audiokinetic

Wwise Fundamentals

2017.2.4

Wwise Fundamentals

Wwise 2017.2.4 Revision 3176

Table of Contents

List of Tables

List of Examples

Chapter 1. Introducing Wwise

Introducing Wwise

Based on a profound understanding of the needs of both sound designers and audio programmers, Audiokinetic has created Wwise, an innovative solution dedicated to the art of audio design. Several years in the making, Wwise has been developed with the following premises in mind:

- Providing a complete authoring solution.
- Redefining the production workflow for audio and motion.
- Improving pipeline efficiency.
- Pushing the boundaries of gameplay immersion using audio and motion.

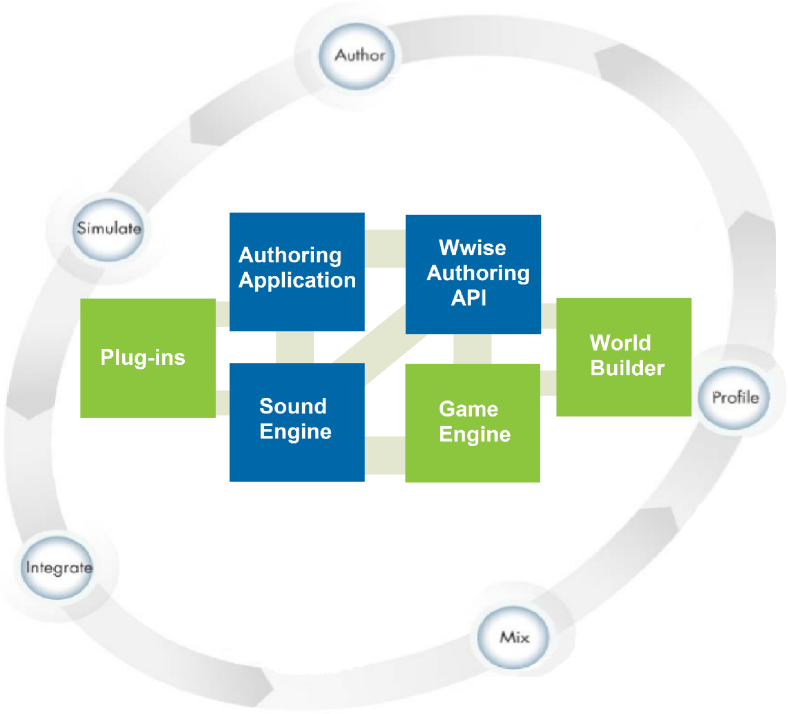
This powerful and comprehensive audio pipeline solution consists of the following:

- **A powerful authoring application**—a non-linear authoring tool for creating audio and motion asset structures, defining propagation, managing sound, music, and motion integration, profiling playback, and creating SoundBanks.
- **An innovative sound engine**— a sophisticated sound engine that manages audio and motion processing, performs a comprehensive set of diverse functions, and is highly optimized for each platform.
- **A game simulator**—a LUA script interpreter that reproduces exactly how sounds and motion will behave in the game, allowing you to validate specific behaviors and profile the performance of Wwise on each platform before the integration of Wwise into your game's sound engine.
- **A plug-in architecture**—a completely scalable plug-in architecture for quickly expanding the audio immersion in the game. Several plug-ins are available, including:
  - Source plug-ins for generating audio and motion, such as a tone generator.
  - Effect plug-ins for creating audio effects, such as a reverb.
- **An interface between Wwise and world builders** (**Wwise Authoring API**)—a unique plug-in interface with external game world builders or 3D applications that enables external applications to seamlessly communicate with Wwise. From the Wwise Authoring API, you can easily modify everything that can normally be modified using the Sound Engine API.

The Wwise Production Pipeline

At the foundation of Wwise is the production pipeline—a new and innovative way to work that tightly integrates the necessary tools allowing you to perform a variety of tasks in real time within the game itself.

- **Author**—build sound, motion, and music structures and define properties and behaviors.
- **Simulate**—validate artistic direction and simulate game play.
- **Integrate**—integrate early without additional programming.
- **Mix**—mix properties in game in real time.
- **Profile**—profile in real time to ensure game constraint compliance.



The Wwise Project

Wwise is a project-based system, which means that all the audio and motion information related to a particular game for each and every platform will be in one project.

Within this project, you can do any and all of the following:

- Manage the sound, voice, music, and motion assets in your game.
- Define object properties and playback behaviors.
- Create Events, both Action and Dialogue Events, that trigger audio and motion in game.
- Create prototypes and simulations.
- Troubleshoot and profile all aspects of the audio and motion in your project.

Projects also contain the SoundBanks that are generated for each platform and language version that you are creating for your game.

How Wwise Manages the Assets in Your Project

A typical game can have thousands of sound, music, and motion assets, so your Wwise project must be able to manage these assets efficiently and effectively, especially when you are creating different versions of the same game for each platform and language.

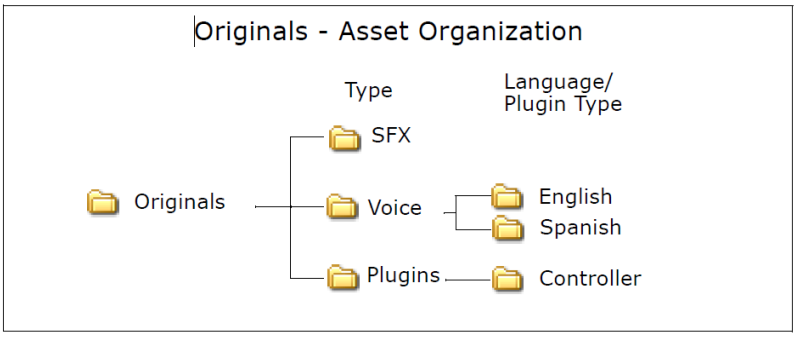
Originals

The first thing to understand is that Wwise is non-destructive, which means that you can edit the assets in your project without affecting the original files themselves. When you import a file into Wwise, a copy of the file is stored in the project's "Originals" folder.

Depending on the type of file you are importing, the file will be stored in one of the following folders:

- Plug-ins
- SFX
- Voice

If files are flagged as Voice or plug-in files, they are then further subdivided by language or plug-in type. The following illustration demonstrates how Wwise organizes the original assets imported into your project.



Platform Versions

From these Wwise "Originals", you can create the versions for each game platform. These platform versions are stored in the project's "cache" folder.

To help Wwise manage the contents of the cache folder more effectively, the converted assets are divided by:

- **Platform** (Windows®, Xbox One™, PlayStation®4, and so on)
- **Type** (Plug-ins, SFX, or Voice). If assets are flagged as Voice or plug-in files, they are further subdivided by:
  - **Language** (English, French, Spanish, and so on)
  - **Plug-in Type** (Controller)

Chapter 2. Integrating Audio in your Game

The Wwise Fundamental Approach

Before jumping into the code and using the Wwise SDK, you should understand the unique approach Wwise uses for building and integrating audio into your game. There are also a few concepts that you should be familiar with in order to work efficiently and get the most out of Wwise.

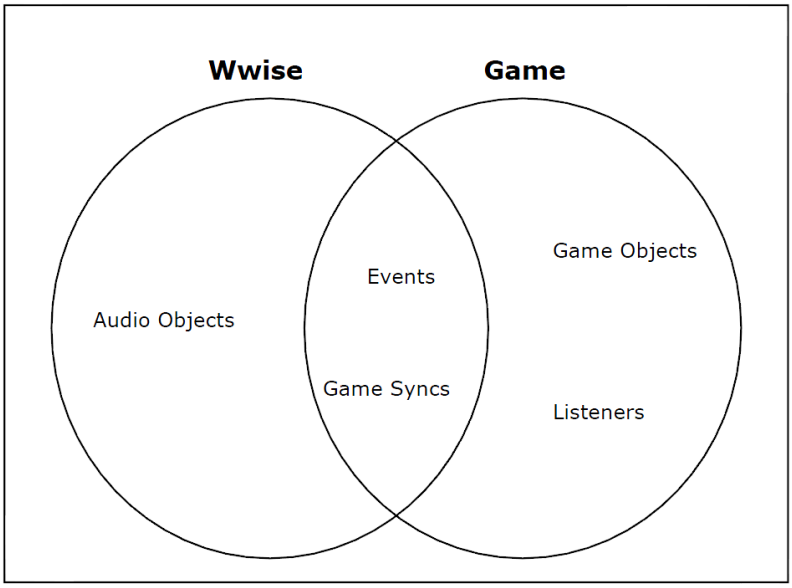
The Wwise approach to building and integrating audio in a game includes five main components:

- Audio Objects
- Events
- Game Syncs
- Game Objects
- Listeners

Each of the components will be discussed in further detail in the following sections, but before moving on, you should understand where each of these components fits in and how they relate to one another.

One of the goals of Wwise was to create a clear distinction between the tasks of the programmer and those of the designer. For example, the audio objects, which represent the individual sounds in your game are created and managed exclusively within the Wwise application by the sound designer. Game objects and listeners, on the other hand, which represent specific game elements that emit or receive audio, are created and managed within the game by the programmer. The final two components, Events and Game Syncs, are used to drive the audio in your game. These two components create the bridge between the audio assets and the game components and are therefore integral to both Wwise and the game.

The following illustration demonstrates where each of these components are created and managed.


Wwise
Game
Audio Objects
Events
Game Syncs
Game Objects
Listeners

Chapter 3. The Project Hierarchy

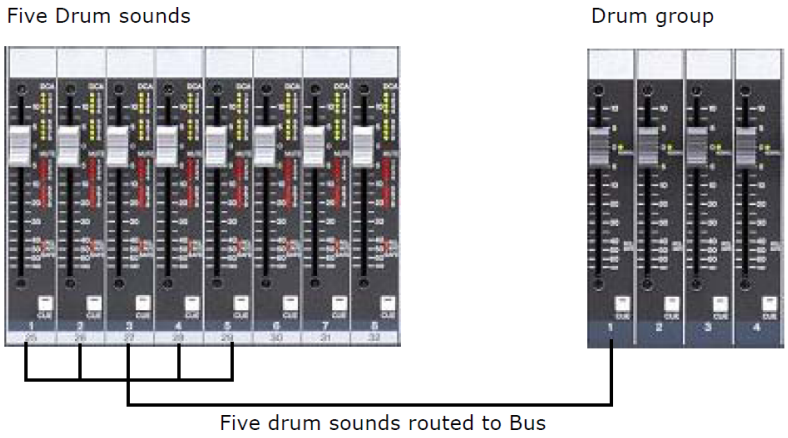
The Project Hierarchy

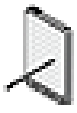
The assets that you import into your project are the foundation for your project hierarchy. The project hierarchy evolved from traditional mixing techniques where different instruments were routed to a bus, so that you could control their sound properties as a single mixed sound. For example, each of the hi-hat, ride, crash, bass drum, and snare sounds could be routed to a single bus so that you could control their volume and other parameters as if they were one entity.

In Wwise, a similar approach is used to organize and group the sounds, motion objects, and music in your project. By grouping sound, motion, and music objects in such a manner, you begin to build a hierarchical project structure that creates parent-child relationships between the various objects. This unique and efficient way to create and manage the audio and motion in your game gives you more control and flexibility to build a realistic and immersive environment for your game.

The Wwise project hierarchy consists of three distinct levels:

- **Actor-Mixer Hierarchy**—groups and organizes all the sound and motion assets in your project using a series of Wwise-specific objects.
- **Interactive Music Hierarchy**—groups and organizes all the music assets in your project using a series of Wwise-specific objects.
- **Master-Mixer Hierarchy**—defines the routing and output of the different sound, motion, and music structures using one or more output busses.





Wwise Project Hierarchies are Workgroup Ready

Working as part of a team is crucial in today's game development environment. Although only one Wwise project can be used per game, you can divide up a Wwise project's hierarchies into different work units so that different people can

work on the project concurrently. Work units are distinct XML files that contain information related to a particular section or element within your project. These work units can help you organize and manage the different elements within a project. If you are working as part of a team, these work units can also be managed by your source control system to make it easier for the different members of your team to work on the project concurrently.

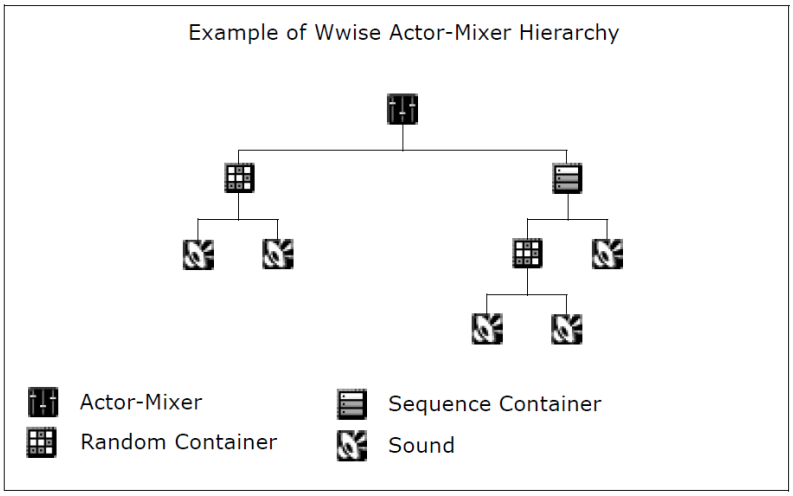
Understanding the Actor-Mixer Hierarchy

The Actor-Mixer Hierarchy groups and organizes all the sound and motion assets in your project. At the base of this hierarchy are all your individual sound and motion objects. You can define the properties and behaviors of these individual objects, but you can also take these objects and group them together so that you can define their properties and behaviors as a unit. To accommodate the complex nature of audio within a game, different types of objects can exist within the Wwise project hierarchy. Each object type has a set of properties, such as volume, pitch, and positioning, and a set of unique behaviors, such as random or sequence playback. By using different object types to group sounds within your project hierarchy, you can define specific playback behaviors of a group of sounds within the game. You can also define these properties and behaviors at different levels within the hierarchy to obtain different results.

Since motion is generally tied to audio in a game, Wwise uses the same principles and workflow for generating motion. This means that you can organize the motion assets for your game into hierarchies, and assign properties and behaviors in the same way as your audio assets.

You can use a combination of the following object types to group your assets and build a structure for your project:

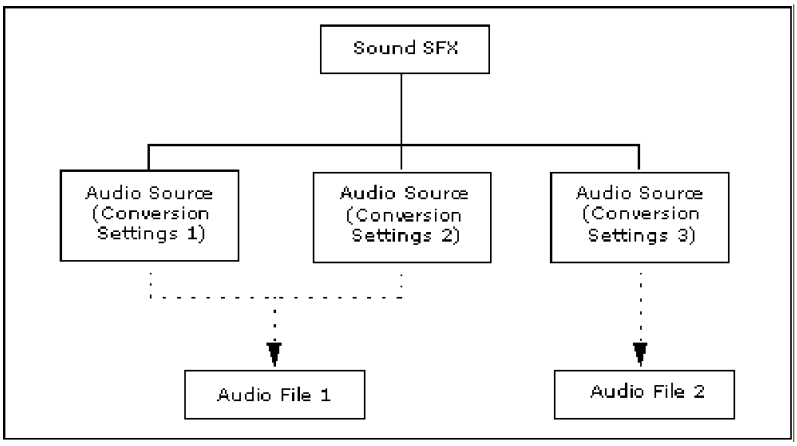
- Sound objects
- Motion FX objects
- Containers
- Actor-Mixers



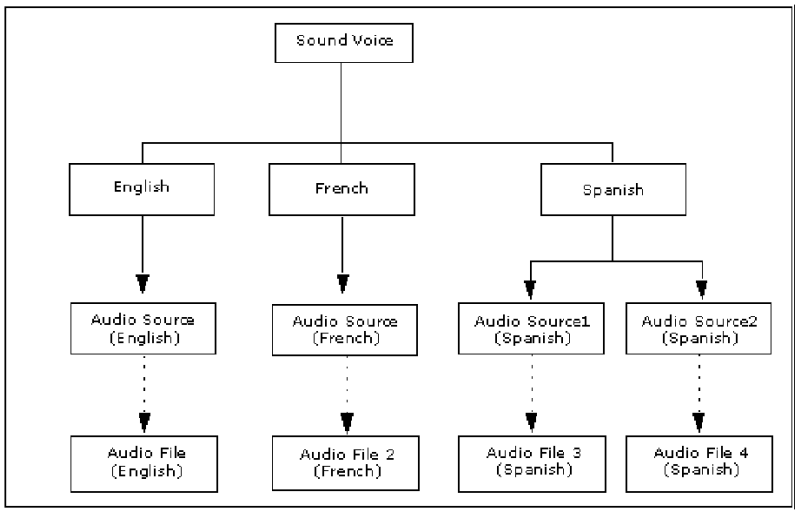
Audio Objects

The various voice and sfx assets in your game are represented in Wwise by special audio objects called sound objects. These sound objects contain sources that are linked to the original audio file.

The audio source is a separate layer between the imported audio file and the sound object. By adding an abstraction layer, you can have multiple sources and audio files all contained within the same sound object.



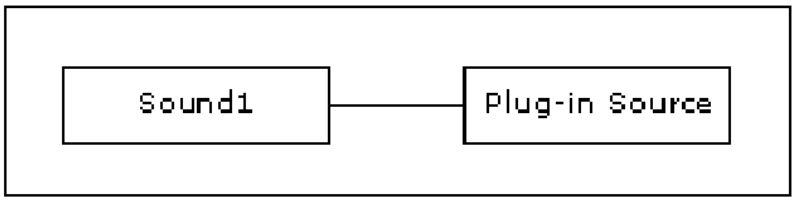
This not only makes it easy to test different conversion settings but also allows you to efficiently manage multi-language development.



Note: Wwise uses a similar method to manage the music and motion assets in your project.

Source Plug-ins

Sound objects not only support audio sources, but they also support plug-in sources.

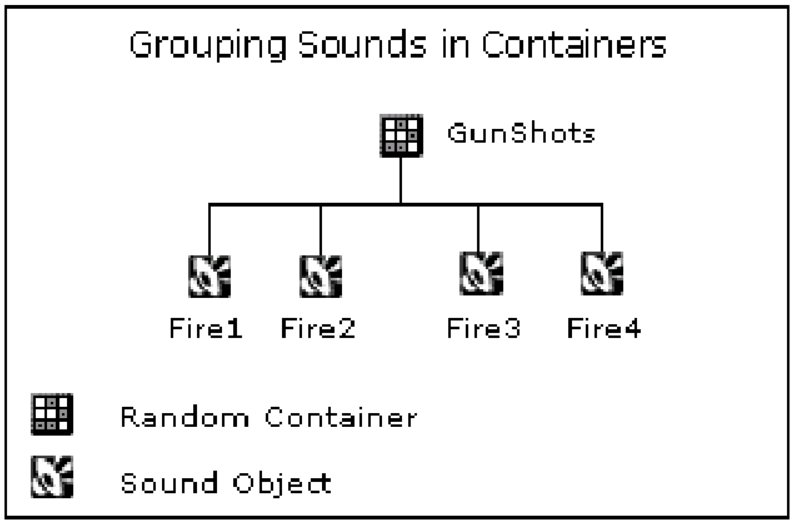


Wwise ships with a variety of source plug-ins, including a tone generator, silence, and audio input plug-in. In addition to being immediately useful in your production pipeline, the main purpose of these is to be used in conjunction with provided source code as a reference for programmers who are interested in building their own. With the creation and management of audio objects being done in Wwise by the sound designer, programmers are now free to develop a variety of source plug-ins, pushing the envelope in audio design and enhancing the overall experience of the game.

Building a Hierarchy of Audio Objects

These sound objects can be grouped together to create a hierarchical project structure. Audio properties and behaviors can be applied at different levels in the hierarchy to give you the control and flexibility you need to build a realistic and immersive game experience.

Containers are used to group the sound objects within your project. They are mainly used to play a group of objects according to a certain behavior, such as Random, Sequence, Switch, and so on. For example, you can group all the gun shot firing sounds into a Random Container so that a different sound will be played each time the gun is fired in game.



All these audio objects are routed through a hierarchy of busses where additional properties and effects can be applied at a global level.

Audio Objects - Roles and Responsibilities

The following table shows you which tasks related to audio objects are the responsibility of the sound designer and which ones are the responsibility of the programmer:

Table 3.1. Audio Objects - Roles and Responsibilities

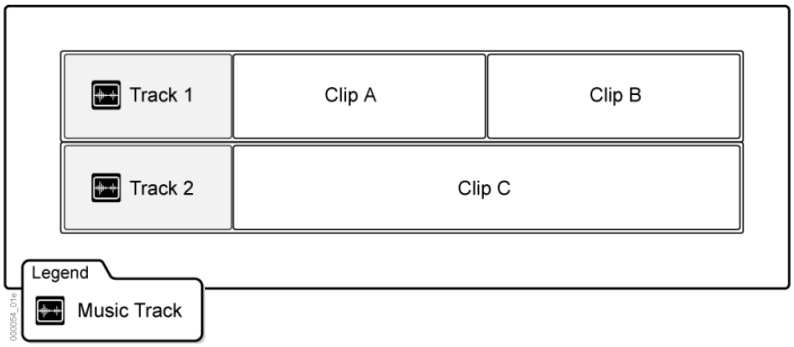
Tasks	Sound Designer (Wwise)	Programmer (Game Code/Tools)
Create sound objects for game audio assets	X	
Group objects and build project hierarchy	X	
Define sound properties and behaviors	X	
Route audio objects through busses	X	
Develop source plug-ins		X

Understanding the Interactive Music Hierarchy

Wwise offers you great flexibility when it comes to creating the interactive music for your project. There is an almost infinite number of ways to assemble interactive music objects into a game score. However, following some kind of consistent structure can make your workflow more efficient.

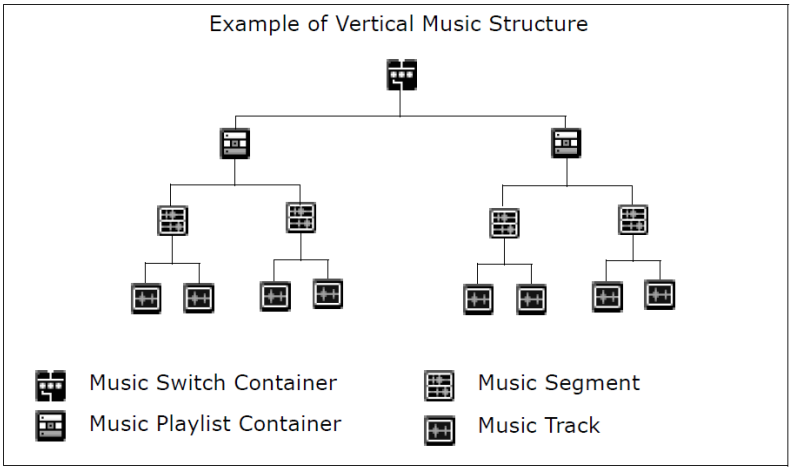
Two of the basic structures that can be applied to interactive music projects are as follows:

- **A vertical project structure** is one in which you re-sequence the game score by shuffling the tracks contained in music segments. This is similar to the track mixing used in music production. It can help you make a varied score out of long, multi-tracked segments.



- **A horizontal project structure** is one in which you vary the game score by changing which segments are played at any given time. To do this, you can arrange short discrete segments in the Interactive Music Hierarchy, much like you would arrange objects in the Actor-Mixer Hierarchy. In this way, you can make a compelling score from a selection of short music segments while minimizing console requirements.

Typically, you'll use a combination of both these structures to make efficient use of the resources you have available for your project. A good structure lets you show off your music, and make the most of your console resources.



Understanding the Master-Mixer Hierarchy

On top of the Actor-Mixer and Interactive Music hierarchies sits the Master-Mixer hierarchy. The Master-Mixer hierarchy is a separate hierarchical structure of busses that allows you to re-group and mix the many different sound, music, and motion structures within your project and prepare them for output. The Master-Mixer hierarchy is divided into two sections: one for sound and music, and one for motion. Each section consists of a top-level "Master Bus" and any number of child busses below it.

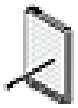
You can choose to route sound, music, and motion structures through these busses using the main categories within your game. For example, you may want to group all the different audio structures into the following four categories:

- Voice
- Ambience
- Sound Effects
- Music

These busses not only create the final level of control for the sound, music, and motion structures within your project but they can also determine which sounds are affected by environmental effects such as Reverb. Because they sit on top of your project hierarchy, you can use them to create the final mix for your game. Depending on the platform, certain effects, including environmental effects, may also be applied to the busses to create that immersive experience that your game requires.

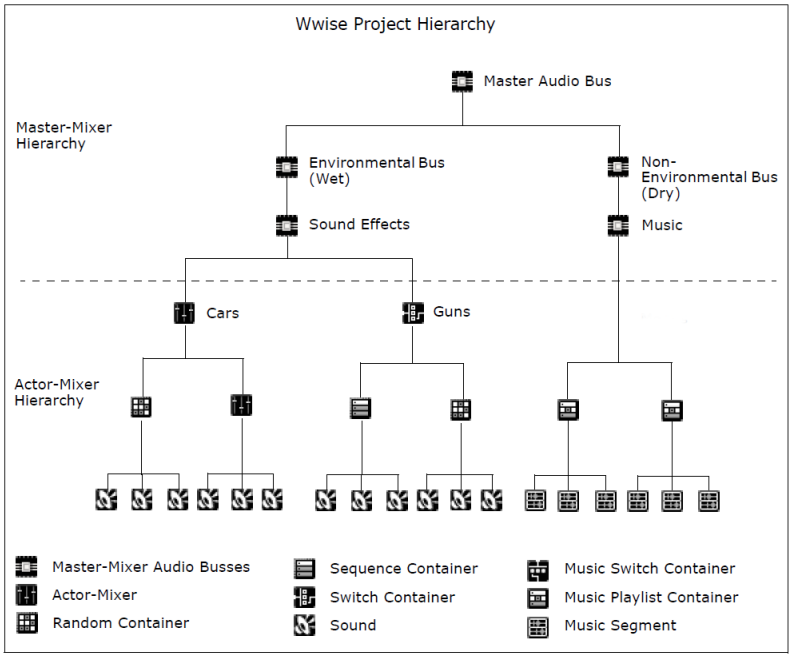
You can also use the Audio Bus structure to troubleshoot problems within your game. For example, you may want to solo specific voice, ambient sounds, or sound effects busses, to identify specific sounds or music.

The following illustration shows an example of a Master Audio Bus hierarchy that uses two preliminary busses to separate the environmental versus the non-environmental sounds and then uses several other audio busses to regroup some of the sound structures in the Actor-Mixer hierarchy and some of the music structures in the Interactive Music hierarchy.



Note

A similar hierarchy can be created at the same level under the Master Motion Bus for all the motion structures in your project.

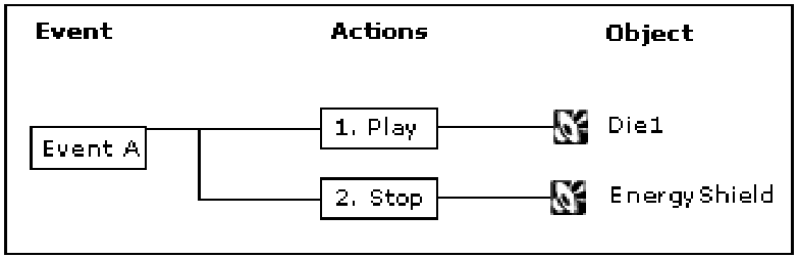


Chapter 4. Understanding Events

Understanding Events

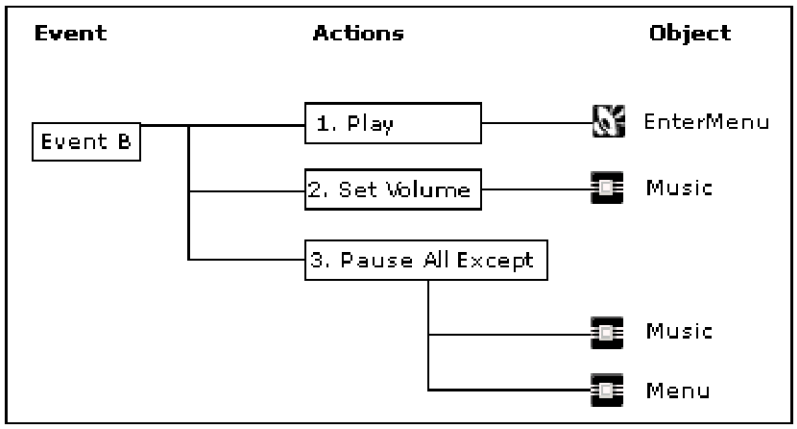
Wwise uses Events to drive the audio in your game. These Events apply actions to the different sound objects or object groups in your project hierarchy. The actions you select specify whether the Wwise objects will play, stop, pause, and so on. For example, let's say you are creating a first-person shooter game and you want to create an Event for when the player dies. This Event will play a special “Die” sound and will stop the “EnergyShield” sound that is currently playing.

The following illustration demonstrates how this Event would look in Wwise:



The sound designer can pick from a long list of action types to drive the audio in game, including Mute, Set Volume, Enable Effect Bypass, and so on. For example, let's say you created a second Event for when the player leaves the game to enter the menu. This Event will play the “Enter_Menu” sound, decrease the volume of the music bus by -10dB, and pause everything else.

The following illustration demonstrates how this Event would look in Wwise.



To accommodate as many situations as possible, there are two different types of Events:

- **“Action” Events**—these Events use one or more actions, such as play, stop, pause and so on, to drive the sound, music, and motion in game.

- **Dialogue Events**—these Events use a type of decision tree with States and Switches to dynamically determine what object is played.

After Events are created in Wwise, they can be integrated into the game engine so that they are called at the appropriate times in the game. Events can be created and integrated into the game engine early in the development process. You can continue to fine-tune the Event without having to re-integrate it into the game engine.

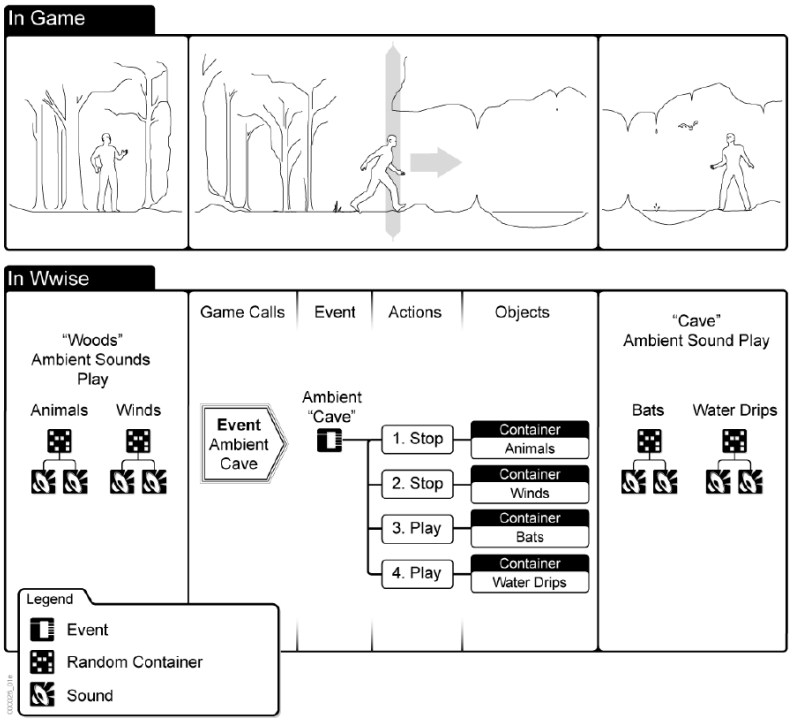
Action Events

To drive the sound, music, and motion in your game, Wwise uses “action” Events. These Events apply actions to the different structures within your project hierarchy. Each of these Events can contain one action or a series of actions. The actions you select will specify whether the Wwise objects will play, pause, stop, and so on.

Example 4.1. Using Action Events - Example

Let's say the character in your game must enter a cave to retrieve some hidden documents. When the character enters the cave from the woods, the ambient sounds in the game should change. To trigger this change, you must create an Event that will contain a series of actions that will stop the ambient “Woods” sounds and play the ambient “Cave” sounds. This Event will be integrated into the game engine and at the moment the character enters the cave, the game engine calls the specific Event that you created in Wwise.

The following illustration demonstrates how the game engine triggers an Event to change the ambient sounds playing in a game:

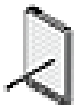


To deal with the transitions that occur between sound, music, or motion objects, each Event action also has a set of parameters that you can use to delay, or fade in and fade out incoming and outgoing objects.

Dialogue Events

To drive the dynamic dialogue in your game, Wwise uses the Dialogue Event, which is basically a set of rules or conditions that determines which piece of dialogue to play. The Dialogue Event allows you to re-create a variety of different scenarios, conditions or outcomes that exist in your game. To ensure that you cover every situation, Wwise also allows you to create default or fallback conditions.

All these conditions are defined using a series of State and Switch values. These State and Switch values are combined to create paths which define the particular conditions or outcomes in the game. Each path is then associated with a specific sound object in Wwise. As the game is played and Dialogue Events are called, the game verifies the existing conditions against those defined in the Dialogue Event. The condition or State/Switch path that matches the current situation in game determines which piece of dialogue is played.



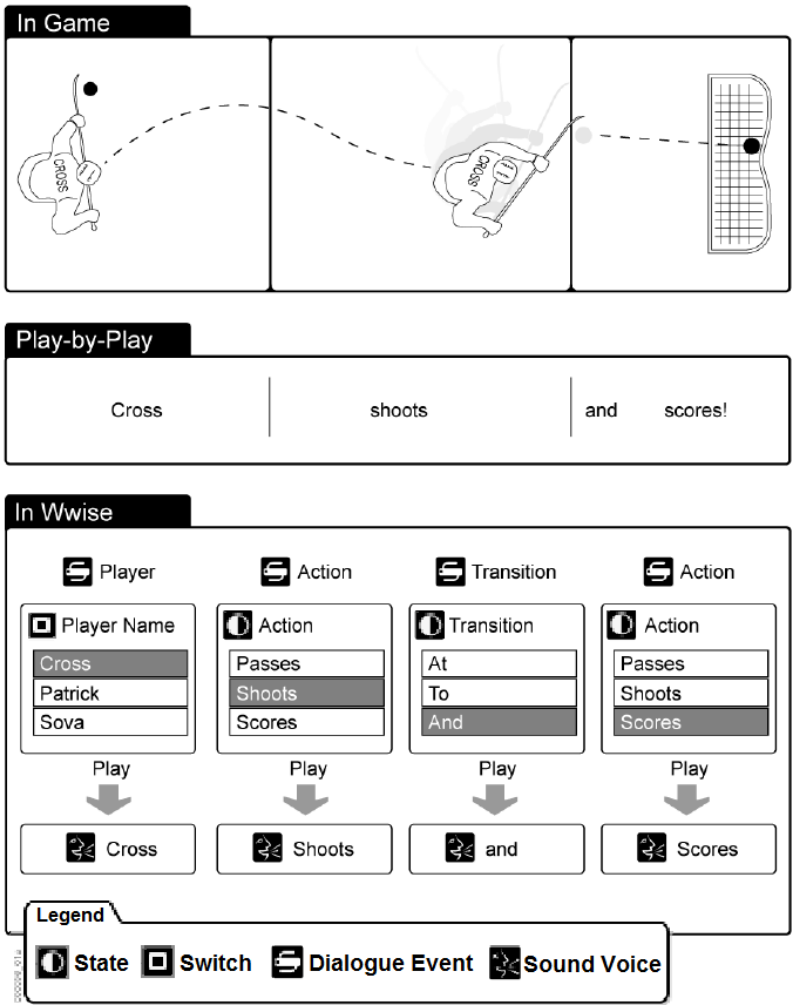
Note

Although Dialogue Events were initially created to handle game dialogue, they are not reserved explicitly for dialogue and can be used for a variety of other purposes in your game.

Example 4.2. Using Dialogue Events - Example

Let's say that you are creating a hockey game with a play-by-play commentary. When a player shoots and scores, you want the play-by-play commentary to correspond to the action in game. To set up the different possibilities and outcomes in Wwise, you will need to create Dialogue Events for Players, Actions, Transitions, and so on. Each of these Events will contain a set of corresponding State and Switch values that you have created for your game. You must create a State/Switch path that defines each condition or outcome and then assign an appropriate voice object to each State/Switch path. During gameplay, the game will match the current State/Switch values against the paths you defined in Wwise to determine which voice object to play.

The following illustration demonstrates how Dialogue Events created in Wwise can generate a play-by-play commentary that says “Cross shoots and scores!”:



Defining Event Scope

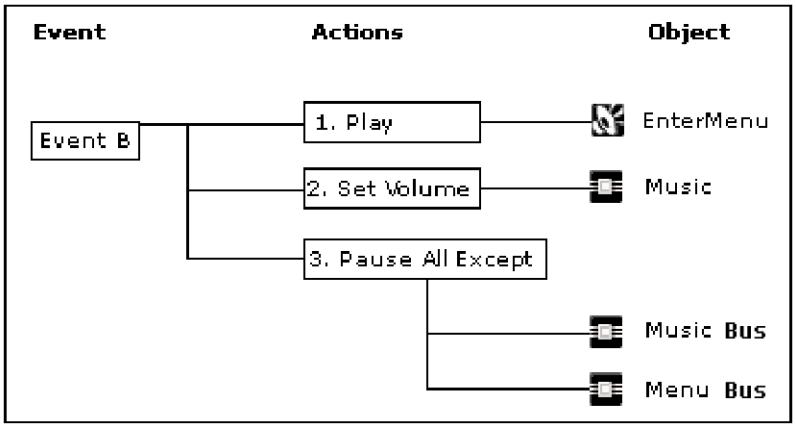
Every action within an Event has a corresponding scope setting. The scope determines whether the Event action is applied globally to all game objects or to the specific game object that triggered the Event. For some actions, the sound designer can choose the scope, and for other actions, the scope is predetermined.

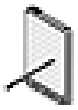
If we look again at EventB, for example, the scope of each Event action would be as follows:

Table 4.1. Defining Event Scope

Event Action	Scope	Comments
Play > Menu_Enter	Game Object	The scope is set to Game Object because play Events are always triggered by a single game object.
Set Volume > Music	Global	The scope is set to Global because the Set Volume action is applied to a bus, which, by its very nature, is global.
Pause All Except > Music	Global	The scope is automatically set to Global because the Pause All Except action is applied to the music bus, which, by its very nature, is global.

The following illustration demonstrates how this Event would look in Wwise.





Note

Scope is an important concept that applies to many elements in Wwise. Understanding the scope of each element will help you decide when to use each element in different situations.

Integrating Events into your Game

After creating the Events for your game, the sound designer can package them into SoundBanks. These SoundBanks are then loaded into your game, where the Events can be triggered by your game's code. For example, when the player is killed, you would play the special “Die” sound and stop the “EnergyShield” sound by triggering the corresponding event.

To integrate these Events into your game, the programmer must specify onto which game object the Event Actions will be performed. This is done by posting each Event. An Event should be posted by your game's code whenever you want the audio to change. You can post events using strings or IDs.

Benefits of Using Wwise Events

One main advantage to this method for triggering sound in your game is that it gives the sound designer additional control and flexibility without requiring any additional programming. All Events are created in Wwise by the sound designer and they are then integrated into the game by the programmer. Once Events are integrated in the game, the sound designer can continue working on them, changing or modifying the actions they contain, or the objects to which they refer. Since your game is still triggering the same Event, the changes made by the sound designer will take effect in the game without requiring extra work from the developer, and without recompiling the code.

Events - Roles and Responsibilities

The following table shows you which tasks related to Events are the responsibility of the sound designer and which ones are the responsibility of the programmer:

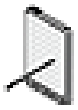
Table 4.2. Events - Roles and Responsibilities

Tasks	Sound Designer (Wwise)	Programmer (Game Code/Tools)
Creating Events	X	
Assigning Event Actions to audio objects	X	
Defining the scope of Event Actions	X	
Posting Events in game		X

Chapter 5. What are Game Objects?

What are Game Objects?

Game objects are the central concept in Wwise because every Event triggered in the sound engine is associated with a game object. A game object generally represents a particular object or element in your game that can emit a sound, including characters, weapons, ambient objects, such as torches, and so on. In some cases, however, you may want to assign game objects to different parts of an in-game element. For example, you can assign a different game object to different parts of a giant character so that the footstep sounds and the character's voice emanate from different locations within the 3D sound space.

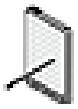


Note

If you are familiar with the Unreal game engine, game objects in Wwise are similar to Actors in Unreal.

For every game object, Wwise stores a variety of information that it will use to determine how each sound will be played back in game. Any of the following types of information may be associated with the game object:

- Property offset values of an audio object associated with the game object, including volume and pitch.
- 3D position and orientation.
- Game Sync information, including States, Switches, and RTPCs.
- Environmental effects.
- Obstruction and Occlusion.



Note

Unlike other properties, attenuation is applied on the audio object and not on the game object. This gives the sound designer more flexibility to control the attenuation for each sound individually. The 3D Game Object view in Wwise allows the sound designer to view the game objects to which sounds are associated, the position of the game objects in relation to the listener, along with the attenuation radius for each sound.

Registering Game Objects

Before you can use game objects, the programmer needs to register them in the game code. When you no longer need the game objects, you should un-register them, because the sound engine will continue to store their related information (3DPosition, RTPC, Switches, and so on) until the game object associated with these values is unregistered.

Scope - Game Objects vs Global

By using game objects, Wwise introduces the concept of scope, which was discussed briefly in the Events section. The scope determines the level at which properties and Events are applied to the sounds in your game. You now have the choice to apply these elements at the game object level or globally. The specific situation and/or action that is taking place in game, will determine the scope and ultimately the approach you take in Wwise.

For example, let's say you are creating a first-person shooter game. The main character in your game must navigate the city streets to capture the enemy's flags. As the character walks through the city, you will hear his footsteps. If want to change the properties or sounds associated with these footsteps, you will only want to apply these changes locally at the level of the game objects specifically related to the main character's feet. On the other hand, if your character submerges himself underwater, all the sounds that continue to play within the surrounding environment, such as explosions and vehicles, will need to be modified. In cases like these, you will want the changes to be made on a global scale.

Benefits of Using Game Objects

By using game objects, the management of audio has been simplified because programmers only have to keep track of game objects and not the individual sounds.

Once the game objects are created, programmers only need to post Events, set up the Game Syncs, including Switches, States, and RTPCs, and in-game environments. The specific details of which sound is played and how it will play are defined by the sound designer in Wwise. By using this approach, you can save a huge amount of time when dealing with the multitude of sounds associated with the various entities within your game.

Game Objects - Roles and Responsibilities

The following table shows you which tasks related to game objects are the responsibility of the sound designer and which ones are the responsibility of the programmer:

Table 5.1. Game Objects - Roles and Responsibilities

Tasks	Sound Designer (Wwise)	Programmer (Game Code/Tools)
Associate game objects to 3D objects in the game	X	X
Register/Unregister game objects		X
Update game object positioning information		X
Set the Attenuation for each audio object	X	
Defining Event scope	X	

Chapter 6. What are Game Syncs?

What are Game Syncs?

After the initial game design is complete, you can start looking at how you could use Wwise elements called Game Syncs to streamline and handle the changes and alternatives that are part of the game. You can define which of the five different kinds of Game Syncs you will need to achieve the best results possible to enhance the visuals of the game.

- **States**—a change that occurs in game that affects the properties of existing sounds, music, or motion on a global scale.
- **Switches**—a representation of the alternatives that exist for a particular game element that may require completely new sounds, music, or motion.
- **RTPCs**—properties that are mapped to variable Game Parameter values in such a way that changes to the Game Parameter values modify the properties themselves.
- **Triggers**—a response to a spontaneous occurrence in the game that launches a stinger, which is a brief musical phrase that is superimposed and mixed over the currently playing music.

When you are building your game project, you have to juggle quality, memory usage restrictions, and the time constraints that you face. Using Game Syncs strategically can simplify your work, economize on memory, and help to build a truly immersive game experience.

Understanding States

States are basically “mixer snapshots” or global offsets or adjustments to the game audio and motion properties that represent changes in the physical and environmental conditions in the game. Using States can streamline the way you design your audio and motion, and help you optimize your assets.

States as “mixer snapshots” allow for level of detail and control over the resulting sound output and can be combined with multiple States with expected results. When an object registers to multiple States, a single property can be affected by multiple value changes. In this scenario, each change of value is added up together. For example, when two States in two different State Groups have a volume change of -6 dB, and both become active simultaneously, the resulting volume will be -12 dB.

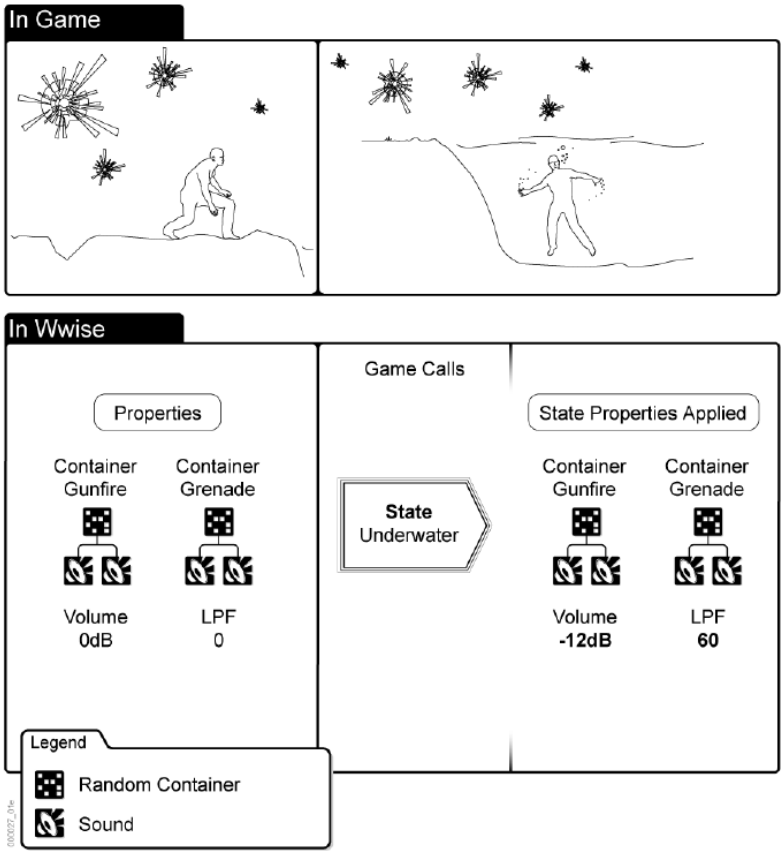
When you create and define these “mixer snapshots”, you are really creating different property sets for a sound, music, or motion object without adding to memory or disk space usage. These property sets define a set of rules that govern the playback of a sound during a given State (or

States). When you apply these property changes globally to many objects, you can quickly create realistic soundscapes that better represent the audio and enhance the game. By altering the properties of sounds, music, or motion already playing, you are able to re-use your assets and save valuable memory.

Example 6.1. Using States - Example

Let's say you want to simulate the sound treatment that occurs when a character goes underwater. In this case you could use a State to modify the volume and low pass filter for sounds that are already playing. These property changes should create the sound shift needed to recreate how gunfire or exploding grenades would sound when the character is under water.

The following illustration demonstrates how the properties for the volume and low pass filter for the gunfire and grenade sound objects are affected when the underwater State is called by the game.



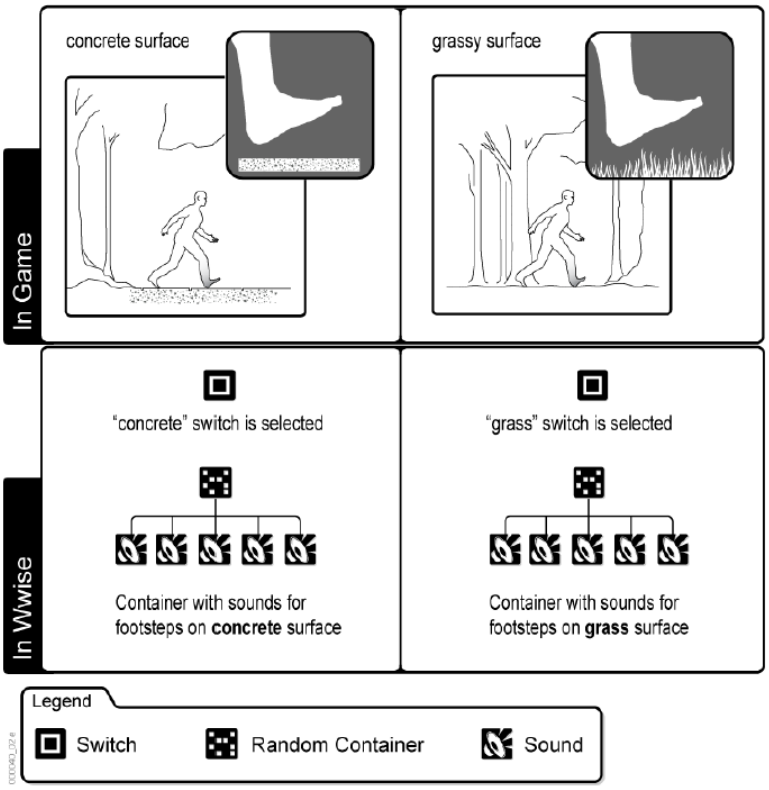
Understanding Switches

In Wwise, Switches represent the different alternatives that exist for a particular game object within the game. Sound, music, and motion objects are organized and assigned to Switches so that the appropriate sound or motion object will play when a change is made from one alternative to another in game. The Wwise objects that are assigned to a Switch are grouped into a Switch Container. When an Event signals a change, the Switch Container verifies the Switch and the correct sound, music, or motion object is played.

Example 6.2. Using Switches - Example

Let's say you are creating a first-person shooter game, where the main character can walk and run through a variety of different environments. Within each environment, you have different ground surfaces, such as concrete, grass, and dirt, and you want different footstep sounds for each of these surfaces. In this case, you can create Switches for the different ground surfaces and then assign the different footstep sounds to the appropriate Switch. When the main character is walking on a concrete surface, the “concrete” Switch will become active and its corresponding sounds will play. If the character then moves from a concrete surface to a grassy surface, the “grass” Switch will become active and its corresponding sounds will play.

The following illustration demonstrates how the active Switch determines which footstep sound is played.



Understanding RTPCs

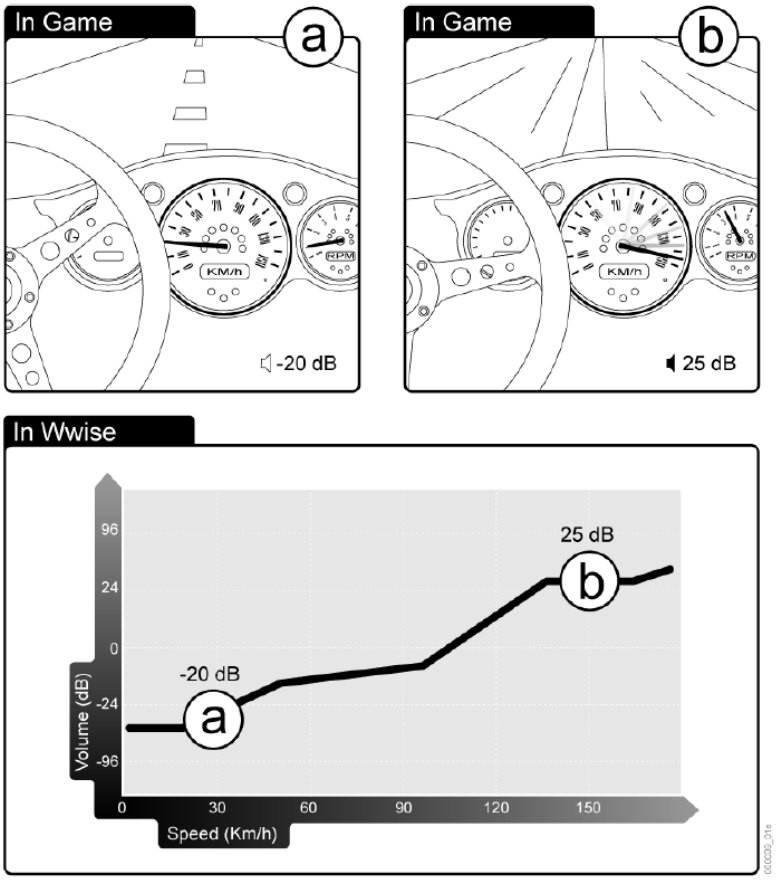
Real-time Parameter Controls (RTPCs) enable you to edit specific object properties in real time based on real-time parameter value changes that occur within the game. Using RTPCs, you can map the Game Parameters to property values, and “automate” property changes to enhance the

realism of your game. The parameter values are displayed in a graph view, where one axis represents either the Switch Group or the property values in Wwise, and the other axis represents the in-Game Parameter values. By mapping property values to Game Parameter values, you create an RTPC curve that defines the overall relationship between the two parameters. You can create as many curves as necessary to create a rich and immersive experience for the players of your game.

Example 6.3. Using RTPCs - Example

Let's say you are creating a racing game. The volume and pitch of the engine sounds need to fluctuate as the speed and RPM of the car rise and fall. In this case, you can use RTPCs to map the pitch and volume level of a car's engine sounds to the speed and RPM values of an in-game car. As the car accelerates, the property values for pitch and volume will react based on how you have mapped them.

The following illustration demonstrates how the volume is affected by the speed of the racing car in the game, based on how it was mapped in Wwise.



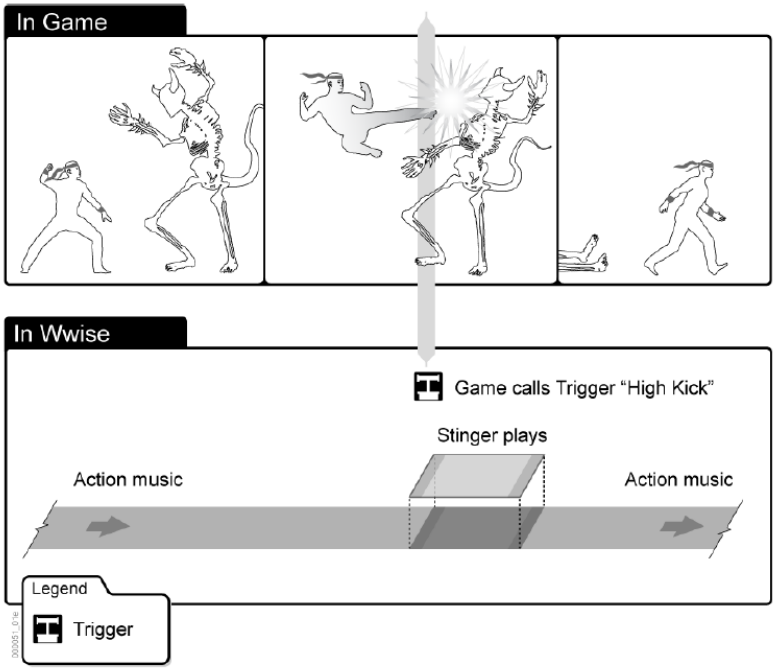
Understanding Triggers

Like all Game Syncs, a Trigger is a Wwise element that is called by the game and then defines a specific response in Wwise to accommodate what is happening in the game. More specifically, in interactive music a trigger responds to a spontaneous occurrence in the game and launches a stinger. The stinger, which is a brief musical phrase that is superimposed and mixed over the currently playing music, is a musical reaction to the game. For example, when a ninja draws his weapon, you might want to insert a musical sforzando-type effect over the action music already playing to add even more impact to the scene. The game would call the Trigger which in turn would launch the stinger and your music clip would play over the ongoing score.

Example 6.4. Using Triggers - Example

Let's say that you have created a fighting game where your main character is a ninja fighter. At several points in the game your character goes into action mode where he fights his enemies. When your character lands a powerful kick, you want to place a music clip that will intensify the auditory impact of that scene. To build your music for these sequences, you will need to create a Trigger, perhaps named “High Kick” to be called at these points in the game. In addition, you will define the short music segment that will provide a quick blast of brass to add some “kick”.

The following illustration demonstrates the Trigger mechanism that plays a stinger at a key point in the game.



Game Syncs - Roles and Responsibilities

The following table shows you which tasks related to Game Syncs are the responsibility of the sound designer and which ones are the responsibility of the programmer:

Table 6.1. Game Syncs - Roles and Responsibilities

Tasks	Sound Designer (Wwise)	Programmer (Game Code/Tools)
Create Switch Groups and Switches	X	
Create State Groups and States	X	
Define State Transition Time	X	
Subscribe Switch Containers to Switch and State Groups	X	
Setting up Triggers	X	
Post State and Switch information from the game engine to the Wwise Audio Engine		X

Chapter 7. Creating Simulations

Creating Simulations

With all game projects, there is a great deal of experimenting that goes on to make everything just right. To assist you with these tasks, Wwise has a powerful simulation environment called the Soundcaster. The Soundcaster can be used at any point in the development process to build audio and motion simulations using any of the Wwise objects and Events in your project.

You can use the Soundcaster for a variety of tasks, including:

- Prototyping and experimenting.
- Developing a proof of concept.
- Auditioning sound and music objects simultaneously.
- Profiling audio and motion in your game.
- Mixing and testing audio and motion.

Not only can you create simulations in Wwise using Wwise Events, sound, motion, and music objects, but you can also connect to a game and create simulations using the sounds, motion, and music triggered by the game itself. The simulations you create can be saved as Soundcaster Sessions so that you can return to a simulation at any point in the development process.

Chapter 8. Profiling and Troubleshooting

Profiling and Troubleshooting

One of the biggest challenges for game developers is to create rich and immersive experiences for game players while respecting the limitations and constraints of the various platforms. In Wwise, there are many ways to tailor your game audio and motion to the various platforms. You can, however, take it one step further by using Wwise's Game Profiler and Game Object Profiler to test how your audio and motion performs on each platform. These two sets of tools allow you to profile specific aspects of the audio and motion in your game at any point in the production process on any platform. You can connect to a remote game console and then capture profiling information directly from the sound engine. By monitoring the activities of the sound engine, you can detect and troubleshoot specific problems related to memory, voices, streaming, effects, SoundBanks, and so on. You can profile in game, use the Game Simulator and Soundcaster, or use the Wwise Authoring API to profile prototypes even before they have been integrated into your game.

To help you find the information you need, the Game Profiler layout is divided into the following three views:

- **Capture Log**—a log that captures and records all information coming from the sound engine.
- **Performance Monitor**—a graphical representation of the performance, such as CPU, memory, and bandwidth, for each activity performed by the sound engine. The information is displayed in real time as it is captured from the sound engine.
- **Advanced Profiler**—a comprehensive set of sound engine metrics that can help you monitor performance and troubleshoot problems.

The Game Object Profiler layout contains the following views:

- **Game Object Explorer**—the control center for the Wwise game object profiling tools, where you select game objects and listeners to be watched in real time.
- **Game Object 3D Viewer**—A three-dimensional visual representation of game objects and listeners.
- **Game Sync Monitor**—A tool for analyzing RTPC values in real time. During gameplay, graphs are drawn for the RTPC values that change for watched game objects.

Because these views are so tightly integrated, you can locate problem areas, determine which Events, actions, or objects are causing the problems, determine how the sound engine is handling the different elements, and then fix the problems quickly and efficiently.

Chapter 9. Understanding SoundBanks

Understanding SoundBanks

To effectively manage the audio and motion components of a game, Wwise puts all the audio and motion data for your game into banks. A bank is basically a file that contains your game's audio and motion data, media, or both. These banks are loaded into a game's platform memory at a particular point in the game. By loading only what is necessary, you can optimize the amount of memory that is being used for audio and motion by each platform. Banks are the product of all your work and contain the final audio and/or motion content that becomes part of your game.

In Wwise, there are two types of banks:

- **Initialization bank**—a special bank that contains all the general information about a project, including information on the bus hierarchy, and information on States, Switches, RTPCs, and Environmental effects. The Initialization bank is automatically created when Wwise generates the SoundBanks. The Initialization bank is usually loaded once at the beginning of your game so that all the general project information is easily accessible during game play. By default, the Initialization bank is named “Init.bnk”.
- **SoundBank**—a file that contains a combination of Event data, sound and motion structure data, and/or media files. Unlike the Initialization bank, SoundBanks are generally loaded and unloaded at different points in the game to better utilize platform memory usage. Event and project structure metadata can also be added to different SoundBanks than the media allowing you to load media files only when they are absolutely required. Because all platforms are different, Wwise allows you to easily tailor the SoundBanks for each platform and generate the SoundBanks for all platforms simultaneously. Wwise also provides you with tools for troubleshooting any issues related to your SoundBanks to make sure that you are respecting the limitations of the different platforms.

To help you work more efficiently, a SoundBank layout is available in Wwise. This layout contains all the views you will need to create, manage, and generate the SoundBanks for your project, including the SoundBank Manager, SoundBank Editor, Project Explorer, and Event Viewer.

File Packager

The SoundBanks generated for a Wwise project as well as any streamed media files can be grouped into one or more packages using the File Packager stand-alone utility. A file package is a self-contained unit that abstracts a file system, which means you can avoid some of the limitations

of a platform's file system, including the limit on the length of filenames as well as the actual number of files. File packages can also help you better manage language versions as well as downloadable content that is made available post release.

Chapter 10. The Wwise Sound Engine

The Wwise Sound Engine

Using Wwise, you can build astonishing sound, music, and motion structures and package them into SoundBanks, but you also need a powerful and reliable sound engine to deliver the sound and motion you designed. At a very basic level, the Wwise sound engine manages and processes all aspects of the audio and motion within your game in real time. It was designed to be easily integrated into your game development pipeline for inclusion in the final game.

Normally, these integration and processing tasks require significant amounts of programming, but the Wwise sound engine dynamically creates the processing pipeline, leaving developers free to customize the engine in order to fit the specific needs of any game on any supported platform.

Through its sophisticated behavior and tight integration with the Wwise authoring application, the sound engine is also able to perform the following functions:

- Handles common playback behaviors, including random and sequential play, which are created and defined in the authoring application by the sound designer.
- Handles fades and cross fades out-of-the-box that are created and fine-tuned in the authoring application by the sound designer.
- Manages the priority of sound and motion objects using playback limits, specific priority settings, and virtual voices all set by the sound designer in the authoring application.
- Supports an unlimited number of environments out-of-the-box through a simple API. In addition, because the sound engine creates and destroys the environmental routing dynamically, it ensures a consistent experience on all platforms and reduces the memory footprint and CPU workload.
- Supports the naturally occurring conditions of obstruction and occlusion when the source is partially or completely blocked by elements within a game.
- Supports up to 8 different listeners in the game.
- Contains debugging instrumentation code that is exposed visually in real time in the authoring application. As a result, the sound designer can analyze the profiling output and take proper action in real time while the game is running.

For more information about the capabilities of the Wwise sound engine, refer to the Wwise SDK documentation.

Chapter 11. Listeners

Listeners

A listener represents a microphone in the game. A listener has a position and orientation in the game's 3D space. During game play, the coordinates of the listener are compared with the game object's position, so that 3D sounds associated with game objects can be assigned to the appropriate speakers to mimic a real 3D environment. Programmers must ensure that the listener's positional information is kept up to date; otherwise sounds will be rendered through the wrong speakers.

Multiple Listeners

In a single-player game where you always see only one point of view in the game, one listener is enough. However, if multiple players can play on the same system, or if multiple views are displayed at the same time, each view requires its own listener so audio is appropriately rendered for all of these views. The Wwise sound engine supports up to eight listeners.

By default, every registered game object is assigned to the first listener only. However, the programmer has the flexibility to dynamically assign or unassign any game object from any listener.

There are many challenges to implementing audio for multiple listeners because the positioning of sound sources doesn't always make sense in relation to what each player is seeing. This is mostly caused by a game using only one set of speakers to reproduce the 3D environment for several players. Wwise offers a variety of tools and techniques to offset this limitation so that the audio experience is as realistic as possible for all players. For more information about how Wwise handles multiple listeners, refer to the section “Integrating Listeners” in the SDK.

Listeners - Roles and Responsibilities

The following table shows you which tasks related to listeners are the responsibility of the sound designer and which ones are the responsibility of the programmer:

Table 11.1. Listeners - Roles and Responsibilities

Tasks	Sound Designer (Wwise)	Programmer (Game Code/Tools)
Setting the listener's positional information		X
Assigning game objects to listeners		X
Managing multiple listeners		X

Chapter 12. Division of Tasks Between Designer and Programmer

Division of Tasks Between Designer and Programmer

As you have probably already figured out, Wwise takes an approach to sound design and integration that is different from that of other sound engines. More control is given to the sound designer, which means repetitive and time consuming tasks often handled by developers are reduced to a minimum, letting both sound designers and developers concentrate on more interesting, creative work. Most dependencies have been eliminated so that each group can focus on their core competencies and work together more efficiently. Wwise recognizes two distinct roles with specific tasks for each: sound designer and audio programmer.

Sound Designer Responsibilities

The sound designer is responsible for:

- Creating audio hierarchies and behaviors.
- Creating audio Events.
- Integrating audio Events into your world building application.
- Setting sound properties and sources.
- Defining sound positioning (3D).
- Determining audio signal routing and mixing aspects for all sounds.
- Assigning real time parameter controls and game States.
- Defining effect property sets for the various environments in your game.
- Defining volume and LPF obstruction and occlusion properties.
- SoundBank management and optimization.
- Customizing multiple platforms.
- Performing language localization.

Audio Programmer Responsibilities

The audio programmer is responsible for:

- Integrating the Wwise sound engine into the game engine.
- Integrating audio Events using code.
- Registering game objects that emit the sounds in your game.
- Calling AK::SoundEngine::PostEvent() methods to trigger Events that contain one or several audio actions.
- Manage the loading and unloading of SoundBanks as specified by the Sound Designer or by the game using the SoundBanks definition file and File Packager Utility.
- Setting the positions of 3D game objects in relation to the listener's coordinate system.

- Triggering State and Switch changes and updating real time parameter controls.
- Setting the percentage of each environmental effect that is applied to the sounds in your game.
- Calculating obstruction and occlusion values for each game object in relation to the listener.
- Managing memory resources, including loading and unloading of SoundBanks, handling streaming, registering of plug-ins, and so on.
- Writing source and effect plug-ins. The sound engine's plug-in architecture allows game developers to expand Wwise's functionality to meet their specific game requirements.
- Integrating the Wwise Authoring API into your development tools.

Project Planning

At the beginning of the project, audio programmers and sound designers should meet to discuss the various resources that will be allocated for the audio of the game title. These resources include, but are not limited to memory, disk space, number of streams, and size of SoundBanks.

Based on game design and technological constraints, the designer and developer must work together to decide:

- How Events will be integrated and triggered by the game engine.
- Project Hierarchy and Work Unit organization for Workgroups.
- Player perspective and Listener positioning.
- Soundbank loading/ unloading strategy.
- Music integration and special needs for interactive music.
- Which Game Parameters should be used as Real Time Parameter Controls (RTPC).
- Which global elements of the game can drive the Mix and State mechanisms.
- Which sound structures need a Switch mechanism.

Chapter 13. Conclusion

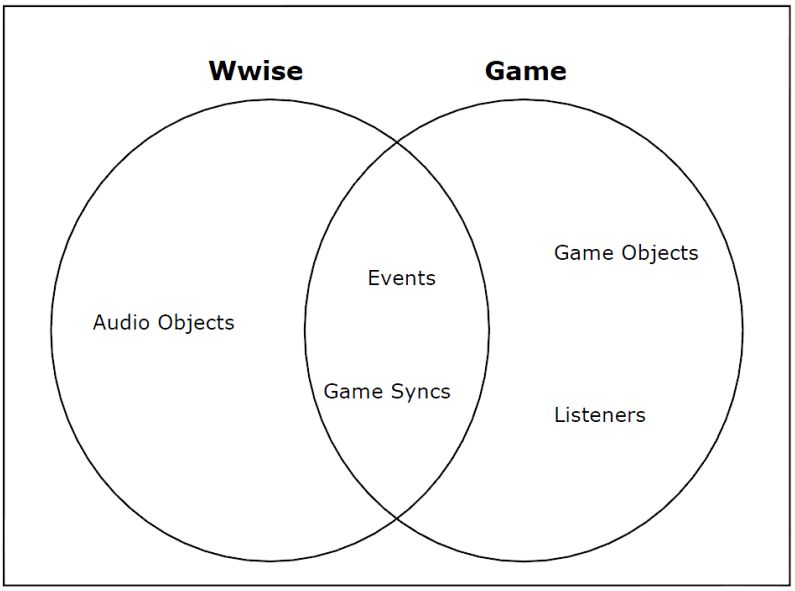
Conclusion

From its inception, Wwise has attempted to create a clear distinction between the roles of the sound designer and the programmer. Each group has its own core competencies, and should be able to focus on the tasks that push game audio to the next level, enhancing the overall gaming experience.

From this separation of tasks, came the five main components that make up the core of Wwise:

- Audio objects
- Events
- Game Syncs
- Game objects
- Listeners

Each component falls under the responsibility of either the sound designer or the programmer, as shown in the following illustration.



There are two components that are integral to both Wwise and the game: Events and Game Syncs. These two components, which drive the audio in your game, create the necessary bridge between the audio assets in Wwise and the components managed in the game.

Wwise represents a paradigm shift in the way audio is developed and integrated in video games. It does require game designers and developers to approach their work in a new way, but it also allows them to work more efficiently and to focus on their areas of expertise.

Now that you have a basic understanding of Wwise's approach to game audio development, you are ready to jump in, and take full advantage of all Wwise has to offer.