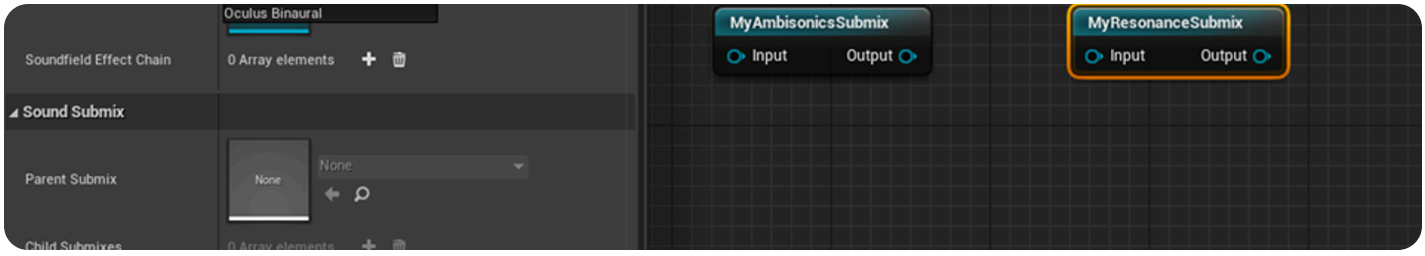
Native Soundfield Ambisonics Rendering

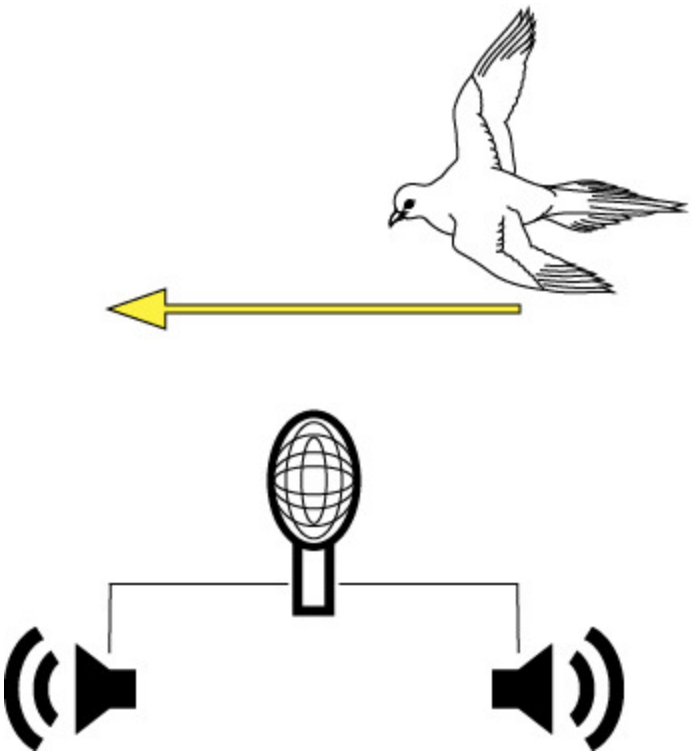
An overview on how soundfield ambisonics is used in the Unreal Audio Engine.



A **soundfield** is an audio representation of a spatial area, real or virtual. One of the most challenging problems in both real-world and virtual audio is how to best capture and represent a soundfield.

The Challenge of Capturing and Representing Soundfields

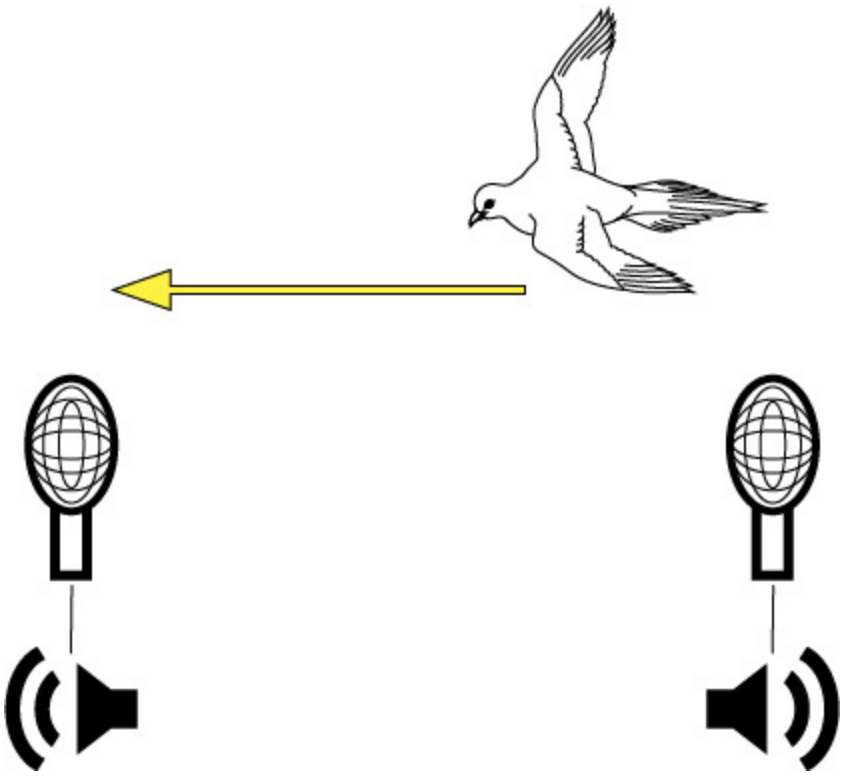
If you have ever taken microphones to a beach, a forest, or any other locale that you wanted to capture as a soundscape, you already have hands-on experience with the difficulties you might encounter. For example, say that you bring a single cardioid microphone into the forest and you are able to get a great audio recording of a bird cawing as it flies from your right side to your left. Assume that your microphone is equally sensitive in all directions, and that the bird is continually cawing at the same volume. You can play back that recording at equal volume in the left and right channels of your home speaker setup:



The caw of the bird flying past the microphone will natively have the same volume in both speakers.

Because of the limitations of this recording setup, you did not record the bird in a way that would give the listener an idea of where the bird is in relation to that listener, or in which direction the bird is flying. The recording will sound louder in both speakers as the bird gets closer to the microphone, and quieter in both as the bird flies away.

If you were able to record this bird with two microphones spaced some distance apart, one for each speaker you play the sound back on, you would have vastly more information:



With two microphones, the caw of the bird will be louder in the first mic and softer in the second, then louder in the second as the sound grows softer in the first.

As the bird moves from right to left, it becomes louder in the left microphone and quieter in the right microphone.

By using two microphones to record a stereo audio asset, we've already captured important spatial information from the soundfield into our audio stream: how far to the left or right of the microphones the bird currently is. If we added more microphones, we could capture more spatial information about the bird's location and motion in space, including if the bird flies behind our microphone array instead of in front of it, and even how high the bird is above our array of microphones.

Downmixing is the process of taking an audio source and rendering it as a number of speaker outputs. If you're familiar with the way that downmixing occurs in games, you might notice that it is analogous to the microphone array described in the previous diagram.

Typically, an audio source is played in each speaker, and the loudness is directly proportional to how close the source is to where that speaker would be in the game world relative to the other speakers.

A typical panning system is analogous to placing an omnidirectional microphone where a speaker might be in the game world. When we downmix our game to stereo, we are preserving information about how far to the left or right of the player an audio source is.

Downmixing audio to 5.1 or 7.1 gives additional information about whether the source is in front of or behind the player, based on how loud the source is in the surround-left and surround-right channels compared to the front-left and front-right channels.

For many applications, this may be enough information for the player to be able to localize an audio source. However, for many use cases, it would be ideal to also have elevational information. There are many ways of retaining the 3D position of an audio source when **encoding** (capturing) a representation of a soundfield that allow easy **decoding** (playback) on any speaker configuration needed. These methods provide a rich spatial audio experience.

One of the most popular soundfield representations for retaining both elevational and azimuthal information about audio when downmixing sources is called **ambisonics**.

Ambisonics in Unreal Engine

Ambisonics is a specific soundfield representation that projects positional audio onto a sphere using **spherical harmonics**. You may recognize spherical harmonics from their use in lighting for video games — ambisonics is the equivalent for sound. The higher the **order** of ambisonics we use, the higher the **spatial resolution** of our ambisonic audio stream will be, resulting in easier-to-localize audio when we decode the ambisonic audio stream to speaker positions.

There are various microphone arrays available that even allow you to record directly to an ambisonic audio file. Because you can rotate ambisonic-encoded audio as you see fit, ambisonics is a great choice for **ambient audio beds** (background sounds). As Unreal Engine plays back ambisonic audio sources, we can dynamically rotate the ambisonic audio bed as the player looks around, giving the impression that the ambient audio is truly coming from fixed directions in the game world.

Importing and Playing an Ambisonic Audio Asset

Unreal Engine only supports **first-order ambisonic assets**.

A first order ambisonics asset has four channels — the **omni component**, or **W channel**, and the **vector components**, or **X**, **Y**, and **Z channels**.

Unreal Engine supports two channel orders: **FuMa (Furse-Malham)**, and **ACN (Ambisonics Channel Number)**.

FuMa channel ordering sequences the ambisonics audio channel components as W,Y,Z,X.

ACN channel ordering uses W,X,Y,Z.

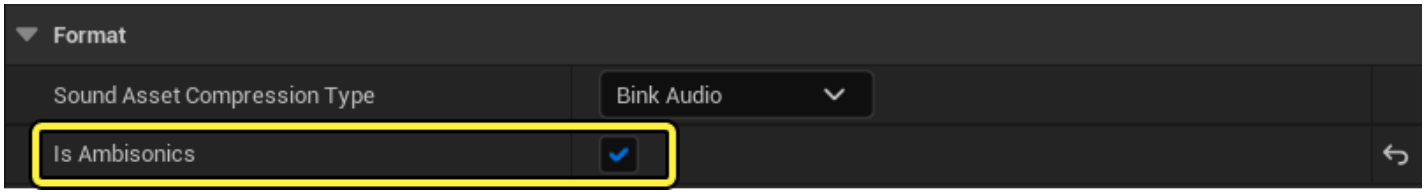
Since first-order ambisonic audio uses four channels, any imported file that does not have four channels will be treated as a normal, non-ambisonic audio asset.

To import an ambisonic `.wav` file into the Content Browser, you should end the name of the asset with either:

* `_fuma.wav` If your ambisonics file uses **FuMa** channel ordering.

* `_ambix.wav` If your ambisonics file uses **ACN** channel ordering.

This will tell the engine what the channel order of the original ambisonic asset is. After dragging your file into the Content Browser, when you open the Sound Wave asset, you should see that the **Is Ambisonics** checkbox is already checked in the Sound Wave inspector panel:

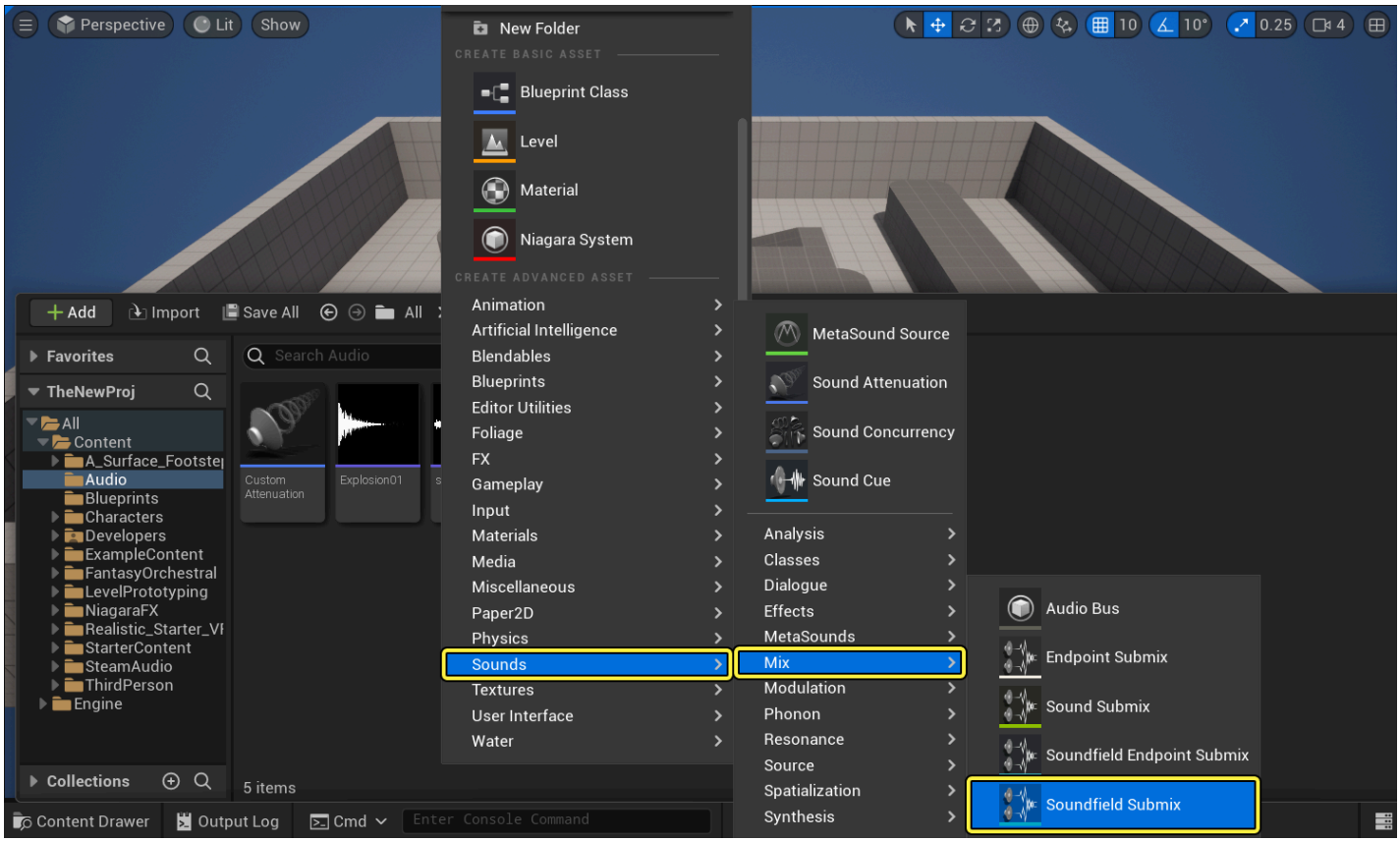


If you forget to end the name of your file with `_ambix` or `_fuma` before you import the `.wav` file, you can still check this box yourself. The engine will assume that the original file you imported used **ACN** channel ordering if you do this.

After your ambisonic file is imported, you can treat it like any other Sound Wave in the engine, with the important exception that **rotation matters**. When you play back an ambisonic Sound Wave via an audio component in the world, the engine will rotate the ambisonic bed based on the rotation between the current rotation of the audio component that is playing and the current rotation of the player in the game world.

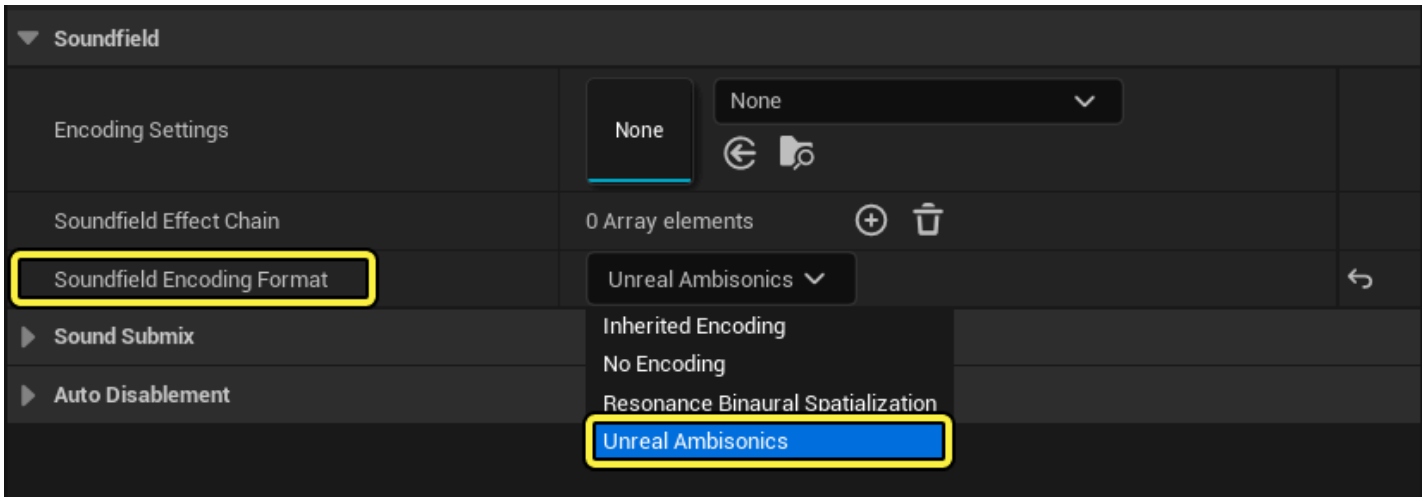
Soundfield Submixes

In addition to playing back ambisonic Sound Waves like you would any other Sound Wave, you can use ambisonics directly in the submix graph. To do this, you will need to create a **Soundfield Submix**:



Soundfield Submixes can be used and connected to other submixes in the submix graph, but are able to use any soundfield representation currently implemented in the engine or implemented by a currently enabled plugin.

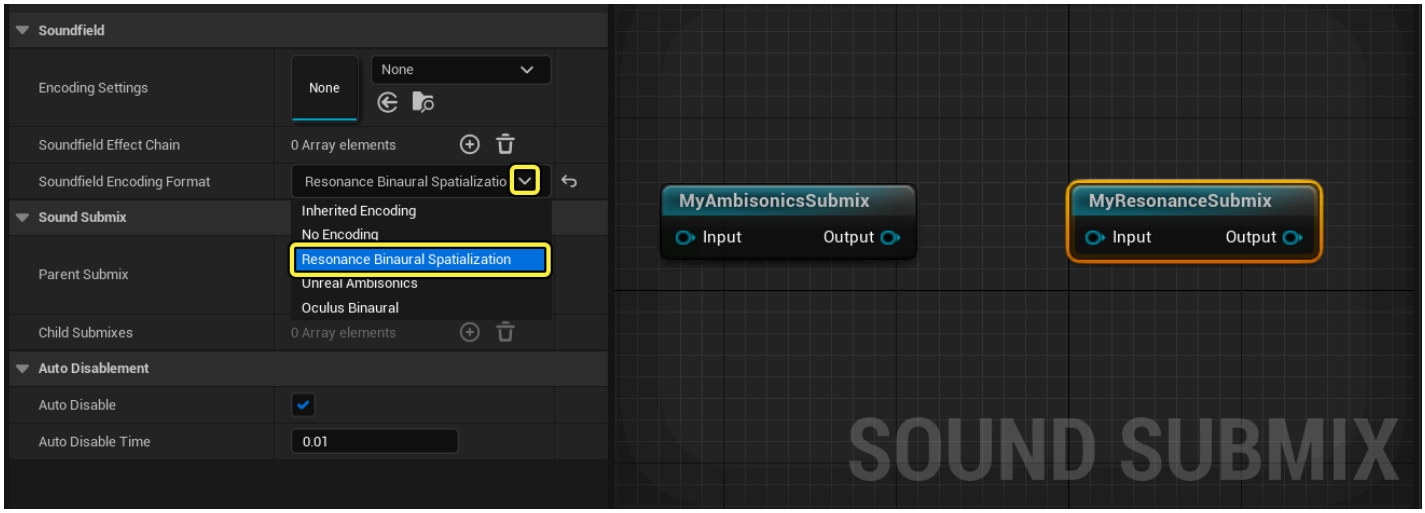
If you open your newly created Soundfield Submix and look at the inspector panel, you will see a dropdown menu for **Soundfield Encoding Format** that includes **Unreal Ambisonics** as one option:



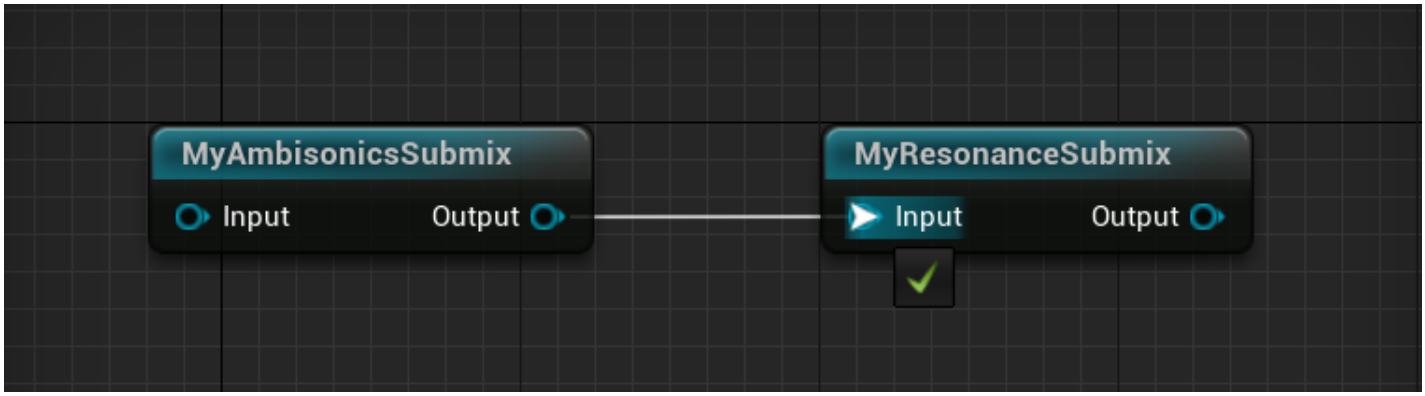
After you select **Unreal Ambisonics**, you can send any audio source to this submix — whether or not it is an ambisonic source — and it will be encoded or mixed down to a single ambisonic stream. If you are sending an ambisonic source that has some rotation on it to an ambisonic stream, don't worry — the engine will rotate the ambisonic source audio based on the source rotation before it is mixed down.

If you have the **Google Resonance** or **Oculus Audio** plugins enabled, you may see some other entries in the **Soundfield Encoding Format** menu. These allow you to send both non-ambisonic and ambisonic sources, as well as other submixes, directly to Resonance or Oculus in the submix graph.

Let's say that you want to take the output of your ambisonic submix and render it to binaural audio with Google Resonance. To do this, create another soundfield submix and drop it onto the submix graph, then select **Resonance Binaural Spatialization** as its **Soundfield Encoding Format**:

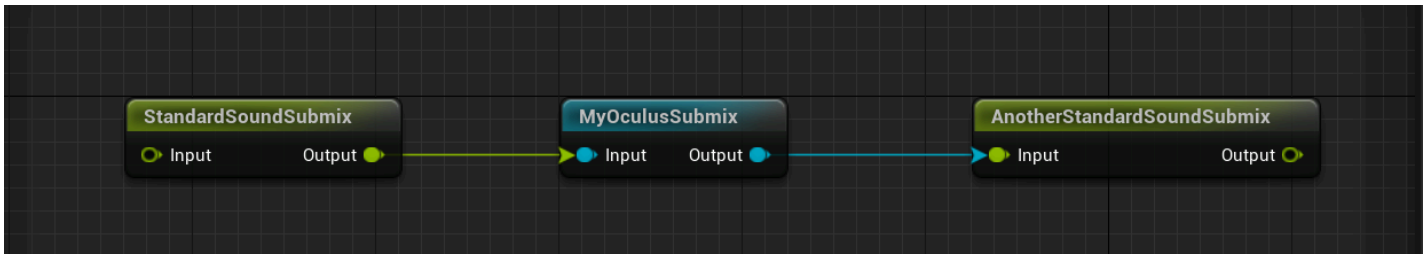


Now all you need to do is connect the output of your **Ambisonics Submix** to your **Resonance Submix**.



You can also send both ambisonic and non-ambisonic audio sources to your Resonance Submix and they will be rendered to a binaural stereo output. You can even send the output

of non-soundfield submixes to soundfield submixes, and they will automatically be encoded to whatever the format of that submix is. In the following graph, we take the output of a normal, downmixed submix and mix it in with our Oculus Audio binaural audio stream before sending the output of that to a different, non-soundfield submix:

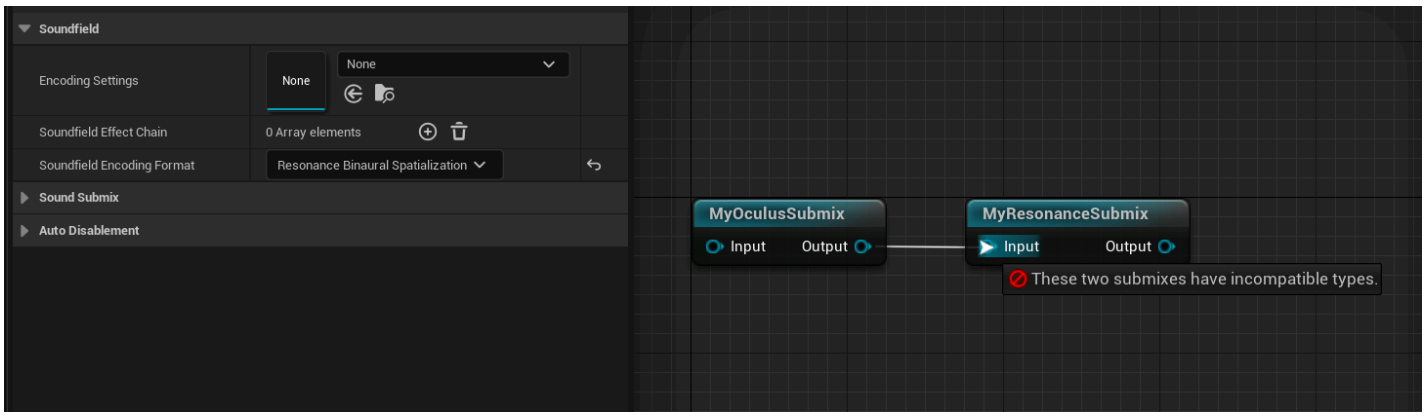


It is important to remember that once sources are sent to a submix, they are downmixed to the format of that submix.

For example, any spatial information that would have been received from sending an audio source to MyOculusSubmix will be lost if its output is sent solely to the StandardSoundSubmix instead.

Soundfield Encoding Compatibility

Not all soundfield encoding formats are compatible. For example, you cannot connect a submix that uses the Oculus Audio soundfield format to a Google Resonance soundfield submix, or vice versa:



Click image for full size.

However, you can have sources send their output to either of these soundfield submixes in your project as needed.

Conclusion

Both native ambisonic rendering and soundfield submixes are a major step forward for developers to control and extend every aspect of the spatial audio pipeline in Unreal Engine. In the future, more soundfield formats will be added to this system. Meanwhile, if you are curious about writing your own soundfield format, check out ` ISoundfieldFormat.h `in the **Audio Extensions** module.