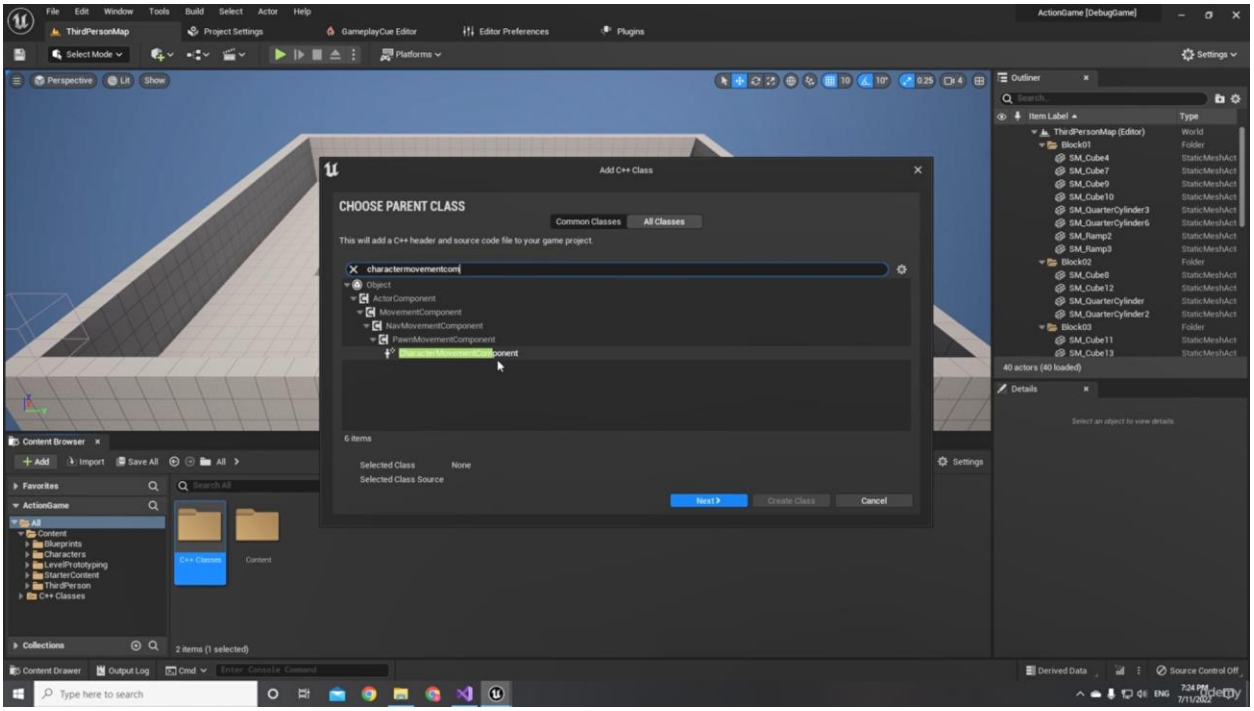
1. Movement Component, Movement Attributes

In this lecture we learn how to add our own CharacterMovementComponent and we set-up and connect speed attribute.

1. We start by making a sub-class of CharacterMovementComponent. (we put it in our ActorComponents folder)



To let our character use movementComponent we should use an ObjectInitializer **constructor** and we can grab that from our **Actor file.**

In our character.h:

Public:

AActionGameCharacter(const FObjectInitializer& ObjectInitializer);

We should call Super as usual and we do it this way.

In our character.cpp inside of the constructor before the {} we do this as the Initialization:

AActionGameCharacter::AActionGameCharacter(const FObjectInitializer& ObjectInitializer) :

Super(ObjectInitializer.SetDefaultSubobjectClass<UAG_CharacterMovementComponent>(ACharacter::CharacterMovementComponentName))

{ // old constructor data}

Then we move all the stuff we had in the previous constructor and delete it.

ACharacter::CharacterMovementComponentName is a predefined static member variable of the ACharacter class that **represents the name of the character's movement component.**

The purpose of using this constructor and specifying the custom subclass for the movement component is to **replace the default movement component provided by ACharacter with a custom implementation** (UAG_CharacterMovementComponent). This allows you to modify or extend the behavior of the character's movement, physics, and other related functionalities by implementing your own logic within UAG_CharacterMovementComponent.

By setting the default subobject class for the character's movement component in this constructor, all instances of AActionGameCharacter will use UAG_CharacterMovementComponent instead of the default UCharacterMovementComponent provided by ACharacter.

---

2. Then we add our other attributes:

```
    UPROPERTY(BlueprintReadOnly, Category = "Stamina", ReplicatedUsing = OnRep_Stamina)
    FGameplayAttributeData Stamina;
    ATTRIBUTE_ACCESSORS(UAG_AttributeSetBase, Stamina)

    UPROPERTY(BlueprintReadOnly, Category = "Stamina", ReplicatedUsing = OnRep_MaxStamina)
    FGameplayAttributeData MaxStamina;
    ATTRIBUTE_ACCESSORS(UAG_AttributeSetBase, MaxStamina)

    UPROPERTY(BlueprintReadOnly, Category = "MovementSpeed", ReplicatedUsing = OnRep_MaxMovementSpeed)
    FGameplayAttributeData MaxMovementSpeed;
    ATTRIBUTE_ACCESSORS(UAG_AttributeSetBase, MaxMovementSpeed)

protected:

    virtual void GetLifetimeReplicatedProps(TArray<class FLifetimeProperty>& OutLifetimeProps) const;

    virtual void PostGameplayEffectExecute(const struct FGameplayEffectModCallbackData& Data) override;

    UFUNCTION()
    virtual void OnRep_Health(const FGameplayAttributeData& OldHealth);

    UFUNCTION()
    virtual void OnRep_MaxHealth(const FGameplayAttributeData& OldMaxHealth);

    UFUNCTION()
    virtual void OnRep_Stamina(const FGameplayAttributeData& OldStamina);

    UFUNCTION()
    virtual void OnRep_MaxStamina(const FGameplayAttributeData& OldMaxStamina);

    UFUNCTION()
    virtual void OnRep_MaxMovementSpeed(const FGameplayAttributeData& OldMaxMovementSpeed);
};
```

```cpp
void UAG_AttributeSetBase::OnRep_Stamina(const FGameplayAttributeData& OldStamina)
{
    GAMEPLAYATTRIBUTE_REPNOTIFY(UAG_AttributeSetBase, Stamina, OldStamina);
}

void UAG_AttributeSetBase::OnRep_MaxStamina(const FGameplayAttributeData& OldMaxStamina)
{
    GAMEPLAYATTRIBUTE_REPNOTIFY(UAG_AttributeSetBase, MaxStamina, OldMaxStamina);
}

void UAG_AttributeSetBase::OnRep_MaxMovementSpeed(const FGameplayAttributeData& OldMaxMovementSpeed)
{
    GAMEPLAYATTRIBUTE_REPNOTIFY(UAG_AttributeSetBase, MaxMovementSpeed, OldMaxMovementSpeed);
}

void UAG_AttributeSetBase::GetLifetimeReplicatedProps(TArray<class FLifetimeProperty>& OutLifetimeProps) const
{
    Super::GetLifetimeReplicatedProps(OutLifetimeProps);

    DOREPLIFETIME_CONDITION_NOTIFY(UAG_AttributeSetBase, Health, COND_None, REPNOTIFY_Always);
    DOREPLIFETIME_CONDITION_NOTIFY(UAG_AttributeSetBase, MaxHealth, COND_None, REPNOTIFY_Always);
    DOREPLIFETIME_CONDITION_NOTIFY(UAG_AttributeSetBase, Stamina, COND_None, REPNOTIFY_Always);
    DOREPLIFETIME_CONDITION_NOTIFY(UAG_AttributeSetBase, MaxStamina, COND_None, REPNOTIFY_Always);
    DOREPLIFETIME_CONDITION_NOTIFY(UAG_AttributeSetBase, MaxMovementSpeed, COND_None, REPNOTIFY_Always);
}
```

3. Then we have to apply the movement speed:

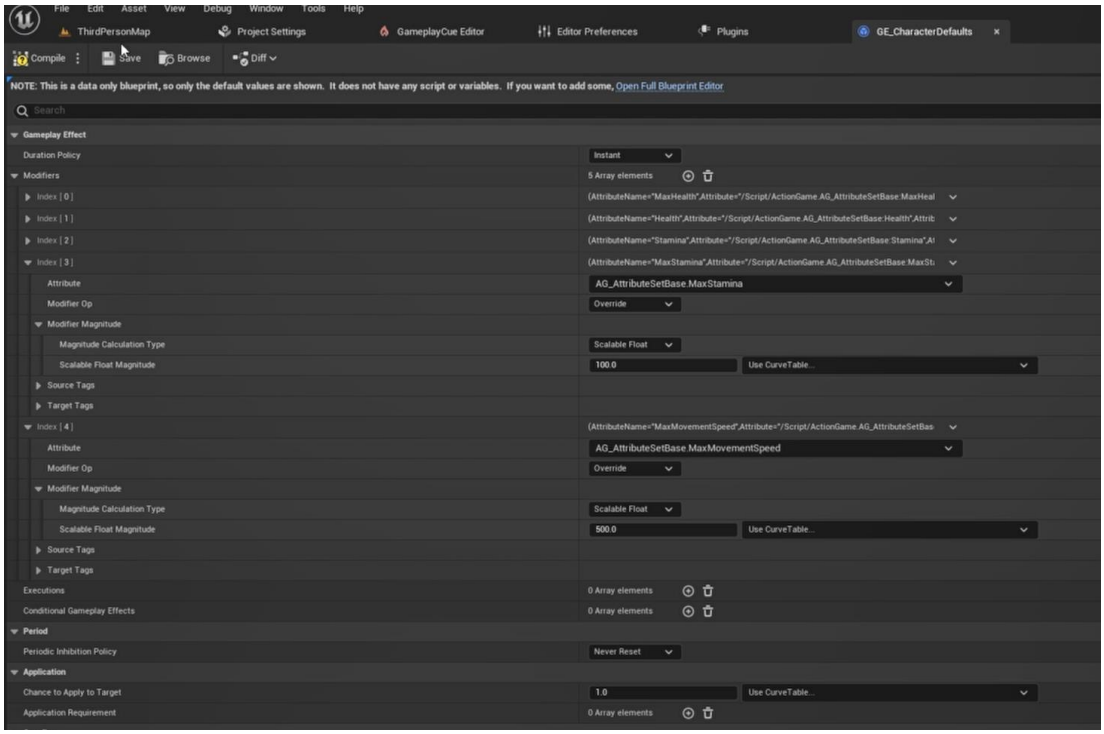
First we have to get our character then its movement component. (the "else if")

```cpp
void UAG_AttributeSetBase::PostGameplayEffectExecute(const struct FGameplayEffectModCallbackData& Data)
{
    Super::PostGameplayEffectExecute(Data);

    if (Data.EvaluatedData.Attribute == GetHealthAttribute())
    {
        SetHealth(FMath::Clamp(GetHealth(), 0.0f, GetMaxHealth()));
    }
    else if(Data.EvaluatedData.Attribute == GetMaxMovementSpeedAttribute())
    {
        ACharacter* OwningCharacter = Cast<ACharacter>(GetOwningActor());
        UCharacterMovementComponent* CharacterMovement = OwningCharacter ? OwningCharacter->GetCharacterMovement() : nullptr;

        if (CharacterMovement)
        {
            const float MaxSpeed = GetMaxMovementSpeed();

            CharacterMovement->MaxWalkSpeed = MaxSpeed;

        }
    }
}
```

The MaxWalkSpeed parameter, it is a speed cap for the walking-mod, so by changing it we will change the speed at which we should walk in the moment.

```cpp
#include "AbilitySystem/AttributeSets/AG_AttributeSetBase.h"
#include "GameplayEffectExtension.h"
#include "GameFramework/CharacterMovementComponent.h"
#include "GameFramework//Character.h"
#include "Net/UnrealNetwork.h"
```

Then we have to set-up the defaults effects:



Now we can manipulate character speed by one or more GameplayEffec's at once, for example some speed potions can increase speed like for 10% and some Debuffs decrease the speed by 20%.

---

2. Locomotion Blendspace, Movement Debug

The most common approach template for movement animations is to have a **Blendspace that is driven by a velocity**. It is network friendly and fail-safe.

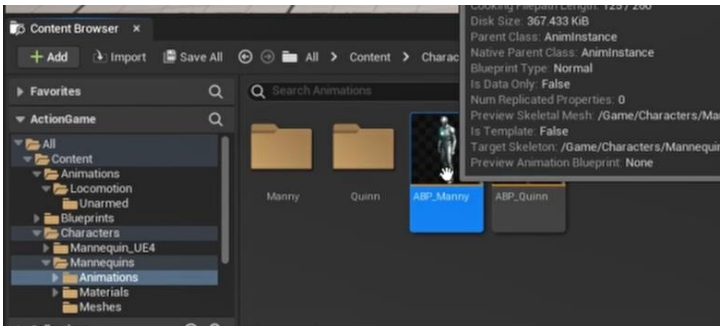
Strafe blend space is for when we want to have our character always looks forward.

This one I've done 3 times already and maybe 4 in future so you better not mess around ☺

1. Create a BlendSpace and choose your skeleton.
2. Horizontal Axis is named **Side** and its range is from -600 to 600 and its division is 10.
3. Vertical Axis is named **Forward** and its range is from -600 to 600 and its division is 10.

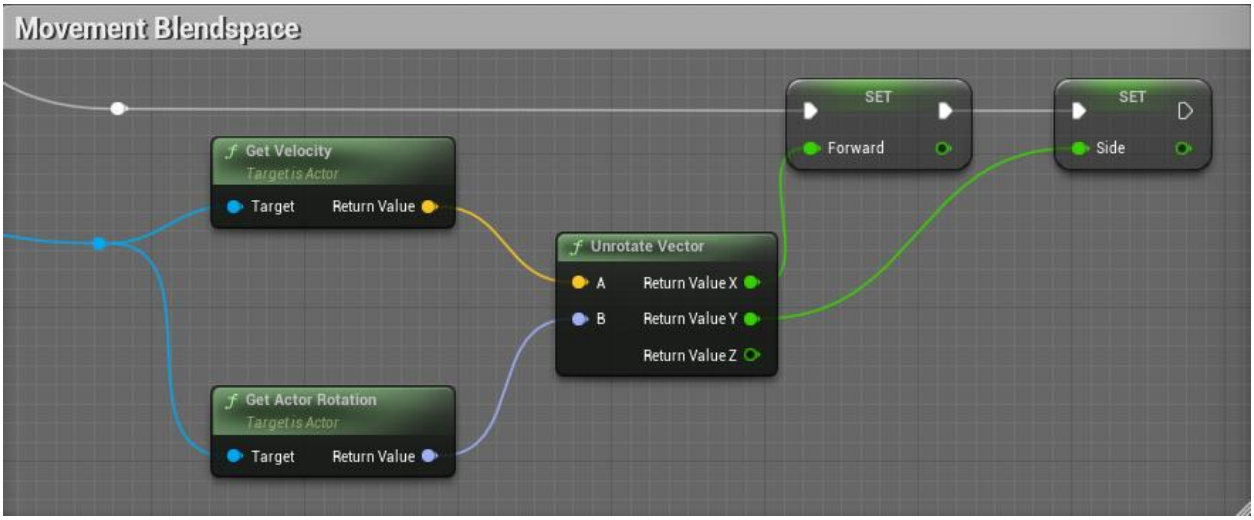
Side and Forward represents our speed in that direction and the more the speed is the more we blend to that animation. And we put the animations in their place.

We use the parent class ABP_Manny to modify the rest.



We add our BlendSpace to the locomotion in AnimGraph we also replace the idle with our solo idle animation2

Then we create 2 new variables for each axis and connect them. In the event graph we calculate them with Tick by adding one more row to it.



To see the actual results, we should change some settings in our character blueprint:

- Make Orient Rotation to Movement to false.
- Make Use Controller Desired Rotation to True.

---

3. Animation Instance, Animation Data, Animation Data Asset

Right now all of our animations are Hard coded in to the animation blueprint but in the real production we want to our animation to be data driven and that's what we will implement in this part.

1. We create our custom AnimInstance c++ class.
2. We use a similar approach as we did with the character and first create a specific Data class **(in ActionGameType.h).**

- First is the reference to the BlendSpace.
- Second, is the reference to the Idle Animation.

```
USTRUCT(BlueprintType)
struct FCharacterAnimationData
{
    GENERATED_USTRUCT_BODY();

    UPROPERTY(EditDefaultsOnly)
    class UBlendSpace* MovementBlendSpace = nullptr;

    UPROPERTY(EditDefaultsOnly)
    class UAnimSequenceBase* IdleAnimationAsset = nullptr;
};
```

3. We create our custom DataAsset c++ class.
4. As with the CharacterDataAsset we use our new structure as a field to set-up.

```
public:

    UPROPERTY(EditDefaultsOnly)
    FCharacterAnimationData CharacterAnimationData;
};
```

5. Then we create a Blueprint Asset from our AnimData Asset class and we'll set its references.
6. We'll have to some way connect this to our character and because every character should have some defaults animations, and we already did this for defaults Effects and Abilities, we add the reference to our Blueprint DataAsset to our CharacterData.

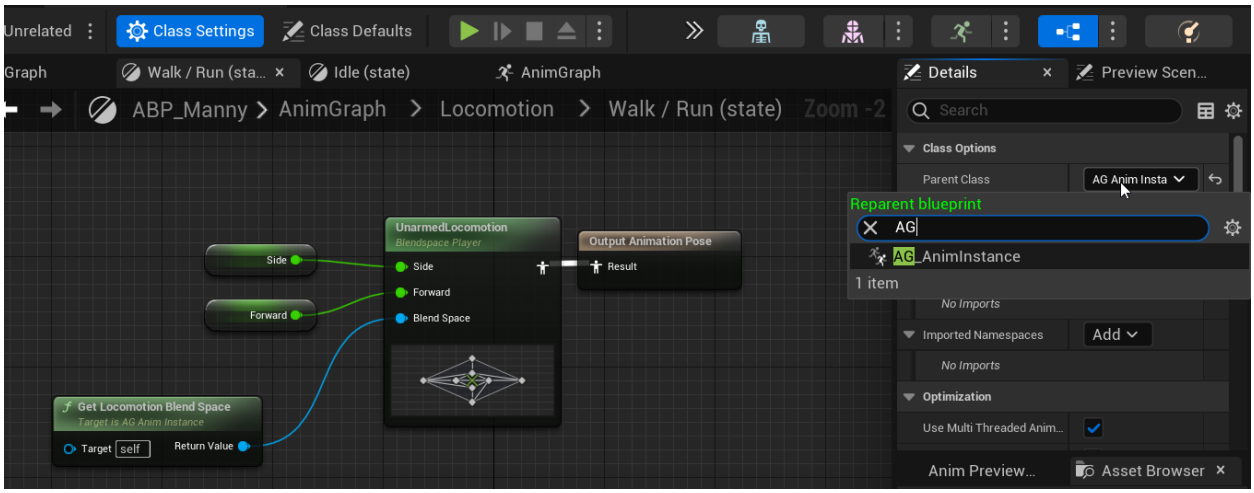
```
USTRUCT(BlueprintType)
struct FCharacterData
{
    GENERATED_USTRUCT_BODY();

    UPROPERTY(BlueprintReadWrite, EditAnywhere, Category = "GAS")
    TArray<TSubclassOf<class UGameplayEffect>> Effects;

    UPROPERTY(BlueprintReadWrite, EditAnywhere, Category = "GAS")
    TArray<TSubclassOf<class UGameplayAbility>> Abilities;

    UPROPERTY(BlueprintReadWrite, EditAnywhere, Category = "Animation")
    class UCharacterAnimDataAsset* CharacterAnimDataAsset;

};

USTRUCT(BlueprintType)
struct FCharacterAnimationData
{
    GENERATED_USTRUCT_BODY();

    UPROPERTY(EditDefaultsOnly)
    class UBlendSpace* MovementBlendSpace = nullptr;
```

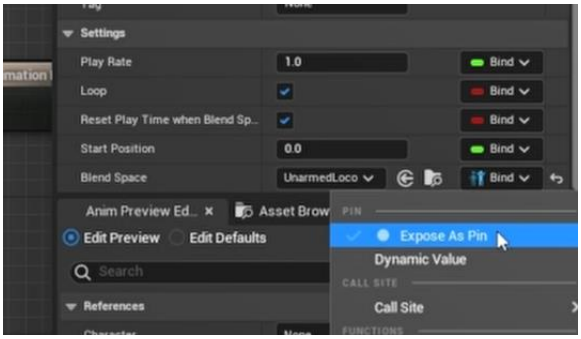
7. Then we set the reference in unreal.
8. We add some getters to our animation data fields.
9. Also we make a some defaults AnimAssets references right to the AnimInstance so we never return nullptr and it will be set at least.

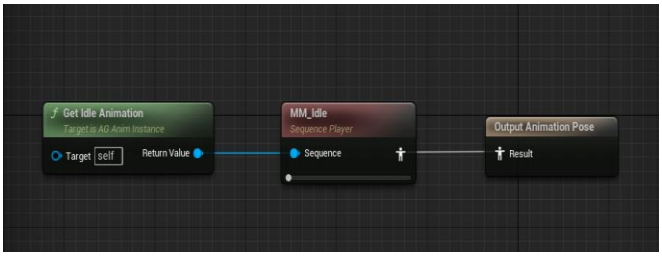
```
protected:

    UFUNCTION(BlueprintCallable, meta = (BlueprintThreadSafe))
    class UBlendSpace* GetLocomotionBlendSpace() const;

    UFUNCTION(BlueprintCallable, meta = (BlueprintThreadSafe))
    class UAnimSequenceBase* GetIdleAnimation() const;

    UPROPERTY(BlueprintReadWrite, EditAnywhere, Category = "Animation")
    class UCharacterAnimDataAsset* DefaultCharacterAnimDataAsset;
};
```

```cpp
#include "ActionGameCharacter.h"
#include "ActionGameTypes.h"
#include "Animation/AnimSequenceBase.h"
#include "Animation/BlendSpace.h"
#include "DataAssets/CharacterAnimDataAsset.h"
#include "DataAssets/CharacterDataAsset.h"

UBlendSpace* UAG_AnimInstance::GetLocomotionBlendSpace() const
{
    if (AActionGameCharacter* ActionGameCharacter = Cast<AActionGameCharacter>(GetOwningActor()))
    {
        FCharacterData Data = ActionGameCharacter->GetCharacaterData();

        if (Data.CharacterAnimDataAsset)
        {
            return Data.CharacterAnimDataAsset->CharacterAnimationData.MovementBlendSpace;
        }
    }

    return DefaultCharacterAnimDataAsset? DefaultCharacterAnimDataAsset->CharacterAnimationData.MovementBlendSpace : nullptr;
}

UAnimSequenceBase* UAG_AnimInstance::GetIdleAnimation() const
{
    if (AActionGameCharacter* ActionGameCharacter = Cast<AActionGameCharacter>(GetOwningActor()))
    {
        FCharacterData Data = ActionGameCharacter->GetCharacaterData();

        if (Data.CharacterAnimDataAsset)
        {
            return Data.CharacterAnimDataAsset->CharacterAnimationData.IdleAnimationAsset;
        }
    }

    return DefaultCharacterAnimDataAsset ? DefaultCharacterAnimDataAsset->CharacterAnimationData.IdleAnimationAsset : nullptr;
}
```

10. We have to change the parent of our existing animation blueprint in class setting, parent class. And then use the new created functions and set the default reference. Also we can expose blend space as a pin.







With this approach we can easily set-up character animation just by switching animation data Asset. Like you want different locomotion animation for a human or a troll.

---

4. Advanced physical materials

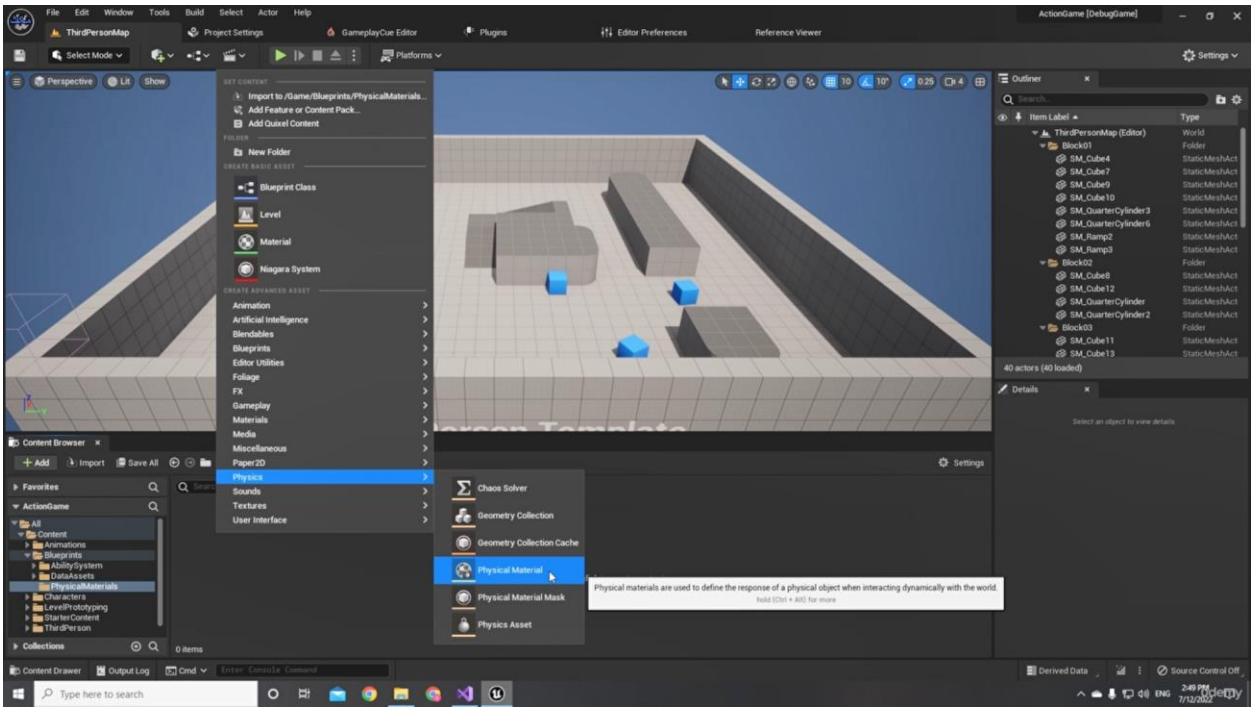
Create a PhysicalMaterial class that can tell us required information by itself. Our smart material can tell us what sound or vfx to play for footsteps.

We add the sound for now:

```
#include "CoreMinimal.h"
#include "PhysicalMaterials/PhysicalMaterial.h"
#include "AG_PhysicalMaterial.generated.h"

UCLASS()
class ACTIONGAME_API UAG_PhysicalMaterial : public UPhysicalMaterial
{
    GENERATED_BODY()

public:

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = PhysicalMaterial)
    class USoundBase* FootStepSound = nullptr;
};
```

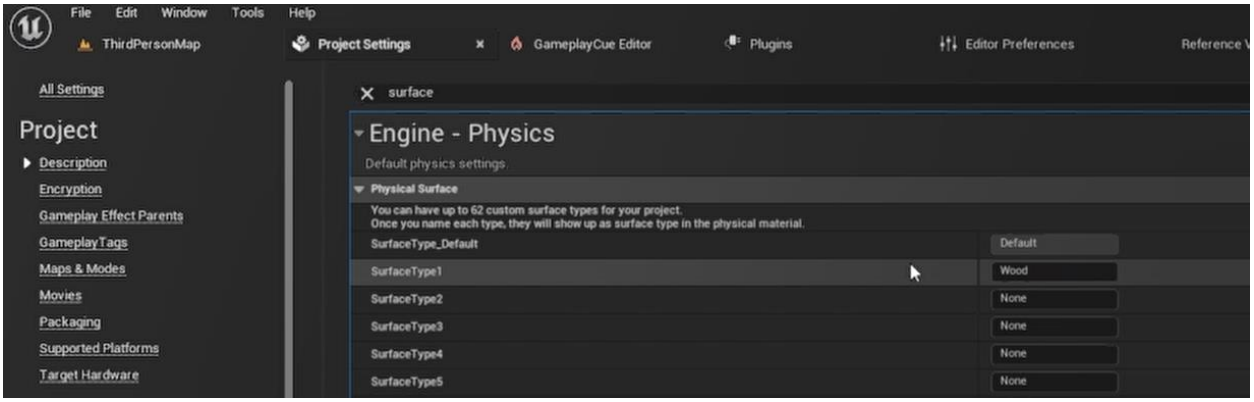
Then we create a Physical material of that class.

The surface type for this one we set as Default.

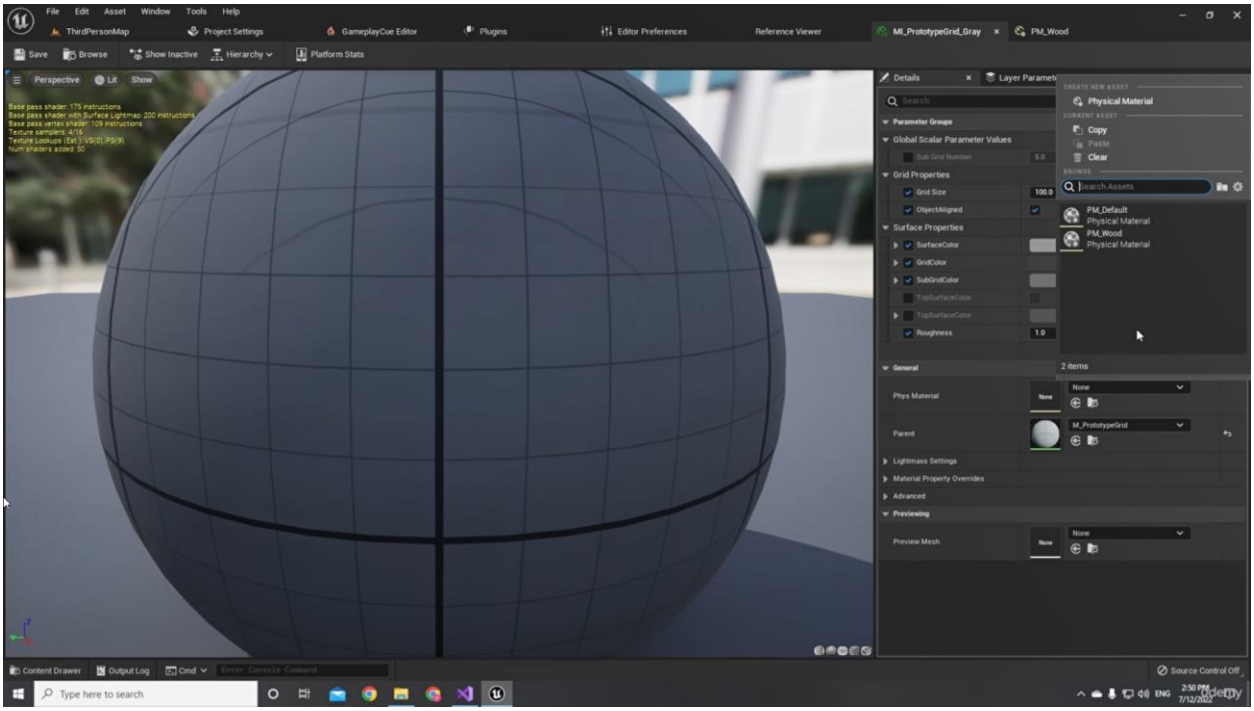


Then we create another one for Wood Physical Material

In project settings we add wood in surface type:



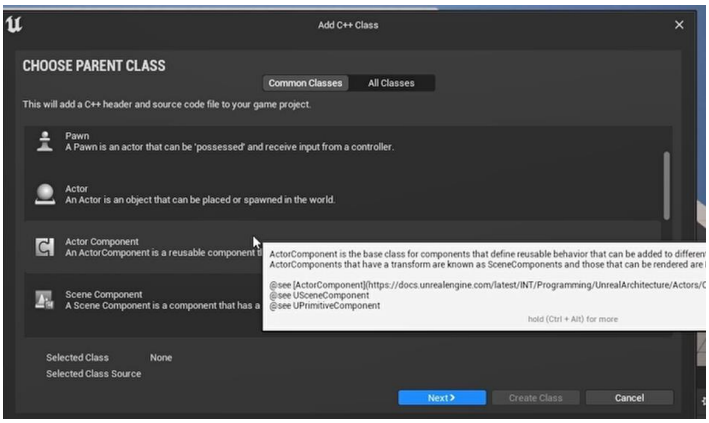
Then we have to assign the PhysicalMaterial to existing Materials.



5. Footsteps Component, Step Anim Notify

We now implement the FootstepsComponent that we are going use it with advanced material.

1. We create it from Actor Component Class.
2. Because the movement of the footsteps depends on the animation, we will need to specify a notify that trigger at



the right moment. So we create an AnimNotify class too.

3. We have to add our footsteps component to our character as the default sub-object.

```
UPROPERTY(BlueprintReadOnly)
class UFootstepsComponent* FootstepsComponent;
```

and in the cpp:

```
FootstepsComponent =
CreateDefaultSubobject<UFootstepsComponent>(TEXT("FootstepsComponent"));
```

4. For the footstepsNotify we will need an Enum to determent which foot is it now. And with that we would consider different foot sockets for our calculations. (this is in ActionGameTypes.h)

```
UENUM(BlueprintType)
enum class EFoot : uint8
{
	Left  UMETA(DisplayName = "Left"),
	Right UMETA(DisplayName = "Right")
};
```

In FootstepsComponent Class header we delete Tick function and in cpp we set *.bCanEverTick to false.

Also you should check the skeleton mesh for SocketName or make the component editable in the editor

```
#include "CoreMinimal.h"
#include "Components/ActorComponent.h"
#include "ActionGameTypes.h"
#include "FootstepsComponent.generated.h"


UCLASS( ClassGroup=(Custom), meta=(BlueprintSpawnableComponent) )
class ACTIONGAME_API UFootstepsComponent : public UActorComponent
{
	GENERATED_BODY()

public:
	// Sets default values for this component's properties
	UFootstepsComponent();

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

	UPROPERTY(EditDefaultsOnly)
	FName LeftFootSocketName = TEXT("foot_l");

	UPROPERTY(EditDefaultsOnly)
	FName RightFootSocketName = TEXT("foot_r");

public:

	void HandleFootstep(EFoot foot);
};
```

5. We create a public function to handle footsteps.

Somethings to note:

FCollisionQueryParams is a data structure in Unreal Engine that is used to specify parameters for collision queries. Collision queries are used to detect collisions between objects in a game.

FCollisionQueryParams contains information such as the collision channel to use for the query, whether to ignore certain actors or components, and whether to trace against complex collision shapes or simplified ones.

LineTraceSingleByChannel function on the game world object. This function is used to perform a line trace test against the world geometry to determine if there are any collisions along a given line.

In game engines, the game world object is an instance of a class that represents the game world and its contents. It typically contains all of the objects and entities that exist within the game environment, such as characters, enemies, items, terrain, and other game objects.

In Unreal Engine, the game world object is represented by the UWorld class. This class contains a variety of functions and data structures that allow you to interact with and manipulate the contents of the game world. For example, you can use the LineTraceSingleByChannel function to perform collision tests against the world geometry, or the SpawnActor function to create new game objects at runtime.

The UWorld class is created and managed by the engine, and is typically accessed through a pointer or reference to the current game instance. For example, you might use the GetWorld() function to retrieve a pointer to the current game world object from within an actor or controller class.

The game world object is a central concept in game development, as it **forms the foundation** upon which all other gameplay systems are built. By providing a unified representation of the game environment and its contents, the game world object enables developers to create complex and immersive game experiences that are rich in detail and interactivity.

Location is used as the starting point for the line trace test, and Location + FVector::UpVector * -50.0f is used as the end point. The FVector::UpVector is a constant vector pointing upwards in the world, and multiplying it by a negative scalar value causes the resulting vector to point downwards.

If you were to use different values such as 30 and -60, it would change the length of the line trace test, but as long as the line trace is still **extending** from a point slightly above the character's foot socket to a point below it, it should still work as intended. The important thing is that the **line trace test is long enough to reach the ground or other surface that the character is standing on**, so that it can detect collisions with the environment.

It's worth noting that the specific values used for the line trace test will depend on the scale of the game environment and the size of the objects within it. In some cases, it may be necessary to adjust the line trace length or other parameters to ensure that collisions are detected correctly. However, **in general**, using a **line trace test that extends from slightly above the character's foot to a point below it is a common technique for detecting collisions with the ground or other surfaces in a game.**

```
GetWorld()->LineTraceSingleByChannel(HitResult, Location, Location + FVector::UpVector * -50.0f, ECollisionChannel::ECC_WorldStatic, QueryParams)
```

Here's what each parameter in this function call does:

HitResult: a reference to an FHitResult object that will contain information about the collision if one occurs.

Location: the starting point of the line trace test. This is the location vector that was calculated in the previous code snippet, representing a position slightly above the character's foot socket.

Location + FVector::UpVector * -50.0f: the end point of the line trace test. This is calculated by adding an offset vector pointing downwards by 50 units to the Location vector. This creates a line trace that extends downwards from the character's foot socket.

ECollisionChannel::ECC_WorldStatic: the collision channel to use for the line trace test. This specifies which types of objects the line trace should collide with. In this case, the line trace will only collide with static geometry in the world.

QueryParams: an FCollisionQueryParams object that contains additional parameters for the line trace test. This could include things like whether to ignore certain actors or components, or whether to trace against complex collision shapes or simplified ones.

UGameplayStatics is a static class in Unreal Engine that provides a collection of utility functions for common gameplay tasks. It includes functions for things like playing sounds, spawning particles, and performing line traces, among others. By providing a centralized location for these types of functions, UGameplayStatics makes it easier for game developers to implement common gameplay mechanics without having to write all of the code from scratch.

```
UGameplayStatics::PlaySoundAtLocation(this, PhysicalMaterial->FootStepSound, Location, 1.f);
```

The specific function call you provided, UGameplayStatics::PlaySoundAtLocation, is used to play a sound at a specific location in the game world. The function takes several parameters:

this: a reference to the object that is calling the function. In this case, it appears to be a reference to the current object, which is likely an actor or component.

PhysicalMaterial->FootStepSound: the sound to be played. This is a reference to a sound asset that has been loaded into the game.

Location: the location in the game world where the sound should be played.

1.f: the volume at which the sound should be played. This is a scalar value between 0 and 1.

Other functions available in UGameplayStatics include:

- SpawnEmitterAtLocation: spawns a particle emitter at a specified location in the game world.
- ApplyDamage: applies damage to a specified actor.

- GetPlayerController: retrieves the player controller associated with a specific player.
- ApplyPointDamage: applies point damage to a specified actor at a specific location.
- GetAllActorsOfClass: retrieves an array of all actors in the game world that are instances of a specified class.

```
static TAutoConsoleVariable<int32> CVarShowFootsteps(
	TEXT("ShowDebugFootsteps"),
	0,
	TEXT("Draws debug info about footsteps")
	TEXT(" 0: off/n")
	TEXT(" 1: on/n"),
	ECVF_Cheat);
```

The code is defining a console variable using the TAutoConsoleVariable template class in Unreal Engine. Console variables are a way to expose internal game settings to the player or to developers, allowing them to adjust various game parameters without having to modify the game's source code.

Here's what each parameter in this TAutoConsoleVariable call does:

int32: the data type of the console variable. In this case, it's an integer value.

CVarShowFootsteps: the name of the console variable. This is the name that will be used to access the variable from the console or from code.

0: the default value of the console variable. This is the value that the variable will be set to if it is not explicitly set by the player or by code.

TEXT("Draws debug info about footsteps"): a brief description of what the console variable does. This text is displayed when the player or developer accesses the console variable documentation.

TEXT(" 0: off/n"): a description of the possible values for the console variable. In this case, the variable can be set to 0 (off) or 1 (on).

ECVF_Cheat: a flag that indicates that this console variable can only be modified by cheat commands. This prevents players from modifying the variable in unintended ways and affecting gameplay balance.

Overall, the TAutoConsoleVariable template class is a convenient way to define console variables in Unreal Engine, and can be used to expose a wide range of game settings and parameters to players and developers.

Now the code:

First we have the headers:

```
#include "ActorComponents/FootstepsComponent.h"
#include "PhysicalMaterial/AG_PhysicalMaterial.h"
#include "Kismet/GameplayStatics.h"
#include "ActionGameCharacter.h"
#include "DrawDebugHelpers.h"
```

```
void UFootstepsComponent::HandleFootstep(EFoot foot)
{
	if (AActionGameCharacter* Character = Cast<AActionGameCharacter>(GetOwner()))
	{
		const int32 DebugShowFootsteps = CVarShowFootsteps.GetValueOnAnyThread();

		if (USkeletalMeshComponent* Mesh = Character->GetMesh())
		{
			FHitResult HitResult;
			const FVector SocketLocation = Mesh->GetSocketLocation(foot == EFoot::Left ? LeftFootSocketName : RightFootSocketName);
			const FVector Location = SocketLocation + FVector::UpVector * 20;

			FCollisionQueryParams QueryParams;
			QueryParams.bReturnPhysicalMaterial = true;
			QueryParams.AddIgnoredActor(Character);

			if (GetWorld()->LineTraceSingleByChannel(HitResult, Location, Location + FVector::UpVector * -50.0f, ECollisionChannel::ECC_WorldStatic, QueryParams))
			{
				if (HitResult.bBlockingHit)
				{
					if (HitResult.PhysMaterial.Get())
					{
						UAG_PhysicalMaterial* PhysicalMaterial = Cast<UAG_PhysicalMaterial>(HitResult.PhysMaterial.Get());
						if (PhysicalMaterial)
						{
							UGameplayStatics::PlaySoundAtLocation(this, PhysicalMaterial->FootStepSound, Location,
1.f);
						}

						if (DebugShowFootsteps > 0)
						{
							DrawDebugString(GetWorld(), Location, GetNameSafe(PhysicalMaterial), nullptr,
FColor::White, 4.f);
						}
					}

					if (DebugShowFootsteps > 0)
					{
						DrawDebugSphere(GetWorld(), Location, 16, 16, FColor::Red, false, 4.f);
					}
				}
				else
				{
					if (DebugShowFootsteps > 0)
					{
						DrawDebugLine(GetWorld(), Location, Location + FVector::UpVector * -50.0f, FColor::Red, false, 4, 0, 1);
					}
				}
			}
			else
			{
				if (DebugShowFootsteps > 0)
				{
					DrawDebugLine(GetWorld(), Location, Location + FVector::UpVector * -50.0f, FColor::Red, false, 4, 0, 1);
					DrawDebugSphere(GetWorld(), Location, 16, 16, FColor::Red, false, 4.f);
				}
			}
		}
	}
}
```

```cpp
void UFootstepsComponent::HandleFootstep(EFoot foot)
{
    if (AActionGameCharacter* Character = Cast<AActionGameCharacter>(GetOwner()))
    {
        const int32 DebugShowFootsteps = CVarShowFootsteps.GetValueOnAnyThread();

        if (USkeletalMeshComponent* Mesh = Character->GetMesh())
        {
            FHitResult HitResult;
            const FVector SocketLocation = Mesh->GetSocketLocation(foot == EFoot::Left ? LeftFootSocketName : RightFootSocketName);
            const FVector Location = SocketLocation + FVector::UpVector * 20;

            FCollisionQueryParams QueryParams;
            QueryParams.bReturnPhysicalMaterial = true;
            QueryParams.AddIgnoredActor(Character);

            if (GetWorld()->LineTraceSingleByChannel(HitResult, Location, Location + FVector::UpVector * -50.0f, ECollisionChannel::ECC_WorldStatic, QueryParams))
            {
                if (HitResult.bBlockingHit)
                {
                    if (HitResult.PhysMaterial.Get())
                    {
                        UAG_PhysicalMaterial* PhysicalMaterial = Cast<UAG_PhysicalMaterial>(HitResult.PhysMaterial.Get());
                        if (PhysicalMaterial)
                        {
                            UGameplayStatics::PlaySoundAtLocation(this, PhysicalMaterial->FootstepSound, Location, 1.f);
                        }

                        if (DebugShowFootsteps > 0)
                        {
                            DrawDebugString(GetWorld(), Location, GetNameSafe(PhysicalMaterial), nullptr, FColor::White, 4.f);
                        }
                    }

                    if (DebugShowFootsteps > 0)
                    {
                        DrawDebugSphere(GetWorld(), Location, 16, 16, FColor::Red, false, 4.f);
                    }
                }
                else
                {
                    if (DebugShowFootsteps > 0)
                    {
                        DrawDebugLine(GetWorld(), Location, Location + FVector::UpVector * -50.0f, FColor::Red, false, 4, 0, 1);
                    }
                }
            }
            else
            {
                if (DebugShowFootsteps > 0)
                {
                    DrawDebugLine(GetWorld(), Location, Location + FVector::UpVector * -50.0f, FColor::Red, false, 4, 0, 1);
                    DrawDebugSphere(GetWorld(), Location, 16, 16, FColor::Red, false, 4.f);
                }
            }
        }
    }
}
```

---

6. Now time for our AnimNotify, it would have a trigger to handle FootstepsComponent

```cpp
#include "CoreMinimal.h"
#include "Animation/AnimNotifies/AnimNotify.h"
#include "ActionGameTypes.h"
#include "AnimNotify_Step.generated.h"

/**
 *
 */
UCLASS()
class ACTIONGAME_API UAnimNotify_Step : public UAnimNotify
{
    GENERATED_BODY()

public:

    virtual void Notify(USkeletalMeshComponent* MeshComp, UAnimSequenceBase* Animation) override;

    UPROPERTY(EditAnywhere)
    EFoot foot;
};
```

- We also create a Getter in our Character that our notify can get access to our FootstepsComponent from character.

```
class UFootstepsComponent* GetFootstepsComponent() const;
```

- and in character.cpp

```
UFootstepsComponent* AActionGameCharacter::GetFootstepsComponent() const
{
	return FootstepsComponent;
}
```

Then in our Notify we ask to call HandleFootstep function from our FootstepsComponent.

```
void UAnimNotify_Step::Notify(USkeletalMeshComponent* MeshComp, UAnimSequenceBase*
Animation)
{
	Super::Notify(MeshComp, Animation);

	check(MeshComp);

	AActionGameCharacter* Character = MeshComp ? Cast<AActionGameCharacter>(MeshComp-
>GetOwner()) : nullptr;
	if (Character)
	{
		if (UFootstepsComponent* FootstepsComponent = Character-
>GetFootstepsComponent())
		{
			FootstepsComponent->HandleFootstep(foot);
		}
	}
}
```

The reason for using check instead of if in this case is to provide an extra level of safety against potential programming errors.

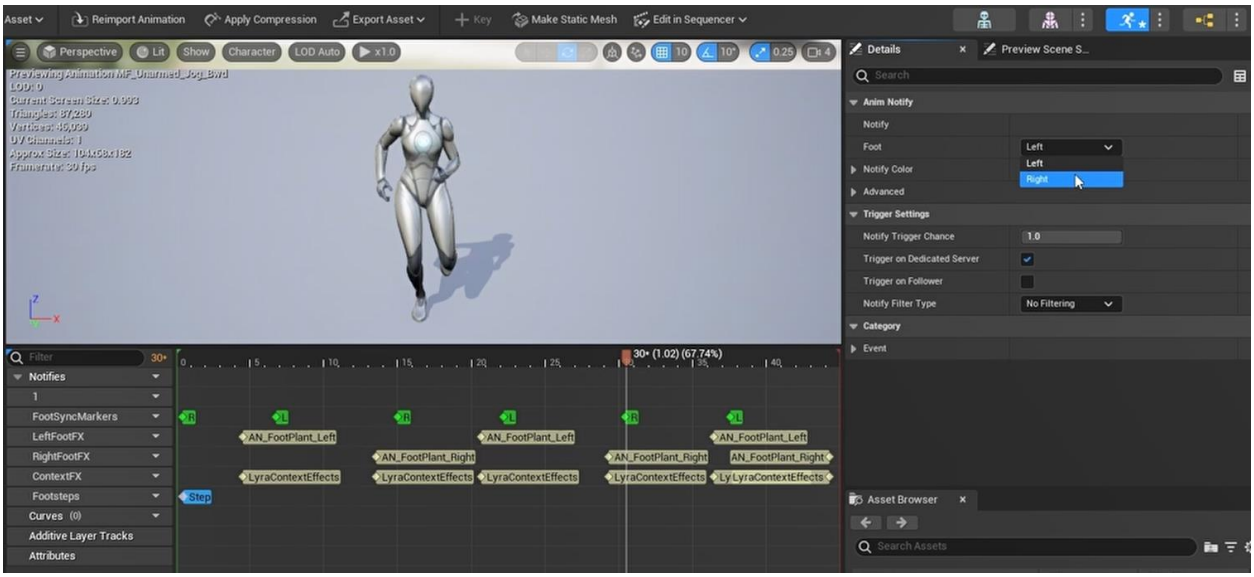
In Unreal Engine, it's possible for a pointer to be nullptr even when the code that assigns it should have ensured that it is not. This can happen due to a variety of reasons, such as memory allocation errors or incorrect initialization of objects.

By using check instead of if, the code is explicitly testing whether the pointer is valid, and if it's not, it's triggering a breakpoint and halting the game. This helps to catch potential programming errors early and prevent further execution of the code, which could lead to undefined behavior or even crashes.

On the other hand, using if to check whether the pointer is valid would allow the code to continue executing even if the pointer is nullptr. This could lead to undefined behavior or crashes further down the line, as the code assumes that the pointer is valid when in fact it is not.

It's worth noting that check is typically only used during development and testing. **In release builds**, check statements are disabled, and the code falls back to the equivalent if statement. This ensures that the final game binary does not contain any check statements, which could impact performance or cause unexpected behavior.

7. Now in your animations you have to add footsteps notify. and do it each time the foot touches the ground, also select the correct foot too.



8. Add footsteps sounds inside of your material if you haven't already.
9. You can debug with the command `ShowDebugFootsteps 1` in the editor.

Something noteworthy:

In Unreal Engine, GetNameSafe() and GetName() are both member functions of the UObject class that are used to retrieve the name of an object However, they differ in how they handle null pointers.

GetName() returns the name of the object as an FName. If the object is nullptr, calling GetName() will result in a runtime error.

GetNameSafe() returns the name of the object as an FString. If the object is nullptr, it returns the string "None" instead of crashing the program. This can be useful for cases where a null check is impractical or where a default value is needed for error handling.