FMOD, an Audio Engine for Video Games, Adapted for Theater

Stephen Swift

Abstract

FMOD is an audio integration tool for video games. It is used to build, process, mix, and route game controlled sound events. Since video games need to be able to react to player choices, FMOD is designed to make it easy to modify audio parameters in real-time. Although it is designed for video game production, it can be adapted for use in theatrical playback and offers many unique capabilities not currently widely employed in theater. This case study will examine how the FMOD audio engine was used to create a responsive score and sound design for the production of *Wood Boy Dog Fish*.

Introduction

Wood Boy Dog Fish, created and produced by the Rogue Artists Ensemble, is a piece of hyper-theater which combines puppetry, physical performance, and highly stylized design to tell the story of Pinocchio. It was absolutely critical that the sound and music harmonized with the onstage physical movement; however, it was imperative to the director that the performance be organic, open to improvisation, and not be locked to a track. The composer and I turned to FMOD to design a score and sonic environment that could respond to the action on stage in a musical and cohesive manner.

The world of the play fluctuated between real and imagined, life and the afterlife—often quickly turning on a dime. It was important that the design support these different moods, and I was interested in processing a common core of sounds in different ways depending on what realm we were in. I knew FMOD was able to automate audio processing in realtime, and I wanted to leverage that power to seamlessly transform my cues through various different aural conditions. I also understood that FMOD was very good at creating multiple variations of sounds and adjusting the pace at which they played. Consider common game events such as gunfire or footsteps breaking out into a run—each repetitive sound has subtle sonic changes applied to it to make the sequence sound more believable. My goal was to use FMOD's automation and modulation tools to create rich and spontaneous soundscapes for the play.

Another major goal was to create adaptive musical cues that would both progress harmonically and closely match the actors' timing (which could wildly fluctuate from night to night). It was important to Adrien Prévost, the composer, that his musical cues had a beginning, middle, and end that resolved musically instead of simply fading out. We considered Ableton Live, but since FMOD also has features to create non-linear music, we decided to run the music tracks through FMOD since I was already planning on using it for other sound cues.

All these objectives resulted in a plan to use FMOD in order to add variation to the sound cues, manipulate effect processing, and arrange music—all dynamically controlled in real-time.

FMOD Overview

In order to describe how FMOD was used to achieve our design objectives, it is important that the reader is familiar with some basic FMOD concepts. While it is beyond the scope of this paper to be an FMOD manual, this section will describe FMOD's relationship with game engines and explain the core features of FMOD's editing tool, FMOD Studio. FMOD has excellent online documentation and tutorials if the reader wishes to explore any topic mentioned here further[1].

FMOD is designed to be implemented into a game engine or other program, not to be run directly from its own editor interface. FMOD's audio engine offers an API with broad platform support (Windows, Mac, Linux, iOS, Android, Web/JavaScript, Xbox, PlayStation, Nintendo, etc). However, FMOD has also developed integration packages for the two major game engines, Unity and Unreal. These integration packages are trivial to add to game projects and provide developers with a wealth of features and tools they would otherwise have to build themselves if using the API alone.

When FMOD is integrated into a game engine, the game's physics drives much of the audio mixing and routing. Typically there is one FMOD audio listener object attached to the game's player controller (the character controlled by the user). Each object in the game that makes a sound has an FMOD sound emitter attached to it. The audio level and roll-off is determined by the distance between the listener and emitter. For example, as the player controller gets closer to a sound emitter object, the object's audio level increases. Similarly, if the object

[1] https://fmod.com/resources/documentation-studio?page=welcome-to-fmod-studio.html

was to the player's left, the audio would be panned to the left. If the object was behind the player, the audio would route through the rear surround channels. The game data drives the FMOD engine, which contains the audio samples and rules for how sound should behave given a certain game state.

FMOD Studio is the editor application used to create and edit audio to be played back by the FMOD engine. The FMOD Studio interface will be comfortable to anyone familiar with a Digital Audio Workstation (DAW) such as Pro Tools or Logic Pro.

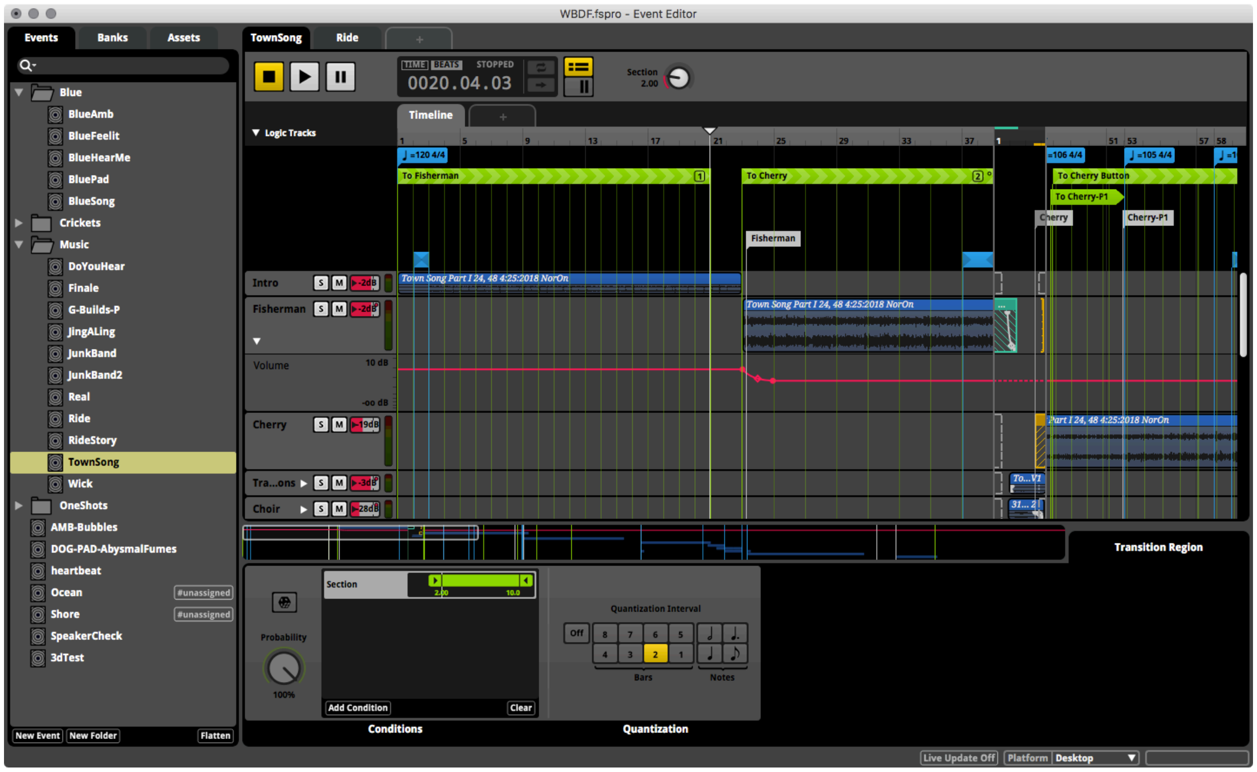


Figure 1. FMOD Studio Event Editor Interface

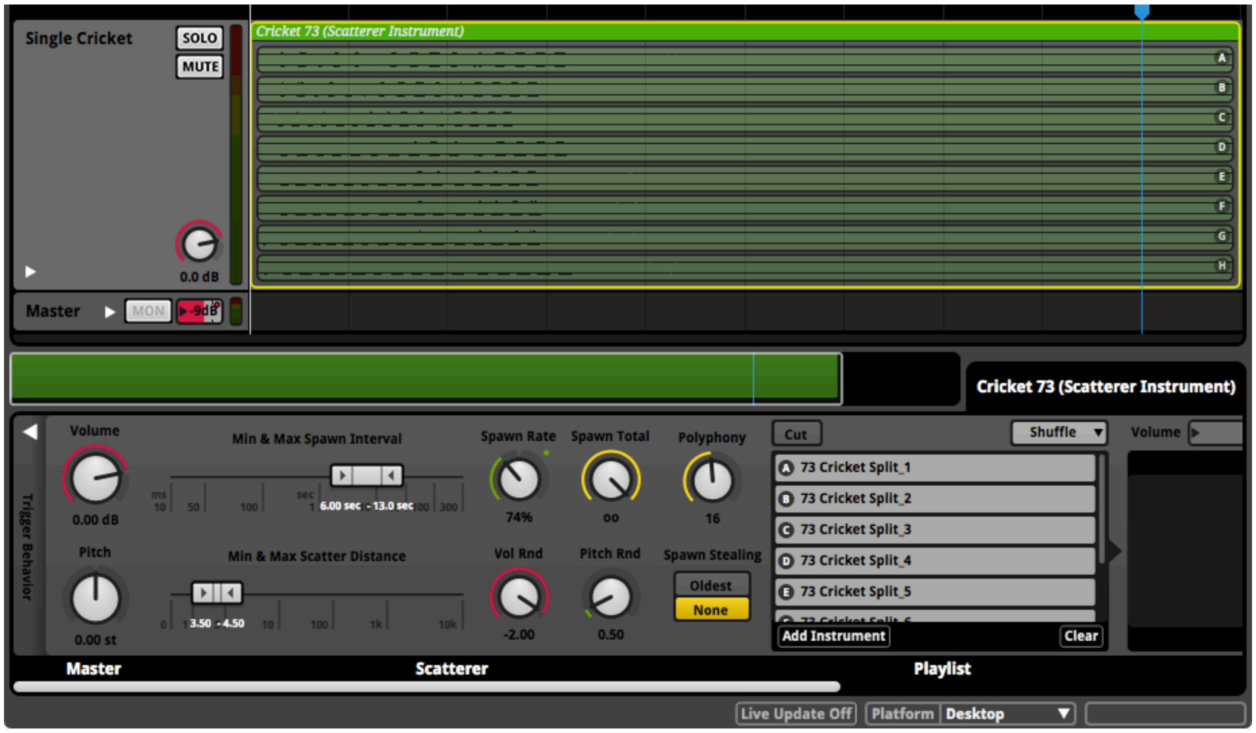
The Browser on the left is used to open and edit FMOD events. The main view contains the event's timeline and tracks. The regions on the tracks are called instruments. The bottom panel is the Deck, which is used to view and edit the properties of the selected item in the editor.

The main building block of an FMOD project is an event. An FMOD event is akin to a theatrical sound cue; it is the container that holds the audio files and related settings. Video games or other programs reference the event to play or stop its audio content. A collection of events is called a bank. Banks provide control over which content is loaded into memory at any given time during runtime.

Each event has its own timeline and set of tracks. Similar to a DAW, FMOD has audio, automation, return, and master tracks. Audio tracks can be routed into the master track or another audio bus. Automation tracks display the automation points and curves along the timeline or parameter range. Return tracks receive the audio sends from the signal chains of other buses and tracks. The master track is the primary bus of the event. It mixes down all the tracks and instruments that route into it. Effects such as eq, reverb, or other processing can be inserted onto an instrument region, track, bus, or output channel.

Instruments can contain one or more audio clips, sub-instruments, whole event references, or other special commands. They output signal when the playhead is over the instrument region. For example, a single instrument is the simplest FMOD instrument. Like an audio region in a DAW, it references a segment of one audio file. In its default mode, it plays back each audio sample in the file linearly as the playhead scrubs across the region. However, it also has an asynchronous mode, which is more like a trigger. When the playhead intersects any part of the region, the instrument begins to play from the beginning of the sample. Asynchronous instruments are more useful than synchronous instruments in most cueing situations where it is preferable for the cue play from the start of the file, not from whatever timecode value the event's playhead is currently at.

One of the most powerful and unique features of FMOD is the scatterer instrument. A scatterer instrument is similar to a QLab group cue set to random mode; it is a collection of sound files that can be played randomly. However, there is a lot of additional variety that can be incorporated besides just randomizing which file is played. The scatterer instrument is designed to create rich surround atmospheres with a limited set of samples by re-triggering periodically on a variable interval rate. With each re-trigger, a sound can have a new random pitch, volume, or spatial position. Almost every setting can be automated, modulated, or randomized.

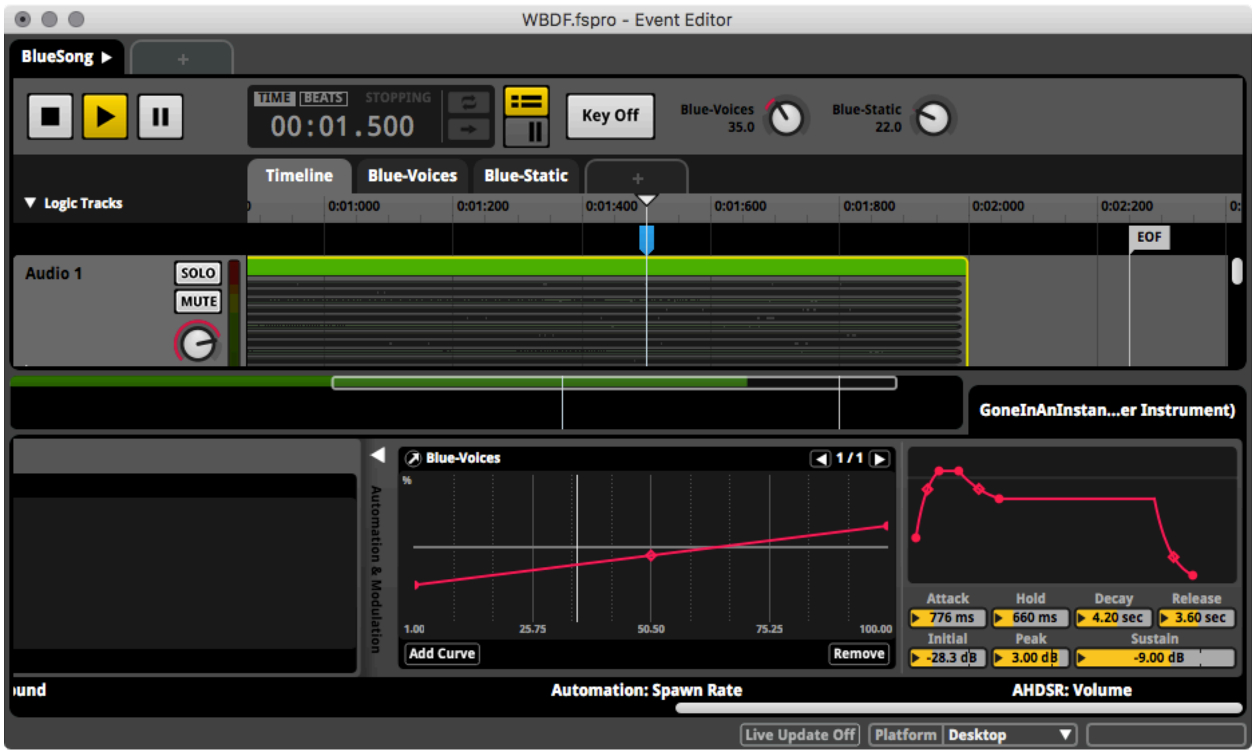


Volume 0.00 dB
Pitch 0.00 st
Min & Max Spawn Interval 6.00 sec - 13.0 sec
Min & Max Scatter Distance 3.50 - 4.50
Spawn Rate 74%
Spawn Total oo
Polyphony 16
Vol Rnd -2.00
Pitch Rnd 0.50
Spawn Stealing: Oldest [✓ None]
Shuffle
73 Cricket Split_1
73 Cricket Split_2
73 Cricket Split_3
73 Cricket Split_4
73 Cricket Split_5

Figure 2. Cricket scatterer instrument.

Figure 2 is a screenshot of the scatterer instrument controls. In this example, eight different samples are loaded into the instrument’s playlist, which is set to randomly play. The interval rate is defined to re-trigger every 6 to 13 seconds, and the spawn rate scales that interval up and down. An infinite spawn total means that the instrument will continue to re-trigger forever until the instrument is stopped. Polyphony limits the number of simultaneous audio streams the instrument can be output at once, 16 in this case. Once the limit is reached, this instrument will not re-trigger until one of the audio streams ends (when the end of an audio clip is reached). Distance controls how close or far the sound will be positioned from the audio listener object thereby adjusting attenuation. The unit of distance is defined by the game engine, which for Unity is meters. Volume randomization sets the maximum level reduction that will be applied to the sound. In this example, the audio sample will be reduced a random amount between 0 to 2 dB. Similarly, the pitch randomization will increase or decrease the sample's pitch up to a half semitone. Each of these settings provides yet another method to add variation to a collection of sound clips. By configuring ranges on multiple controls, an exponential number of sonic variations can be achieved.

Almost every control in FMOD can be modified by automation, modulation, or randomization. Randomization sets the control's value—when the event or instrument is triggered—within a user-defined range. Modulation changes a control's value as the event plays. The most common FMOD modulator is AHDSR (Attack, Hold, Decay, Sustain, Release) envelope modulation. The modulated value is altered when playback is first triggered, over time as it continues to play, and when playback is stopped. Applying modulation per event or instrument is an extremely fast and powerful way to ensure each sound cue or clip has an appropriate ramp-in and tail.



TIME BEATS STOPPING 00:01.500

Blue-Voices 35.0

Blue-Static 22.0

Attack	Hold	Decay	Release
776 ms	660 ms	4.20 sec	3.60 sec

Initial	Peak	Sustain
-28.3 dB	3.00 dB	-9.00 dB

Figure 3. AHDSR Modulation on the scatterer instrument Volume Control.

In Figure 3, a modulator is assigned to an instrument's volume control. When the instrument activates, its volume level ramps from -28 dB to +3 dB over 1.2 seconds, holds at that level for 0.6 seconds, decays down to -9 dB over 4.2 seconds, and sustains at that level. When the instrument receives a stop trigger, it fades over 3.6 seconds before stopping playback.

Besides modulation, properties (e.g. volume, pan, pitch, eq bands, wet/dry mix, etc.) can also be automated. Automated properties are linked to user-defined variables, known as parameters in FMOD. Parameter values are intended to be updated at any time procedurally by the game engine. Custom automation curves are defined for each property or properties the parameter controls. As the parameter value changes, the property value is set according to its automation curve.

Figure 4 shows how the master track's surround panner is automated via a parameter. The white dot represents the pan position of the sound event. The green arc represents the range of the sound's angular extent. No extent means the sound position will always emanate from a single location, while a large extent means the sound is spread across a wider pan range. In this example, as parameter value increases, the panner's surround extent increases.

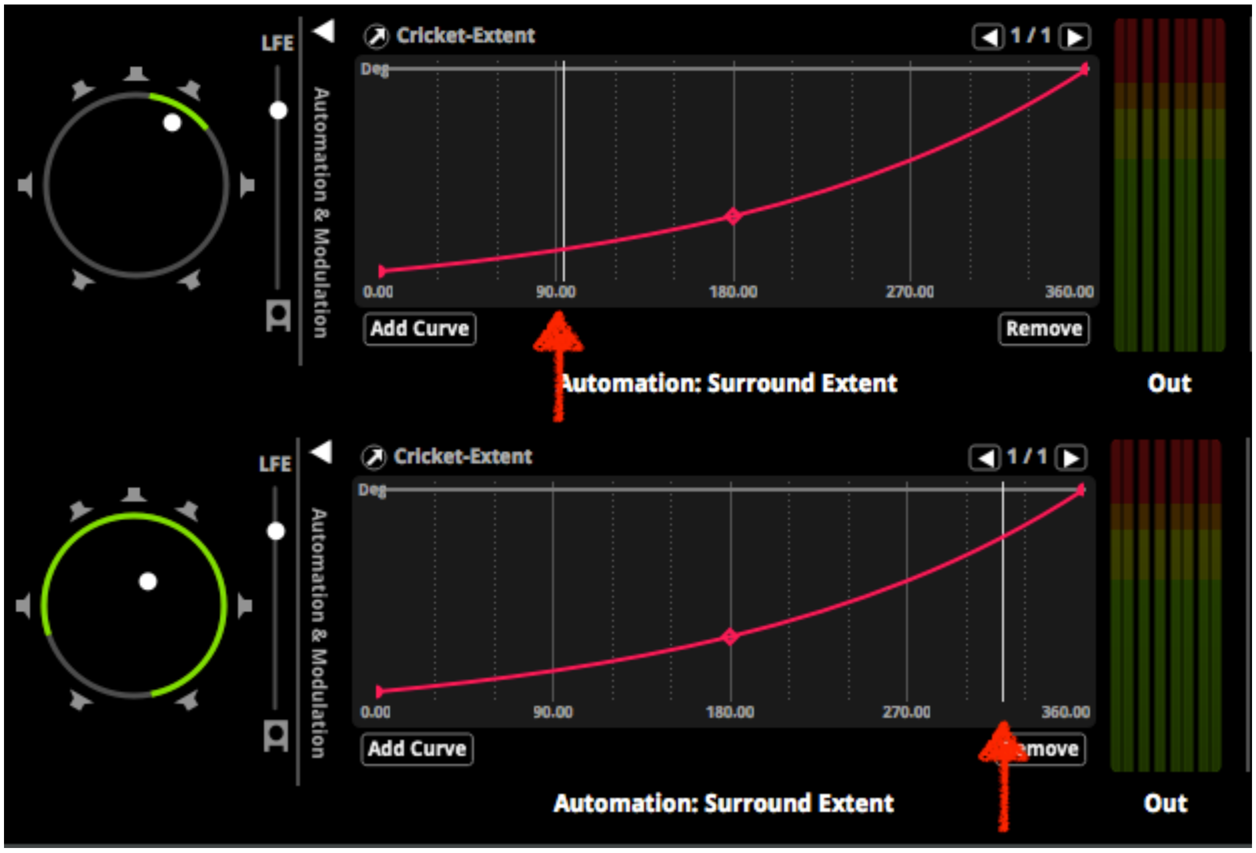


Figure 4. The FMOD Surround Panner with a low and high Surround Extent.

As shown above, the value relationship between the parameter value and the settings it controls does not need to be linear. One parameter can control multiple control property values—each with different automation curve settings. The parameter value range is arbitrary and can be configured to whatever model works best for the design. The automation data can just as easily scale from -5 to 5 as it can from 1 to 1000. The "Cricket-Extent" parameter in Figure 4 controls how wide of a surround pan range to use, so I set the parameter range from 0 to 360 to represent degrees on a circle. Parameters are an extremely powerful tool to dynamically modify the sound behavior and processing.

In addition to controlling instrument or track properties, parameters can control the playhead position on the event timeline by enabling and disabling logic markers. Logic markers are similar to markers on a DAW. They can define tempo, meter and rhythm, loop points, and timecode labels.

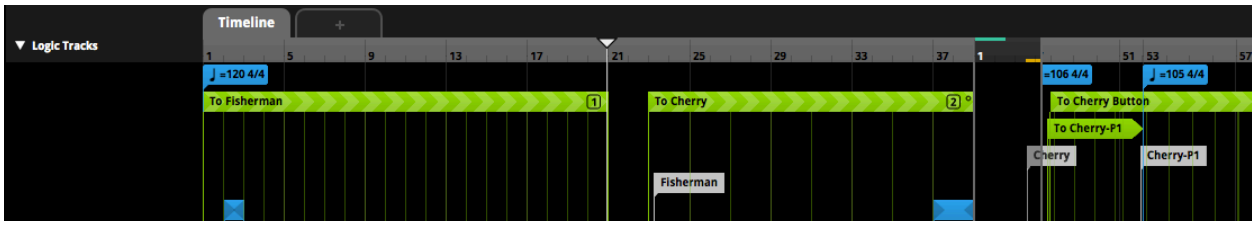


Figure 5. Logic Tracks including Tempo, Destination Markers, Transition, and Loop Regions.

There also are some specialty non-linear markers as well. Transition markers make the playback position jump to a specified destination marker elsewhere in the event. Transition regions also cause the playback position to jump to a destination marker, but specifies a range on the timeline where a transition is allowed—instead of just one specific point in time.

A transition region can be configured to permit the playhead to jump instantaneously or the jump can be quantized to happen on specific bars and beats within the region. In Figure 6, the transition region is only enabled when the "Section" parameter is between 2 and 10. It is also quantized to only place transition points every 2 bars.

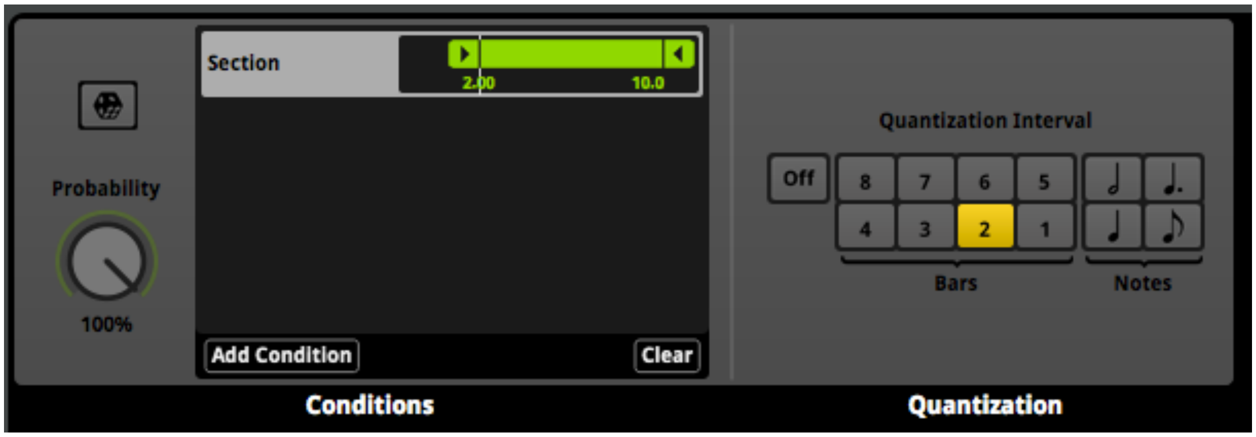


Figure 6. The Deck Inspector Focused on a transition region.

Transition markers have an optional element called the transition timeline. This is a special timeline that can be dynamically inserted between the transition maker and destination marker. It is used to bridge the transition by adding additional content, automation, or crossfades.

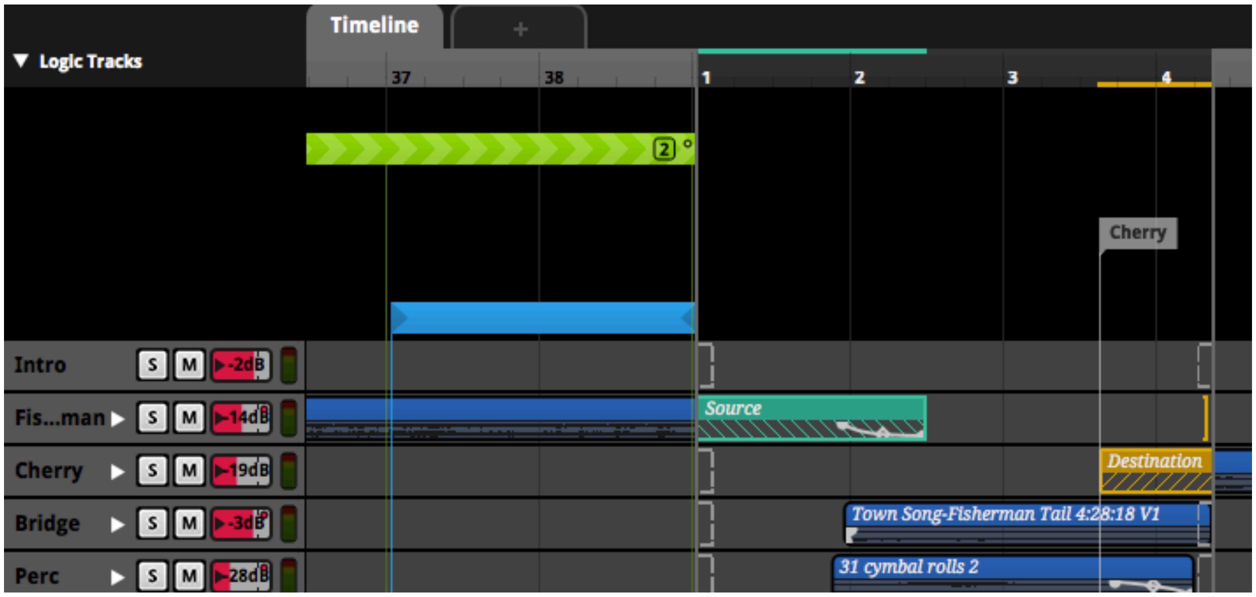


Figure 7. Four Bar Transition Timeline.

The last type of logic marker is a sustain point. Sustain points are especially useful for playing asynchronous instruments such as the scatterer instrument. When the playhead reaches a sustain point, it remains at that set time until the sustain is released by another FMOD trigger.

Sustain points are useful because an FMOD instrument is only active when the playhead is over the instrument's region on the timeline.

In a typical linear DAW such as Logic or Pro Tools, the playhead reads the sample data synchronously. The same is true for FMOD single instruments (instruments that only contain one sample file). A sustain point over a synchronous instrument will sound like the track is paused. Scatterer instruments are asynchronous instruments in that they automatically continue to re-trigger as long as they are active. If there was an exact amount of time the scatterer instrument should run, its region could be stretched to fill that specific amount of time on the timeline. However, if the amount of time desired is variable or undetermined, a sustain point can be used to hold the playhead over the instrument region until the instrument is stopped.

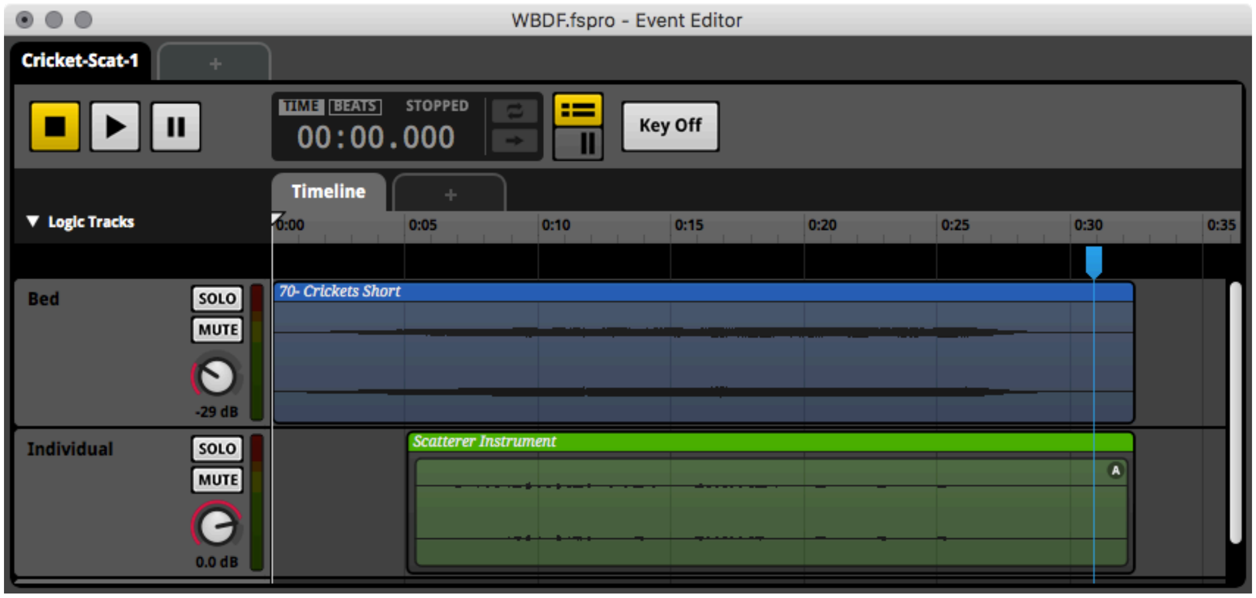


Figure 8. Event with a synchronous single instrument and an asynchronous scatterer instrument.

Figure 8 is an example of how synchronous and asynchronous instruments can be used in tandem. When the event starts, the cricket bed single instrument begins to play. After 5 seconds, the individual cricket scatterer instrument begins to play. The sustain point at the 31 second mark pauses the cricket bed (at this point the sound file is already silent), but the scatterer instrument continues to play. The scatterer instrument will stop one second after the sustain point is released.

To summarize, FMOD is an audio engine and API that is designed to be integrated into other programs. FMOD Studio is the editor application used to create FMOD project content. An FMOD project contains multiple events, each with their own independent timeline and tracks. Instruments are placed on tracks as regions and contain references to audio files. The event's audio signals originate from the instruments, route through the tracks, and are combined on the event's master bus. The audio signal can be altered by configuring a wide variety of track, instrument, or effect property values. The property values can be controlled by modulation or with external parameter data. Finally, the event's playback position can be manipulated through the use of logic markers on the timeline.

Implementation

The previous section explained how to create sound cue variations with the scatterer instrument, how to control event properties with automation and modulation, and how to skip around in the timeline with logic tracks. Using these techniques, I will explain how I created responsive sound and music cues for *Wood Boy Dog Fish*.

One scene in the play called for a single cricket to emanate from a specific location on stage, then grow in intensity until there was a field of crickets chirping all through the audience. To achieve this effect, I created a scatterer instrument with a parameter I called "Cricket-Extent" (see Figure 2 in the previous section for a screenshot of the instrument settings). I mapped the

Cricket-Extent parameter to three properties of the instrument: spawn rate (how quickly the instrument re-triggers), surround extent (how wide the sonic field is), and volume. When the Cricket-Extent value is 0, a cricket sound plays every ten to twenty seconds quietly out of one speaker. As Cricket-Extent grows to 360, each of the automated property values increases as established by their automation curves. The cricket sounds play more frequently, louder, and from a wider range of positions in the room.

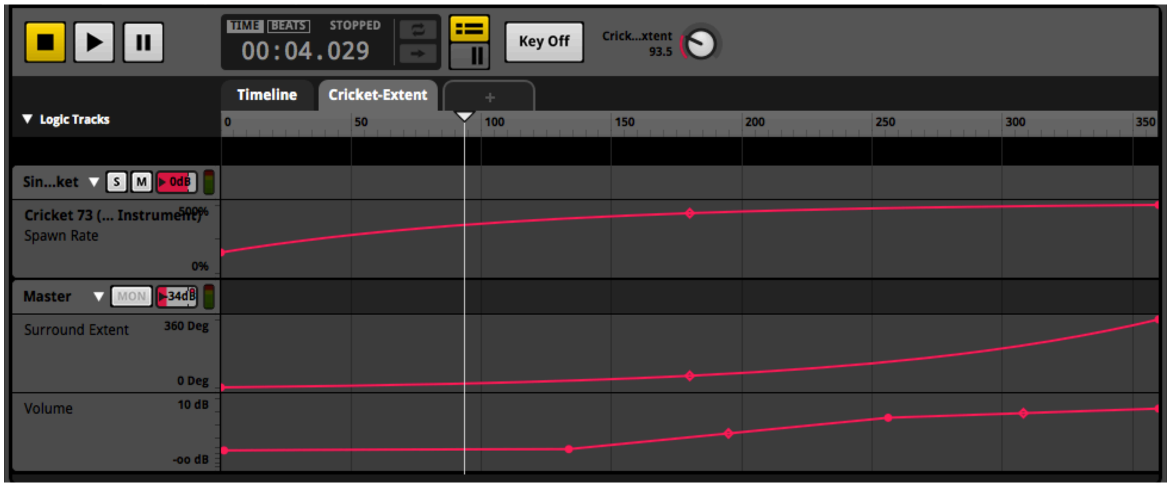


Figure 9. Cricket-Extent mapped to three properties: Spawn Rate, Surround Extent, and Volume.

Another scene features a power struggle between three characters: Gepetto, Blue, and the Dogfish. Geppetto is haunted by the sound of Blue's voice. Blue is trying to communicate with him, but he only hears jumbled fragments. As she tries harder, the voices in his head grow more intense and frequent. Finally, the Dogfish breaks in and pulls Blue back into the depths of the afterlife. She loses her voice and everything suddenly becomes silent.

To achieve this effect, I again used a scatterer instrument with snippets of Blue's dialog. I bussed the output to 6 sends, each with their own delay, pitch shift, and bandpass. On the master track, I processed the returns through a compressor, reverb, eq, and distortion effect. I used two parameters: Blue-Voices which controlled the scatterer instrument's spawn rate, and Blue-Static which controlled the level of distortion and effects level.

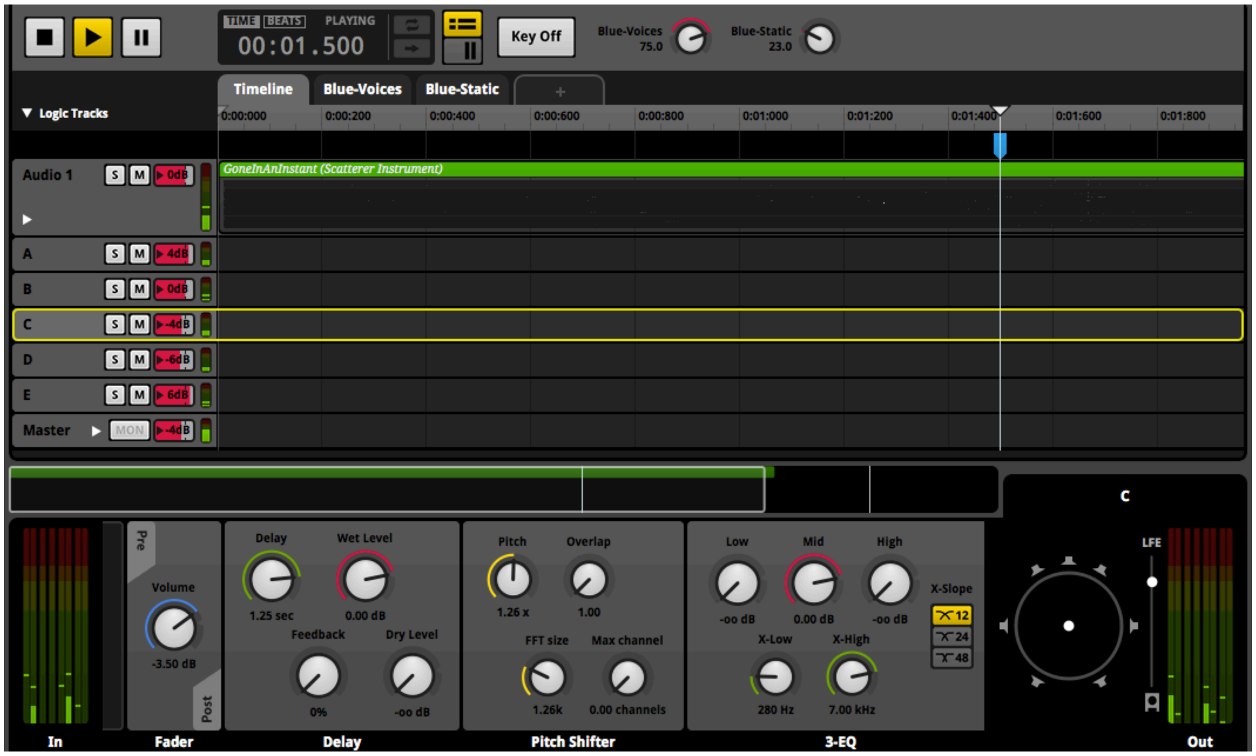


Figure 10. Send Track Signal Chain.

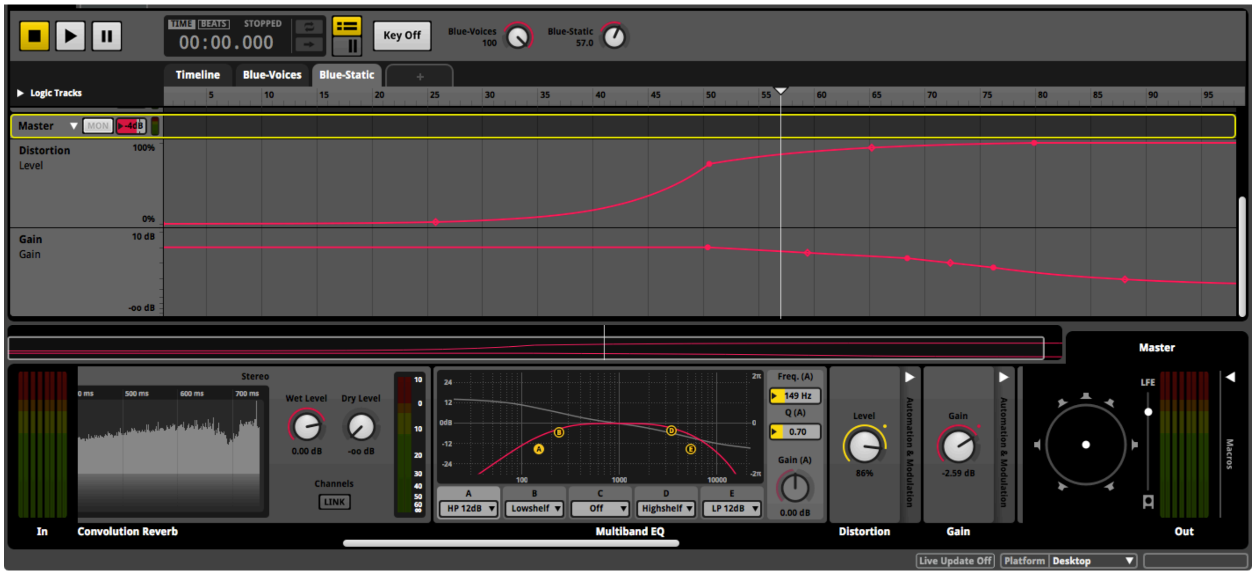


Figure 11. Automation Data for the Blue-Static Parameter.

As seen in Figure 11, the distortion effect ramps up sharply around a Blue-Static value of 40, then levels off around 70. Between this range, the recorded dialog becomes very garbled but it is still recognizable as spoken-word. Beyond 70, the distortion effect begins to only pass noise, and the gain is reduced to deliver a faint static ambience.

In this example, the Blue-Static parameter had no effect on the settings from range 0-20; however, I found it convenient to default to a 0-100 range for most parameters so I could think of them as a percentage. In this case, I wanted a clean signal from 0-20%, distortion to gradually ramp up between 20% to 40%, then rapidly increase to just shy of full distortion at the 50% parameter mark, and finally increase slowly to complete distortion across Blue-Static values 50% to 100%—a nice wide range to also use for a gradual gain reduction.

All the music cues were implemented in FMOD as well. *Wood Boy Dog Fish* had six major character songs throughout the story along with various scene transition cues. Each song was given its own event in FMOD. Figure 12 is a screenshot of the event "TownSong" which contains the material for the show's opening song "Welcome to Shoreside." The song's choruses were separated by sections of dialog, so I created a "Section" parameter in FMOD to control which part of the song or underscore to play at any given moment during the scene.

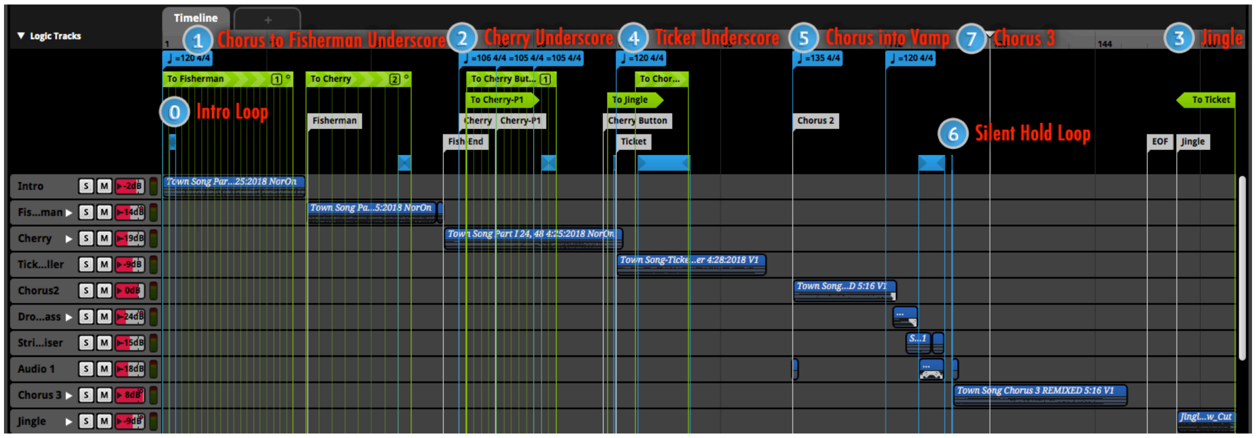


Figure 12. TownSong Section Parameter & Condition Values

Each section of the song was given its own index value. There were eight sections, so the parameter value range went from 0 to 7:

0 - Intro Loop. The event would start playing from the start of the timeline and vamp the first loop region.

1 - Chorus 1 to Fisherman Underscore. When the Section parameter was set to 1, the Intro Loop would devamp and the song would linearly play Chorus 1 into the fisherman underscore until it reached the loop region at the end of the Fisherman region. The "To Fisherman" transition region was only used during tech if we wanted to skip ahead in the song.

2 - Cherry Underscore. The playhead would jump from the "To Cherry" transition region to the "Cherry" destination marker. There was an additional optional cue "2.5" to skip part of the underscore music for Cherry if the scene was running fast—it enabled the "To Cherry-P1" transition marker.

3 - Jingle. The playhead would to play the button on the Cherry Underscore and then jump to play a commercial jingle that interrupts the song. At the end of the Jingle, the playhead jumps to the Ticket Underscore section. By placing the Jingle section at the end of the timeline instead of in order, I was able to keep the song segments located at the same bar numbers as Prévost had in his Logic session. He had originally written this song as one linear track, and it allowed us to easily discuss which bars we should be vamping and using as transition points.

4 - Ticket Underscore. This section always played after the Jingle, so no parameter conditions were needed for the run; however, it was a useful jump point to have during tech.

5 - Chorus 2 into Vamp. This would devamp the loop in the Ticket Underscore section, trigger a needle scratch gag, jump to Chorus 2, and play straight through until the vamp at the end of the second chorus. There was no quantization defined on the “To Chorus 2” transition region since the needle scratch could happen at any time in the song.

6 - Silent Hold. This would devamp the song into a very short silent loop. Essentially, this functioned as a fade out and pause.

7 - Chorus 3. The event would play the rest of the song. When the playhead reaches the “EOF” marker, FMOD stops the event and unloads the audio from memory. This is not a feature built into FMOD, but part of the scripts I developed to trigger FMOD events from Unity.

Most of these transitions were relatively straightforward. They were either immediate jumps or quantized to a bar or phrase. The most complex transition was from Section 1 to 2 (from Fisherman into Cherry). The Fisherman underscore was at 120 BPM while Cherry was at 106 BPM. Prévost and I created a transition timeline to bridge the sections.

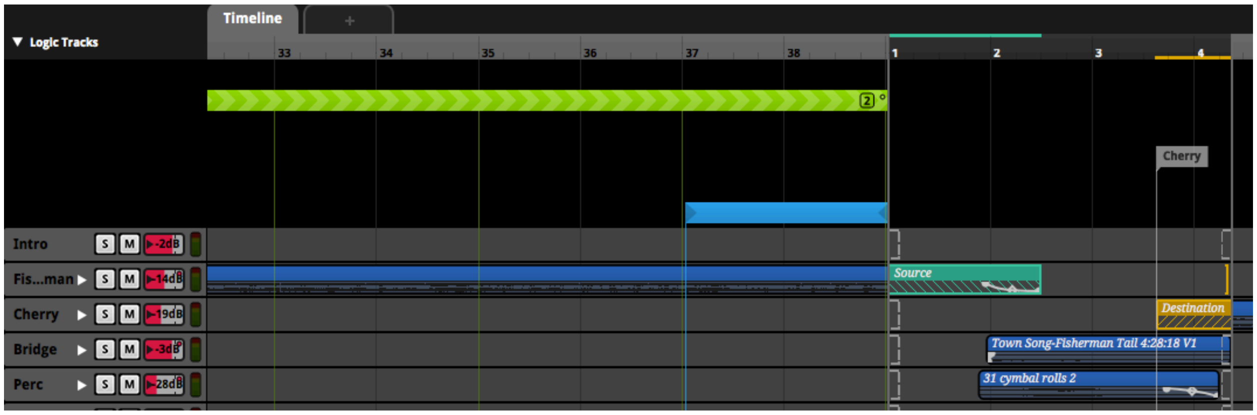


Figure 13. Transition Timeline from Fisherman to Cherry

The bridge emulates a 4 bar transition. We let the Fisherman music continue playing for one more bar, then crossfade to a 4 chord ritardando and cymbal role, play a pickup into the Cherry theme, and finally the playhead is finally back on the main timeline.

Like “Welcome to Shoreside,” our climactic scene has many different musical parts that needed to line up to action on stage. The scene takes place in a theme park dark ride and is through-scored by an epic piece of music with a different musical theme for each segment of the dark ride. The set, prop, and puppet changes for each ride segment were incredibly complex and difficult to execute. If each ride segment performed as intended, each went by very quickly; however, if there were any snags, a segment could run very long and needed additional music.

Similar to the approach with TownSong, I worked with Prévost to set up various loop points and transition regions in the track. Again, a parameter is set up to control which section the song should head to next. In this instance, the song could both run ahead and behind the stage action, so there were transition options configured to either branch out to bonus loops or skip musical phrases depending on how fast or slow the scene was running.

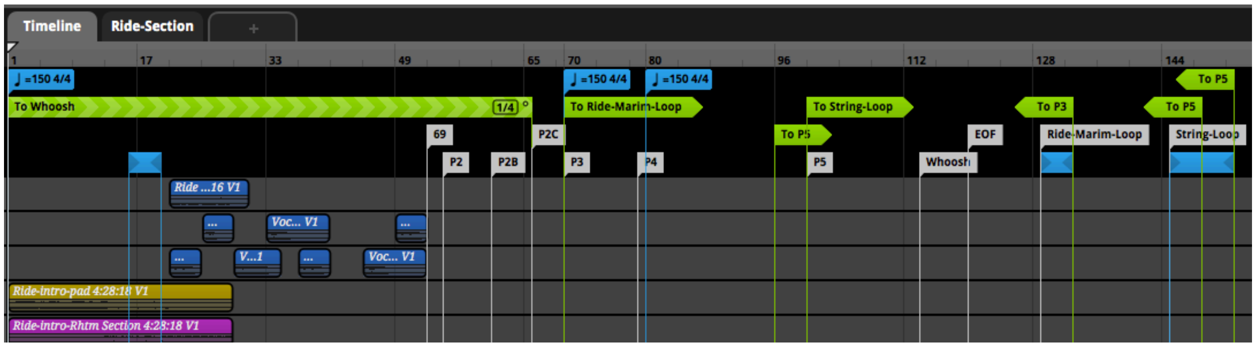


Figure 14. Logic Tracks for the Dark Ride Song

This song also played somberly during some foreshadowing in Act I. Prévost and I came up with a list of stems to mute for this moment, and he provided some additional musical stabs to play instead (tracks 2 and 3 in Figure 14). We created a parameter to toggle between the event’s underscore version and the finale version. Finally, the underscore needed to button out, so I created a “Ride-Whoosh” parameter that when set to 1.0 would activate the “To Whoosh” transition region (measures 1 to 65 in Figure 14).

This song had 24 stems and 6 sections. It is better to have one master event that is used multiple times in different configurations than to have multiple events that are slightly different from one another. If Prévost updated a stem, there was only one place where it needed to be replaced. If we wanted to tweak the balance, we did not have to copy the settings across multiple events. Similarly, many of the transition cues were variations on a theme, so I created one FMOD event per theme and used parameter settings to modify the event configuration for each variation. This minimized the number of changes we needed to track across the show file when audio clips were replaced.

I also greatly appreciated being able to toggle timelines between minutes and seconds to beats and bars. If I had a question about where to place a transition or loop, we could talk in terms of beats and bars instead of time. Prévost and I were both very impressed with the quality of loops and cuts in FMOD. We never experienced any clicks when making our transition jumps, and the quantization always felt in time.

Not all sound cues were run from FMOD. Some cues were simple fire and forget cues which were easier to play in QLab. Some cues were re-used from an earlier workshop—I only programmed cues in FMOD when it served the design. FMOD excelled when arranging music cues, creating long atmospheric beds, drawing complex surround pans, and manipulating audio effects in real-time. It helped me avoid bouncing long soundscapes, creating long unintuitive fade cue stacks, or bouncing out multiple wet/dry audio stems.

FMOD was painful to use when fighting against its assumptions about how sound should mix and route. FMOD's primary market is video games, and it is designed to work with common home audio sound formats (mono, stereo, quadraphonic, 5.1, and 7.1). I was fortunate that our theater setup was a 7.1 system, so it was easy to translate FMOD's panner to the room. However, I could not use FMOD for stage effect speakers since FMOD was limited to 8 channels. It would also be difficult to position sound emitters since those speaker positions are outside of the 7.1 surround field. It was also difficult to route sound to non-adjacent speakers. FMOD for Unity assumes sound events can only exist at one 3D point at any given time. Theatrical sound designers sometimes want to play with an audience's sonic perception by having one sound come from two disparate locations. There are methods to get around Unity's positional limitations such as creating a secondary bus to route to a specific output, but it is not as fast or flexible as setting levels on multiple faders in QLab.

Another pain point was using distance to control volume. This feature was helpful when I needed to create complex pans and could create them simply by moving an object around a 2D plane, but the majority of the cues just needed to build, establish, and fade. Translating Unity's distances into decibels was more trial and error than I had time for. For the next version, I am planning on developing an override that ignores distance and exposes the volume property for each track instead.

Besides improving volume control, reducing the number of steps required to link FMOD event parameters to OSC would have the most impact in making FMOD easier to use in a theatrical setting. The integration prototype I developed requires a fair amount of overhead work to translate an FMOD event into a cue that can be executed by a stage manager or operator.

Integration

We decided early in the process that the majority of the show was still going to be built and run off of QLab. There were many sound cues already built that did not need to be recreated in FMOD. QLab was also the interface our stage management team was most familiar with. We programmed one master workspace to trigger sound, video, and lighting cues. It made sense for QLab to trigger Unity/FMOD cues as well.

I used the open source UnityOSC package from Thomas Fredericks[2] to add OSC support to my Unity project. The OSC address pattern I came up with used the FMOD Event Name as the OSC Container and the FMOD Parameter Name as the OSC Method:

```
“/EventName/ParameterName value”
```

Since FMOD parameter values are floating point numbers, I decided to only put values that could be interpreted as floats in the OSC arguments. However, QLab passed these arguments to Unity as string data; the arguments needed to be converted in a Unity script handler before using them in the FMOD API.

There were a few OSC Method names I wanted to reserve: Play, Pos, & Volume. These would not be used as FMOD parameter names, but instead have special functionality. Sending `“/EventName/Play 1”` would start the FMOD event; sending 0 would stop the event. Pos expects two floats from -10.0 to 10.0 that represent x,y coordinates on a grid. Sending positional data to an event would move the event emitter around in the Unity game engine. FMOD would interpret the geometric relationship between the audio listener and sound emitter and route the

[2] https://github.com/thomasfredericks/UnityOSC

audio signal to the appropriate channels. Finally, Volume was used to set rough levels. FMOD automatically attenuates the sound based on the distance the emitter is from the listener, but this volume control was a quick way to dial in an appropriate range for a sound event—similar to setting trim on a mixing console.

QLab OSC cues can send OSC messages over a duration of time by configuring the cue's fade settings in the inspector. Parameter changes were sent as 1D fades to transition an effect over time. Positional data was sent as 2D fades. Drawing surround pans in QLab's 2D fade coordinate interface was very intuitive and effective. I recommend configuring the OSC fade cues to send data as integers in order to limit the number of messages sent to Unity. When QLab was sending fade data as floats, I discovered that the OSC handler could not keep up with the stream of data. Most of my parameters had a value range of 0 to 100, and I found 100 steps gave me enough precision and control over the automation I was manipulating.

In addition to OSC, I designed a crude user interface in Unity (Figure 15) that allowed me to quickly experiment with configuration values during tech and also served as feedback confirmation that the OSC cues were firing correctly from QLab. The interface also had tools for speaker check, stopping all cues, and global mute—all of which could also be triggered via OSC.

The left half of the screen was dedicated to a panel of all the FMOD Events programmed. The right half of the screen has a visual guide of the sound event positional data. The speaker objects are for reference only and have no effect on the audio signal chain. To make channel assignments, Unity uses the Configure Speakers tool in the Audio MIDI Setup application to map the surround sound channels to an audio interface's output channels (Figure 16).

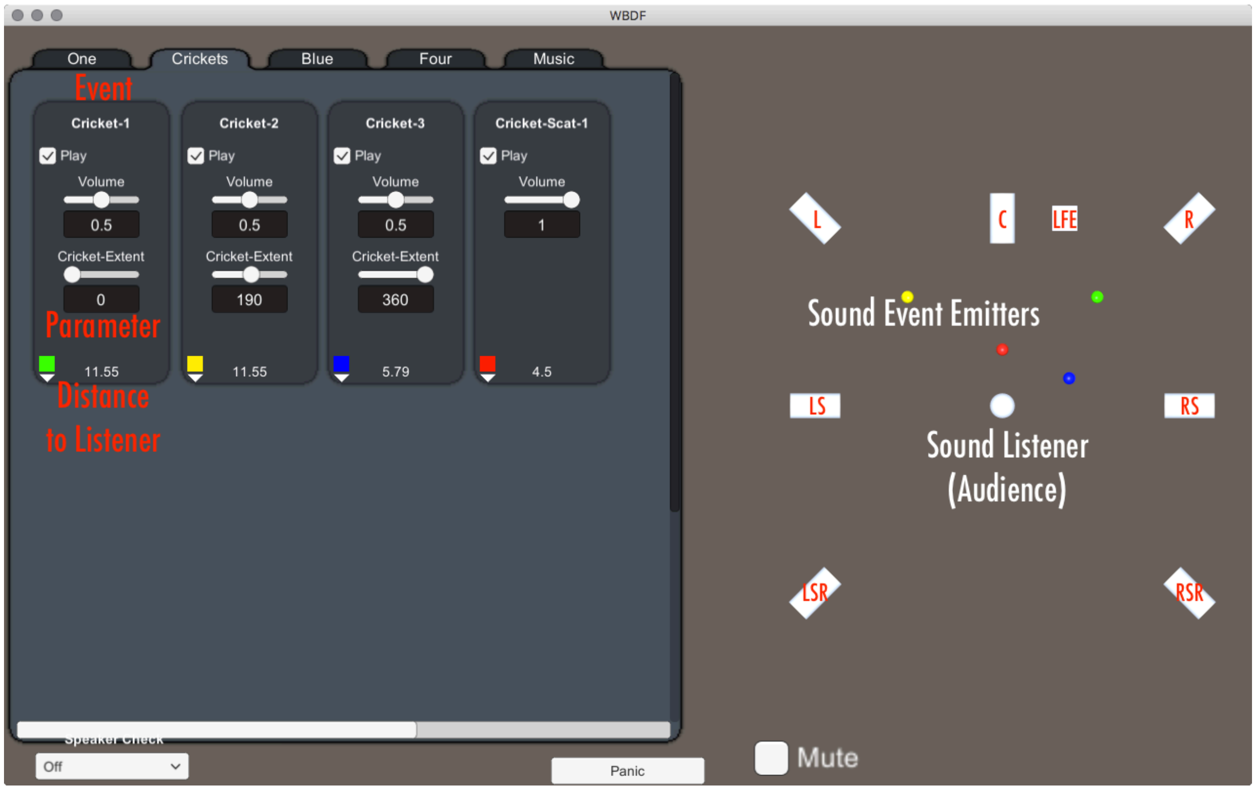
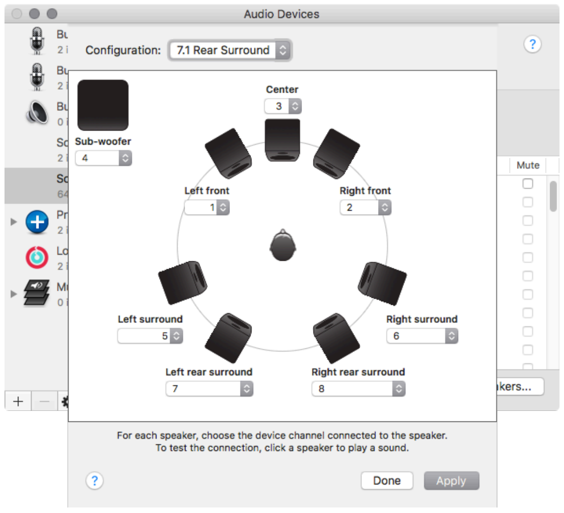


Figure 15. Unity Game Window that ran alongside QLab.



For each speaker, choose the device channel connected to the speaker.
To test the connection, click a speaker to play a sound.

Figure 16. Speaker configuration window in Audio MIDI Setup

Sound designers for video games are accustomed to working with resource-limited game consoles, and FMOD will compress the audio assets if desired. However, it can also build high fidelity PCM 48 kHz sound banks as well. Using 48 kHz 16-bit PCM settings, I did not encounter any performance issues running QLab and the Unity build simultaneously on a 2008 Mac Pro with 16 GB of RAM.

One of the benefits of using FMOD and Unity is that the source sound files are compiled into binary format and cannot be easily be pirated. A downside is that it takes time to compile the sound banks and Unity application. Both programs have tools built in to support rapid iteration. The Unity editor has a play mode and FMOD Studio has a live update mode. When enabled, event banks would automatically sync when switching between apps. In tech, I had FMOD Studio, Unity, and QLab open on the playback machine to edit cues. Making changes to audio content was reasonably quick and easy since FMOD was essentially my DAW; there was no bouncing and file transferring required.

Another benefit FMOD provided was integration with version control tools. Both FMOD and Unity have native support for Perforce. The Perforce server can run on a local or a remote machine. This was my first time using Perforce, and although I was familiar with other versioning tools such as git, I found using the Perforce tools unintuitive and confusing at times. The hardest part was getting the server set up on my machine. However, once the server and repository were working, committing changes and reverting back from FMOD was simple and reassuring. Version control should be a fundamental part of any design workflow.

The most time-consuming aspect of the workflow was creating the event objects and GUI in Unity. Unity was the right tool for this show since it allowed me to quickly test the feasibility of playing FMOD events in a theatrical setting. I wanted to focus on building sound cues, not developing software. However, my hastily made integration tool was also the biggest bottleneck. The current workflow involves setting parameter automation in FMOD, adding the hooks for the parameters in Unity, and then writing the OSC messages in QLab. Since creating the Unity hooks and OSC cues are very boilerplate once the parameters are defined in FMOD, these steps should be automated to streamline the workflow.

A final integration note is that a license is required to use FMOD in a production. FMOD's licensing system is intended to be used by game producers. Games with budgets less than $500,000 can use FMOD free provided they display the FMOD logo. Companies are limited to one free FMOD license a year; after that, there is an additional $2,000 fee. For non-game usage, email the FMOD sales team[3], explain how it is being used, and provide company, production, contact, and budget information. For our production, I explained we were a six week run operating under an LA 99-Seat Theater Agreement with a budget less than $500,000. On the next business day, we were granted a free license provided we credited FMOD and displayed their logo in our program.

Conclusion

FMOD is a very powerful tool, and this case study only scratches the surface of its capabilities. There are so many other dynamic effects that can be designed in FMOD. Some examples include a racing heartbeat (map intensity to spawn rate, volume, equalization, and

[3] https://fmod.com/licensing#faq

different sets of samples), an ambulance pass by (a looping siren instrument with parameter mapped to volume, pitch, eq, and perhaps distortion), understudy cues, scatterer instrument background ambiences, and many more possibilities.

For a designer that is not interested at all in writing code—a current requirement to implement FMOD in an application—they may consider using FMOD as a sound creation tool and recording the output into their DAW. For instance, using a handful of traffic samples and FMOD's scatterer tool, a designer could create a unique 15-minute city ambience in a couple of minutes. They would not get the advantage of FMOD's adaptive real-time playback engine, but it is an incredible design tool compared to layering and editing the various samples manually in a traditional DAW.

The most revealing personal assessment of FMOD's value to me and Prévost was that on our next show I decided I did not have time to program in FMOD, and throughout the process, Prévost lamented that we were not using it. I had been interested in experimenting with FMOD for a while, and *Wood Boy Dog Fish* was the perfect opportunity because a lot of the show was already built, and we had the privilege of focusing on refining the design. It is still more efficient to use traditional playback engines such as QLab. Each event from FMOD required some amount of overhead work to translate it into a cue that could be triggered by a stage manager or operator.

Using FMOD as a design tool opened new paths of creativity for me to explore. I really enjoyed working in FMOD Studio. The DAW paradigm provided a gentle learning curve, it was easy to dial in the sound I was looking for with the built-in processing effects, and I found defining variables to control a group of complex audio properties both intuitive and effective.

Now that I understand how FMOD's features can enhance my designs, I want to invest more time in developing a custom application that has better user interface tools for rapid cue creation.

It was only when I needed to create an interface to control and trigger FMOD that I begin to feel like I was doing more busywork than designing. I wish there was a way to control the FMOD Studio interface directly from OSC or MIDI—similar to how the Ableton Live interface can be mapped. There is support for Mackie and SSL control surfaces, but I did not have any equipment to test it with.

I continue to be enthusiastic about building sonic environments procedurally, especially highly dynamic ones that could really benefit from riding the design parameters live throughout a scene. I believe the next step is to create an FMOD integration tool that can be programmed as quickly as putting together a QLab workspace. In the meantime, I have published my Unity integration tool[4] for anyone interested in experimenting with FMOD and OSC. I also encourage everyone to download FMOD Studio[5] and play with the examples. The examples clearly demonstrate a broad array of FMOD's features and provided me with the inspiration to use it in a show.

[4] https://github.com/swiftster/fmod-theater

[5] https://fmod.com/download