**Multiplayer Programming Quick Start**

<https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/Networking/QuickStart/>

## **Create a simple multiplayer game in C++.**

|  |
| --- |
| **Prerequisite Topics**  In order to understand and use the content on this page, make sure you are familiar with the following topics:   * [Client-Server Model](https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/Networking/Server) * [Networking Overview](https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/Networking/Overview) |

A person on a stage

Description automatically generated with low confidence

Developing gameplay for a multiplayer game requires you to implement **replication** in your game's **Actors**. You must also design functionality specific to the **server**, which acts as the host for the game session, or a **client**, which represents a player connecting to the session. In this step-by-step guide, we will walk you through the process of creating some simple multiplayer gameplay, and you will learn the following:

* How to add replication to a base Actor.
* How to take advantage of **Movement Components** in a network game.
* How to add replication to **variables**.
* How to use **RepNotifies** when a variable changes.
* How to use **Remote Procedure Calls (RPCs)** in C++.
* How to check an Actor's **Network Role** in order to filter calls that are performed within a function.

The end result will be a third-person game where players can throw exploding projectiles at one another. The bulk of the work we do will be creating the projectile and adding a damage response to the Character.

NOTE  
Before we begin, we highly recommend that you review the essentials in the [Client-Server Model](https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/Networking/Server) and [Networking Overview](https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/Networking/Overview) pages. As a point of comparison for this guide, you can refer to the Adding Projectiles to your Game section of the [First Person Shooter Tutorial](https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/ProgrammingWithCPP/CPPTutorials/FirstPersonShooter), which does not introduce replication concepts.

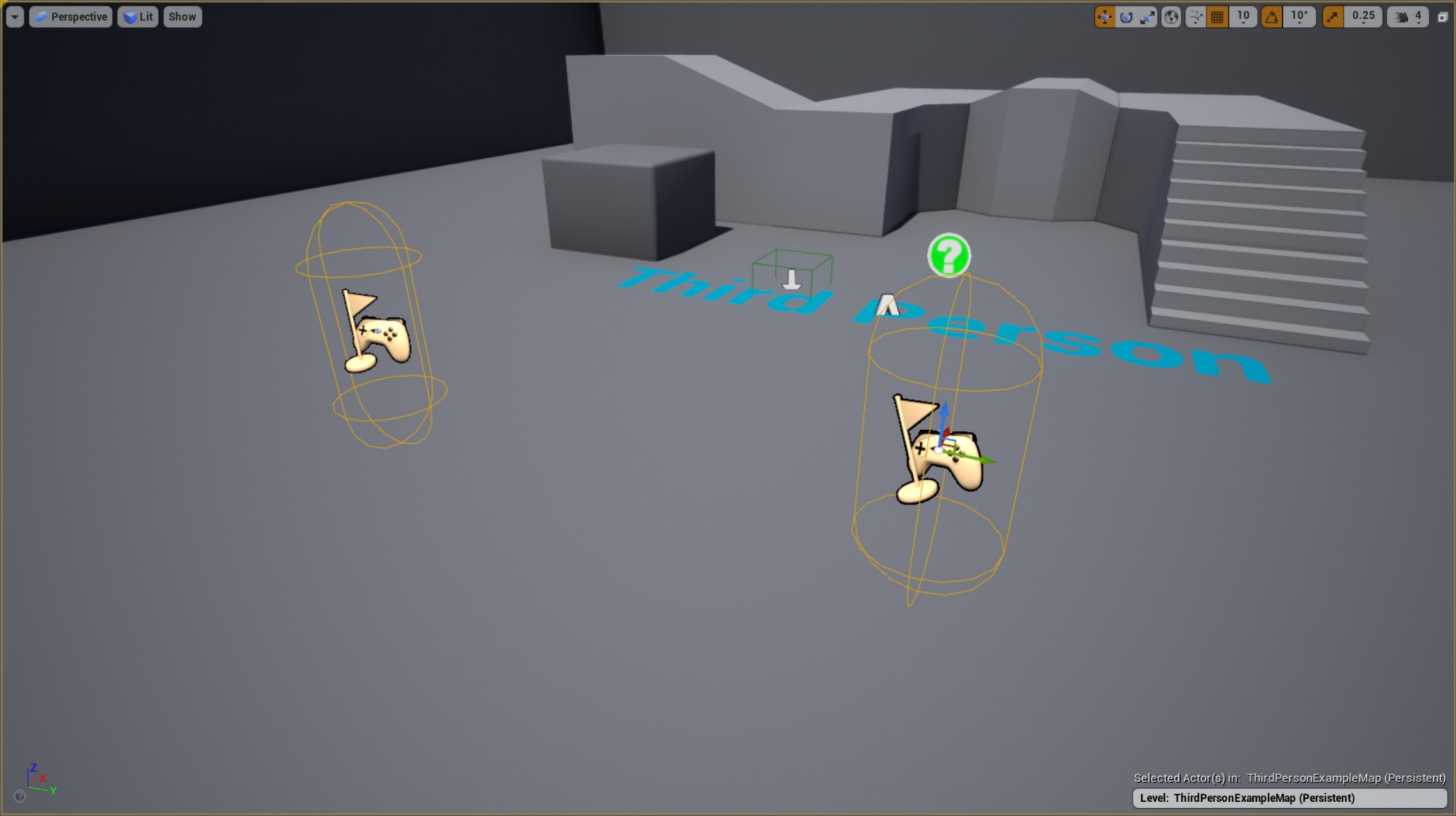
## **Essential Setup**

Open the **Editor** and create a **New Project**. Ensure that it has the following settings:

* Is a **C++ Project**
* Uses the **Third-Person Template**
* Includes **Starter Content**
* Targets **Console and PC**

Once you have applied these settings, name your project **ThirdPerson** and click the **Create** button to continue. The project's C++ files will be created, and the Unreal Editor will open **ThirdPersonExampleMap** automatically.

Click the **ThirdPersonCharacter** standing in this scene and **Delete** it, then ensure that there are two **Player Starts** are present in your map. These will handle spawning your players instead of the manually placed ThirdPersonCharacter that the scene includes by default.



*Click image for full size.*

The Pawns and Characters in most templates have replication enabled by default. In our example, ThirdPersonCharacter already has a **Character Movement Component** that will automatically replicate movement.

[Character Movement Component](https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/Networking/CharacterMovementComponent)

Cosmetic components like the Character's **Skeletal Mesh** and its **Animation Blueprint** are not replicated. However, variables that are relevant to gameplay and movement, like a Character's velocity, are replicated, and the Animation Blueprint reads these variables as they are updated. Each client's copies of the Characters will therefore update their visual representations in a way that is consistent provided that gameplay variables update accurately. Likewise, the [**Gameplay Framework**](https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/Framework) automatically handles spawning Characters at Player Starts and assigning **Player Controllers** to them.

If you start a server with this project and have a client join it you would already have a functioning multiplayer game. However, players would only be able to move and jump with their avatar. Therefore, we will create some additional multiplayer gameplay.

## **Replicating the Player's Health with RepNotifies**

Players need a health value so that we can cause damage to them during gameplay. That value needs to replicate so that all clients have synchronized information about each player's health, and we need to provide feedback to a player when they take damage. This section will demonstrate how it is possible to use a RepNotify to synchronize all essential updates to a variable without relying on RPCs.

NOTE  
Just a quick reminder that 'Role' has been replaced with 'GetLocalRole()' and 'GetRemoteRole()' respectively. You'll notice some sections below that might have previously used 'Role' so just be mindful of the change.

Open ThirdPersonCharacter.h. Add the following Properties under protected:

**ThirdPersonCharacter.h**

/\*\* The player's maximum health. This is the highest that their health can be, and the value that their health starts at when spawned.\*/

UPROPERTY(EditDefaultsOnly, Category = "Health")

**float** MaxHealth;

/\*\* The player's current health. When reduced to 0, they are considered dead.\*/

UPROPERTY(ReplicatedUsing=OnRep\_CurrentHealth)

**float** CurrentHealth;

/\*\* RepNotify for changes made to current health.\*/

UFUNCTION()

**void** OnRep\_CurrentHealth();

We want to strictly control how the player's health is changed therefore these health values have the following constraints:

* MaxHealth does not replicate and is only editable in defaults. This value is pre-computed for all players, and will never change.
* CurrentHealth replicates, but is not editable or accessible anywhere in Blueprint.
* Both MaxHealth and CurrentHealth are protected, which prevents them from being accessed from external C++ classes. They can only be modified within AThirdPersonMPCharacter or other classes derived from it.

This minimizes the risk of causing unwanted changes to a player's CurrentHealth or MaxHealth during live gameplay. We will provide other public functions for getting and modifying these values in a later step.

The Replicated specifier enables the copy of an Actor on the server to replicate the value of a variable to all connected clients any time it changes. ReplicatedUsing does the same thing, but enables us to set a **RepNotify** function that will be triggered when a client successfully receives the replicated data. We will use OnRep\_CurrentHealth to perform updates to each client based on changes to this variable.

Open ThirdPersonCharacter.cpp. Add the following #include statements at the top, underneath the line that reads #include "GameFramework/SpringArmComponent.h":

**ThirdPersonCharacter.cpp**

#include "Math/UnrealMathUtility.h"

#include "Net/UnrealNetwork.h"

#include "Engine/Engine.h"

These provide required functionality for variable replication as well as access to the AddOnscreenDebugMessage function in GEngine, which we will use to output messages to the screen.

In ThirdPersonCharacter.cpp, add the following code at the bottom of the constructor:

**ThirdPersonCharacter.cpp**

//Initialize the player's Health

MaxHealth = 100.0f;

CurrentHealth = MaxHealth;

These will initialize the player's health. Any time a new copy of this Character is created, its current health will be set to its maximum health value.

In ThirdPersonCharacter.h add the following public function declaration just after the AThirdPersonCharacter constructor:

**ThirdPersonCharacter.h**

/\*\* Property replication \*/

**void** GetLifetimeReplicatedProps(TArray<FLifetimeProperty>& OutLifetimeProps) **const** **override**;

In ThirdPersonCharacter.cpp, add the following implementation for this function:

**ThirdPersonCharacter.cpp**

//////////////////////////////////////////////////////////////////////////

// Replicated Properties

**void** AThirdPersonCharacter::GetLifetimeReplicatedProps(TArray <FLifetimeProperty> & OutLifetimeProps) **const**

{

Super::GetLifetimeReplicatedProps(OutLifetimeProps);

//Replicate current health.

DOREPLIFETIME(AThirdPersonMPCharacter, CurrentHealth);

}

The GetLifetimeReplicatedProps function is responsible for replicating any properties we designate with the Replicated specifier, and enables us to configure how a property will replicate. Here we are using the most basic implementation for CurrentHealth. If at any time you add more properties that need to be replicated you must add them to this function as well.

WARNING  
You must call the Super::GetLifetimeReplicatedProps or the inherited properties from your Actor's parent class will not replicate, even if the parent class designates them as being replicated.

In ThirdPersonCharacter.h add the following function declaration under Protected:

**ThirdPersonCharacter.h**

/\*\* Response to health being updated. Called on the server immediately after modification, and on clients in response to a RepNotify\*/

**void** OnHealthUpdate();

In ThirdPersonCharacter.cpp add the following implementation:

**ThirdPersonCharacter.cpp**

**void** AThirdPersonCharacter::OnHealthUpdate()

{

//Client-specific functionality

**if** (IsLocallyControlled())

{

FString healthMessage = FString::Printf(TEXT("You now have %f health remaining."), CurrentHealth);

GEngine->AddOnScreenDebugMessage(-1, 5.f, FColor::Blue, healthMessage);

**if** (CurrentHealth <= 0)

{

FString deathMessage = FString::Printf(TEXT("You have been killed."));

GEngine->AddOnScreenDebugMessage(-1, 5.f, FColor::Red, deathMessage);

}

}

//Server-specific functionality

**if** (GetLocalRole() == ROLE\_Authority)

{

FString healthMessage = FString::Printf(TEXT("%s now has %f health remaining."), \*GetFName().ToString(), CurrentHealth);

GEngine->AddOnScreenDebugMessage(-1, 5.f, FColor::Blue, healthMessage);

}

//Functions that occur on all machines.

/\*

Any special functionality that should occur as a result of damage or death should be placed here.

\*/

}

We will be using this function to perform updates in response to changes to the player's CurrentHealth. Currently its functionality is limited to onscreen debug messages, but additional functionality could be added, like an OnDeath function that is called on all machines in order to trigger a death animation. Note that OnHealthUpdate is not replicated, and we will need to manually call it on all devices.

In ThirdPersonCharacter.cpp add the following implementation for OnRep\_CurrentHealth:

**ThirdPersonCharacter.cpp**

**void** AThirdPersonCharacter::OnRep\_CurrentHealth()

{

OnHealthUpdate();

}

Variables replicate any time their value changes rather than constantly replicating, and RepNotifies run any time the client successfully receives a replicated value for a variable. Therefore, any time we change the player's CurrentHealth on the server, we would expect OnRep\_CurrentHealth to run on each connected client. This makes OnRep\_CurrentHealth the ideal place to call OnHealthUpdate on clients' machines.

## **Making the Player Respond to Damage**

Now that we have implemented the player's health we need to provide a means for modifying the player's health from outside of this class.

In ThirdPersonCharacter.h add the following function declarations under Public:

**ThirdPersonCharacter.h**

/\*\* Getter for Max Health.\*/

UFUNCTION(BlueprintPure, Category="Health")

FORCEINLINE **float** GetMaxHealth() **const** { **return** MaxHealth; }

/\*\* Getter for Current Health.\*/

UFUNCTION(BlueprintPure, Category="Health")

FORCEINLINE **float** GetCurrentHealth() **const** { **return** CurrentHealth; }

/\*\* Setter for Current Health. Clamps the value between 0 and MaxHealth and calls OnHealthUpdate. Should only be called on the server.\*/

UFUNCTION(BlueprintCallable, Category="Health")

**void** SetCurrentHealth(**float** healthValue);

/\*\* Event for taking damage. Overridden from APawn.\*/

UFUNCTION(BlueprintCallable, Category = "Health")

**float** TakeDamage( **float** DamageTaken, **struct** FDamageEvent **const**& DamageEvent, AController\* EventInstigator, AActor\* DamageCauser ) **override**;

The GetMaxHealth and GetCurrentHealth functions provide getters that can access the player's health values from outside of AThirdPersonMPCharacter, both in C++ and in Blueprint. As const functions they provide a safe means of getting these values without allowing them to be modified. We are also declaring functions for setting the player's health and taking damage.

In ThirdPersonCharacter.cpp add the following implementation for SetCurrentHealth:

**ThirdPersonCharacter.cpp**

**void** AThirdPersonCharacter::SetCurrentHealth(**float** healthValue)

{

**if** (GetLocalRole() == ROLE\_Authority)

{

CurrentHealth = FMath::Clamp(healthValue, 0.f, MaxHealth);

OnHealthUpdate();

}

}

SetCurrentHealth provides a controlled means of modifying the player's CurrentHealth from outside of AThirdPersonMPCharacter. It is not a replicated function, but by checking that the Network Role of the Actor is ROLE\_Authority, we restrict this function to execute only if it is called on the server that is hosting the game. It clamps CurrentHealth to values between 0 and the player's MaxHealth, making it impossible to set CurrentHealth to an invalid value, and it also calls OnHealthUpdate to ensure that the server and clients both have parallel calls to this function. This is necessary because the server will not receive the RepNotify.

TIP

While "setter" functions like this are not necessary for every variable, they are preferable for sensitive gameplay variables that change frequently during play, especially if they can be modified by many different sources. This is a best-practice for single-player and multiplayer games alike, as it makes live changes to these variables more consistent, easier to debug, and easier to extend with new functionality.

In ThirdPersonCharacter.cpp add the following implementation for TakeDamage:

**ThirdPersonCharacter.cpp**

**float** AThirdPersonCharacter::TakeDamage(**float** DamageTaken, **struct** FDamageEvent **const**& DamageEvent, AController\* EventInstigator, AActor\* DamageCauser)

{

**float** damageApplied = CurrentHealth - DamageTaken;

SetCurrentHealth(damageApplied);

**return** damageApplied;

}

The built-in functions for applying damage to Actors call the basic TakeDamage function for that Actor. In this case we implement a simple health deduction using SetCurrentHealth.

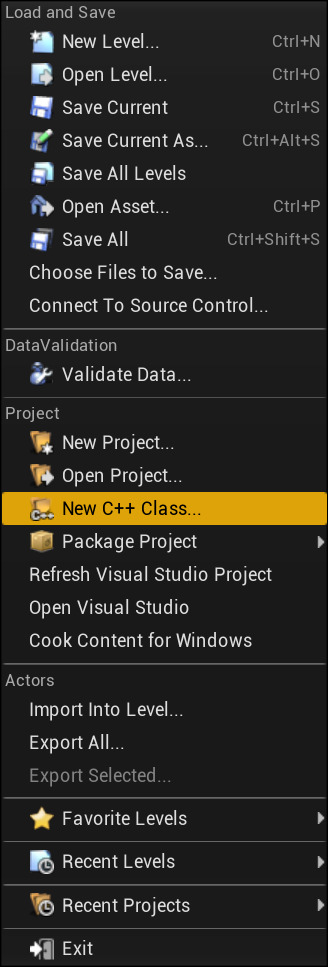
If you have followed this section so far, the following should now be the flow for applying damage to an Actor:

* An external Actor or function calls CauseDamage on our Character, which in turn calls its TakeDamage function.
* TakeDamage calls SetCurrentHealth to change the player's Current Health value on the server.
* SetCurrentHealth calls OnHealthUpdate on the server, causing any functionality that happens in response to changes in the player's health to execute.
* CurrentHealth replicates to all connected clients' copies of the Character.
* When each client receives a new CurrentHealth value from the server, they call OnRep\_CurrentHealth.
* OnRep\_CurrentHealth calls OnHealthUpdate, ensuring that each client responds the same way to the new CurrentHealth value.

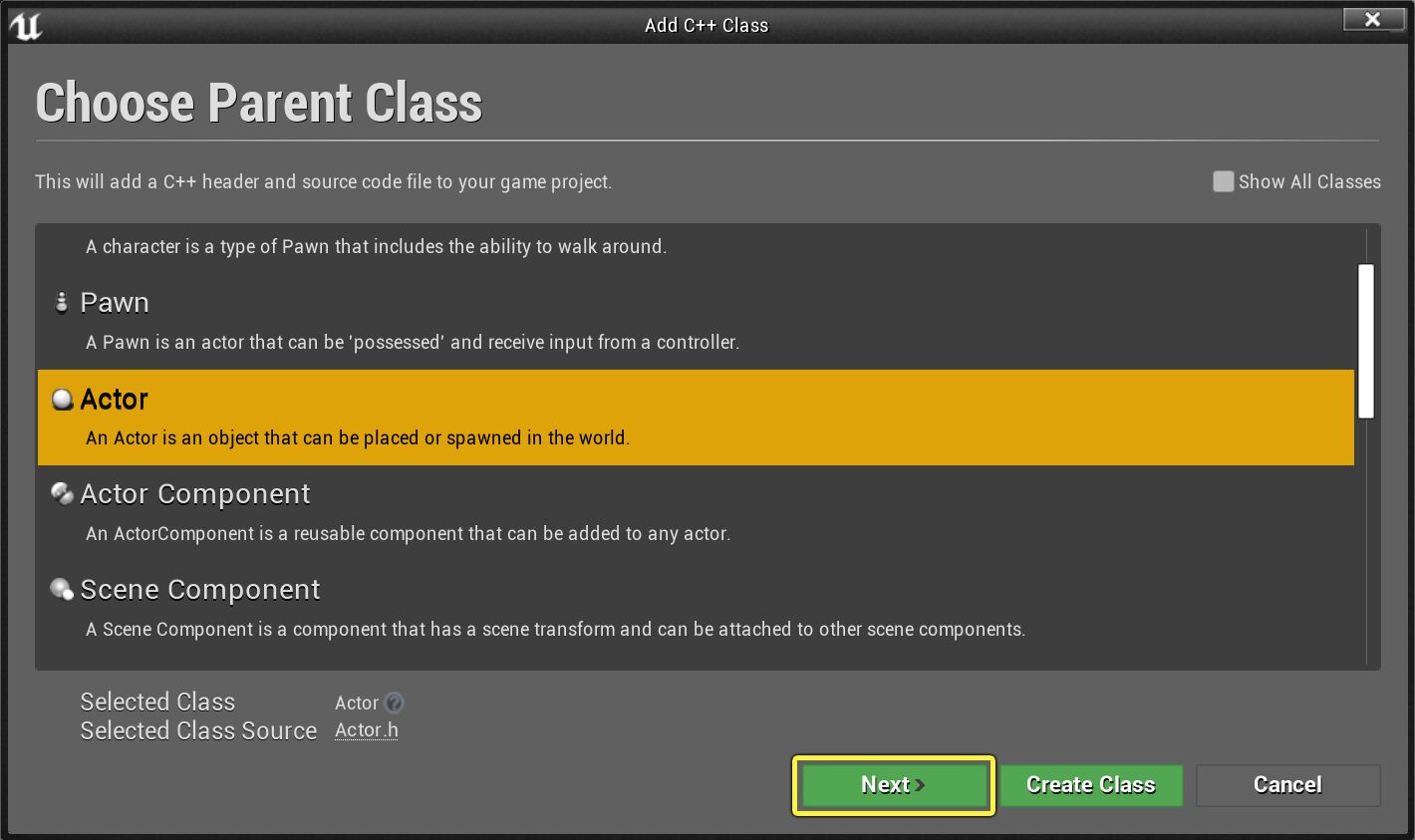
This implementation has two main advantages. First, it condenses the workflow for adding new functionality around two key functions, namely SetCurrentHealth and OnHealthUpdate, which makes maintaining and expanding the code easier for the future. Second, since this implementation does not use any Server, Client, or NetMulticast RPCs, it condenses the amount of information being sent across the network, depending only on the replication of CurrentHealth to trigger all essential changes. Since CurrentHealth would need to replicate regardless of what other functions we implement, this is the most efficient possible model for replicating health changes.

## **Creating a Projectile with Replication**

Inside the Unreal Editor, create a **new C++ class** using either the **File** menu or the **Content Browser**.

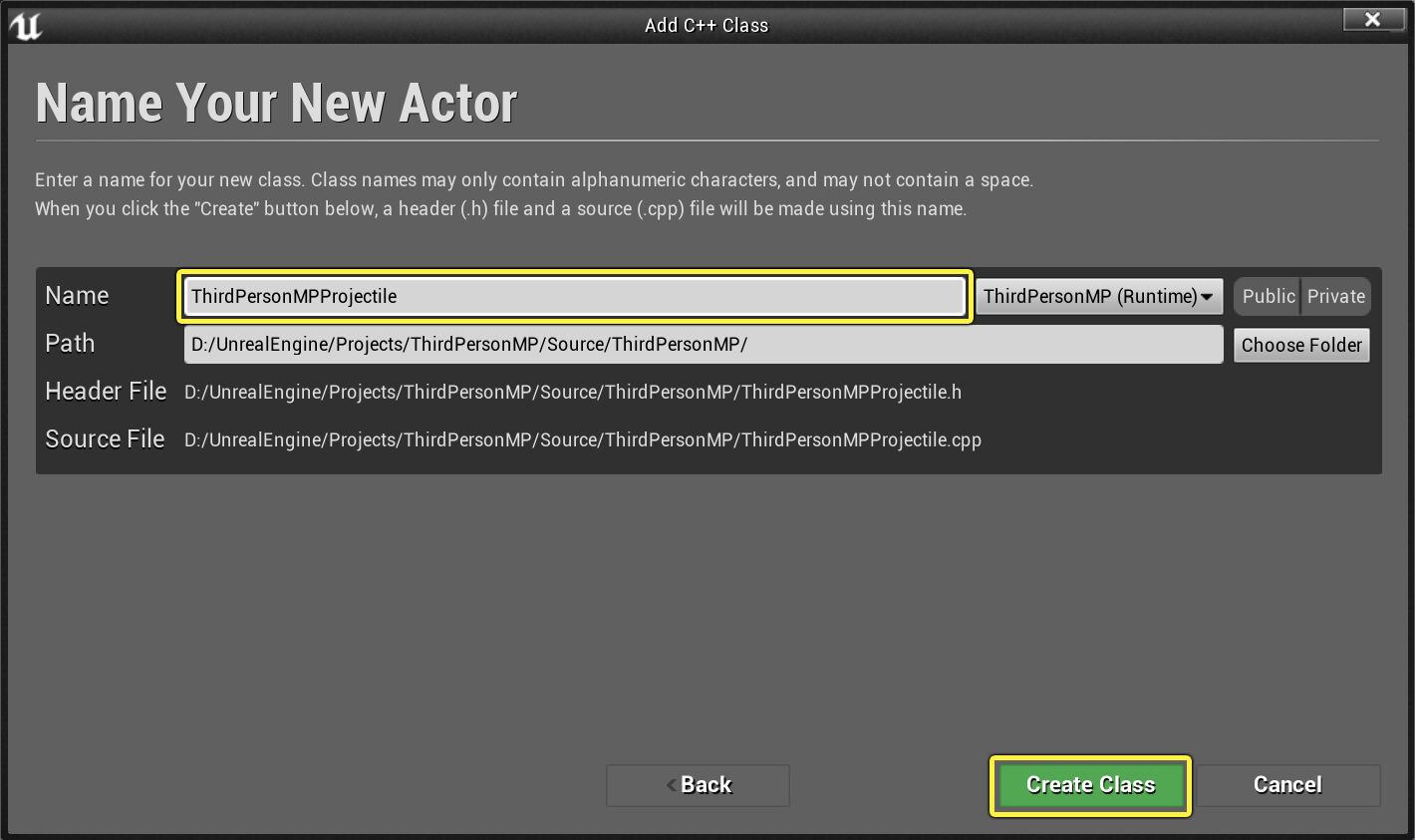


In the **Choose Parent Class** menu, choose **Actor** as the Parent Class and click **Next**.



*Click image for full size.*

In the **Name Your New Actor** menu, name your class **ThirdPersonProjectile** and click **Create Class**.



*Click image for full size.*

Open ThirdPersonProjectile.h and add the following code inside the class definition, under public:

**ThirdPersonProjectile.h**

// Sphere component used to test collision.

UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category="Components")

**class** USphereComponent\* SphereComponent;

// Static Mesh used to provide a visual representation of the object.

UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category="Components")

**class** UStaticMeshComponent\* StaticMesh;

// Movement component for handling projectile movement.

UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category="Components")

**class** UProjectileMovementComponent\* ProjectileMovementComponent;

// Particle used when the projectile impacts against another object and explodes.

UPROPERTY(EditAnywhere, Category = "Effects")

**class** UParticleSystem\* ExplosionEffect;

//The damage type and damage that will be done by this projectile

UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Damage")

TSubclassOf<**class** UDamageType> DamageType;

//The damage dealt by this projectile.

UPROPERTY(EditAnywhere, BlueprintReadOnly, Category="Damage")

**float** Damage;

NOTE  
We precede each of the types in these declarations with the class keyword. This makes each of them a forward declaration of their own classes in addition to being variable declarations, which ensures that their classes will be recognized within the header file. We will be adding `#include`s for them in the CPP file during the next step.

The properties we are declaring will provide us with the following:

* + A **Static Mesh Component** to act as a visual representation of the Projectile.
  + A **Sphere Component** to check for collisions.
  + A **Projectile Movement Component** to move the Projectile.
  + A **Particle System** reference that we are going to use to spawn an explosion effect in a later step.
  + A **Damage Type** for use in damage events.
  + A float value for **Damage** to denote how much health should be subtracted when a Character is hit by this Projectile.

However, none of these are defined yet.

Open ThirdPersonProjectile.cpp and add the following #include statements at the top of the file, underneath the line #include "ThirdPersonProjectile.h":

**ThirdPersonProjectile.cpp**

#include "Components/SphereComponent.h"

#include "Components/StaticMeshComponent.h"

#include "GameFramework/ProjectileMovementComponent.h"

#include "GameFramework/DamageType.h"

#include "Particles/ParticleSystem.h"

#include "Kismet/GameplayStatics.h"

#include "UObject/ConstructorHelpers.h"

We are going to use each of these throughout this walkthrough. The first four are the components we are using, while GamePlayStatics.h will give us access to basic gameplay functions, and ConstructorHelpers.h will give us access to some useful Constructor functions for setting up our components.

Add the following code inside of the constructor in ThirdPersonProjectile.cpp:

**ThirdPersonProjectile.cpp**

bReplicates = **true**;

The bReplicates variable tells the game that this Actor should replicate. By default, the Actor would only exist locally on the machine that spawns it. With bReplicates set to True, as long as an authoritative copy of the Actor exists on the server, it will try to replicate the Actor to all connected clients.

Add the following code inside of the constructor for AThirdPersonProjectile:

**ThirdPersonProjectile.cpp**

//Definition for the SphereComponent that will serve as the Root component for the projectile and its collision.

SphereComponent = CreateDefaultSubobject<USphereComponent>(TEXT("RootComponent"));

SphereComponent->InitSphereRadius(37.5f);

SphereComponent->SetCollisionProfileName(TEXT("BlockAllDynamic"));

RootComponent = SphereComponent;

This will define the SphereComponent when the object is constructed, giving our Projectile collision.

Inside the constructor, add the following code:

**ThirdPersonProjectile.cpp**

//Definition for the Mesh that will serve as our visual representation.

**static** ConstructorHelpers::FObjectFinder<UStaticMesh> DefaultMesh(TEXT("/Game/StarterContent/Shapes/Shape\_Sphere.Shape\_Sphere"));

StaticMesh = CreateDefaultSubobject<UStaticMeshComponent>(TEXT("Mesh"));

StaticMesh->SetupAttachment(RootComponent);

//Set the Static Mesh and its position/scale if we successfully found a mesh asset to use.

**if** (DefaultMesh.Succeeded())

{

StaticMesh->SetStaticMesh(DefaultMesh.Object);

StaticMesh->SetRelativeLocation(FVector(0.0f, 0.0f, -37.5f));

StaticMesh->SetRelativeScale3D(FVector(0.75f, 0.75f, 0.75f));

}

This will define the StaticMeshComponent that we are using as a visual representation. It will automatically try to find the **Shape\_Sphere** mesh inside of **StarterContent** and fill it in for us. The sphere will also be scaled so as to align with our SphereComponent in size.

Inside the constructor, add the following code:

**ThirdPersonProjectile.cpp**

**static** ConstructorHelpers::FObjectFinder<UParticleSystem> DefaultExplosionEffect(TEXT("/Game/StarterContent/Particles/P\_Explosion.P\_Explosion"));

**if** (DefaultExplosionEffect.Succeeded())

{

ExplosionEffect = DefaultExplosionEffect.Object;

}

This will set the asset reference for our ExplosionEffect to be the **P\_Explosion** asset inside of StarterContent.

Inside the constructor, add the following code:

**ThirdPersonProjectile.cpp**

//Definition for the Projectile Movement Component.

ProjectileMovementComponent = CreateDefaultSubobject<UProjectileMovementComponent>(TEXT("ProjectileMovement"));

ProjectileMovementComponent->SetUpdatedComponent(SphereComponent);

ProjectileMovementComponent->InitialSpeed = 1500.0f;

ProjectileMovementComponent->MaxSpeed = 1500.0f;

ProjectileMovementComponent->bRotationFollowsVelocity = **true**;

ProjectileMovementComponent->ProjectileGravityScale = 0.0f;

This will define the Projectile Movement Component for our Projectile. This Component is replicated, and any movement that it performs on the server will be reproduced on clients.

Inside the constructor, add the following code:

**ThirdPersonProjectile.cpp**

DamageType = UDamageType::StaticClass();

Damage = 10.0f;

These will initialize both the amount of Damage that the Projectile will deal to an Actor as well as the Damage Type that will be used in the damage event. Here we are initializing with the base UDamageType, as we have not yet defined any new Damage Types.

## **Making the Projectile Cause Damage**

If you have been following along thus far, then it is possible for you to spawn the projectile on the server, and it will appear and move on all clients. However, if it hits a wall or a blocking object, it will stop. We need it to apply damage to players, and we need to show an explosion effect to all of the connected Clients in the session.

In ThirdPersonProjectile.h add the following code under Protected:

**ThirdPersonProjectile.h**

**virtual** **void** Destroyed() **override**;

In ThirdPersonProjectile.cpp add the following implementation for this function:

**ThirdPersonProjectile.cpp**

**void** AThirdPersonProjectile::Destroyed()

{

FVector spawnLocation = GetActorLocation();

UGameplayStatics::SpawnEmitterAtLocation(**this**, ExplosionEffect, spawnLocation, FRotator::ZeroRotator, **true**, EPSCPoolMethod::AutoRelease);

}

The Destroyed function is called any time an Actor is destroyed. Particle emitters themselves do not normally replicate, but since Actor destruction does replicate, we know that if we destroy this projectile on the server then this function will be called on each connected client when they destroy their own copies of it. As a result, all players will see the explosion effect when the projectile is destroyed.

In ThirdPersonProjectile.h add the following code under Protected:

**ThirdPersonProjectile.h**

UFUNCTION(Category="Projectile")

**void** OnProjectileImpact(UPrimitiveComponent\* HitComponent, AActor\* OtherActor, UPrimitiveComponent\* OtherComp, FVector NormalImpulse, **const** FHitResult& Hit);

In ThirdPersonProjectile.cpp add the following implementations for this function:

**ThirdPersonProjectile.cpp**

**void** AThirdPersonProjectile::OnProjectileImpact(UPrimitiveComponent\* HitComponent, AActor\* OtherActor, UPrimitiveComponent\* OtherComp, FVector NormalImpulse, **const** FHitResult& Hit)

{

**if** ( OtherActor )

{

UGameplayStatics::ApplyPointDamage(OtherActor, Damage, NormalImpulse, Hit, GetInstigator()->Controller, **this**, DamageType);

}

Destroy();

}

This is the function that we are going to call when the Projectile impacts with an object. If the object it impacts with is a valid Actor, it will call the ApplyPointDamage function to damage it at the point where the collision takes place. Meanwhile, any collision regardless of the impacted surface will destroy this Actor, causing the explosion effect to appear.

In ThirdPersonProjectile.cpp add the following code to the Constructor, underneath the line that reads RootComponent = SphereComponent:

**ThirdPersonProjectile.h**

//Registering the Projectile Impact function on a Hit event.

**if** (GetLocalRole() == ROLE\_Authority)

{

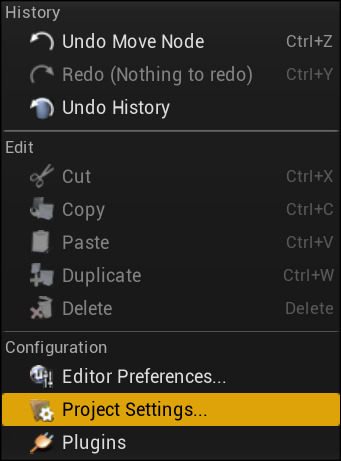
SphereComponent->OnComponentHit.AddDynamic(**this**, &AThirdPersonProjectile::OnProjectileImpact);

}

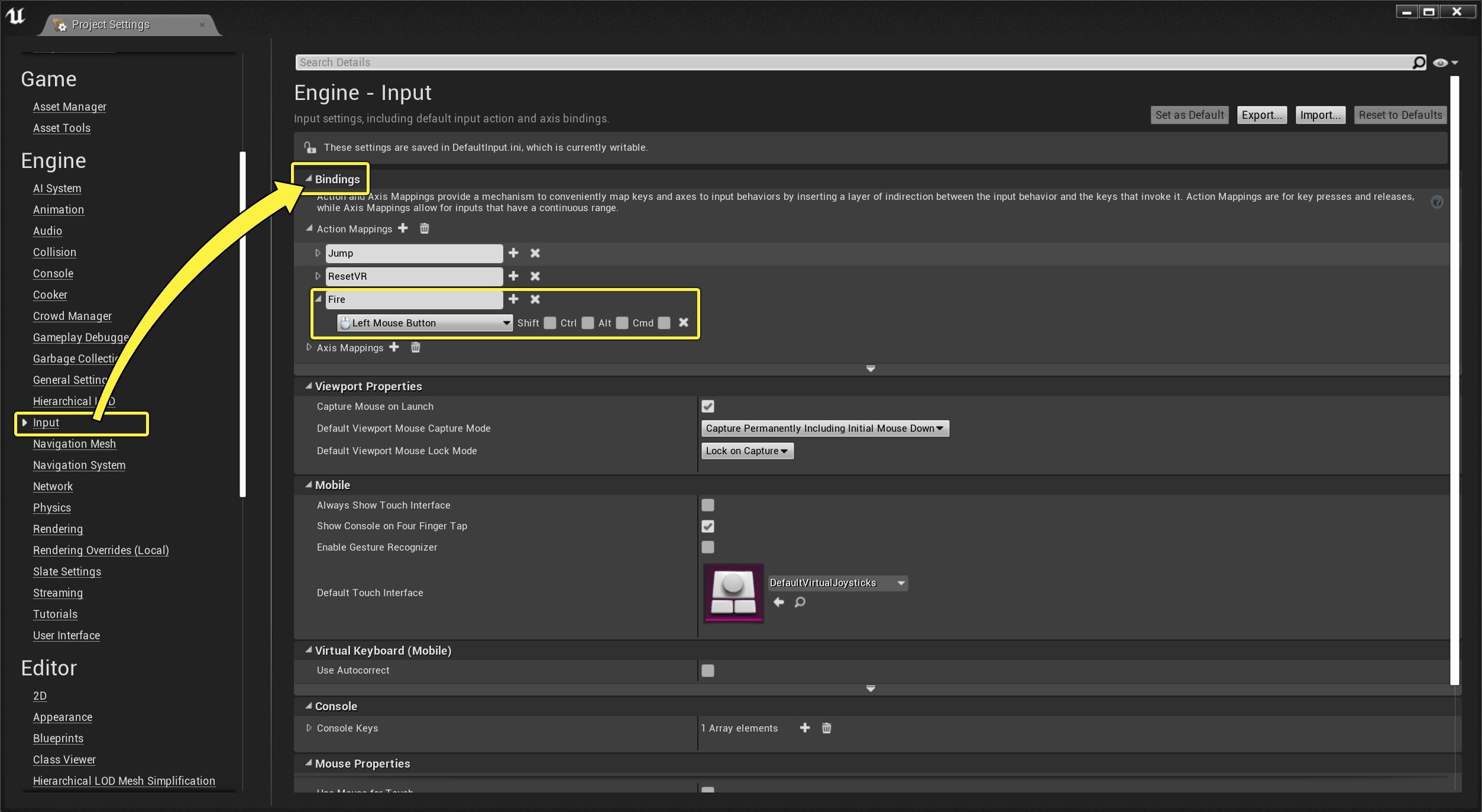
This will register the OnProjectileImpact function with the OnComponentHit event on the Sphere Component, which acts as the projectile's primary collision component. To make especially sure that only the server runs this gameplay logic, we check for GetLocalRole() == ROLE\_Authority before registering OnProjectileImpact.

## **Shooting the Projectile**

Open the **Editor**, then click the **Edit** drop-down menu at the top of the screen, and open your **Project Settings**.



In the **Engine** section, click on **Input** to open up your project's Input Settings. Unfold the **Bindings** section and add a new entry to it. Name it "**Fire**", and select the **Left Mouse Button** as the key this Action is bound to.



*Click image for full size.*

In ThirdPersonCharacter.cpp add the following #include, underneath the line that reads #include "Engine/Engine.h":

**ThirdPersonCharacter.cpp**

#include "ThirdPersonProjectile.h"

This will enable our Character class to recognize the projectile's type and spawn it.

In ThirdPersonCharacter.h add the following code under protected:

**ThirdPersonCharacter.h**

UPROPERTY(EditDefaultsOnly, Category="Gameplay|Projectile")

TSubclassOf<**class** AThirdPersonProjectile> ProjectileClass;

/\*\* Delay between shots in seconds. Used to control fire rate for our test projectile, but also to prevent an overflow of server functions from binding SpawnProjectile directly to input.\*/

UPROPERTY(EditDefaultsOnly, Category="Gameplay")

**float** FireRate;

/\*\* If true, we are in the process of firing projectiles. \*/

**bool** bIsFiringWeapon;

/\*\* Function for beginning weapon fire.\*/

UFUNCTION(BlueprintCallable, Category="Gameplay")

**void** StartFire();

/\*\* Function for ending weapon fire. Once this is called, the player can use StartFire again.\*/

UFUNCTION(BlueprintCallable, Category = "Gameplay")

**void** StopFire();

/\*\* Server function for spawning projectiles.\*/

UFUNCTION(Server, Reliable)

**void** HandleFire();

/\*\* A timer handle used for providing the fire rate delay in-between spawns.\*/

FTimerHandle FiringTimer;

These are the variables and functions we will be using to fire our projectiles. HandleFire is the only RPC we will implement in this tutorial, and it will be responsible for spawning projectiles on the server. Because it has the Server specifier, any attempt to call it on a client will result in the call being directed over the network to the authoritative Character on the server instead.

Because HandleFire has the Reliable specifier as well, it is placed into a queue for reliable RPCs whenever it gets called, and it is removed from the queue when the server successfully receives it. This guarantees that the server will definitely receive this function call. However, the queue for reliable RPCs can overflow if too many RPCs are placed into it at once without removing them, and if it does then it will force the user to disconnect. Therefore, we need to be cautious in how often we allow players to call this function.

In ThirdPersonCharacter.cpp add the following code to the bottom of the constructor:

**ThirdPersonCharacter.cpp**

//Initialize projectile class

ProjectileClass = AThirdPersonProjectile::StaticClass();

//Initialize fire rate

FireRate = 0.25f;

bIsFiringWeapon = **false**;

These will initialize the variables necessary to handle firing the projectile.

In ThirdPersonCharacter.cpp add the following implementations:

**ThirdPersonCharacter.cpp**

**void** AThirdPersonCharacter::StartFire()

{

**if** (!bIsFiringWeapon)

{

bIsFiringWeapon = **true**;

UWorld\* World = GetWorld();

World->GetTimerManager().SetTimer(FiringTimer, **this**, &AThirdPersonCharacter::StopFire, FireRate, **false**);

HandleFire();

}

}

**void** AThirdPersonCharacter::StopFire()

{

bIsFiringWeapon = **false**;

}

**void** AThirdPersonCharacter::HandleFire\_Implementation()

{

FVector spawnLocation = GetActorLocation() + ( GetControlRotation().Vector() \* 100.0f ) + (GetActorUpVector() \* 50.0f);

FRotator spawnRotation = GetControlRotation();

FActorSpawnParameters spawnParameters;

spawnParameters.Instigator = GetInstigator();

spawnParameters.Owner = **this**;

AThirdPersonProjectile\* spawnedProjectile = GetWorld()->SpawnActor<AThirdPersonProjectile>(spawnLocation, spawnRotation, spawnParameters);

}

StartFire is the function that players call on their local machine in order to initiate the firing process, and it restricts how often the user is allowed to call HandleFire based on the following criteria:

* The user cannot fire a projectile if they are already in the middle of firing. This is designated with bFiringWeapon, which is set to true when StartFire is called.
* bFiringWeapon is only set to false when StopFire is called.
* StopFire is called when a timer with a length of FireRate finishes.

This means that when the user fires a projectile, they must wait a number of seconds equal to FireRate before they can fire again. This will function consistently regardless of what kind of input StartFire is bound to. For example, if the user binds the "Fire" command to a scroll wheel or similarly inappropriate input, or if they mash the button repeatedly, this function will still execute at an acceptable interval of time and not overflow the user's queue for reliable functions with calls to HandleFire.

Because HandleFire is a Server RPC, its implementation in the CPP file must have the suffix \_Implementation added to the function name. Our implementation here uses the Character's Control Rotation to get the direction that the camera is facing, then spawn the projectile facing in that direction, enabling the player to aim. The projectile's Projectile Movement Component then handles moving it in that direction.

In ThirdPersonCharacter.cpp add the following at the bottom of the function SetupPlayerInputComponent:

**ThirdPersonCharacter.cpp**

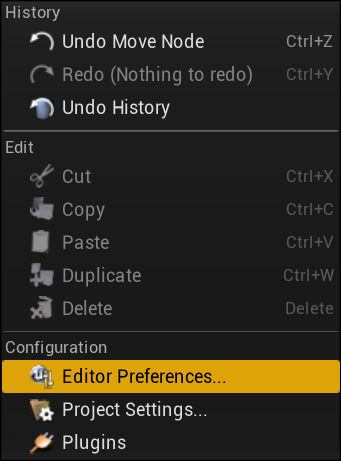
// Handle firing projectiles

PlayerInputComponent->BindAction("Fire", IE\_Pressed, **this**, &AThirdPersonMPCharacter::StartFire);

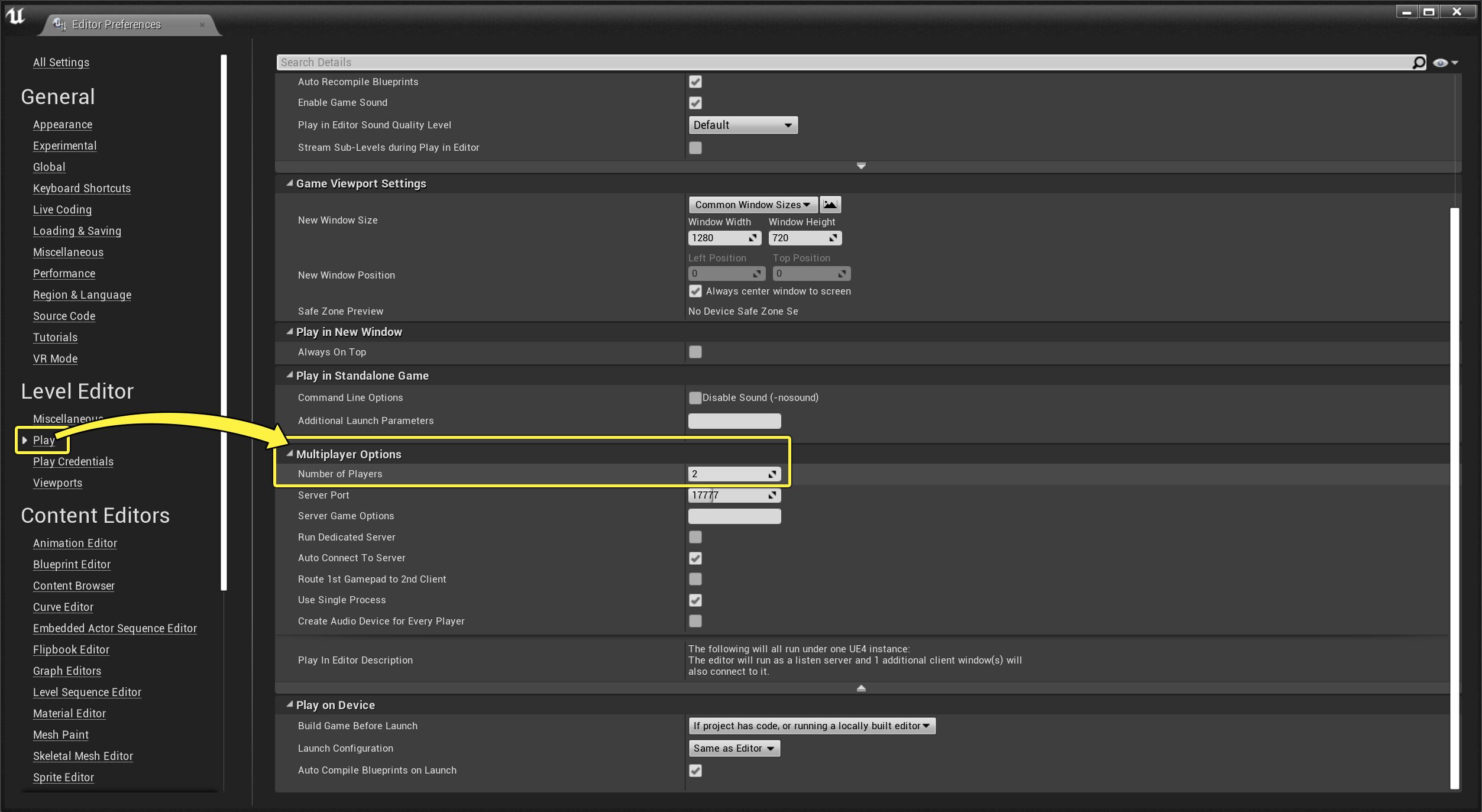
This binds StartFire to the **Fire** Input Action we created in the first step of this section, enabling the user to activate it.

## **Test Your Game**

Open your Project in the Editor. Click the **Edit** drop-down menu, and open **Editor Preferences**.



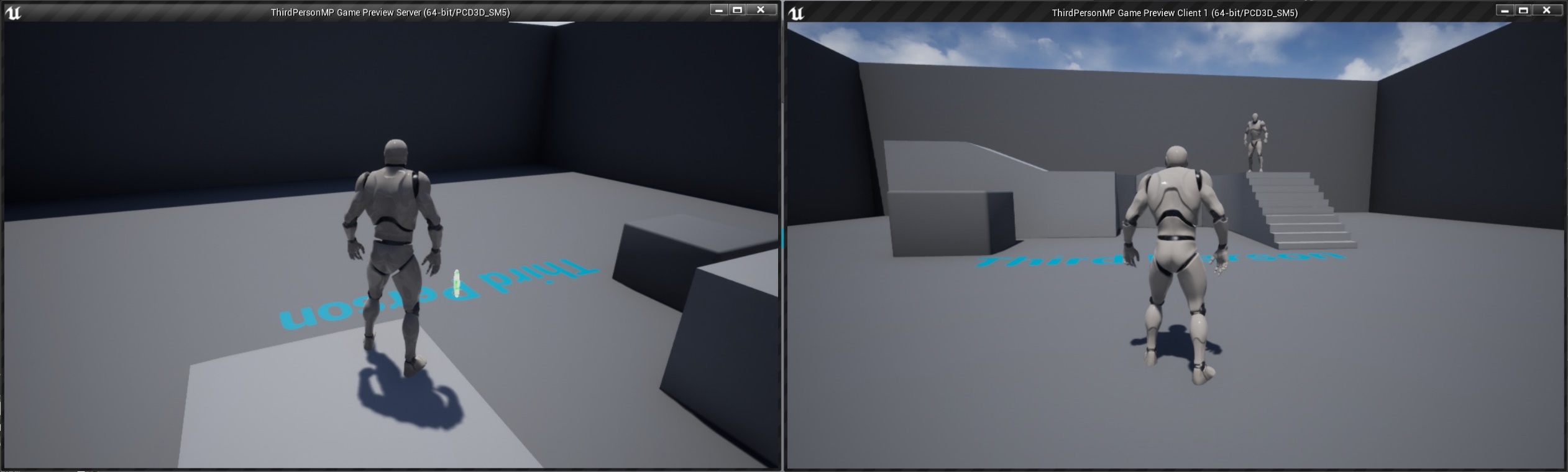
Navigate to the **Level Editor** section and click the **Play** menu. Find the **Multiplayer Options** and Change the **Number of Players** to 2.



*Click image for full size.*

Press the **Play** button. The main **Play in Editor (PIE)** window will start a Multiplayer Session as the Server, and a second PIE window will open and connect as the Client.

## **Final Result**



*Click image for full size.*

Both players in your game should be able to see each other moving, and they should also be able to shoot the custom projectile at each other. When one player is hit by the custom projectile, the explosion particle should appear for both players, and the player taking the hit will receive a "hit" message telling them how much damage they took and their current health, while all other players in the session should not see anything. If a player's health is reduced to 0, they should see a message informing them that they have been killed.

Now that you have completed this walkthrough, you should have a grasp on the basics of building multiplayer functionality in C++, including an overview of variable and component replication, how to work with Network Roles, and when it is appropriate to use RPCs. With this information you should be able to build your own multiplayer games within Unreal's Server-Client model.