

The Kinetic Mandala: Audio A-life in a Web-based 3D Environment

Maribeth Back
Xerox PARC

Maureen Stone
StoneSoup Consulting

ABSTRACT

We describe the use of artificial life (genetic) algorithms to generate auditory behaviors within 3D graphical environments. Small, high-quality soundfiles are algorithmically recombined and their behavior altered to create rich, non-looping sound environments. These environments consist of sound effects as well as musical tones; for example, the sound of a single water drop or a single ocean wave can be replicated and regenerated according to algorithmic rules to create a long-running non-looping sound effect. Sonic parameters such as pitch, start time, intensity, and apparent spatial location are given initial values which are then changed in realtime as the Java-based genetic algorithms generate new values.

This approach is particularly useful in Web-based 3D environments like VRML or Java3D, as it can dramatically reduce the necessity for downloading large soundfiles. It can also be used in real-world systems to produce long-running, interactive, non-looping sound environments. This work can be thought of as a generalizable content-based form of audio compression. Applications lie in awareness systems, entertainment systems, real/virtual interplay, and especially applications in long-term auditory monitoring and display. The test environment we built to illustrate our system is the Kinetic Mandala, an abstract VRML 2.0 world with six rooms, each containing audio artificial life systems.

Keywords: Artificial life, audio design, sound design, audio compression, sound textures, algorithmic composition, VRML audio.

Maribeth Back , Xerox PARC
3333 Coyote Hill Road, Palo Alto, CA 94304
650-812-4726, back@parc.xerox.com

Maureen Stone, StoneSoup Consulting
191 Pine Lane, Los Altos, CA 94022
650-559-9280, stone@stonesc.com

1 INTRODUCTION

Algorithmic devices, including genetic algorithms, have been used by composers for decades as a compositional tool [6, 7]. Using natural genetic systems as a model for recombination allows the creation of clearly related but not repeating sounds. We now envision a strong application for genetic algorithms in generating audio effects and other dynamic behaviors for Web-based 3D environments. We have created sonic ecologies built out of auditory entities, breeding according to simple rules from a soup of sonic DNA: tiny but high-quality soundfiles.

The resultant audio objects and behaviors can be used in 3D graphical environments or in real environments, and can alter themselves in response to changes in these environments, whether systemic or user-initiated. The algorithms that create sound may also drive other dynamic processes such as generation of new geometry, lighting, viewpoint, or animation, in conjunction with the sonic behavior. Work in procedurally generated textures or generative algorithms for geometry or avatar behaviors would be the graphical equivalent of our work in audio.[2] We are also interested in possible uses of these algorithms for awareness or entertainment systems in real environments, or for any system where information is delivered sonically over a long period of time.

1.1 Novelty

Algorithmically produced music is often criticized for its lack of variation. However, in the design of sound environments, the unmusical characteristics of genetic composition are a plus. A stream should not vary too greatly, nor should wind, nor the sound of birds chirping. What we do require from these sounds is believable behavior, and that the human ear be unable to detect a loop during playback. In the design of sound for live theatre, sound loops will often play for as much as two minutes before repeating; even so, some people notice. We avoid this phenomenon by creating parameters for altering the behavior of the auditory object in (apparent) realtime.

One of the most common problems with Web sound is that it often uses the built-in (low-quality) MIDI sounds on a PC's internal soundcard. Unlike most algorithmic composition heard on the Web, our data is not driving a General MIDI set. Our algorithms are designed to use a small number of short, high-quality soundfiles as source material, and to recombine them continuously within certain constraints. They also accept input and can alter their behavior accordingly.

1.2 Compression effect

On the Web, a two-minute soundfile is relatively huge at any reasonable quality. For example, a single two-minute 16-bit

22kHz mono file, (one-quarter the size of CD-quality audio, which is 16-bit 44.1kHz stereo) weighs in at five megs (uncompressed). Though many compression algorithms for audio exist, the standards are not always cross-platform and the compression is often either not at a very high ratio (4:1 is common) or it leaves the audio with undesirable, audible artifacts. These artifacts cause further trouble if the soundfile is then used in any but the simplest fashion. For example, a simple pitch shift (one of the common VRML audio behaviors) will often expose compression-induced artifacts.

Streaming protocols, though showing improvement, do not easily allow interactive audio behaviors; neither are they as high quality as one could wish. It is possible to exploit audio streaming as a delivery mechanism for our algorithms, while doing all the synthesis at our end. In this case the value of our approach lies primarily in the adaptable interactivity of the genetic algorithm over time.

2 KINETIC MANDALA

The Kinetic Mandala, our VRML 2.0 test environment, combines kinetic sculptures and algorithmic sound in a simple 3D graphics environment. [4] Its implementation is a combination of VRML 2.0, Java, and Javascript. [2, 3, 10]. This system has been developed under the CosmoPlayer VRML 97 plugin running in Netscape. Our development configuration is CosmoPlayer 2.1 and Netscape 4.05 running under Windows 95 and DirectX 5 (DirectX 5 provides the best Windows sound support).



Fig. 1. The Kinetic Mandala.

2.1 Graphics in the Kinetic Mandala

The Kinetic Mandala is a portion of a tiling pattern. (See Fig. 1.) It is basically a hexagon made of six triangles, where each triangle is slightly distorted. The floor of each triangular room can be tiled with equilateral triangles. Each triangle is 10 units

on a side. Assuming a unit is a foot creates a room 30 feet on a side with small alcoves in each corner. The walls are twelve feet high, the open doorways eight feet high by five feet wide to give a museum-like quality to the space.

2.2 Room Creation

There are two types of rooms: active rooms that contain sculptures and rich sound environments, and dark rooms that contain only a low ambient sound. Because we wanted to optimize the rendering, there are no lights shining on the walls or floors of the room. Each floor is a single IndexedFaceSet, colored a solid color and texture-mapped with a mottled intensity map. There are two wall PROTOs, one with and one without a door. Each wall is a single IndexedFaceSet with colors mapped to the vertices that create ramps from white to dark gray.

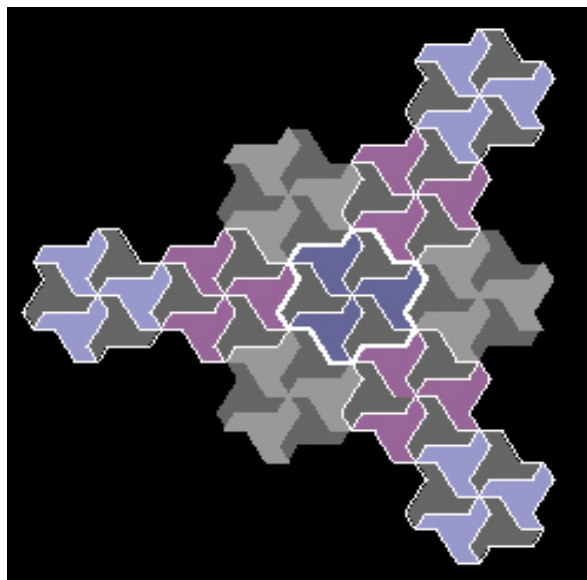


Fig. 2. Creating tiled geometry on the fly and growing sound environments as accompaniment. The original six-room Mandala is at the center.

There are two PROTOs that create complete rooms, including their contents. The activeRoom PROTO takes a list of objects for each corner and for the center. This is how the kinetic sculptures and sounds are inserted in the room. The darkRoom PROTO takes a parameter for the sound in the center. For the rooms with external doors, a MFNode is provided that is instantiated when the door is clicked, and a URL for the sound the door makes when it opens. Dark rooms share two out of three walls with the adjoining active rooms. The mandala PROTO creates the Kinetic Mandala using three calls to activeRoom and three calls to darkRoom. The ultimate goal for graphics in the mandala work is to tile the plane with rooms derived from the initial rooms. (Fig. 2.)

2.3 Kinetic Sculptures

The kinetic sculptures in the mandala were designed by Rich Gold of PARC. To provide a design theme, Rich called the sculptures "goddesses" and named them for attributes such as Joy, Life, Pain, etc. Each goddess is an experiment in using a VRML animation feature. These include a variety of interpo-

lators plus one or more TimeSensor nodes. Only the Danger goddess uses any other programming.

In the Mandala test environment the sculptures provide scale, give focus and add a sense of place for the sound environments to grow.

3 SYSTEM DESIGN FOR SOUND

The VRML 97 specification defines a Sound node that generates spatialized sound in a VRML world.[12] The parameters for the Sound node include location, intensity, and parameters that define the extent of the spatialization defined as two ellipses, one within the other. The Sound node also includes an AudioClip node that describes the sound source. The parameters for the AudioClip node include pitch, startTime and the url for the sound data.

In our work, the sonic data for the AudioClip node is defined by a .wav format file. We dynamically modify pitch, start-Time, location and intensity using a simple a-life algorithm.

3.2 Sounders and SounderSets

Our sonic environments are created by using several Sound nodes plus two levels of scripting to "play" them. Each Sound node is grouped with a script that modifies its parameters and starts it playing, a combination we call a Sounder.

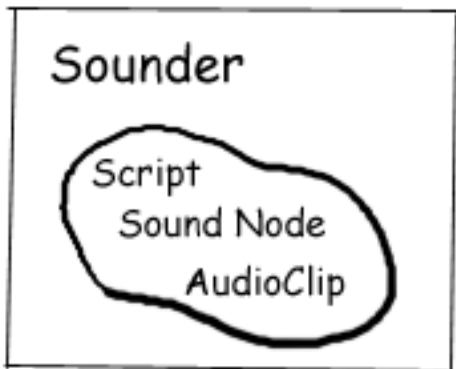


Fig. 3. A Sounder consists of a script, a Sound node, and an AudioClip node.

Related Sounder nodes are collected into a SounderSet. The SounderSet includes a master script that computes the parameters for all the Sounders in the set so that the Sounders can influence each other. (See Figs. 3 and 4.)

The PROTO that implements the Sounder takes the Sound node and a unique id for the Sounder as parameters. The script in the Sounder is written in Javascript, and implements the events initialize, enable, setValues and changeSound.

The PROTO that implements the SounderSet takes the SounderData for each Sounder. This data includes the Sounder itself, plus all the parameters for the a-life algorithm. The script for the SounderSet is written in Java, and implements the events initialize, enable and createValues. Values are created using the simple a-life algorithm described below.

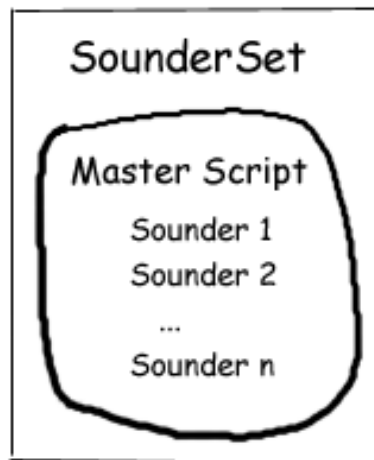


Fig. 4. A SounderSet contains and controls multiple Sounders. Sounders within a SounderSet influence each other by swapping "sound genes."

3.3 Communication and Synchronization

The Sounders communicate with their SounderSets via the createValues (Sounder to SounderSet) and setValues (SounderSet to Sounder) routes. Each Sounder has a unique id that is used to communicate with the SounderSet. This id is sent via the createValues event to trigger the computation of the Sounder parameters, which are returned as the setValues event.

For each Sounder, the Script node sets the parameters of the Sound and AudioClip nodes and starts the .wav file playing. It then waits for the node to finish, sets new parameters and starts it again. While there is an isActive event generated by the AudioClip node, its timing is not accurate in current implementations of VRML. Therefore, we use a TimeSensor by computing the play time from the duration of the .wav file, its pitch, any start delay and the system latency. We route the isActive event from the TimeSensor to the changeSound event on the Sounder to form the inner loop. (See Fig. 5.)

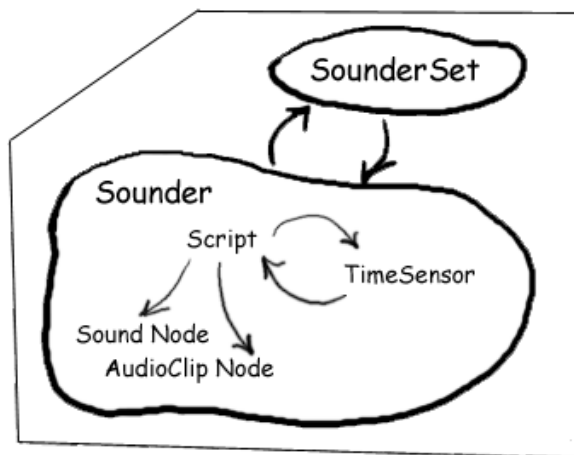


Fig. 5. SounderSets control multiple instances of Sounders, each Sounder having an inner loop where the Sound node and AudioClip Node fields are actually rewritten.

3.4 Initialization

A `SounderSet` is a Group of `Sounder` nodes instantiated in the scene graph. `Sounder` nodes are Group nodes, containing a `Script`, `TimeSensor` and a `Sound` node. To make the typed-in description of a `SounderSet` more concise, the instantiation of these nodes is performed by the initialization scripts of the `Sounder` and the `SounderSet` PROTOs.

The initialization procedure for the `SounderSet` node parses the `MFNode` of `SounderData`, which contains the `Sounder` node and the a-life parameters for each `Sounder`. For each `Sounder`, it uses the a-life parameters to initialize the `SoundGene`, `SoundBot` and `SoundGenerator` classes. It adds the `createValues` route from the `Sounder` node to the `SounderSet`. It then creates a `MFNode` of all the `Sounder` nodes in the set and adds them to the `SounderSet` Group node.

The `Sounder` node initialization procedure adds the `Sound` node to the `Sounder` Group node, which already contains the `Script`, the `TimeSensor` and the route between them.

We discovered that when we instantiated `Sound` nodes in this manner, their location did not follow the transformation stack. We have compensated for this apparent bug by explicitly providing a location parameter to the `SounderSet`. This is sent to each `Sounder` once it is instantiated and used to set the location field of the `Sound` node.

3.4 Enable and Disable Functions

An important part of the implementation of `Sounders` and `SounderSets` is the enable/disable function. While most VRML implementations cull `Sound` nodes that cannot be heard, the most expensive part of the Kinetic sound implementation is the scripts. Therefore, for both operational and aesthetic reasons, we must be able to start and stop the scripts gracefully.

Each `Sounder` has an enable eventIn whose value is stored as the boolean flag, `isEnabled`. This flag controls whether the script plays the `Sound` node and requests parameters. When `enable=false` is received, no further events are sent by the script. However, if the `Sound` node is active at the time, it is allowed to play to completion. When `enable=true` is received, the script starts the `Sound` node with the current parameters, then requests new ones.

Each `SounderSet` also has an enable eventIn. This is passed directly to the `Sounders` in the `SounderSet`. One design consideration is whether there should also be a way to restart the `SounderSet` with its initial parameters. We do not currently provide one.

In our current implementation, the `SounderSet` enable event is triggered by a `ProximitySensor` that is included in the `SounderSet` definition. This was a pragmatic decision for this prototype; a more general implementation would keep the `ProximitySensor` external to the `SounderSet` implementation.

3.5 Artificial Life Algorithm

We modify six `Sounder` parameters (`startWait`, `intensity`, `pitch`, `x`, `y`, `z`). Each is used to define a `SoundGene` for the a-life algorithm. A set of `SoundGenes` is collected into a `SoundBot`, giving one `SoundBot` for each `sounder` in the `SounderSet`. The

`SoundGrower` class updates the `SoundGenes` for each `SoundBot` by pairing it with another, then implementing a simple gene-swapping algorithm.

Each `SoundGene` contains the current value, `delta`, `min`, `max`, `swapping probability (sp)` and `mutation probability (mp)`. The initial value for these parameters is provided in the `SounderData` in the VRML file. The default behavior for each iteration is to simply add the `delta` value to the current value. Each gene in a `SoundBot` is updated independently of its other genes; it is only influenced by the corresponding gene in the paired `SoundBot`.

For each iteration, the `swapping probability` is used to determine whether to swap the `delta` values between paired genes. Then, the `mutation probability` is used to determine whether to randomly modify the `delta` value (up to 30%, plus or minus). A minimum and maximum bound are used to define a range for the parameter. If the new value exceeds the range, it is set to the boundary value and the sign of the `delta` value is inverted.

The Java implementation includes the three a-life classes `SoundGene`, `SoundBot` and `SoundGrower` plus the extension to the VRML `Script` class that provides the communication between VRML and Java programs.

3.6 Selection algorithms

One major issue in the design of genetic algorithms is selection: how the algorithm chooses which entities are best suited to continue breeding to develop toward some desired end. [5, 8, 9] In our current embodiment, we do not implement selection algorithms. Unlike most systems using genetic algorithms, we are making use of the behavior of the phenotypes (individuals), rather than the genotypes. Sound is produced by the presence of the individual itself, rather than by selecting fitness parameters toward some desired goal for the genotype. Because the soundfiles are regenerated based on their previous state, the new sounds bear a trackable relationship to the previous generation, lending an informative non-random quality to the sound environment thus generated.

Another problem in most a-life systems is removal of older generations; because sound is a time-based medium, we do not actively kill off our `soundBots`, but allow them to play out their inherent length. A later iteration of this work is likely to include both selection algorithms and a life-length parameter for the audio file.

4 SOUND DESIGN

The art in the application of these algorithms lies in matching a set of behaviors to the inherent qualities of a particular soundfile, while keeping in mind the design intent or application. For example, if a sound environment that seems realistic is the desired result, then too much variance in pitch might not work on a frog croak. But a raindrop sound can tolerate a much wider pitch range before it fails to “read” as a raindrop. If strict realism is not an aim in the design of a particular sonic environment, then initial values for behaviors can be set with much greater freedom.

A sound designer designs the sonic behaviors through manipulating the set of initial values (`SounderData` within a `SounderSet`, within the “`sounder`” .wrl files). The commented `SounderSet` code in Appendix A shows a set of initial condi-

tions for a single soundBot. It calls a single short pitched sound, and initializes it with six genes.

For the Kinetic Mandala, we chose not to mimic auditory reality, but rather to create rich, non-repeating audio textures. We took several different design approaches, described below.

4.1 Room Designs

Room One in the Kinetic Mandala has four separately generated sound environments, one in the room's center and one in each of the room's three corners. This Room shows the most dramatic compression effects of our system; each of these complex, non-looping sound environments uses only one small soundfile. A typical download for one of these a-life environments is 16K for the VRML and 10k for the soundfile.

In Room One, the sculpture "Life" is the focal point for a rich, rippling texture created from a single raindrop sound, which rapidly replicates and mutates through a wide variety of pitches, locations, and intensities. "Thought" uses a similar structure – a short sample of a single frog croak – but is more limited in the pitch range. "Joy" is a more complex instance, with several different sounderData value sets applied to the same soundfile (a single gibbon whoop). So, although the download time is almost exactly the same as the others (a few more lines of text in the VRML file), the resultant behavior is much more complex. Finally, Room One's center sound is a longer file of chime sounds (the file itself was generated from an a-life algorithm) with much lower, slower mutation and replication rates.

Room Two shows more experimentation with the relationship between audio content and the initial conditions set for the a-life generator. This room's sound environments include a bit more overtly musical content, as in "Impatience," which sends a single pitched percussion sound chasing itself up and down the scale. "Wealth" combines whispered words and phrases with the sound of coins falling musically, and "Time" is a set of percussive sounds reminiscent of the workings of a clock. It is the one environment in the Mandala that requires strict time relationships between the sound files. Room Two's center sound is a single long low vibe note, which layers on top of itself nicely without getting in the way of the other, higher-pitched environments.

Room Three pushes the system close to its current practical limit. Each environment here uses several source soundfiles, each with its own set of genetically induced behaviors. This means that each goddess sculpture has several associated SounderSets, each with a number of Sounders; this pushes the number of scripts running at any one time up into the teens.

Room Three's "Danger" goddess features several short, sharp metallic sounds, which have limited influence on each other. "Pain" uses short sharp sounds as well, but these have a more organic feel, and have been given a much stronger interconnection in behavior. "Indecision" recombines the sounds of thundercracks and howling winds to create an ever-evolving storm environment. Room Three's center sound is a low moaning cello note which recombines with itself to build a solid sound floor for the other environments.

The three Dark Rooms all share the same a-life algorithm: a low vibe sound slowly mutating and folding upon itself.

4.2 Other design possibilities

By tuning the initial parameter set, very different sonic behaviors arise. For example, one can create a sonic space-worm: the multiple instances of the file connect with each other and maintain a pitch-wise and spatial cohesion for a certain period of time. The result is a set of sounds, obviously related melodically and spatially, that appear to crawl back and forth between stereo speakers attached to the host computer.

In another example, if we use these algorithms dynamically to create new geometry in the VRML world (for example, a new room, with characteristics inherited from Rooms One and Three), then a sound environment should also be created for the new room, with sonic characteristics inherited from Rooms One and Three.

5 Performance and Pragmatics

The Kinetic Mandala is fragile because it stresses both the sound and the script node implementation in a VRML browser. It creates a relatively large number of Sound nodes that it plays in rapid succession. This has influenced the design of our demonstration and limits the number of systems it can run under. We have developed it in using the CosmoPlayer 2.1 plugin to Netscape 4.05 running in Windows 95. While we hope that it will eventually run under other configurations, this is the only one we have extensively tested.

5.1 Demo Design

The Sonic Mandala emphasized the use of spatialized sound by blending the sound from three sound sources for each room. [11] For the Kinetic Mandala, we found that two SounderSets was the most we could run simultaneously and maintain any sort of graphics performance. Therefore, we changed the design to include a room sound, and constrained the goddess sounds to play one at a time.

The full Kinetic Mandala should include nine SounderSets (one for each of the goddess sculptures), three room SounderSets, and three instances of the dark room sound. However, we found that a fully sonified Mandala would not load without crashing Netscape/CosmoPlayer. Our solution was to split the demonstration into individual rooms plus the simplified Mandala.

5.2 Performance Issues

The Kinetic Mandala runs a relatively large number of scripts for a VRML world. Running scripts is expensive in current implementations, though we suspect the real cost is the communication between the browser and the script, not the execution of the script itself. A graphics accelerator should help, though we have no experience with this; the boards we used didn't accelerate much in Windows 95 (we believe this was an OpenGL implementation problem). Even though we are aiming for concise use of .wav data, it helps the performance of the system to set the Netscape memory cache to around 5000 KBytes. Setting it larger seems to cause Netscape problems.

6 Conclusion

For some applications, using genetic algorithms for the generation of rich sonic environments in VRML 2.0 or other

Web3D systems is a viable alternative to streaming audio or simply downloading huge audio files. We expect that entertainment or awareness applications will be the first to implement these systems, along with the inclusion of user-specific preferences for growing sonic environments.

Future work will focus more deeply on the relationship between content of the audio files and the design of the a-life algorithms. We are also experimenting with recombinant audio behaviors at different scales, from basic audio synthesis to large-scale Darwinian audio ecologies. Other traditional aspects of a-life systems, such as selection and evolution, invite exploration in greater depth.

ACKNOWLEDGMENTS

The authors would like to extend special thanks to Rich Gold of the Research in Experimental Documents group at Xerox PARC, who designed the kinetic "goddess" sculptures in the Kinetic Mandala. We are also indebted to David Chamberlin of SGI for his help designing the right way to drive the Sound Nodes in CosmoPlayer, and Don Brutzman and Chris Marrin for general help with VRML/Java issues.

REFERENCES

[1] Ames, A., Nadeau, D., and Moreland, J. VRML 2.0 Sourcebook, John Wiley & Sons, Inc., 1996.

[2] Biota.org, a working group of the VRML Consortium: <http://www.digitalspace.com/papers/ngarden.com>

[3] Carey, Rikk, and Gavin Bell. The Annotated VRML 2.0 Reference Manual, Addison-Wesley Developers Press, 1997. Appears online as The Annotated VRML97 Reference Manual, at <http://www.wasabisoft.com/Book/Book.html>

[4] *The Kinetic Mandala*. Demo at http://www.parc.xerox.com/red/projects/web3d/mandala/kinetic_mandala/index.html

[5] Langton, Christopher, ed. *Artificial Life: An Overview* (Complex Adaptive Systems). MIT Press, Cambridge, MA, 1995.

[6] J.L. Leach, J.P. Fitch: "Nature, Music and Algorithmic Composition," in *Computer Music Journal* 19(2):pp. 23-33, 1995.

[7] Loy, Gareth. *Composing with Computers: A Survey* in *Current Directions in Computer Music Research*, ed. Max Mathews and John R. Pierce. MIT Press, Cambridge, MA, 1991.

[8] Mitchell, Melanie. *Introduction to Genetic Algorithms* (Complex Adaptive Systems Series). MIT Press, Cambridge, MA. April 1998.

[9] Reynolds, Craig. *Competition, Coevolution, and the game of Tag*. In the proceedings of *Artificial Life IV*, R. Brooks and P. Maes, editors, MIT Press, Cambridge, MA, pp. 56-69. 1994.

[10] Roehl, Bernie, ed. *Late Night VRML 2.0 with Java*. Ziff-Davis Press, 1997.

[11] *The Sonic Mandala*. Demo, VRML 97, Monterey, CA. http://www.parc.xerox.com/red/projects/web3d/mandala/sonic_mandala/index.html

[12] The VRML Consortium. *The VRML 97 Specification*. Available at <http://www.vrml.org/Specifications/VRML97/>

Appendix A: VRML 2.0 proto code, where initial values are set.

```
#VRML V2.0 utf8
```

```
PROTO ExSnd [
```

```
    eventIn SFBool enable
    field SFVec3f location 0 0 0
```

```
]
```

```
{
```

```
    SounderSet {
```

```
        enable IS enable    #turn it on upon approach
        location IS location #put it where the associated kinetic sculpture is
        sounderData [
```

```
            SounderData{
```

```
                sounder Sounder {
```

```
                    sound Sound {
```

```
                        direction 0 0 1    #sound faces screen
                        minFront 6.0
                        minBack 6.0
                        maxFront 31.0
                        maxBack 11.0
                        intensity 1.0 #initial loudness
                        source AudioClip {
                            url "sound/waterblocks/wb1.wav"
```

```
                    }
```

```
                }
```

```
                #the exact length of this particular sound file
                duration 0.08
```

```
                #the order in which the Java class processes this piece
                id 0
```

```
            }
```

```
            #time before sound first plays
            #initialWait [0.5, 0.5, 0, 1, 1, 1]
```

```
            #time between sound plays (loop=FALSE)
            startWaitP [0.1, 0.2 0 3 0.5 0.1]
```

```
            #loudness; pos vals only, 1 max
            intensityP [0.5, 0.1 0.5 1 0.5 0.1]
```

```
            #pitch of sample, fractional vals ok
            pitchP [0.5, -0.3 0.5 1.3 0.5 0.1]
```

```
            #x sound location (interacts with VRML ellipse def)
            xP [0, 0 -5 5 0.5 0.1]
```

```
            #y location (it's inited already up off the ground)
            yP [0, 0 -5 5 0.5 0.1]
```

```
            #z location
            zP [0, 0 -5 5 0.5 0.1]
```

```
        }
```

```
    ]
```

```
}
```

```
}
```