DIGITAL ARTS & ENTERTAINMENT

Unreal Wwise Events

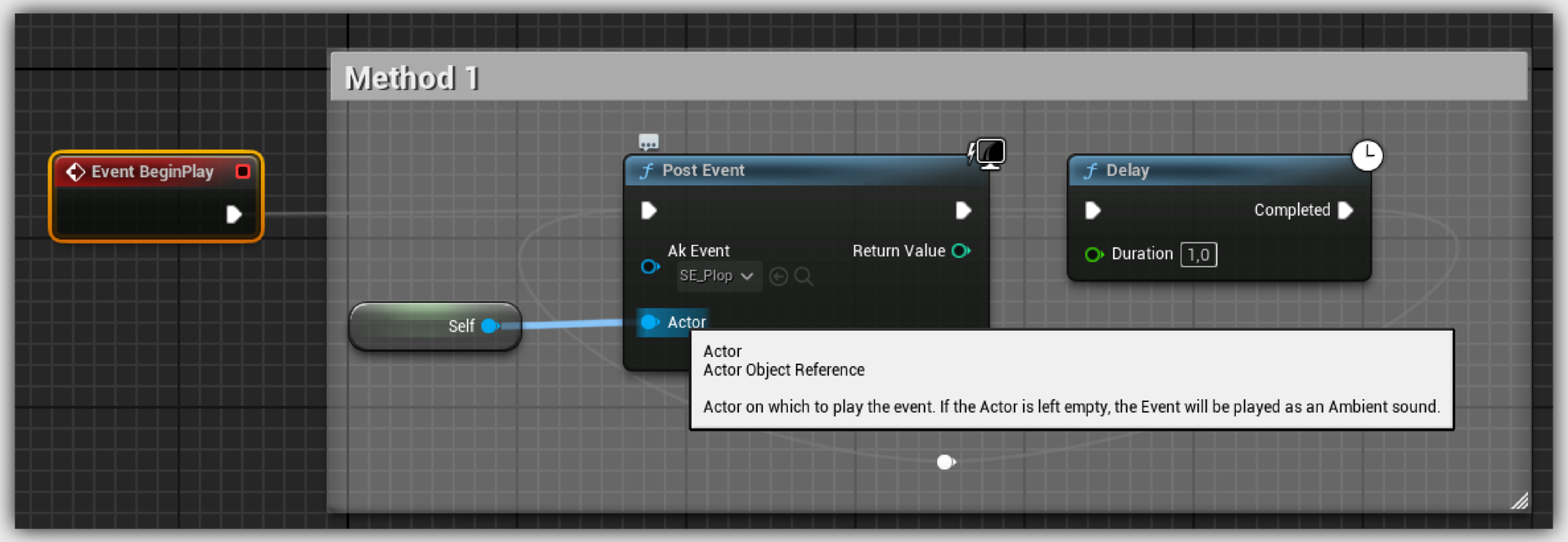
howest
university of applied sciences

Meltem Ozcelik (Concept: Amandine Gerard) / Daniil Shashenkov (Concept: Tooth Wu) / Joao Desager (Concept: Tan Zhi Hui)

Wwise Events

We've now played an ambient sound through a simple actor `AkAmbientSound`.

Obviously, we trigger these sound events via code as well. With visual scripting we can do it like this:



Todo: create the C++ variant of this.

Wwise Events

Just as we did in the first lab for the enhanced input, we now need to add the Wwise modules to our C++ project to be able to use them. So in your Build.cs file add `AkAudio` and `WwiseSoundEngine`

```
using UnrealBuildTool;
public class Playground : ModuleRules
{
    public Playground(ReadOnlyTargetRules Target) : base(Target)
    {
        PCHUsage = PCHUsageMode.UseExplicitOrSharedPCHs;
        PublicDependencyModuleNames.AddRange(new string[] {
            "Core", "CoreUObject", "Engine", "InputCore", "EnhancedInput",
            "AkAudio", "WwiseSoundEngine"
            });

        PrivateDependencyModuleNames.AddRange(new string[] {  });
    }
}
```

Wwise Events

Create a new Actor class, have a member variable `PlopEvent` of the type `UAKAudioEvent*`

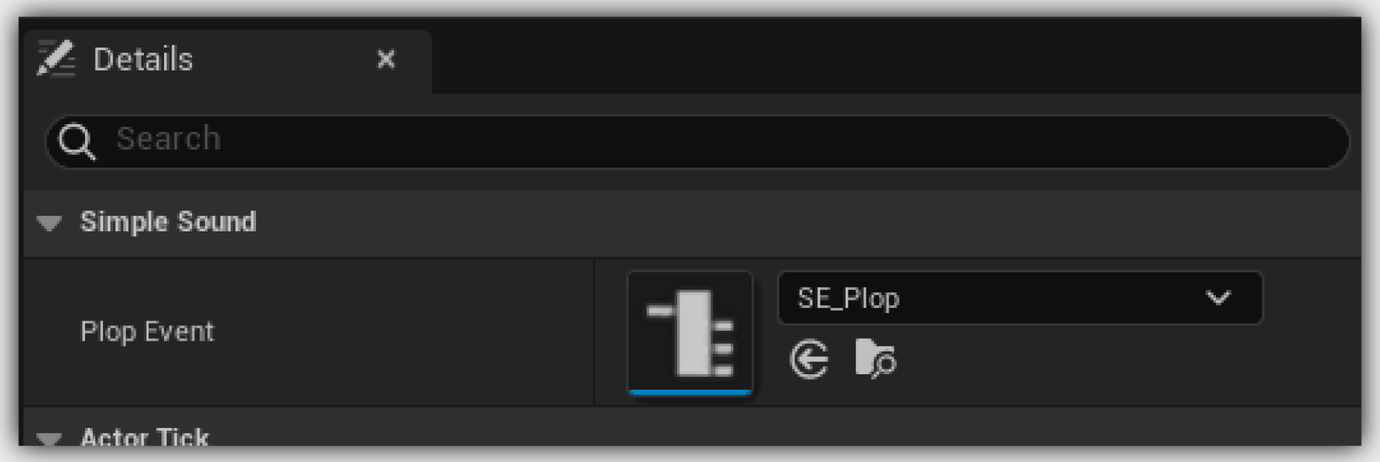
```
UCLASS()
class AMethod1 : public AActor
{
    GENERATED_BODY()
public:
    ASimpleSound();
protected:
    virtual void BeginPlay() override;
private:
    UPROPERTY(EditAnywhere, meta = (AllowPrivateAccess = "true"))
    class UAkAudioEvent* PlopEvent;
};
```

And implement `BeginPlay`

```
void AMethod1::BeginPlay()
{
    Super::BeginPlay();

    FOnAkPostEventCallback nullCallback;
    UAkGameplayStatics::PostEvent(PlopEvent, this, 0, nullCallback);
}
```

Wwise Events

Create a blueprint class from the C++ class and assign the SE_Plop event in the blueprint editor.



Place the blueprint in the scene and hit play, you should hear the plop.

Now make it loop.

Wwise Events

A `Delay` function does not really exist; delays are timers you set to go off at certain times via a global timer manager.

```
FTimerHandle UnusedHandle;
GetWorldTimerManager().SetTimer(
        UnusedHandle, this, &ASimpleSounds::LoopSound, PauseInBetween, false);
```

https://docs.unrealengine.com/5.3/en-US/gameplay-timers-in-unreal-engine/

Wwise Events

We can now call this function in BeginPlay to have it loop:

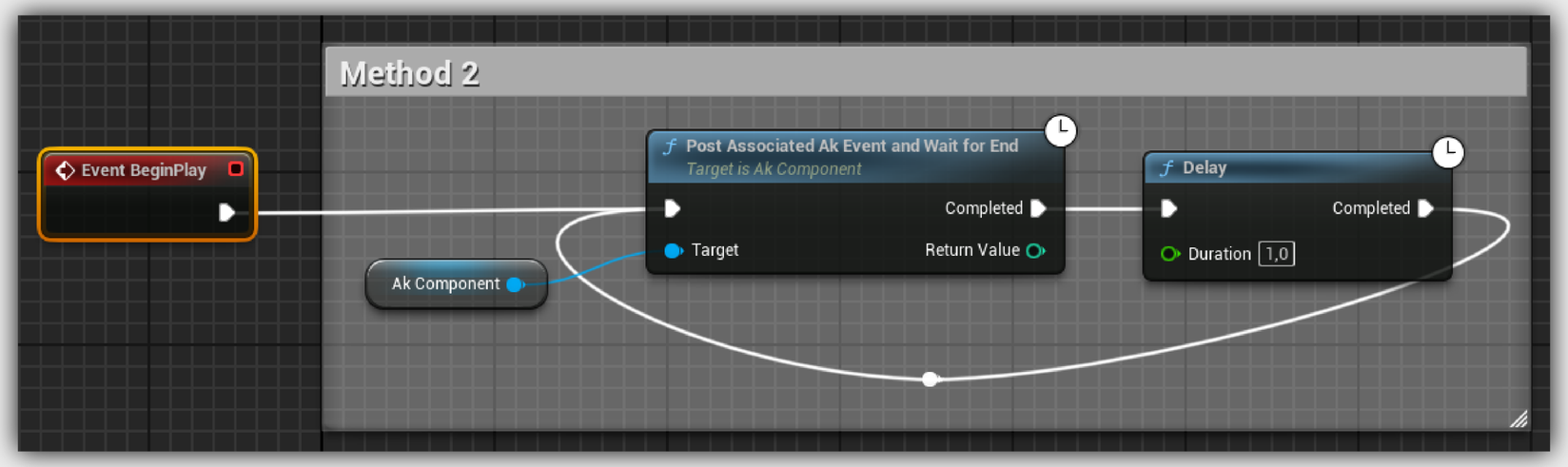
```
void AMethod1::LoopSound()
{
    FOnAkPostEventCallback nullCallback;
    UAkGameplayStatics::PostEvent(PlopEvent, this, 0, nullCallback);

    FTimerHandle UnusedHandle;
    GetWorldTimerManager().SetTimer(
        UnusedHandle, this, &AMethod1::LoopSound, PauseInBetween, false);
}
```

Notice that `FOnAkPostEventCallback`, we'll come back to that later.

Wwise Events - method 2

The previous approach enables us to play a simple sound. But if we want more control like for example occlusion and attenuation we'll need something more advanced.

Better is to use the "Ak" component. We can add that component to an actor and then we can use this function:



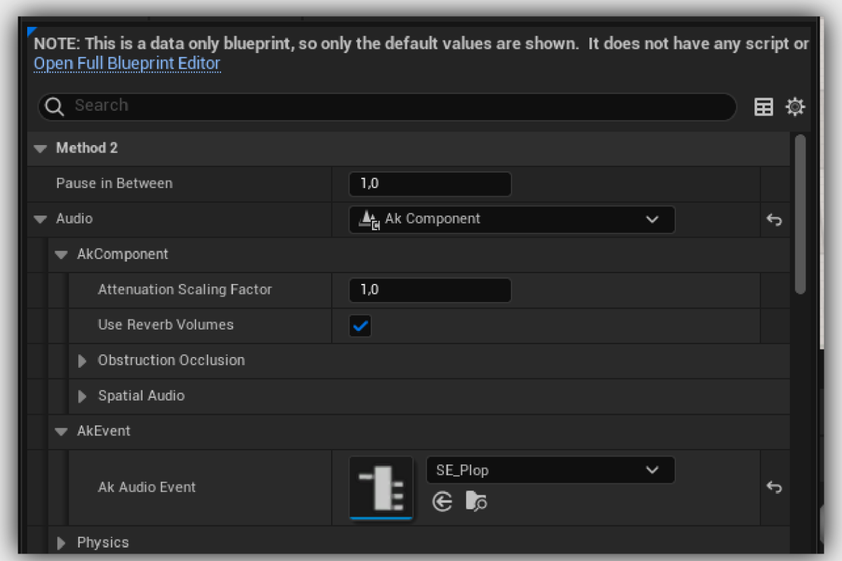
Write this in C++. Create an actor class that performs this exact code.

Wwise Events - method 2

We'll need an AkComponent now

```
AMethod2::AMethod2()
{
    PrimaryActorTick.bCanEverTick = false;

    RootComponent = CreateDefaultSubobject<USceneComponent>(TEXT("Root"));

    Audio = CreateDefaultSubobject<UAkComponent>(TEXT("Audio"));
    Audio->SetupAttachment(RootComponent);
}
```

Create a blueprint class from this and set the event on the component:



Wwise Events - method 2

With that we could do something like

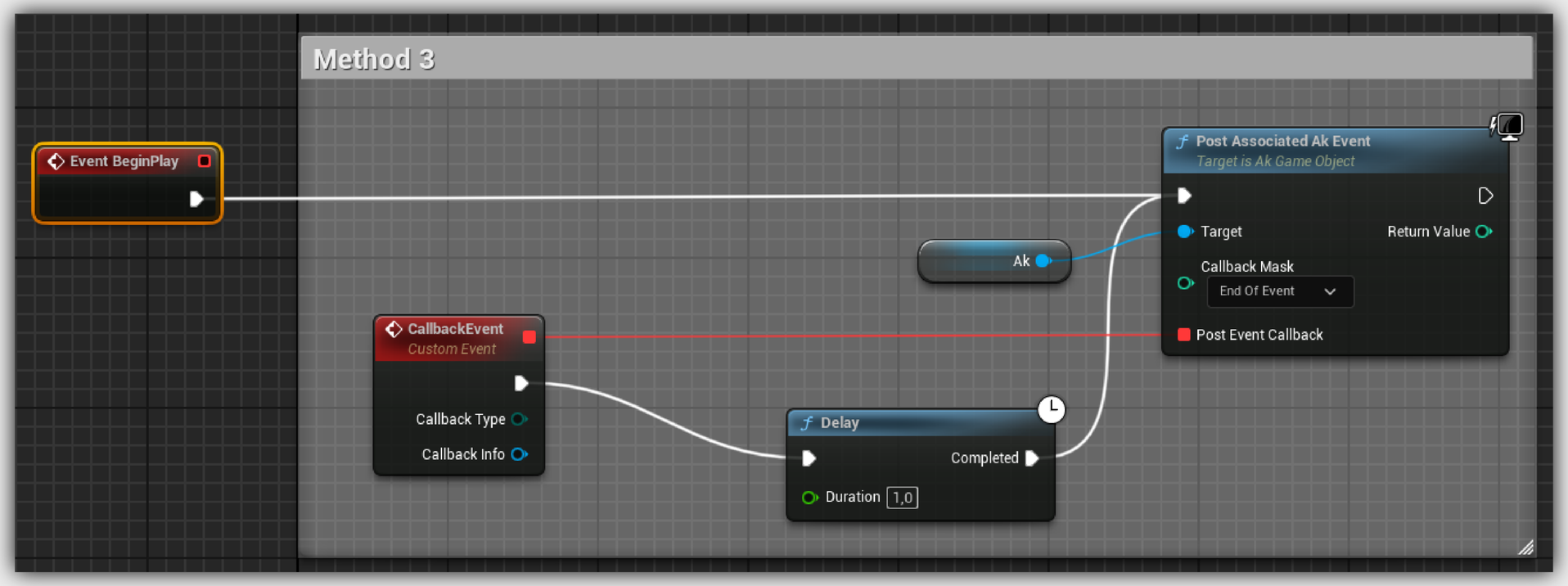
```
void AMethod2::WaitAndPlayAgain()
{
    FTimerHandle UnusedHandle;
    GetWorldTimerManager().SetTimer(
        UnusedHandle, this, &AMethod2::PlaySound, PauseInBetween, false);
}

void AMethod2::PlaySound()
{
    FLatentActionInfo info;
    info.CallbackTarget = this;
    info.ExecutionFunction = "WaitAndPlayAgain";
    info.Linkage = 0;
    Audio->PostAssociatedAkEventAndWaitForEnd(info);
}
```

(`BeginPlay` calls the function `PlaySound`)

However, this is becoming overly complicated, the above approach is more intended for use by blueprints

Wwise Events - method 3

There is a third approach:



Write this in C++. Create an actor class that performs this exact code.

Wwise Events - method 3

The class setup is very similar to Method 2, but the methods change a bit.

```
void AMethod3::WaitAndPlayAgain(EAkCallbackType CallbackType, UAkCallbackInfo* CallbackInfo)
{
    FTimerHandle UnusedHandle;
    GetWorldTimerManager().SetTimer(
        UnusedHandle, this, &AMethod3::PlaySound, PauseInBetween, false);
}

void AMethod3::PlaySound()
{
    FOnAkPostEventCallback EndSoundDelegate;
    EndSoundDelegate.BindUFunction(this, "WaitAndPlayAgain");
    Audio->PostAssociatedAkEvent(AK_EndOfEvent, EndSoundDelegate);
}
```

It's cleaner, and we get more info in the callback. Remember that `FOnAkPostEventCallback`?

Wwise Events - method 3

We could write it more like Method1, where we did not care about the callback:

```
void AMethod3::LoopSound()
{
    FOnAkPostEventCallback nullCallback;
    Audio->PostAssociatedAkEvent(0, nullCallback);

    FTimerHandle UnusedHandle;
    GetWorldTimerManager().SetTimer(
        UnusedHandle, this, &AMethod3::LoopSound, PauseInBetween, false);
}
```

Then what are the pros and cons of these two approaches?

Wwise Events

These are some interesting webpages you could read about this topic, however some parts of them contain outdated versions of both Wwise and Unreal and no longer apply.

https://alessandrofama.com/tutorials/wwise/unreal-engine/events

https://blog.audiokinetic.com/en/coding-wwise-in-ue-for-beginners/

Unreal Events

Up until now, we went C++ all the way. Is it better to use either C++ coding or Blueprint visual scripting?

Unreal Events

Up until now, we went C++ all the way. Is it better to use either C++ coding or Blueprint visual scripting?

The answer is **both**.

The game and gameplay logic is often done in C++, while the visuals, the representation of the logic can be done with visual scripting.

For example our door. Game logic wise we need to know whether the door is open or not. How that door opens, if there's a sound playing while opening, if there is a particle effect playing while opening, (etc) is of little to no concern to the game logic.

All we need to do is to inform the blueprint that the change has happened and leave it up to the artists and sound designers what they do with it.

We can use either

- Blueprint Native Event
- Dynamic Multicast Delegate

Blueprint Native Event

In the header of our Door class we can add:

```
UFUNCTION(BlueprintNativeEvent, Category = Door)
void OnOpen(float Duration);

void OnOpen_Implementation(float Duration);
```

And in the cpp

```
void ADoor::OnOpen_Implementation(float Duration)
{
    UE_LOG(LogTemp, Display, TEXT("Test"));
}
```

Notice we only implemented the second function. This one will be called if the Blueprint class **did not override** OnOpen.

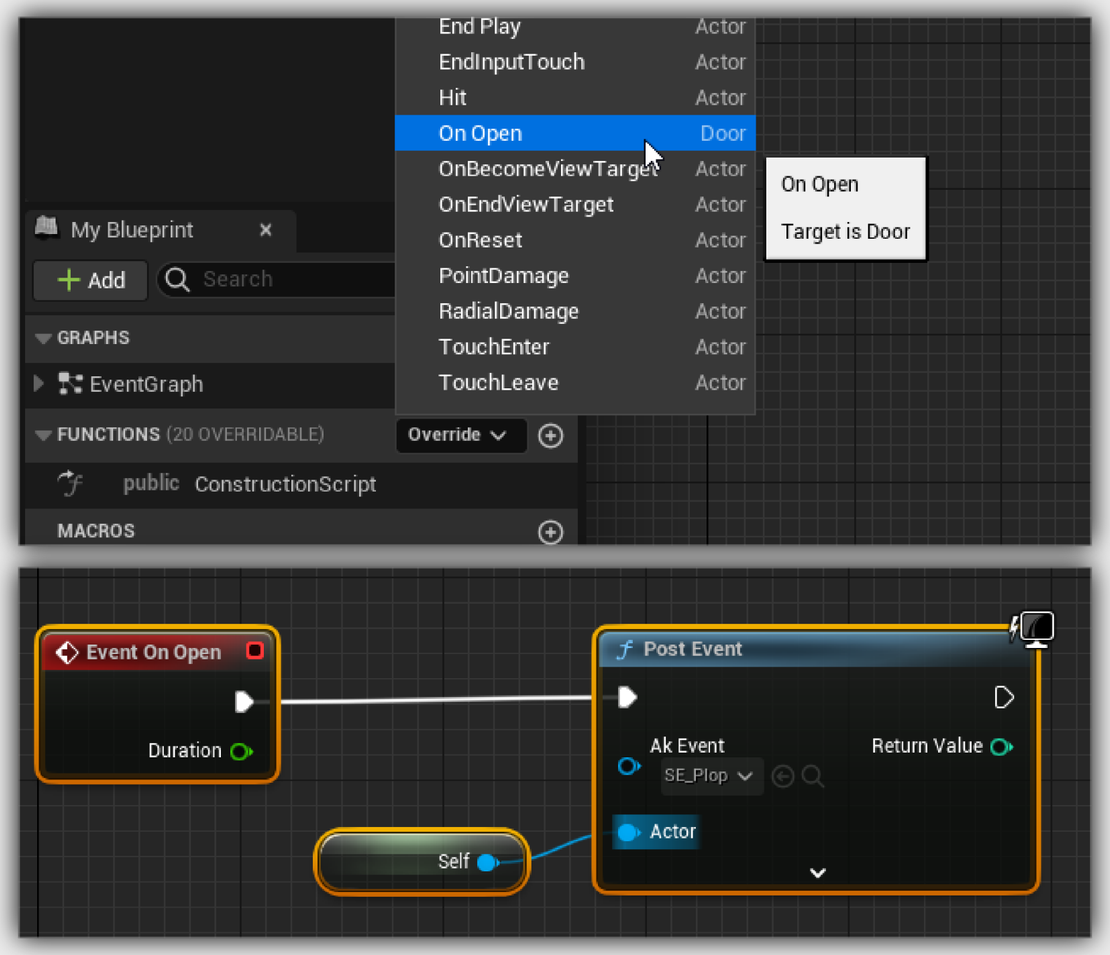
In our ToggleDoor function we call this OnOpen:

```
void ADoor::ToggleDoor()
{
    if (IsOpen)
        DoorTimeline->Reverse();
    else {
        DoorTimeline->Play();
        OnOpen(SwingDuration);
    }
    IsOpen = !IsOpen;
}
```

Verify your code by opening a door, it should log "Test" in the console.

Blueprint Native Event

You can now override that function if you so desire.



Notice that "Test" is not logged anymore in the console! We are overriding the function (Event is an unfortunate name for this).

Dynamic multicast delegate

A dynamic multicast delegate is exactly like an event as we saw in the first lab, but it's not tied to a specific type. In the Door header file, add:

```
DECLARE_DYNAMIC_MULTICAST_DELEGATE_OneParam(FDoorDelegate, float, Duration);

UPROPERTY(BlueprintAssignable, Category = "Door")
FDoorDelegate OpenDelegate;
```

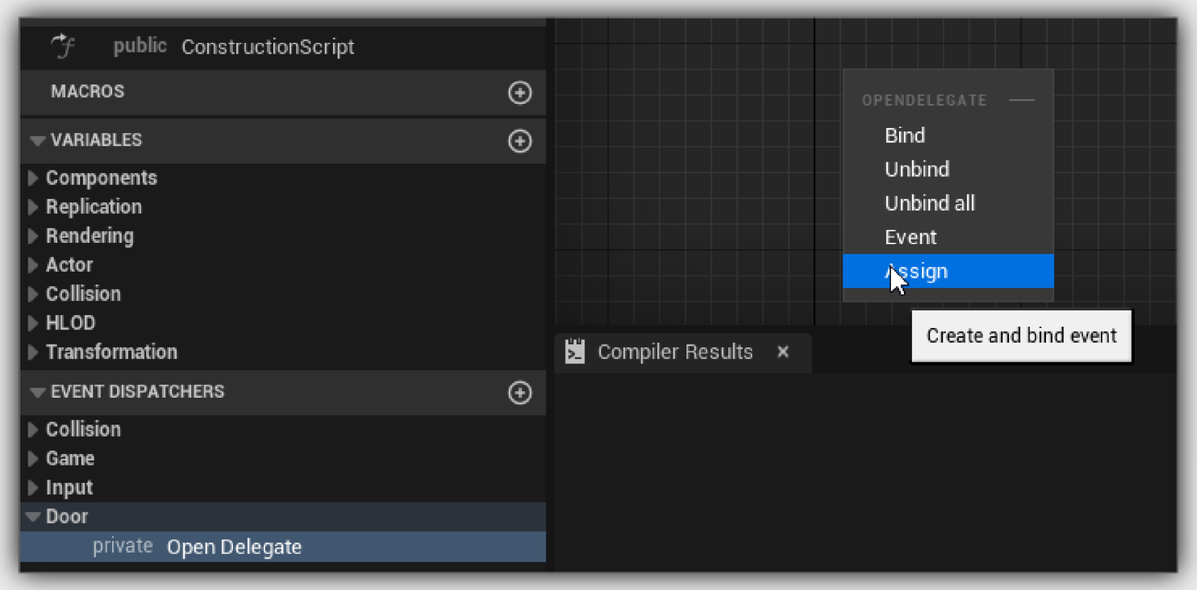
Broadcast it in the `ToggleDoor` function:

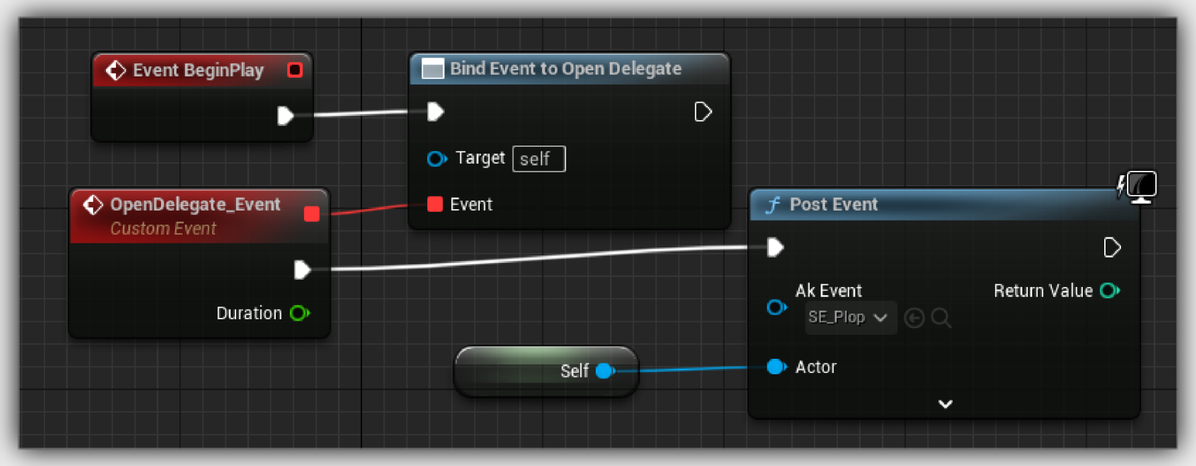
```
void ADoor::ToggleDoor()
{
    if (IsOpen)
        DoorTimeline->Reverse();
    else {
        DoorTimeline->Play();
        OpenDelegate.Broadcast(SwingDuration);
    }
    IsOpen = !IsOpen;
}
```

You can now add functions to this delegate like we did with the lever, but also...

Dynamic multicast delegate

Assign a function in the blueprint





Notice that we add the event handler in the BeginPlay function, just as we would do in C++.

With this approach all other event handlers that subscribed to this event will be executed too.