Building Augmented Reality Spatial Audio Compositions for iOS

A Guide for Use of AR Positional Tracking in iOS 11 and Beyond

v 1.2 (Updated 23 April 2018)

Introduction and Terms

This document outlines the procedures to develop and release immersive spatial audio applications for iOS using AR positional tracking. Guidance to build similar spatial audio applications using GPS positional tracking, and GPS-based audio positioning, on iOS and Android can be found in a separate, similar document on the tcwav.com website.

- AR Positional Tracking: Used in this guide to refer to the practice of tracking a user's position and orientation in an environment at a scale granular and accurate enough to simulate real-world sound source positioning. Apple uses an approach called visual inertial odometry to do this. In plain language, visual inertial odometry means using a combination of the camera and sensors in your iOS device to accomplish positional tracking.
  - Google Tango is another method to accomplish positional tracking. As of writing, Tango only works on two specific Tango-enabled phones. Some of the practices in this guide could similarly be applied to Google Tango.

Spatial Audio Positioning

There's a ton of research online about digital audio spatialization, and several software implementations that integrate nicely with Unity—RealSpace3D Audio, Oculus Spatializer, Google VR (now Resonance Audio), and more. In this guide, we will use the Oculus Spatializer—now built into Unity. It is effective, highly customizable, reliable, and easy to implement. Throughout the guide, we will also provide instruction for Google VR because it is free (at least for non-commercial applications), well documented, and enables a unique capability: Audio Rooms, which are detailed later in this guide. You can learn about Google's technology here. (*Note: Google VR guidance still applicable, but as the newest iteration isn't working on iOS11 for us, we recommend you skip it).

The Google VR audio technology does not include effective reverb or distortion options, and it does not allow for easily customized, distance-based rolloff adjustments (reverb levels, volume, etc.) For these reasons, we use the default Unity spatialization in our work.

We'd encourage anyone to do additional audio research for advanced spatial implementation and understanding. One topic to start with is binaural processing and recording—to drastically simplify, a method of recording and playing audio that simulates our ears. Another is multi-mic recording and spatialization—a good example was demonstrated by Shaun Crook on the 4DSOUND system in Amsterdam in 2014.

Shaun used 16 microphones spaced throughout a real room to record footsteps walking and ping pong balls bouncing, then later mapped each microphone's recording to a similar position in space for playback. You could hear the balls and steps moving as if they were there.

There are many places to look for inspiration and to credit here—Google, for building nice Google VR documentation; partners at the 4DSOUND Hack Lab in 2014 (particularly Peter Kirn and the 4DSOUND Team) for some of the compositional concepts; and the group BLUEBRAIN for releasing previous spatial audio applications using GPS.

Now, to building:

Software and Costs

Please note this guide was written from a PC development standpoint. It should be easy to follow for Mac as well, but we'd welcome any inputs to adjust for Mac users.

You will need several pieces of software and hardware to develop and publish functioning AR application or iOS. Additionally, there are fees to publish to the Apple App Store. Depending on the scale and intent (i.e. monetary) of your application, you should review all the Terms of Service to see if you should sign up for premium plans with some of these providers.

Overview:

- Unity Personal (Free) – this will be our primary environment for app creation.
    - You will need a version of Unity that supports Apple's ARkit. As of writing, the most up-to-date is the 2017.2.0b beta.
- Google VR SDK for Unity (Free/Optional) – audio spatializer for Google VR
- To execute the build for iOS: a Mac (or an understanding friend with one).
    - Apple unfortunately limits iOS development to macOS, but you can develop on and export from PC and then borrow a Mac for a few hours. (You'll need about 5GB of space free, and a handy iOS device.)
    - Xcode for macOS (Free)
        - You'll need to ensure you have the latest version of Xcode that supports Apple's ARkit. As of writing, this is Xcode 9 beta 5.
    - To publish to app store, an Apple Developer account ($99/year to publish apps)

Downloads and Initialization:

1. Download and install Unity from here. You only need "Unity Personal" for non-commercial purposes.

    a. Be absolutely sure during installation that you select the iOS SDK as part of the install.
2. If you choose to use Google VR, download the [Google VR SDK](#) for Unity.
3. Download the Unity ARKit plugin through the Unity asset store in the app.
4. To get an idea of how the app will look on your phone without having to build the app each time, grab “Unity Remote” for your phone. It’s limited but it’s free. Unity is currently working on adapting Unity Remote to serve as a better AR input.

Beginning your Project

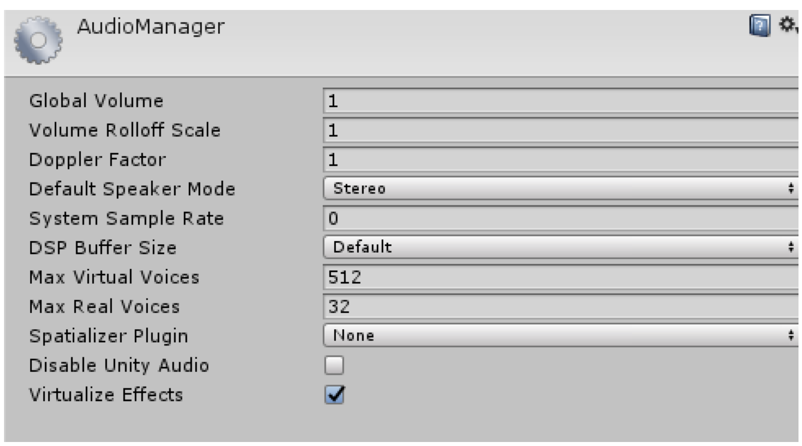
Open the ARkit demo scene “UnityAROcclusion.” Recommend you re-title the scene, and title and save your project. Please note that, as we go, you will want to frequently save both your project and your scene.

Applying the Default Unity Audio Spatialization

If you are going to use Google VR for audio, skip this section.

Go to Edit->Project Settings->Audio and ensure “Virtualize Effects” is checked.

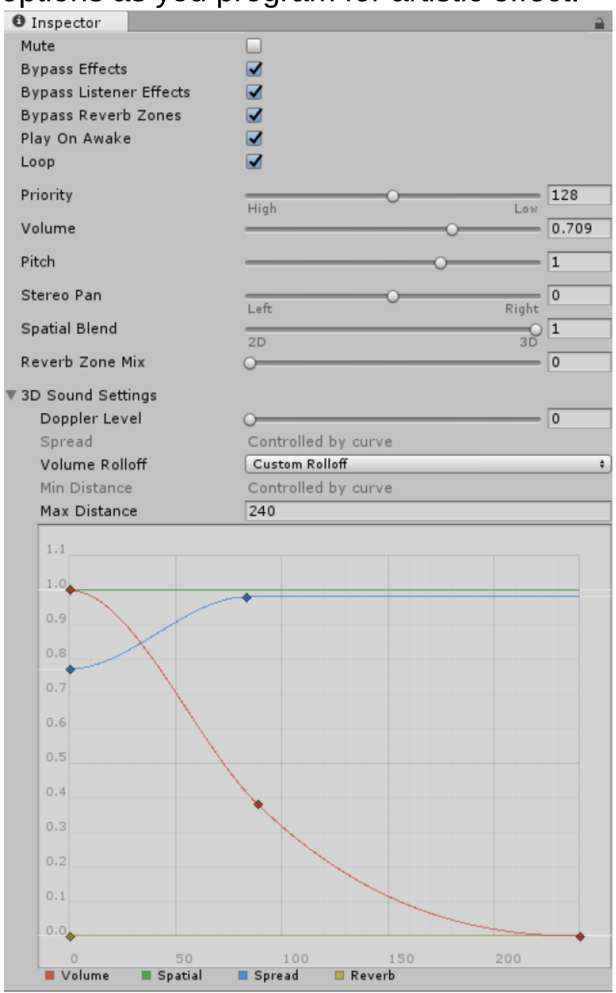
Also, where you see “Spatializer Plugin->None” (as below), select “Oculus Spatializer.”



We will now add our first audio object to our scene.

- GameObject->3D Object->Sphere.
- Add the “Audio Source” component in the Sphere’s inspector:
- Import an audio file into your Unity project assets folder, then select it in “Audio Source” by hitting the small circle. You can ignore the “Output” field. You now have a sound object.

- It is a good practice to (by default) set your Audio Source rolloff to Linear versus Logarithmic, turn off Doppler (if you don't like the sound of it, like us), and set the scale (size) of the sphere to a number two times the size of the max distance. Then you will be able to visually see the audio distance as you program.
    - Note: an audio source distance of 10 would correspond roughly to a sphere size of 20.
    - You can adjust rolloff patterns intuitively using the 3D Sound settings by setting the rolloff to "Custom Rolloff" or by clicking the rolloff graph to modify it (double-click to add points). Always ensure the tail end of your line or curve hits 0 before it reaches the right end of the graph--otherwise the object's volume will never decrease to 0, no matter the distance.
    - If using Oculus Spatializer, you will see a button for "Spatialize." Click it.
- We also suggest you bypass effects and zones by default, then selectively reenable these options as you program for artistic effect.



An example "Audio Source" with custom rolloffs.

- Add a color to the sphere for identification. Go to Assets-Create->Material. Change the color and drop it onto your sphere you just created in the Hierarchy pane. You may create many spheres, we strongly suggest using multiple colors to keep track.

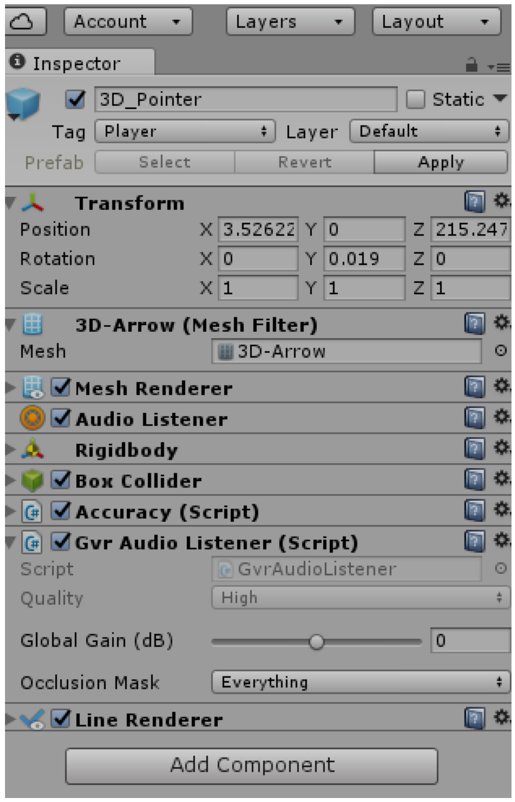
Using Google VR Audio Spatialization

If you are going to use Unity's default audio spatialization, skip this section.

To use GoogleVR, import it into your project by going to Assets->Import Package->Custom Package and finding the downloaded package. Wait a moment and hit "Import."

- Hit "Play" to compile a build and see if the import worked correctly. Ours did not work right away—we received compiler errors. We had to find and run the Backwards Compatibility script here.
- We still had errors and had to delete two folders: the GVR Video folder and the Demos folder. This is a common issue with Google VR 1.10. Hit "Play" again and it should work.
- Finally, set the application to use the Google Spatializer. Go to Edit->Project Settings->Audio and set the Spatializer plugin to GVR Audio Spatializer.

Now we can add the Google VR plugins. First, go to the object "Main Camera" under "Camera Parent." This represents the user's location, so it will be where we want our "listener" to be. In the inspector, hit "Add Component" and add GVR Audio Listener. Leave the standard Audio Listener on.

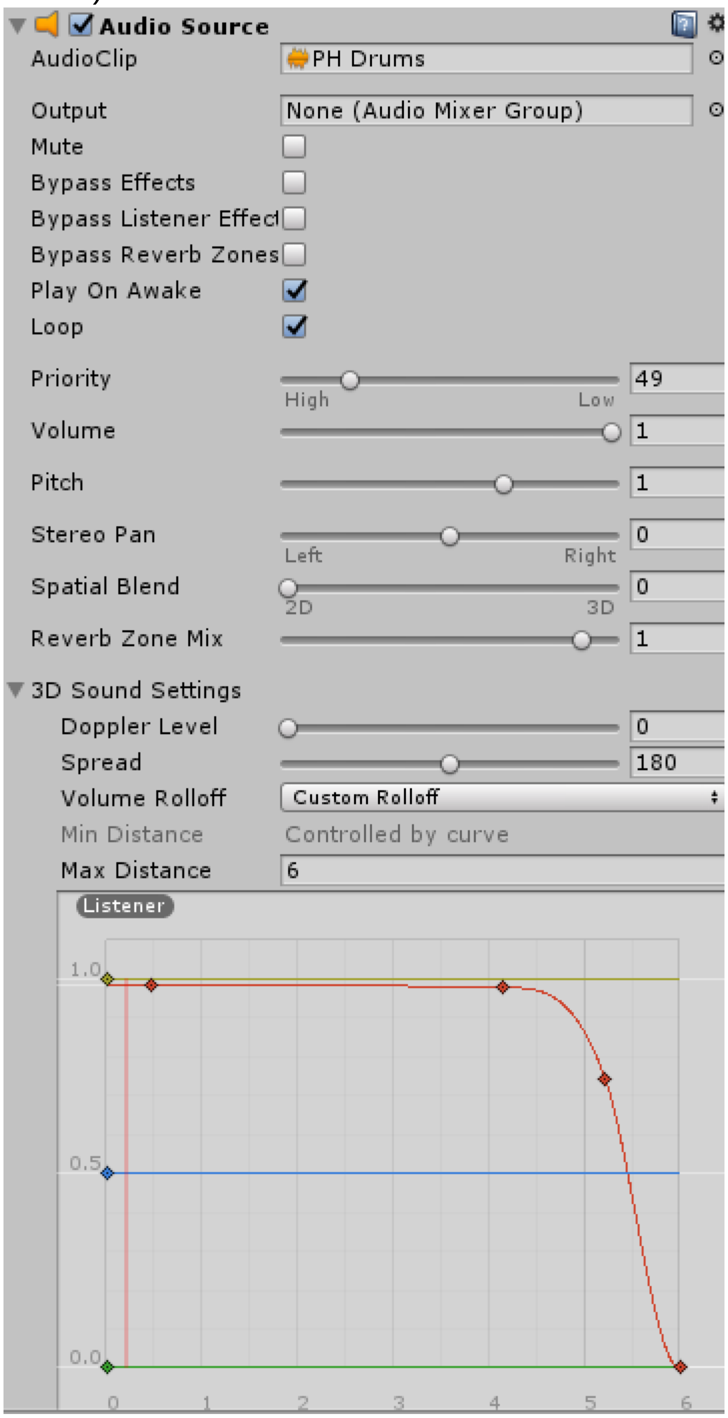


We will now add our first audio object to the scene.

- GameObject->3D Object->Sphere.
- Add “GVR Audio Source” as a component in the Sphere’s inspector. Uncheck the standard “Audio Source” if it appears.
- Import an audio file into your Unity project assets folder, then select it in “GVR Audio Source” by hitting the small circle. You now have a sound object.
    - It is a good practice to (by default) set your Audio Source rolloff to Linear versus Logarithmic, turn off Doppler (if you don’t like the sound of it, like us), and set the scale (size) of the sphere to a number two times the size of the max distance. Then you will be able to visually see the audio distance as you program.
        - Note: an audio source distance of 10 would correspond roughly to a sphere size of 20.
- Add a color to the sphere for identification. Go to Assets-Create->Material. Change the color and drop it onto your sphere you just created in the Hierarchy pane. You may create many spheres, we strongly suggest using multiple colors to keep track.

Quick Notes on Audio Sources

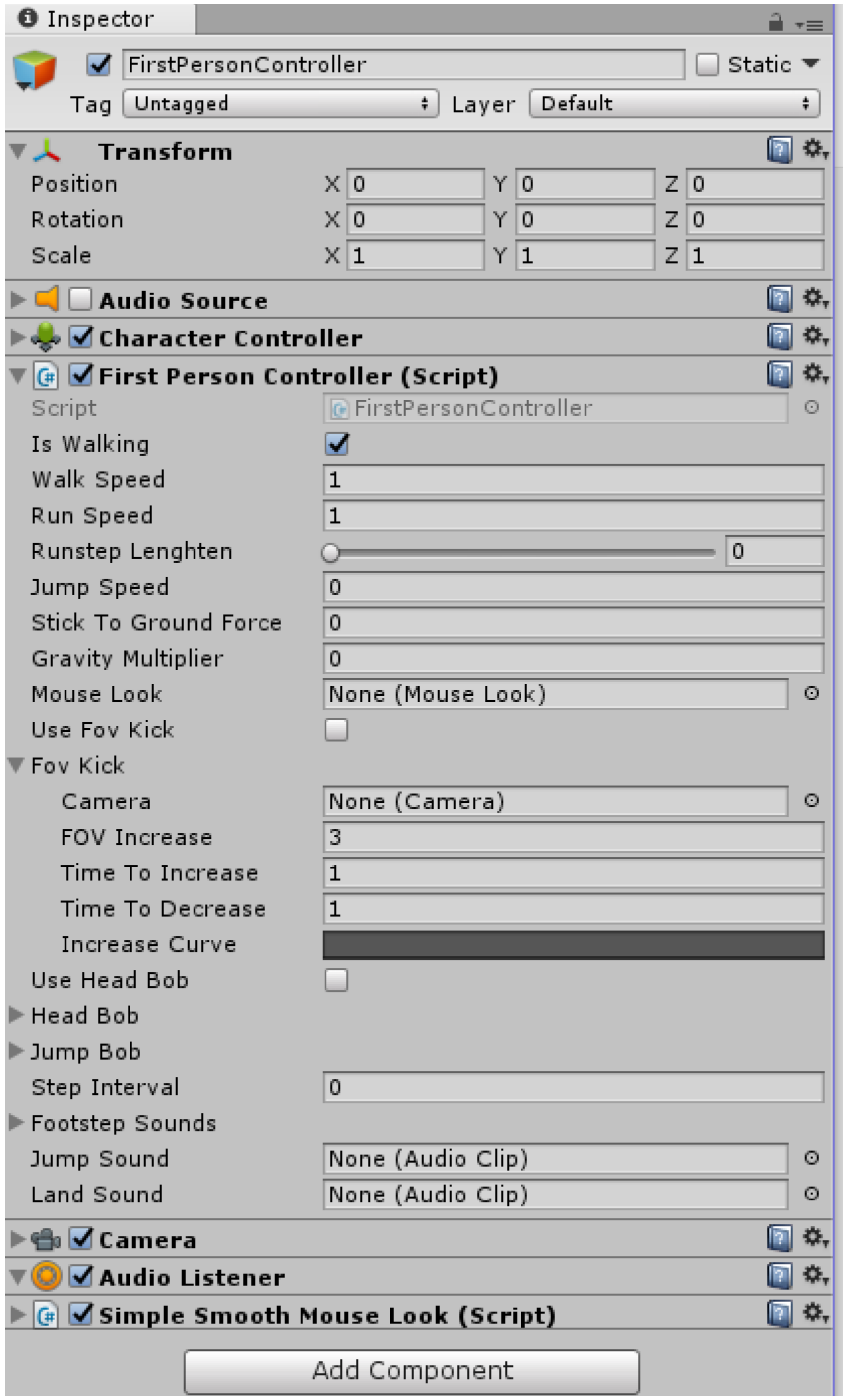
- A 2D spatial blend will eliminate much of the 3D sound positioning of the source
- A "180" spread (as pictured below) will make your sound sound less like it's coming from a center point source and more spread over the circumference of the sphere. This works very well for sound sources you'd like to remain equal in both ears no matter the orientation of the listener (like, perhaps, bass and drums).



Input for Programming

Particularly on PC, developing and testing AR applications on iOS can be a cumbersome process. We recommend you instead add a first-person controller, third-person controller, and additional cameras to the scene. In this way, you can simulate walking through the environment on your computer from multiple vantage points.

- An excellent First-Person Controller can be found in the Asset Store. Navigate there and download “Standard Assets” by Unity. After installation, find the First Person Controller and add it to your scene. You can also add a mouse look script, as we have done here, and an Audio Listener.



- Remember that this controller will conflict with your AR controller. Make a note to yourself:
  - To use the First-Person Controller, ensure this object is ON and “Camera Parent” and “AR Camera Manager” are OFF.
  - For building to AR, turn OFF the First-Person Controller and turn ON “Camera Parent” and “AR Camera Manager.”

- Some Unity betas turn “Mobile Input” on by default. This option seems to override the ability to use arrow keys to navigate the space. You can turn it off from the top menu.

Your Spheres in Space

You now have spheres in an environment you can navigate, and you also know how to create sound objects. Now we will discuss positioning those sound objects spatially.

Positioning

- All you need to do is set the X, Y, and Z coordinates in the sphere’s “Transform” component. Just adjust and experiment. Please note that sometimes your created spheres will inherit default values of approximately -625. We are unsure why this happens, but it’s an easy fix—just reset them to 0.

Movement

You can use Unity Animations or use custom scripts to control your object movement.

There are tons of tutorials on the Internet, and you can use any knowledge of C# or Java to write your own. To get started, here is code for a new script that will oscillate a sphere back and forth between two points. Just select your sphere in Unity and go to Add Component->New Script->Create and Add (C#). Call it “TCWMoveOscillate” (or whatever you want, but then adjust the name in the script code later accordingly). Right click the script and select “Edit Script.” Now, populate it with this code and hit “Save.” Hit play in Unity and you will see a sphere begin to move. We have created several movement scripts and are happy to share—just reach out to us.

```
using UnityEngine;
using System.Collections;
public class TCWMoveOscillate : MonoBehaviour
{
	public Vector3 pointB;

	IEnumerator Start()
	{
		var pointA = transform.position;
		while (true) {
			yield return StartCoroutine(MoveObject(transform, pointA, pointB,
3.0f));
			yield return StartCoroutine(MoveObject(transform, pointB, pointA,
3.0f));
		}
	}
```

```
	IEnumerator MoveObject (Transform thisTransform, Vector3 startPos, Vector3 endPos, 
float time)
	{
		var i = 0.0f;
		var rate = 1.0f / time;
		while (i < 1.0f) {
			i += Time.deltaTime * rate;
			thisTransform.position = Vector3.Lerp (startPos, endPos, i);
			yield return null;
		}
	}
}
```

Other Movement Patterns

You will find videos and descriptions of some movement patterns on our Concepts page, http://www.tcwav.com/concepts.html

Spatially Generative Music

One specific type of movement that really excites us, and needs more explanation than the concept video on tcwav.com, is spatially generative music. This randomizes the location of spheres within a set range. Imagine each sphere as a note or quick melody, and they pop up at random places around the listener. The listener will never hear the same sequence and spaces twice.

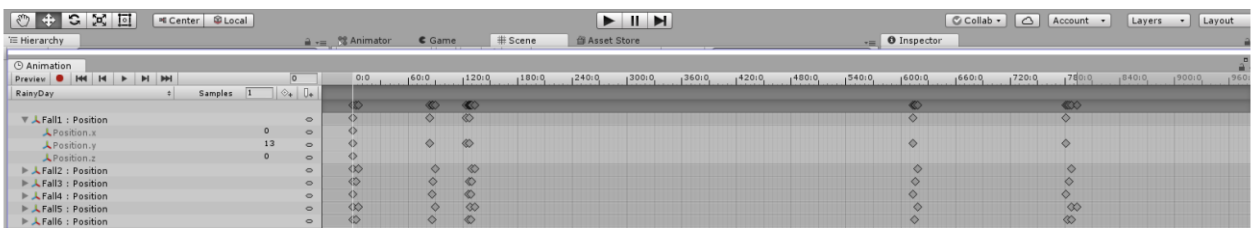
This code will randomize the position of a sphere across the X and Z planes. Adjust and/or it to multiple spheres to create spatially generative music.

```
using UnityEngine;
using System.Collections;

public class Random : MonoBehaviour
{

    private Vector3 _centre;
    public Vector3 Velocity = new Vector3(0, 0, 0);
    float x;
    float y;
    float z;
    public float speed = 5f;

    // Use this for initialization
    void Start()
    {
        _centre = transform.position;
        InvokeRepeating("LaunchProjectile", 2.0f, 2f);
```

```
    }

    // Update is called once per frame
    void Update()
    {
    }

    private void LaunchProjectile()
    {
        _centre += Velocity * Time.deltaTime;


        var offset = new Vector3(UnityEngine.Random.Range(-2, 2), y,
UnityEngine.Random.Range(-2, 2));

        transform.localPosition = _centre + offset;

    }
}
```

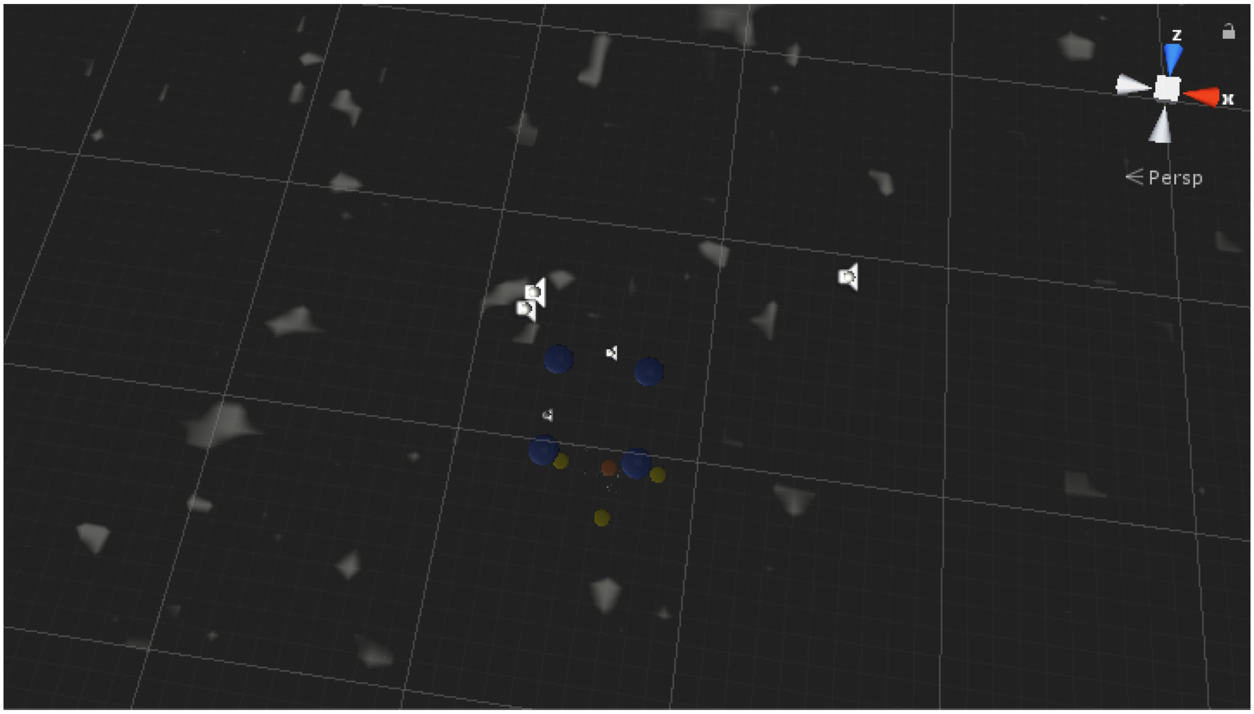
Rain

The rain functionality is detailed in a concept video. In short, we use Unity's animator to control the Y coordinates of grouped objects. In this way, they "drop" through the ground.



Unity's Animator window

Naturally, when rain hits the ground, it should stop. Fortunately, there are ways to alter the audio sounds when they "hit" the "ground"—so that either they quickly fade, distort, or anything else you'd want.



Raindrops (white) collide with our "ground," a giant rectangle.

Add a distortion filter and a lowpass filter to your raindrops, turn them off then add this script, "TCWCollision." This script will turn on the distortion filter and lowpass filter as soon as a collision is detected. Ensure you have a "Box Collider" component on your giant rectangle, with "Is Trigger" checked, and a "Sphere Collider" on each raindrop (you don't need "Is Trigger" checked here). This will ensure Unity is detecting the collisions for these objects.

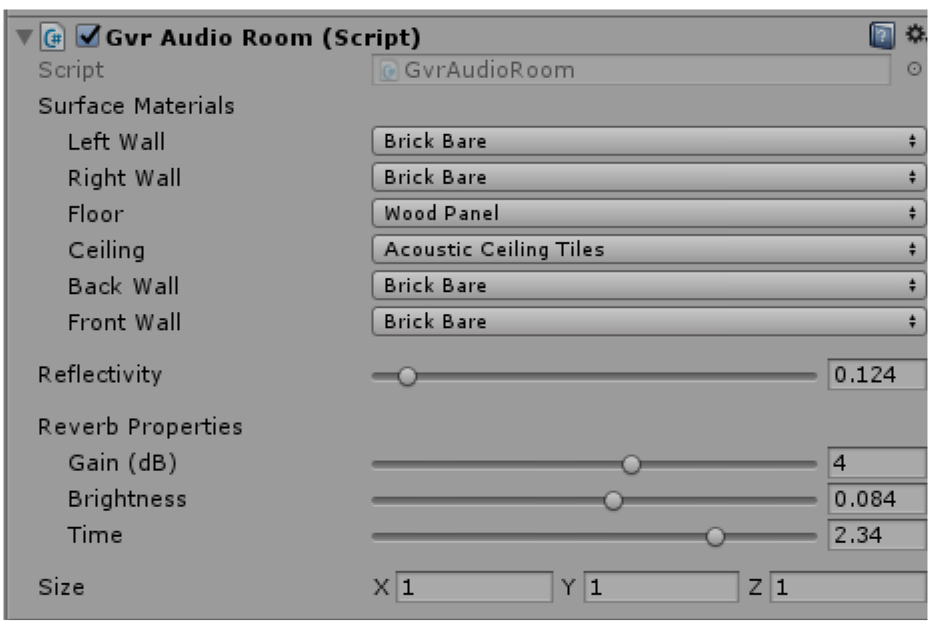
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class TCWCollision : MonoBehaviour {

    // Use this for initialization
    private AudioDistortionFilter myLight;
    private AudioLowPassFilter myLight2;


    void Start()
    {
        myLight = GetComponent<AudioDistortionFilter>();
        myLight2 = GetComponent<AudioLowPassFilter>();
    }


    void OnTriggerEnter(Collider other)
    {
```

```
        myLight.enabled = !myLight.enabled;
        myLight2.enabled = !myLight2.enabled;


    }

}
```

Audio Rooms

The Google VR Audio SDK also includes “GVR Audio Rooms,” which virtually simulate rooms of different materials (wood panels, concrete, etc.) We found the reflectivity to be a bit much generally, and the implementation not perfect—but these rooms can certainly be implemented to interesting effect.

One practical use we implemented was to use the rooms to create a subtle reverb tail.



Here, you will see the reflectivity is down, but the time and gain on the reverb are fairly high.

This subtle reverb is great at masking sharp transitions or melding various sounds together. Here, we placed a room (purple) over the entire spatial range of our piece. This room can be made by creating a cube instead of a sphere and following the positioning methods described earlier. The spatially generative sounds, as described above, originally sounded too jarring when they suddenly moved mid-note. The reverb on this room smoothed the transitions out nicely.

- Ensure your other sounds all have “Bypass Room Effects” checked to avoid unwanted reverb. Only leave Room Effects enabled on the objects where you want the extra reverb.

A large “room” of reverb

<u>Reverb Zones</u>

By default, Unity includes Chorus, Distortion, Reverb, EQ filters, and more. These are all fairly self-explanatory (note: they do not directly work with Google VR spatialization). It also includes “Reverb Zones,” which can be very effective for your compositions. Similar to the Google VR Audio Rooms described above, reverb zones are great at masking sharp transitions or melding various sounds together. Here, use the same rectangle we used for the GVR Audio Room example. This room can be made by creating a cube instead of a sphere and following the positioning methods described earlier. The spatially generative sounds, as described above, originally sounded too jarring when they suddenly moved mid-note. The reverb zone on this rectangle smoothed the transitions out nicely.

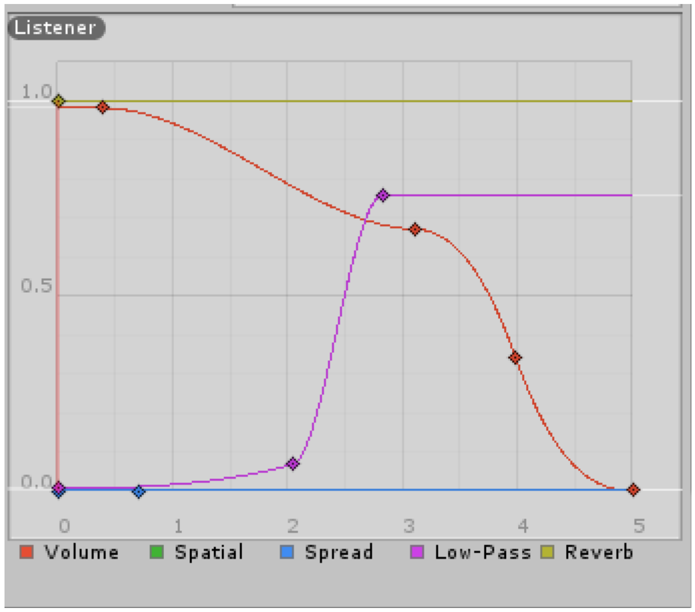
- Ensure your other sounds all have “Bypass Reverb Zones” checked to avoid unwanted reverb. Only leave Reverb Zones enabled on the objects where you want the extra reverb.

- Make sure you set the Min and Max distances appropriately for your objects. This is easily done by picking an extreme reverb, like “Arena,” and adjusting the settings during runtime through quick trial and error.

A reverb zone

Controlling All Audio Effects with Distance

Elsewhere in this guide, we’ve discussed how to adjust curves on audio sources to impact the amount of volume, spatial blend, spread, or reverb based on the distance between the object’s center and the player. Unity allows these same curves to control its low-pass filter frequency if you add the low-pass effect.



However, if you want to control things like echo or distortion, or want to use entirely different object's distances to control audio, you need a different approach. We'll walk through an example. This is a bit more complex than above but opens up a whole world of creative possibilities.

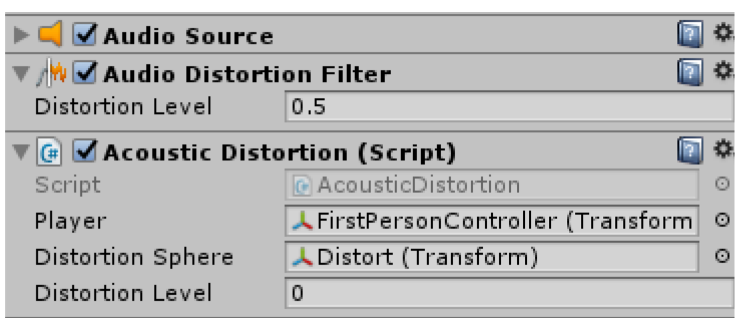
Create a sphere called "Distort" and place it at X=6, Y=0.2, Z=0. Create another sphere with a sound of Max Distance=5 and place it at X=0, Y=0, Z=0. We'll call this "Acoustic" for the acoustic guitar we used in our composition. We are going to modify the sound of "Acoustic" based on the distance between our player and the "Distort" sphere.

- Add the Audio Distortion Filter to "Acoustic."
- Create a new script called "AcousticDistortion" and add it to "Acoustic"
- Insert this code:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class AcousticDistortion : MonoBehaviour
{


    public Transform Player;
    public Transform DistortionSphere;
    public float distortionLevel;

    float distanceBetweenThem;
    // Use this for initialization
    void Start()
    {
        AudioDistortionFilter bob = gameObject.GetComponent<AudioDistortionFilter>();
```

```
    }

    void Update()
    {
        distanceBetweenThem = Vector3.Distance(Player.position,
DistortionSphere.position);


        if (Vector3.Distance(Player.position, DistortionSphere.position) < 6.2)
        {
            distortionLevel = 0;
            gameObject.GetComponent<AudioDistortionFilter>().distortionLevel = 0;

        }

        if (Vector3.Distance(Player.position, DistortionSphere.position) > 7.82)
        {
            distortionLevel = 0.79f;
            gameObject.GetComponent<AudioDistortionFilter>().distortionLevel = 0.79f;

        }


        //D
        else
        {
            gameObject.GetComponent<AudioDistortionFilter>().distortionLevel =
distanceBetweenThem / 2 - 3.055f;
            distortionLevel = (distanceBetweenThem / 2 - 3.055f);
        }

        Debug.Log("Distance between obj1 and obj2 is " + distanceBetweenThem);
    }
}
```

- In the inspector, ensure you assign “Distortion Sphere” to our object “Distort” and “Player” to our “FirstPersonController.” (Important: remember later to change this from “FirstPersonController” to our AR “Main Camera” for your final release or for testing.)



- You will see that this code calculates the distance between object “Player” and object “Distortion Sphere,” divides them by 2, and subtracts 3.055. This math is based on the distance we set between “Distortion” and “Acoustic” earlier.

    - The code also determines if the player is relatively close to the DistortionSphere and sets distortion to 0.
    - The code also determines if the player is relatively far from the DistortionSphere and caps distortion at .79, as the distortion gets unmusical between 0.8-1.0

As you test this code, you'll see that the half of our “Acoustic” sound closer to the “Distort” sphere remains undistorted. As you walk away from the “Distort” sphere, the distortion turns on, gradually increasing until it caps at .79. You can use this logic with any sound effect to build soundscapes that can vary dramatically based on user location or based on the movement of other objects. Some other handy effect code:

- When using reverb instead of distortion, adjust both:
    - `gameObject.GetComponent<AudioReverbFilter>().room`
    - `gameObject.GetComponent<AudioReverbFilter>().dryLevel`
- When using echo:
    - `gameObject.GetComponent<AudioEchoFilter>().wetMix`

Additional Points

Here are some random notes that didn't fit elsewhere:

iOS development specifics:

- Starting from iOS 10, Apple requires you to set the 'NSLocationWhenInUseUsageDescription' field in an App's info.plist file when location services are used--same for Camera & Microphone. You can set it in the iOS Player Settings 'Location Usage Description,' ‘Camera Usage Description,’ etc. We just used the text “Camera required for audio spatialization” for the camera; when combining the AR with GPS, we do the same for “Location Usage Description.” You should get warnings both when building the project in Unity & in Xcode if you are attempting to use location or camera services but this description is not set—however, sometimes you will not, and the app will simply not work correctly. Set it and save it early.

Unity Remote:

- Unity Remote was a great help on iOS, but remember that your phone likely won't be using its own data during tests. You will need to build the test app to your phone via XCode to use the phone's sensors. (Unity is working on an update as of guide writing).

Contact/About TCW

We are a duo from Washington, DC. You can contact us at tcw@tcwav.com.

- If you need help or get stuck, please do not hesitate to ask us. This process can be frustrating, especially if you aren't familiar with Unity. Please also share your creations with us—we can't wait to see what you do.
- If you are interested in having us compose a spatial audio composition for your space, or know someone who may be interested, please reach out to us or let them know.
- We are also interested in collaborating on spatial compositions.

Learn more about "Inauguration" – our debut spatial audio composition – on our website tcwav.com. The app launched 20 January 2017 in the Apple Store and Google Play Store.