Distributed Programming

Lecture 05 - Multiplayer and Network Communication in Unreal Engine

Edirlei Soares de Lima
<edirlei.lima@universidadeeuropeia.pt>

Multiplayer in Unreal Engine

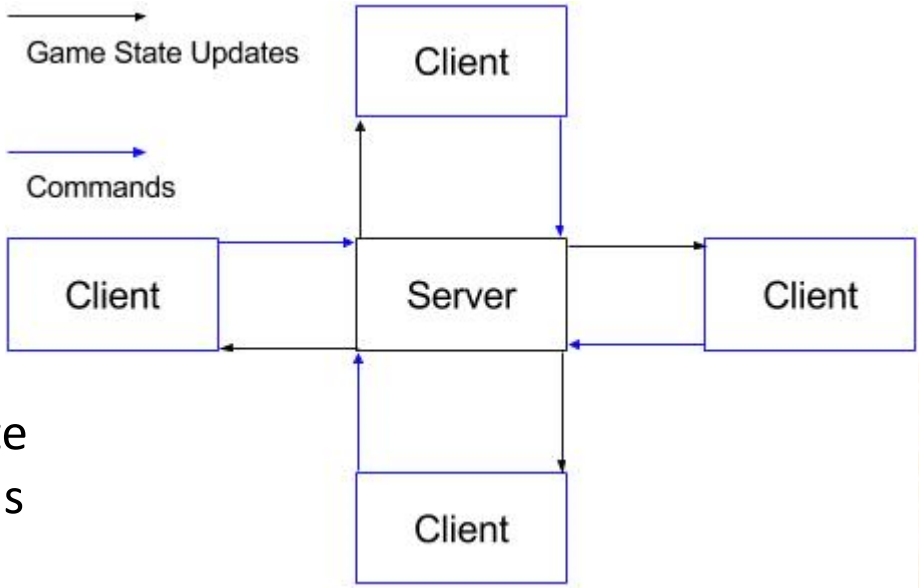
- The Unreal Engine is built with multiplayer gaming in mind.
  - Is very easy to extend a single player experience to multiplayer.
  - Even single-player games, use the networking architecture.

- The engine is based on a **client-server model**.
  - The server is authoritative and makes sure all connected clients are continually updated.

- Actors are the main element that the server uses to keep clients up to date.
  - The server sends information about the actors to clients;
  - Clients have an approximation of each actor and the server maintains the authoritative version.

Multiplayer in Unreal Engine

- **The server is in charge of driving the flow of gameplay.**
  - It handles network connections, notifies clients when gameplay starts and ends, when is time to travel to a new map, along with actor replication updates.
  - Only the server contains a valid copy of the GameMode actor. Clients contain only an approximate copy of the actors, and can use it as a reference to know the general state of the game.



Multiplayer in Unreal Engine

- **Network Modes:**
  - **Standalone**: the server runs on a local machine and not accept clients from remote machines. Is used for single-player games.
  - **Dedicated Server**: the server has no local players and can run more efficiently by discarding sound, graphics, user input, and other player-oriented features. Is used for multiplayer games hosted on a trusted and reliable server where high-performing are needed.
  - **ListenServer**: is a server that hosts a local player, but is open to connections from remote players as well. Is good for games where users can set up and play their own games without a third-party server.
  - **Client**: the machine is a client that can connect to a dedicated or listen server, and therefore will not run server-side logic.

Multiplayer in Unreal Engine

- The core element of the networking process in Unreal Engine is **Actor Replication**.
  - The server maintains a list of actors and updates the client periodically so that the client can have a close approximation of each actor (that is marked to be replicated).

- Actors are updated in two main ways:
  - Property updates;
  - RPCs (Remote Procedure Calls).

- Properties are replicated automatically (any time they change) and RPCs are only replicated when executed.

Property Replication

- Example of property to be replicated: actor's health.
- Each actor maintains a list of properties that can be marked for replication to clients.
  - Whenever the value of the variable changes on the server side, the server sends the client the updated value.
  - Property updates only come from the server (i.e.: the client will never send property updates to the server).
  - Some properties replicate by default (e.g.: Location, Rotation, etc.).
- Actor property replication is reliable.

Property Replication

- **Replicate a property:**
  1. Set the replicated keyword:

```
UPROPERTY(replicated)
float health;
```

  2. Implement the GetLifetimeReplicatedProps function:

```
void MyClass::GetLifetimeReplicatedProps(TArray<FLifetimeProperty>&
                                         OutLifetimeProps) const{
  Super::GetLifetimeReplicatedProps(OutLifetimeProps);
  DOREPLIFETIME(MyClass, health);
}
```

Default replication rule: replicates to all clients

  3. Enable replication in the constructor method:

```
SetReplicates(true);
```

Remote Procedure Calls (RPCs)

- Example of RPC: a function to spawn an explosion to each client at a certain location.

- RPCs are functions that are called locally, but executed remotely on another machine.
  - Primary use: to do unreliable gameplay events that are temporary or cosmetic in nature.
  - E.g.: play sounds, spawn particles, or do other temporary effects that are not crucial to the Actor functioning.

- By default, RPCs are unreliable. To be reliable, a especial keyword (`Reliable`) must be used in the definition of the RPC.

Remote Procedure Calls (RPCs)

- **Defining a RPC:**
  - To declare a function as an RPC that will be called on the server, but executed on the client:

```
UFUNCTION(Client)
void ClientRPCFunction();
```

  - To declare a function as an RPC that will be called on the client, but executed on the server:

```
UFUNCTION(Server)
void ServerRPCFunction();
```

  - To declare a function as an RPC that will be called from the server, and then executed on the server and on all connected clients:

```
UFUNCTION(NetMulticast)
void MulticastRPCFunction();
```

RPC Validation

- The validation function for an RPC allows the detection of bad parameters or cheating:
  - It can notify the system to disconnect the client who initiated the call.
- Example:

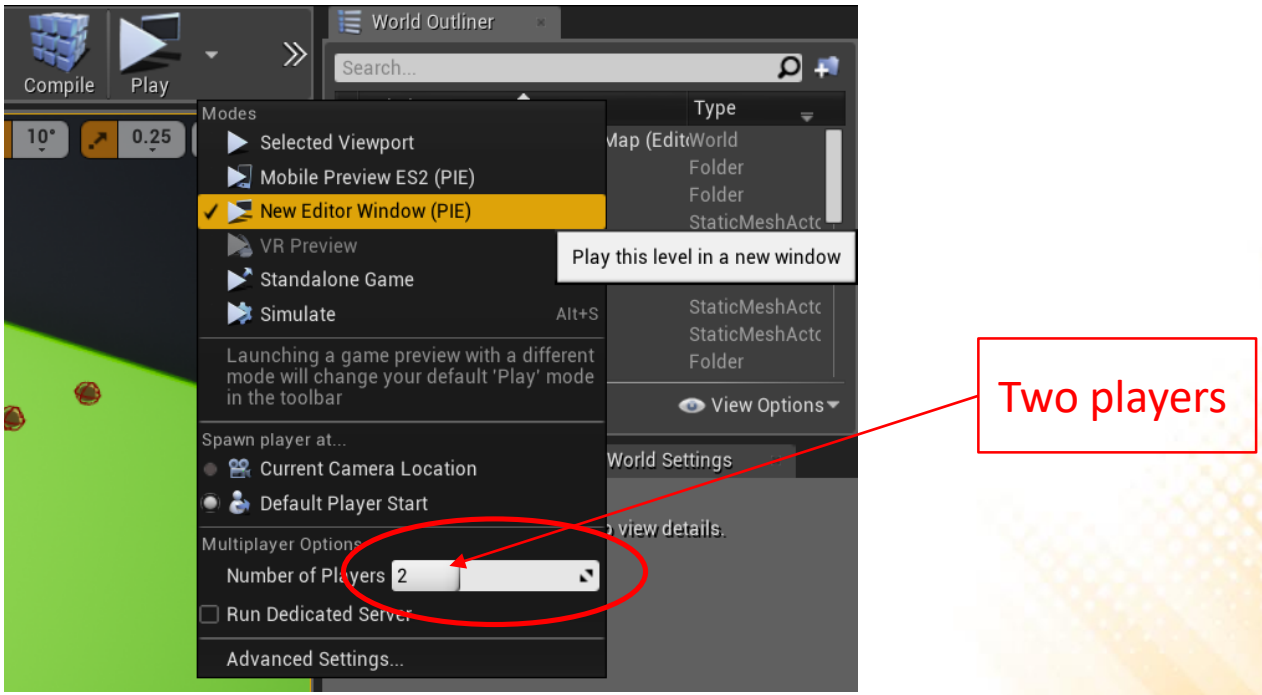
```
UFUNCTION(Server, WithValidation)
void SomeRPCFunction(int32 AddHealth);
```

```
bool SomeRPCFunction_Validate(int32 AddHealth){
  if (AddHealth > MAX_ADD_HEALTH){
    return false;
  }
  return true;
}

void SomeRPCFunction_Implementation(int32 AddHealth){
  Health += AddHealth;
}
```

Back to the First Game

- Can we transform our first game into a multiplayer game?
  - Yes! We just need to fix some small issues.

- Test the game in a multiplayer setup:



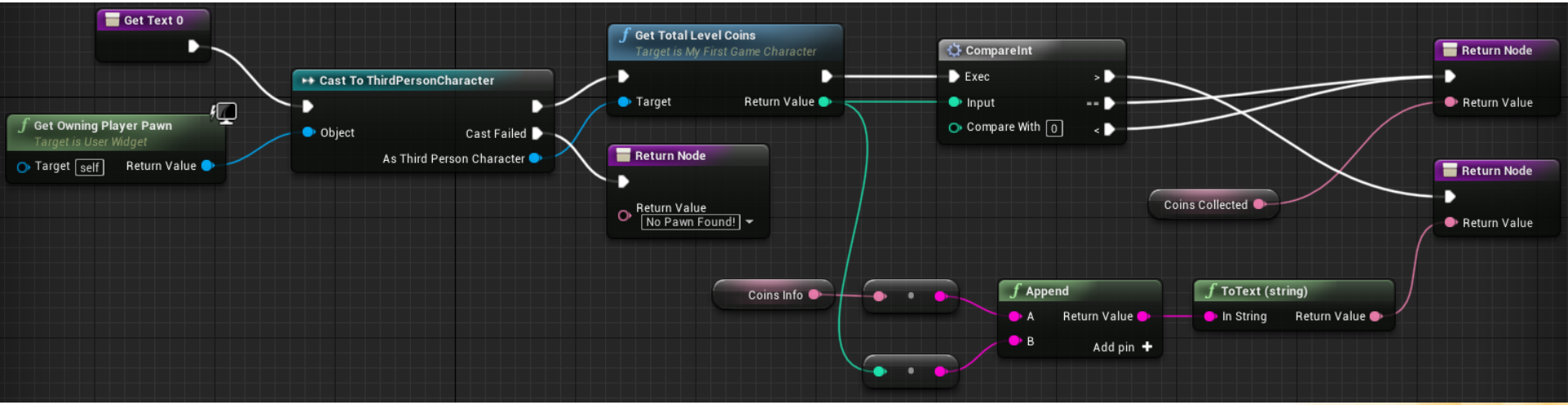
Back to the First Game

- What works in multiplayer:
  1. Player movement;
  2. Zombie movement;
  3. Coins;

- What doesn’t work in multiplayer?
  1. No game mode in the client and no total number of collected coins;
  2. End level area and the game over messages;
  3. Enemy animation and walk speed;

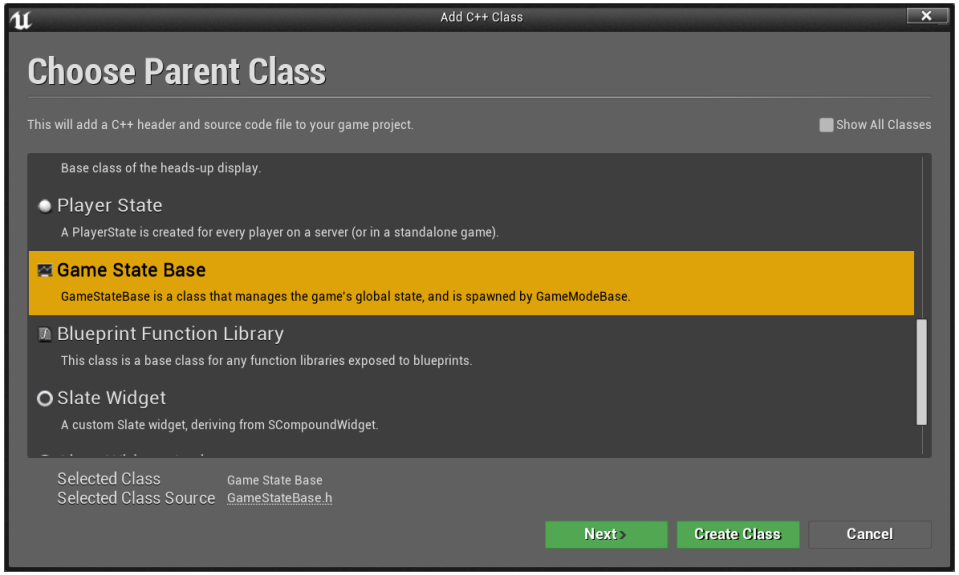
Back to the First Game

- First issue: there is no game mode in clients.
- Solution:
  - Logic change: now multiple players can collect coins cooperatively, so its more appropriate to show the number of remaining coins in the level;
  - Move the function that computes the number of coins in the level to the character class (will be locally computed);
  - Change the blueprint that access the game mode to show the number collected coins;



Back to the First Game

- Second issue: end level area and the game over messages.
  - **Problem 1**: disable the player controller on all clients/server
    - Issue: the CompleteLevel function runs only on the server, because it is implemented in the Game Mode class.
- Solution:
  - Create a multicast function in a Game State class;



Back to the First Game

- Second issue: end level area and the game over messages.
  - **Problem 1**: disable the player controller on all clients/server
    - Issue: the CompleteLevel function runs only on the server, because it is implemented in the Game Mode class.
- Solution:
  - Create a multicast function in a Game State class;

MyGameStateBase.h

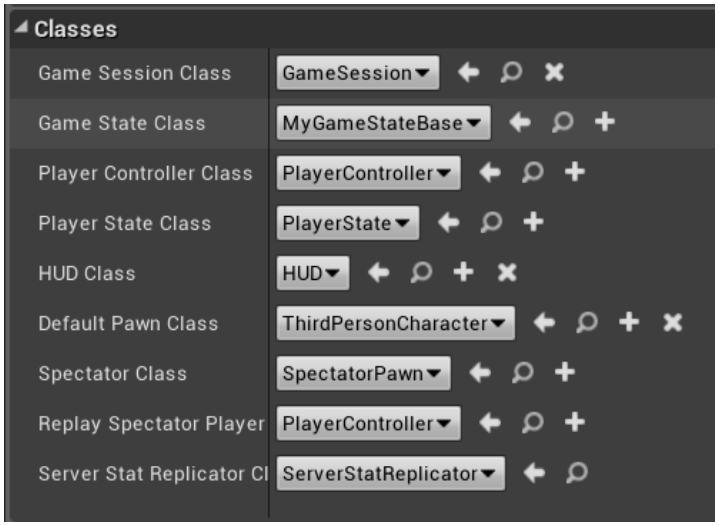
```
public:
  UFUNCTION(NetMulticast, Reliable)
  void MulticastOnLevelComplete(APawn* character, bool succeeded);
```

MyGameStateBase.cpp

```
void AMyGameStateBase::MulticastOnLevelComplete_Implementation(
                              APawn* character, bool succeeded){
  for (FConstPawnIterator it = GetWorld()->GetPawnIterator(); it; it++){
    APawn* pawn = it->Get();
    if (pawn && pawn->IsLocallyControlled()){
      pawn->DisableInput(nullptr);
    }
}
```

Back to the First Game

- Second issue: end level area and the game over messages.
  - **Problem 1**: disable the player controller on all clients/server
    - Issue: the CompleteLevel function runs only on the server, because it is implemented in the Game Mode class.
- Solution:
  - Set the new Game State class in the blueprint instance of the Game Mode:



Back to the First Game

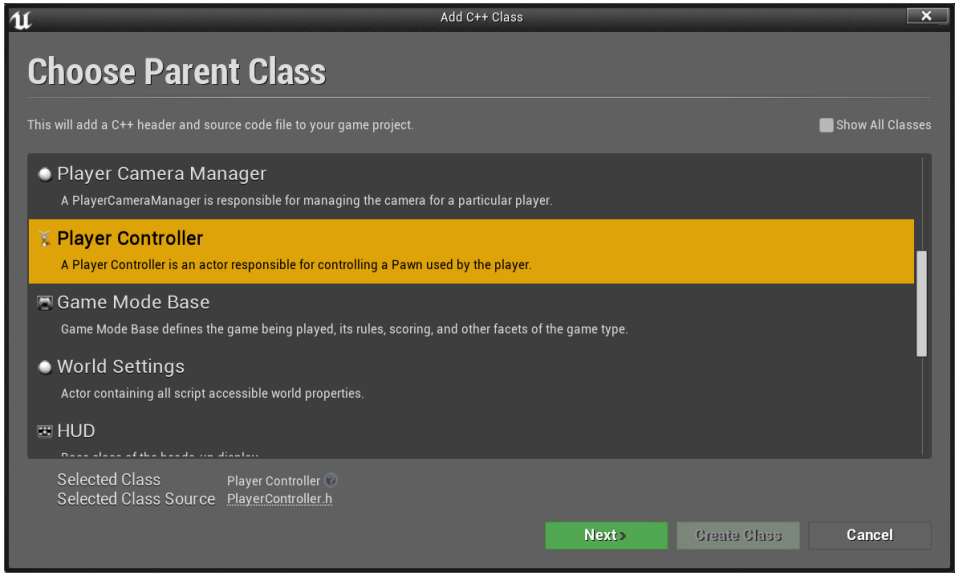
- Second issue: end level area and the game over messages.
  - **Problem 1**: disable the player controller on all clients/server
    - Issue: the CompleteLevel function runs only on the server, because it is implemented in the Game Mode class.
- Solution:
  - Call the multicast function in CompleteLevel function;

MyFirstGameGameMode.cpp

```
...

AMyGameStateBase *gameState = GetGameState<AMyGameStateBase>();
if (gameState)
{
  gameState->MulticastOnLevelComplete(character, succeeded);
}

...
```

Back to the First Game

- Second issue: end level area and the game over messages.
  - **Problem 2**: show the “Level Completed Message” for all clients/server.
    - Issue: we need a class to create the widget that exist only once on the clients.
- Solution:
  - Create the widget in a new Game Controller class;



Back to the First Game

- Second issue: end level area and the game over messages.
  - **Problem 2**: show the "Level Completed Message" for all clients/server.
    - Issue: we need a class to create the widget that exist only once on the clients.
- Solution:
  - Expose the OnLevelCompleted to blueprint implementation:

MyPlayerController.h

```
public:
  UFUNCTION(BlueprintImplementableEvent, Category = "Gameplay Events")
  void OnLevelCompleted(APawn* charact, bool succeeded);
```

  - Create a blueprint for the new player controller and implement the event:

Back to the First Game

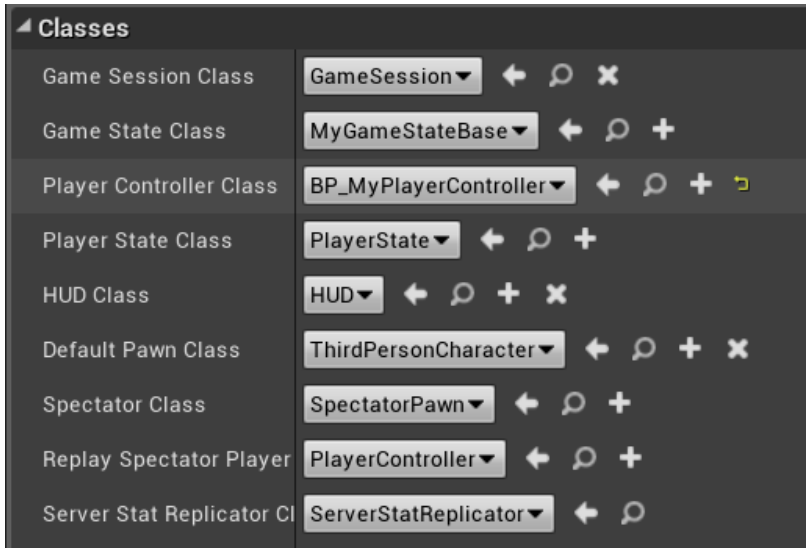
- Second issue: end level area and the game over messages.
  - **Problem 2**: show the "Level Completed Message" for all clients/server.
    - Issue: we need a class to create the widget that exist only once on the clients.
- Solution:
  - Call the OnLevelCompleted function in Multicast function:

```
void AMyGameStateBase::MulticastOnLevelComplete_Implementation(
                                  APawn* character, bool succeeded){
  ...

  for (FConstPlayerControllerIterator it =
       GetWorld()->GetPlayerControllerIterator(); it; it++){
    AMyPlayerController* pController = Cast<AMyPlayerController>
                                                  (it->Get());
    if ((pController) && (pController->IsLocalController())){
      pController->OnLevelCompleted(character, succeeded);
    }
  }
}
```

MyGameStateBase.cpp

Back to the First Game

- Second issue: end level area and the game over messages.
  - **Problem 2**: show the "Level Completed Message" for all clients/server.
    - Issue: we need a class to create the widget that exist only once on the clients.
- Solution:
  - Set the new Player Controller class in the blueprint instance of the Game Mode:



Back to the First Game

- Second issue: end level area and the game over messages.
  - **Problem 3**: move the camera to spectator viewpoint in all clients/server.
- Solution:
  - Call the SetViewTargetWithBlend function for all player controllers:

```
void AMyFirstGameGameMode::CompleteLevel(APawn* character,
                                         bool succeeded){
  ...

  for (FConstPlayerControllerIterator it =
       GetWorld()->GetPlayerControllerIterator(); it; it++){
    APlayerController* playerController = it->Get();
    if (playerController){
      playerController->SetViewTargetWithBlend(foundViewpoints[0],0.8f);
    }
  }
  ...
}
```

MyFirstGameGameMode.cpp

Back to the First Game

- Third issue: enemy animation and walk speed.
  - **Problem**: synchronize the animation and walk speed with all clients/server.
- Solution:
  - Create a variable to represent a chasing state and replicate it to all clients:

EnemyCharacter.h

```
protected:
  ...
  UPROPERTY(Replicated)
  bool isChasing;
```

EnemyCharacter.cpp

```
void AEnemyCharacter::GetLifetimeReplicatedProps(Tarray
<FLifetimeProperty>& OutLifetimeProps) const
{
  Super::GetLifetimeReplicatedProps(OutLifetimeProps);
  DOREPLIFETIME(AEnemyCharacter, isChasing);
}
```

Back to the First Game

EnemyCharacter.cpp

```
AEnemyCharacter::AEnemyCharacter(){
  ...
  isChasing = false;
  SetReplicates(true);
}

void AEnemyCharacter::EnemyComponentHit(UPrimitiveComponent*
          HitComponent, AActor* OtherActor, UPrimitiveComponent*
          OtherComp, FVector NormalImpulse, const FHitResult& Hit){
  if (Role == ROLE_Authority){
    SeePlayer(Cast<APawn>(OtherActor));
  }
}

void AEnemyCharacter::SeePlayer(APawn *pawn){
  if ((pawn) && (AIController) && (!target)){
    ...
    isChasing = true;
  }
}
```

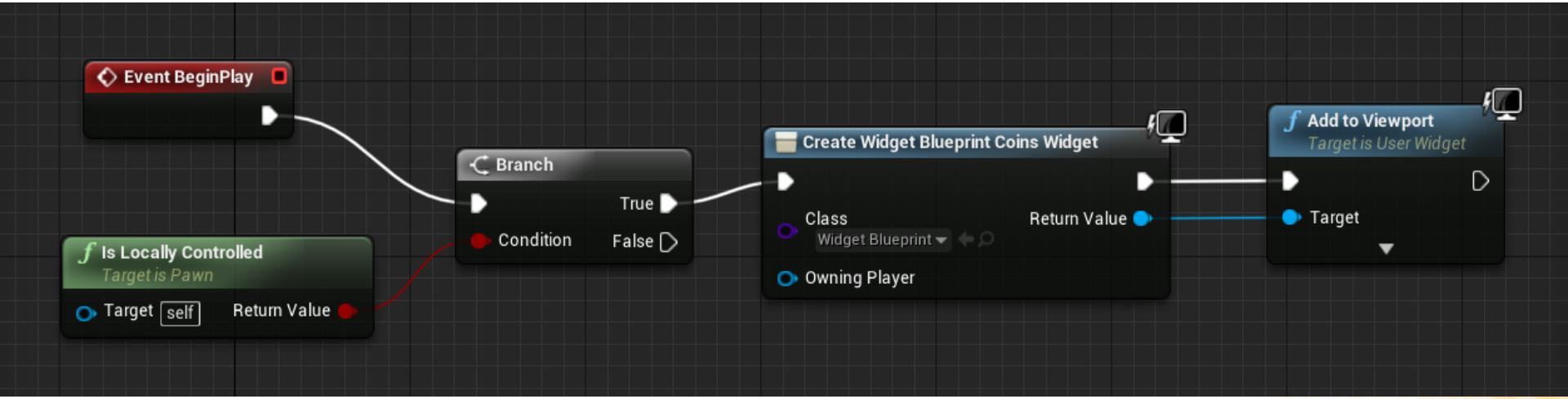
Back to the First Game

EnemyCharacter.cpp

```
void AEnemyCharacter::Tick(float DeltaTime){
  Super::Tick(DeltaTime);
  if ((target)&& (Role == ROLE_Authority)){
    if (FVector::Dist(GetActorLocation(),
       target->GetActorLocation()) > SensingComponent->SightRadius){
       ...
       isChasing = false;
    }
  }
  if ((isChasing)&& (this->GetMesh()->GlobalAnimRateScale != 2.5f)){
    this->GetMesh()->GlobalAnimRateScale = 2.5f;
    this->GetCharacterMovement()->MaxWalkSpeed = 150.0f;
  }
  else if (this->GetMesh()->GlobalAnimRateScale != 1.0f){
    this->GetMesh()->GlobalAnimRateScale = 1.0f;
    this->GetCharacterMovement()->MaxWalkSpeed = 50.0f;
  }
}
```

Back to the First Game

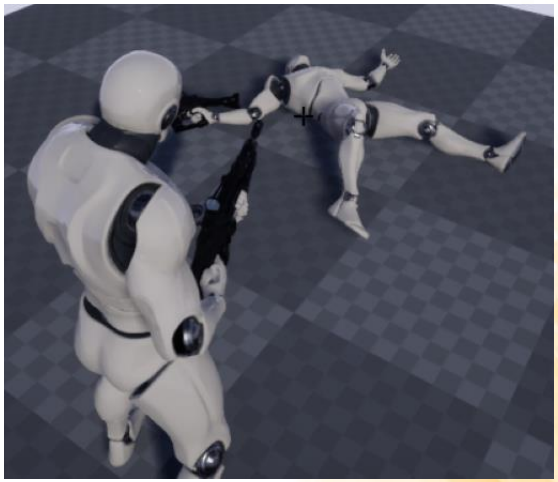
- Fourth issue: runtime error message.
  - **Error**: Blueprint Runtime Error: "Accessed None trying to read property CallFunc_Create_ReturnValue".
- Solution:
  - A small change in the ThirdPersonCharacter blueprint solves the issue:



New Multiplayer Game

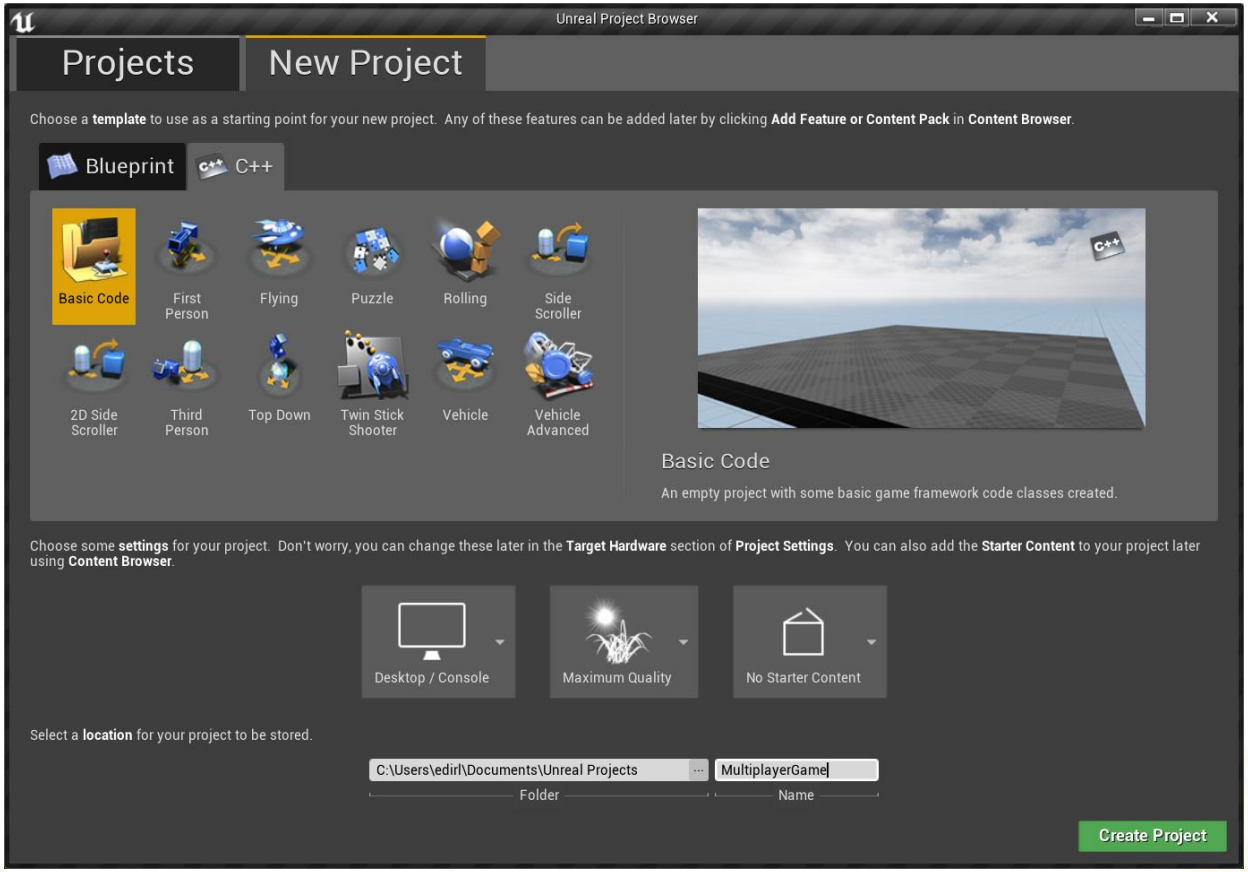
- **Concept:** 3rd person shooter multiplayer game.

- **Gameplay elements:**
  - Player Character: can walk, jump, crunch, and shoot;
  - Weapon: used by the player character to shoot other players;
  - Player Health: when hit by a shot, the player character losses a specific amount of health points;
  - Death: after losing all health points, the player character dies;



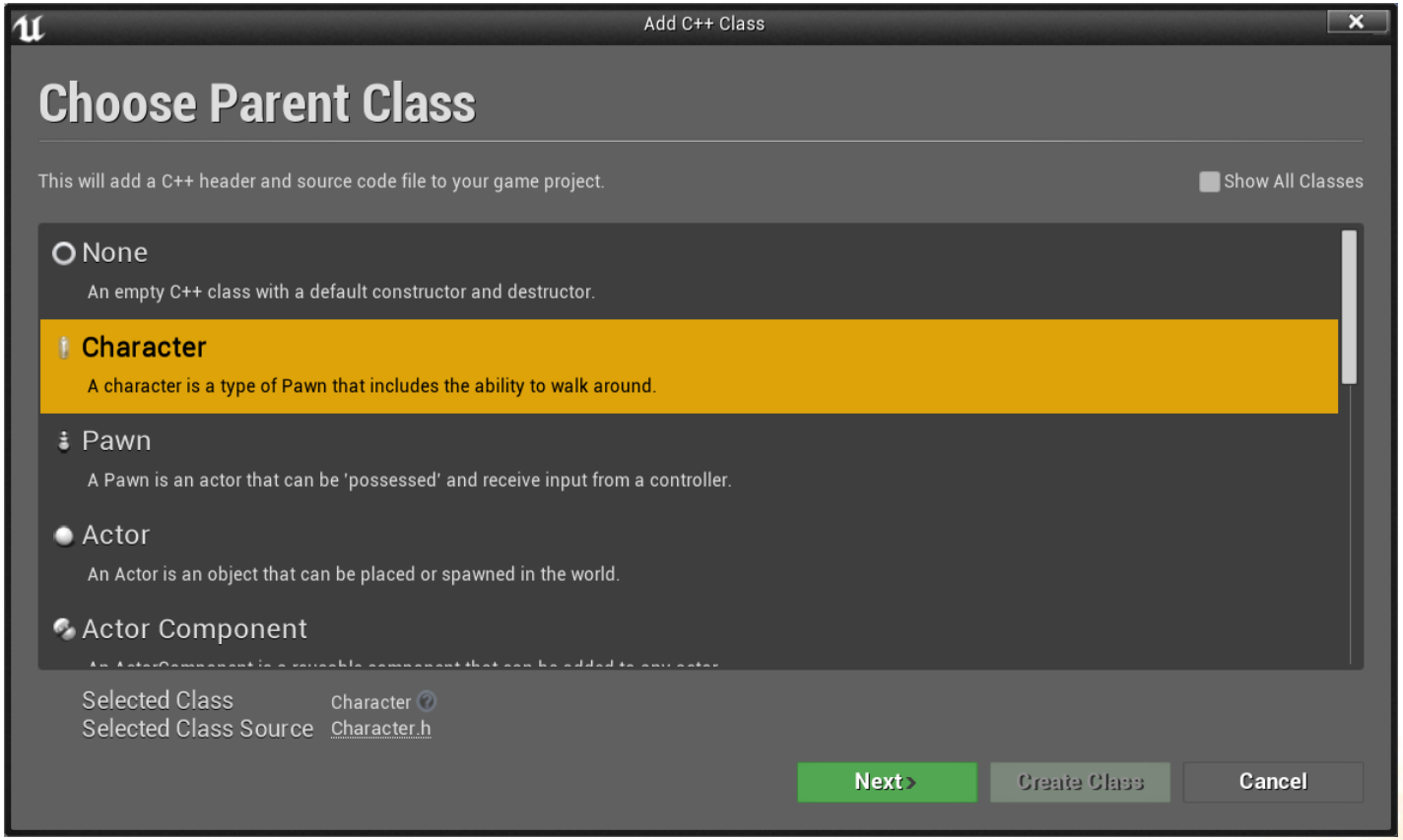
Multiplayer Game

- Create a new C++ empty project:



Multiplayer Game

- Create a new character class:



Multiplayer Game

- Implement the character movement:

MyCharacter.h

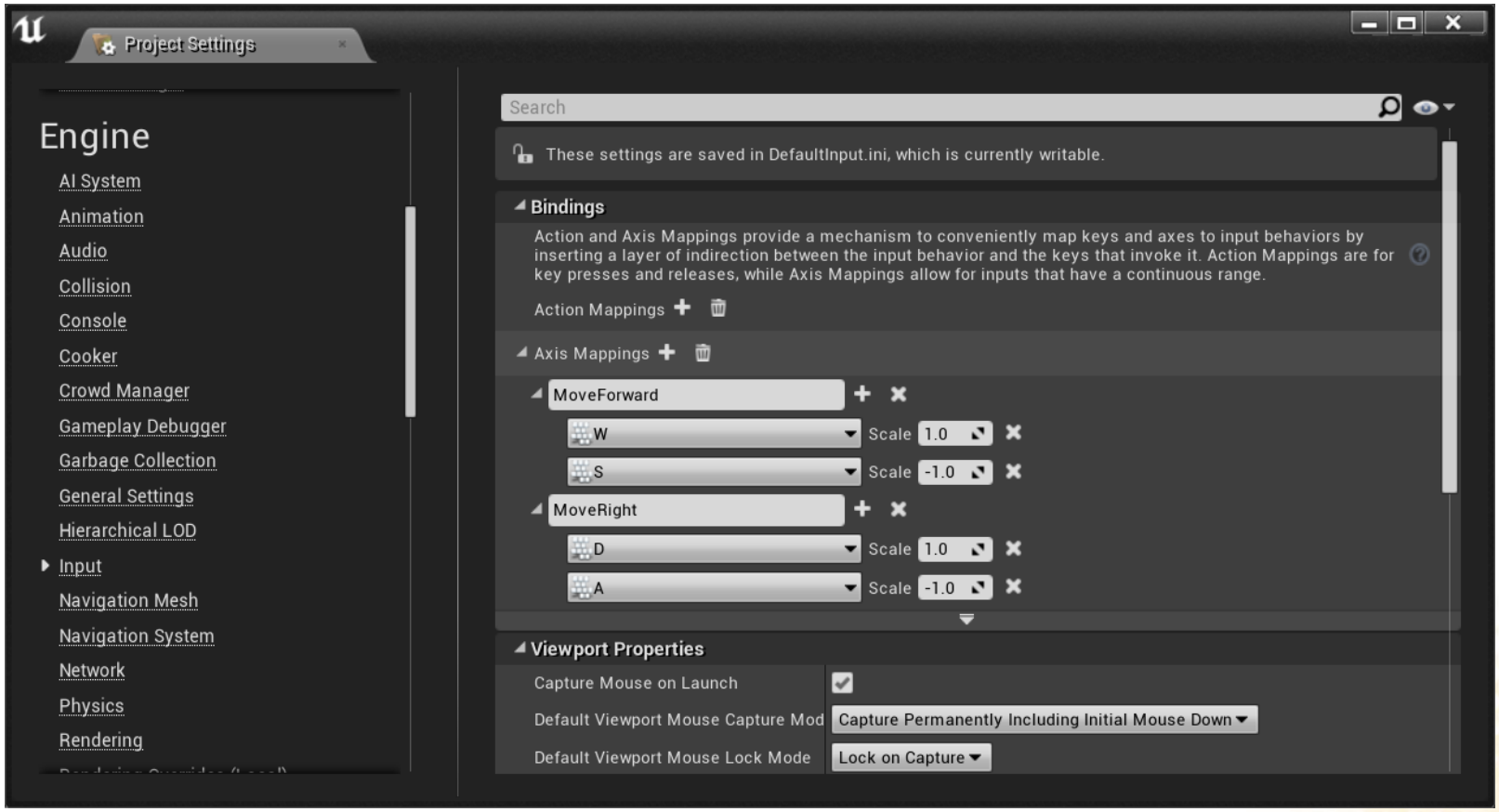
```
protected:
  void MoveForward(float value);
  void MoveRight(float value);
```

MyCharacter.cpp

```
void AMyCharacter::MoveForward(float value){
  AddMovementInput(GetActorForwardVector(), value);
}
void AMyCharacter::MoveRight(float value){
  AddMovementInput(GetActorRightVector(), value);
}
void AMyCharacter::SetupPlayerInputComponent(UInputComponent*
                                             PlayerInputComponent){
  Super::SetupPlayerInputComponent(PlayerInputComponent);
  PlayerInputComponent->BindAxis("MoveForward", this,
                                 &AMyCharacter::MoveForward);
  PlayerInputComponent->BindAxis("MoveRight", this,
                                 &AMyCharacter::MoveRight);
}
```

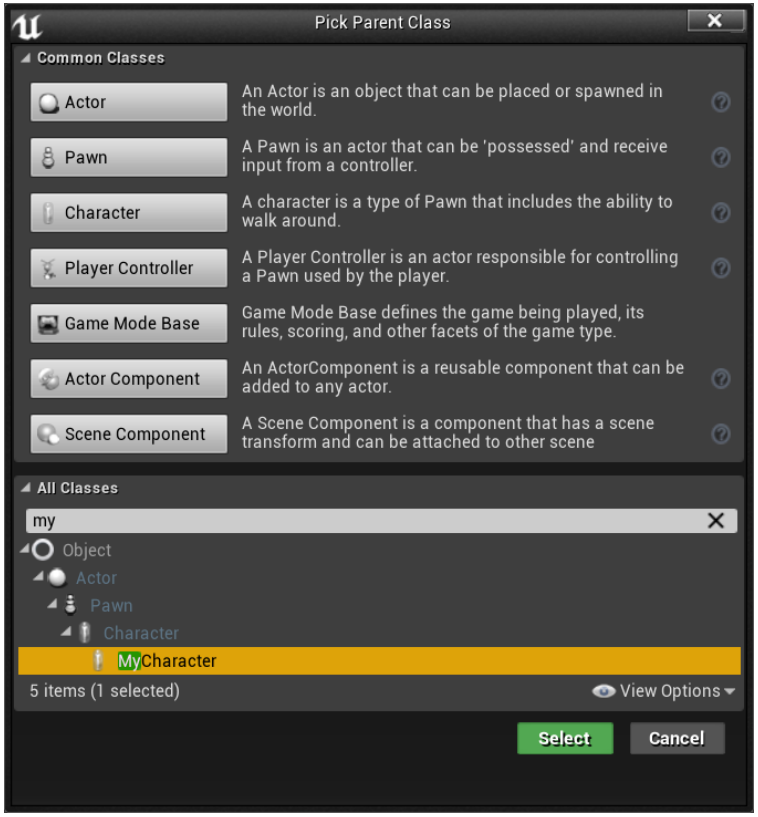
Multiplayer Game

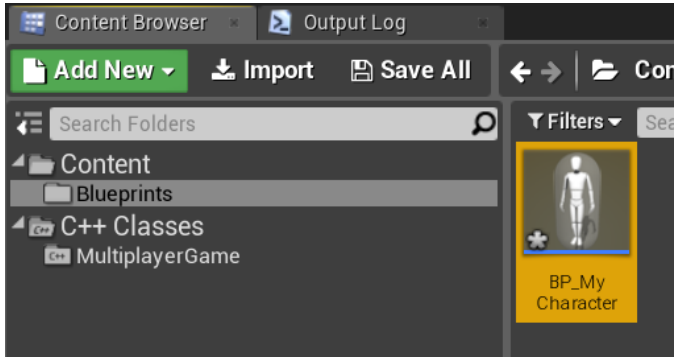
- Setup the axis keys in the project settings:

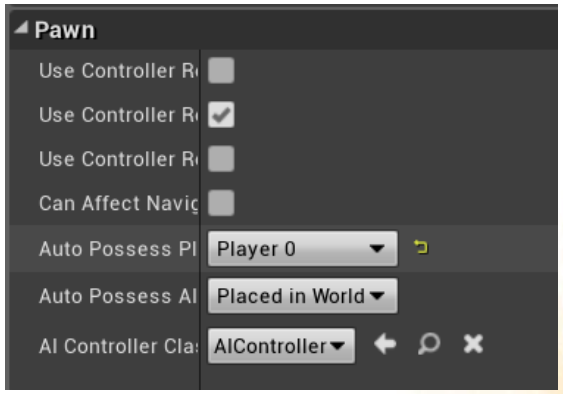


Multiplayer Game

- Create a blueprint for the character class and test the player movement in the level:







Multiplayer Game

- Implement the camera movement and setup the axis keys:

MyCharacter.cpp

```
void AMyCharacter::SetupPlayerInputComponent(UInputComponent*
                                                PlayerInputComponent){
  ...

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

Multiplayer Game

- Create a 3rd person camera and a spring arm components in the character class:

MyCharacter.h

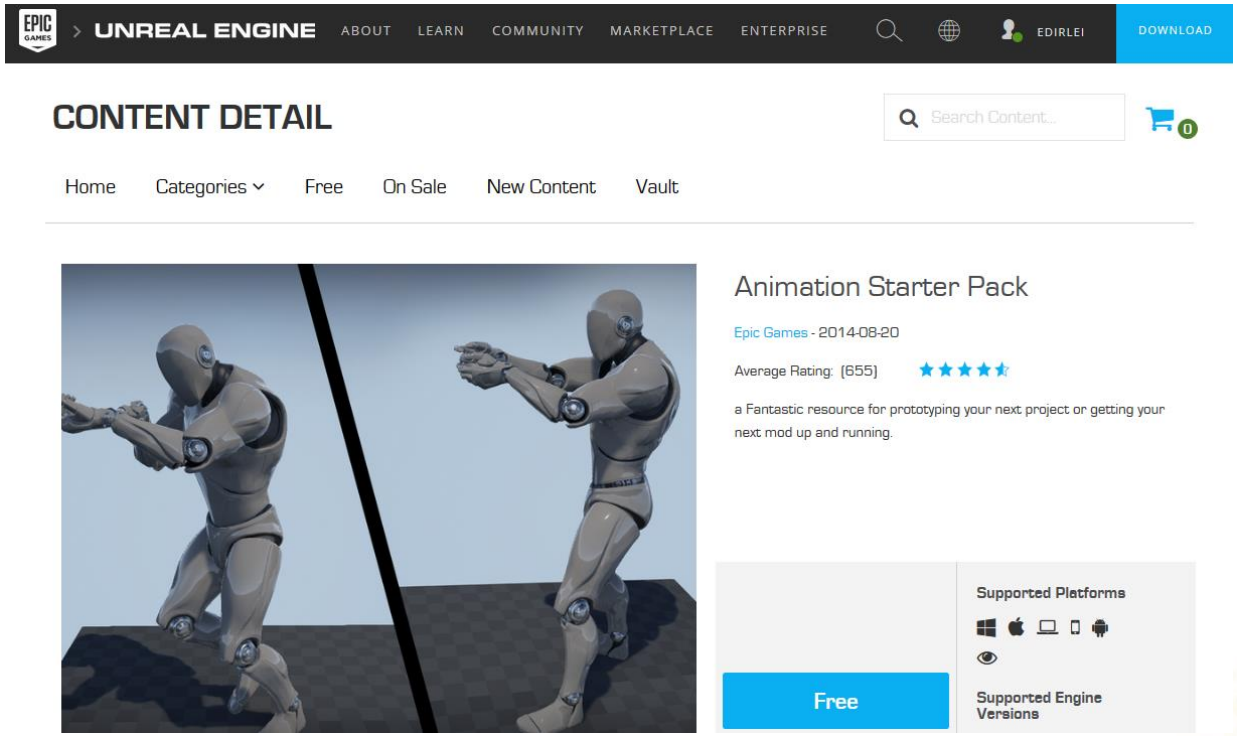
```
protected:
  UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Components")
  class UCameraComponent * CameraComponent;

  UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Components")
  class USpringArmComponent * SpringArmComponent;
```

MyCharacter.cpp

```
AMyCharacter::AMyCharacter(){
  PrimaryActorTick.bCanEverTick = true;
  SpringArmComponent = CreateDefaultSubobject<USpringArmComponent>
                                          ("SpringArm Component");
  SpringArmComponent->bUsePawnControlRotation = true;
  SpringArmComponent->SetupAttachment(RootComponent);

  CameraComponent = CreateDefaultSubobject<UCameraComponent>
                                        ("Camera Component");
  CameraComponent->SetupAttachment(SpringArmComponent);
}
```

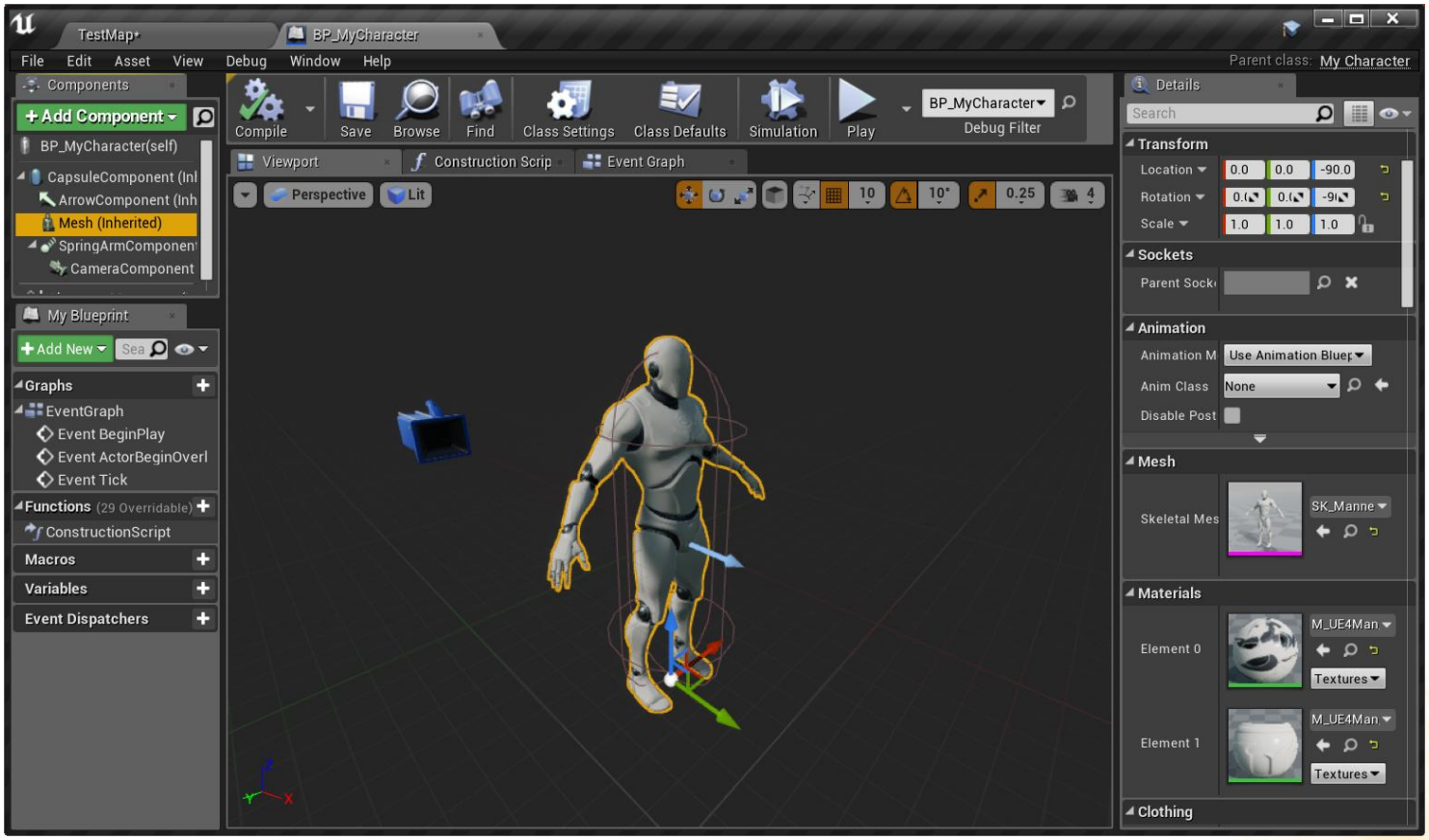
Multiplayer Game

- Download and import the Animation Starter Pack:
  - https://www.unrealengine.com/marketplace/animation-starter-pack



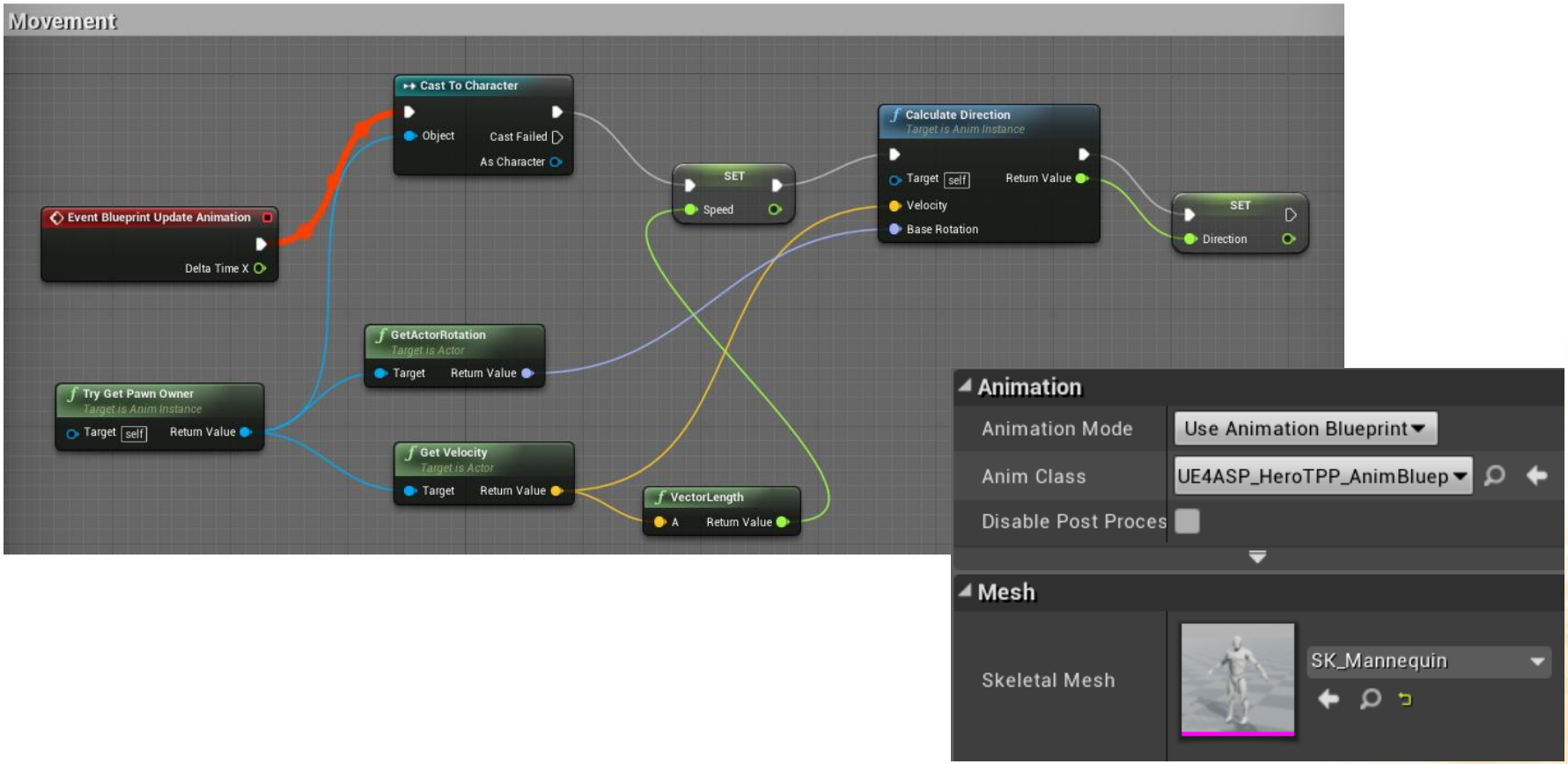
Multiplayer Game

- Add and setup the model mesh in the character blueprint:



Multiplayer Game

- Clear the animation blueprint (UE4ASP_HeroTPP_AnimBlueprint) and setup the animation in the character blueprint.



Multiplayer Game

- Implement the crouch action:

MyCharacter.h

```
protected:
  ...
  void BeginCrouch();
  void EndCrouch();
```

MyCharacter.cpp

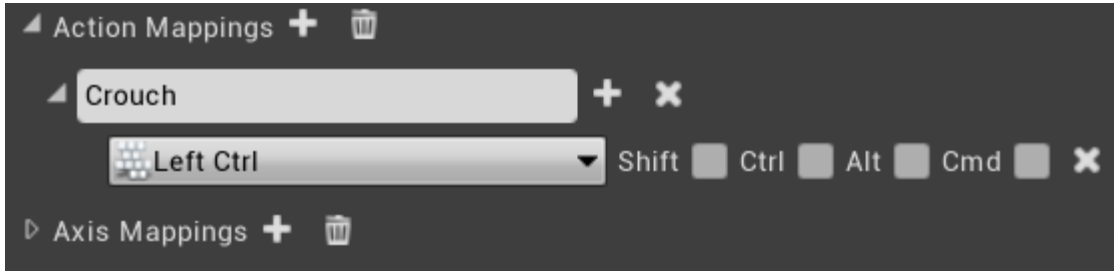
```
void AMyCharacter::BeginCrouch(){
  Crouch();
}

void AMyCharacter::EndCrouch(){
  UnCrouch();
}

AMyCharacter::AMyCharacter(){
  ...
  GetMovementComponent()->GetNavAgentPropertiesRef().bCanCrouch = true;
}
```

Multiplayer Game

- Implement the crouch action:

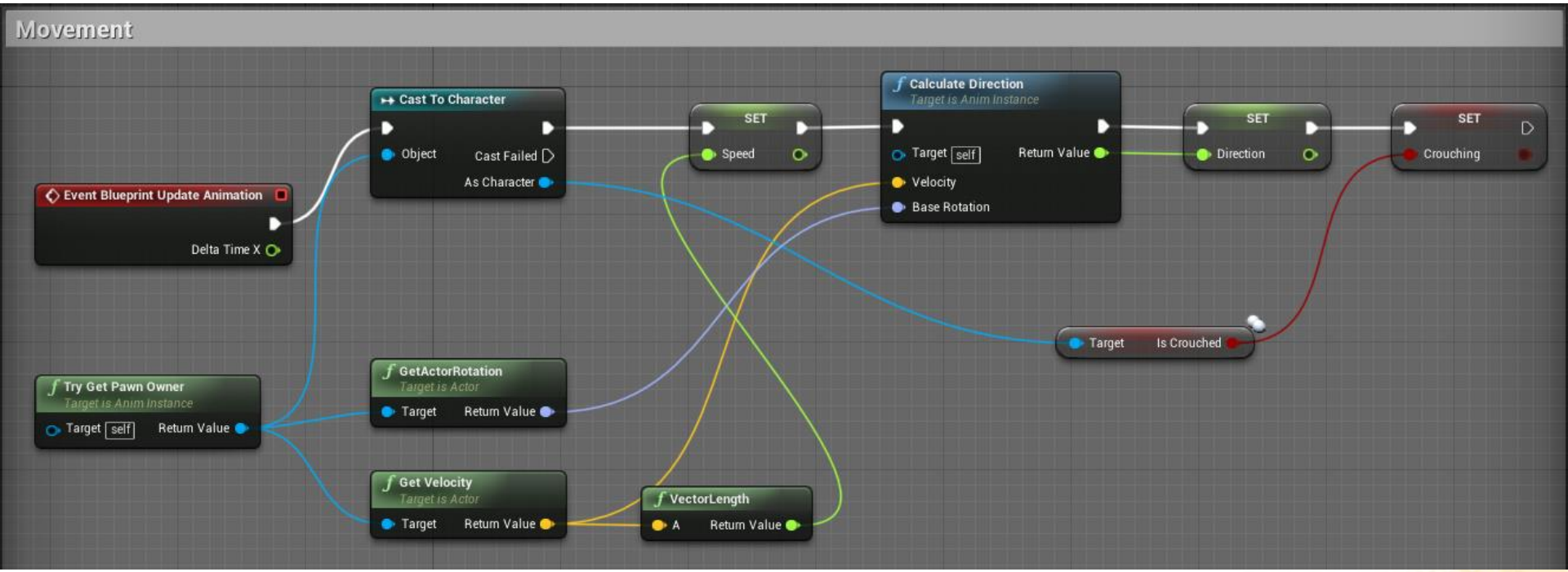
MyCharacter.cpp

```
void AMyCharacter::SetupPlayerInputComponent(UInputComponent*
                                             PlayerInputComponent){
  ...

  PlayerInputComponent->BindAction("Crouch", IE_Pressed, this,
                                   &AMyCharacter::BeginCrouch);
  PlayerInputComponent->BindAction("Crouch", IE_Released, this,
                                   &AMyCharacter::EndCrouch);
}
```



Multiplayer Game

- Set the crouch variable in the animation blueprint:

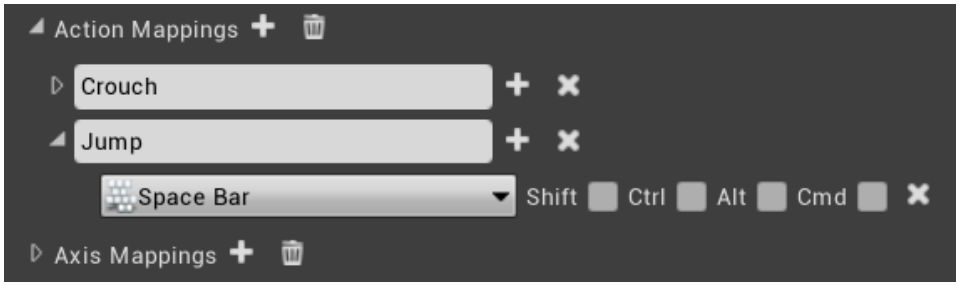


Multiplayer Game

- Implement the jump action:

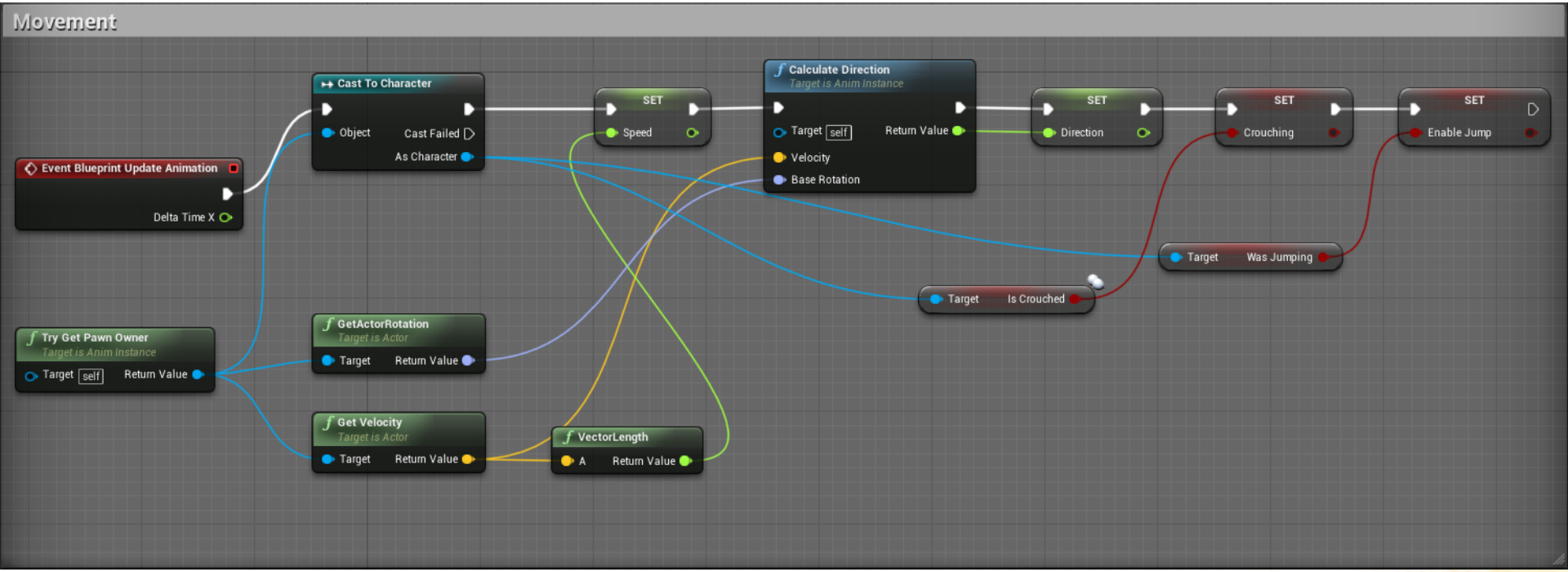
MyCharacter.cpp

```
void AMyCharacter::SetupPlayerInputComponent(UInputComponent*
                                                PlayerInputComponent){
  ...

  PlayerInputComponent->BindAction("Jump", IE_Pressed, this,
                                   &AMyCharacter::Jump);
}
```



Multiplayer Game

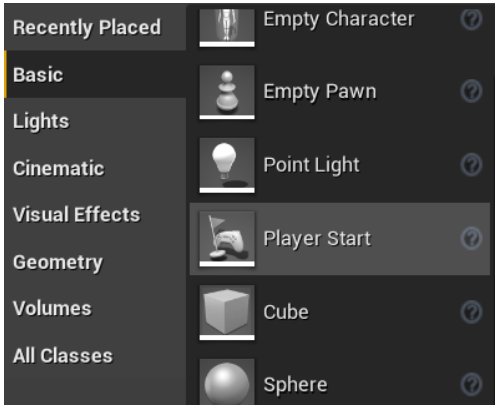
- Implement the jump action:



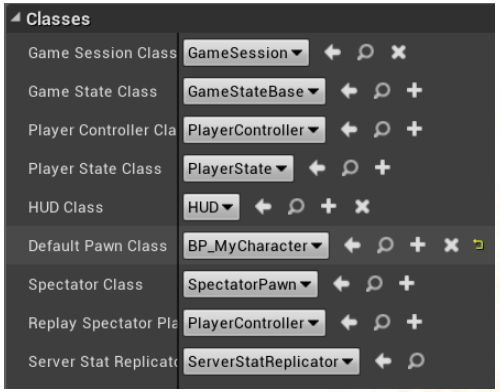
Multiplayer Game

- Setup the game to be played in multiplayer:

1. Delete the character from the map and add two or more “Player Start” actors to the map.



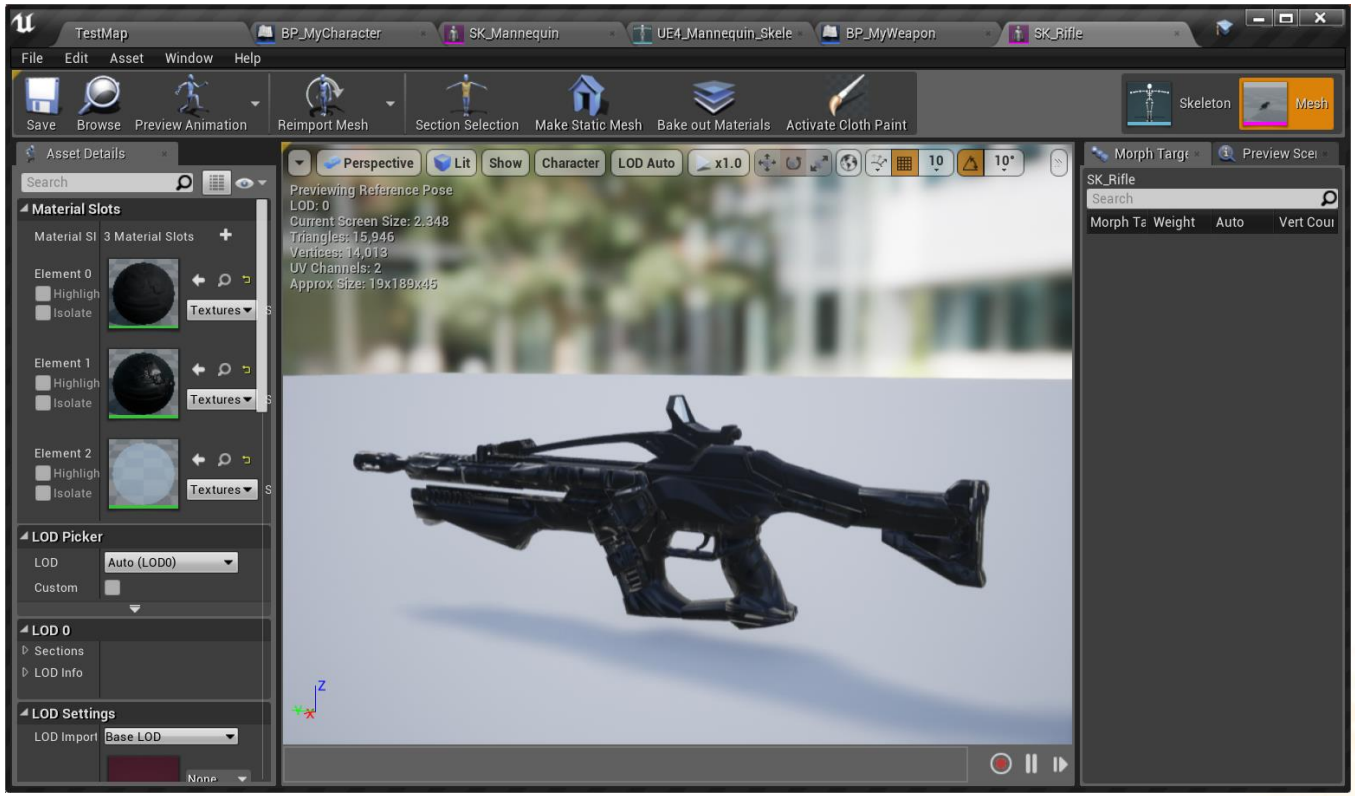
2. Create a new Game Mode blueprint and set our character blueprint as default pawn class.



3. Set the new Game Mode as the Game Mode for the map in the World Settings.

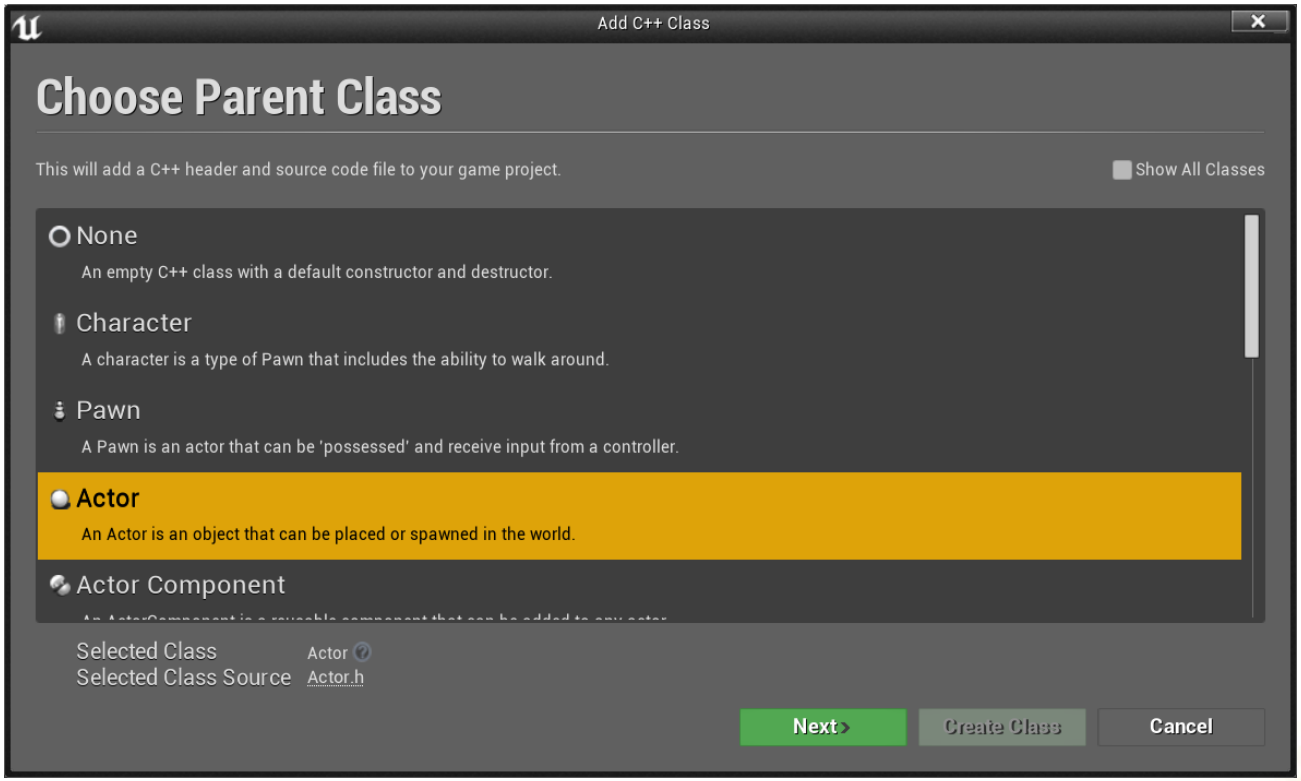
Multiplayer Game

- Add a weapon: import the weapon mesh
  - http://www.inf.puc-rio.br/~elima/dp/weapon.zip



Multiplayer Game

- Create a C++ class for the weapon:



Multiplayer Game

- Implement the crouch action:

MyWeapon.h

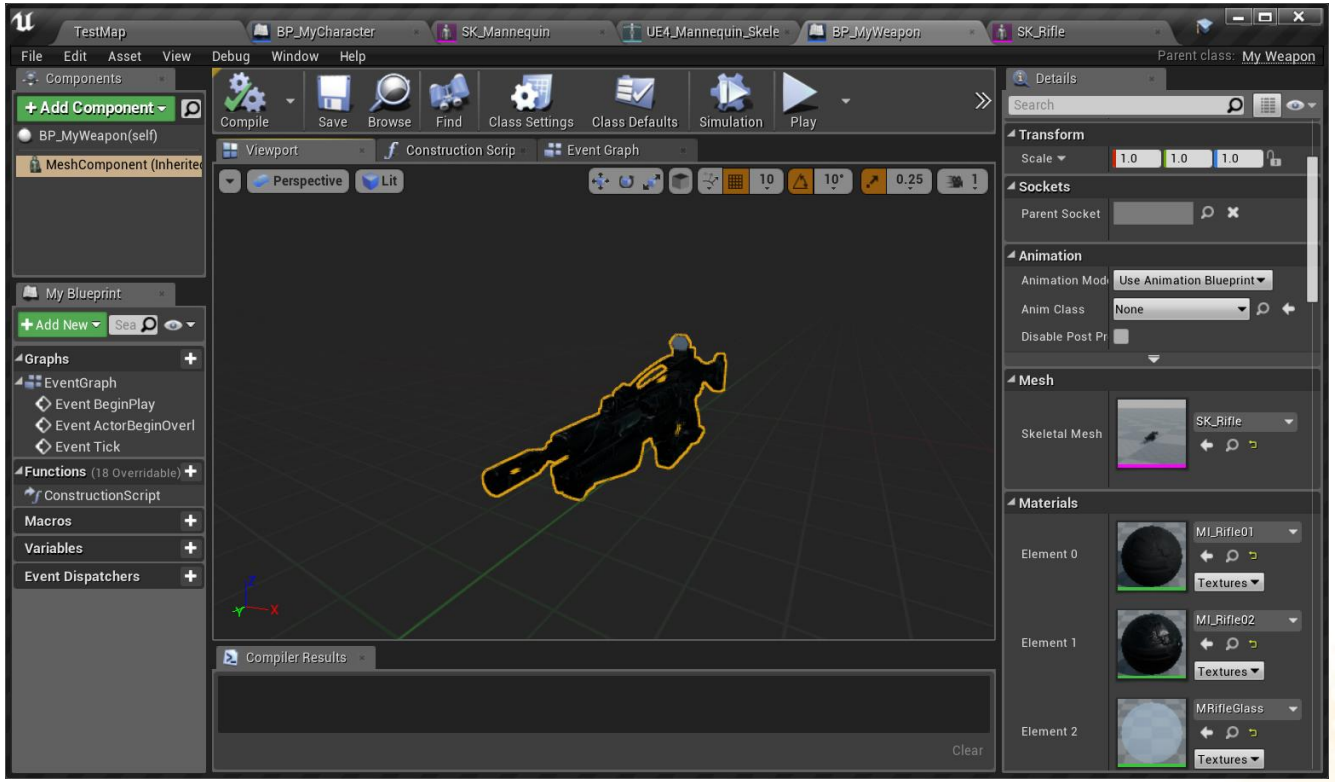
```
protected:

  UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Components")
  class USkeletalMeshComponent* MeshComponent;
```

MyWeapon.cpp

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

  MeshComponent = CreateDefaultSubobject<USkeletalMeshComponent>
                                        ("Mesh Component");
  RootComponent = MeshComponent;
}
```

Multiplayer Game

- Create a blueprint for the weapon class and select the weapon mesh:

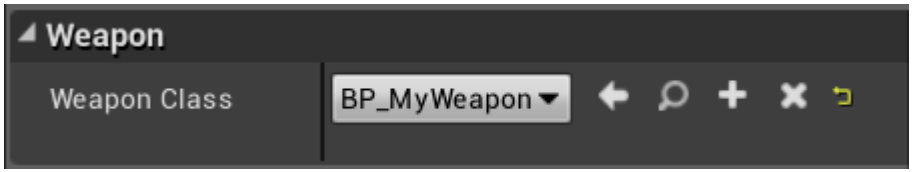


Multiplayer Game

- Spawn the weapon and attach it to the character hand in the BeginPlay event of the character blueprint:

MyCharacter.h

```
protected:
  class AMyWeapon* CurrentWeapon;

  UPROPERTY(EditDefaultsOnly, Category = "Weapon")
  TSubclassOf<AMyWeapon> WeaponClass;
```



Multiplayer Game

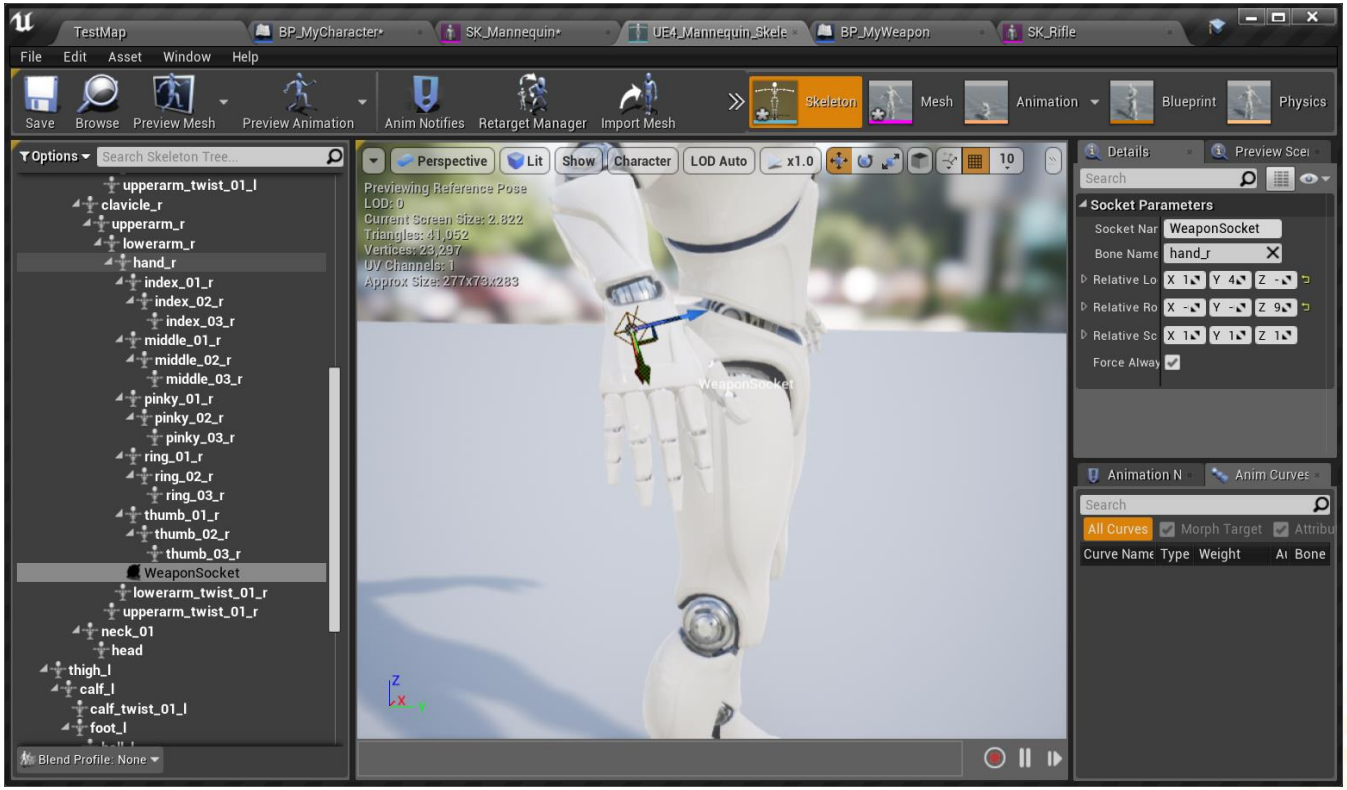
- Spawn the weapon and attach it to the character hand in the BeginPlay event of the character blueprint:

MyCharacter.cpp

```
void AMyCharacter::BeginPlay(){
  Super::BeginPlay();
  if (WeaponClass){
    FActorSpawnParameters parameters;
    parameters.SpawnCollisionHandlingOverride =
                        ESpawnActorCollisionHandlingMethod::AlwaysSpawn;
    CurrentWeapon = GetWorld()->SpawnActor<AMyWeapon>(WeaponClass,
                            FVector::ZeroVector, FRotator::ZeroRotator,
                            parameters);
    if (CurrentWeapon){
      CurrentWeapon->AttachToComponent(GetMesh(),
              FAttachmentTransformRules::SnapToTargetNotIncludingScale,
              "WeaponSocket");
      CurrentWeapon->SetOwner(this);
    }
  }
}
```

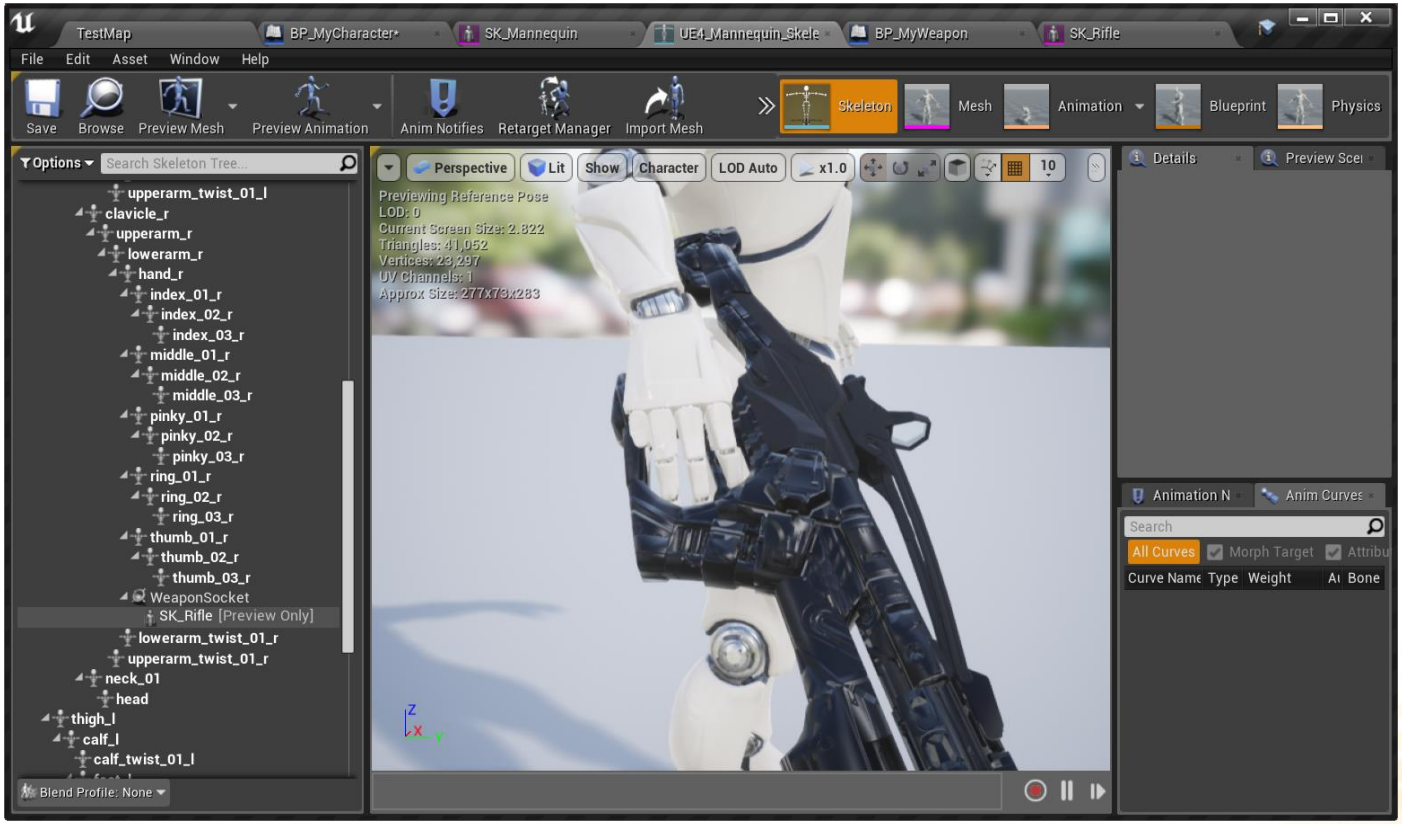
Multiplayer Game

- Create a socket (“WeaponSocket”) in the skeleton of the model to correctly attach the weapon to the character’s hand:



Multiplayer Game

- Add the weapon as a “Preview Asset” into the socket and adjust its position to the character’s hand:



Multiplayer Game

- Implement the code to shoot with the weapon:

MyWeapon.h

```
protected:

  ...

  UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category = "Weapon")
  TSubclassOf<UDamageType> DamageType;

  ...

public:

  ...

  void Fire();
```

Multiplayer Game

- Implement the code to shoot with the weapon:

MyWeapon.cpp

```
void AMyWeapon::Fire()
{
  AActor* weaponOwner = GetOwner();
  if (weaponOwner)
  {
    FVector eyeLocation;
    FRotator eyeRotation;
    weaponOwner->GetActorEyesViewPoint(eyeLocation, eyeRotation);

    FVector endPoint = eyeLocation + (eyeRotation.Vector() * 1000);

    FCollisionQueryParams cparams;
    cparams.AddIgnoredActor(weaponOwner);
    cparams.AddIgnoredActor(this);
    cparams.bTraceComplex = true;

    ...
```

Multiplayer Game

- Implement the code to shoot with the weapon:

MyWeapon.cpp

```
    ...

    FHitResult hit;
    if (GetWorld()->LineTraceSingleByChannel(hit, eyeLocation, endPoint,
                                             ECC_Visibility, cparams)){
      AActor* hitActor = hit.GetActor();
      UGameplayStatics::ApplyPointDamage(hitActor, 10.0f,
                                 eyeRotation.Vector(), hit,
                                 weaponOwner->GetInstigatorController(),
                                 this, DamageType);
    }
    DrawDebugLine(GetWorld(), eyeLocation, endPoint, FColor::Red,
                  false, 1.0f, 0, 1);
  }
}
```

Multiplayer Game

- Implement the code to shoot with the weapon:

MyCharacter.h

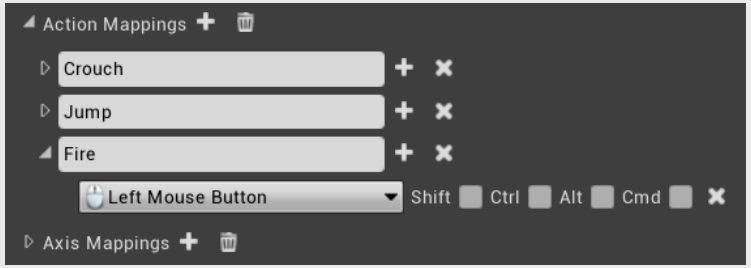
```
protected:

  ...

  void Fire();

  ...

public:

  ...

  virtual FVector GetPawnViewLocation() const override;
```

Multiplayer Game

- Implement the code to shoot with the weapon:

MyCharacter.cpp

```
void AMyCharacter::Fire()
{
  if (CurrentWeapon)
  {
    CurrentWeapon->Fire();
  }
}

void AMyCharacter::SetupPlayerInputComponent(UInputComponent*
                                             PlayerInputComponent){
  ...

  PlayerInputComponent->BindAction("Fire", IE_Pressed, this,
                                   &AMyCharacter::Fire);
}
```



Multiplayer Game

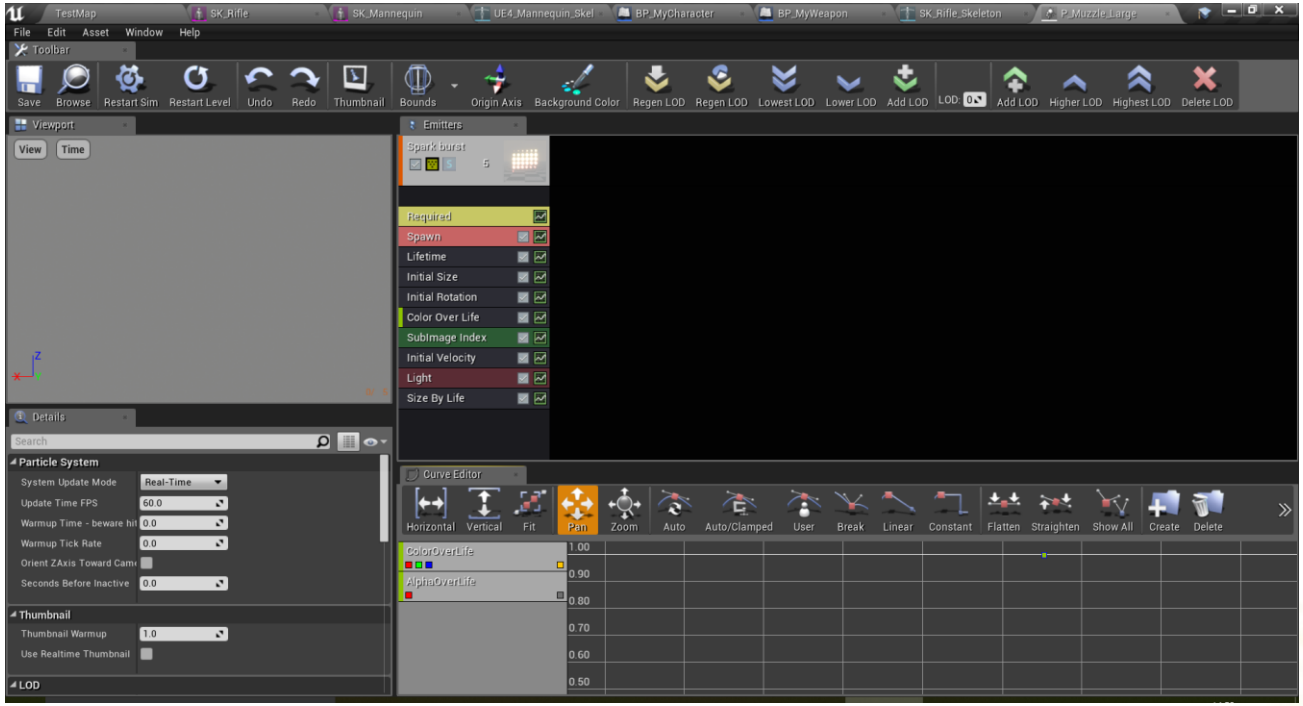
- Implement the code to shoot with the weapon:

MyCharacter.cpp

```
FVector AMyCharacter::GetPawnViewLocation() const
{
  if (CameraComponent)
  {
    return CameraComponent->GetComponentLocation();
  }
  return Super::GetPawnViewLocation();
}
```

Multiplayer Game

- Add particle effects to the weapon: import the particle effects
  - http://www.inf.puc-rio.br/~elima/dp/weapon_effects.zip



Multiplayer Game

- Implement the code spawn the particle systems:

MyWeapon.h

```
protected:

  ...

  UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category = "Weapon")
  UParticleSystem* FireEffect;

  UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category = "Weapon")
  UParticleSystem* ImpactEffect;

  UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category = "Weapon")
  UParticleSystem* TraceEffect;

  void PlayFireEffects(FVector endPoint);

  void PlayImpactEffects(FVector impactPoint);

  ...
```

Multiplayer Game

- Implement the code spawn the particle systems:

MyWeapon.cpp

```
void AMyWeapon::PlayFireEffects(FVector endPoint)
{
  if (FireEffect)
  {
    UGameplayStatics::SpawnEmitterAttached(FireEffect, MeshComponent,
                                           "MuzzleFlashSocket");
  }
  if (TraceEffect)
  {
    UParticleSystemComponent* instTrace =
       UGameplayStatics::SpawnEmitterAtLocation(GetWorld(), TraceEffect,
       MeshComponent->GetSocketLocation("MuzzleFlashSocket"));
    instTrace->SetVectorParameter("Target", endPoint);
  }
}
```

Multiplayer Game

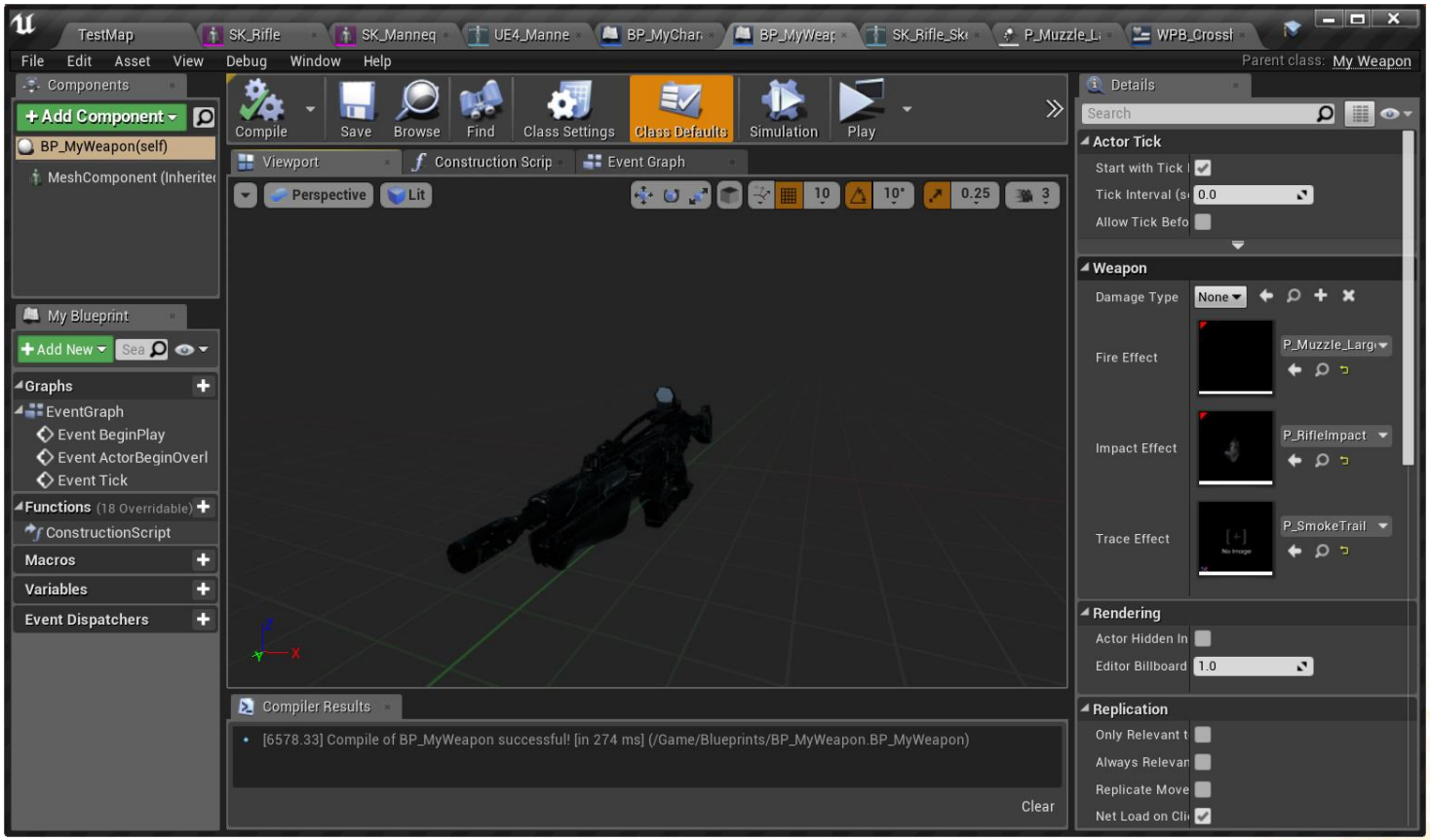
- Implement the code spawn the particle systems:

MyWeapon.cpp

```
void AMyWeapon::Fire(){

  ...

  FHitResult hit;
  if (GetWorld()->LineTraceSingleByChannel(hit, eyeLocation, endPoint,
                                           ECC_Visibility, cparams)){
    ...

    PlayImpactEffects(hit.ImpactPoint);
    endPoint = hit.ImpactPoint;
  }

  PlayFireEffects(endPoint);
}
```

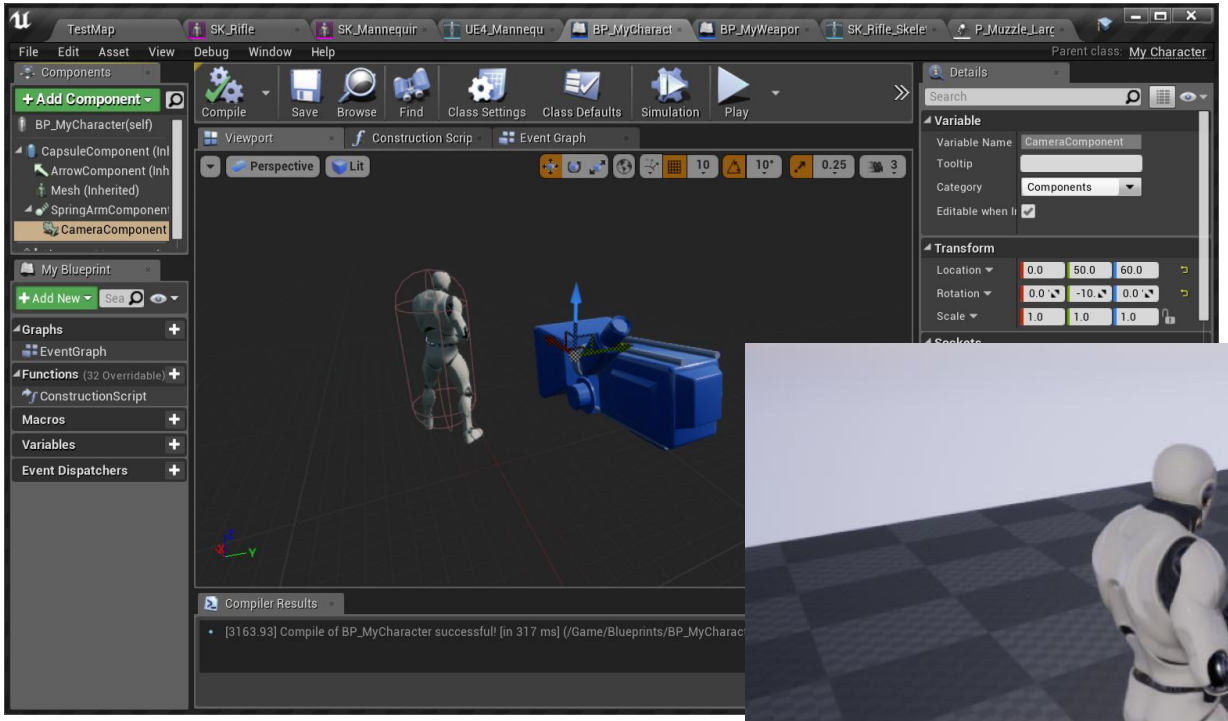
Multiplayer Game

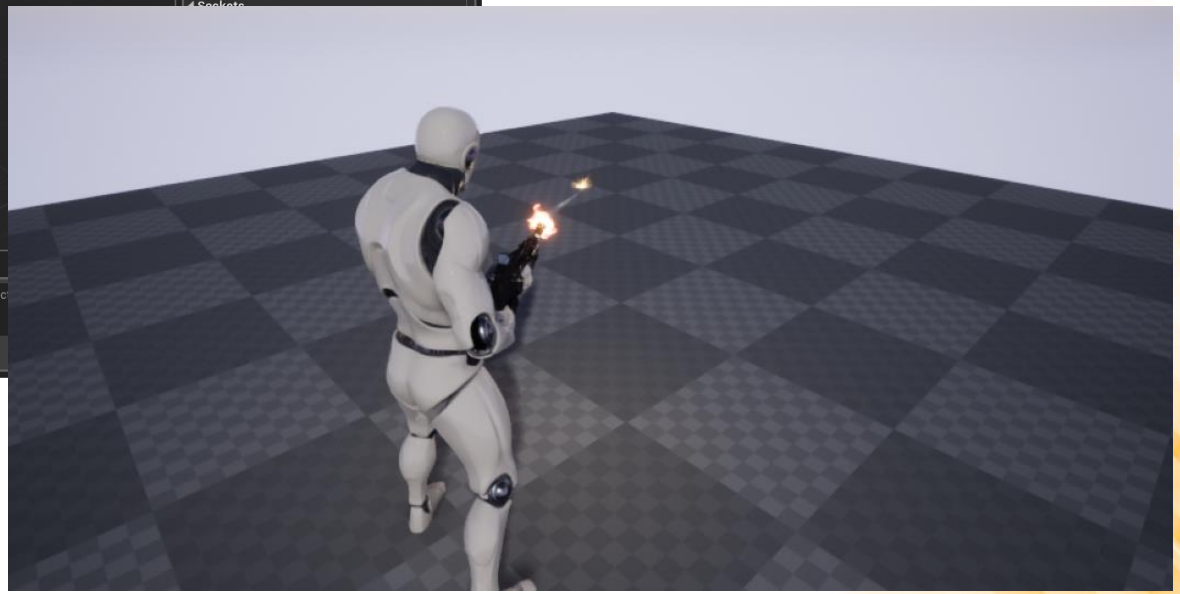
- Setup the weapon effects:



Multiplayer Game

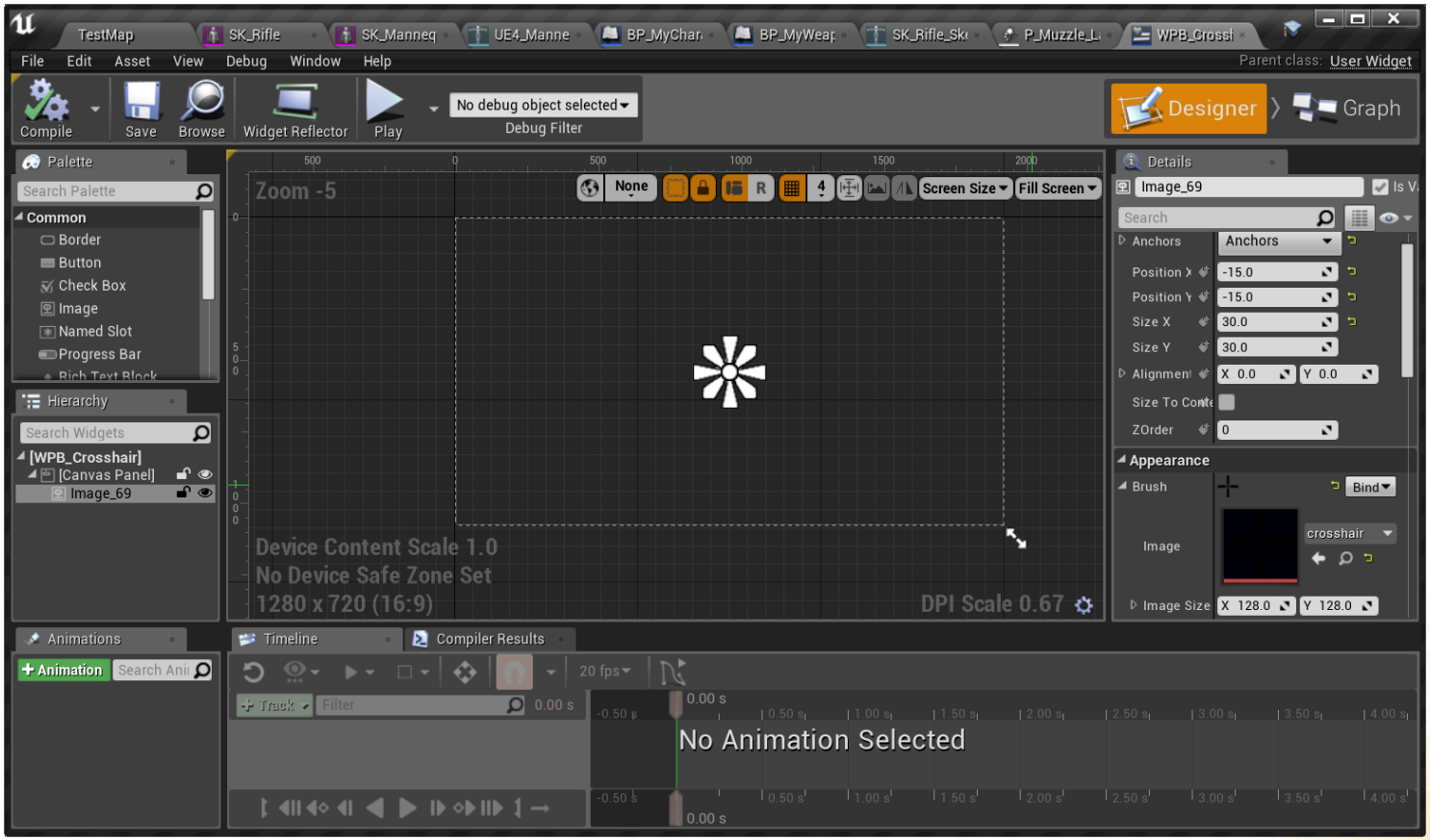
- Adjust the camera position and rotation in the character blueprint for a better view:





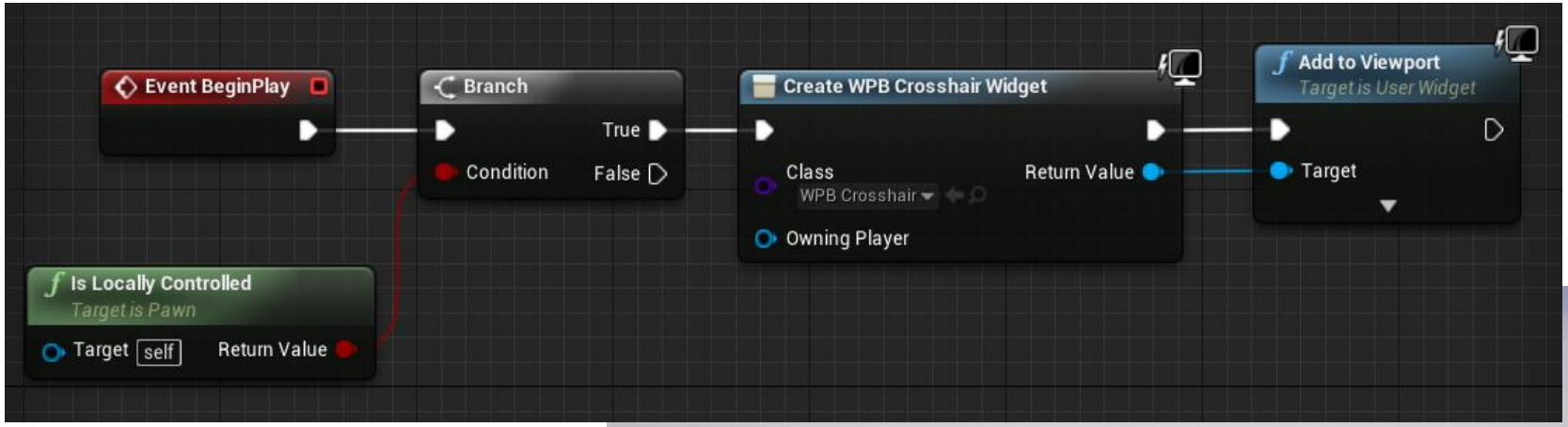
Multiplayer Game

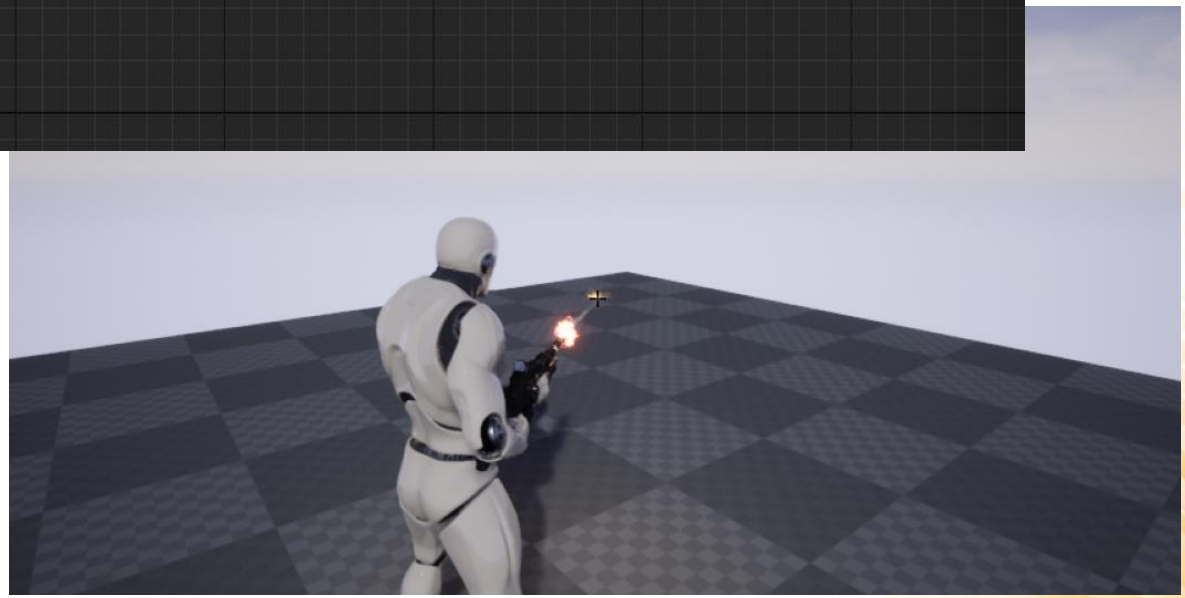
- Add a crosshair to the center of the screen: create a widget blueprint with a crosshair image at the center.



Multiplayer Game

- Add a crosshair to the center of the screen: instantiate the widget in the BeginPlay event of the character.





Multiplayer Game

- **Setup the weapons for multiplayer:** only the server should create the weapons, but currently the weapons are being created in the BeginPlay event, which also runs on clients.

MyCharacter.cpp

```
void AMyCharacter::BeginPlay(){
  Super::BeginPlay();
  if ((Role == ROLE_Authority)&&(WeaponClass)){
    FActorSpawnParameters parameters;
    parameters.SpawnCollisionHandlingOverride =
                        ESpawnActorCollisionHandlingMethod::AlwaysSpawn;
    CurrentWeapon = GetWorld()->SpawnActor<AMyWeapon>(WeaponClass,
                 FVector::ZeroVector, FRotator::ZeroRotator, parameters);
    if (CurrentWeapon){
      CurrentWeapon->AttachToComponent(GetMesh(),
                FAttachmentTransformRules::SnapToTargetNotIncludingScale,
                "WeaponSocket");
      CurrentWeapon->SetOwner(this);
    }
  }
}
```

Multiplayer Game

- Setup the weapons for multiplayer:

MyWeapon.cpp

```
AMyWeapon::AMyWeapon()
{
  ...

  SetReplicates(true);
}
```

MyCharacter.h

```
protected:

  ...

  UPROPERTY(Replicated)
  class AMyWeapon* CurrentWeapon;

  ...
```

Multiplayer Game

- Setup the weapons for multiplayer:

MyCharacter.cpp

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

...

void AEnemyCharacter::GetLifetimeReplicatedProps(
                    TArray<FLifetimeProperty>& OutLifetimeProps) const
{
  Super::GetLifetimeReplicatedProps(OutLifetimeProps);
  DOREPLIFETIME(AEnemyCharacter, isChasing);
}
```

Multiplayer Game

- When a client fires the weapon, the Fire function must be called on the client and on the server:

MyWeapon.h

```
proteceted:
  UFUNCTION(Server, Reliable, WithValidation)
  void ServerFire();
```

MyWeapon.cpp

```
void AMyWeapon::ServerFire_Implementation()
{
  Fire();
}

bool AMyWeapon::ServerFire_Validate()
{
  return true;
}
```

Multiplayer Game

- When a client fires the weapon, the Fire function must be called on the client and on the server:

MyWeapon.cpp

```
void AMyWeapon::Fire()
{
  if (Role != ROLE_Authority)
  {
    ServerFire();
  }

  ...
```

Multiplayer Game

- The server must replicate the hit points to all clients (except to the one that fired):

MyWeapon.h

```
protected:

  ...

  UPROPERTY(ReplicatedUsing=OnReplicateHitPoint)
  FVector_NetQuantize HitPoint;

  UFUNCTION()
  void OnReplicateHitPoint();

  ...
```

Multiplayer Game

- The server must replicate the hit points to all clients (except to the one that fired):

MyWeapon.cpp

```
void AMyWeapon::GetLifetimeReplicatedProps(TArray<FLifetimeProperty>&
                                           OutLifetimeProps) const {
  Super::GetLifetimeReplicatedProps(OutLifetimeProps);
  DOREPLIFETIME_CONDITION(AMyWeapon, HitPoint, COND_SkipOwner);
}

void AMyWeapon::OnReplicateHitPoint()
{
  PlayFireEffects(HitPoint);
  PlayImpactEffects(HitPoint);
}
```

Multiplayer Game

- The server must replicate the hit points to all clients (except to the one that fired):

MyWeapon.cpp

```
void AMyWeapon::Fire(){
  if (Role != ROLE_Authority){
    ServerFire();
  }
  AActor* weaponOwner = GetOwner();
  if (weaponOwner){
    FVector eyeLocation;
    FRotator eyeRotation;
    weaponOwner->GetActorEyesViewPoint(eyeLocation, eyeRotation);

    FVector endPoint = eyeLocation + (eyeRotation.Vector() * 1000);
    FCollisionQueryParams cparams;
    cparams.AddIgnoredActor(weaponOwner);
    cparams.AddIgnoredActor(this);
    cparams.bTraceComplex = true;
    ...
```

Multiplayer Game

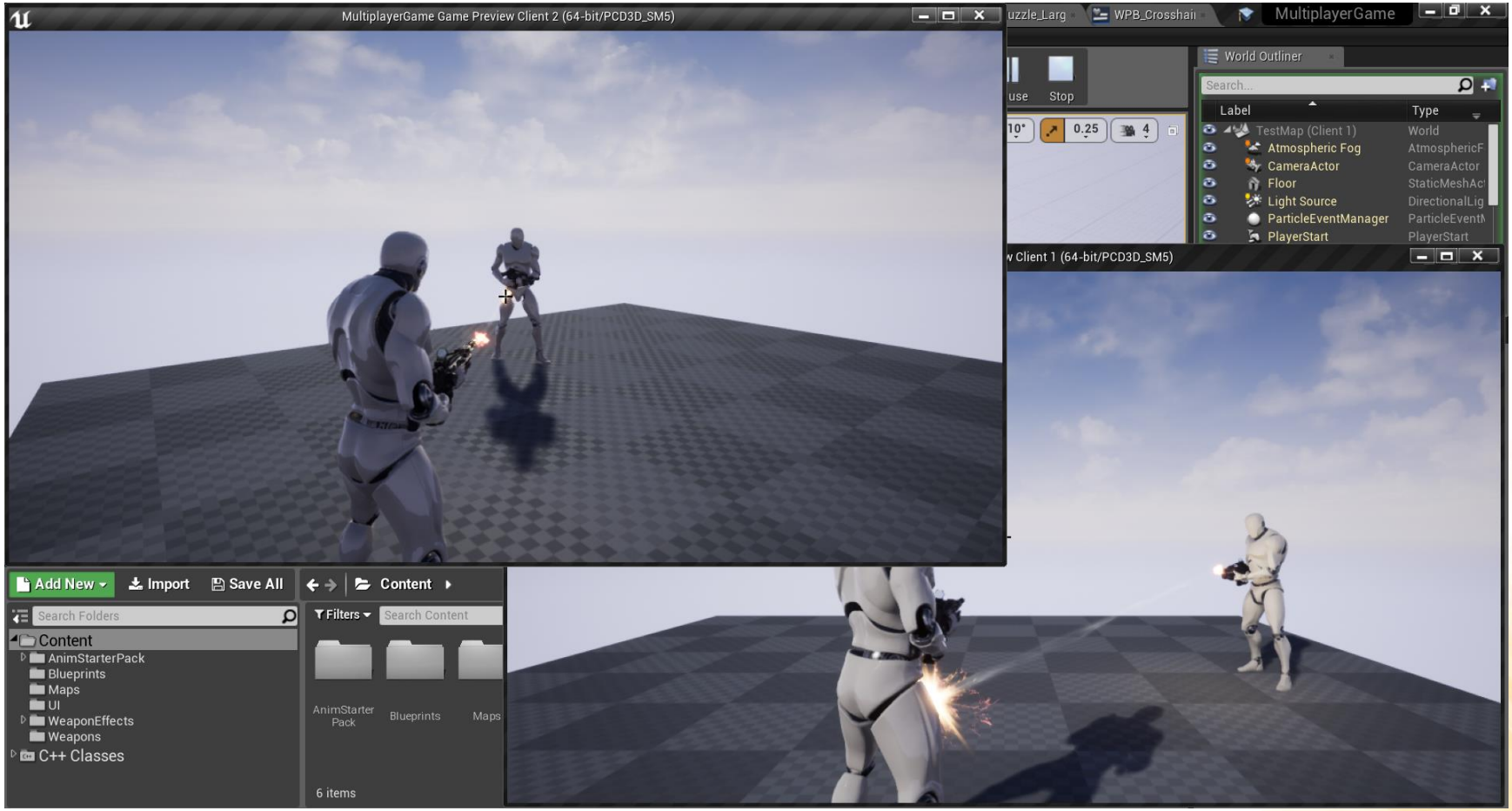
- The server must replicate the hit points to all clients (except to the one that fired):

MyWeapon.cpp

```
    ...
    FHitResult hit;
    if (GetWorld()->LineTraceSingleByChannel(hit, eyeLocation, endPoint,
                                             ECC_Visibility, cparams)){
      AActor* hitActor = hit.GetActor();
      UGameplayStatics::ApplyPointDamage(hitActor, 10.0f, eyeRotation.
                  Vector(), hit, weaponOwner->GetInstigatorController(),
                  this, DamageType);
      PlayImpactEffects(hit.ImpactPoint);
      endPoint = hit.ImpactPoint;
    }
    PlayFireEffects(endPoint);
    if (Role == ROLE_Authority){
      HitPoint = endPoint;
    }
  }
}
```

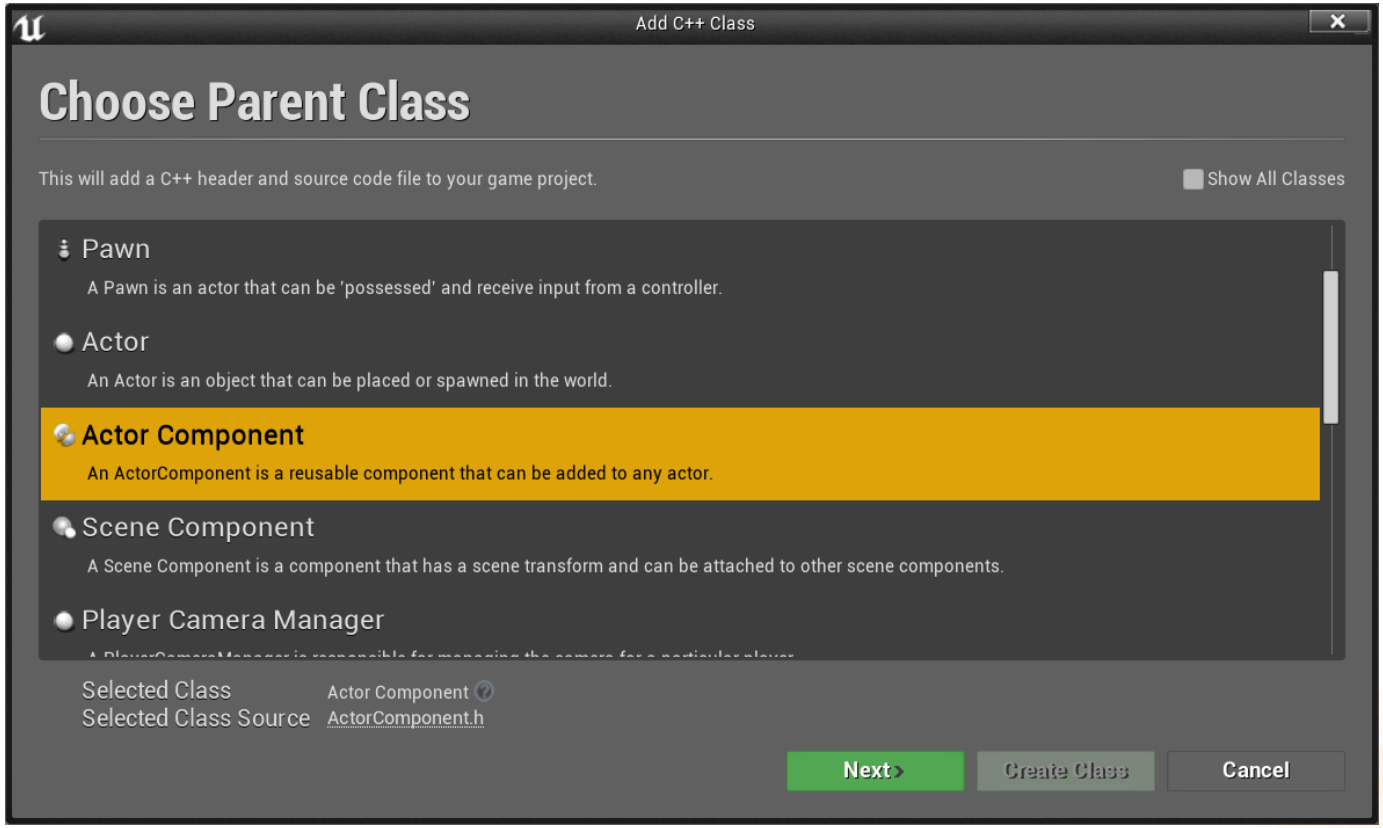
Multiplayer Game

- Test the game:



Multiplayer Game

- Create a new Component class (C++) to implement the Health Component:



Multiplayer Game

- Implement the Health Component:

MyHealthComponent.h

```
DECLARE_DYNAMIC_MULTICAST_DELEGATE_SixParams(FOnHealthChangedSignature,
            UMyHealthComponent*, HealthComponent, float, health, float,
            damage, const class UDamageType*, DamageType, class
            AController*, InstigatedBy, AActor*, DamageCauser);
...

protected:
  UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category = "Health")
  float Health;

  UFUNCTION()
  void OnTakeDamage(AActor* DamagedActor, float Damage, const class
          UDamageType* DamageType, class AController* InstigatedBy,
          AActor* DamageCauser);
  ...
public:
  UPROPERTY(BlueprintAssignable, Category = "Events")
  FOnHealthChangedSignature OnHealthChangedEvent;
```

Multiplayer Game

- Implement the Health Component:

MyHealthComponent.cpp

```
UMyHealthComponent::UMyHealthComponent(){
  Health = 100;
}
void UMyHealthComponent::BeginPlay(){
  Super::BeginPlay();
  AActor* myOwner = GetOwner();
  if (myOwner){
    myOwner->OnTakeAnyDamage.AddDynamic(this,
                                        &UMyHealthComponent::OnTakeDamage);
  }
}
void UMyHealthComponent::OnTakeDamage(AActor* DamagedActor, float
                Damage, const class UDamageType* DamageType, class
                AController* InstigatedBy, AActor* DamageCauser){
  Health = Health - Damage;
  if (Health < 0)
    Health = 0;
  OnHealthChangedEvent.Broadcast(this, Health, Damage, DamageType,
                                 InstigatedBy, DamageCauser);
}
```

Multiplayer Game

- Add the Health Component to the character:

MyCharacter.h

```
...

protected:
  UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category = "Player")
  bool IsDead;

  UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Components")
  class UMyHealthComponent * HealthComponent;

  ...

  UFUNCTION()
  void OnHealthChanged(class UMyHealthComponent* healthComp,
              float health, float damage, const class UDamageType*
              DamageType, class AController* InstigatedBy, AActor*
              DamageCauser);
  ...
```

Multiplayer Game

- Add the Health Component to the character:

MyCharacter.cpp

```
AMyCharacter::AMyCharacter()
{
  ...
  HealthComponent = CreateDefaultSubobject<UMyHealthComponent>
                                                    ("Health Component");
  HealthComponent->OnHealthChangedEvent.AddDynamic(this,
                                          &AMyCharacter::OnHealthChanged);
  ...
}

void AMyCharacter::BeginPlay()
{
  Super::BeginPlay();
  IsDead = false;

  ...
}
```

Multiplayer Game

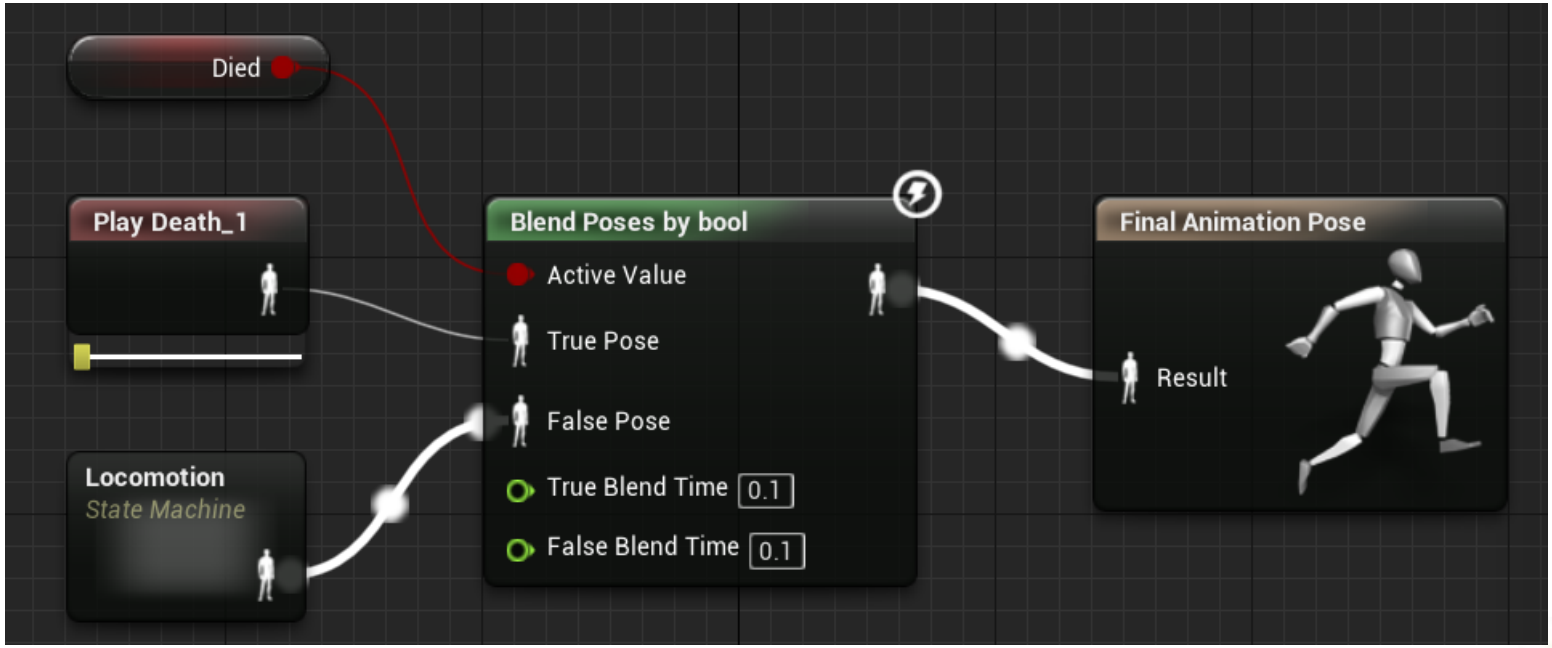
- Add the Health Component to the character:

MyCharacter.cpp

```
void AMyCharacter::OnHealthChanged(UMyHealthComponent* HealthComponent,
                float health, float damage, const class UDamageType*
                DamageType, class AController* InstigatedBy, AActor*
                DamageCauser){
  if ((health <= 0)&&(!IsDead))
  {
    IsDead = true;
    GetMovementComponent()->StopMovementImmediately();
    GetCapsuleComponent()->SetCollisionEnabled(ECollisionEnabled::
                                                NoCollision);
  }
}
```

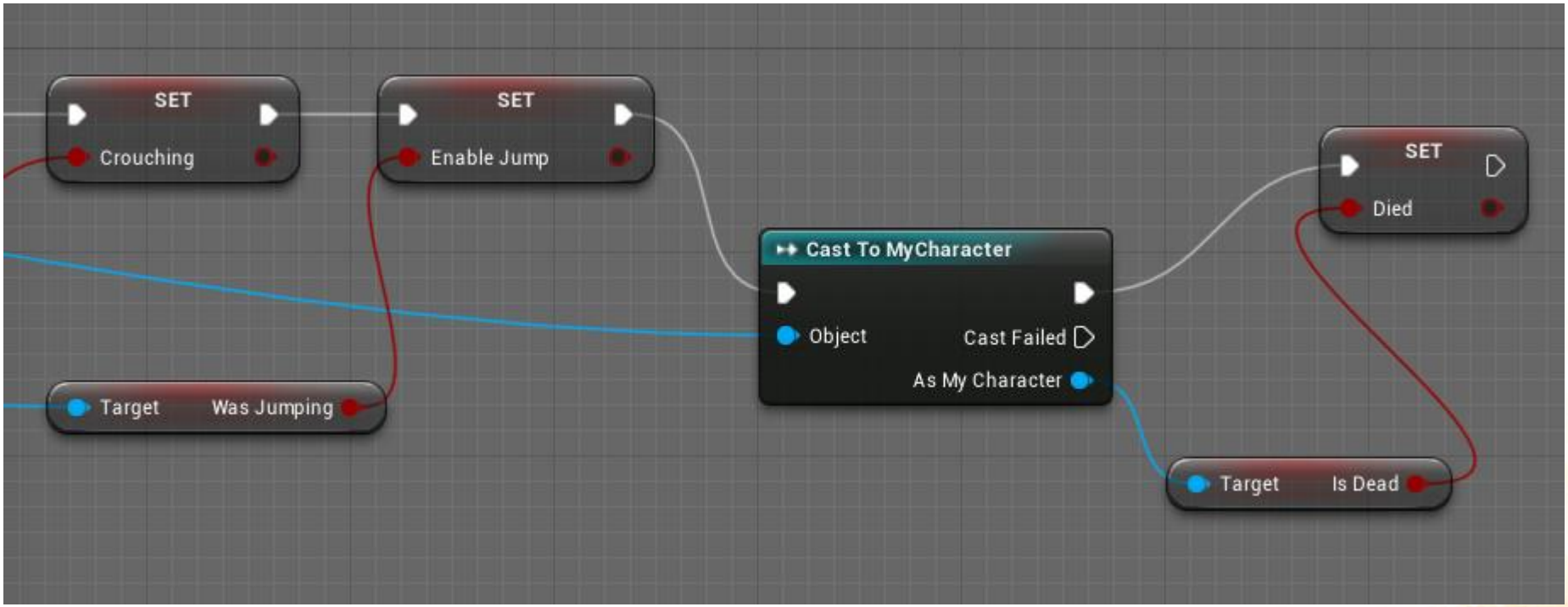
Multiplayer Game

- Implement the death animation logic:



Multiplayer Game

- Implement the death animation logic:



Multiplayer Game

- Replicate the health over the network:

MyHealthComponent.h

```
protected:
  UPROPERTY(Replicated, EditDefaultsOnly, BlueprintReadOnly,
            Category = "Health")
  float Health;
```

MyHealthComponent.cpp

```
UMyHealthComponent::UMyHealthComponent(){
  ...
  SetIsReplicated(true);
}
void UMyHealthComponent::BeginPlay(){
  Super::BeginPlay();
  if (GetOwnerRole() == ROLE_Authority)  {
    AActor* myOwner = GetOwner();
    if (myOwner){
      myOwner->OnTakeAnyDamage.AddDynamic(this,
                                  &UMyHealthComponent::OnTakeDamage);
    }
  }
}
```

Multiplayer Game

- Replicate the health over the network:

MyHealthComponent.cpp

```
void UMyHealthComponent::GetLifetimeReplicatedProps(Tarray
                          <FLifetimeProperty>& OutLifetimeProps) const
{
  Super::GetLifetimeReplicatedProps(OutLifetimeProps);
  DOREPLIFETIME(UMyHealthComponent, Health);
}
```

Multiplayer Game

- Replicate the death animation over the network:

MyCharacter.h

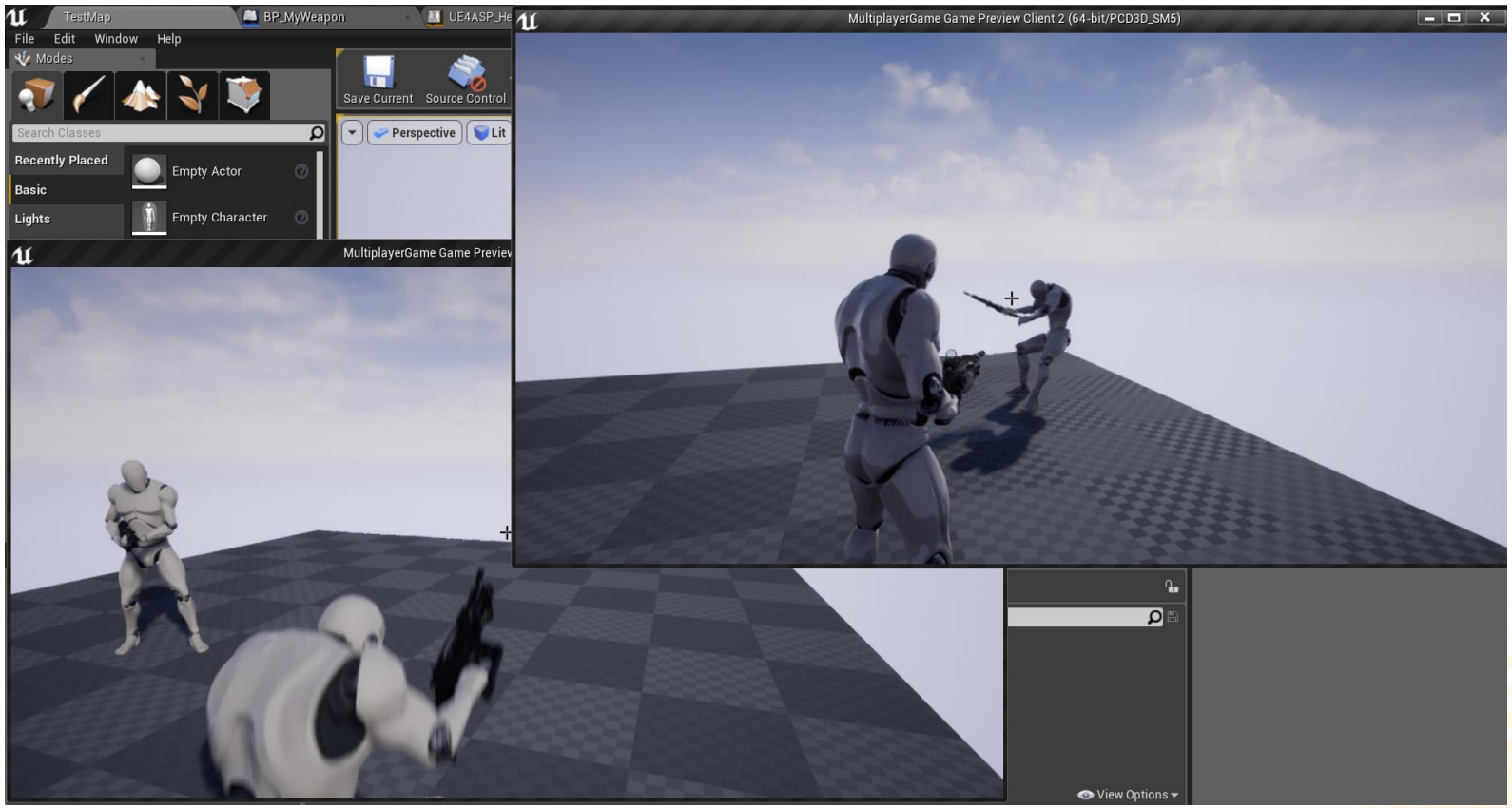
```
protected:
  UPROPERTY(Replicated, BlueprintReadOnly, Category = "Player")
  bool IsDead;
```

MyCharacter.cpp

```
void AMyCharacter::GetLifetimeReplicatedProps(TArray<FLifetimeProperty>&
                                                OutLifetimeProps) const {
  Super::GetLifetimeReplicatedProps(OutLifetimeProps);
  DOREPLIFETIME(AMyCharacter, CurrentWeapon);
  DOREPLIFETIME(AMyCharacter, IsDead);
}
```

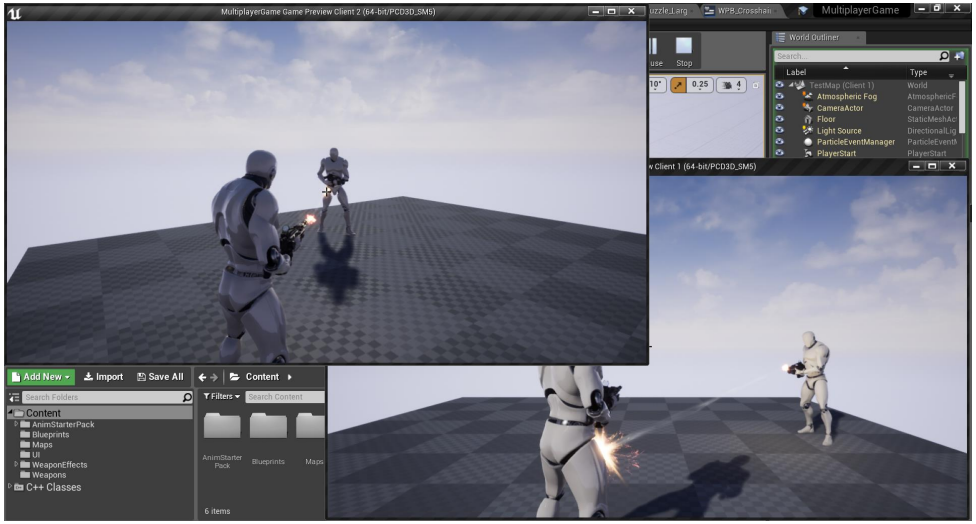
Multiplayer Game

- Test the results:



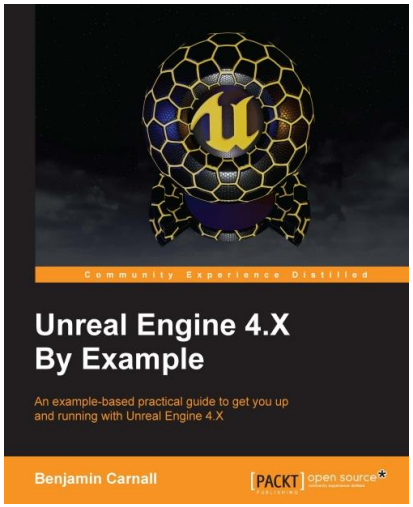
Exercise 1

1) Continue the development of the multiplayer game and add the following features:
   - Display the player health on the HUD;
   - When the player is killed, display a “game over” message;
   - When all other players are killed, display a “you win” message;



Further Reading

- Carnall, B. (2016). **Unreal Engine 4.X By Example**. Packt Publishing. ISBN: 978-1785885532.



- **Web Resources:**
  - Networking and Multiplayer in Unreal Engine - https://docs.unrealengine.com/en-us/Gameplay/Networking
  - Network Guide - https://wiki.unrealengine.com/index.php?title=4_13%2B_Network_Guide