46

Creating Dynamic Soundscapes Using an Artificial Sound Designer

Simon Franco

46.1 Introduction

A game's audio is the end result of the work done by the sound designer. A game's sound designer will typically create audio content (sound effects and music) and then create sound events to trigger that audio content. Sound events are often authored using middleware tools such as Wwise [Wwise 06] or FMOD [FMOD 02] and are triggered by the game.

Game audio is often triggered and managed via a fixed set of conditions—and often these conditions have no connection to each other. For example, sound effects and music can be triggered by a number of different systems, such as the level scripting system, the animation system, and game code reacting to events.

These disconnected methods for triggering audio present us with a number of problems. The player will always have the same audio experience every time they play the game. This is the result of static correlations between in-game triggers and audio events. After playing the game for some time, the soundscape can come across as predictable and boring. You may also have systems competing to play audio cues that serve the same

purpose, such as trying to set the game's ambience. In addition, these isolated methods for triggering audio could accidentally inform the player of hidden information. For example, a piece of music starting may accidentally inform the player of a hostile character hidden around a corner. This can lead to scripted sequences of game play being ruined.

In addition to these limited methods for triggering audio, our techniques for mixing audio at runtime are typically very primitive. The sound designer will often create a pool of audio mix snapshots for use in the game [Bridgett 09]. Each audio snapshot will typically hold information on volume settings, volume curves, and various filter settings for each sound category. Categories tend to be groups of similar sound types, such as footsteps. Unfortunately, there are also problems when using audio mix snapshots:

- An appropriate snapshot which complements the game's current state must be chosen and applied. This can be done in a number of ways—for example, having a level designer script when to apply an audio snapshot, or by having some game code monitor for a condition to be met.
- Each snapshot always contains a fixed collection of settings. Typically, these audio snapshots represent a game state, such as a calm moment, being in a safe area, or being in a combat situation. This requires the audio designer to decide ahead of time which game scenarios they would like to create a snapshot for, and author the appropriate snapshots.

This article discusses the idea of developing an Artificial Sound Designer to solve these issues. The Artificial Sound Designer avoids these problems and can make intelligent audio decisions by monitoring the game's current state, as well as retaining knowledge of the game's previous states.

46.2 The Artificial Sound Designer

The purpose of the Artificial Sound Designer (ASD) is to ensure that the player has the richest possible experience by ensuring that they have a varied soundscape and making that soundscape closely match the game's state. The ASD is built around a rule set that represents the knowledge and experience of a human sound designer. At a high level, it is composed of the following pieces:

- An event system to pass information about the game's state to the ASD.
- A database that holds state information on all relevant game objects.
- The rule set, which examines the events raised in this frame, the state of the database, and the audio state, in order to determine which audio actions to execute (if any).

These component pieces are to be used to facilitate the Artificial Sound Designer. Events are posted to the ASD from in-game. We then use those posted events to update the ASD database. Then the ASD examines the events raised plus its database to change the soundscape and play appropriate sounds. Depending on the situation, the ASD may execute one or more of the following actions when a rule is satisfied:

- Play a sound effect or piece of music, and associate that piece of audio with a game object if needed. For example, we may want a piece of audio to track an object in the world.
- Adjust the volume/DSP settings for categories of sounds. This can be used to "duck out" (i.e., reduce the volume of) sounds deemed unimportant to the current in-game situation, or to increase the volume of sounds we want the player to focus on (such as an imminent threat).
- Configure any underlying systems that generate events for the ASD. For example, if you have a system to count the number of hostile characters within proximity to the player—then the ASD must be provided with a way to configure that system's radius.

46.3 Generating Events

The Artificial Sound Designer uses events as a means of being notified of what is happening in game. Events typically have a type in order to help categorize and describe the type of event, such as a footstep, an explosion, or a gun firing. As well as having a type, an event will also have a subject. The subject of the event would typically be a reference to the game object triggering the event. The event will also store any other associated information that was part of that event. For example, a footstep event would typically also store the material that the character stepped on.

We have two basic types of events that can be posted. Information-only events notify the ASD of changes in the game state, which may not be directly related to a sound emitting action. For example, if a game object has been spawned or despawned, or if the game is paused or resumed, then we can use an information-only event to pass this knowledge to the AI. These events may not cause sounds to be played directly, but they can affect the way in which we play other sounds.

The second type of event is the play-request event. These events are used when we want to play a specific type of sound, such as a gunshot. These events replace where previously we would have called directly into the sound system. For example, we now post an event at the point when an explosion occurs, or when the animation system causes feet to strike the ground (so that we can play footstep sounds). By using play-request events, we give the ASD the opportunity to make decisions about which sound sample to play and how loud to play it (e.g., should the volume of gunshot events be reduced so that we can hear enemy speech?). The ASD can evaluate the event and use the player's current context, and the state of the game objects involved, to drive its decisions.

Listing 46.1 shows an example of a general event class that can be inherited from and extended for each type of event. Each event should have a process function which will update the subject game object's database record appropriately for that event type. It can also return whether it is an information-only event or a play-request event.

46.4 Creating and Maintaining the Database

The database contains the Artificial Sound Designer's knowledge. This section will discuss the elements making up that database.

Listing 46.1. An example Event base class.

```
class Event
{
public:
        Event(GameObject * obj) : m_object(obj) {}
        virtual void process(GameObjectRecord &) = 0;
        virtual const GameObject * get_subject_game_object() const
                {return m_object;}
        virtual EVENT_TYPE get_type() const = 0;
        virtual bool is_play_request_event() const
                {return false;}
protected:
        GameObject * m_object;
}
```

46.4.1 The Database Structure

The database contains a number of tables. We will need a table for game objects, a table for sounds previously played, and a table for the ASD to store any additional data it wishes to keep. This last table is used to help the ASD keep track of nonaudio or game object state information. For example, we may want to store the time since the player was last under threat.

Each game object is registered with the ASD's database and, along with a reference to the object, we store a set of flags describing the object's state. This is so the system can query its database of knowledge in order to determine information on a game object, and on the world state.

As well as storing information within a game object's record on the object's actual state, the record should also contain data describing the object's perceived state to the player. For instance, we may wish to store the last position where it was seen by the player. Although the majority of the information about each object is stored in the database, certain information (such as the object's current position) may be queried directly from the engine. This allows us to avoid storing redundant information.

The sound-history table stores a record of each type of sound, along with its category, when it was last played and how many times it was played. This is so the ASD can keep track of what it has previously played. This knowledge is used when selecting which sounds to play back. For example, we may want to avoid playing the same piece of music too often, or perhaps not play a tension piece of music if we had just played a piece of combat music.

This rich body of information allows the Artificial Sound Designer to make informed decisions when selecting audio relating to a particular game object. These decisions take into account not only the object's characteristics, but also its history with the player. For example, if the player is currently engaging in combat with an enemy that had previously dealt damage to the player, or even killed the player, then we could use that information to select appropriate music and dialog (such as playing more intense music, or having the character taunt the player).

Depending on the multithreaded nature of your game and audio engine, it may also be advisable to create a database table to store a record of the audio state (i.e., the sounds that

are playing and their settings). By having a separate record of which sounds are playing, we avoid the problem of the sound state changing unexpectedly while we are deciding what audio to play next.

46.5 Defining the Artificial Sound Designer Rule Set

The Artificial Sound Designer has two separate rule sets, which can be thought of in a similar manner to a rule based system [Negnevitsky 11]. One rule set is consulted when making changes to the overall soundscape. Other rule sets can be assigned to the different types of play-request events. These rules are then consulted when processing play-request events in order to select the most appropriate sample to play. The human sound designer needs the facility to easily create and edit these conditional rules for the ASD to process. The rules must be listed in priority order, with the highest priority rules first. When a conditional rule is satisfied, it will perform one or more actions.

To form these rules, the sound designer must have access to typical logical operators such as AND, OR, and NOT. The sound designer will then use these to form the conditions within the rules which the system will process.

The simplest way to implement this is to use an embedded scripting language such as Lua [LUA 93]. This presents an easy way for the sound designer to formulate rules. It also provides easy methods to wrap access to events, the database, and perform sound actions in a Lua interface.

In addition, consideration should be given to adding extra logging functionality to record which rules were fired, and which sound actions were executed. This will help the sound designer and programmer understand how decisions about the game's soundscape were reached.

In Listing 46.2 we show a sample pseudocode snippet to modify the in-game music.

Listing 46.2. A pseudocode sample rule set to control music selection.

```
if (EventRaisedThisFrame(PLAYER_DEATH)) then
        PlayMusic(MUS_GAME_OVER)
elseif (EventRaisedThisFrame(SCRIPTED_MUSIC) and
SoundSystem:PlayingMusic()) then
        StopMusic();
elseif (Database:Objects:NumberOfObjectsInRangeOfPlayer
(GRENADE, NO_FLAGS, 5.0f) > 1) then
        PlayMusic(MUS_WARNING)
elseif (Database:Objects:NumberOfObjectsInRangeOfPlayer
(ENEMIES,SEEN|HEARD,16.0f) > 15) then
        PlayMusic(MUS_BATTLE)
elseif (Database:Objects:NumberOfObjectsInRangeOfPlayer
(ENEMIES,SEEN|HEARD,16.0f) >= 1) then
        PlayMusic(MUS_DANGER)
elseif (not SoundSystem:PlayingMusicInCategory(MUS_CALM)) then
        PlayMusic(Database:Sound:GetLeastPlayedSampleInCategory
(MUS_CALM))
endif
```

46.6 Updating the Artificial Sound Designer

Once per frame, after the other game systems have had a chance to post events, the Artificial Sound Designer will perform its update. This update consists of three phases: updating the database, changing the soundscape, and playing requested audio.

46.6.1 Updating the Database

In the first phase, we process the events raised during the current frame and incorporate them into the database. A simple way to implement this is to give each type of event a `Process()` function, which will handle updating the corresponding database record. We can then simply loop over the events, calling process on each, in order to bring the database up to date.

46.6.2 Changing the Soundscape

Now that the database is up to date, we can perform the main update for the Artificial Sound Designer. During this phase the ASD will process its main rule set in order decide whether to play any new sounds, or change the playback of sounds already in progress (for instance by stopping all sounds of a particular type, or changing the volume of one or more categories).

46.6.3 Playing Requested Audio

In the final phase, we process the play-request events. As discussed in an earlier section, these events are sent from the game in order to request that specific sounds be played. Playing requested audio typically involves processing each event that has requested audio playback. The Artificial Sound Designer can examine the event parameters and use a rule set for that event type (if one has been set) to select the actual audio sample to play. For example, you may have an NPC shout "Who's there?" when they spot an intruder. If the database has information stating that the NPC had seen the player before then we change the speech to "There he is again!" to reflect this. This persistence helps re-enforce the player's interactions with the game world.

46.7 Conclusion

Creating a dynamically changing soundscape that responds closely to the player's actions helps to deliver a rich and varied audio experience. Using an Artificial Sound Designer empowers your sound designers to create a more immersive experience for the player.

Designing the rule set used by your game to shape the soundscape requires careful consideration. Where possible, work should be done to ensure that there is only a short turnaround between changing the rule set and testing it in game.

References

[Bridgett 09] R. Bridgett. "The Future Of Game Audio—Is Interactive Mixing The Key?" http://www.gamasutra.com/view/feature/4025/the_future_of_game_audio__is_.php, 2009.

[FMOD 02] Firelight Technologies. "FMOD." http://www.fmod.org, 2002.
[LUA 93] "Lua." http://www.lua.org, 1993.
[Negnevitsky 11] M. Negnevitsky. *Artificial Intelligence: A Guide to Intelligent Systems*, Addison-Wesley, 2011, pp. 25–54.
[Wwise 06] Audiokinetic "Wwise." http://www.audiokinetic.com, 2006.