Lab 9 - Extending AI Behaviours in a Multiplayer Environment

COMP280-Multiplayer Game Programming

Purpose: Extending AI Behaviours in a Multiplayer Environment

Contents

1. Introduction

Enhancing an enemy character's behavior in a multiplayer game has the following benefits:

- Makes the game more challenging and exciting for players
- It can also help create a more immersive experience, as enemies become smarter, faster, and stronger.
- New abilities or upgrading of the existing ones, can make the game stand out from other similar titles on the market.

By the end of this lab, we'll understand better:

- how to manage AI Actors in a multiplayer game.
- how to ensure effective communication within a networked environment.

In this lab we'll do the following:

- Making AI opponents more challenging
- Implementing an alert system
- Adding health to the AI
- Adding a weapon system to a character
- Creating AI variations

> From this lab and on, the labs are not individual but **Studio-based** (that is, it's enough for one member of the Studio to upload the lab deliverables). This way the studios can be focused on creating the final project. It is still suggested that you try to do the lab individually as well, but it is not mandatory; it's better for the studio to be engaged as a whole.

1.1. Technical requirements

- We are going to use **Unreal Engine 5.4**.
- You have to have finished the **Lab 8**.

2. Making AI opponents more challenging

So far, our undead lackey is equipped with a (more or less) keen sense of vision, allowing it to peer into the abyss of the dungeon, scouting for unsuspecting prey. However, even the sneakiest of thieves can unexpectedly bump into a hindrance while tip-toeing through the shadows. The cunning Lichlord knows this all too well and has bestowed upon his minions the added gift of acute hearing, so not even a pin drop goes unnoticed! In this section, we'll implement a **noise system** based on player character movement. The game logic we'll be adding is based on the following requirements:

- The thief character will make a **noise** when **sprinting**
- The **noise level** will be based on the character **statistics**
- The AI minions will **react** when they **hear** a noise

Let's add the capability/pitfall to **make noise** to our character.

2.1. Making some noise

To let our thief character make noise while it’s sprinting, we'll add a new component – a **pawn noise emitter**. This component will not spawn an actual sound or noise, but it will **emit a signal** that can be intercepted by the **pawn-sensing** component you have attached to the minion character. - Open the **US_Character.h** header file, and in the **private** section, add the following code:

```
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Stealth", meta = (AllowPrivateAccess = "true"))
TObjectPtr<UPawnNoiseEmitterComponent> NoiseEmitter;
```

- The component has been declared, let's **initialize** it. Open the **US_Character.cpp** file and add the necessary include declaration at the top of the file (below all others):

```
#include "Components/PawnNoiseEmitterComponent.h"
```

- Find the constructor, and just after the **FollowCamera** initialization, add these two lines:

```
NoiseEmitter = CreateDefaultSubobject<UPawnNoiseEmitterComponent>(TEXT("NoiseEmitter"));
NoiseEmitter->NoiseLifetime = 0.01f;
```

After the component creation, we just initialize its **lifetime** to a really low value (i.e., 0.01) – this value indicates the time that should pass before the new noise emission overwrites the previous one. As we use the **Tick()** event to **emit** the noise, and this event is executed every frame, we don't need a high value.

Look for the **Tick()** function, and just before its closing bracket, add the following code:

```
if (GetCharacterMovement()->MaxWalkSpeed == GetCharacterStats()->SprintSpeed)
{
  auto Noise = 1.f;
  if(GetCharacterStats() && GetCharacterStats()->StealthMultiplier)
  {
    Noise = Noise / GetCharacterStats()->StealthMultiplier;
  }
  NoiseEmitter->MakeNoise(this, Noise, GetActorLocation());
}
```

In the previous code, we verify whether the character is sprinting and proceed further only if the result is affirmative. We then compute the noise, based on a unity value divided by the **StealthMultiplier** character. As you will remember from Lab 6, **Replicating Properties Over the Network**, this value is declared inside the character statistics data table, and it grows as the character levels up. This means the higher the multiplier, the lower the noise made by the character. After the noise has been evaluated, it is emitted by the **NoiseEmitter** component by using the **MakeNoise()** method. Now that our character has picked up the skill of making noise while sprinting, it's time we equip our undead minions with some sharp-eared talents and set them to action!

2.2. Enabling the hearing sense

The minion character already has the ability to hear noise through the pawn-sensing component, but at the moment, this ability is not used. You'll need to open the **US_Minion.h** header file and add the following declaration to the **protected** section:

```
UFUNCTION()
void OnHearNoise(APawn* PawnInstigator, const FVector& Location, float Volume);
```

Note that his is a simple callback declaration that will be used to handle the hearing of any noise. Next, add the following method declaration to the **public** section:

```
UFUNCTION(BlueprintCallable, Category="Minion AI")
void GoToLocation(const FVector& Location);
```

This is a simple utility function that we will use to send the minion to the origin of the noise. Now, open the **US_Minion.cpp** file and look for the **PostInitializeComponents()** implementation. Just before the closing bracket, add the **delegate binding** for the hearing event:

```
GetPawnSense()->OnHearNoise.AddDynamic(this, &AUS_Minion::OnHearNoise);
```

Now, implement the **OnHearNoise()** function by adding the following code:

```
void AUS_Minion::OnHearNoise(APawn* PawnInstigator, const FVector& Location, float Volume)
{
  GEngine->AddOnScreenDebugMessage(-1, 5.f, FColor::Green, TEXT("Noise detected!"));
  GoToLocation(Location);
  UAIBlueprintHelperLibrary::SimpleMoveToLocation(GetController(), PatrolLocation);
}
```

Once a noise has been detected, we send the minion to the location where it was generated. As you can see, we don't check whether the noise instigator is our thief character – the Lichlord has commanded his minions to meticulously investigate any and all audible disturbances, leaving no corner unexplored! Finally, add the implementation for the **GoToLocation()** function:

```
void AUS_Minion::GoToLocation(const FVector& Location)
{
  PatrolLocation = Location;
  UAIBlueprintHelperLibrary::SimpleMoveToLocation(GetController(), PatrolLocation);
}
```

Here, we just set **PatrolLocation** and send the minion there (it's nothing fancy but extremely useful, as you will see later in the chapter). The minion is now ready, so compile your project and start some testing.

2.3. Testing the hearing sense

To test the brand-new hearing sense feature, start a game session and walk around the minions, paying attention to not enter their sight cone of vision. The minions won't notice the character unless it starts sprinting. At that moment, you should get a debug message, and the minion will start chasing the thief. In this section, we've added a new sense to the minion characters; this will make the game more tactical for the players – running around the dungeon like there's no tomorrow won't be an optionable solution! In the upcoming section, we'll implement a messaging system that enables minions to message it's "mates" upon discovering fresh prey.

3. Implementing an alert system

Let's work on a system that allows an AI character to alert its fellow minions once it detects a player character. At first glance, you might assume that the code logic to alert nearby AI opponents could be implemented directly inside the minion class – it's just a matter of sending them a message, isn't it? But there's more to it than meets the eye, dear reader. It seems the Lichlord has bigger aspirations for communication than you had anticipated. Fear not, for he has dictated that you make use of a Gameplay Framework class that has lurked unnoticed in the shadows until this moment – the Game Mode. As you will remember from Chapter 4, Setting Up Your First Multiplayer Environment, a Game Mode is a class that manages a game's rules and settings – this includes tasks such as communicating with AI Actors in the level. Alerting them of a new intruder in the dungeon is definitely a feature we want to have in this class.

3.1. Declaring the Game Mode functions

As usual, you will start by declaring the needed functions inside the class header – in this case, you will need just one, called AlertMinions(). Open the US_GameMode.h header file and declare it in the public section:

```
UFUNCTION(BlueprintCallable, Category = "Minions")
void AlertMinions(class AActor* AlertInstigator, const FVector& Location, float Radius);
```

Although this function may appear pretty simple, it will provide valuable information such as which minion has detected something, the position where to investigate, and the distance at which fellow minions should be alerted. Now, open the US_GameMode.cpp file and add the following include declarations at the very top of the code:

```
#include "US_Minion.h"
#include "Kismet/GameplayStatics.h"
```

As you already know, those declarations are needed to properly implement the code you will write in the class. Once you have added those lines, you can add the following method implementation:

```
void AUS_GameMode::AlertMinions(AActor* AlertInstigator, const FVector& Location, const float Radius)
{
  TArray<AActor*> Minions;
  UGameplayStatics::GetAllActorsOfClass(GetWorld(), AUS_Minion::StaticClass(), Minions);
  for (const auto Minion : Minions)
  {
    if(AlertInstigator == Minion) continue;
    if (const auto Distance = FVector::Distance(AlertInstigator ->GetActorLocation(), Minion-
>GetActorLocation()); Distance < Radius)
    {
      if (const auto MinionCharacter = Cast<AUS_Minion>(Minion))
      {
        MinionCharacter->GoToLocation(Location);
      }
    }
  }
}
```

The code looks for all the classes that extend AUS_Minion in the level through GetActorsOfClass() and stores them in an array. After that, it loops through this array, computing the distance between each minion and the alerting one. If the distance is within range (i.e., the Radius property), the AI will be commanded to go to that location and investigate through the GoToLocation() function. The alert behavior for the Game Mode has been implemented; this means it's now possible for minions to call for assistance whenever they detect an intruder.

3.2. Making the AI send alert messages

Sending messages from the AI character is a pretty straightforward task, as the Game Mode is reachable from any Actor in the game as long as it's on the server – as you may already know, this is an awesome feature provided by the Unreal Engine Gameplay Framework. So, let's open the US_Minion.h file and declare the alert radius for the soon-to-be-sent messages in the private section:

```
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category = "Minion AI", meta = (AllowPrivateAccess =
"true"))
float AlertRadius = 6000.0f;
```

Using a configurable radius range will come in handy to create different types of minions – do you want a super-alert, ear-piercing sentinel? Set it to a very high value! Or go for a slimy, self-serving AI that's just in it for the Lichlord's favors and undead promotions by setting it to zero – this way, none of the fellow minions will be alerted and the sentinel will (hopefully) be granted a pat on the head by its lord upon reaching the player character. The choice is yours! To implement the function, open the US_Minion.cpp file and add the following include at the very beginning of the file:

```
#include "US_GameMode.h"
Then, locate the Chase() method. Before its closing bracket, add this code:
if(const auto GameMode = Cast<AUS_GameMode>(GetWorld()->GetAuthGameMode()))
{
```

```
  GameMode->AlertMinions(this, Pawn->GetActorLocation(), AlertRadius);
}
```

As you can see, once the Game Mode has been retrieved, we just send the alert message with the opportune parameters. It's now time to compile the project and do some testing.

3.2.1 Testing the alert implementation

Start a new game session, and once the number of minions is good enough, let your character be detected by one of them. Once alerted, all nearby minions will begin investigating the area, posing a serious danger to the player as more and more AI characters spot them, leading to a potentially dangerous chain reaction. Figure 9.2 shows one such situation – the player has not been stealthy enough, and AI opponents have started detecting the character while alerting each other.

Figure 9.2 – The alert system in action In this section, you implemented a messaging system for your AI opponents and learned the power of having a centralized place to manage your gameplay logic. In the next section, you will use the Gameplay Framework damage system to let players defeat enemies. Did you really believe I'd allow the poor thief hero to rot in the Lichlord's grasp without any aid? Well, think again, my dear reader!

4. Adding health to the AI

In this part of the project, you will add a health system to the minion AI to make it possible to defeat it during gameplay. You will also add a spawn system so that when the opponent is defeated, the player will be rewarded with a well-deserved prize. To implement such features, we need to open the minion class and start doing some coding – open the US_Minion.h header, and in the private section, add these two declarations:

```
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category="Health", meta = (AllowPrivateAccess = "true"))
float Health = 5.f;
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category="Pickup", meta = (AllowPrivateAccess = "true"))
TSubclassOf<class AUS_BasePickup> SpawnedPickup;
```

The first one is used to keep track of the enemy health, while the second one will contain the class of the item pickup that will be spawned once the minion is defeated. Both of them can be modified in a child Blueprint Class (thanks to the EditDefaultsOnly property specifier), so you can build your own variations of the minion. Now, locate the protected section and add the damage handler declaration:

```
UFUNCTION()
void OnDamage(AActor* DamagedActor, float Damage, const UDamageType* DamageType, AController*
InstigatedBy, AActor* DamageCauser);
```

The header is now complete, so it's time to open the US_Minion.cpp file and implement the health system. As usual, start by adding the needed include declarations at the top of the file:

```
#include "US_BasePickup.h"
```

Next, declare the base pickup that will be spawned when the character is defeated; you'll use the pickup coin you created in Chapter 6, Replicating Properties Over the Network. Locate the constructor, and just before the closing bracket, add this code:

```
static ConstructorHelpers::FClassFinder<AUS_BasePickup>
SpawnedPickupAsset(TEXT("/Game/Blueprints/BP_GoldCoinPickup"));
if (SpawnedPickupAsset.Succeeded())
{
```

```
  SpawnedPickup = SpawnedPickupAsset.Class;
}
```

This code logic should be familiar, as we get a Blueprint asset from the project library and assign it the SpawnedPickup reference. Then, we need to implement the damage handler logic. Locate the PostInitializeComponents() method and add this line of code:

```
OnTakeAnyDamage.AddDynamic(this, &AUS_Minion::OnDamage);
```

Here, we just bind the OnDamage handler to the OnTakeAnyDamage delegate. As a last step, we need to implement the OnDamage() method, so add this code to your class:

```
void AUS_Minion::OnDamage(AActor* DamagedActor, float Damage, const UDamageType* DamageType,
AController* InstigatedBy, AActor* DamageCauser)
{
  Health -= Damage;
  if(Health > 0) return;
  if(SpawnedPickup)
  {
    GetWorld()->SpawnActor<AUS_BasePickup>(SpawnedPickup, GetActorLocation(), GetActorRotation());
  }
  Destroy();
}
```

What this function does is to subtract the Damage value from the Health property; if the minion reaches zero health, it will immediately spawn the prize (i.e., the pickup) and then it will destroy itself. In this section, you created a simple health system for the AI opponent, by adding the Health property and keeping track of its value as damage is taken – whenever the minion has been defeated, it will spawn a coin or a similar prize, ready to be picked up by the nearest (or swiftest) character! Unfortunately for your players, the hapless band of thieving heroes is, at the moment, ill equipped to dispatch the Lichlord's minions in the treacherous underground realm! Fear not, for we shall come to their aid by adding a splendid weapon inventory to their arsenal in the next section.

5. Adding a weapon system to the character

Your beloved character has been longing for a weapon system ever since you started implementing it. In this section, we shall finally grant its wishes and provide it the ability to wield (not-so) powerful tools of destruction. Let's make our character stronger and more formidable by arming it with an amazing weapon! Since our character hero is a sneaky thief who prefers to avoid direct combat with stronger and more heavily armored opponents, we will focus on a throwing dagger system. In order to avoid adding cluttered code in the US_Character class, you'll implement a brand-new component that will handle the weapon logic – this means that you'll work on the following features:

- A component that will be added to the character and handle the player input and dagger spawn logic
- A dagger weapon that will be thrown at runtime and cause damage to the enemy opponents

As a first step, we will create the weapon projectile that will be spawned by the character when attacking during gameplay.

5.1. Creating a dagger projectile

The first thing to do is create a projectile class that will serve as a throwable dagger. To do so, in the Unreal Editor, create a new C++ class that will extend Actor, and call it US_BaseWeaponProjectile.

Once it has been created, open the US_BaseWeaponProjectile.h file, and in the private section, add the following component declarations:

```
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category="Components", meta = (AllowPrivateAccess = "true"))
TObjectPtr<class USphereComponent> SphereCollision;
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category="Components", meta = (AllowPrivateAccess = "true"))
TObjectPtr<UStaticMeshComponent> Mesh;
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Components", meta = (AllowPrivateAccess = "true"))
TObjectPtr<class UProjectileMovementComponent> ProjectileMovement;
```

As you can see, we will add a collision area to check hits during gameplay, a static mesh for the dagger model, and projectile logic to make the dagger move once it has been thrown. Remaining in the private section, add the Damage property with a base value of 1:

```
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category="Weapon", meta = (AllowPrivateAccess = "true"))
float Damage = 1.f;
```

Then, in the public section, add the usual getter methods for the components:

```
FORCEINLINE USphereComponent* GetSphereCollision() const { return SphereCollision; }
FORCEINLINE UStaticMeshComponent* GetMesh() const { return Mesh; }
FORCEINLINE UProjectileMovementComponent* GetProjectileMovement() const { return ProjectileMovement; }
Finally, we need to add a handler for when the weapon makes contact with its target. Add the following code to the protected section:
UFUNCTION()
void OnHit(UPrimitiveComponent* HitComponent, AActor* OtherActor, UPrimitiveComponent* OtherComp, FVector NormalImpulse, const FHitResult& Hit);
```

The header is ready, so we need to implement the logic – open the US_BaseWeaponProjectile.cpp file and add the necessary include declarations at its top:

```
#include "US_Character.h"
#include "US_CharacterStats.h"
#include "Components/SphereComponent.h"
#include "Engine/DamageEvents.h"
#include "GameFramework/ProjectileMovementComponent.h"
```

Then, locate the constructor and add the following code:

```
SphereCollision = CreateDefaultSubobject<USphereComponent>("Collision");
SphereCollision->SetGenerateOverlapEvents(true);
SphereCollision->SetSphereRadius(10.0f);
SphereCollision->BodyInstance.SetCollisionProfileName("BlockAll");
SphereCollision->OnComponentHit.AddDynamic(this, &AUS_BaseWeaponProjectile::OnHit);
RootComponent = SphereCollision;
Mesh = CreateDefaultSubobject<UStaticMeshComponent>("Mesh");
Mesh->SetupAttachment(RootComponent);
Mesh->SetCollisionEnabled(ECollisionEnabled::PhysicsOnly);
Mesh->SetRelativeLocation(FVector(-40.f, 0.f, 0.f));
Mesh->SetRelativeRotation(FRotator(-90.f, 0.f, 0.f));
static ConstructorHelpers::FObjectFinder<UStaticMesh>
StaticMesh(TEXT("/Game/KayKit/DungeonElements/dagger_common"));
if (StaticMesh.Succeeded())
{
  GetMesh()->SetStaticMesh(StaticMesh.Object);
}
ProjectileMovement = CreateDefaultSubobject<UProjectileMovement Component>("ProjectileMovement");
ProjectileMovement->UpdatedComponent = SphereCollision;
ProjectileMovement->ProjectileGravityScale = 0;
ProjectileMovement->InitialSpeed = 3000;
ProjectileMovement->MaxSpeed = 3000;
ProjectileMovement->bRotationFollowsVelocity = true;
```

```
ProjectileMovement->bShouldBounce = false;
bReplicates = true;
```

This code logic is lengthy but straightforward to comprehend – we just create and initialize the necessary components: SphereCollision has some basic values you should be familiar with:

- Mesh is set to a dagger model and rotated and positioned in order to align with the overall Actor
- ProjectileMovement has gravity disabled and a speed that will move the Actor fast and simulate a real dagger

One thing to mention is that we bind the OnHit() method to the OnComponentHit delegate through the AddDynamic helper macro. Also, note the final line of code that activates replication for the weapon – always keep in mind that Actors are not replicated by default! Now, add the OnHit() implementation:

```
void AUS_BaseWeaponProjectile::OnHit(UPrimitiveComponent* HitComponent, AActor* OtherActor,
UPrimitiveComponent* OtherComp, FVector NormalImpulse, const FHitResult& Hit)
{
  auto ComputedDamage = Damage;
  if (const auto Character = Cast<AUS_Character>(GetInstigator()))
  {
    ComputedDamage *= Character->GetCharacterStats()->DamageMultiplier;
  }
  if (OtherActor && OtherActor != this)
  {
    const FDamageEvent Event(UDamageType::StaticClass());
    OtherActor->TakeDamage(ComputedDamage, Event, GetInstigatorController(), this);
  }
  Destroy();
}
```

The code can be divided into three main parts:

- In the first part, we compute the damage, starting from the Damage base value. If the instigator (i.e., the character that spawned the projectile) is US_Character, we get its damage multiplier from the statistics and update the provoked damage. This means the higher the level of the character, the higher the damage.

> For a refresher on how character statistics are managed, please take a look back at Chapter 6, Replicating Properties Over the Network.

- The second part of the code verifies whether the launched projectile has hit an Actor. If it has, it will then inflict the corresponding amount of damage.
- The last and final part simply destroys the projectile – its mission is finished, and this means it should be removed from the game.

With the projectile all set up and ready to go, it's time to implement some spawn logic so that your thief hero can unleash the full power of this shiny new weapon.

5.2. Implementing the weapon component

Let's start by creating a class that will add new features to the character. As you may remember from Chapter 4, Setting Up Your First Multiplayer Environment, a component will let you implement reusable functionality and can be attached to any Actor or another component. In this case, we will implement a weapon system, with a Scene Component, which has the ability to be placed somewhere inside the parent Actor (i.e., the component has a Transform property) – this will allow you to position the component somewhere inside the character and act as a spawn point for the thrown projectiles. Let's start by creating the class. To do so, create a new class that extends from a Scene component and call

it US_WeaponProjectileComponent. Once the creation process has finished, open US_WeaponProjectileComponent.h, and in the private section, add the following declarations:

```
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category="Projectile", meta = (AllowPrivateAccess = "true"))
TSubclassOf<class AUS_BaseWeaponProjectile> ProjectileClass;
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category="Input", meta=(AllowPrivateAccess = "true"))
class UInputMappingContext* WeaponMappingContext;
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category="Input", meta=(AllowPrivateAccess = "true"))
class UInputAction* ThrowAction;
```

As you can see, we declare the projectile class (i.e., the projectile we previously created, or a subclass of it). Then, we declare the necessary elements that will let us take advantage of the enhanced input system. As we don't want to add dependencies to the main character, we will use a different mapping context from the one used in Chapter 5, Managing Actors in a Multiplayer Environment – this will let us implement a flexible combat system and add as many features as we want, without adding clutter to the main character class. Imagine the thrill of watching your sneaky thief hero slipping through the shadows, quietly backstabbing the most despised minions of the dreaded Lichlord! The possibilities for mayhem and mischief will be endless! Okay, let's stop dreaming and get back to coding. In the public section, add a setter for the projectile class that will allow you to change the spawned dagger projectile:

```
UFUNCTION(BlueprintCallable, Category = "Projectile")
void SetProjectileClass(TSubclassOf<class AUS_BaseWeaponProjectile> NewProjectileClass);
```

This function has nothing to do with throwing things around, but it will be most useful if you plan to add a weapon pickup to your game, in order to improve your character's fighting skills. Lastly, in the protected section, declare the Throw() action and its corresponding server call:

```
void Throw();
UFUNCTION(Server, Reliable)
void Throw_Server();
```

This code will let us spawn the thrown daggers during gameplay from the server – always remember that the server should be in command when generating replicated Actors. Now that the header file is finished, open the US_WeaponProjectileComponent.cpp file to start implementing its features. As usual, locate the top of the file, and add the include declarations for the classes we will use:

```
#include "EnhancedInputComponent.h"
#include "EnhancedInputSubsystems.h"
#include "US_BaseWeaponProjectile.h"
#include "US_Character.h"
```

Then, in the constructor, add this single line of code:

```
ProjectileClass = AUS_BaseWeaponProjectile::StaticClass();
```

Here, we just declare the base projectile that will be spawned when the throw action is triggered; you will obviously be able to change it in the derived Blueprint Classes if you need a different weapon. Now, locate the BeginPlay() method, and just after the Super::BeginPlay() declaration, add this code:

```
const ACharacter* Character = Cast<ACharacter>(GetOwner());
if(!Character) return;
if (const APlayerController* PlayerController = Cast<APlayerController>(Character->GetController()))
{
  if (UEnhancedInputLocalPlayerSubsystem* Subsystem =
ULocalPlayer::GetSubsystem<UEnhancedInputLocalPlayerSubsystem>(PlayerController->GetLocalPlayer()))
  {
    Subsystem->AddMappingContext(WeaponMappingContext, 1);
  }
  if (UEnhancedInputComponent* EnhancedInputComponent = Cast<UEnhancedInputComponent>(PlayerController->InputComponent))
  {
    EnhancedInputComponent->BindAction(ThrowAction, ETriggerEvent::Triggered, this,
```

```
&UUS_WeaponProjectileComponent::Throw);
  }
}
```

In the previous code, we check that the component owner is our US_Character class, in order to get its controller and initialize the mapping context and its actions. Note that this initialization is done inside the BeginPlay() function, which means that these steps will be done just once – that is, when the game is started – to be sure that there is an Actor owner and a corresponding controller. Now, implement the throw logic by adding the following method implementations:

```
void UUS_WeaponProjectileComponent::Throw()
{
  Throw_Server();
}
void UUS_WeaponProjectileComponent:: Throw_Server_Implementation()
{
  if (ProjectileClass)
  {
    const auto Character = Cast<AUS_Character>(GetOwner());
    const auto ProjectileSpawnLocation = GetComponentLocation();
    const auto ProjectileSpawnRotation = GetComponentRotation();
    auto ProjectileSpawnParams = FActorSpawnParameters();
    ProjectileSpawnParams.Owner = GetOwner();
    ProjectileSpawnParams.Instigator = Character;
    GetWorld()->SpawnActor<AUS_BaseWeaponProjectile>(ProjectileClass, ProjectileSpawnLocation,
ProjectileSpawnRotation, ProjectileSpawnParams);
  }
}
```

As you can see, the Throw() method simply calls the server-side implementation that will spawn the projectile from the component location. You are already familiar with the spawn action (do you remember the minion spawner?), but there is an important thing to notice this time – we use the FActorSpawnParameters structure to set the owner of the projectile and, most importantly, the instigator (i.e., the Actor that spawned the object). This property is used by the projectile to retrieve the character statistics and handle the damage multiplier, code logic we implemented in the previous section. Lastly, add the setter method that will let you change the weapon spawned by the character:

```
void UUS_WeaponProjectileComponent::SetProjectileClass(TSubclassOf<AUS_BaseWeaponProjectile>
NewProjectileClass)
{
  ProjectileClass = NewProjectileClass;
}
```

The component has now been properly set – you just need to attach an instance of it to the thief character to make it fully operational.

5.3. Attaching the WeaponProjectile component to the character

Now that you have created a weapon component, it's time to add it to the character. Open the US_Character.h header file, and in the private section, add the component declaration:

```
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Weapon", meta = (AllowPrivateAccess = "true"))
TObjectPtr<class UUS_WeaponProjectileComponent> Weapon;
```

And, as usual, add the corresponding getter utility method to the public section:

```
FORCEINLINE UUS_WeaponProjectileComponent* GetWeapon() const { return Weapon; }
```

Then, open the US_Character.cpp source file and include the component class declaration at the top of the file:

```
#include "US_WeaponProjectileComponent.h"
```

Now, locate the constructor, and just after the noise emitter creation and initialization, add the following code:

```
Weapon = CreateDefaultSubobject<UUS_WeaponProjectileComponent>(TEXT("Weapon"));
Weapon->SetupAttachment(RootComponent);
Weapon->SetRelativeLocation(FVector(120.f, 70.f, 0.f));
```

As you can see, after we create the component, we attach it to the root component of the character and position it at a relative location, set to (120, 70, 0). If you want your character to be left-handed, you can just use a negative value for the X coordinate (i.e., −120.f). As hard as it may be to believe, the code to attach the weapon component to the character is complete; the code logic is already handled in the component itself, so you can sit back, relax, and let everything fall into place like a well-oiled machine! You can now switch back to the Unreal Editor and compile your project – once finished, you can open the BP_Character Blueprint, and you will see that the character is now equipped with a WeaponProjectile component, with Projectile Class set to a default value, as depicted in Figure 9.3:

Figure 9.3 – The WeaponProjectile component attached to the character Actor With the WeaponProjectile component attached to the character, the last thing to do is to create a mapping context for the player input and an input action for the throw logic.

5.4. Adding an input system for the weapon

In the final part of this section, you will define the mapping context and the input action for the throw interaction. This is something you are already familiar with, as you previously created similar assets in Chapter 5, Managing Actors in a Multiplayer Environment. So, without further ado, let's open the Content Browser and navigate to the Content | Input folder. We will create the throw action asset in the following steps.

5.4.1 Setting up the input mapping context for the throw interaction

To create the action asset for the throw interaction, follow these steps:

1. Right-click in the Content Browser and select Input | Input Action, naming the newly created asset IA_Throw.
2. Double-click on the asset to open it, and from the Value Type dropdown, select Digital (bool).
3. Double-check that the Consume Input checkbox is ticked.

Now that the action has been set, let's create a mapping context for the weapon interactions.

5.4.2 Setting up the input mapping context for the weapon interactions

To create the weapon context mapping, follow these steps:

1. Right-click in the Content Browser and select Input | Input Mapping Context, naming the asset IMC_Weapons. Double-click on the asset to open it.
2. Add a new mapping context by clicking on the + icon next to the Mappings field.

3. From the drop-down menu that will be added, select IA_Throw to add this action to the mapping context.
4. Click twice on the + icon next to the drop-down menu to add two other control bindings for this action (one is set by default). In the drop-down menu next to each new field, use the following properties:
5. The first binding should be set to Left Ctrl from the Keyboard category
6. The second binding should be set to Gamepad Face Button Right from the Gamepad category
7. The third binding should be set to Left Mouse Button from the Mouse category

Now that the assets are ready, it's time to add them to the character:

1. Open the BP_Character Blueprint, select the Weapon component, and in the Details panel, locate the Input category.
2. In the Weapon Mapping Context field, assign the IMC_Weapons asset.
3. In the Throw Action field, assign the IA_Throw asset.

Now that the input settings have been properly updated, it's time to do some testing to check that everything works properly.

5.5. Testing the weapon system

It's time to show the Lichlord's minions who the boss is and wreak some havoc in their underground lairs! Let's give them a taste of our hero's targeting skills by starting a game session. During gameplay, your character should be able to spawn a dagger whenever using the throw action – for example, by clicking the left mouse button. The dagger should destroy itself whenever it hits something and provoke damage to the AI minions. Whenever a minion reaches zero health, it should be removed from the game, and a coin should spawn in the level. Collecting enough coins will make your character level up, and consequently, the character itself will provoke additional damage when hitting any enemy. Figure 9.7 shows the character throwing some daggers during gameplay:

Figure 9.7 – The dagger attack in action In this section, you implemented a weapon system through a new component that is attached to your character and a projectile Actor that can be spawned in the game and properly replicated through the network. In the upcoming section, you will introduce some diverse variations of the AI opponents, with the aim of enhancing the game's variety and overall enjoyability.

6. Creating AI variations

Now that we've got the AI opponents all set up and ready to go for some epic battles, let's add some more variations to AI minions and make things more interesting and engaging. If a project has been well planned, changing the behavior of an AI – even a basic one such as the one we created in this chapter – is usually just a matter of adjusting some settings! In this section, you'll create a new AI opponent, starting from the basic US_Minion class, and you will tweak its property in order to give it different behavior.

6.1. Creating an AI sentinel

Let's craft some undead sentinels (guards) that will have keen senses and be more territorial. We'll start by creating a Blueprint Class, inheriting from the basic minion. Open the Content Browser and complete the following steps:

1. In the Blueprints folder, right-click and select **Blueprint** Class.
2. From the window that pops up, select **US_Minion** from the **All Classes** section (base class).
3. Name the newly created Blueprint **BP_MinionSentinel**, and then double-click on it to open it.
4. In the Details panel, locate the **Minion AI** category and apply the following settings:

Setting	Value
Alert Radius	6000.0
Patrol Speed	60.0
Chase Speed	20.0
Patrol Radius	1000.0

5. $Take\ Snapshot$
6. Then, in the AI section, apply the following settings:

Setting	Value
Hearing Threshold	600.0
LOSHearing Threshold	1000.0
Sight Radius	2500.0
Peripheral Vision Angle	60.0

7. $Take\ Snapshot$

With these settings, we'll create a minion that will patrol an approximately small area, changing direction frequently and moving very slowly. Its senses will be keen, and its alert radius will be larger than a regular minion. When an intruder has been spotted, the sentinel will slow down, calling for help, letting its more aggressive counterparts handle the chase. It’s not one for combat, but it’s still on the lookout for any suspicious activity!

8. Add a couple of glowing eyes for this darkness-gazing undead character by changing **Element 5** in the mesh **Materials** list to the **M_Base_Emissive** material asset.

We've created a new AI with just a couple of tweaks to the **Details** panel. Let’s create another one, a more aggressive undead minion.

6.2. Creating an AI miniboss

Let's use the same process that worked previously to create a new AI that will deal with hero intruders in a totally different way. Open the Content Browser and complete the following steps:

1. In the **Blueprints** folder, right-click and select **Blueprint Class**.
2. From the window that pops up, select **US_Minion** from the **All Classes** section (base class).

3. Name the newly created Blueprint **BP_MinionMiniboss**, and then double-click on it to open it.
4. In the **Details** panel, locate the **Minion AI** category and apply the following settings:

Setting	Value
Alert Radius	100.0
Patrol Speed	100.0
Chase Speed	400.0
Patrol Radius	50000.0

5 $Take\ Snapshot$ 6. In the **AI** section, apply the following settings:

Setting	Value
Hearing Threshold	200.0
LOSHearing Threshold	400.0
Sight Radius	200.0
Set Peripheral Vision Angle	20.0

7. $Take\ Snapshot$

This AI opponent has a dull behavior while patrolling (low perception, slower movement, etc.), but it will become dangerously fast once alerted. Let's give the minibos an armor makeover. To do this, follow these steps:

1. Select the **Mesh** component and change the **Skeletal Mesh Asset** property to the **skeleton_warrior** asset.
2. Change the **Mesh** scale to a value of **1.2** to make it bigger.
3. Set the **Health** property to a value of **20** to make it more damage-resistant.
4. Note that this foe is about to go from “meh” to menacing!
5. $Take\ Snapshot$

One can get really creative with the enemy opponents and test out all sorts of different behaviors and tactics. And if something just isn’t working, no worries! Just delete it and start afresh with something new in just a few short minutes.

Let's use the pickup spawn system we added earlier in this chapter to spice up the game? Depending on how rare or dangerous the defeated minion is, you could have it spawn different types of coins! Once you have your undead army ready to go, you can get back to your enemy spawner and add the Blueprints to the system – something we will do in the next subsection.

6.3. Updating the minion spawner

As you may have guessed, adding your brand-new minion varieties is just a matter of putting their Blueprints inside a spawner. To do so:

- select the **spawner** you previously added to the level,
- in the **Details** panel, locate the **Spawnable Minions** array property in the **Spawn System** category – there should already be one item in the list, **US_Minion**.
- Add as many items as you wish, selecting the spawnable minions you need for that particular spawn area.
- *Take Snapshot*

> If you chose to work with five elements, allocate f.e. a 20% chance for each to be added to the level every time the **Spawn()** method is called. Since the basic minion utilizes three of these elements, there is a 60% chance that it will appear as an opponent, compared to the sentinel and miniboss, which only have a 20% probability of spawning.

- Once you are happy with your setup, you can test the game.
- *Take Snapshot*

In this final section, you created some variations for the base minions; by changing some of their base properties, you changed the way they behave during gameplay, making your level more engaging and diverse.

7. Summary

In this lab we'll did the following:

- Made AI opponents more challenging
- Implemented an alert system
- Added health to the AI
- Added a weapon system to a character
- Created AI variations

8. Zip Project and Submit

- Click **File** → **Zip Project**
- Keep the default folder, select the name **US_LTOL_{YourInitials}_Lab9.zip**
- When the zipping finishes click on the provided link **Show in Explorer**
- Upload the zip in the corresponding folder (Lab9).
- Upload also the Snapshots document (if any) and the video wolkthrough (if any).

You can zip all the above in one file (project zip + shapshots document + video), or you can submit them separately.