Lab 10 - Enhancing the Player Experience

COMP280-Multiplayer Game Programming

Purpose: Enhancing the Player Experience in a Multiplayer Environment

Note

From lab 9 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.

Contents

1. Enhancing the Player Experience

To improve the game we need to add a good look and feel (polish) to it. A great-looking game will create an immersive experience that will engage players and make them want to keep coming back for more.

Here we will concentrate on improving certain aspects, such as:

- using different **animations** together for the player, and
  - **synchronizing** them over the network,
- adding non-player characters (NPCs) to interact with, and
  - using and synchronizing animations for NPCs.
- providing a **purpose** for your players to fight for (the rescue of some imprisoned comrades!
- Other ideas (noise, keys, cooldown, timer, data tables, ...)

By the end of this lab, we'll:

- have a nice **prototype** for our multiplayer game, and
- we'll be prepared to embark on the next phase — **optimizing** the game (time permitting).

In this lab, we'll deal with:

- **Animating** the character
- Adding **NPC** Actors
- Making further **improvements** to the game

1.1. Technical requirements

- We are going to use **Unreal Engine 5.4**.
- You have finished the **Lab 9**.

2. Animating the character

Let's add an **animation system** to the player(s). Here we'll create **simple animations** that will work on the networked environment of our project. This will involve:

- creating **Blueprints** designed for the animation system, and
- establishing their **connection** to your character class

We'll:

- create the needed animation **assets**, and
- add the needed **code** to make everything work properly.

2.1. Creating the animation assets

Animating characters in Unreal Engine involves creating **Animation Blueprints** that handle:

- the **motion**, and
- the **logic** of character *movement* and *actions*.

Here we don't deal very much with this topic, as it's not the main focus for game programming! That said, having some basic knowledge of how things work under the hood it's good to make your game development skills more “rounded”.

To **create** a simple but fully functional animation system for the player character, we will need three assets:

- An asset to **control** the **movement transitions** from **idle** to **walk** to **run**, and vice versa (**BlendSpace** - similar to **AnimationController** in Unity).
- An asset used to **play** the **throw animation** (similar to **AnimationClip** in Unity).
- A **Blueprint** to control the above assets

To do this, we need a **folder** to put all the assets in. So:

- Open the Unreal Editor
- In **Content Drawer**, create a new **folder** called **Animations**.

Now, we're ready to add the first asset.

2.1.1 Creating the movement Blend Space

ⓘ A **Blend Space** is a special asset that allows for the blending of animations based on the values of **two** inputs. It allows multiple animations to be **blended** by plotting them onto a **one-** or **two-dimensional graph**. Animators and game developers often use blend spaces to create **smooth** and **realistic** transitions between different animations for characters in games.

We will need to **blend** three animations – the **idle**, **walk**, and **sprint** ones – that will be managed depending on the character's **speed**.

To **create** this Blend Space, complete the following steps:

1. Inside the **Animations** folder, right-click and select **Animation → Blend Space**.
2. From the **Pick Skeleton** window that will pop up, select **rogue_Skeleton**.
3. Name the newly-created asset **BS_WalkRun**
4. Double-click it to open the **Blend Space Editor** window.
5. In Asset Details, locate the Axis Settings category. Then, in the Horizontal Axis section, do the following:
6. Set the Name value to Speed
7. Set the Maximum Axis value to 500
8. Leave the Vertical Axis section as it is (i.e., set to None), as we won't use it.

What we have done here is initialize the main setting values for the animation blend, exposing the Speed property that will be used by the controlling Blueprint we will be adding later. Now, you will be adding the animation assets that will be blended together. 5. Locate the rogue_Idle animation in the Asset Browser and drag it into the graph at the center of the editor. This will create a point in the coordinate system of the diagram. 6. Select the point and set its Speed property to 0 and its None property to 0. - $Take Snapshot$

Now, we will add two more assets to the graph – one for the walk animation and one for the run animation. 7. Drag the rogue_Walk asset into the graph and set its Speed property to 45 and its None property to 0. 8. Again, drag the rogue_Walk asset into the graph and set its Speed property to 100 and its None property to 0. 9. Drag the rogue_Run asset into the graph and set its Speed property to 500 and its None property to 0. - $Take Snapshot$

To test the animation blends on the character, you can simply press the Ctrl key and hover the mouse on the zone of the graph you want to check – you will see the character start a walk-and-move cycle, the animation assets blending seamlessly. The Blend Space is complete, so we can now start creating the asset that will handle the throw animation.

2.1.2 Creating the throw Animation Montage

An Animation Montage is a type of asset that enables the combination of multiple animations and their selective play from a Blueprint. Animation Montages are commonly used for creating complex animation sequences such as attack combos, cutscenes, and other interactive gameplay elements. In our project, we will use one to play the single-throw animation from the controlling Blueprint. To create the Animation Montage, complete the following steps:

1. Inside the Animations folder, right-click and select Animation | Animation Montage. Then, from the Pick Skeleton window that will pop up, select rogue_Skeleton.
2. Name the newly created asset AM_Throw and double-click it to open the Animation Montage Editor window.
3. From the Asset Browser, drag the rogue_Throw asset – in the DefaultGroup.DefaultSlot line – onto the timeline at the center of the editor.

4. $Take\ Snapshot$

This Montage and the previous Blend Space asset will be controlled by a dedicated Blueprint that we are going to add to the project in the next steps.

2.1.3 Creating the character Animation Blueprint

An Animation Blueprint is a specialized type of Blueprint that is used to create and control complex animation behaviors for Actors in the game. It defines how animations should be processed and blended together, as well as how animation inputs should be mapped. In our case, we need to control the Blend Space Speed parameter in order to let the character walk and run when needed, and start the throw Animation Montage when the character is attacking. To create the Animation Blueprint, complete the following steps:

1. Inside the Animations folder, right-click and select Animation | Animation Blueprint. Then, from the Create Animation Blueprint window that will pop up, select rogue_Skeleton.
2. $Take\ Snapshot$
3. Name the newly-created asset AB_Character and double-click it to open the editor window.

> If you are not already familiar with Animation Blueprints, you will notice some similarities to a regular Blueprint class, such as the My Blueprints and Event Graph tabs. If it is not already selected, open Event Graph to start some Visual Scripting code and then continue with the following steps.

3. Add an Event Blueprint Initialize Animation node.
4. Click and drag from the Return Value outgoing pin of Try Get Pawn Owner (which will already be present in the graph) and add a Cast To US_Character node.
5. Connect the event execution pin to the cast node incoming execution pin.
6. From the As US Character outgoing pin of the cast node, click and drag, selecting the Promote to Variable option – this will create a variable and add a corresponding Set node to the graph that will be automatically connected to the cast node. Name the variable Character.
7. From the outgoing pin of the Set Character node, click and drag to add a Character Movement getter node.
8. From this getter node outgoing pin, click and drag and select Promote to Variable. Name the variable Movement Component and connect the Set Movement Component node that will be automatically added to the graph to the execution pin of the Set Character node.
9. $Take\ Snapshot$

This visual script is executed when the Blueprint is initialized and basically sets the variables you will need later on, during gameplay. Now, locate the Event Blueprint Update animation node that should be already present in the graph. 9. From the Variables section, drag a getter node for the Character property. Right-click it and select the Convert to Validated Get option; this will change the node into an executable one that will check whether the Character variable is valid. 10. Connect the Event Blueprint Update Animation execution pin to the incoming execution pin of the Get Validated Character node. 11. In the Variables section, create a new variable of type float and call it CharacterSpeed. Drag a Set node for this variable into the graph. 12. From the Variables section, drag a Get node for the Movement Component variable. 13. Click and drag from the outgoing pin of the Movement Component node and create a Get Velocity property node. 14. Click and drag from the outgoing pin of the Get Velocity node and create a Vector Length XY node. 15. Connect the outgoing pin of the Vector Length XY node to the incoming pin of the Set Character Speed node. - $Take\ Snapshot$

This Visual Scripting code basically tracks the velocity magnitude of the character and stores it in the Character Speed variable, which will be used in the following steps to blend the movement animation. Next, select the AnimGraph tab of the editor, which will display a single Output Pose – this node represents the final animation pose of the character. We now need to tell the graph how to animate the character. 16. Drag the Character Speed property from the Variables section to create a getter node. 17. Click and drag from the Character Speed outgoing pin and create a Blendspace Player 'BS_WalkRun' node. 18. Click and drag the outgoing pin of the Blendspace Player 'BS_WalkRun' node and create a Slot 'Default Slot' node – we will use this node from the C++ code to execute the throw Animation Montage. 19. Connect the outgoing pin of Slot 'Default Slot' to the incoming pin of the Output Pose node. - $Take\ Snapshot$

With this final step, the Animation Blueprint is complete; now, you just need to connect it to the character Blueprint to make it work.

2.2. Adding the Animation System to the character

To add the animation system to the character, you just have to declare the **Animation** Blueprint inside the Blueprint class. To do so:

- Open the **BP_Character** Blueprint
- Select the **Mesh** property.
- In the **Details** panel, locate the **Anim Class** property.
- From the drop-down menu next to it, select **AB_Character**.
- *Take Snapshot*

If you test the game right now, you should see the character starting the animation loop and reacting to the player input when walking and running. However, the run animation will be weirdly jumpy and buggy – this is happening because these animations are not replicated and are just checking the character speed to update. From a technical point of view, the speed value (i.e., MaxWalkSpeed) is just stored in the server instance of the character, but the client will have its own MaxWalkSpeed value. While this may be acceptable if you are just moving an Actor around, as the server will be constantly updating the Actor position, animating a Skeletal Mesh component based on its speed is a totally different beast. In fact, the Animation System is using the local value (i.e., the client one) and the system will continuously conflict between server and client data, resulting in a broken animation. That's why we need to move the start-and-stop sprint logic we implemented in Chapter 7, Using Remote Procedure Calls (RPCs), from the server to the client and call the corresponding methods as multicast ones so that all the clients will be aware of this change. To do so, open the **US_Character.h** header file and add the following client declarations:

```
UFUNCTION(NetMulticast, Reliable)
void SprintStart_Client();
UFUNCTION(NetMulticast, Reliable)
void SprintEnd_Client();
```

As you can see, we used the NetMulticast specifier in order to let all the clients know that the character has started sprinting. Additionally, this call needs to be a Reliable one so you are guaranteed to send all the data to the recipients without any packet loss.

For a refresher on RPCs and the NetMulticast specifier, please refer to Lab 7, Using Remote Procedure Calls (RPCs).

- Open the **US_Character.cpp** file
- Locate **SprintStart_Server_Implementation()** and **SprintEnd_Server_Implementation()**. !!! Tip We are going to move all the content of both methods to the corresponding client-side calls. To do so:
- Remove all the content (i.e., the code in between the brackets)
- In **SprintStart_Server_Implementation()**, add the client-side call:

```
SprintStart_Client();
```

- For the **SprintEnd_Server_Implementation()** method, add the following:

```
SprintEnd_Client();
```

- Move the previously removed code to the client-side implementations:

```
void AUS_Character::SprintStart_Client_Implementation()
{
  if (GetCharacterStats())
  {
   GetCharacterMovement()->MaxWalkSpeed = GetCharacterStats()->SprintSpeed;
  }
}
void AUS_Character::SprintEnd_Client_Implementation()
{
  if (GetCharacterStats())
  {
   GetCharacterMovement()->MaxWalkSpeed = GetCharacterStats()->WalkSpeed;
  }
}
```

The overall behavior will then be as follows:

- The client controlled by the player receives the movement inputs and sends this data to the server
- The server handles this input and sends the update request to all clients

- All the clients update the MaxWalkSpeed value accordingly

Now:

- **compile** the project,
- **playtest** the game

> Note that our character can now:
>
> - move (walk)
> - sprint, and
> - is animated

2.2.1 Challenge

Try:

- working on the **minion** character, and
- implementing the same **animation logic**.

2.3. Adding the throw animation

What's missing at the moment is the throw animation and, in this case, network synchronization is something we really want – every connected player in the game will need to see the character animation whenever it is throwing the dagger in the dungeon, and this animation should be played at the same time for all clients. The first thing to do is to ensure that the WeaponProjectile component will be properly replicated. To do so:

- Open the **US_Character.cpp** file.
- In the **constructor**, locate the **Weapon** component initialization
- Add the following line of code:

```
Weapon->SetIsReplicated(true);
```

- Open **US_WeaponProjectileComponent.h**
- In the **private** section, add the following **Animation Montage** reference:

```
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly,Category="Projectile",meta=(AllowPrivateAccess="true"))
UAnimMontage* ThrowAnimation;
```

- Now, in the **protected** section, add the following declaration:

```
UFUNCTION(NetMulticast, Unreliable)
void Throw_Client();
```

> Note that this is the **throw** method that will be executed from the client side. Notice that we are **RPC multicasting** to all clients with the **Unreliable** property specifier – even though we want this animation synchronized over the network, it is just an **aesthetic** add-on, so we can afford to lose the data over the network. The other clients won't see the animation, but the dagger will be spawned anyway.

With the header declarations complete, do **implementation**:

- Open the **US_WeaponProjectileComponent.cpp** file
- Add the **client-side throw** method:

```
void UUS_WeaponProjectileComponent::Throw_Client_Implementation()
{
 const auto Character = Cast<AUS_Character>(GetOwner());
 if (ThrowAnimation != nullptr)
 {
  if (const auto AnimInstance = Character->GetMesh()->GetAnimInstance(); AnimInstance != nullptr)
  {
   AnimInstance->Montage_Play(ThrowAnimation, 1.f);
```

```
    }
  }
}
```

The code will get the owner of this component and, if it is of the **US_Character** type, will play the **Animation Montage**. This method will be called from its **server-side** counterpart, so:

- Locate the **Throw_Server_Implementation()** method.

We could just execute the method call, but we need to give a slight **delay** to the spawn logic because:

- the throw animation will take some **time** to complete, and
- spawning the dagger ahead of time would return ugly visual feedback to the player.

To do so:

- Remove all the content of the function.
- Replace it with the following code.

```
if (ProjectileClass)
{
  Throw_Client();
  FTimerHandle TimerHandle;
  GetWorld()->GetTimerManager().SetTimer(TimerHandle, [&]()
  {
    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);
  }, .4f, false);
}
```

We have moved the spawn logic inside a **timer handle** to delay the spawn process while calling the client-side throw logic, in order to start the animation immediately.

Unreal Engine provides more advanced methods for synchronizing animations beyond simply delaying method calls, such as **Animation Notifies**. However, for the purpose of this lab, the delay method is a quick and dirty solution that will suffice for our needs.

As the last step:

- Open the **BP_Character** Blueprint,
- Select the **Weapon** component and, in the **Details** panel,
- Look for the **Throw Animation** property
- Assign the **AM_Throw** montage we have already created.
- **Playtest** the game

The character should:

- **throw** the dagger, and
- **synchronize correctly with the throw animation.

- *Take Snapshot*

Here we:

- scratched the surface of animation with UE
- created a basic animation system that networked players will appreciate.

In the next section, we'll:

- animate AI characters
- add more fun to the game by having players rescue someone.

3. Adding NPC Actors

While taking a leisurely stroll in the underground and dodging or stabbing zombies can be amusing, let's not forget the big bucks the king's shelling out for us. We've got a rescue mission on our hands – liberate his knights from the Lichlord's dungeons before they're turned into undead abominations. Time to get down to business, my fearless developer! In this section, you'll create an Actor Blueprint that will serve as a prisoner your beloved thief needs to rescue (in order to get more experience points). To implement such a system, you will make good use of the Interactable interface you implemented in Chapter 7, Using Remote Procedure Calls (RPCs).

3.1. Creating the NPC character

The NPC you will be creating is a simple, replicated Actor that will cheer when the player has interacted with it and will grant some experience points. The first things we need are the Animation Montages that will play the idle and cheer animations. To do so, complete the following steps:

1. In the Animations folder, add a new Animation Montage asset based on the knight_Skeleton asset and call it AM_KnightIdle.
2. Add the knight_Idle animation to the DefaultGroup.DefaultSlot section of the montage.
3. Add another Animation Montage asset based on the knight_Skeleton asset and call it AM_KnightCheer.
4. Add the knight_cheer animation to the montage.

With these two animation assets ready, you can start creating the prisoner Blueprint. Open the Blueprints folder and complete the following steps. 5. Create a new Blueprint based on the Actor class and name it BP_KnightPrisoner. Double-click it to open it. 6. In the Components panel, add a Skeletal Mesh component. 7. In the Details panel, tick the Replicates property in order to replicate the Actor over the network. 8. Create an Integer variable and call it EarnedXp. Set its Default Value to 20. Tick the Instance Editable property to make the variable public. 9. Create an Anim Montage Object Reference variable and call it MontageIdle. Set its Default Value to AM_KnightIdle. Tick the Instance Editable property. 10. Create another Anim Montage Object Reference variable and call it MontageCheer. Set its default value to AM_KnightCheer. Tick the Instance Editable property. With these base settings available, you can start adding some Visual Scripting to the Event Graph in order to make the Actor fully functional. You will start from the Begin Play event to start the idle animation. To do so, complete the following steps:

1. Add an Event BeginPlay node to the graph.
2. From the Components panel, drag into the graph a reference of the Skeletal Mesh component.
3. From the Variables panel, drag a getter node for the MontageIdle variable.
4. From the Event BeginPlay execution pin, create a Play Animation node.
5. Connect the Skeletal Mesh pin to the Target pin of the Play Animation node.
6. Connect the Montage Idle pin to the New Anim to Play pin of the Play Animation node.
7. Tick the Looping property of the Play Animation node.
8. $Take\ Snapshot$

Then, create a custom event that will start the cheer animation when the prisoner is rescued by the player character. This event needs to be executed as a Multicast event in order to start the animation on all clients. To do so, complete the following steps:

1. Right-click on the graph and create a Custom Event node, naming it CharacterCheer. With the event selected, locate the Replicates property in the Details panel and, from its drop-down menu, select Multicast, leaving the Reliable checkmark unticked
   - $Take\ Snapshot$
2. From the Components panel, drag into the graph a reference to the Skeletal Mesh component.
3. From the Variables panel, drag a getter node for the MontageCheer variable.
4. From the Event BeginPlay execution pin, create a Play Animation node.
5. Connect the Skeletal Mesh pin to the Target pin of the Play Animation node.
6. Connect the Montage Idle pin to the New Anim to Play pin of the Play Animation node.
7. Tick the Looping property of the Play Animation node.
8. $Take\ Snapshot$

The last step needed to make the Actor work properly is to make it interactable with the player character by implementing the US_Interactable interface. To do so, complete the following steps:

1. Open the Class Settings panel and locate the Interfaces category.
2. In Implemented Interfaces, add the US_Interactable interface.
3. In the My Blueprint panel, locate the Interfaces category and right-click the Interact method, selecting Implement event. An Event Interact node will be added to the Event Graph.
4. Click and drag from the event Character Instigator pin to add a PlayerState node and connect its Target pin to the Character Instigator pin of the Event Interact node.
5. Click and drag from the outgoing pin of the PlayerState node and create a Cast To US_PlayerState node. Connect its incoming execution pin to the outgoing execution pin of the Event Interact node.
6. Click and drag from the As US PlayerState of the cast node and create an Add Xp node. Connect its incoming execution pin to the Success execution pin of the cast node.
7. From the Variables panel, drag a getter node for the EarnedXp variable. Connect its outgoing pin to the Value pin of the Add Xp node.
8. Click and drag from the outgoing pin of the Add Xp node and create a Character Cheer node to complete the graph
9. *Take Snapshot*

You may have noticed that we didn't use any authority checks in the previous graph; this is because we know that this event will only be called on the server. The Blueprint is now complete, so it's time to do some testing.

3.2. Testing the NPC Actor

To test the Blueprint, you can drag an instance of it into the level and start a gameplay session. The thief character should be able to reach the NPC and, if we use the interaction button, the animation should show them cheering. The hero who liberates the NPC characters will receive a well-deserved pool of experience points as a reward. Time to level up and become an even greater hero!

- *Take Snapshot*

Well, it seems we now have a new prisoner to play with! But why settle for one when we can have variations? Feel free to get creative and give our captive some fresh looks to keep things interesting. By creating child Blueprints and changing the Actor's Skeletal Mesh component and Animation Montages, you will be able to make good use of the barbarian and mage models available in the project. You may even create a rogue prisoner variation – who says we can't go off-script a little? The king may have paid us to rescue his knights and warriors, but hey, a skilled hero or two in the Thief Guild never hurt anyone!

Congratulations – you've completed this part of the adventure. Now it's time to let your imagination run wild and add your own game logic! In the next section, I won't be teaching you any new techniques, but I'll provide you with some fresh ideas to enhance the gameplay and make it more exciting.

4. Making further improvements to the game

Now that you have solid knowledge of how the Unreal multiplayer system works, it's time to unleash your creativity and bring your own ideas to life, making your game truly unique and personalized. In this section, I will give you some hints on how to spice up your project, but don't hesitate to add your own twist to make it uniquely yours.

4.1. Making noise!

Currently, the minions' hearing senses are only utilized to detect when a character is running. Why not tweak the system and let other elements in the game alert the Lichlord's minions? Unfortunately, PawnNoiseEmitterComponent can only be used on, well... pawns, so you cannot attach it to other Actors (it simply won't work); however, in Chapter 9, Extending AI Behaviors, you built a strong system to alert enemy minions that makes use of the Game Mode. As the Game Mode can be reached by any Actor in the level, you can exploit the AlertMinions() function and send messages that will call for help when activated. One of the best ways to use this method is through traps – whenever the player character steps into one such device, all the minions around will be alerted. Some examples of this kind of game feature include the following:

- Creaking doors: Whenever the character opens a door, it will make a creaking or squeaking sound that will alert the Lichlord's servants to the intruders.

- Traps: Some dungeon areas will be more protected than others – set some mechanical devices that will rally all nearby enemies. After all, this is just a matter of creating a collision area and calling a method in the Game Mode!
- Magical items: Create some magical artifact that can be interacted with by the player. The Lichlord is a sneaky one: he cast an alert spell, which condemned the hapless thief hero to their inevitable fate. Whenever the character tries to use that juicy item, an alarm will be sent to nearby minions, alerting them. Think of the possibilities! You can even use the floating book we created at the beginning of the project.

4.2. Finding and Using Keys!

Opening doors in a dungeon can be a fun game, but things can get even more interesting when you come across a locked door. Why not give it a try and see what other surprises lie in wait? In Chapter 7, Using Remote Procedure Calls (RPCs), you created the US_Interactable interface and made use of the Interact() method. However, the interface also exposes the CanInteract() method, which can be used to check whether the Actor can be interacted with. A door may implement a system that will only return true to the CanInteract() method if the player character has a key – this means creating a key pickup item and adding the US_Character system to track whether they have one or more keys to use. These locked doors can be used to keep the NPCs locked in some cellars and only able to be freed if the corresponding key is found somewhere in the level. Watch out for the Lichlord! His prisoners are double-locked up tighter than a merchant’s coin purse in the deepest and most heavily guarded cells of his dungeon!

4.3. Adding more weapons in the player's arsenal

While it’s nice to have a pointy dagger to throw at your hated opponents, it’s even nicer to have a magical one that will inflict more damage, or even defeat enemies with a single hit. You can implement a pickup Blueprint that will make good use of the SetProjectileClass() function you implemented in the US_WeaponProjectileComponent class. Upon picking up the item, the character will be granted a variant of the US_BaseWeaponProjectile class with augmented damage. You can even think about letting defeated enemies drop weapon pickups instead of coins! As an additional feature, you may even think to create throwing rocks that will send alert messages upon hitting the ground – just remember to enable gravity for the projectile. Having items that can be thrown and that will make noise to alert minions and direct them far away from the player characters will add some fresh gameplay logic that will improve the overall game experience. Get ready for a wickedly clever twist, and use your quick wit to dazzle the Lichlord’s mindless servants! With brains like yours, who needs brawn?

4.4. Adding Cooldown on Firing of Weapons

Currently, players have the ability to throw an unlimited number of daggers during gameplay. While this may be enjoyable initially, it will ultimately disrupt the balance of the game and result in a monotonous experience over time. To make things more interesting (and in favor of the Lichlord’s shadowy plans), limit the players to just one throwing dagger projectile at a time. Once the character has thrown the weapon, it won’t be able to throw it again until the dagger has been recovered.

4.4.1 Implementation Guide 1

– Once the character throws the projectile, set the **ProjectileClass** weapon component to a null value - Notice that the character won’t be able to spawn any more objects (fire more). - Upon hitting something:

- the thrown weapon will **spawn** a dagger pickup (as before)
- then will destroy itself.

- Notice that this will force the character to get to the dropped weapon and pick it up in order to attack again.

4.4.2 Implementation Guide 2

- Give the character a limited number of knives
- Use a **count** variable to check whether the character has any knives available every time the player tries to throw one.

4.5. Adding presure with a timer

At the moment, your characters can walk around calmly and take their time in rescuing the prisoners. Why not spice things up by adding a time counter? The Lichlord is hosting a grand celebration with the intention of transforming the king's knight into a loyal member of his undead army! Your hero must hurry up before it is too late!

4.5.1 Implementation Guide

- Use the **US_GameMode** class
- Create a **time manager** that:
    - will start as soon as the first player enters the dungeon
- – if the players can't free every captive from the dungeon, they loose!
    - Notice that it's all or nothing for this quest!

4.6. Tables, tables everywhere!

As your project progresses, it will become increasingly difficult to keep track of all the variations in enemies and weapons. To ease the pain, you can use the struct and data table system introduced in Chapter 6, Replicating Properties Over the Network, to create dedicated structures for the throwing weapons and AI opponents. Let your creative side run wild and come up with a ton of amazing Blueprint options based on the stats you like best – get ready for your hero adventurers to embark on thrilling journeys full of surprises in your game!

4.7. Need some help?

As you may have noticed from the previous subsections, once you become familiar with your game, the potential outcomes become limitless. You can add any new gameplay logic and test it until you are happy. On my end, I'll be working on creating exciting new features for the game and storing them in my own GitHub repository. Feel free to check in from time to time to see what wild and crazy ideas I've come up with! The link to the repository is https://github.com/marcosecchi/unrealshadows-ltol. And if you come up with a clever idea, feel free to contact me and tell me about it – if time permits, I will try to implement it and upload it to the repository in order to make this project grow! Summary Throughout this chapter, you fine-tuned the gameplay logic and added the finishing touches. You began by incorporating some nice animations for character movements and attacks, elevating the game's overall appeal. Additionally, you created someone for the players to rescue: a prisoner Actor that can be interacted with and that will grant the thief hero some well-deserved experience points. Last but not least, I shared a few fresh ideas to take your gameplay to the next level. By incorporating these ideas, you can make the game truly your own and one of a kind. So, get creative and have fun! Get ready for the next chapter, where you'll dive into debugging and testing a networked game. This will take your development skills to the next level, something that's necessary if you want to become a top-notch multiplayer programmer!

5. Summary

In this lab we did the following:

- used different **animations** together for the player, and
    - **synchronized** them over the network,
- added **non-player characters (NPCs)** to interact with, and
    - used and **synchronized** their animations.
- provided a **purpose** for our players to fight for (the rescue of some imprisoned comrades!)
- introduced some **other improvement ideas** (noise, keys, cooldown, timer, data tables, ...)

6. Zip Project and Submit

- Click **File** → **Zip Project**
- Keep the default folder, select the name **US_LTOL_{YourInitials}_Lab10.zip**
- When the zipping finishes click on the provided link **Show in Explorer**
- Upload the zip in the corresponding folder ($Lab10$).
- Please make a short walkthrough video ~1 min demonstrating the functionality and include it in the deliverables
- Upload also the Snapshots document (if any) and the video walkthrough (if any).

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