

# SFX System - Comprehensive Technical Manual

---

**Version:** 1.0 **Last Updated:** 2025-10-19 **Unity Compatibility:** Unity 2020.3 LTS and above

---

## Table of Contents

---

1. [System Overview](#)
  2. [Architecture](#)
  3. [Core Components](#)
  4. [Audio Containers](#)
  5. [Events System](#)
  6. [Bus & Mixing](#)
  7. [State System](#)
  8. [Voice Management](#)
  9. [Advanced Features](#)
  10. [Music System](#)
  11. [Performance & Optimization](#)
  12. [API Reference](#)
  13. [Best Practices](#)
  14. [Troubleshooting](#)
-

# 1. System Overview

---

## What is the SFX System?

The SFX System is a professional-grade audio middleware for Unity, inspired by industry-standard audio tools. It provides advanced audio features including:

- **Container-based audio organization** with multiple playback modes
- **Event-driven audio triggering** with complex action sequences
- **Hierarchical bus mixing** with volume control and ducking
- **State-based audio management** for dynamic game states
- **Voice virtualization and LOD** for performance optimization
- **RTPC (Real-Time Parameter Control)** for dynamic audio parameters
- **Occlusion and spatial audio** support

## Key Benefits

- **Designer-Friendly:** ScriptableObject-based workflow, no coding required for basic setup
  - **Performance:** Automatic voice pooling, virtualization, and LOD management
  - **Professional:** Industry-standard patterns from AAA audio middleware
  - **Flexible:** Supports simple 2D sounds to complex 3D spatial audio
  - **Scalable:** Handles projects from small indie games to large productions
- 

# 2. Architecture

---

## System Design Philosophy

The SFX System follows these architectural principles:

1. **Data-Driven:** Audio assets are ScriptableObjects, allowing designers to work in the Unity Editor
2. **Event-Based:** Audio is triggered via events, not direct API calls to sources
3. **Hierarchical:** Buses form a tree structure for mixing and routing
4. **Pooled:** Audio voices are pre-allocated and reused for efficiency

5. **Modular:** Partial classes separate concerns (Core, Music, Buses, States, etc.)

## Component Hierarchy

```

AudioManager (MonoBehaviour Singleton)
├── Voice Pool (AudioVoiceEnhanced objects)
├── Bus Hierarchy (AudioBus ScriptableObjects)
├── Event Library (AudioEvent ScriptableObjects)
└── State Groups (AudioState ScriptableObjects)

Audio Assets
├── Containers (AudioContainer ScriptableObjects)
│   ├── RoutingContainer (simple playback)
│   ├── RandomContainer (weighted random selection)
│   ├── SequenceContainer (sequential playback)
│   ├── SwitchContainer (switch-based selection)
│   └── BlendContainer (layered mixing)
├── Events (AudioEvent ScriptableObjects)
├── Buses (AudioBus ScriptableObjects)
└── States (AudioState ScriptableObjects)

```

## Core Data Flow

```

1. Game Code → Post AudioEvent
2. AudioEvent → Execute Actions (Play, SetRTPC, SetState, etc.)
3. Action → Select Container (via Switch, State, etc.)
4. Container → Request Voice from AudioManager
5. AudioManager → Allocate AudioVoiceEnhanced from pool
6. Voice → Assign to Bus, apply gain stack
7. Voice → Play AudioClip through AudioSource
8. Runtime → Voice management (virtualization, LOD, occlusion)
9. Completion → Return voice to pool

```

## 3. Core Components

### 3.1 AudioManager

**Location:** `Core/AudioManager.Core.cs` (and partial classes)

The central singleton managing all audio operations.

## Key Responsibilities

- **Voice Lifecycle:** Allocating, tracking, and returning audio voices
- **Bus Management:** Hierarchical volume control and mixing
- **Event Processing:** Triggering and managing audio events
- **State Management:** Game state-based audio modifications
- **RTPC System:** Real-time parameter control
- **Performance:** Voice virtualization, LOD, and occlusion

## Initialization

The AudioManager initializes automatically on scene load via `Awake()` :

```
private void Awake()
{
    // Singleton pattern
    if (Instance != null && Instance != this)
    {
        Destroy(gameObject);
        return;
    }
    Instance = this;
    DontDestroyOnLoad(gameObject);

    InitializeCore();           // Voice pool creation
    InitializeBuses();          // Bus hierarchy setup
    LoadAudioAssets();          // Load events and states from Resources

    // Start background systems
    StartCoroutine(VoiceManagementUpdate());
    if (enableOcclusion) StartCoroutine(OcclusionUpdate());
}
```

## Inspector Configuration

Property	Description	Default
<code>masterMixerGroup</code>	Root AudioMixerGroup for all audio	null
<code>masterVolume</code>	Global volume multiplier (0-1)	1.0
<code>muteAll</code>	Mute all audio output	false
<code>maxRealVoices</code>	Maximum physically playing voices	32
<code>maxVirtualVoices</code>	Maximum virtualized voices	64
<code>voiceUpdateInterval</code>	Voice management update rate (seconds)	0.1
<code>enableOcclusion</code>	Enable raycast-based occlusion	true
<code>occlusionMask</code>	LayerMask for occlusion raycasts	~0 (all)
<code>occlusionUpdateInterval</code>	Occlusion update rate (seconds)	0.2
<code>enableLOD</code>	Enable distance-based voice LOD	true
<code>lodDistances</code>	Distance thresholds for LOD levels	[10, 25, 50, 100]

## 3.2 AudioVoiceEnhanced

**Location:** `Events/AudioVoiceEnhanced.cs`

Represents a single playing audio instance with advanced features.

## Structure

```
public class AudioVoiceEnhanced : AudioVoice
{
    public GainStack gainStack;           // Layered volume control
    public AudioEvent sourceEvent;        // Originating event
    public AudioContainer container;       // Source container
    public AudioBus bus;                  // Assigned bus
    public int priority;                   // 0-255 (see VoicePriority enum)
    public bool isVirtual;                 // Virtualization state
    public float virtualTime;              // Virtual playback time
    public double scheduledStartTime;      // DSP scheduled start
    public double scheduledEndTime;        // DSP scheduled end
    public AudioLowPassFilter lowPassFilter; // Cached filter reference
}
```

## GainStack

Voices use a multiplicative gain stack for flexible volume control:

```
public class GainStack
{
    public float baseGain = 1f;           // Container/Event base volume
    public float busGain = 1f;            // Bus volume contribution
    public float occlusionGain = 1f;       // Occlusion attenuation
    public float rtpcGain = 1f;           // RTPC-driven gain
    public float schedulerGain = 1f;       // Crossfade/scheduler gain

    public float GetFinalGain() =>
        baseGain * busGain * occlusionGain * rtpcGain * schedulerGain;
}
```

## 3.3 Voice Priority

**Location:** Core/AudioSystemEnums.cs

Priority determines voice stealing behavior:

```
public enum VoicePriority
{
    Low = 0,          // Background ambience, least important
    Medium = 64,       // Standard gameplay sounds
    High = 128,        // Important feedback (damage, pickups)
    Critical = 255     // UI, dialogue, never steal
}
```

---

## 4. Audio Containers

---

Containers define how audio clips are organized and played.

### 4.1 Container Base Class

**Location:** Containers/audio-container-base.cs

All containers inherit from `AudioContainer` `ScriptableObject`.

## Common Properties

Property	Description
<code>containerName</code>	Unique identifier for the container
<code>description</code>	Designer notes and documentation
<code>tags</code>	Searchable tags for organization
<code>mixerGroup</code>	AudioMixerGroup override (optional)
<code>enableVolumeRandomization</code>	Randomize volume per play
<code>volumeRandomMin/Max</code>	Volume range in dB (-12 to +12)
<code>enablePitchRandomization</code>	Randomize pitch per play
<code>pitchRandomMin/Max</code>	Pitch range in cents (-1200 to +1200)
<code>is3D</code>	Enable 3D spatial audio
<code>minDistance</code>	Distance for full volume (if 3D)
<code>maxDistance</code>	Distance for zero volume (if 3D)
<code>rolloffMode</code>	Logarithmic, Linear, or Custom

## 4.2 RoutingContainer

**Location:** `Containers/routing-container.cs` **Menu:** `Audio System/Routing Container`

Simplest container type. Plays all assigned clips simultaneously.

### Use Cases

- Simple one-shot sounds (footsteps, impacts)
- Multi-layered sounds (explosion with debris layer)



## Properties

Property	Description
audioClips	List of AudioClips to play
volume	Base volume (0-1)
loop	Loop all clips

## Example

```
// Create via Assets > Create > Audio System > Routing Container
// Assign AudioClips in the Inspector
// Play from code:
[SerializeField] private RoutingContainer explosionSound;

void OnExplode()
{
    explosionSound.Play(transform.position);
}
```

## 4.3 RandomContainer

**Location:** Containers/random-container.cs **Menu:** Audio System/Random Container

Plays one random clip per trigger, with weighting and repeat avoidance.

### Use Cases

- Footsteps with variation
- Weapon fire variations
- Environmental ambience
- Voice line selection

## Properties

Property	Description
<code>audioClips</code>	List of <code>WeightedAudioClip</code> entries
<code>avoidRepeatLast</code>	Number of previous clips to exclude (0-10)
<code>useWeighting</code>	Enable weighted random selection
<code>volume</code>	Base volume (0-1)
<code>loop</code>	Loop the selected clip

## WeightedAudioClip Structure

```
[System.Serializable]
public class WeightedAudioClip
{
    public AudioClip clip;
    public float weight = 1f;           // 0-10 (higher = more likely)
    public float volumeMultiplier = 1f; // Per-clip volume (0-1)
}
```

## Example

```
[SerializeField] private RandomContainer footstepSounds;

void PlayFootstep()
{
    // Plays one random footstep, avoiding the last 2 played
    footstepSounds.Play(transform.position);
}
```

## 4.4 SequenceContainer

**Location:** Containers/sequence-container.cs **Menu:** Audio System/Sequence Container

Plays sounds in a defined order with multiple playback modes.

## Use Cases

- Musical sequences
- Dialogue trees
- Tutorial sequences
- Alarm patterns

## Properties

Property	Description
<code>entries</code>	List of <code>SequenceEntry</code> items
<code>playbackMode</code>	Forward, Reverse, PingPong, Random
<code>loopSequence</code>	Loop the entire sequence
<code>volume</code>	Base volume (0-1)

## SequenceEntry Structure

```
[System.Serializable]
public class SequenceEntry
{
    public AudioClip clip;
    public float volumeMultiplier = 1f; // Per-clip volume
    public bool loop;                    // Loop this specific clip
    public float delayAfter = 0f;        // Delay before next clip (0-5s)
}
```

## Playback Modes

- **Forward:** 0 → 1 → 2 → 3 → (loop to 0)
- **Reverse:** 3 → 2 → 1 → 0 → (loop to 3)
- **PingPong:** 0 → 1 → 2 → 3 → 2 → 1 → 0 → ...
- **Random:** Random order each time

## Methods

```
public void ResetSequence();           // Reset to index 0
public void SetIndex(int index);       // Jump to specific index
```

## Example

```
[SerializeField] private SequenceContainer alarmSequence;

void TriggerAlarm()
{
    alarmSequence.ResetSequence();
    alarmSequence.Play(); // Plays sequence from start
}
```

## 4.5 SwitchContainer

**Location:** Containers/switch-container.cs **Menu:** Audio System/Switch Container

Selects a child container based on a game state switch.

### Use Cases

- Surface-dependent footsteps (grass, metal, wood)
- Weather-dependent ambience
- Character-specific voice lines
- Weapon-type sounds

### Properties

Property	Description
switchGroupName	Name of the switch group to monitor
switchEntries	List of SwitchEntry mappings
defaultContainer	Fallback if no match found
volume	Volume multiplier

## SwitchEntry Structure

```
[System.Serializable]
public class SwitchEntry
{
    public string switchValue;      // Switch value to match
    public AudioContainer container; // Container to play
}
```

## Example

```
// Setup in Inspector:
// Switch Group Name: "Surface_Type"
// Switch Entries:
//   - "Grass" → GrassFootstepsContainer
//   - "Metal" → MetalFootstepsContainer
//   - "Wood" → WoodFootstepsContainer
// Default Container: GenericFootstepsContainer

[SerializeField] private SwitchContainer footstepSwitch;

void PlayFootstep()
{
    // Set the current surface type
    AudioManager.Instance.SetSwitch("Surface_Type", "Metal");

    // Play automatically selects the correct container
    footstepSwitch.Play(transform.position);
}
```

## 4.6 BlendContainer

**Location:** Containers/blend-container.cs **Menu:** Audio System/Blend Container

Plays multiple containers simultaneously with RTPC-driven crossfading.

### Use Cases

- Layered music (intro/loop/intensity layers)
- Vehicle engine (idle/load/redline layers)
- Proximity-based sound design
- Weather intensity blending

## Properties

Property	Description
<code>blendEntries</code>	List of BlendEntry layers
<code>blendParameterName</code>	RTPC parameter name to monitor
<code>volume</code>	Master volume for all layers
<code>loop</code>	Loop all layers

## BlendEntry Structure

```
[System.Serializable]
public class BlendEntry
{
    public AudioContainer container;    // Layer to play
    public AnimationCurve volumeCurve; // Volume vs. RTPC value
}
```

## How It Works

1. All layers start playing simultaneously
2. RTPC value (0-1) is evaluated against each layer's curve
3. Layer volumes update dynamically in real-time

## Example

```
// Setup in Inspector:
// Blend Parameter Name: "MusicIntensity"
// Blend Entries:
//   - AmbientLayer → Curve: 1.0 at x=0, 0.0 at x=1
//   - CombatLayer → Curve: 0.0 at x=0, 1.0 at x=1

[SerializeField] private BlendContainer musicBlend;

void Start()
{
    musicBlend.Play(); // All layers start
    AudioManager.Instance.SetRTPC("MusicIntensity", 0f); // Ambient only
}

void EnterCombat()
{
    // Smoothly transition to combat layer over 2 seconds
    AudioManager.Instance.TransitionRTPC("MusicIntensity", 1f, 2f);
}
```

---

## 5. Events System

---

**Location:** Events/AudioEvent.cs

Events are the primary interface for triggering audio in your game.

### 5.1 AudioEvent Structure

AudioEvents are ScriptableObjects that define one or more actions to execute.

## Properties

Property	Description
eventName	Unique event identifier
actions	List of EventAction items to execute
priority	Voice priority (Low/Medium/High/Critical)
maxInstances	Instance limit (0 = unlimited)
stealBehavior	Oldest/Quietest/Furthest/LowestPriority
cooldown	Minimum time between posts (seconds)

## 5.2 Event Actions

Each action defines a single operation when the event is posted.

### ActionType Enum

```
public enum ActionType
{
    Play,           // Play a container
    Stop,           // Stop active instances
    Pause,          // Pause active instances
    Resume,         // Resume paused instances
    SetSwitch,      // Change a switch value
    SetRTPC,        // Change an RTPC value
    SetState,       // Activate a state
    TriggerDucking, // Duck a bus
    CrossFade       // Crossfade between containers
}
```



## Action Properties

Property	Description	Relevant Actions
type	Action to execute	All
container	Target container	Play, CrossFade
targetBus	Target bus	Play, TriggerDucking
delay	Delay before execution (seconds)	All
fadeDuration	Fade/transition time (seconds)	Stop, RTPC, State, CrossFade
fadeCurve	Animation curve for fades	Stop
switchGroup	Switch group name	SetSwitch
switchValue	Switch value	SetSwitch
rtpcName	RTPC parameter name	SetRTPC
rtpcValue	RTPC value	SetRTPC
stateName	State name	SetState
fadeFromContainer	Source container	CrossFade
crossfadeType	Linear/EqualPower	CrossFade

## 5.3 Posting Events

### From Code

```
// Simple post
[SerializeField] private AudioEvent footstepEvent;
footstepEvent.Post(gameObject, transform.position);

// With handle for runtime control
AudioHandle handle = footstepEvent.Post(gameObject, transform.position);
handle.SetVolume(0.5f);
handle.SetPitch(2f); // +2 semitones
handle.Stop(0.5f);   // Fade out over 0.5s
```

### Event Handles

`AudioHandle` provides runtime control over posted events:

```
public class AudioHandle
{
    public bool isPlaying { get; }    // Is event currently playing?
    public float time { get; }        // Current playback time
    public float duration { get; }    // Total duration

    // Control methods
    public void SetVolume(float linear);    // 0-1 volume
    public void SetPitch(float semitones);  // Pitch in semitones
    public void Stop(float fadeTime = 0.1f); // Stop with fade
    public void Pause();                   // Pause playback
    public void Resume();                   // Resume playback
    public void Dispose();                  // Clean up callbacks

    // Event callbacks
    public event Action OnStarted;          // Fired when playback starts
    public event Action OnLoop;             // Fired on loop point
    public event Action OnFinished;         // Fired when playback ends
}
```

## 5.4 Voice Stealing

When `maxInstances` is exceeded, the system steals a voice based on `stealBehavior`:

- **Oldest:** Replace the longest-playing instance
- **Quietest:** Replace the quietest instance

- **Furthest:** Replace the instance furthest from listener
- **LowestPriority:** Replace the lowest-priority instance

## 5.5 Example: Complex Event

```
// Create via Assets > Create > Audio System > Audio Event
// Configure in Inspector:

Event Name: "Explosion"
Priority: High
Max Instances: 3
Steal Behavior: Furthest
Cooldown: 0.1

Actions:
  [0] Play
      Container: ExplosionContainer
      Target Bus: SFX_Bus
      Delay: 0

  [1] TriggerDucking
      Target Bus: Music_Bus
      Fade Duration: 0.1

  [2] SetRTPC
      RTPC Name: "ScreenShake"
      RTPC Value: 1.0
      Fade Duration: 0.5
```

## 6. Bus & Mixing

**Location:** Events/AudioBus.cs , Core/AudioManager.Buses.cs

Buses provide hierarchical mixing, routing, and effects control.

### 6.1 AudioBus Structure

Buses are ScriptableObjects forming a parent-child hierarchy.

## Properties

Property	Description
<code>busName</code>	Unique bus identifier
<code>parentBus</code>	Parent bus in hierarchy (null = root)
<code>mixerGroup</code>	Unity AudioMixerGroup for routing
<code>volumeDb</code>	Volume in decibels (-80 to +20)
<code>volumeMultiplier</code>	Additional volume multiplier (0-1)
<code>mute</code>	Mute this bus
<code>solo</code>	Solo this bus (mutes all others)
<code>sends</code>	List of effect sends
<code>enableDucking</code>	Enable ducking behavior
<code>duckingTargets</code>	Buses to duck when triggered
<code>duckingAttack</code>	Duck attack time (0-1s)
<code>duckingRelease</code>	Duck release time (0-2s)

## 6.2 Bus Hierarchy Example

```

Master Bus (0 dB)
├── SFX Bus (-3 dB)
│   ├── Weapon Bus (-6 dB)
│   ├── Footstep Bus (-12 dB)
│   └── UI Bus (0 dB)
├── Music Bus (-6 dB)
│   ├── Music_Intro Bus (0 dB)
│   └── Music_Loop Bus (0 dB)
└── Ambience Bus (-9 dB)

```

## 6.3 Volume Calculation

Final volume is multiplicative through the hierarchy:

```
finalVolume = busVolume * parentVolume * duckingValue * soloMute
```

Example:

```
Weapon Bus = -6 dB (0.501)
SFX Bus (parent) = -3 dB (0.707)
Master Bus (grandparent) = 0 dB (1.0)
Final = 0.501 * 0.707 * 1.0 = 0.354 (approximately -9 dB)
```

## 6.4 Ducking System

Ducking temporarily lowers the volume of target buses.

### Setup Example

```
// Configure in Inspector:
Dialogue_Bus:
  Enable Ducking: true
  Ducking Attack: 0.05
  Ducking Release: 0.2
  Ducking Targets:
    - Music_Bus: -6 dB
    - SFX_Bus: -3 dB
```

### Trigger Ducking

```
// Automatically via event action
AudioEvent dialogueEvent → TriggerDucking on Dialogue_Bus

// Manually from code
AudioBus dialogueBus = AudioManager.Instance.GetBus("Dialogue_Bus");
dialogueBus.TriggerDucking(holdDuration: 2f);
```

## 6.5 Effect Sends

Effect sends route audio to auxiliary effects buses.

```
[System.Serializable]
public class EffectSend
{
    public string sendName = "Reverb";
    public float sendLevel = 0.5f; // 0-1 send amount
    public bool prePost = false;    // false = post-fader, true = pre-fader
}
```

## 6.6 Bus API

```
// Get bus
AudioBus sfxBus = AudioManager.Instance.GetBus("SFX");

// Set volume
sfxBus.SetVolume(-6f); // dB

// Transition volume
AudioManager.Instance.SetBusVolume("SFX", -12f, transitionTime: 2f);

// Mute/Solo
sfxBus.SetMute(true);
sfxBus.SetSolo(true);

// Get all buses
List<AudioBus> allBuses = AudioManager.Instance.GetAllBuses();
```

# 7. State System

**Location:** Events/AudioState.cs , Core/AudioManager.States.cs

States represent game states that modify audio properties.

## 7.1 State Groups

States are organized into mutually exclusive groups:

- **State Group:** "GameplayState"
  - States: "Menu", "Playing", "Paused", "GameOver"

- **State Group:** "LocationState"
  - States: "Indoors", "Outdoors", "Underwater"
- **State Group:** "CombatState"
  - States: "Peace", "Alert", "Combat"

## 7.2 AudioState Structure

### Properties

Property	Description
stateName	Unique state identifier
stateGroup	State group this belongs to
busVolumes	Bus volume changes
switchValues	Switch changes
rtpcValues	RTPC changes
effectProperties	Effect parameter changes

## 7.3 State Property Types

### BusVolumeProperty

```
[System.Serializable]
public class BusVolumeProperty
{
    public AudioBus bus;
    public float volumeDb = 0f; // -80 to +20
}
```

## SwitchProperty

```
[System.Serializable]
public class SwitchProperty
{
    public string switchGroup;
    public string switchValue;
}
```

## RTPCProperty

```
[System.Serializable]
public class RTPCProperty
{
    public string parameterName;
    public float value = 0f;
}
```

## EffectProperty

```
[System.Serializable]
public class EffectProperty
{
    public AudioBus bus;
    public string propertyName; // e.g., "LowpassCutoff"
    public float value = 0f;
}
```



## 7.4 Using States

### Create State

```
// Assets > Create > Audio System > Audio State

State Name: "Underwater"
State Group: "LocationState"

Bus Volumes:
  - Music_Bus: -12 dB
  - SFX_Bus: -6 dB

RTPC Values:
  - "UnderwaterMix": 1.0

Effect Properties:
  - Master_Bus > "LowpassCutoff": 800
```

### Activate State

```
// Immediate transition
AudioManager.Instance.SetState("Underwater");

// Smooth transition over 2 seconds
AudioManager.Instance.SetState("Underwater", transitionTime: 2f);

// Set state in specific group
AudioManager.Instance.SetStateInGroup("LocationState", "Underwater", 1f);
```

### Query State

```
// Get active state in a group
AudioState currentLocation = AudioManager.Instance.GetActiveState("LocationState");

// Get active state name
string currentLocationName = AudioManager.Instance.GetActiveStateName("LocationState");

// Get all active states
IReadOnlyDictionary<string, AudioState> allStates =
    AudioManager.Instance.GetAllActiveStates();
```

## 7.5 State Transition Example

```
public class LocationManager : MonoBehaviour
{
    void OnEnterUnderwater()
    {
        // Smoothly transition to underwater state
        AudioManager.Instance.SetState("Underwater", 1f);

        // Set underwater switch for sound selection
        AudioManager.Instance.SetSwitch("Location", "Underwater");
    }

    void OnExitUnderwater()
    {
        AudioManager.Instance.SetState("Outdoors", 1f);
        AudioManager.Instance.SetSwitch("Location", "Outdoors");
    }
}
```

---

## 8. Voice Management

---

**Location:** Core/AudioManager.OcclusionPerf.cs

The system automatically manages voice allocation, virtualization, and performance.

### 8.1 Voice Lifecycle

1. Request → AudioManager.GetVoice()
2. Allocate → Dequeue from pool or create new
3. Configure → Set clip, position, bus, priority
4. Play → AudioSource.Play()
5. Monitor → Check completion, virtualization, occlusion
6. Cleanup → ReturnVoice() → Enqueue back to pool

### 8.2 Voice Virtualization

When voice count exceeds `maxRealVoices`, the system virtualizes low-priority voices.

## Virtualization Process

1. Sort real voices by importance (distance × volume × priority)
2. Pause least important voices
3. Track virtual playback time
4. When capacity available, un-virtualize highest importance voices

## Importance Calculation

```
float distanceFactor = 1.0 - (distance / maxDistance);
float volumeFactor = gainStack.GetFinalGain();
float priorityFactor = priority / 255f;
float importance = distanceFactor * volumeFactor * priorityFactor;
```

## 8.3 Voice LOD (Level of Detail)

LOD system adjusts voice behavior based on distance:

```
lodDistances = [10f, 25f, 50f, 100f]
```

Distance < 10m:	Full quality, real voice
10m - 25m:	Candidate for virtualization
25m - 50m:	High virtualization priority
50m - 100m:	Very high virtualization priority
Distance > 100m:	Auto-stop (not implemented yet)

## 8.4 Voice Statistics

```
AudioManager.AudioStatistics stats = AudioManager.Instance.GetStatistics();

Debug.Log($"Active Voices: {stats.activeVoices}/{stats.totalVoices}");
Debug.Log($"Available Voices: {stats.availableVoices}");
Debug.Log($"Active Loops: {stats.activeLoops}");
Debug.Log($"Registered Events: {stats.registeredContainers}");
```

## 8.5 Debug Visualization

```
List<AudioVoiceDebugInfo> debugInfo =  
    AudioManager.Instance.GetActiveVoicesDebug();  
  
foreach (var info in debugInfo)  
{  
    Debug.Log($"Voice {info.id}: " +  
        $"Event={info.eventName}, " +  
        $"Container={info.containerName}, " +  
        $"Virtual={info.isVirtual}, " +  
        $"Priority={info.priority}, " +  
        $"Distance={info.distance:F1}m, " +  
        $"Volume={info.volumeDb:F1}dB");  
}
```

---

## 9. Advanced Features

### 9.1 RTPC (Real-Time Parameter Control)

RTPCs allow dynamic audio control based on game parameters.

#### Common Use Cases

- Music intensity based on combat proximity
- Wind volume based on weather
- Engine pitch based on vehicle speed
- Reverb amount based on room size

## API

```
// Set RTPC value (0-1 normalized, or any float)
AudioManager.Instance.SetRTPC("MusicIntensity", 0.75f);

// Transition RTPC smoothly
AudioManager.Instance.TransitionRTPC("MusicIntensity", 1f, duration: 3f);

// Get current value
float intensity = AudioManager.Instance.GetRTPC("MusicIntensity");

// Get all RTPCs
IReadOnlyDictionary<string, float> allRTPCs =
    AudioManager.Instance.GetAllRTPCs();
```

## RTPC Listeners

Components can listen for RTPC changes:

```
public class CustomAudioComponent : MonoBehaviour, AudioManager.IRTPCListener
{
    void Start()
    {
        AudioManager.Instance.RegisterRTPCListener(this);
    }

    void OnDestroy()
    {
        AudioManager.Instance.UnregisterRTPCListener(this);
    }

    public void OnRTPCChanged(string parameter, float value)
    {
        if (parameter == "MusicIntensity")
        {
            // React to RTPC change
            UpdateMusicIntensity(value);
        }
    }
}
```

## 9.2 Switch System

Switches select different audio based on game state.

## API

```
// Set switch value
AudioManager.Instance.SetSwitch("Surface_Type", "Metal");
AudioManager.Instance.SetSwitch("Weather", "Rainy");
AudioManager.Instance.SetSwitch("Character", "Player1");

// Get switch value
string surface = AudioManager.Instance.GetSwitch("Surface_Type");

// Get all switches
IReadOnlyDictionary<string, string> allSwitches =
    AudioManager.Instance.GetAllSwitches();
```

## 9.3 Occlusion System

Automatic raycast-based occlusion with low-pass filtering.

### How It Works

1. Every `occlusionUpdateInterval` seconds (default 0.2s)
2. Raycast from listener to each active 3D voice
3. If occluded by `occlusionMask` geometry:
  - Reduce voice volume to 30%
  - Enable low-pass filter (cutoff ~500-800 Hz)
4. Smoothly interpolate changes over time

### Configuration

```
// In AudioManager Inspector:
enableOcclusion = true
occlusionMask = Everything except "IgnoreOcclusion" layer
occlusionUpdateInterval = 0.2f
```

### Performance Note

Occlusion is relatively expensive (raycasts per frame). Limit to important 3D sounds: - Use higher update interval (0.2-0.5s) - Exclude small/unimportant sounds - Use layers to exclude irrelevant geometry

## 9.4 Crossfading

**Location:** `Core/AudioManager.Crossfade.cs`

Smooth transitions between containers.

### API

```
AudioManager.Instance.CrossFade(  
    from: oldMusicContainer,  
    to: newMusicContainer,  
    duration: 2f,  
    type: AudioEvent.CrossfadeType.EqualPower  
);
```

### Crossfade Types

- **Linear:** Linear volume ramp (may have volume dip in middle)
- **EqualPower:** Sine-based curve (constant perceived loudness)

---

## 10. Music System

---

**Note:** The SFX System focuses on sound effects, ambience, and UI audio. For advanced interactive music features (beat synchronization, BPM detection, musical transitions), see the standalone **MusicManager** system located in `Assets/Scripts/MusicSystem/`.

The MusicManager provides: - Beat/measure synchronization with callbacks - Advanced BPM detection and analysis - Interactive music containers - Musical transitions with timing quantization - Stingers, playlists, and multi-track support

For details, refer to the MusicSystem documentation.

---

## 11. Performance & Optimization

---

### 11.1 Voice Pooling

- All voices are pre-allocated at initialization
- No runtime `new` allocations
- Voices recycled via `GetVoice()` / `ReturnVoice()`

### 11.2 Voice Management Settings

Recommended settings by project size:

Project Size	maxRealVoices	maxVirtualVoices	voiceUpdateInterval
Small (Mobile)	16	32	0.2s
Medium (PC Indie)	32	64	0.1s
Large (AAA)	64	128	0.05s

### 11.3 Optimization Tips

#### 1. Use Events, Not Direct Calls

```
// Good
explosionEvent.Post(gameObject, transform.position);

// Bad (bypasses pooling, priority, etc.)
AudioSource.PlayClipAtPoint(clip, position);
```

#### 2. Set Appropriate Priorities

- Critical: UI, dialogue
- High: Player actions, important feedback
- Medium: Gameplay sounds
- Low: Ambient, distant sounds

#### 3. Limit Instance Counts



```
// In AudioEvent:
maxInstances = 5 // Prevent 100 simultaneous explosions
```

#### 4. Use 2D Audio When Possible

- 2D audio is cheaper than 3D
- Only use 3D for positioned sounds

#### 5. Optimize Occlusion

- Higher update interval (0.2-0.5s)
- Exclude small sounds from occlusion
- Use simplified occlusion geometry layer

#### 6. Container Selection

- RoutingContainer: Cheapest, simple playback
- RandomContainer: Moderate cost, history tracking
- BlendContainer: Most expensive, multiple simultaneous voices

## 11.4 Memory Considerations

- **ScriptableObjects**: Shared data, minimal memory overhead
- **Voice Pool**:  $(\text{maxRealVoices} + \text{maxVirtualVoices}) \times \sim 1 \text{ KB per voice}$
- **AudioClips**: Loaded via Resources system or AssetBundles

## 11.5 Profiling

Use Unity Profiler to monitor: - `AudioManager.VoiceManagementUpdate()` CPU cost - `AudioManager.OcclusionUpdate()` CPU cost (if enabled) - Active voice count vs. limit - Voice virtualization frequency

---

## 12. API Reference

---

### 12.1 AudioManager Core API

```
// Singleton access
AudioManager.Instance

// Voice management
AudioVoice GetVoice()
void ReturnVoice(AudioVoice voice)

// Global controls
void SetMasterVolume(float volume)      // 0-1
float GetMasterVolume()
void MuteAll(bool mute)
void StopAllSounds()
void PauseAll()
void UnpauseAll()

// Statistics
AudioStatistics GetStatistics()
List<AudioVoiceDebugInfo> GetActiveVoicesDebug()
int GetVirtualVoiceCount()
```

### 12.2 Switch & RTPC API

```
// Switches
void SetSwitch(string group, string value)
string GetSwitch(string group)
IReadOnlyDictionary<string, string> GetAllSwitches()

// RTPCs
void SetRTPC(string parameterName, float value)
float GetRTPC(string parameterName)
void TransitionRTPC(string parameterName, float targetValue, float duration)
IReadOnlyDictionary<string, float> GetAllRTPCs()
void RegisterRTPCListener(IRTPCListener listener)
void UnregisterRTPCListener(IRTPCListener listener)
```

## 12.3 Bus API

```
// Bus access
AudioBus GetBus(string busName)
List<AudioBus> GetAllBuses()

// Volume control
void SetBusVolume(string busName, float volumeDb, float transitionTime = 0f)
void TransitionBusVolume(AudioBus bus, float targetVolumeDb, float transitionTime)

// Effect control
void SetBusEffectProperty(AudioBus bus, string propertyName, float value, float transitionTime = 0f)
```

## 12.4 State API

```
// State activation
void SetState(string stateName, float transitionTime = 0.5f)
void SetStateInGroup(string stateGroup, string stateName, float transitionTime = 0.5f)

// State queries
AudioState GetActiveState(string stateGroup)
string GetActiveStateName(string stateGroup)
IReadOnlyDictionary<string, AudioState> GetAllActiveStates()
```

## 12.5 Container API

```
// All containers
AudioVoice Play(Vector3 position = default, GameObject parent = null)
void Stop()
void StopImmediate()
bool HasActiveVoices()
int GetActiveVoiceCount()

// Extension methods
AudioVoice PlayAtPosition(Vector3 position)
AudioVoice PlayAttached(GameObject target)
AudioVoice PlayWithVolume(float volumeScale, Vector3 position = default)
```

## 12.6 Event API

```
// Post event
AudioHandle Post(GameObject source = null, Vector3 position = default)

// Handle control
bool isPlaying { get; }
float time { get; }
float duration { get; }
void SetVolume(float linear)
void SetPitch(float semitones)
void Stop(float fadeTime = 0.1f)
void Pause()
void Resume()
void Dispose()

// Handle callbacks
event Action OnStarted
event Action OnLoop
event Action OnFinished
```

## 13. Best Practices

### 13.1 Project Organization

```
Assets/Audio/
├── Resources/
│   ├── Audio/
│   │   ├── Events/           (AudioEvent assets loaded at runtime)
│   │   └── States/          (AudioState assets loaded at runtime)
│   ├── Containers/         (AudioContainer assets, referenced by events)
│   ├── Buses/              (AudioBus assets)
│   ├── AudioClips/         (Actual audio files)
│   │   ├── SFX/
│   │   ├── Music/
│   │   └── Ambience/
│   └── Mixers/              (Unity AudioMixer assets)
```



## 13.2 Naming Conventions

```
Events:      Play_Footstep, Play_Explosion, Stop_Music
Containers:  Footstep_Grass_RC, Explosion_Large_RC, Music_Combat_BC
Buses:      SFX_Bus, Music_Bus, Ambience_Bus
States:      State_Underwater, State_Combat, State_Paused
```

**Suffixes:** - `_RC` : RoutingContainer - `_RnC` : RandomContainer - `_SC` : SequenceContainer - `_SwC` : SwitchContainer - `_BC` : BlendContainer

## 13.3 Event Design

### 1. One Event Per Game Action

-  Good: `Play_Footstep` event
-  Bad: Play footstep directly from code

### 2. Use Actions for Complex Behaviors

```
Event: "Dialogue_Start"
Action 1: Play → Dialogue_Container
Action 2: TriggerDucking → Music_Bus
Action 3: SetRTPC → "DialogueActive" = 1.0
```

### 3. Limit Instances

- Set `maxInstances` for events that may trigger rapidly

## 13.4 Bus Hierarchy Design

```
Master (-3 dB)          // Headroom for mixing
├── SFX (0 dB)
│   ├── Weapon (-6 dB)    // Louder than footsteps
│   ├── Footstep (-12 dB) // Subtle
│   └── UI (0 dB)         // Always audible
├── Music (-6 dB)         // Background
└── Dialogue (0 dB)       // Priority
    ├── VO_Player (0 dB)
    └── VO_NPC (-3 dB)
```

**Tips:** - Leave ~3-6 dB headroom on master - Group related sounds (all weapons under Weapon bus) - Dialogue should typically be loudest and unduckable

## 13.5 State Management

### 1. Use State Groups for Independent Systems

```
GameState: Menu, Playing, Paused
LocationState: Indoors, Outdoors, Underwater
CombatState: Peace, Alert, Combat
```

### 2. States vs. Switches

- **States:** Global game states affecting multiple systems
- **Switches:** Local per-sound variations

### 3. Transition Times

- Menu → Gameplay: 1-2s
- Combat transitions: 0.5-1s
- Location changes: 2-3s
- Pause/Unpause: 0.1-0.3s

## 13.6 Performance Guidelines

### 1. Voice Limits

- PC: 32-64 real voices
- Mobile: 16-24 real voices
- Consoles: 32-48 real voices

### 2. Occlusion

- Only enable for important 3D sounds
- Use 0.2-0.5s update interval
- Create simplified "OcclusionGeometry" layer

### 3. RTPC Updates

- Don't update every frame unless necessary
- Use `TransitionRTPC()` for smooth changes
- Batch RTPC updates in `FixedUpdate` or slower

## 14. Troubleshooting

---

### 14.1 Common Issues

#### "AudioManager not found in scene"

**Cause:** No AudioManager GameObject exists.

**Solution:** 1. Create empty GameObject 2. Add AudioManager component 3. Configure inspector properties 4. AudioManager persists via `DontDestroyOnLoad()`

#### "No available audio voices"

**Cause:** Voice pool exhausted.

**Solution:** 1. Increase `maxRealVoices` and `maxVirtualVoices` 2. Set `maxInstances` on frequently-triggered events 3. Check for voice leaks (voices not returning to pool)

#### "Event X has no audio clips"

**Cause:** Container referenced by event has no clips assigned.

**Solution:** 1. Open the container asset 2. Assign AudioClips in the `audioClips` list 3. Save the asset

#### "Container not playing at correct position"

**Cause:** Container `is3D` not enabled, or parent GameObject incorrect.

**Solution:**

```
// Ensure container has is3D = true in Inspector
// Pass correct position and parent:
container.Play(transform.position, gameObject);
```

#### "Occlusion not working"

**Cause:** Occlusion disabled or incorrect layer mask.

**Solution:** 1. Check `AudioManager.enableOcclusion = true` 2. Verify `occlusionMask` includes geometry layers 3. Ensure AudioListener exists in scene 4. Check that sounds are 3D ( `spatialBlend = 1` )

## "Ducking not triggering"

**Cause:** Ducking not configured or targets incorrect.

**Solution:** 1. Check AudioBus `enableDucking = true` 2. Verify `duckingTargets` list has target buses 3. Ensure `TriggerDucking` action exists in event 4. Check bus has voices assigned to it

## 14.2 Debug Techniques

### Log All Active Voices

```
void LogActiveVoices()
{
    var voices = AudioManager.Instance.GetActiveVoicesDebug();
    foreach (var v in voices)
    {
        Debug.Log($"{v.eventName} | {v.containerName} | " +
            $"Virtual: {v.isVirtual} | Vol: {v.volumeDb:F1} dB");
    }
}
```

### Monitor Voice Count

```
void OnGUI()
{
    var stats = AudioManager.Instance.GetStatistics();
    GUI.Label(new Rect(10, 10, 200, 20),
        $"Voices: {stats.activeVoices}/{stats.totalVoices}");
}
```



## Visualize Occlusion

```
void OnDrawGizmos()
{
    var listener = ListenerUtil.Get();
    if (listener == null) return;

    var voices = AudioManager.Instance.GetActiveVoicesDebug();
    foreach (var v in voices)
    {
        if (!v.is3D) continue;

        Gizmos.color = v.isVirtual ? Color.red : Color.green;
        Gizmos.DrawLine(listener.position, v.position);
        Gizmos.DrawWireSphere(v.position, 0.5f);
    }
}
```

## 14.3 Performance Debugging

### Profile Voice Management

```
void Update()
{
    // Add [Conditional("DEVELOPMENT_BUILD")] in production
    UnityEngine.Profiling.Profiler.BeginSample("AudioManager.VoiceManagement");
    // Voice management logic here
    UnityEngine.Profiling.Profiler.EndSample();
}
```

### Check Voice Allocation Rate

```
private int lastVoiceCount = 0;

void Update()
{
    int current = AudioManager.Instance.GetStatistics().activeVoices;
    if (current != lastVoiceCount)
    {
        Debug.Log($"Voice count changed: {lastVoiceCount} → {current}");
        lastVoiceCount = current;
    }
}
```

---

## Appendix A: Quick Start Checklist

---

### Setup (5 minutes)

- ☐ Create GameObject named "AudioManager"
- ☐ Add AudioManager component
- ☐ Create Resources/Audio/Events folder
- ☐ Create Resources/Audio/States folder
- ☐ Create folder for containers and buses

### First Sound (10 minutes)

- ☐ Create RoutingContainer: `Assets > Create > Audio System > Routing Container`
- ☐ Assign AudioClip(s) to container
- ☐ Create AudioEvent: `Assets > Create > Audio System > Audio Event`
- ☐ Add "Play" action to event, assign container
- ☐ Move event to `Resources/Audio/Events/`
- ☐ Reference event in MonoBehaviour script
- ☐ Call `myEvent.Post(gameObject, transform.position)`
- ☐ Test in Play mode

### First Bus (5 minutes)

- ☐ Create AudioBus: `Assets > Create > Audio System > Audio Bus`
- ☐ Set bus name (e.g., "SFX\_Bus")
- ☐ In AudioEvent, set action's `targetBus` to this bus
- ☐ Test volume control: `AudioManager.Instance.SetBusVolume("SFX_Bus", -6f)`

### First State (10 minutes)

- ☐ Create AudioState: `Assets > Create > Audio System > Audio State`
- ☐ Set `stateName` and `stateGroup`
- ☐ Add BusVolumeProperty entry, select bus, set volume

- [ ] Move state to `Resources/Audio/States/`
  - [ ] Trigger state: `AudioManager.Instance.SetState("MyState", 1f)`
-

## Appendix B: Example Scenarios

---

### Scenario 1: Footstep System with Surface Types

```
// 1. Create containers for each surface
//   - Footsteps_Grass_RC (RandomContainer with grass clips)
//   - Footsteps_Metal_RC (RandomContainer with metal clips)
//   - Footsteps_Wood_RC (RandomContainer with wood clips)

// 2. Create SwitchContainer
//   - SwitchGroupName: "Surface_Type"
//   - Switch Entries:
//       "Grass" → Footsteps_Grass_RC
//       "Metal" → Footsteps_Metal_RC
//       "Wood" → Footsteps_Wood_RC

// 3. Create AudioEvent "Play_Footstep"
//   - Action: Play → FootstepSwitchContainer → SFX_Bus
//   - Priority: Low
//   - MaxInstances: 4

// 4. Script:
public class PlayerFootsteps : MonoBehaviour
{
    [SerializeField] private AudioEvent footstepEvent;

    void OnFootstep() // Called by animation event
    {
        // Set switch based on ground material
        string surface = DetectSurface();
        AudioManager.Instance.SetSwitch("Surface_Type", surface);

        // Play footstep
        footstepEvent.Post(gameObject, transform.position);
    }

    string DetectSurface()
    {
        if (Physics.Raycast(transform.position, Vector3.down, out RaycastHit hit, 2f))
        {
            if (hit.collider.CompareTag("Metal")) return "Metal";
            if (hit.collider.CompareTag("Wood")) return "Wood";
        }
        return "Grass";
    }
}
```

## **Scenario 2: Layered Music with Combat Intensity**

```

// 1. Create containers
//   - Music_Ambient_RC (ambient music loop)
//   - Music_Tension_RC (tense percussion loop)
//   - Music_Combat_RC (full combat music loop)

// 2. Create BlendContainer "Music_Dynamic_BC"
//   - Blend Parameter Name: "CombatIntensity"
//   - Blend Entries:
//       Music_Ambient_RC → Curve: (0,1) to (1,0)
//       Music_Tension_RC → Curve: (0,0) to (0.5,1) to (1,0)
//       Music_Combat_RC → Curve: (0,0) to (1,1)
//   - Loop: true

// 3. Create AudioEvent "Start_Music"
//   - Action: Play → Music_Dynamic_BC → Music_Bus

// 4. Script:
public class CombatMusicManager : MonoBehaviour
{
    [SerializeField] private AudioEvent startMusicEvent;
    [SerializeField] private float intensityTransitionSpeed = 0.5f;

    private float targetIntensity = 0f;
    private float currentIntensity = 0f;

    void Start()
    {
        startMusicEvent.Post();
        AudioManager.Instance.SetRTPC("CombatIntensity", 0f);
    }

    void Update()
    {
        // Smooth RTPC transition
        if (Mathf.Abs(currentIntensity - targetIntensity) > 0.01f)
        {
            currentIntensity = Mathf.MoveTowards(
                currentIntensity,
                targetIntensity,
                intensityTransitionSpeed * Time.deltaTime
            );
            AudioManager.Instance.SetRTPC("CombatIntensity", currentIntensity);
        }
    }

    public void OnEnemiesNearby(int count)
    {
        if (count == 0)
            targetIntensity = 0f;          // Ambient
        else if (count < 3)

```

```
        targetIntensity = 0.5f;    // Tension
    else
        targetIntensity = 1f;      // Combat
    }
}
```

## Scenario 3: Underwater State Transition

```
// 1. Create AudioStates
//   State "Normal":
//     - State Group: "Location"
//     - Bus Volumes: (default)
//
//   State "Underwater":
//     - State Group: "Location"
//     - Bus Volumes:
//       SFX_Bus: -6 dB
//       Music_Bus: -12 dB
//     - RTPC Values:
//       "LowpassFilter": 800
//
// 2. Create switch-based containers for underwater variants
//   - Switch Group: "Location"
//   - Switches: "Normal", "Underwater"

// 3. Script:
public class UnderwaterVolume : MonoBehaviour
{
    [SerializeField] private float transitionTime = 1f;

    void OnTriggerEnter(Collider other)
    {
        if (other.CompareTag("Player"))
        {
            // Smoothly transition to underwater state
            AudioManager.Instance.SetState("Underwater", transitionTime);
            AudioManager.Instance.SetSwitch("Location", "Underwater");
        }
    }

    void OnTriggerExit(Collider other)
    {
        if (other.CompareTag("Player"))
        {
            AudioManager.Instance.SetState("Normal", transitionTime);
            AudioManager.Instance.SetSwitch("Location", "Normal");
        }
    }
}
```



## Appendix C: Audio Conversion Formulas

---

### dB to Linear

```
float linear = Mathf.Pow(10f, dB / 20f);
```

Examples:

- 0 dB → 1.0
- 6 dB → 0.501
- 12 dB → 0.251
- 20 dB → 0.1
- 40 dB → 0.01
- 80 dB → 0.0001 (effective silence)

### Linear to dB

```
float dB = 20f * Mathf.Log10(linear);
```

Examples:

- 1.0 → 0 dB
- 0.5 → -6.02 dB
- 0.25 → -12.04 dB
- 0.1 → -20 dB
- 0.01 → -40 dB

### Cents to Pitch Ratio

```
float pitchRatio = Mathf.Pow(2f, cents / 1200f);
```

Examples:

- 0 cents → 1.0 (no change)
- 100 cents → 1.059 (+1 semitone)
- 700 cents → 1.498 (+7 semitones, perfect fifth)
- 1200 cents → 2.0 (+1 octave)
- 1200 cents → 0.5 (-1 octave)

## Appendix D: Glossary

Term	Definition
<b>AudioBus</b>	Hierarchical mixing group for volume control and routing
<b>AudioContainer</b>	ScriptableObject defining how audio clips are organized and played
<b>AudioEvent</b>	ScriptableObject defining a sequence of audio actions to execute
<b>AudioHandle</b>	Runtime handle for controlling a posted event
<b>AudioState</b>	ScriptableObject representing a game state with audio property changes
<b>AudioVoice</b>	Runtime instance of a playing sound
<b>Crossfade</b>	Smooth transition between two audio sources
<b>DSP</b>	Digital Signal Processing - sample-accurate audio timing
<b>Ducking</b>	Temporarily lowering volume of one bus when another plays
<b>GainStack</b>	Multiplicative volume layers (base × bus × occlusion × RTPC × ...)
<b>LOD</b>	Level of Detail - adjusting audio based on distance
<b>Occlusion</b>	Muffling audio when geometry blocks line-of-sight
<b>Priority</b>	Voice importance (0-255) for stealing decisions
<b>RTPC</b>	Real-Time Parameter Control - dynamic audio parameters
<b>State Group</b>	Collection of mutually exclusive states
<b>Switch</b>	Game state value used for container selection
<b>Virtualization</b>	Pausing low-priority voices to free resources
<b>Voice Stealing</b>	Stopping low-priority voices when pool exhausted

## Appendix E: Additional Resources

---

### Unity Documentation

- [AudioSource](#)
- [AudioMixer](#)
- [ScriptableObject](#)

### Recommended Reading

- *The Audio Programming Book* - Richard Boulanger & Victor Lazzarini
- *Game Audio Implementation* - Richard Stevens & Dave Raybould
- *Designing Sound* - Andy Farnell

### Community

- Unity Audio Forums: <https://forum.unity.com/forums/audio.18/>
- Game Audio Subreddit: <https://www.reddit.com/r/GameAudio/>

---

### End of Comprehensive Manual

For questions, issues, or feature requests, please contact the development team or open an issue in the project repository.