How to build a Custom Audio Editor with Unreal Engine: A Simple Tool That Lets You Place, Edit, and Spatialize Sound in VR

Unreal Engine* from Epic Games has a powerful virtual reality (VR) editor option, but something they did not include is the ability to edit and place sounds while inside VR. It can be troublesome to have to constantly restart the editor after adjusting a sound to test what it sounds like in VR. So we decided to create a sound editor that allows game developers and sound designers to quickly place, edit, and test spatialized sound inside VR! This will prevent the user from having to constantly enter and exit the editor.

System requirement:
-Unreal Engine 4.18.1 or greater
-Visual Studio* 2017
-HTC Vive*

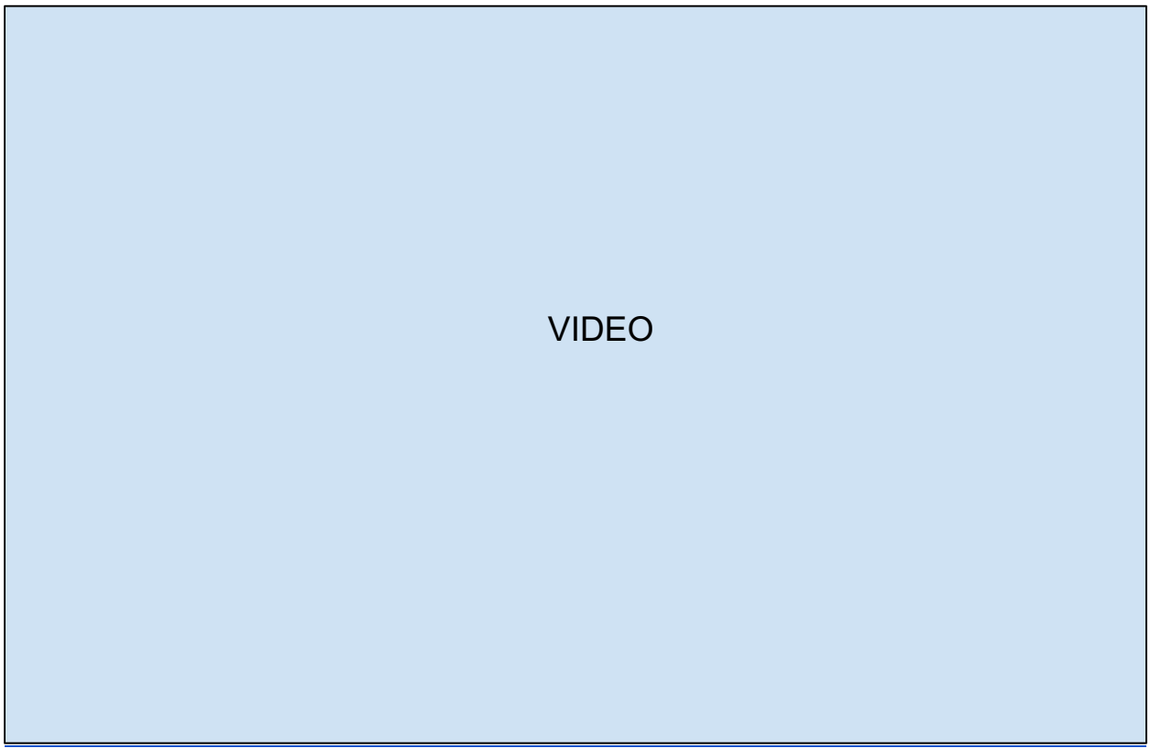
What you'll learn:
-Motion controller interaction
-How to create a custom C++ class
-VR UI
-Saving editor changes
-Sound spatialization parameters

Below, we will walk you through step-by-step to demonstrate how we made this custom audio editor tool for Unreal Engine from start to finish:

Project link download

Before we begin, you need to do a couple of things. Download and unzip the project folder. You also need to make sure you have at least version 4.18.1 of Unreal Engine installed.

When you have downloaded and unzipped the folder, right-click on Intel_VR_Audio_Tools.uproject and select "Generate Visual Studio project files." After that completes, open the project. A popup that says "Missing Intel_VR_Audio_Tools Modules" will appear. Click "Yes" to start the rebuild; this should take less than 20 seconds. This is needed because of how you are dynamically finding .wav files that have been added to the project, which will be explained in the Custom C++ Class section.

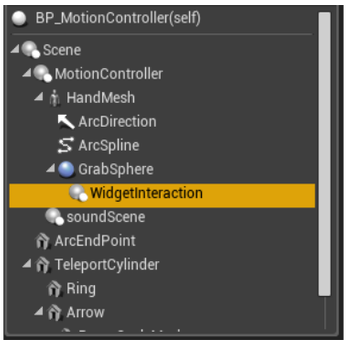


Setting up the VR player:

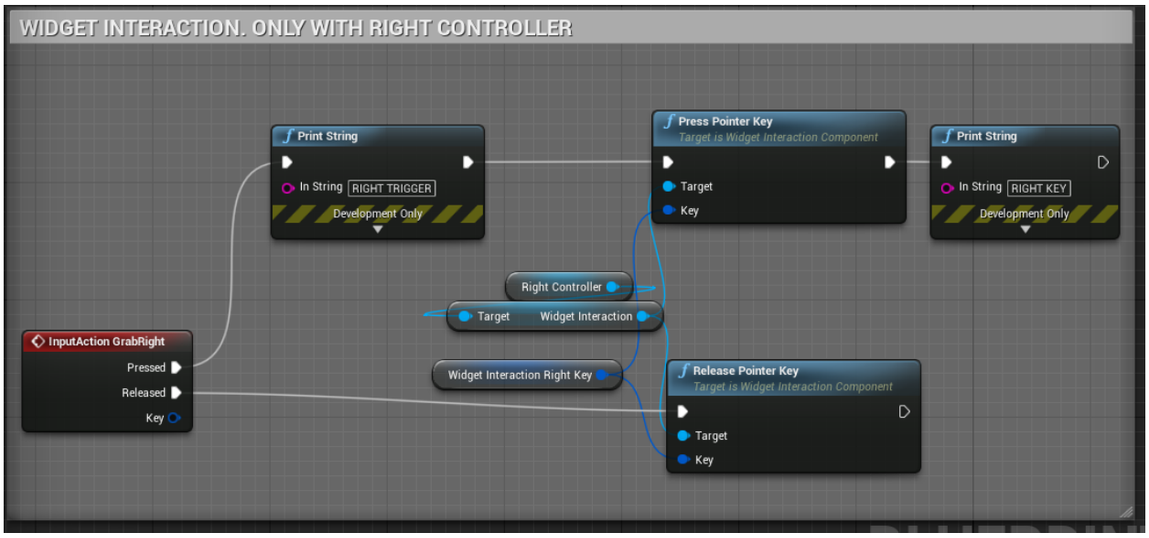
We start with Unreal's VR template and choose the MotionControllerPawn as our pawn, which has motion control set up and allows movement by teleporting.

Motion Controller Interaction:

Before the motion controller can interact with 3D widgets, a WidgetInteraction component needs to be added to BP_MotionController, which is located in the VirtualRealityBP folder. We also need to add a scene component for the sound selector widget, called soundScene.



Press and Release Pointer keys are attached to the event called when the right trigger is pulled. We need to add to the MotionControllerPawn, which is also located in the VirtualRealityBP folder.



Custom C++ Class:

The reason for rebuilding the project is because while making this tutorial, the issue of knowing the names and locations of the sounds and dynamically updating a widget to match all those files appeared daunting. Luckily, Unreal Engine has some stuff to help us out.

The IntelSoundComponent is a C++ class that can be added to any blueprint to dynamically locate and load a .wav file into a USoundWave, which is how Unreal loads a sound file.

First, we right-click in the content browser and create a new C++ class named IntelSoundComponent. This action creates an IntelSoundComponent.cpp file and an IntelSoundComponent.h file.

Next, we add some includes which are needed to locate and manage files.

Includes added in IntelSoundComponent.cpp are Paths.h, FileManager.h and Runtime/Engine/Classes/Sound/SoundWave.h (which for some reason need everything before SoundWave.h).

Image (IntelSoundComponent.cpp)

```
#include "IntelSoundComponent.h"
#include "Paths.h"
#include "FileManager.h"
#include "Runtime/Engine/Classes/Sound/SoundWave.h"

bool exists;
FString dir, soundDir;
TArray<FString> soundFiles;

// Sets default values for this component's properties
UIntelSoundComponent::UIntelSoundComponent()
{
	// Set this component to be initialized when the game starts, and to
be ticked every frame.  You can turn these features
	// off to improve performance if you don't need them.
	PrimaryComponentTick.bCanEverTick = true;

	//Empty soundFiles TArray. Easiest way if new wave files are added.
	soundFiles.Empty();

	//the way Unreal Engine calls the project's root directory
	dir = FPaths::ProjectDir();

	//Combining Root with the folder location for the sounds.
	//This could probably be an external folder if needed with the help of
( IPlatformFile& PlatformFile =
FPlatformFileManager::Get().GetPlatformFile(); )
	soundDir = dir + "Content/Sounds";

	//UE4 returns a bool if the directory exists or not.
	exists = FPaths::DirectoryExists(soundDir);

}
```

Now we create a a bool named exists; 2 FString variables named dir and SoundDir; and a TArray of FStrings named soundFiles. Since soundFiles is a TArray, we are able to call soundFiles.Empty(); which empties the TArray. We believe it's also the fastest way if new wave files are added. Then, we set FString dir to FPaths::ProjectDir(); (which gives the root location of the project). Next, we set FString soundDir to dir + "Content/Sounds" because that is the folder we put our .wav files into. FPaths has another method that can check if a directory exists, so we set our bool to exists = FPaths::DirectoryExists(soundDir);.

Image (IntelSoundComponent.cpp)

```
// Called when the game starts
void UIntelSoundComponent::BeginPlay()
{
	Super::BeginPlay();

	//UE4 way of managing files
	IFileManager &fileManager = IFileManager::Get();

	//UE_LOG(LogTemp, Warning, TEXT("%s"), &fileManager);


	if (exists == true){

		//Extensions to sound files. Was using .wav, but .uasset seems
to work when there is and isn't an editor.
		FString ext = "/*.wav";
```

```
                FString ext2 = "/*.uasset";

                //path = FPaths::ProjectDir() + Content/Sounds + /*.uasset
                FString path = soundDir + ext2;

                //This finds file in the given array, with the given path
                //the true bool is saying to look for files while false bool
is saying to not look for directories
                fileManager.FindFiles(soundFiles, *path, true, false);
```

On BeginPlay() we start by instantiating IFileManager by using IFileManager &fileManager = IFileManager::Get();. We do this to debug and test if the .wav files are being found with fileManager.FindFiles, which search for .uassets instead of the .wav files we were using before, because .uassets are more reliable when sharing projects.
Image (IntelSoundComponent.cpp)

```
//Setting soundFileArray to soundFiles to pass into blueprint.
void UIntelSoundComponent::soundArray(TArray<FString> &soundFileArray) {

        soundFileArray = soundFiles;

}

//loading a wav file as a USoundWave so Unreal can set the sound chosen with
LoadObject<USoundWave> for blueprint
USoundWave* UIntelSoundComponent::setWavToSoundWave(const FString &fileName)
{

        USoundWave* swRef;
        FString name = fileName;

        swRef = LoadObject<USoundWave>(nullptr, *name);

        return swRef;

}
```

Lastly, in the .cpp, we create two functions that will be exposed as blueprint nodes. SoundArray (which passes the soundFiles TArray into blueprints) and setWavToSoundWave (which took a while for us to figure out because we had to find a way to dynamically reference a .wav file in a way that Unreal could understand, which is a USoundWave). For this problem we discovered LoadObject. This function loads an object at runtime into any type that we set, if possible. For us, it was LoadObject(nullptr, *name);—*name being the sound that was chosen by VR player.

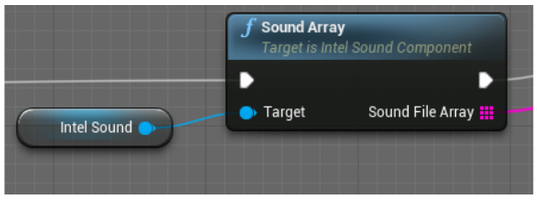
In the IntelSoundComponent.h we create two UFUNCTIONS as a way to make the two functions in the .cpp blueprint callable.

Image (IntelSoundComponent.h)

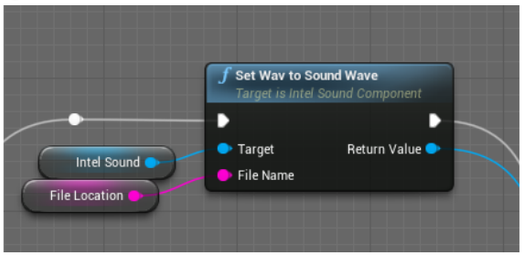
```
#include "CoreMinimal.h"
#include "Components/SceneComponent.h"
#include "Runtime/Engine/Classes/Sound/SoundWave.h"
#include "IntelSoundComponent.generated.h"



UCLASS( ClassGroup=(Custom), meta=(BlueprintSpawnableComponent) )
class INTEL_VR_AUDIO_TOOLS_API UIntelSoundComponent : public USceneComponent
{
        GENERATED_BODY()


public:
```

```
        // Sets default values for this component's properties
        UIntelSoundComponent();

        //Blueprint function to expose soundFiles into blueprint.
        UFUNCTION(BlueprintCallable, Category = IntelAudio)
                void soundArray(TArray<FString> &soundFileArray);

        //Blueprint function passing a wav converted in USoundwave into
blueprint.
        UFUNCTION(Category = IntelAudio, BlueprintCallable)
                USoundWave* setWavToSoundWave(const FString &fileName);
```

Blueprint function to expose sound files into blueprint.

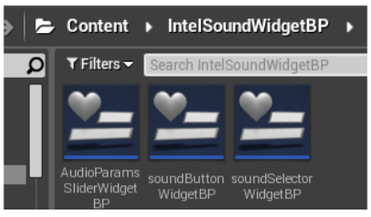


Blueprint function passing a .wav file converted in USoundWave into blueprint.

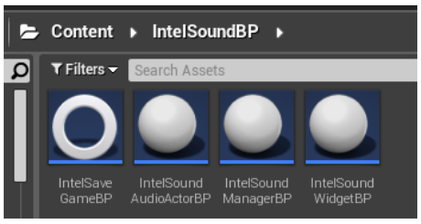


Setting up UI:

We need to set up three UMG widgets.

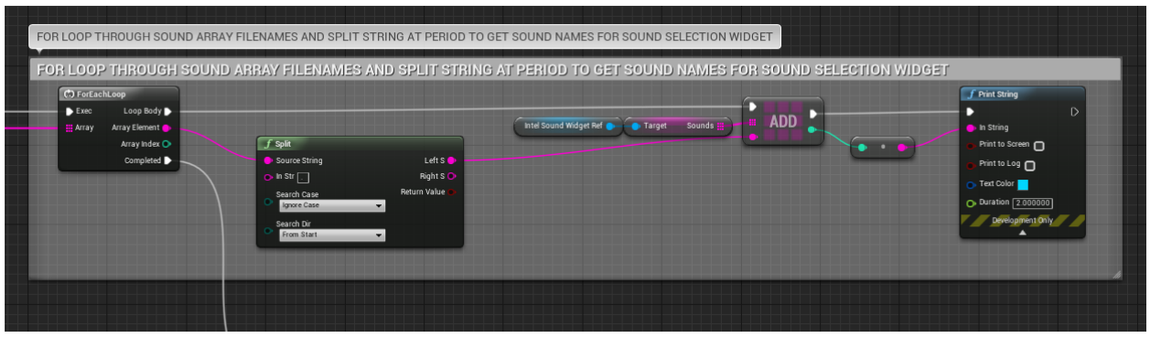


We create the blueprints needed to manage those UMG widgets.



We have a couple of widgets for this project. AudioParamsSliderWidget is the widget that pops up when a sound is selected. soundButtonWidgetBP is just a button widget for the sounds in the Content/Sounds folder. We put a widget called soundSelectorWidgetBP in the level by having an actorBP we create called IntelSoundWidgetBP get the sounds from the SoundArray C++ node and populate soundSelectorWidgetBP with soundButtonWidgetBPs. (We could do this dynamically but then we would have to get a reference to the newly spawned actor every time we began play.) All this happens in the IntelSoundManagerBP, which we also placed in the level from the start.

Image (IntelSoundManagerBP)

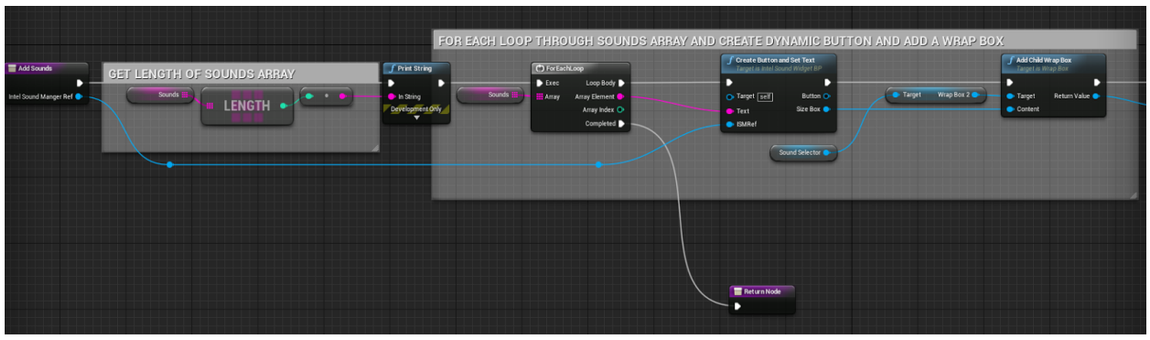


In the image above, we get the soundFiles TArray of FStrings and split at the period in the name of the (name of sound).wav. We send that string into an array of strings in IntelSoundWidget to name the buttons being dynamically populated.

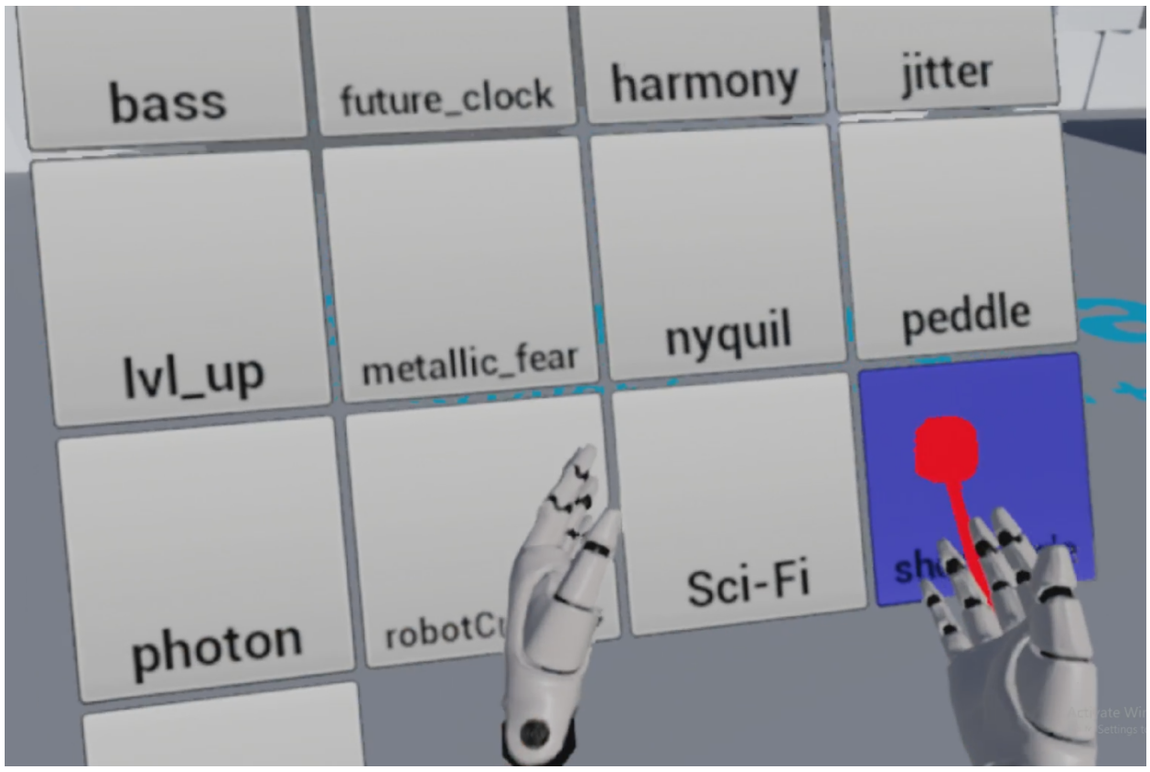
Image (IntelSoundWidgetBP)

In the IntelSoundWidgetBP we spawn the soundUI,

add sounds,

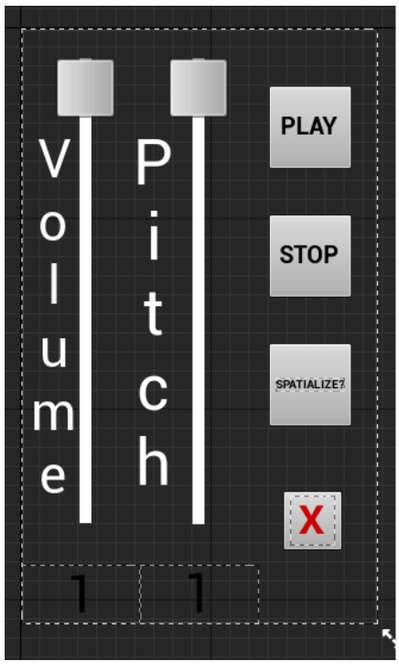


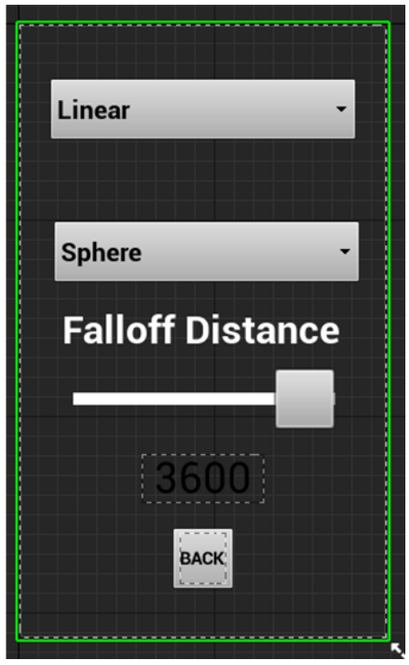
and if we don’t use the Set Widget node the widget would spawn but not be visible in game.



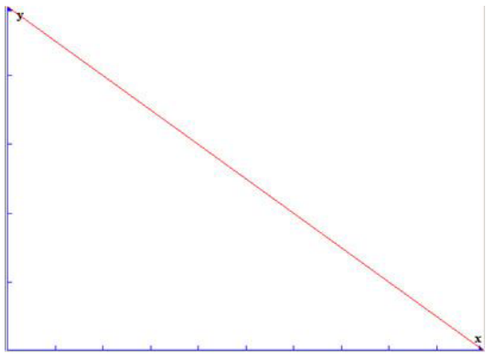
Sound Parameters:

Once the player selects a sound from the widget an IntelSoundAudioActorBP actor will spawn. In this actor we see the AudioParamsSliderWidgetBP, and if Spatialize? is clicked, three attenuation settings will be exposed to be changed through the widget.

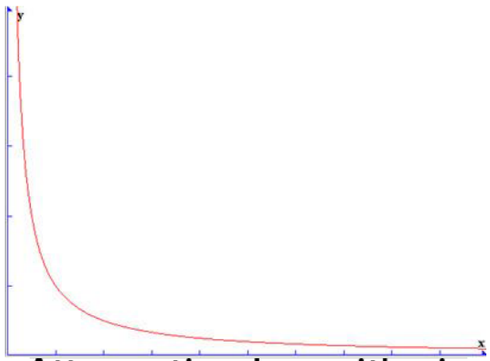




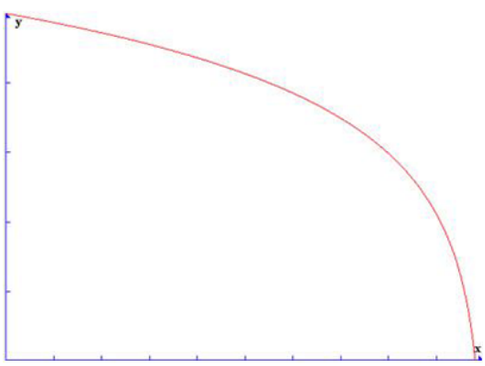
Sound attenuation is essentially the ability of a sound to lower in volume as the player moves away from it.



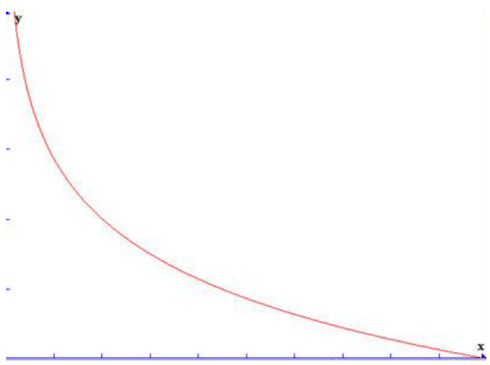
Attenuation Linear



Attenuation Logarithmic



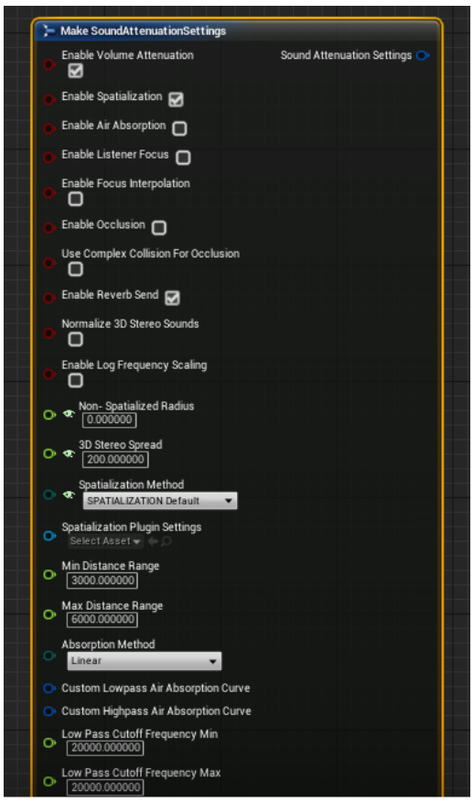
Attenuation LogReverse

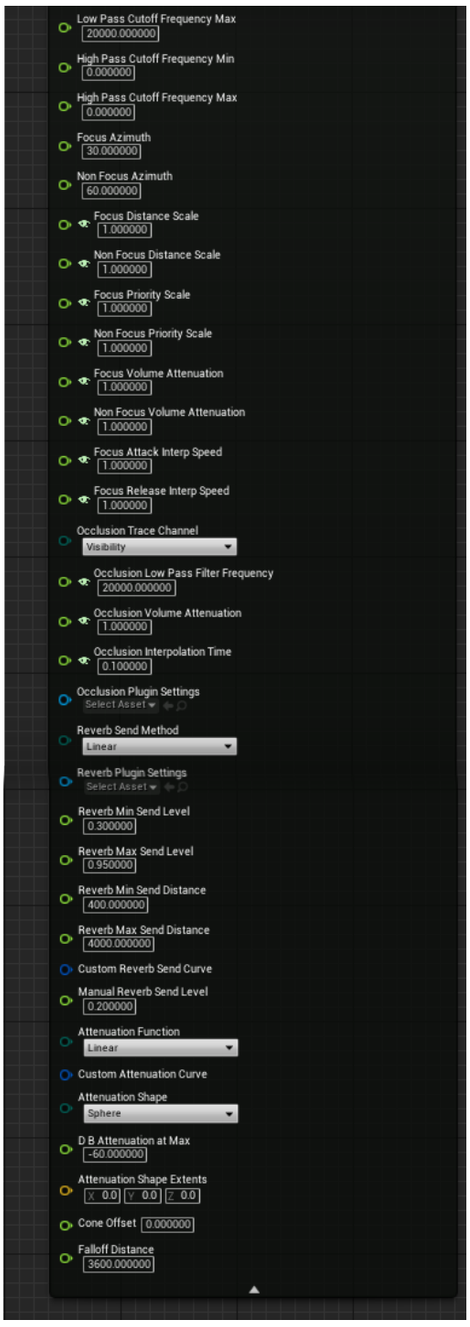


Attenuation Inverse

The three settings exposed are Attenuation Function, Attenuation Shape, and the Falloff Distance.

There are plenty more settings that could be exposed with more time. Here are images of the Attenuation Setting struct in Unreal.





We believe the three settings we chose are the most basic and fundamentally needed settings. Showing debug lines when changing settings is something we are working on. We are looking for a way to use the attenuation setting debug lines Unreal uses to show attenuation in the editor in the game, but we have not found that answer. So, we might get the shape extents of the chosen attenuation shape and function and use the Unreal built-in draw debug lines nodes.

Saving on Exit:

When we exit the game and have spawned sounds, moved them around, and played with the audio parameters, we save all the variables that are important using IntelSaveGameBP through IntelSoundAudioActorBP.

Image (IntelSaveGameBP)

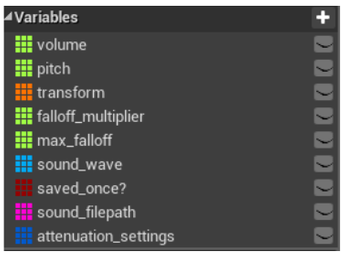


Image (IntelSoundManagerBP)

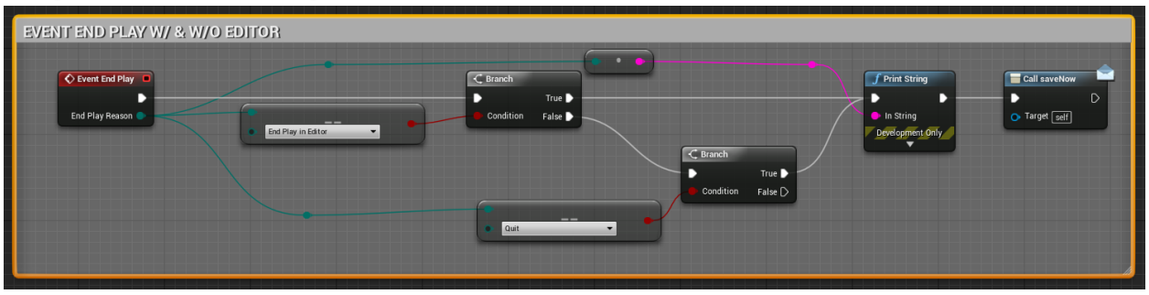
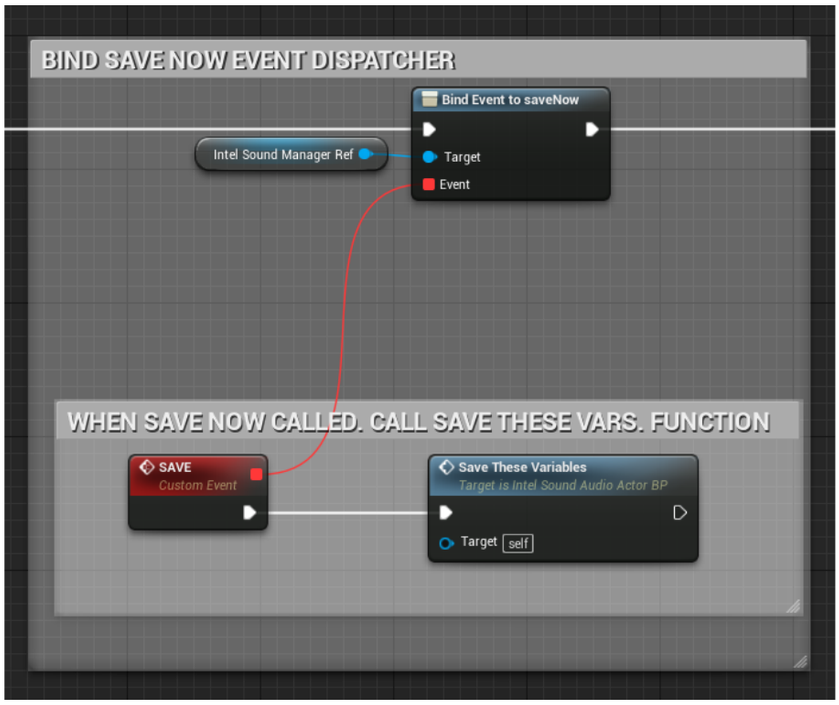
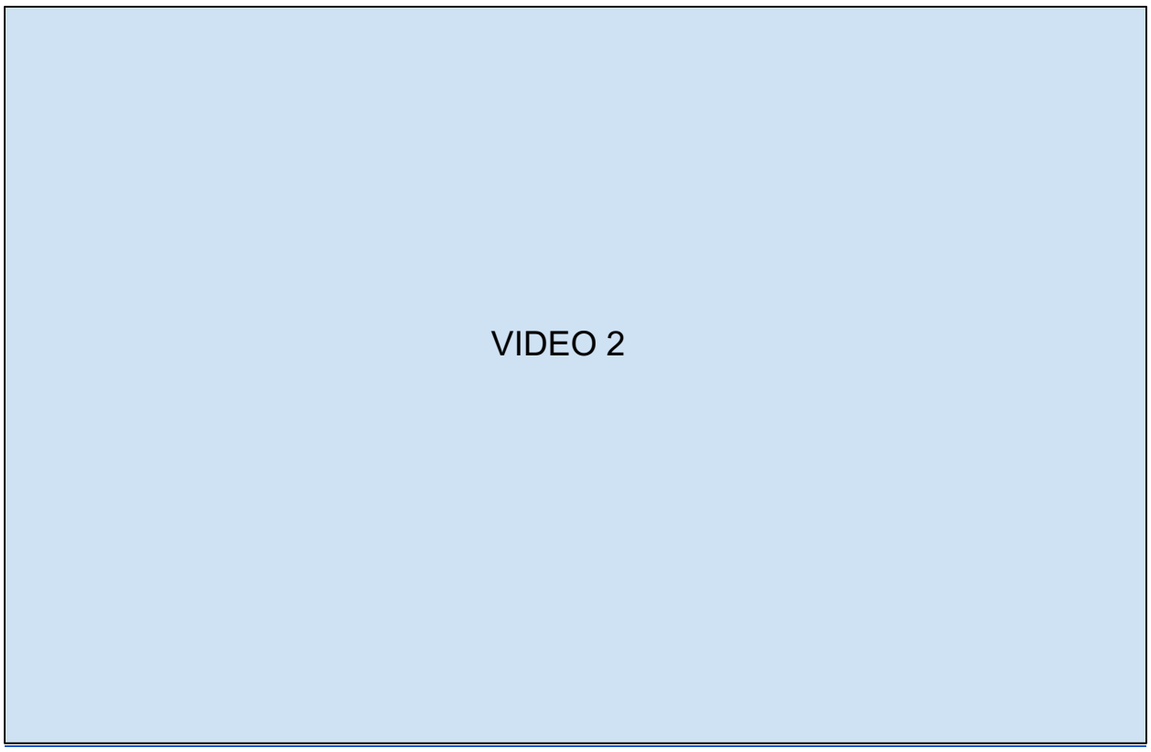


Image (IntelSoundAudioActorBP)



Now if everything worked correctly, we should be able to edit any sounds in the folder inside VR.



Notices

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

The products and services described may contain defects or errors known as errata which may cause deviations from published specifications. Current characterized errata are available on request.

Copies of documents which have an order number and are referenced in this document may be obtained by calling 1-800-548-4725 or by visiting www.intel.com/design/literature.htm.

This sample source code is released under the Intel Sample Source Code License Agreement.

Intel and the Intel logo are trademarks of Intel Corporation in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others.