthbox giving this amounth health percent.
    UPROPERTY(EditAnywhere, BlueprintReadOnly, Category= "Components",meta = (AllowPrivateAcc
ess = true))
    float GiveHealthValue = 10.0f;

};
```

HealthBox.cpp

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


#include "HealthBox.h"
#include "Components/StaticMeshComponent.h"
#include "ContraCharacter.h"
#include "Kismet/GameplayStatics.h"


// Sets default values
AHealthBox::AHealthBox()
{
    // 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 = false;

    HealthBoxMesh = CreateDefaultSubobject<UStaticMeshComponent>(TEXT("HealthBox Mesh"));
    HealthBoxMesh->OnComponentHit.AddDynamic(this, &AHealthBox::OnHit);
    RootComponent = HealthBoxMesh;

}

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

// When hit the player on healthbox, it will give 10.0f health value. After taking health, it
 will destroy own.
void AHealthBox::OnHit(UPrimitiveComponent* HitComponent, AActor* OtherActor, UPrimitiveCompo
nent* OtherComponent,
            FVector NormalImpulse, const FHitResult& Hit)
{
    AContraCharacter* ContraCharacter = Cast<AContraCharacter>(OtherActor);
    if (!ContraCharacter) { return; }

    if(ContraCharacter && OtherActor != this)
    {
        ContraCharacter->Health += GiveHealthValue;
```

```
        FindPointer();
        UGameplayStatics::SpawnEmitterAtLocation(this,GiveHealthParticle,GetActorLocation());
        UGameplayStatics::SpawnSoundAtLocation(this, GiveHealthSound, GetActorLocation());
        Destroy();
    }
}

void AHealthBox::FindPointer()
{
    if (!GiveHealthSound) { return; }
    if (!GiveHealthParticle) { return; }
}
```