4-22-2013

Using Procedural Audio to Control an Algorithmic Composition that is Controlled by a Computer Game

Brian del Toro
bdelt001@fiu.edu

DOI: 10.25148/etd.FI13080501

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

USING PROCEDURAL AUDIO TO CREATE AN ALGORITHMIC COMPOSITION THAT IS CONTROLLED BY A COMPUTER GAME

A thesis submitted in partial fulfillment of

the requirements for the degree of

MASTER OF MUSIC

by

Brian del Toro

2013

To: Dean Brian Schriner
College of Architecture and the Arts

This dissertation, written by Brian del Toro, and entitled Using Procedural Audio to Control an Algorithmic Composition that is Controlled by a Computer Game, having been appointed in respect to style and intellectual content, is referred to you for judgment.

We have read this dissertation and recommend that it be approved.

_____
Orlando Garcia

_____
James Webb

_____
Jacob Sudol, Major Professor

Date of Defense: April 22, 2013

The dissertation of Brian del Toro is approved.

_____
Dean Brian Schriner
College of Architecture and the Arts

_____
Dean Lakshmi N. Reddi
University Graduate School

Florida International University, 2013

DEDICATIONS

To Amy, the girl I love.

To my grandfather William, who always said, "don't work too hard!"

To my parents Catherine and Jorge who let me stay up, as late as I wanted, playing video games.

ABSTRACT OF THE THESIS

USING PROCEDURAL AUDIO TO CREATE AN ALGORITHMIC COMPOSITION THAT IS CONTROLLED BY A COMPUTER GAME

by

Brian Del Toro

Florida International University, 2013

Miami, Florida

Professor Jacob Sudol, Major Professor

The purpose of this project is to create a game audio engine based on procedural audio. I designed an audio engine in the visual programming language Max/MSP that I call *High Score*. *High Score* receives data from a game that I designed in the game development program Unity3D. The data that it receives controls an algorithmic composition that serves as the musical score of the game as well as several processes that synthesize various sound effects in the game. This approach to game audio proves to be very flexible and offers new aesthetic possibilities for game music and sound design.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE PAGE

I. Introduction

An audio engine is a program responsible for triggering sound effects caused by events in a game, changing music based on game states, and even changing effect parameters or signal routing chains according to player input. In most current audio engines, all of the sounds and music are pre-recorded files. Andrew Farnell refers to this as the "*data model* of game audio." [1] There are two main reasons that the data model is the prevailing approach to game audio. Recordings provide a high degree of realism that would otherwise be difficult to achieve procedurally and have a fixed computational "cost" in terms of memory and processing power.

My project stems from a less common approach: the synthesis of both sound effects and music based on real-time game parameters and player input. This approach is best described as procedural audio, where procedural is defined as: “Relating to or comprising memory or knowledge concerned with how to manipulate symbols, concepts, and rules to accomplish a task or solve a problem.” [2] Procedural methods offer new solutions to the creative, technical and economic problems that game audio faces, while also presenting a new aesthetic framework for generating music and sounds.

In my thesis, I will describe a procedural-based audio engine called *High Score* that I designed in Max/MSP. *High Score* communicates with a game that I designed using the game development program Unity3D. Data sent from the game to Max/MSP controls the parameters of an algorithmic composition and triggers sound effects synthesized in real time.

---

[1] Andrew Farnell, *Designing Sound* (Cambridge: MIT Press, 2010), 318.

[2] "Procedural." Merriam Webster Online Dictionary.2013.(accessed January 15, 2013)

II. History of Procedural Audio

Procedural audio precedes the current data driven model of game audio. An examination of early video game sound technology reveals a period of rapid innovation in the synthesis of music, sound effects and speech, fueled by the success of arcades and home video game consoles. A classic example of an early arcade game that linked gameplay to music is *Space Invaders* (Midway, 1978), which according to Karen Collins "set an important precedent for continuous music, with a descending four tone loop of marching alien feet that sped up as the game progressed." [3]

In the earliest game consoles and arcade terminals, Programmable Sound Generator (PSG) chips such as Bob Yannes' Sound Interface Device (SID), featured in the Commodore 64, synthesized sound effects and music in real time.[4] According to Collins, "Most PSGs were subtractive synthesis chips" which were used in both home and arcade consoles.[5] Subtractive synthesis is a method that employs filters to attenuate or remove specific frequency components of a timbrally rich sound. PSG chips were used well in to the mid-1990's, appearing in major video game consoles such as the NES and Super NES. However, other methods were also developed.

---

[3] Karen Collins, Game Sound: An Introduction to the History, Theory, and Practice of Video Game Music and Sound Design (Cambridge: MIT Press, 2008), 12.

[4] Karen Collins, "Loops and Bloops," *Soundscapes* vol.8 (February 2006): http://www.icce.rug.nl/~soundscapes/VOLUME08/Loops_and_bloops.shtml (accessed January 15, 2013).

[5] Collins, *Game Sound,* 10.

The major factor that drove the early procedural approach to game audio was the limit on memory available at the time. For example, 5.25" Floppy disks, a common format for Commodore 64 games, only provided 170 kilobyte (kB) of storage, of which music data synthesized by the Yannes' SID might normally occupy between 5 and 10 kB.[6]

In his article *The Synthesis of Complex Spectra by means of Frequency Modulation*, John Chowning first described FM synthesis as a method for the simulation of acoustic instruments.[7] The Sega Genesis, released in 1988, was a hugely successful home console that featured the YM2616, an FM synthesis chip by Yamaha as well as a simple digital sampler.[8] Composers and sound designers took advantage of the Genesis' capabilities, producing realistic sound effects and continuous music that eschewed the simple looped melodies of the previous generation of consoles in favor of more intricate scores.

The Rise of Data-Driven Audio

When the Nintendo Entertainment System (NES) made its debut in the United States in 1985, it boasted a "custom-made five-channel PSG chip." [9] Whereas the first four channels held synthesizer channels, the fifth channel was a primitive sampler that could play speech or recorded sound effects.[10] By 1995, digital audio was the predominating technology in game

[6] Collins, Loops and Bloops

[7] John Chowning, "The Synthesis of Complex Spectra by means of Frequency Modulation," *Journal of the Audio Engineering Society* 21, no.7 (September 1973)

[8] Collins, *Game Sound*, 40

[9] Collins, *Game Sound*, 25

[10] ibid, 25

audio. The Sega Saturn, Nintendo 64 and Sony Playstation consoles were all capable of playing increasingly realistic recorded sounds.

Since then, the data model has dominated the game audio industry. Nearly any modern computer available today, including video game consoles, desktops, laptops, tablets and mobile phones can play multiple channels of standard CD quality audio. There is no further need to point out the many advances in computer technology over the last twenty years that led to this situation. It suffices to say that there is currently little to no fundamental distinction between audio quality or production methods in computer games and that of any other medium such as television or film.

III. Current State of Procedural Audio

Two concepts central to games and game audio are *interactivity* and *nonlinearity*.[11] Collins defines interactive audio as "sound events that react to the player's direct input" and uses the term nonlinear to refer to "the fact that games provide many choices for players to make and that every gameplay will be different."[12]

Procedural audio precisely addresses these two issues in games. One of the things that procedural methods aim to achieve is an accurate model of the behavior of sounds. In the data model, the playback of a sound file corresponds to an event in a game. This often leads to simple and repetitive interactions between the player and the sound environment of the game. By designing a game such that the audio engine has access to more information about an event than merely when it happened, a higher level of interactivity and player immersion in the game can be achieved. Procedural audio also accounts for nonlinearity by creating models for sounds that respond to a wide range of player interactions.

[11] Collins, *Game Sound*, 3.

[12] ibid, 4.

Procedural Audio Advantages

Farnell lists five advantages to procedural audio: deferred form, default forms, variety, variable cost and dynamic level of audio detail.[13] Deferred form refers to an exploratory type of sound design where sound design is done inside the game, by adjusting the various parameters of a sound model according to player input. Default forms are one of the most attractive concepts in procedural audio: the idea that sound models that are derived from existing physics models that can generate acceptably realistic sounds by default, only requiring fine tuning from a sound designer once they are in place. Variety is perhaps the first and most obvious advantage. Procedurally generated sounds can be designed such that they never exactly repeat, much like no two acoustic sounds are ever exactly the same. Variable cost refers to the computational cost of performing a particular action. For example, two audio files of the same size and quality will always have the same computational cost. However, some procedural models are computationally more efficient, or "cheaper," than others. Dynamic level of audio detail refers to a process that analyzes what is perceptually relevant to the player. This process deactivates or modifies sound models as less detail is required, therefore taking advantage of the model's variable cost.

[13] Farnell, *Designing Sound* 321.

The Adoption of Procedural Audio

One of the major factors preventing the widespread use of procedural audio and algorithmic composition in games is the lack of integrated software. Game development software such as Unity3D offer limited controls over audio implementation. For example, the most common approach in Unity3D and other game development programs is to trigger the playback of a sound via an in-game event. In this case, music is simply started at the beginning of a level and programmed to loop endlessly. More robust audio "middleware" programs such as Wwise and FMod allow complex playback, basic randomization of sounds and some mapping of musical parameters to in-game data. These programs are intended to help sound designers create sophisticated audio engines without having to manually code them. What these middleware programs lack is any general platform for integrating procedural audio.

One company, Audiogaming, is pioneering the use of procedural audio with software that can be integrated into existing middleware programs. These procedural models synthesize a range of sounds within a particular category such as: motors, footsteps, gestures, wind, rain and fire.[14] However, by providing realistic synthesized physical models that can be recorded, they still facilitate the data-driven approach to sound design. Two other experiments in procedural audio include *Game Audio Framework* (GAF), developed by the CNAM/CEDRIC laboratory in Paris,[15] and Robert Hamilton's UDKOSC, that is currently being developed at the Center for

[14] www.audiokinetic.com

[15] Olivier Veneri, Stephane Gros and Stephane Natkin, "Procedural audio for Games with GAF," cedric.cnam.fr.http://cedric.cnam.fr/index.php/publis/article/view?id=1568 (accessed May 25, 2011)

Computer Research and Acoustics at Stanford.[16] GAF consists of a custom-built programming language called GAFScript that handles the creation of sound "objects" and communication between an audio engine and game. UDKOSC uses a message protocol called Open Sound Control (OSC) to send messages from the game development program Unreal Development Kit to any program that can receive OSC messages such as the audio programming environments Max/MSP, PD, or Super Collider.

[16] Robert Hamilton, "UDKOSC," CCRMA Wiki.https://ccrma.stanford.edu/wiki/UDKOSC (accessed September 9, 2012)

IV. Unity3D and Max/MSP

Unity3D is a game engine that allows the development of games for multiple platforms. I chose to build my game in Unity3D for a number of reasons. Most importantly, Unity3D has a simple workflow for creating, positioning and modifying game objects in a manner that requires minimal scripting experience. Unity3D users have access to a large library of free game assets such as textures, 3d models, animations, particle effects and sound effects. Creating the networking portion of the game, however, did require some C# scripting. This was done in Unity3D's Integrated Development Environment (IDE), Monodevelop (IDEs are computer programs that facilitate the writing, modification and execution of computer code). The integration of Unity3D and Monodevelop allows Unity3D's large library of assets to be modified with custom-written code, which was an important part of creating a communication network between the game and audio engine.

I based my game on an example called "Unity3D Island Demo", that contains all of the basic assets I knew would be required: terrain textures, plant and water objects as well as scripts for character motor control and "mouse look." Mouse look is simply a script that allows a player to look around by moving the mouse. My game embodies a typical computer game style commonly known as a "First Person Shooter." The player controls a character from a first person perspective and moves using the computer keyboard: W to move forward, A to step left, S to move backward, D to step right and Space Bar to jump. The mouse is used to look up and down, as well as to turn the character left or right. I also added a script allowing the player to shoot a cube as a primary means of interacting with the environment. **Figure 1** shows a screen shot from the game that I designed that communicates with *High Score*.

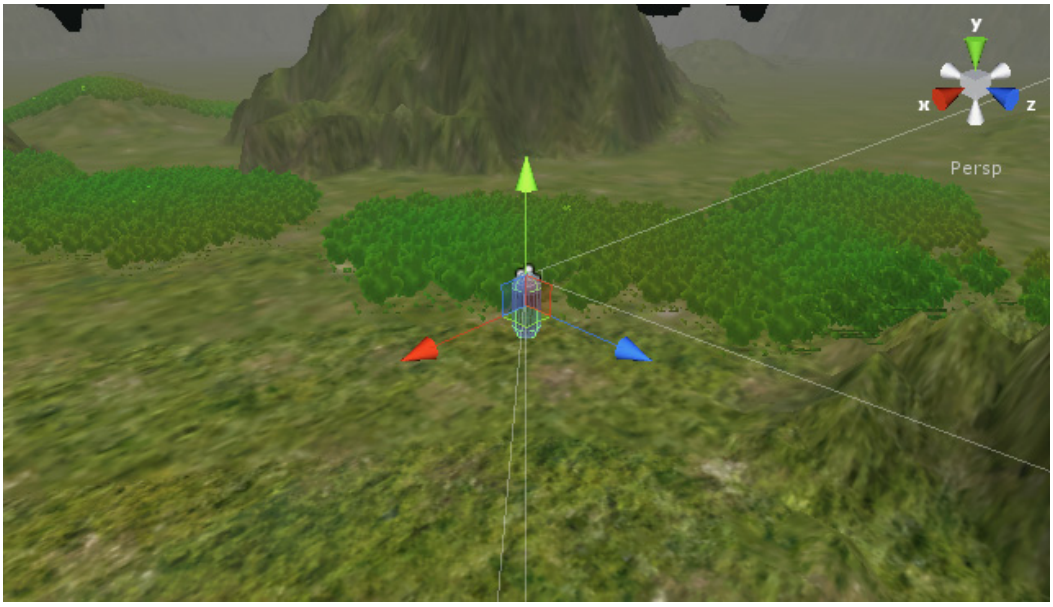

Figure 1 shows the icon representing the player in the game and its field of view.

Max/MSP

Max/MSP is a programming language specifically designed for dealing with audio. What separates Max/MSP from typical text-based programming languages is its visual interface. Programming in Max/MSP involves graphical "objects" which perform specific functions. These functions range from simple ones such as adding two numbers to complex ones such as managing multiple instances of entire programs written in Max/MSP. Programs are written by connecting these objects with lines referred to as "patch cords," and/ programs written in Max/MSP are therefore called "patches." "Sub-patches" are Max/MSP patches that are embedded inside other patches. A typical Max/MSP patch vaguely resembles a flowchart in which data generally moves from top to bottom. Max/MSP's functionality can be enhanced with user-made objects that are called "externals." For example, *High Score* uses an external called "sadam.updreciever" to receive data sent by Unity3D. "Sadam.udpReceiver" listens for

messages that are broadcast on the local network. This object was created by Adam Siska, and is freely available on his website as part of his "Sadam" collection of externals.[17]

Communication between the Game and Audio Engine

One of the major hurdles to implementing my audio engine was finding a way to send data from Unity3D to Max/MSP. I ultimately chose to send the data using a networking protocol known as User Datagram Protocol (UDP). UDP is a method of sending messages, known as *datagrams*, between computers or computer applications. David P. Reed first defined UDP in the form of a Request for Comments (RFC) in 1980.[18] RFCs are documents that, upon review from the Internet Engineering Task Force (IEFT), can be adopted as internet standards.

In Unity3D, a script titled "UDPNetworkmanager" sends data that it receives from other scripts within Unity3D, namely "UDPmove" and "Csharpshoot" (see appendix for C# scripts). UDPmove transmits the player's location on the game map as XYZ coordinates. Csharpshoot transmits the message "bullet" whenever the player shoots a cube. These scripts convert a string of text such as "bullet" in to a byte array. A byte array is a container for any kind of data, reduced to its binary form. UDPNetwork manager then broadcasts the byte array on the local network that is designated by the IP address 127.0.0.1, port 80. For easier separation of data streams, the data for turning the audio engine on and off is sent on port 81. An IP address is the

[17] Adam Siska, "Sadam Library," Hungarian Computer Music Foundation http://sadam.hu/en/software (accessed October 30, 2012)

[18] Jonathan Postel and David P.Reed."User Datagram Protocol" *Internet Engineering Task Force Request for Comments 768,* (August 1980)

address of a particular device on a network. By using the loopback address 127.0.0.1 a computer can send messages to itself. This method of sending datagrams from one application to another on the same computer allows Unity3D to control Max/MSP without having it integrated in to its own software architecture.

The game that I developed is a very simple one that serves only to demonstrate some of the possible applications of a procedural audio engine. In the game a single player can explore his or her nearby surroundings. As the player gets closer to sound sources such as water or fire, they will hear procedurally generated sounds emanating from them. The player can interact with these sound sources by throwing cubes at them. Exploring the map drives an algorithmic composition that reacts to the player's activity level and position on the map. A more detailed description of how *High Score* reacts to the player's actions is presented in Section V.

V. Audio Engine

High Score has two main components or "layers": the synthesis layer, where sound is actually produced, and the mapping layer, where data received from Unity3D is interpreted as control data for various sub-systems such as instruments and sound design modules. Following a detailed description of both layers I provide an explanation of the interaction between the player and audio engine.

Synthesis Layer

The synthesis layer is comprised of two distinct frequency modulation (FM) synthesizers named “Synth1” and “Synth2” that can both perform complex musical textures thanks to a system of managing polyphonic voices in Max/MSP. The built-in “poly~” object manages the instantiation and termination of multiple copies of the same sub-patch, which is crucial in designing a polyphonic synthesizer. As mentioned above, in Max/MSP a sub-patch is a collection of objects that have been "encapsulated" and visually represented as a single object.

Synth1 contains a poly~ sub-patch titled "FMpolysimple" (**Figure 2** below). FMpolysimple generates percussive sounds for the musical portion of the engine and any other sounds that can be synthesized from frequency modulation using sine waves only. FMpolysimple is a basic implementation of FM synthesis with four parameters: carrier frequency, harmonicity ratio, an envelope for changing the frequency modulation over time, and an amplitude envelope.

In FM synthesis, the frequency of a signal called the "carrier" is affected by a "modulator" signal. If the change in the modulator's frequency over time is slow enough for humans to perceive, it appears more or less as a form of vibrato, a rapid fluctuation in pitch. Once the modulating signal's frequency is high enough, the carrier's pitch ceases to sound like it is fluctuating and the carrier acquires a different timbre that corresponds to the frequency of the modulator. The frequency of the modulator is often calculated or defined by the harmonicity ratio when it is a multiple of the carrier frequency. The amplitude of the modulating signal is referred to as the depth of modulation.

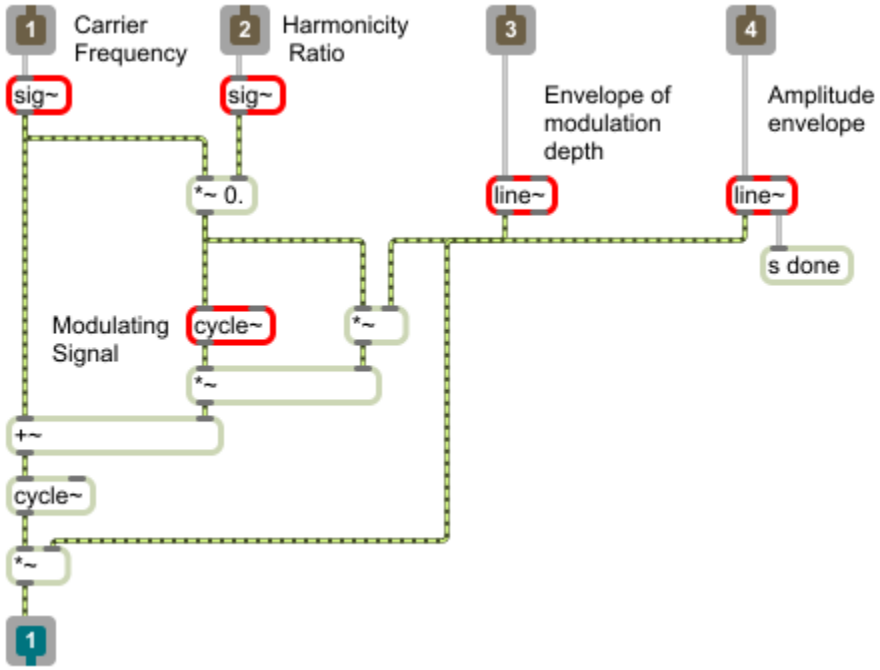

Figure 2 shows a basic FM model and its parameters, located within FMpolysimple

Inside of FMpolysimple, these parameters are received as one list of values that are separated into sub-lists – that correspond to the aforementioned envelopes – and single values that correspond to the carrier frequency and harmonicity ratio (**Figure 3** below).

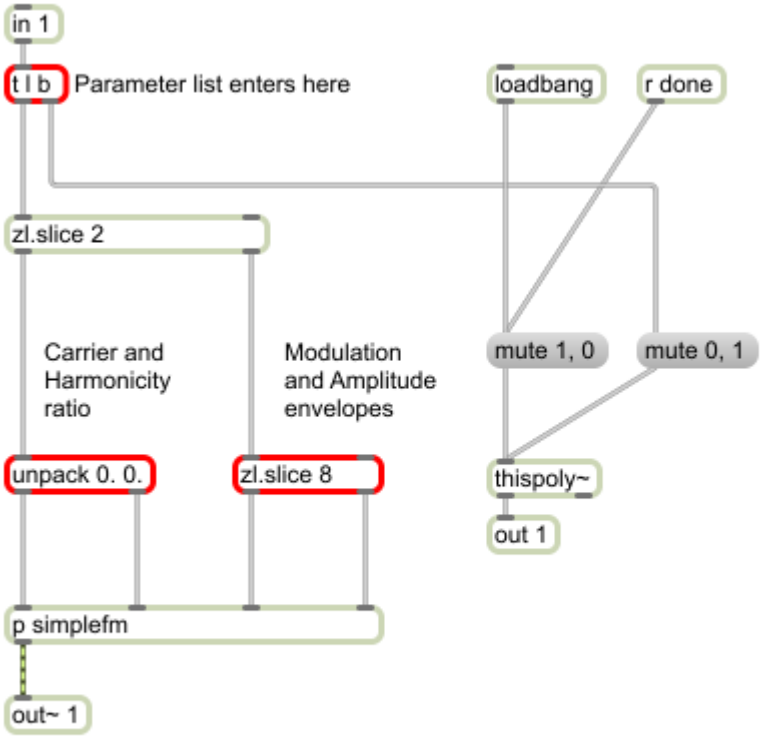

Figure 3 shows the section of FMpolysimple that receives and processes parameter lists

Parameters for this synthesizer are generated in another part of the main patch titled "FM1" that contains four different sub-instruments. While each sound is actually a single note in a strict sense, the repeated use of similar percussive timbres are limited to a certain range that approximates the behavior of a simple drum kit. The ranges for each instrument's parameters are located in a sub-patch named for the general sound that instrument makes. For example, the parameters for the lowest sounding instrument in FM1 are located in "boots" (**Figure 4** below). The other instruments are "cats", "tss", and "chk", respectively.

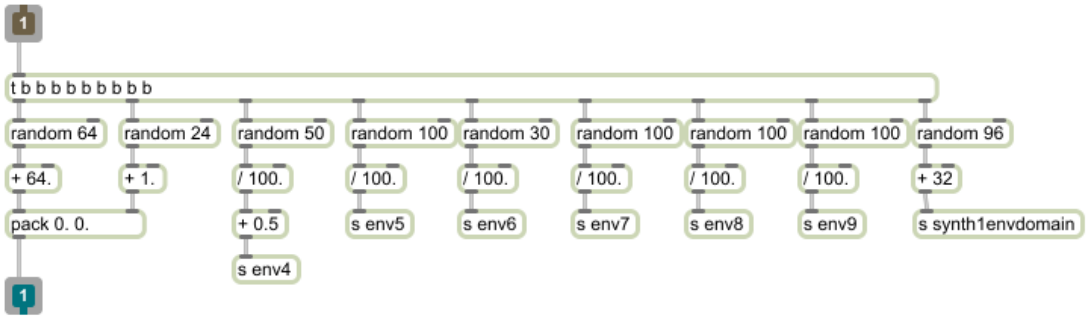

Figure 4 A list of random parameters is generated each time "boots" is activated. The parameters are randomized within a particular range so that each instrument maintains a unique identity while changing slightly with each "hit."

A process that lies closer to the mapping layer of the engine activates each instrument in FM1. A master clock called “transport” controls the timing of musical events in Max/MSP. When active, objects that are linked to the transport by the use of a special time notation are synchronized. In the mapping section for Synth1 (**Figure 5** below), metronomes that are linked to the transport activate the instruments discussed above.

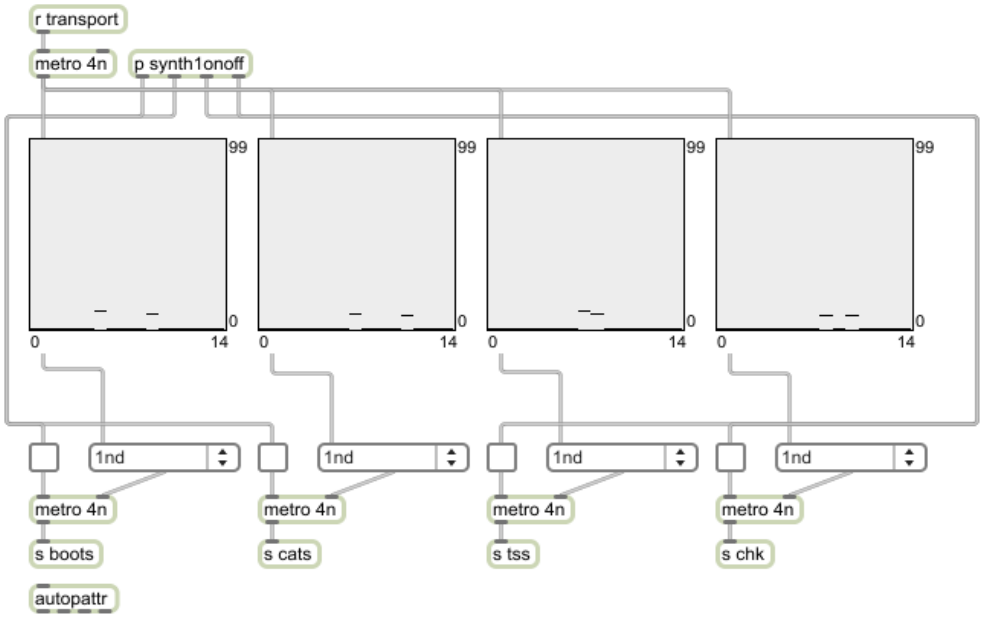

Figure 5 Each instrument in Synth1 is linked to a metronome, shown here in the sub-patch "Synth1map."

These metronomes can run at rates that correspond to subdivisions of the main pulse emitted by the transport. A random number that corresponds to a metronome rate is sent from a table on every beat of this main pulse. Each table is essentially a random number generator, where each point has an X and Y value. An X value is output from the table based on probability set by its corresponding Y value. This selection process drives the semi-random rhythm generator that activates the percussive instruments in Synth1. Finally, each instrument's metronome can be turned on and off via a similar table (**Figure 6** below). This higher-level control over the instrumentation is tied directly to the mapping layer of the audio engine.

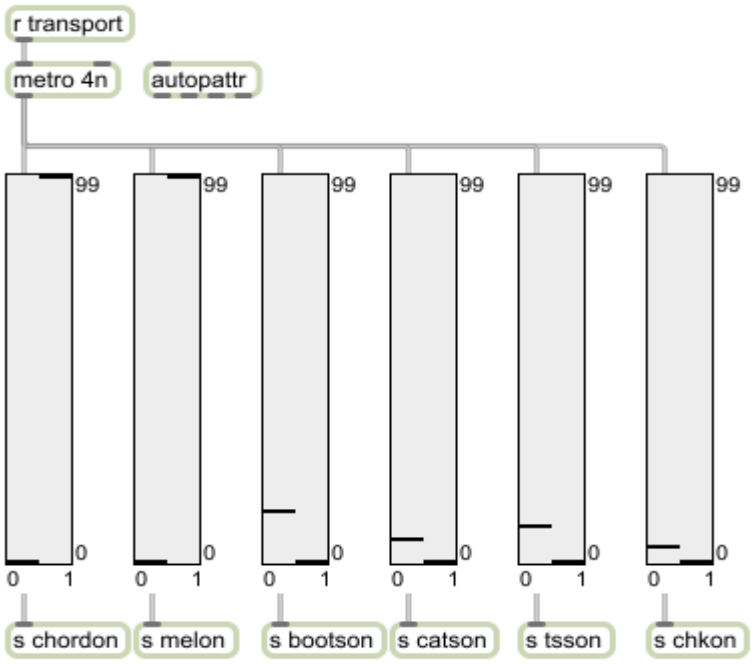

Figure 6 These tables store the probability values that determine whether or not instruments in the synthesis layer are active.

Synth2 is based on a similar FM synthesis model as Synth1 with some modifications. Synth2 contains a poly~ sub-patch titled "FMpolylegato". At the heart of FMpolylegato, and distinct from the simpler form of Synth1 is an oscillator that interpolates between a triangle and rectangle waveform (**Figure 7** below). This oscillator functions by summing a triangle waveform with the difference of triangle and rectangle waveforms. This oscillator immediately gives Synth2 a distinct sound from Synth1 and allows far more control in shaping timbre and, to the author's knowledge, is a completely novel way of generating FM tones. Two independently controlled oscillators with this interpolation feature serve as the carrier and modulating signal. The interpolation is controlled by a value ranging from 0 to 1, where 0 outputs a triangle wave, 1 outputs a rectangle wave and any value in-between outputs an interpolated waveform, such that 0.5 would be a waveform which is half way between the shape of a triangle and rectangle wave.

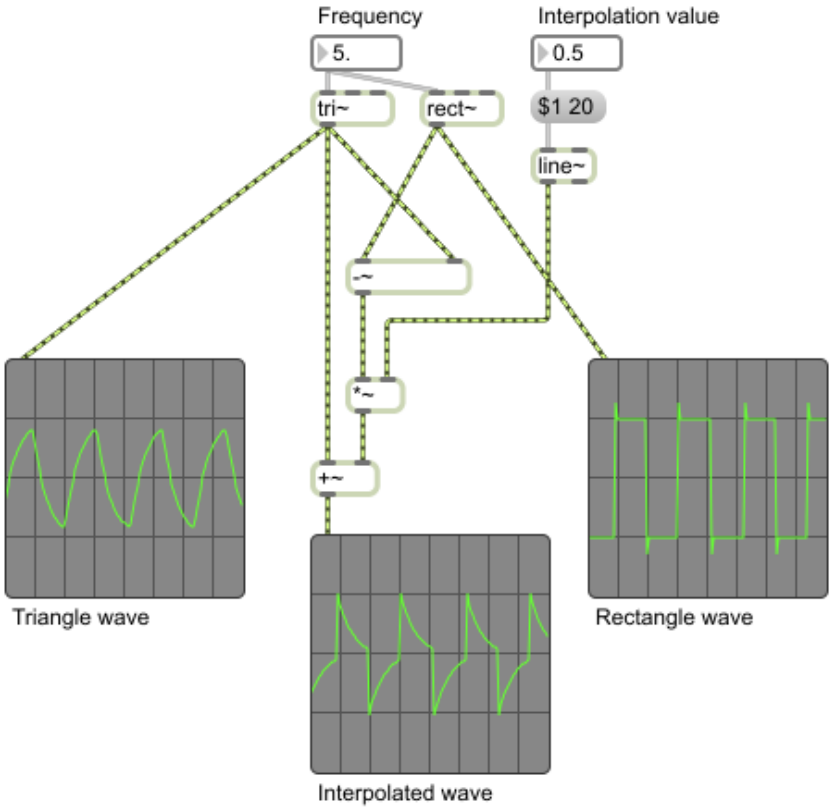

Figure 7 The interpolated waveform shown here is exactly half way between a triangle and rectangle waveform.

Parameters for attack, decay, sustain, and release (ADSR) modify the modulation depth for Synth2 over time. In addition, Synth2 has a parameter called "ramp" that defines the time it takes to glide between notes. This means that FMpolylegato (**Figure 8** below) has a total of ten parameters: carrier frequency, carrier waveform value, modulator frequency, modulator waveform value, depth of modulation, ramp time between notes, attack, decay, sustain of the modulation depth and the amplitude envelope.

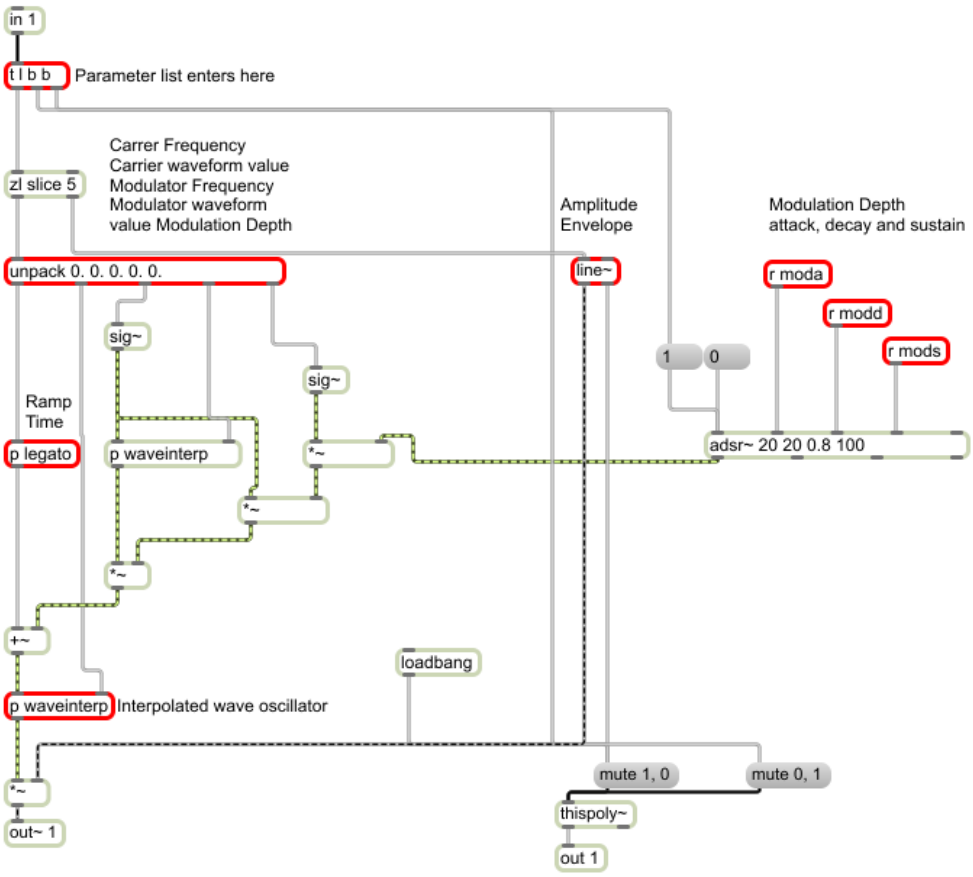

Figure 8 Each instance of FMpolylegato represents one voice in Synth2.

Like Synth1, each note is stored as a list of parameters that can be adjusted instantaneously. Two sub-instruments that are located in the sub-patch FM2 (**Figure 9** below) and named for their generic musical functions, "chordsynth" and "melodysynth" generate parameters for Synth2.

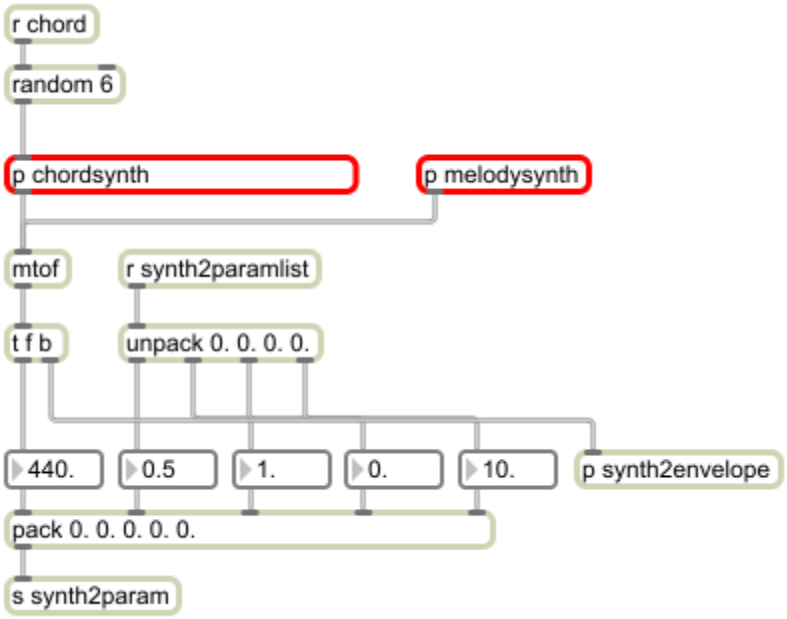

Figure 9 The Chordsynth and Melodysynth instruments are both located in the sub-patch FM2. Each parameter list sent to Synth2 generates a new instance of FMpolylegato with those parameters.

Melodysynth (**Figure 10** below) is an example of a basic algorithmic musical system. For every beat of its corresponding metronome, a random number is sent to a table that defines a musical scale set in the mapping layer (see "Mapping Later" section below). This value passes through an additional process that decides whether or not to raise the note by one or two octaves. The result corresponds to a MIDI note number and gets converted to a frequency value before triggering the list of FM parameters generated by the mapping layer. Unlike instruments in Synth1, parameters for melodysynth generally remain consistent between notes to approximate the behavior of a melodic instrument.

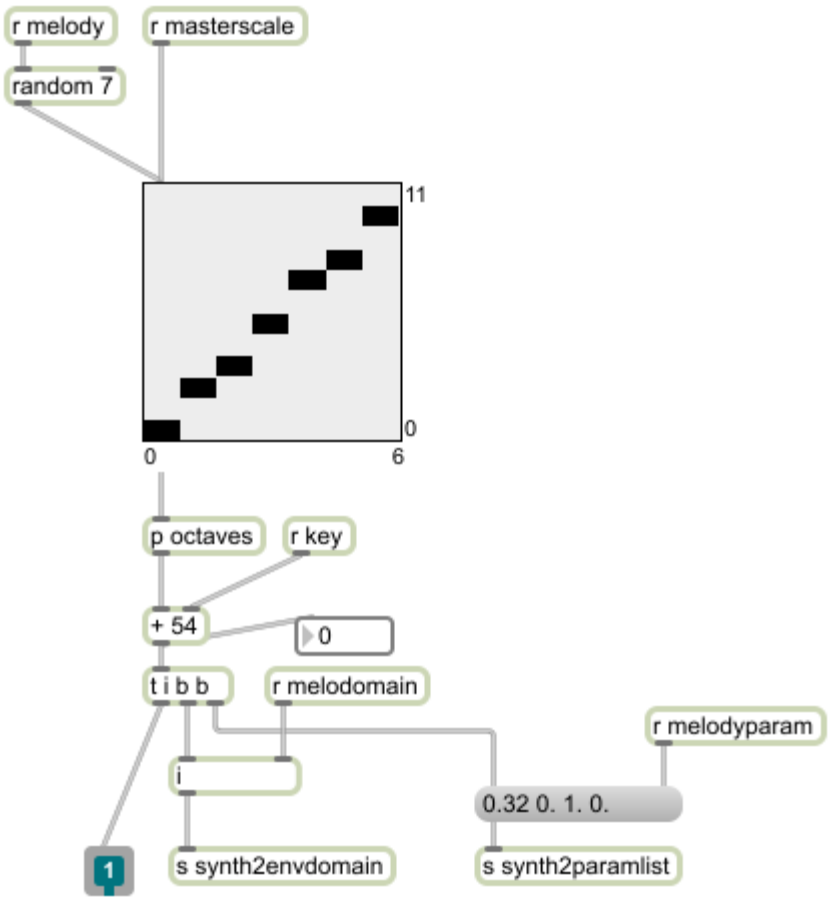

Figure 10 Melodysynth generates melodies that are constrained to a musical scale stored in a table. This scale is set by "masterscale," located in the mapping layer.

Chordsynth (**Figure 11** below) functions in a similar manner as melodysynth. For each chord, a random value, plus up to three notes above it are sent to a process which ensures that notes occur within one octave by using a modulo seven operation. Modulo is a mathematical operation that output the remainder of a division. For example, eight divided by seven leaves a remainder – or a modulo seven value – of one, therefore the note corresponding to number eight actually corresponds to number one. This process creates a simple musical grammar that produces chords that vary in quality, inversion and range. Each chord is also mapped to the scale

set by "masterscale" in the mapping layer (see "Mapping layer" below), producing a modal progression of chords. Each of these note values then passes through the same octave displacement processes as those in melodysynth.

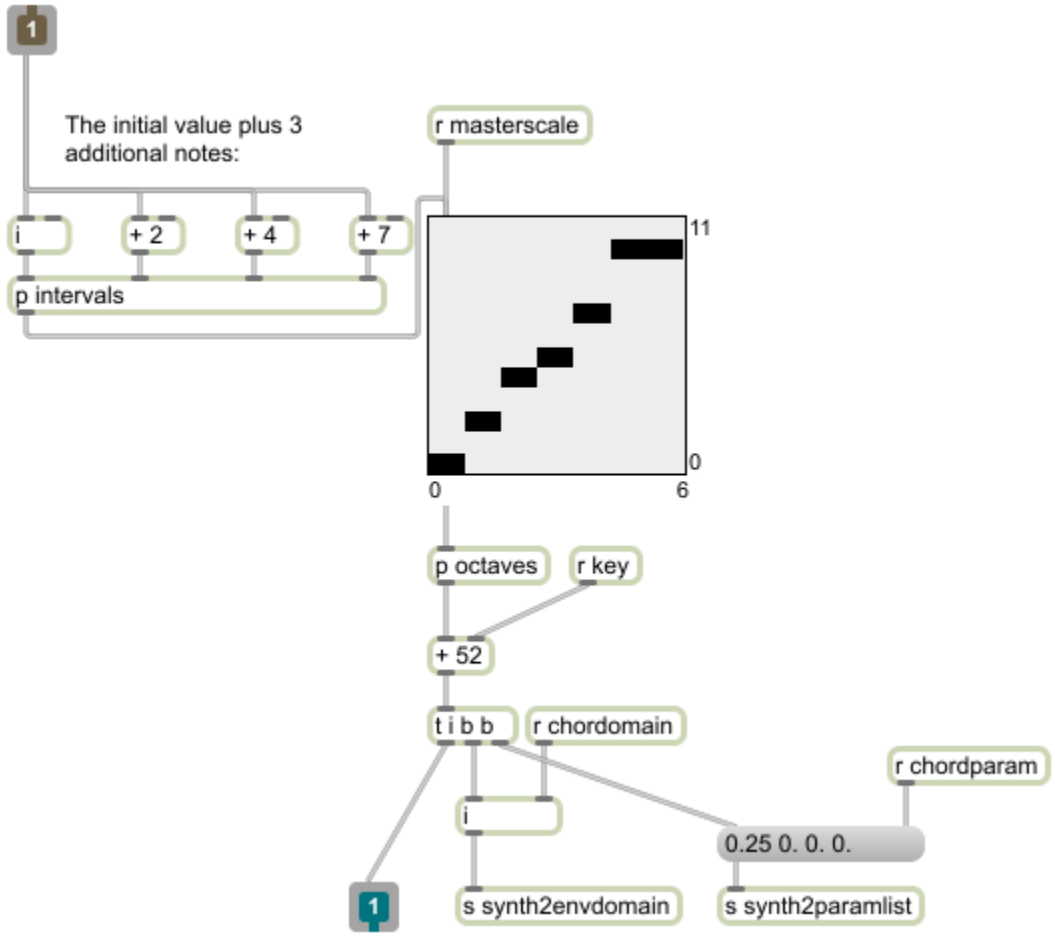

Figure 11 Chordsynth generates four notes in rapid succession in the same way that Melodysynth generates one note at a time.

The final component of the synthesis layer is a collection of three self-contained sound design modules for generating water, fire and footstep sounds. I based my sound design modules on Andrew Farnell's models for synthesizing natural sounds from his book on procedural audio

Designing Sound.[19] Each of the sound design modules in *High Score* is an attempt to synthesize a sound type and, more important to the philosophy of procedural audio, the behavior of a natural sound.

Two of the modules are more complex, and each one has two possible states – “unperturbed” and “perturbed.” In their "unperturbed" state, the modules behave as one would expect: running water gently bubbles, fire hisses and crackles. When a player interacts with an object in the game that is tied to a sound design module, he or she temporarily "perturbs" the system, resulting in an interaction sound. The footstep module is the simplest and has only one state in which it generates footstep sounds with random variations in pitch.

The water sound module is made up of two components. A pink noise generator produces the sound of water in the distance. In synthesis, noise generally refers to a sound that contains a wide band of frequencies. White noise refers to such a sound, where all frequencies have an equal amount of energy. Pink noise refers to a sound where the energy of a frequency band is inversely proportional to the frequency (**Figure 12** below).

[19] Farnell, *Designing Sound*, 407.

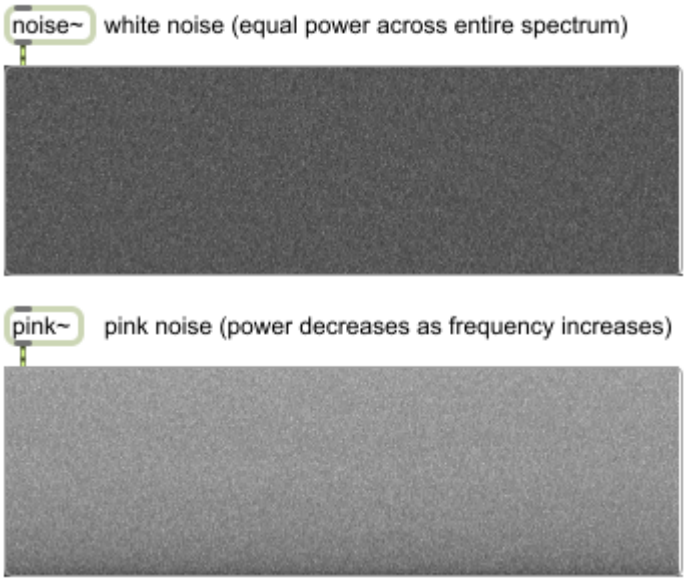

Figure 12 shows two sonograms of white and pink noise. A sonogram is a graph of frequency over time, where the darkness of the graph corresponds to energy at that frequency. Notice that the sonogram of pink noise is lighter at higher frequencies.

In the water module a sine wave oscillator produces the sound of running water from close up, as well as a splashing sound when the player shoots a cube at the water (**Figure 13** below). A sine wave is the simplest component of a synthesized sound. A simple running water sound is generated by rapidly changing the frequency of the oscillator. The range of random frequencies that the oscillator has defines its state. This range is set in the mapping layer. The unperturbed state of the water module refers to the range of 400 Hz to 600 Hz. When a player shoots a cube at the water, this range is increased to 1200 Hz.

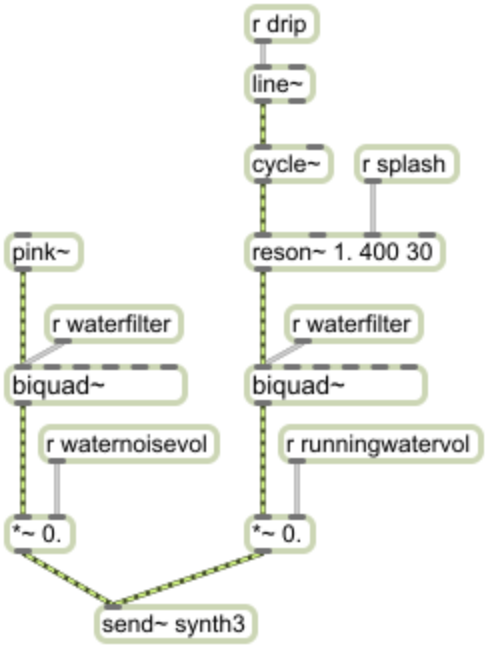

Figure 13 A pink noise generator and sine wave oscillator generate the component sounds of running water in the water module.

The fire sound module (**Figure 14** below) is based entirely on filtering a single source of white noise. Three sub-modules model the component sounds of a fire: "firehiss", "firecrackling", and "fireroar". Firehiss generates a high-pitched and mostly continuous hiss that is the result of the noise source passing through a high-pass filter. Firecrackling produces short bursts of noise by rapidly changing the center frequency of a narrow range of frequencies controlled by a filter. Fireroar produces a low, rumbling sound that is the result of the noise source passing through a low-pass filter. By blending these three components, the sound of a roaring fire can be reasonably approximated. The unperturbed state of the fire module is defined by the behavior of the three sub-modules: a quiet but audible hiss, few crackles and a low-pitched roar. When a player shoots a cube at a flame, the frequency of the hiss and roar, as well as the amount of crackling all momentarily increase.

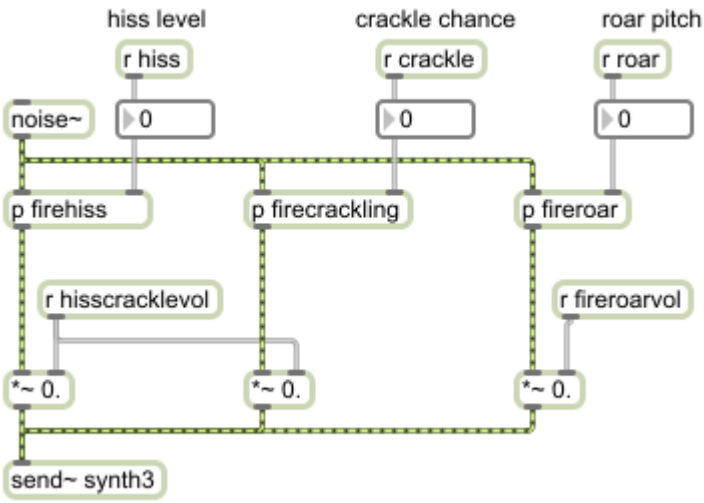

Figure 14 These three sub-modules make up the fire module. The player's position on the map, proximity to the sound source and interaction (via shooting cubes) alters the behavior of the modules.

A simple footstep module is also included (**Figure 15** below). Since the terrain of my game consists entirely of grass, a short burst of white noise will suffice to approximate the sound of a footstep on grass. This white noise passes through a filter whose center frequency is randomized with each footstep. This avoids the repetitive nature of the typical "clip clop" footstep and produces natural sounding movement.

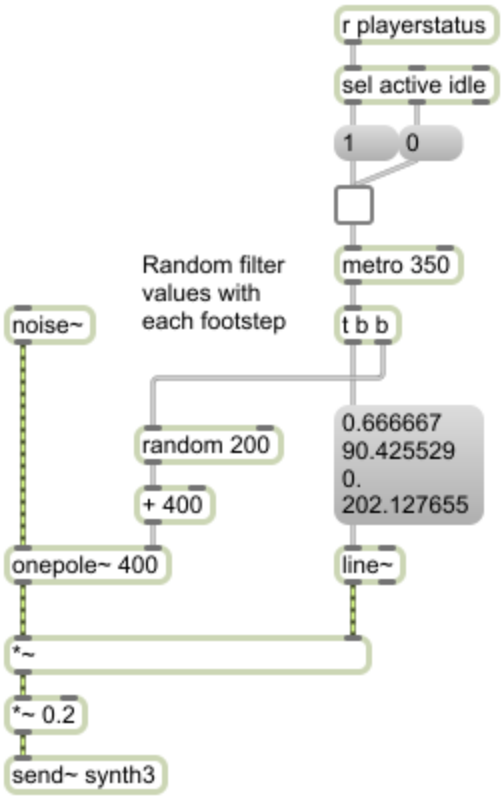

Figure 15 The footstep module is activated only when the player is moving.

Mapping Layer

Higher-level controls for both synthesizers are located in the Mapping layer. This layer of the engine contains controls for turning sub-instruments within the synthesizers on and off, adjusting the rate of instrument metronomes, setting parameters for each note, receiving data from Unity3D, interpreting that data and adjusting global presets to reflect the current state of the game.

Before either synthesizer plays a note, a table for each sub-instrument decides whether or not to activate a metronome for that instrument. These tables are located in a sub-patch titled "instrumentation." Each table is updated every quarter note. This decision is made based on probability values stored for two possible states that correspond to the metronome being on or off.

Tables for possible metronome rates for each instrument are located in the sub-patches titled "Synth1map" and "Synth2map" (**Figure 16** below).

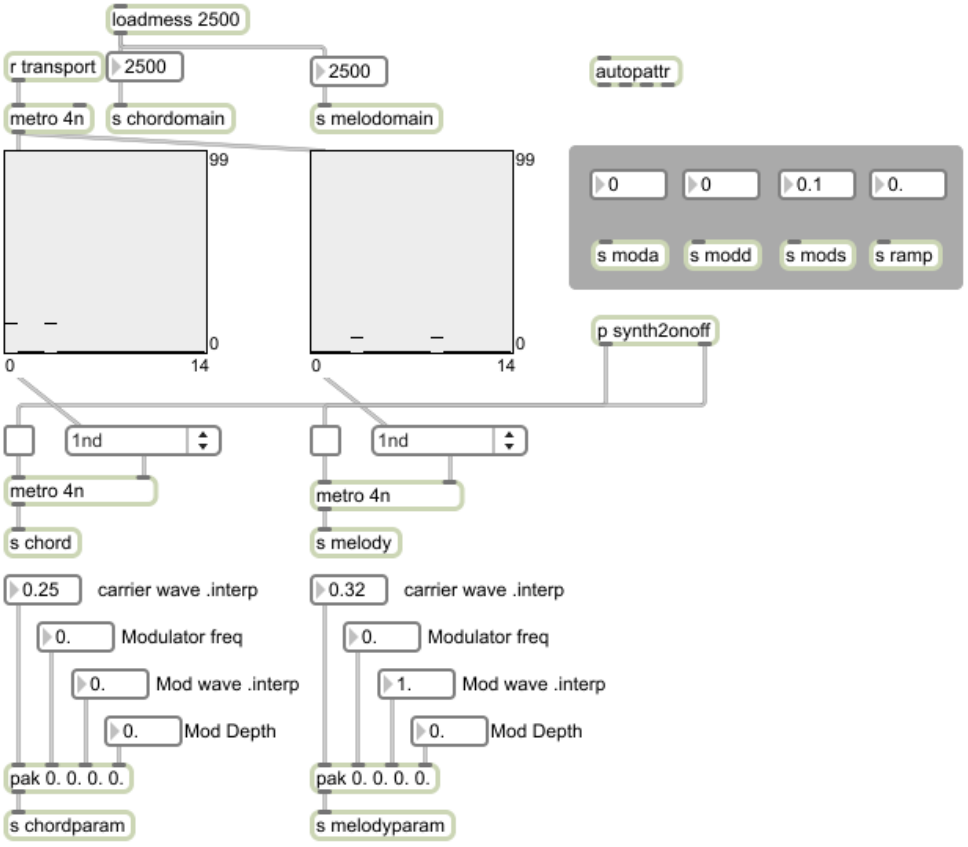

Figure 16 Synthmap2 contains controls for Synth2. I composed the music of *High Score* by adjusting these controls and saving the desired settings as a preset.

These tables' states can be stored as part of a global preset, that can be recalled or interpolated between, depending on the game state. For example, when the game state is set to "idle", all of the percussive instruments in Synth1 are disabled and the instruments in Synth2 are set to slower metronome rates.

The game state is determined entirely by the player's activity and proximity to a sound source. Currently, only two player states are defined: idle and active. The activity level is measured by querying the rate of change of the player's position on the map in the XZ plane (the Y plane corresponds to height) once every second. This is done by timing the interval between changes in the players X and Z position independently in a sub-patch titled "playeractivity" (**Figure 17** below).

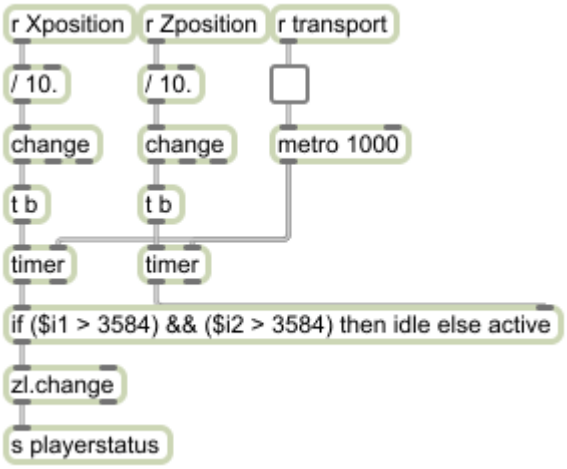

Figure 17 The "change" object only reports changes in the data sent to it. The rate of change is measured every 1000 milliseconds (every second).

Since a player could conceivably move only in one plane (if they are moving in a straight line), the rate of change is measured every 3.5 seconds. If both the X and Z positions

have not changed after 3.5 seconds, the message "idle" is sent to the mapping layer, where it triggers an interpolation from the current global preset to preset 0. This interpolation occurs over eight seconds, allowing the player time to notice that the music is responding to their inactivity. If the player begins moving before preset 0 is reached, the message active is sent, causing an interpolation back to the "active" setting, preset 1.

When a player approaches a sound source in the game, a corresponding preset is activated. Presets for sound sources diminish musical activity by adjusting instrument metronome rates and lower the volume of Synth1 and Synth2. This action allows the sound sources to come in to focus as important objects in the game. The volume of a sound module is mapped to the distance from the player to the sound source. For example, in the water module the sound of water in the distance generated by pink noise becomes audible when the player is less than 100 units away and increases until the player is standing at the water's edge. The sound of running water generated by a sine wave oscillator becomes audible when the player is less than 75 units away and increases similarly. Once the player is standing close to the water or walking around in it, the action of throwing a cube triggers a splash sound, generated by momentarily increasing the range of random frequencies sent to the oscillator.

Data sent from the various C# scripts in Unity3D (see **Appendix**) is received in a sub-patch in Max/MSP titled "networking" (**Figure 18** below). In Unity3D, a script titled "UDPstartup" sends a stream of data corresponding to how long the game has been running, in milliseconds. This same script sends a message "Unity3Dappclosed" when the game stops running. These messages start and stop the audio engine and also reset and initialize the global preset. Unity3D also sends the player's current position on the map from a script titled "UDPmove". When the networking section of the engine receives this stream of data, it arrives

as a byte array that must be immediately converted to ASCII characters. This produces a list such as "(256.0, 1.0, 229.0)" with quotations included. This list corresponds to the XYZ coordinates of the player. Further manipulation of this list is required to remove the quotations, parentheses, and to isolate each number from the list. The message "bullet" is sent from the script titled "csharpshoot" whenever the player shoots a cube. Finally, the "UDPnetworkmanager" script enables the overall communication between Unity3D and Max/MSP.

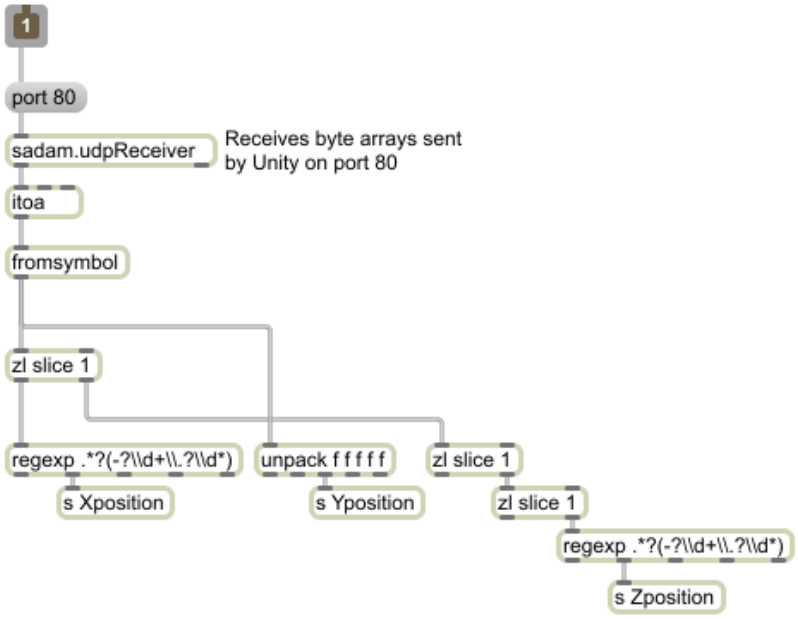

Figure 18 The networking sub-patch receives and processes lists using Adam Siska's sadam.udpReceiver object and the "regexp" object (allows the use of "regular expressions" for text manipulation).

VI. Interactivity and Nonlinearity

As mentioned above in Section III, two important concepts in games and game audio are *interactivity* and *nonlinearity*. I incorporated these concepts into each of the musical and sound design components of *High Score.*

The sound design modules are the simplest elements in terms of interactivity and nonlinearity. As explained in the previous chapter, the water and fire sound design modules have an unperturbed state and a perturbed state that results from the player's interaction. The footstep module provides a simple interaction and takes advantage of nonlinearity to produce realistic results. The fire module is based on white noise, an entirely random signal. The water module is based on the rapid succession of random frequencies produced by a sine wave. For both the water and fire modules, nonlinearity is the key to modeling the behavior of natural sounds.

A more abstract implementation of interactivity and nonlinearity exists within the percussive instruments of Synth1. It is important to note that each instrument played by Synth1 exists only as a list of parameters. Every time a list Synth1 receives a list, it generates a note with those parameters. By keeping the lists consistent within certain boundaries, each instrument maintains its identity while subtly changing each time they are activated. This behavior struck me as conceptually similar to acoustic percussive instruments that are never hit in exactly the same way twice. I use this behavior to give an organic quality to sounds that are obviously synthesized.

In *High Score*, a player controls the overall musical texture in two ways. The player's level of activity causes a gradual shift from the "active" music to the "idle" music or vice versa.

The player's position on the map defines what the active music is. Active music can be one of three themes: traveling, fire or water. Each theme has a unique musical scale and unique parameters for each synthesizer. The idle music is defined by slower metronome rates for all instruments and long sustained notes for the instruments of Synth2, giving it a rich, lush texture. The traveling theme is defined by faster metronome rates, more emphasis on the percussive sounds of Synth1 and short percussive chords. The fire theme is defined by a melodic element produced in Synth2 that alternates between extremely short, fast notes and longer slow notes. I intended this melodic theme to mimic the random wavering and lapping of a flame. Sustained chords that waver in pitch characterize the water theme. I designed this theme to mimic the distortion of an image reflected in rippling water.

I feel that these features highlight the most attractive prospect of procedural audio: defining an alternative concept of "realism" within sound. While data-driven audio delivers unparalleled realism in terms of the reproduction of recorded sound, procedural audio reproduces the behavior of sound. These behaviors can serve to simulate natural sounds, as with the water and fire sound design modules, or to generate musical material.

VII. Conclusion

The implementation of both procedural audio and algorithmic composition is a choice to apply a common philosophy to different aspects of a game audio engine. This philosophy stems from the idea of emergent phenomena[20], whereby the interactions of simple parts yield unpredictable global behavior in a complex system. Games frequently exploit this kind of phenomena by providing players with a complex set of simple rules, constraints and options for solving problems. By combining procedural audio and algorithmic composition, the sonic environment of a game can better reflect the interaction between the game and player.

This approach may not be suitable for all situations, but represents what I hope is a new frontier in a relatively new medium. An ideal approach would leverage the advantages of both data-driven and procedural audio to create a realistic sound environment that responds intelligently to player actions.

In the future, I would like to design a similar system for use across mobile devices and web browsers. I also envision helping design a manifestation of a data-flow programming language such as Max/MSP that allows more flexibility than current applications being integrated into audio middleware and game development platforms to facilitate the widespread use of procedural audio and algorithmic composition. I chose to develop *High Score* using Max/MSP and Unity3D because they are among the most commonly used tools in their respective categories. By demonstrating a procedural audio engine made with these two

[20] Vince Darley, "Emergent Phenomena and Complexity." in *Artificial life IV,* ed.Rodney Brooks and Pattie Maes 411-417.Cambridge, Mass: MIT Press, 1994.

programs, I hope to encourage others to explore the fascinating potentials of procedural approaches to sound design and music.

Bibliography

Collins, Karen. "Loops and Bloops: Music of the Commodore 64" *Soundscapes* (February 2008), www.icce.rug.nl/~soundscapes/VOLUME08/Loops_and_bloops.shtml

———.2008. Game sound: An introduction to the history, theory, and practice of game music and sound design. Cambridge, Massachusetts: MIT Press 2008.

Chowning, John M. 1973. "The synthesis of complex audio spectra by means of frequency modulation." *Journal of the Audio Engineering Society* 21, no.7: 526.

Darley, Vince "Emergent Phenomena and Complexity." in *Artificial life IV: proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems*. Edited by Rodney Brooks and Pattie Maes, Cambridge, Mass: MIT Press, 1994.

Farnell, Andrew. "An introduction to procedural audio and its application in computer games." http://obiwannabe.co.uk/html/papers/proc-audio/proc- audio.html (accessed 9/4, 2012).

———.2010.*Designing sound*. Cambridge, Massachusetts: MIT Press.

Postel, Jonathan and David P.Reed. "User Datagram Protocol" *Internet Engineering Task Force Request for Comments 768,* (August 1980)

Appendix

UDPstartup

```
using UnityEngine;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
using System.Net;
using System.Net.Sockets;


public class UDPstartup : MonoBehaviour {

        IPEndPoint ep;
        //Socket client;
        UdpClient client;

        // Use this for initialization
        void Start ()  {

                //port and IP Data for Socket Client
                string IP = "127.0.0.1";
                int port = 81;

                ep = new IPEndPoint(IPAddress.Parse(IP),port);

                //socket that allows sending data
                //client = new Socket(AddressFamily.InterNetwork, SocketType.Dgram, ProtocolType.Udp);
                client = new UdpClient();

        }

        // Update is called once per frame
        void Update ()
{

                //convert string in to byte array
                byte[] packetData = System.Text.ASCIIEncoding.ASCII.GetBytes(Time.realtimeSinceStartup.ToString());

                client.Send(packetData, packetData.Length, ep);
}

        void OnApplicationQuit()
        {       //convert string in to byte array
                byte[] packetData = System.Text.ASCIIEncoding.ASCII.GetBytes("unityappclosed");

                client.Send(packetData, packetData.Length, ep); }
}
```

UDPNetworkmanager

```
using UnityEngine;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
using System.Net;
using System.Net.Sockets;


public class UDPmove : MonoBehaviour {

        IPEndPoint ep;
        //Socket client;
        UdpClient client;

        // Use this for initialization
        void Start ()  {

                //port and IP Data for Socket Client
                string IP = "127.0.0.1";
                int port = 80;

                ep = new IPEndPoint(IPAddress.Parse(IP),port);

                //socket that allows sending data
                //client = new Socket(AddressFamily.InterNetwork, SocketType.Dgram, ProtocolType.Udp);
                client = new UdpClient();

        }

        // Update is called once per frame
        void Update ()
{

                UDPNetworkmanager.GetInstance().SendMessage(transform.position.ToString());
                //convert string in to byte array
                //byte[] packetData = System.Text.ASCIIEncoding.ASCII.GetBytes(transform.position.ToString());

                //client.Send(packetData, packetData.Length, ep);

 }
 }
```

UDPmove

```
using UnityEngine;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
using System.Net;
using System.Net.Sockets;


public class UDPmove : MonoBehaviour {

        IPEndPoint ep;
        //Socket client;
        UdpClient client;

        // Use this for initialization
        void Start ()  {

                //port and IP Data for Socket Client
                string IP = "127.0.0.1";
                int port = 80;

                ep = new IPEndPoint(IPAddress.Parse(IP),port);

                //socket that allows sending data
                //client = new Socket(AddressFamily.InterNetwork, SocketType.Dgram, ProtocolType.Udp);
                client = new UdpClient();

        }

        // Update is called once per frame
        void Update ()
{

                UDPNetworkmanager.GetInstance().SendMessage(transform.position.ToString());
                //convert string in to byte array
                //byte[] packetData = System.Text.ASCIIEncoding.ASCII.GetBytes(transform.position.ToString());

                //client.Send(packetData, packetData.Length, ep);

 }
 }
```

csharpshoot

```
using UnityEngine;
using System.Collections;

public class csharpshoot : MonoBehaviour {

        public Transform prefabBullet;
        public int shootForce = 100;
        public float fireDelay = 1;

        private bool _canFire = true;
        private float _delayTimer;

        // Use this for initialization
        void Start () {

        }

        // Update is called once per frame
        void Update () {

                if(_delayTimer < Time.time)
                {
                        _canFire = true;
                }

                if(_canFire && Input.GetMouseButton(0) )
                {
                        Transform instanceBullet = Instantiate(prefabBullet, transform.position, Quaternion.identity) as
Transform       ;
                        instanceBullet.rigidbody.AddForce(transform.forward * shootForce);

                        UDPNetworkmanager.GetInstance().SendMessage("bullet");
                                _canFire = false;
                                _delayTimer = Time.time + fireDelay;
                }

                Destroy(GameObject.Find("Bullet(Clone)"), 1.0f);

        }
}
```