Building Spatial Music Compositions for Smartphones Using GPS

A Guide for Relative and Absolute Locative Audio Apps on iOS and Android

v 2.1 (Updated 14 June 2022)

Introduction and Terms

This document outlines the procedures to develop and release spatial audio applications using the GPS sensors and sensors in smartphones. It uses two terms throughout:

- Absolute positioning: Objects that are attached to physical spots in the real world (i.e. the Washington Monument). No matter where a user launches their app, these objects will always appear at the same place on Earth (but do not have to remain there, as we will discuss below).
- Relative positioning: Objects whose locations in the real world are determined relative to the location a user launches the application (i.e. 15 meters in front of the user).

These two types of positioning allow us to approach spatial audio very differently: we can score specific places on the Earth using absolute positioning (gardens, monuments, streets, etc), or we can generate compositions that will work in people's backyards or local parks. It is entirely possible to combine both types of positioning in one composition.

For a more comprehensive look at immersive audio formats and terminology, see "Compositional Possibilities of New Interactive and Immersive Digital Formats" by Daniel Dehaan.

First, we will quickly discuss the technology we are using:

GPS and Compass

Our positioning data will not be as accurate or as detailed as we might like—GPS is generally only accurate to a distance of several meters (or worse, depending on your phone, connection and location). However, over decently sized spaces, and combined with the finer accuracy of a phone compass, it will do a good job outdoors.

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, Resonance Audio (Google), and more. This doesn't account for third-party

tools, like FMOD and WWise. In this guide, we will use the Resonance Audio. It is effective, highly customizable, reliable, and easy to implement.

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 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, creating a simulation of an entire room.

There are many places to look for inspiration and to credit here—Google, for building nice Resonance Audio documentation; partners at the 4DSOUND Hack Lab in 2014 (particularly Peter Kirn and the 4DSOUND Team) for some of the compositional concepts; and the groups BLUEBRAIN, Matmos and other for releasing previous absolute positioning audio applications.

Now, to building:

Software and Costs

You will need several pieces of software and hardware to develop and publish functioning applications for both iOS and Android. There is a $60 software cost we could not avoid, which is detailed below. Additionally, there are fees to publish to the Apple App Store and Google Play 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.
    - The most stable version of Unity for Android development is 5.3.7 as of February 2017. Use this version to save yourself from a ton of headaches.
- Resonance Audio for Unity (Free/Optional) – audio spatializer for Resonance Audio
- MapNav for Unity ($60) – our GPS/compass implementation
    - There are several open-source efforts to implement GPS into Unity, but none seem to work as well as MapNav, which is very effective and easy-to-use.
- 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)
    - To publish to app store, an Apple Developer account ($99/year to publish apps)
- To easily distribute for Android: $25 fee for the Google Play store

Downloads and Initialization:

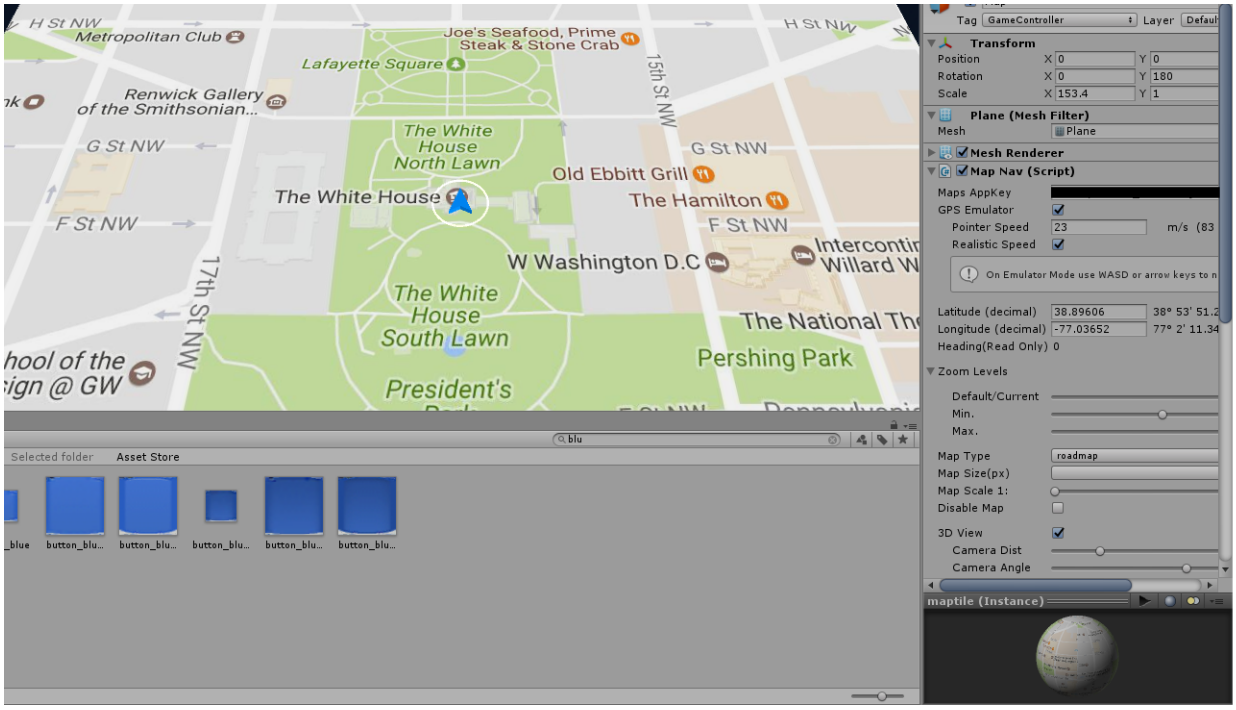
1. Download and install Unity from here. You only need “Unity Personal” for non-commercial purposes.
    a. Be absolutely sure during installation that you select the iOS and Android SDKs as part of the install.
2. If you choose to use Resonance Audio, you can download resources here: Resonance Audio.
3. Buy the Unity Asset “MapNav”
4. Consider phone emulators for testing. BlueStacks is wonderful for Android emulation.

Beginning your Project

Open the MapNav demo scene “MapNav_3D_Demo.” Title and save your project. Please note that, as we go, you will want to frequently save both your project and your scene.

At this point, you may want to switch from the default MapQuest service to Google Maps, Bing Maps, or something else. At minimum, you should change your API key. We switched to Google Maps due to their free licensing. MapQuest will charge you after 15,000 pings per month. To switch:

- Follow the directions here. The longer code didn’t quite work for us, but this did:
    - ```
      url= "http://maps.googleapis.com/maps/api/staticmap?center="+fixLat+","+fix
      Lon+"&zoom="+zoom+"&scale=2&size=640x640&format=jpg&maptype="+maptype[index
      Type]+"&sensor=false&key="+key;
      tempLat = fixLat;
      tempLon = fixLon;
      ```
- Generate a Google API key. (Ensure you hit “Enable” on the site).
- Enter your Google API key on the MapNav script in the “Map” Inspector (see black box below).
- Run your scene (toggle the GPS Emulator on in the MapNav script and hit “Play”) and ensure that Google Maps are appearing.
      ```



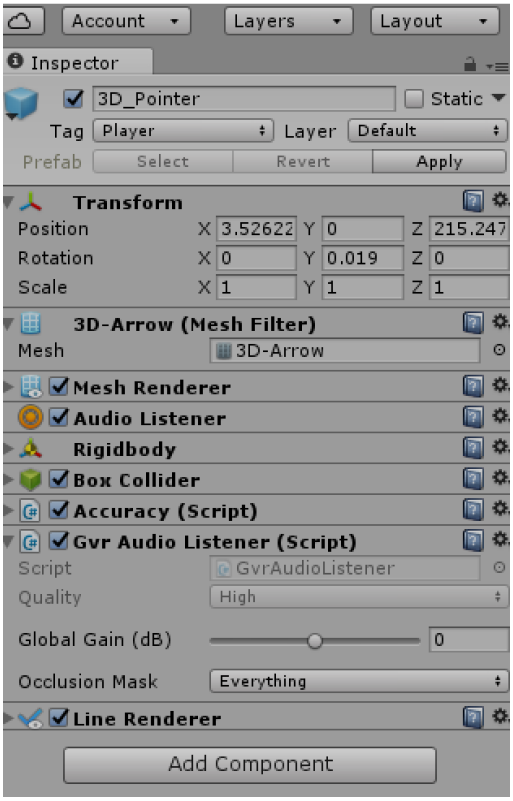
Google Maps appearing. Note the settings in the MapNav script on the right; we blacked out our personal API Key.

Applying Resonance Audio Spatialization

To use Resonance Audio, 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.
- Also, set the application to use the Resonance Audio spatialized. Go to Edit->Project Settings->Audio and set the Spatializer plugin and Ambisonic Decoder plugin to Resonance Audio.

Now we can add the Resonance Audio plugins. First, go to the object “3D Pointer.” This pointer represents the user’s location, so it will be where we want our “listener” to be. In the inspector, hit “Add Component” and add Resonance Audio Listener. Leave the standard Audio Listener on.



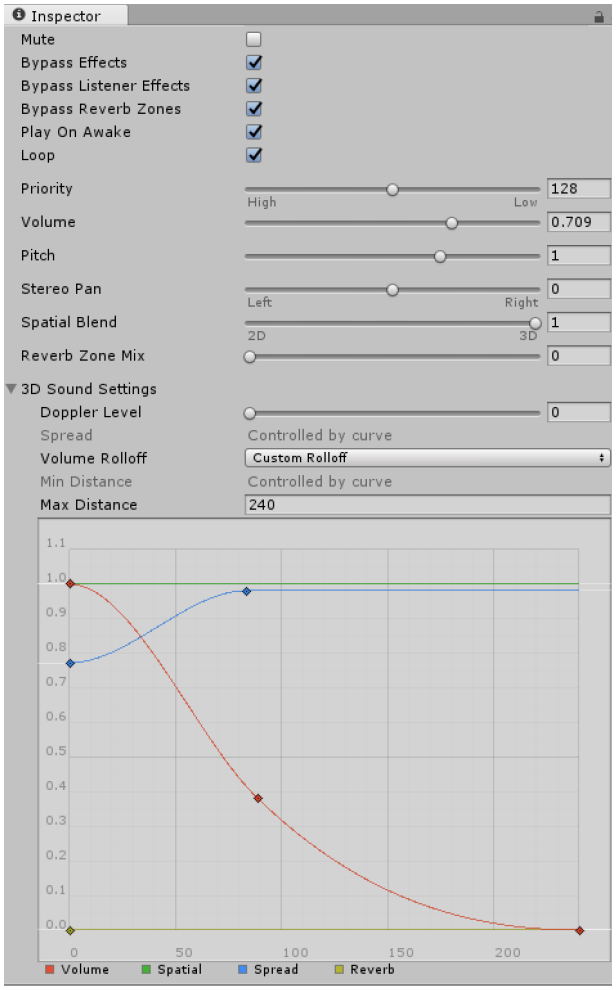
Note: this picture shows an older version of the listener called "GVR Audio Listener"

Next, we turned off or removed all the pre-built audio sources in the MapNav demo (like the BirdSounds and the Bus) to remove distractions. If you don't want to delete them, you will want to mute their "Audio Source" components and add the GVR Audio Source, as detailed below.

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

- GameObject->3D Object->Sphere.
- Import an audio file into your Unity project assets folder, then select it in "Audio Source" by hitting the small circle. You now have a sound object. (You can also use the "Resonance Audio Source" option for different controls. If you do this, uncheck the standard "Audio Source").
    - 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.
- We also suggest you bypass effects and zones by default, then selectively reenable these options as you program for artistic effect. Ensure you “Spatialize” the effects at first.

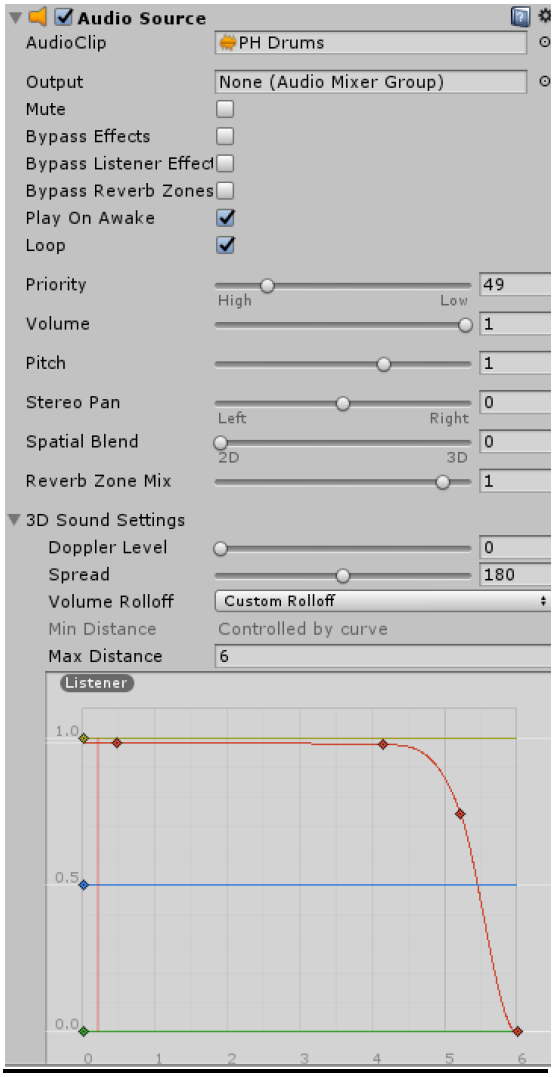


An example “Audio Source” with custom rolloffs

- 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.

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). It can sound natural to increase the spread of a sound toward its center.

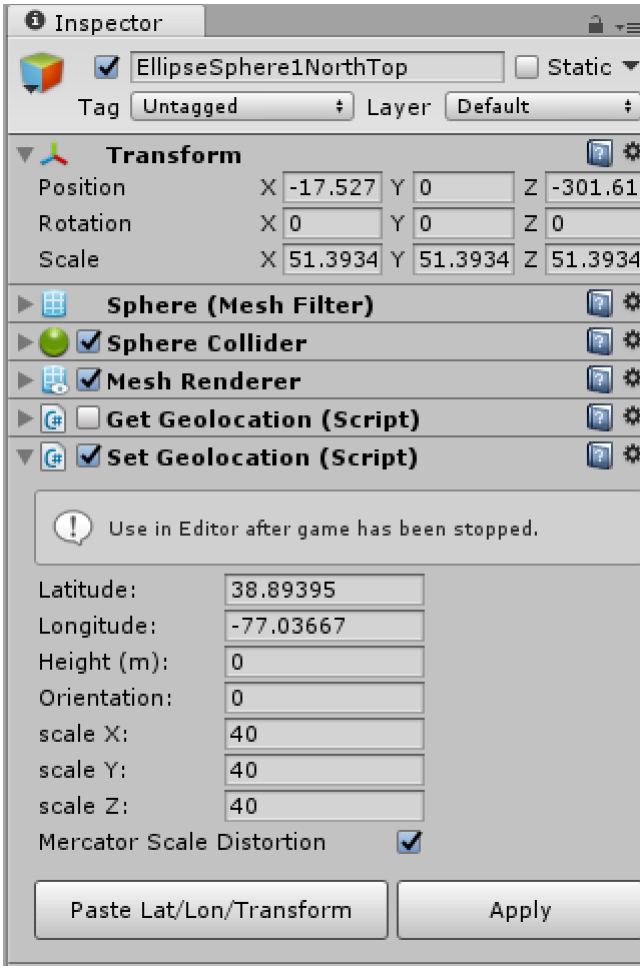


Positioning Your Spheres

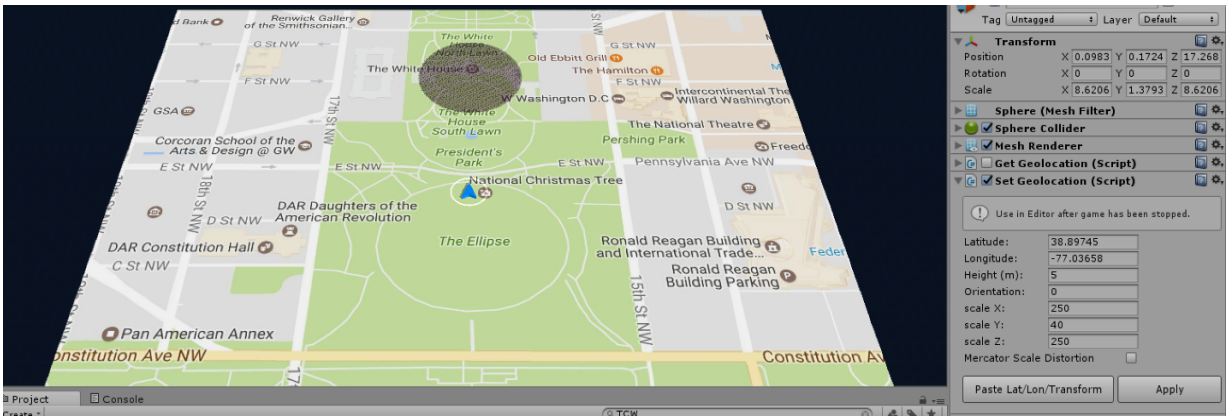
You now have a working map with working compass and GPS implementation. You also know how to create sound objects. Now we will discuss positioning those sound objects spatially.

Absolute Positioning

- Add MapNav's "Get Geolocation" and "Set Geolocation" scripts to the sphere you just created.
    - As you begin to grow your app, consider making empty Unity objects, positioning those using GPS, and making all your spheres children of that parent. Then you only need to GPS position as little as one object for a whole composition.
- Instructions for Set/Get Geolocation are here. You can also just type in Lat/Long coordinates, which you can find on Google Maps by double-clicking a spot (repeatedly) at a small scale. Make sure "Get" is checked off and "Set" is checked on once you are done. Also make sure you hit "Apply" after each coordinate change.
    - Always do your scaling and positioning in "Set Geolocation," it overrides the standard scale controls.
    - Mercator Distortion will scale your objects relative to their position on the Earth. We generally unchecked it so that we could relate the absolute sphere sizes to absolute audio rolloff ranges. There's no implicit harm in leaving it on.



Here's what your inspector should look like. Note the Mercator distortion changes the scale from "40" to "51.3934" in Washington, DC. With distortion off, it will be consistently "40."



A black mark/sound over the White House. No matter where you launch the app, or what orientation you have, the black mark will stay there.

Relative Positioning

- Relative positioning is much simpler than absolute positioning. All you need to do is set the X,Y, and Z coordinates in the sphere's "Transform" component. No need to add "Get Geolocation" or "Set Geolocation." Just adjust and experiment.
- Be cautious that Mercator Distortion may impact any compositions that use relative positioning depending on where a user is.

No matter where you launch the app from, these spheres will launch at these positions around you.

Relative Movement

Now we're going to move some sounds. "Relative Movement" is used to describe movement for objects that use Relative Positioning, as described above, or are children of a parent object that is positioned using GPS. This is easier than Absolute Movement for spheres that are individually placed using GPS, detailed further below. You can use Unity Animations to control your object movement, which is much easier than scripting. For scripting, you can use custom scripts to control your object movement.

There are tons of tutorials on the Internet for scripts, 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. 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;
             }
       }
}
```

Absolute Movement:

Movement for individual objects that have been placed through Absolute Positioning using GPS is harder, but still very doable. The critical thing to remember in any coding is to never use “Transform.position” but to use “Transform.offset” – which will not alter the initial geospatial positioning but make all movements relative to that initial position.

We built our transformation offsets into new geolocation scripts to keep things easy, but you could build them separately as well. To build them into geolocation scripts, duplicate MapNav’s “Set Geolocation” script, rename it, and update your duplicated script code with this new script name toward the top. Then you can build your movement scripts into your new script. We are not providing full scripts because we are not authorized to share MapNav’s code, but here is an example of what to add to a script to make your sphere move in a circle. Remember, this is an offset, so the circular movement will be centered around the point you’ve picked based on the radius you specify in the code.

Add these up top:

```
      [Range(0, 200)]
    public float RotateSpeed = 1f;
    [Range(0, 200)]
    public float Radius = 1f;
    [Range(0, 360)]
    public float TCWEllipse;
    private Vector3 _centre;
    private float _angle;
    public Vector3 Velocity = new Vector3(0, 0, 0); //Would introduce movement along x y
or z after each spin
```

Add these to IEnumerator Start ()

```
    _centre = transform.position;
    _angle = TCWEllipse;
```

Add these at bottom:

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

    _angle += RotateSpeed * Time.deltaTime;

    var offset = new Vector3(Mathf.Sin(_angle), 0, Mathf.Cos(_angle)) * Radius;

    transform.position = _centre + offset;
}
```

And adjust parameters in the Inspector, accordingly.

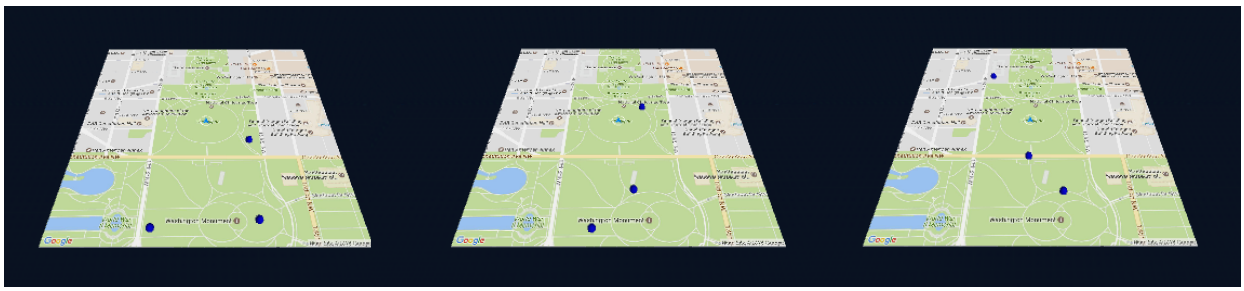
- If you do use this same method, ensure you are updating both the original MapNav “Set Geolocation” coordinates, hitting “Apply,” and updating your new script with the same coordinates every time you make a coordinate change for the initial position. Uncheck the original “Set Geolocation” script so that it’s disabled.

Other Movement Patterns

You will find videos and descriptions of other 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, 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. Perhaps the distance changes an effect, like their pitch. The listener will never hear the same sequence and spaces twice.



These blue spheres are randomly changing locations every 12 seconds per the script below from TCW's "Inauguration"

Here is positioning code written as an offset for absolute positioning (it leverages the Absolute Positioning Circle script, above). It will randomly move the spheres along the x and z axes within the specified offset ranges of -300 to 300 and -1100 to 300 (A script that will work for relative/child positioning can be found in our AR guide):

Put these up top:

```
Vector3 pos;
float x;
float y;
float z;
```

Add this in your "IEnumerator Start()" under existing code:

```
InvokeRepeating("TCWRandomLocation", 2.0f, 12f);
```

At the end (in place of the "private void Update()"):

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

_angle += RotateSpeed * Time.deltaTime;

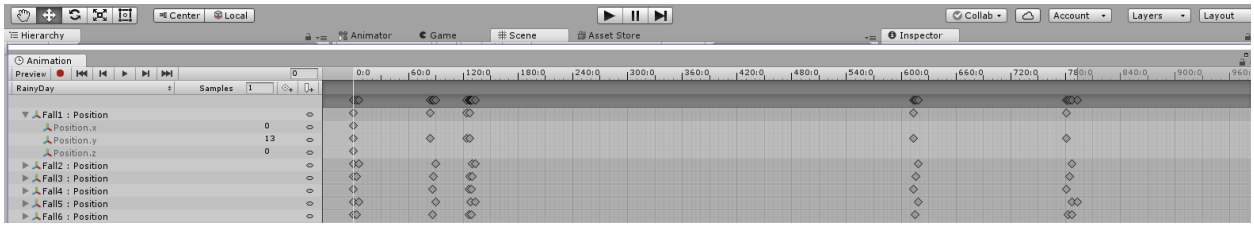

var offset = new Vector3(UnityEngine.Random.Range(-300, 300), y,
UnityEngine.Random.Range(-1100, 300));

transform.position = _centre + offset;
```

You may notice in the implementation of absolute positioning randomization that sounds are cutting off too abruptly. For Resonance Audio, see the "Audio Rooms" section on the next page. For Unity's default audio spatialization, see "Unity Audio Effects."

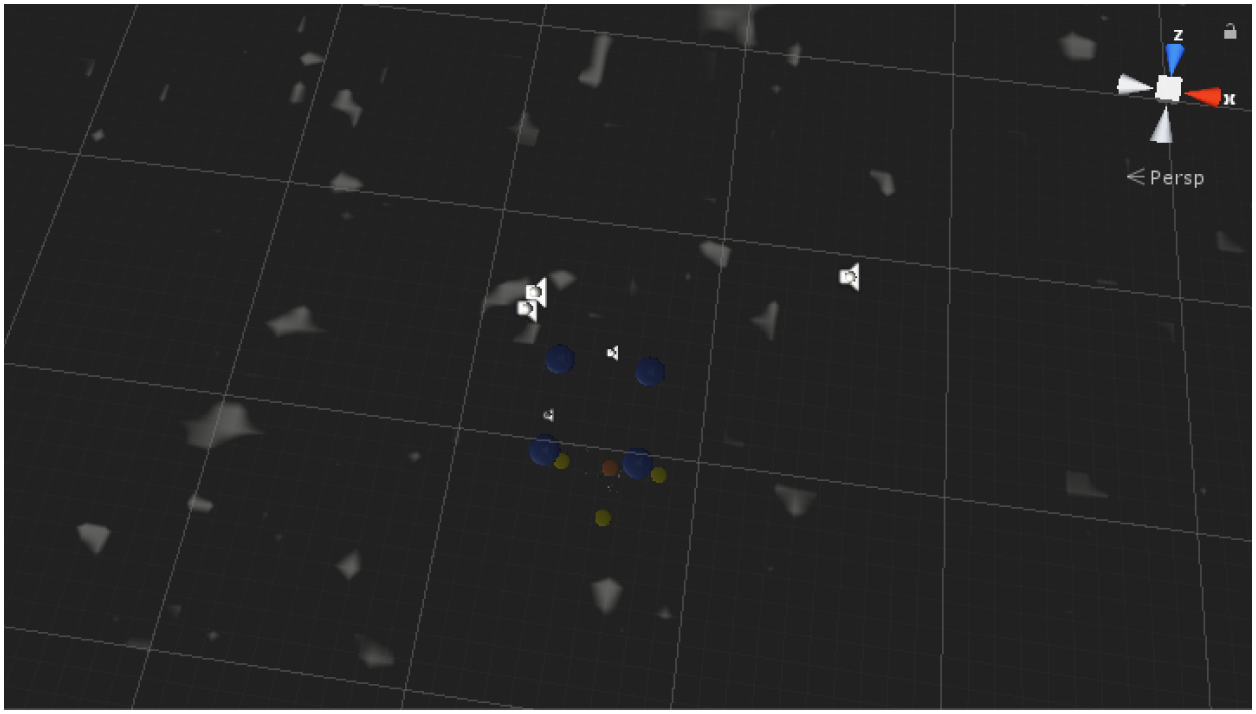
Rain

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;

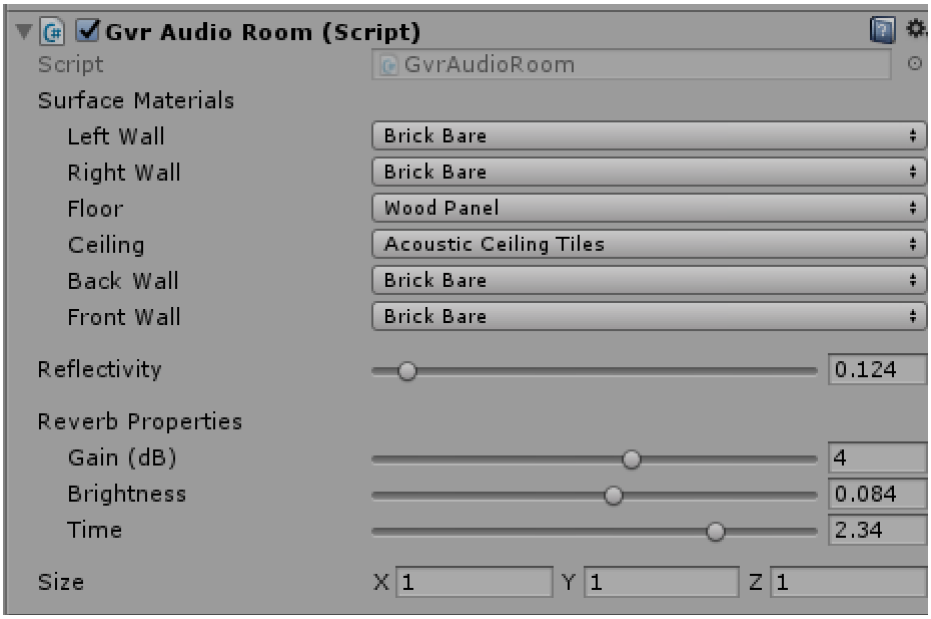

    }

}
```

Audio Rooms

The Resonance Audio package also includes “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. They seem fairly resource-intensive.

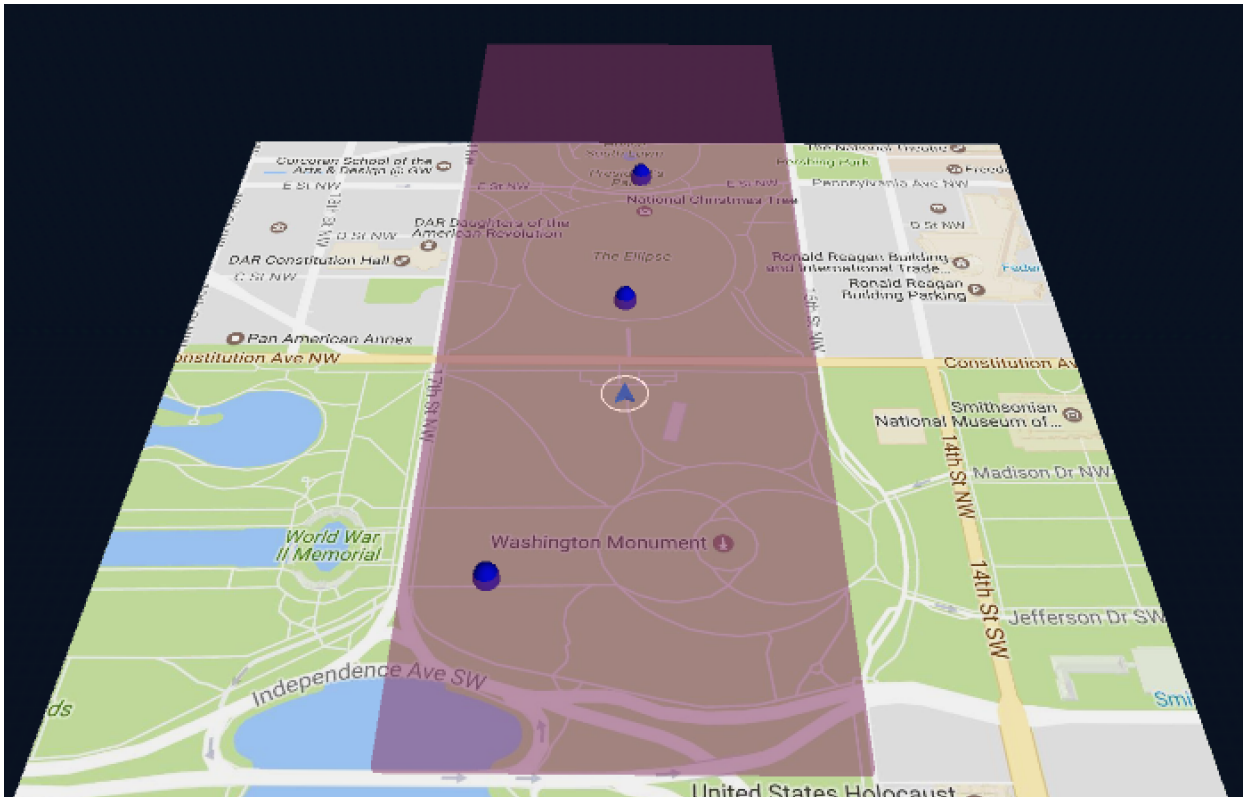
One practical use we implemented was to use the rooms to create a subtle reverb tail.



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.



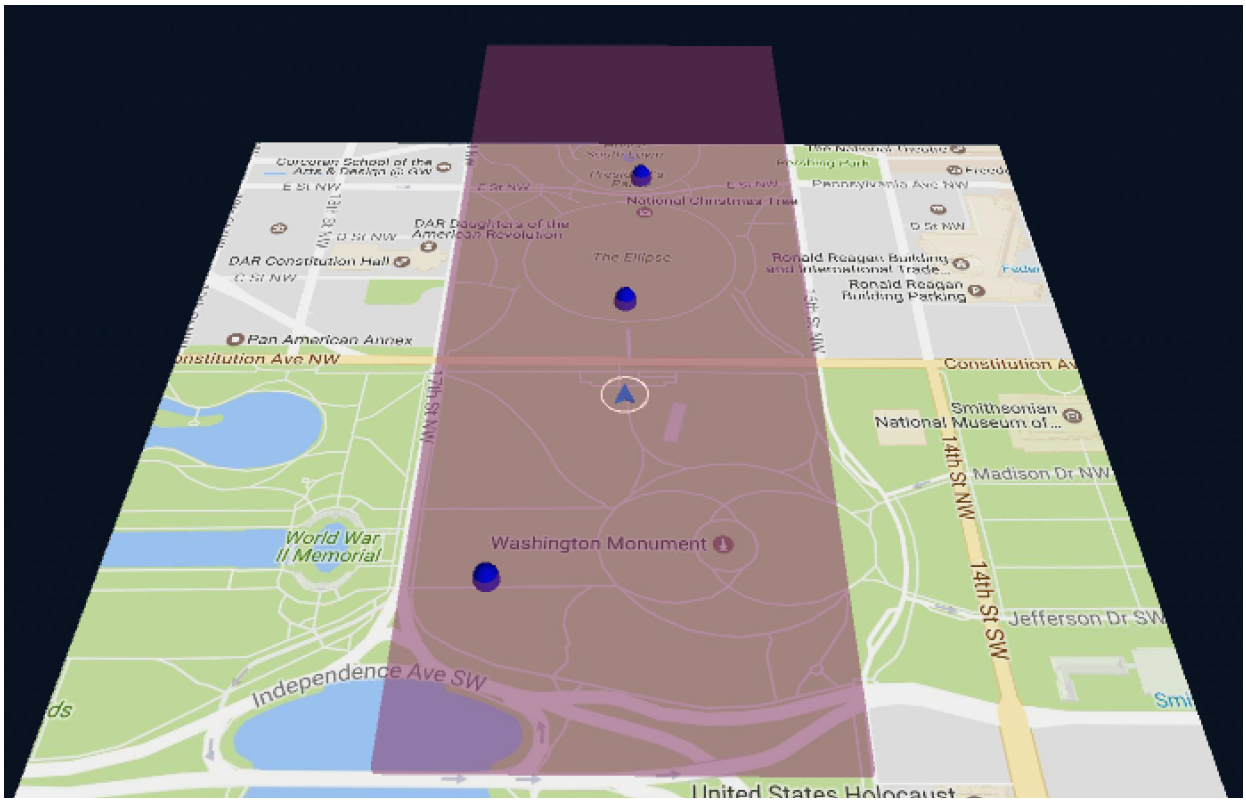
A large “room” of reverb

Unity Audio Effects

By default, Unity includes Chorus, Distortion, Reverb, EQ filters, and more. These are all fairly self-explanatory. It also includes “Reverb Zones,” which can be very effective for your compositions. Similar to the Resonance 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 Resonance 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.

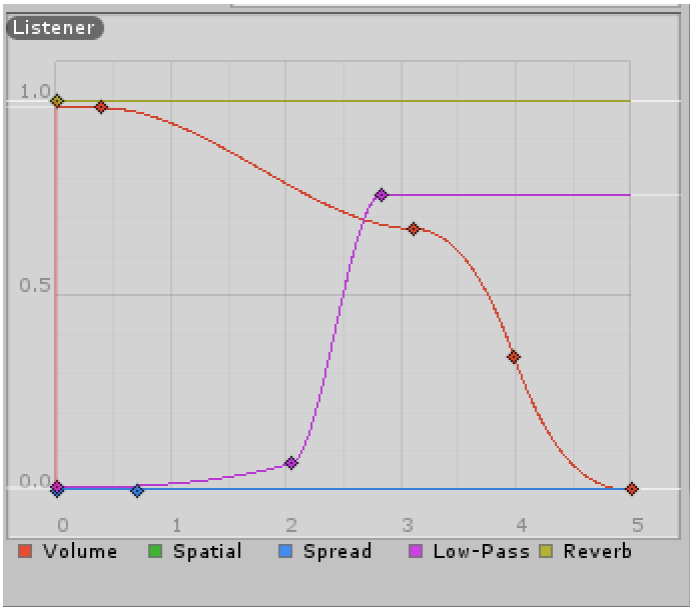
- Ensure your other sounds all have “Bypass Reverb Zones” checked to avoid unwanted reverb. Only leave Reverb Zones enabled on the objects where you want the extra reverb.
- Make sure you set the Min and Max distances appropriately for your objects. This is easily done by picking an extreme reverb, like “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 in your world. Create another sphere with a sound. 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;

        }

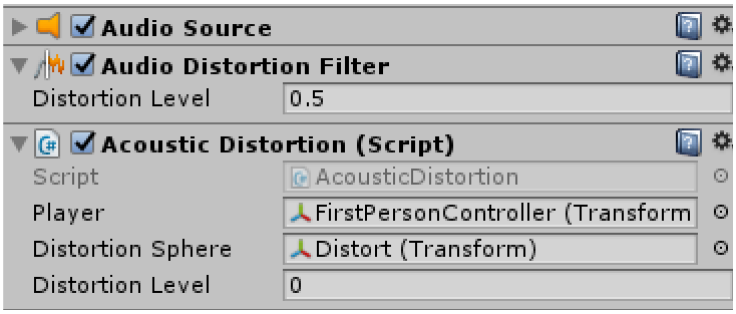
        if (Vector3.Distance(Player.position, DistortionSphere.position) > 7.82)
        {
            distortionLevel = 0.79f;
            gameObject.GetComponent<AudioDistortionFilter>().distortionLevel = 0.79f;

        }


        //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.”



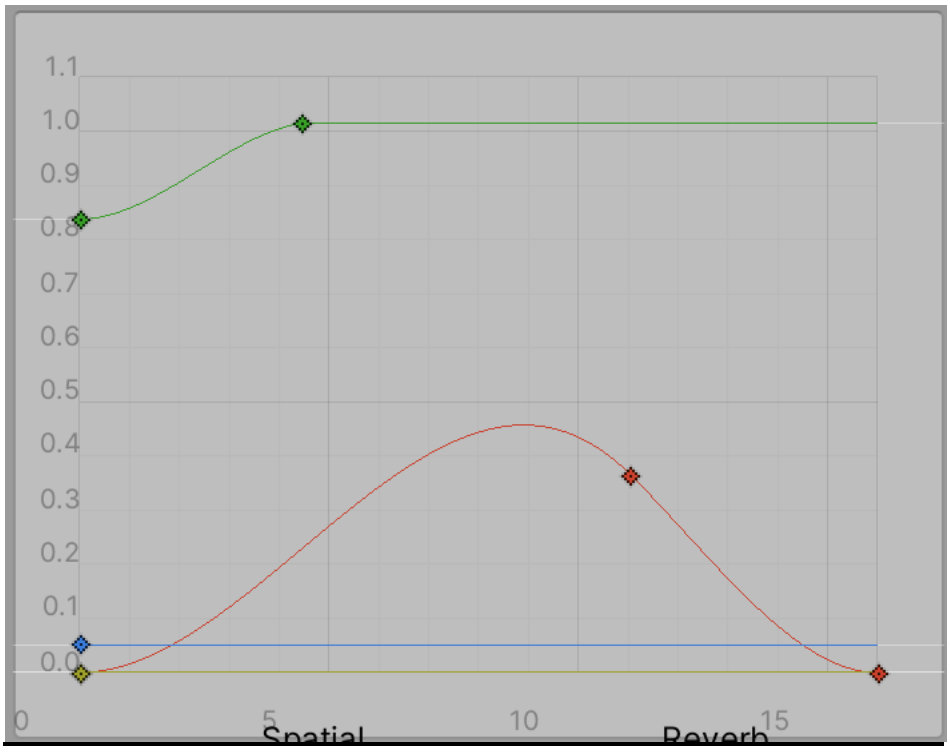
- You will see that this code calculates the distance between object “Player” and object “Distortion Sphere,” divides them by 2, and subtracts 3.055. This math is based on the distance we set between “Distortion” and “Acoustic” earlier, which was very small (~6 in Unity measurements). Adjust the math based on the distance between objects.

    - 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`

Another great way to implement distance-based effects is to take a normal (clean) sound, insert it onto a sphere, and duplicate the sphere. Now, use your DAW or similar to add effects you want to the clean sound, and save the effected sound into Unity. Add it to the duplicated sphere, and set it so the volume rises or oscillates as the listener moves farther away. Now, the clean sound appears to become effected as the listener moves from it.



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'. We just used the text "Location required for audio spatialization." You should get warnings both when building the project in Unity & in Xcode if you are attempting to use location services but the description is not set—however, sometimes you will not, and the app will simply not work correctly. Set it and save it early.

Android development specifics:

- We successfully built our app targeting OSes 4.4 (Kit Kat) and up with the default Unity audio spatialization.
- If you have conflicting Android Manifests (Standard, Daydream, and Cardboard)—you can likely standardize them using the text in "Android-Manifest Cardboard."

Unity:

- Don't change your Map Scale in the Map script once you've begun coding. It will disrupt your positioning offsets.
- Your sounds will likely all play at once while your app finds its GPS signal. To stop this, follow this procedure:
    - Create an empty game object and nest all your sound objects under it.
    - Put a script on it called TCWAudio. Set all your audio objects to be inactive in the Inspector (by unchecking their boxes in the top left of the Inspectors). This script, when called, will change them all to active

```
using UnityEngine;
using System.Collections;

public class TCWAudio : MonoBehaviour
{
void Start()
{
  gameObject.SetActiveRecursively(true); // activates the child sounds
                                                                    }}
```

In the "MapNav" script, add up top:

```
public GameObject GPSFixAudio;
```

Find this text in the MapNav script:

```
 //Successful GPS fix
```

```
 gpsFix = true;
//Update Map for the current location
          StartCoroutine(MapPosition());
```

Right below it, add:

```
GPSFixAudio = GameObject.FindGameObjectWithTag("AllAudio");
      GPSFixAudio.GetComponent<TCWAudio>().enabled = true;
```

Note—this method uses “FindGameObjectWithTag” instead of directly finding the GameObject name. We like this approach, because it allows us to create multiple parent objects that we can turn on and off. You will, of course, need to add a tag like “AllAudio” to the parent object you just created for it to work. Consider adding a slight delay (“WaitForSeconds”) if this method is not working perfectly. In recent coding, Apple has complained about the MapNav setting to quit if Location Permission aren’t enabled. You may want to delete or modify those lines of code in the MapNav script.

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 on our website tcwav.com.