AudioScape: A Pure Data library for management of virtual environments and spatial audio

Mike Wozniewski
La Société Des Arts Technologiques
1195 Saint-Laurent boulevard
Montreal, Quebec
mike@mikewoz.com

Zack Settel
La Société Des Arts Technologiques
1195 Saint-Laurent boulevard
Montreal, Quebec
zack@sat.qc.ca

Jeremy R. Cooperstock
McGill University
Centre for Intelligent Machines
3480 University Street
Montreal, Quebec
jer@cim.mcgill.ca

ABSTRACT

We present a Pure Data library for managing 3-D sound and accomplishing signal processing using spatial relations. The framework is intended to support applications in the areas of immersive audio, virtual/augmented reality systems, audio-visual performance, multimodal sound installations, acoustic simulation, 3-D audio mixing, and many more.

Keywords

PD Library, Spatial Audio, 3-D graphics

1. INTRODUCTION

Pure Data (Pd) is a well-suited environment for the creation and control of various forms of media. As a result, many interesting projects have been realized, and a variety of novel tools are becoming available for artists and multimedia developers. The programming environment is however highly geared towards the control of audio processing, leaving interaction with visual content as added functionality that must be installed afterwards. Many libraries have been made available over the past few years to deal with image and video processing (e.g. GridFlow, PDP/PiDiP, Framestein, PixelTango), yet the management of 3-D content is still poorly supported.

The GEM library [3] is one available tool for generic control of 3-D objects. It allows users to create OpenGL scenes from within PD, however the interaction tends to be very low-level. Users need to ideally have some knowledge of 3-D graphics programming, since control of the scene is typically accomplished by loading primitives, manipulating vertices, and manually applying transformations to achieve a certain effect. This is not necessarily a problem for artists who wish to use PD for intricate 3-D modelling. It does however become quite tedious for those who wish to create game-like worlds, realistic virtual environments, or high-quality 3-D simulators. Furthermore, the GEM library has no built-in modelling of audio in 3-D space. As a result, spatial audio effects have to be computed independently from the visual 3-D display, which adds more memory and computation demands on the computer.

With our approach, we assume that other applications will be used for 3-D modelling (e.g. 3D Studio Max, Maya, Blender), and focus instead on the spatial arrangement of those models within the scene. Instead of providing users with control over individual vertices and transforms, a higher-level type of interaction is supported. Users can load models such as human avatars, buildings, concert halls, or any other type of 3-D object. Control of these models involves simple translation/rotation/scaling in space, but can also control keyframe animations exported with the model.

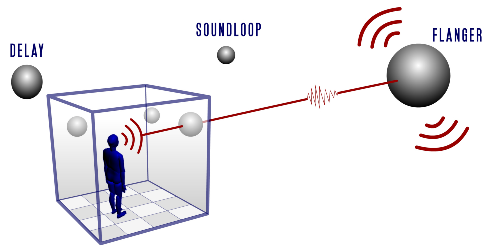


Figure 1: Immersion in a 3-D audio scene, where a user is surrounded by effects and sound sources.

The most important contribution of our library however, is the physical modelling of sound within the virtual scene. Users can place virtual microphones at arbitrary 3-D locations, and captured sound is processed according to the laws of physics. This includes simulated diffraction around objects, decay with distance, frequency-dependent damping due to absorption, and reverberation. Moreover, users can place digital signal processing (DSP) at localized positions in 3-D space, allowing for the modification of audio as it travels through space. This kind of flexibility allows for the creation of three-dimensional instruments where users can steer signals in 3-D space to control the musical output. This spatial audio processing along with the graphical rendering is processed and controlled in realtime.

2. VIRTUAL 3-D AUDIO SCENES

In our framework [7, 8], a virtual audio scene is defined by a number of *soundNodes*, *soundSpaces*, and *soundConnections.* The soundNodes are the basic building block of a scene, and contain parameters that model the spatial char-

id	A unique ID for the node
parent	The ID of this node's parent
dsp	A DSP plugin that computes audio
maps[]	List of mapping plugins that define control
gfx	Filename of a 3-D graphical model
pos = (x, y, z)	3-D position (in the local coordinate system)
scale = (x, y, z)	The scale of the object in three dimensions
radn = (θ, ϕ, ψ)	3-D vector representing direction of radiation (sound source orientation)
sens = (θ, ϕ, ψ)	3-D vector representing direction of sensitivity (sound sink orientation)
radnRolloff	A function or look-up table returning radiation values for various angles of incidence
sensRolloff	A function or look-up table returning sensitivity values for various angles of incidence
radnFactor	A distortion factor that affects the sampling of the radnRolloff
sensFactor	A distortion factor that affects the sampling of the sensRolloff

Table 1: List of important soundNode parameters.

acteristics of an audiovisual element, as well as logical parameters relating to control and DSP (listed in Table 1).

2.1 Steerable Sound

It is important to note that every soundNode has two independent steering vectors: one that describes the direction of radiation (*radn*), and one that describes the direction of sensitivity (*sens*). A soundNode can thus behave as both a sound 'source' that emits sound into the scene, and as a sound 'sink' that captures audio at a particular 3-D location. This is an important distinction from other 3-D audio engines, since it allows nodes to be used as signal processing units that perform a modification to sound at a specific 3-D location.

Sound effects nodes such as the harmonizers, flangers, or any other type of DSP can thus be created. However, instead of using patch cables or wires to connect different DSP units, our framework uses 3-D space as the medium for signal flow. There are no knobs or sliders to control levels. Rather, the audio that travels between soundNodes is processed (attenuated, delayed and filtered) according to the laws of physics. Manipulating the relative distance and orientation of soundNodes becomes the method of controlling audio processing. Thus, the principal operators for mixing (e.g. bussing, panning, filtering) become spatial in nature (e.g. translation, rotation, scaling).

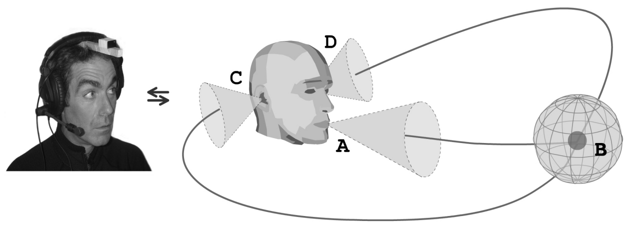


Figure 2: A scene modelled with AudioScape.

As an example, Figure 2 shows how a user wearing a headset can be modelled with soundNodes. The audio captured from the user's microphone is represented by a soundNode labelled A, which radiates the signal into the scene based on the user's head orientation and a *rolloff function*. Typically, a person's voice travels in all directions as a spherical wavefront. With our framework, users have the ability to choose the directivity of their sound signals by defining rolloff functions.

Figure 3 shows an example of a soundNode and its radiation. The sound signal will radiate with full gain (value of 1.0) along the direction specified by the *radn* vector, and fall off according to the *radnRolloff* function. Hence, we can think of a 'rolloff' as some function or procedure that provides gain values given an *angle of incidence* to the steering vector.

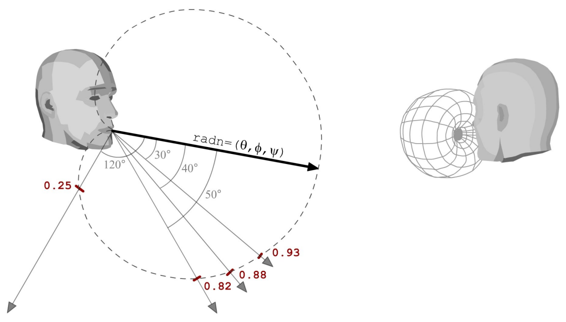


Figure 3: Directivity of a soundNode is shown. The sound radiates along the *radn* vector, and falls off according to a cardioid-shaped rolloff function.

In this case the rolloff function has a cardioid shape, but any type rolloff function can be defined by the user or specified by a look-up table if a desired effect cannot be described algebraically. For best results however, the function should evaluate to 1.0 if the angle of incidence is 0 radians, and should be a valid monotonically decreasing function to the range of π radians. The reason for these requirements is that there is also an additional control parameter called *radnFactor* (or *sensFactor* in the case of a sink node). This parameter allows for the distortion of a rolloff function, essentially by stretching or squishing it.

2.2 Physically Modelled Audio Propagation

Moving our attention back to the example in Figure 2, we notice three other soundNodes. The node labelled B is a sound processing unit with omni-directional sensitivity and radiation. It will capture sound from the user's microphone, apply some modification, and emit the result back into the scene. Nodes C and D represent the two loudspeakers of the user's headphones, and will capture the resulting audio signal for playback in the headset. It should however be noted that the captured sound is first processed to simulate the effects of travelling through 3-D space. This is managed using the *soundConnection* mechanism.

Rather than forcing an identical propagation model for every node, we construct a directed soundConnection between every source-sink pair in the scene. This soundConnection is in fact a data structure, containing various parameters that describe how sound travels between the nodes, allowing a user to specify whether the sound should be delayed, whether it should decay with distance, and whether it should be filtered to simulate diffraction and absorption. Furthermore, soundConnections need not exist for every possible pairing in the scene. They are only created as needed, to specify exactly which soundNodes can feed others with sound. If a soundConnection does not exist between two nodes, then the physical modelling of sound propagation between those nodes is not computed, and precious CPU cycles are saved.

In the example that we have been studying, there are exactly three soundConnections present: $(A \rightarrow B)$, $(B \rightarrow C)$, and $(B \rightarrow D)$. There is no connection from $(B \rightarrow A)$, since that would cause a feedback loop. Likewise, there is no connection from $(A \rightarrow C)$ or $(A \rightarrow D)$, because the user wants to hear only the processed voice in this case, and not the direct (raw) signal. If desired, the user could create such connections and mix in some of the raw signal. Such an effect would be similar to how we hear ourselves in nature. We thus start to see the power of the connection mechanism, since it allows the user to specify exactly how audio travels within the environment.

It is important to note that our audio porpagation model can *deliberately* violate the rules of physics. The fact that audio can be tightly focused and travel only along a narrow pathway is obviously unnatural. Furthermore, for the purpose of musical creation and performance, users might not desire realistic models of sound propagation. Rather, they may wish to exaggerate or diminish certain acoustic properties for artistic purposes. For example, Doppler shift is often emphasized in sound tracks for cinema because it adds dramatic effect. Also, to conserve timing and intonation in musical pieces, it may be desirable to eliminate distance-based delay of sound. Allowing for such rule-bending is a useful feature of our system.

2.3 Volumetric Sound Processing

In nature, the size, shape, and the types of objects within an environment will affect the propagation of sound. Our brains are remarkably adept at uncovering this hidden information and inferring properties about our surroundings. Users expect these acoustic cues when they travel through virtual spaces, hence we provide a mechanism for defining enclosed areas within a scene. These *soundSpaces* are similar to soundNodes, yet rather than capturing and emitting audio at a fixed point in space, they operate on a volume defined by an arbitrarily shaped 3-D model (defined in 3D Studio Max, Maya, Blender, etc.). Signals from within the volume will be captured with unity gain, while signals from outside the volume will be captured and attenuated according to the emitter's distance from the boundary and the absorption coefficient of the volume's surface.

The reverberation model for soundSpaces allows for the specification of delay times (for direct sound, and 1^{st} & 2^{nd} order reflections), a reverberation time, and a filter to simulate frequency-dependent damping over time. However, if the user wishes, this model can be replaced with any type of DSP plugin such as a flanger or harmonizer. Thus, rather than walking into an enclosed space and hearing the reverbaration of your voice or instrument, you would hear a flanged or harmonized version of your sound signal instead. This offers many interesting possibilities for artistic creation.

3. ARCHITECTURAL DECISIONS

In the case of real-time applications for 3-D gaming and virtual reality systems, several toolkits and APIs are available, including Microsoft DirectX, OpenAL, X3D, and MPEG-4. These toolkits allow developers to move sound sources around a virtual scene and have them properly rendered. Unfortunately, these toolkits are not sophisticated enough for highly interactive sound applications, particularly for those geared towards artists and musicians. They offer only simple mechanisms to integrate spatial audio into applications, and often have an impoverished audio representation.

For instance, most APIs have no method to specify the directivity of sounds and instead consider all sources as omnidirectional. In the cases where directional sounds are supported, these are typically implemented with linear attenuation applied as a function of angle (e.g. X3D [1]). There is no support for complex radiation patterns that are emitted by traditional musical instruments, or the cardioids that are commonly found in audio equipment. Furthermore, most APIs only support a single listener located at one position in space, and lack the possibility to perform 'unnatural' sound propagation. In general, it is difficult to manage more complex sonic interactions between objects that may be important to artists. For example, it might be interesting to copy signals to different areas of the scene, attach two sets of headphones to a soundcard for two different users, or boost the Doppler Effect for dramatic purposes.

One noteworthy standard for describing interactive audio scenes is AudioBIFS from MPEG-4 (ISO 14496-1) [6]. The representation of 3-D audio is quite interesting and serves as inspiration for the work presented here. AudioBIFS borrows the scene-graph hierarchy from VRML to organize nodes, but introduces specialized audio processing nodes such as 'Sound', 'AudioSource', 'AudioFX', and 'ListeningPoint'. These can be attached to graphical objects anywhere within the scene, allowing developers to create scenes for interactive mixing and musical control within a 3-D context.

Our work uses an similarly uses a scene graph organization using the open-source library called OpenSceneGraph (OSG) [2] to manage the arrangement of objects in space and efficiently control the geometric transformations performed on them. The main difference between our system and AudioBIFS, is that we provide the ability for dynamic control and reconfiguration of the entire scene in real-time using Pure Data. BIFS on the other hand are binary in format and cannot change dynamically. The scene must be authored in advance and then compiled for distribution. Interactivity is accomplished by allowing certain fields to be *exposed* and updated by a content server. The developer must pre-define all possible interactions during the authoring phase. In our system, there is no differentiation between an authoring mode or interactive mode; they are one and the same.

3.1 OpenSceneGraph

OpenSceneGraph (OSG) is a graphics toolkit that provides a high-level interface to OpenGL. Rather than writing and optimizing low-level graphics calls, OSG allows a developer to concentrate more on the organization and interaction of 3-D content. We aim for typical scenes to be composed of several independent 3-D models such as people, musical instruments, audio equipment, and architectural elements such as furniture, walls, and floors. The spatial interactions between these elements will likely be high-level, involving simple behaviours like translation, rotation, and scaling. We anticipate that the need to control these models at the vertex level will be rare, although the ability to animate various components of these models will be required - for example, articulating the body parts of a human avatar. With these constraints in mind, we have adopted OSG, mainly because of the *scene graph* data structure, which provides a hierarchical organization for virtual scenes.

A scene graph is simply a tree-shaped data structure that organizes the logical and spatial elements of a 3-D scene. Each element is represented as a node in a tree, including things like models, transformations, lights, textures, and cameras. The interesting thing about this organization is that it provides an efficient method to propagate information to groups of objects. Particularly, any spatial transformations performed on a parent node will be automatically propagate to all child nodes. If we again consider the example in Figure 2, we note that the user's headset is a rigid structure, with nodes A, C, and D sharing a common geometric reference. If the user's head moves or rotates, then all three of these nodes should move and rotate together. We thus set the 'parent' for the user's ears (C and D) to be node A and then whenever that node is updated, the change will automatically propagated down. Figure 4 shows the resulting scene graph. It also shows that node B is not affected since it has no geometric relation to the user's head.

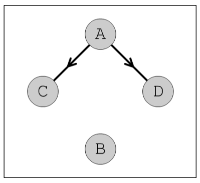


Figure 4: The scene graph associated with Figure 2.

4. THE 'AUDIOSCAPE' LIBRARY

In order to use OpenSceneGraph within the Pure Data environment, we use the API provided with Pd to create custom externals that encapsulate OSG classes and utility functions. The library is written in C++, however the Pd API is written in C, so we use the extern "C" declaration to allow the use of a C++ compiler. This still allows Pd to instantiate our objects, yet gives us the ability to incorporate code from OSG. The end functionality is a set of patchable objects containing references to OSG classes that can be placed on canvases and updated using the Pd messaging system. For example, the [soundNode] object[1] extends the `osg::Referenced` class so that OSG can maintain reference counting and automatic garbage collection. It also contains pointers to instances of other classes (`osg::Node`, `osg::Group`, `osg::PositionAttitudeTransform`, etc.) that are used to manage the scene graph and the associated parameters and methods. Three other core objects exist as part of the library: [AudioScape], [soundSpace], and [soundConnection].

The [AudioScape] object is the overall manager of the virtual scene and allows for the creation (and destruction) of other library objects. Table 2 lists the various methods that are available.

It should be noted that AudioScape's *createNode*, *createSpace*, and *connect* methods are simply helper functions. A user can also just place a patchable [soundNode] object on a Pd canvas, and it will become properly registered with the

[1]Square brackets indicate an patchable Pd object.

createNode [nodeID]	Creates a soundNode with the given nodeID
createSpace [nodeID]	Creates a soundSpace with the given nodeID
destroy [nodeID]	Destroys the soundNode with the given nodeID
connect [srcID] [snkID]	Create a soundConnection between the srcID and snkID
disconnect [srcID] [snkID]	Destroy the soundConnection between the srcID and snkID
saveXML [filename]	Save current scene to an .xml file
loadXML [filename]	Load a scene from an .xml file
clear	Clears the current scene

Table 2: Messages recognized by [AudioScape].

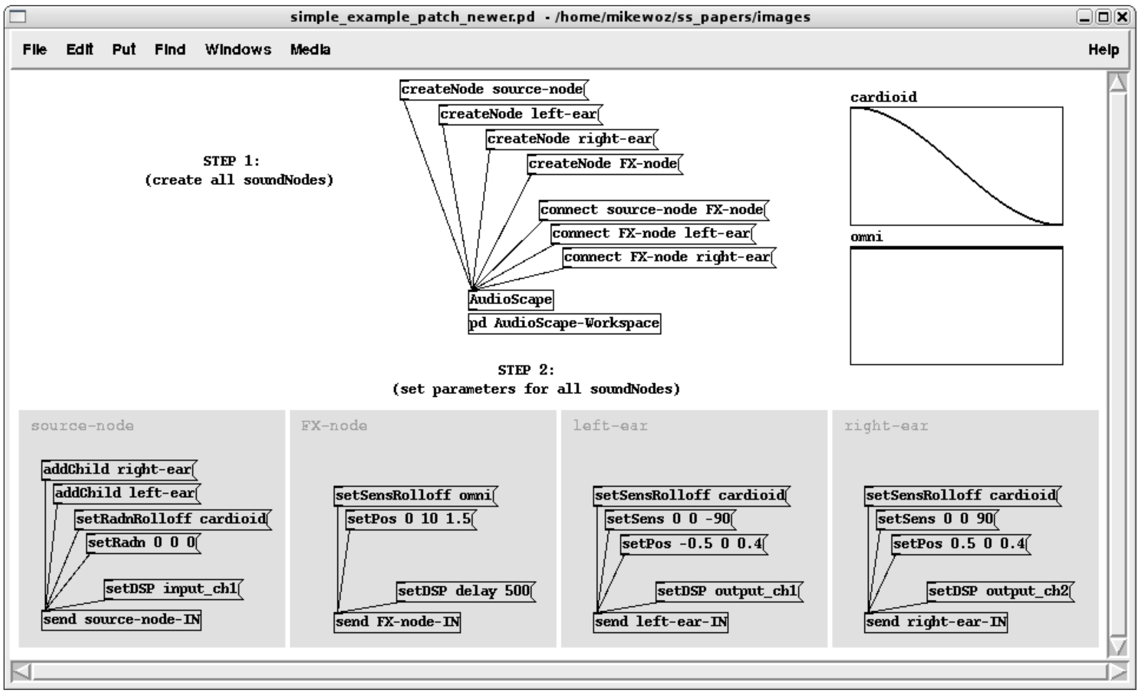
AudioScape engine. The helper methods are however convenient since with one command, they create all the appropriate Pd objects and connect them as necessary.[2] In fact, there is also an very useful abstraction called [node-wrapper] which can (and should) be used instead of directly placing instances of the objects on canvases. The [node-wrapper] contains various subpatchers with a variety of helper algorithms. These include the necessary [send] and [receive] objects to pass messages to the nodes, as well as mechanisms for dynamic creation and deletion of DSP and mapping plugins (which we will discuss below).

4.1 Example Using Pd Patches

Given the framework described so far and the simple example seen earlier in Figure 2, we will now look at how one can build this scene using Pd patches. Various message boxes need to be defined and the appropriate object boxes need to be instantiated. Figure 5 shows a screenshot of such a patch. First, the [AudioScape] object must appear somewhere, thereby instantiating the library and making all objects accessible. The construction of the scene begins by sending messages to [AudioScape] that create each of the four soundNodes (source-node, right-ear, left-ear, and FX-node). In doing so, the library will dynamically create the appropriate [node-wrappers] on the 'AudioScape-workspace' canvas. Conveniently, the [node-wrapper] sets up the appropriate objects to receive messages for each soundNode. These receivers have a standard naming convention with "-IN" appended to the node's id. Thus, to update parameters of the source-node, one can simply send a message to "source-node-IN".

After the soundNodes have been created, parameters are set so the nodes behave like the scene in Figure 2. Note for instance that the right-ear and left-ear soundNodes have been added as children to the source-node. This means that the ear positions and orientations will be described in a local coordinate system relative to the source-node. The left-ear position of $(-0.5, 0, 0.4)$ is not an absolute position, but is rather the offset from the position of the source-node, locating the ear 0.4 units above the source-node and 0.5 units to the left. Each ear also has cardioid sensitivity pointing 90° away from the direction in which the source is radiating its sound.

[2]This is accomplished using internal messages to Pd [4], which allow for dynamic creation and connection of Pd patches without using the mouse and keyboard.

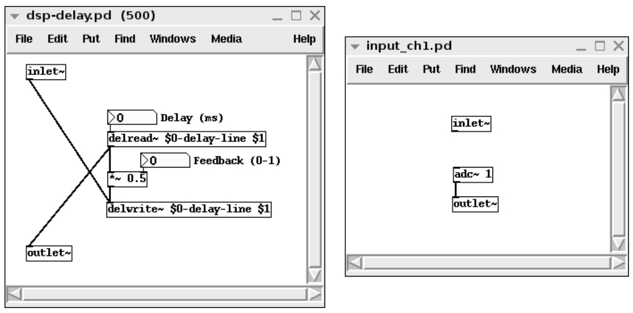

simple_example_patch_newer.pd - /home/mikewoz/ss_papers/images
File Edit Put Find Windows Media Help
cardioid
omni
source-node
FX-node
left-ear
right-ear

```
STEP 1:
(create all soundNodes)

createNode source-node
createNode left-ear
createNode right-ear
createNode FX-node
connect source-node FX-node
connect FX-node left-ear
connect FX-node right-ear
AudioScape
pd AudioScape-Workspace

STEP 2:
(set parameters for all soundNodes)

addChild right-ear
addChild left-ear
setRadnRolloff cardioid
setRadn 0 0 0
setDSP input_ch1
send source-node-IN

setSensRolloff omni
setPos 0 10 1.5
setDSP delay 500
send FX-node-IN

setSensRolloff cardioid
setSens 0 0 -90
setPos -0.5 0 0.4
setDSP output_ch1
send left-ear-IN

setSensRolloff cardioid
setSens 0 0 90
setPos 0.5 0 0.4
setDSP output_ch2
send right-ear-IN
```

Figure 5: The Pd patch associated with Figure 2.

4.1.1 *DSP plugins*

The ultimate aim of this framework is to help artists to organize DSP processes in order to build complex interactive musical systems. For this reason we keep the DSP methodology quite generic, allowing a user to specify almost arbitrary audio processing within each soundNode. The soundNode's *dsp* parameter is simply a pointer to a Pd plugin, and the user is free to build any type of processing within that patch. The one constraint is that the abstraction must have only one input and one output (i.e., a monophonic patch). This may seem like a significant constraint, but polyphonic DSP can easily be accomplished by grouping nodes together in a scene graph and copying audio signals between them.

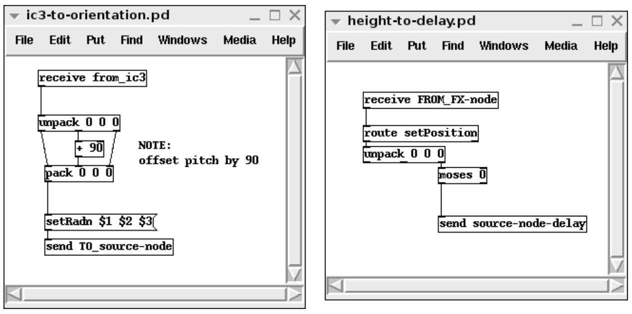

dsp-delay.pd (500)
File Edit Put Find Windows Media Help
input_ch1.pd
File Edit Put Find Windows Media Help

```
inlet~
0 Delay (ms)
delread~ $0-delay-line $1
0 Feedback (0-1)
*~ 0.5
delwrite~ $0-delay-line $1
outlet~

inlet~
adc~ 1
outlet~
```

Figure 6: Two DSP plugins: A delay patch that takes the delay time as an argument (left), and a patch to read input from the soundcard (right).

Hence, considering again our example, we define DSP plugins for each ear. The function of these plugins will be to send the captured audio signal to the soundcard's output channels (presumably to a pair of connected headphones). The DSP plugin for the source-node on the other hand, accepts the soundcard's input channel (presumably a microphone) and emits the result into the scene. Figure 6 shows the latter plugin. Note that the inlet is ignored, and a sound signal is only connected to the outlet of the patch. This is indicative of a 'source' soundNode. A 'sink' soundNode would on the other hand accept a connection from the inlet and ignore the outlet. The figure also shows an FX-node, which uses both; it takes the incoming signal, delays it, and outputs the result into the scene, thus acting both as a source and sink.

4.1.2 *Control & Parameter Mapping*


ic3-to-orientation.pd
File Edit Put Find Windows Media Help
height-to-delay.pd
File Edit Put Find Windows Media Help

```
receive from_ic3
unpack 0 0 0
+ 90
pack 0 0 0
NOTE:
offset pitch by 90
setRadn $1 $2 $3
send TO_source-node

receive FROM_FX-node
route setPosition
unpack 0 0 0
moses 0
send source-node-delay
```

Figure 7: Two mapping plugins. The left routes orientation sensor data to the radiation direction of a soundNode. The right maps node height to the delay time of its corresponding DSP plugin.

Similarly to providing arbitrary DSP possibilities, we allow for almost any mappings to be defined using plugins created in Pd. These are simple Pd patches that can intercept messages (from sensors, metronomes, or any soundNode parameter update) and re-route them as the user sees fit. For example, in Figure 7 we see a patch that intercepts data received from an orientation sensor and uses it to control the radiation direction of a soundNode. The other patch in that figure takes the current height of a soundNode to control the delay time of the DSP.

5. GRAPHICAL RENDERING

It should be noted that the OSG is mainly used as a 3-D rendering library. It is thus very easy to render our virtual audio scene graphically for visual feedback and as an additional component of the user interface. One could design an application for just one laptop and a set of headphones, but if immersion is desired then multiple screens are needed and hence multiple rendering machines need to be deployed. As it turns out, OSG provides efficient mechanisms for distributed rendering by employing clever algorithms for scene graph data structures. By keeping track of the bounding region for each node's subgraph, OSG can prune entire sections of a scene that do not need to be rendered. The only thing that we need to do, is set the camera view for each machine, and an efficient distribution of rendering is realized, where anything not in the viewing volume will not need to be computed.

As a result, we have written a custom OSG application that does not require Pd, and can be executed on a number of rendering machines. This application includes a daemon that listens and responds to network messages from a master controller. Messages can, for example, be sent to set the camera view. Yet more importantly, messages are sent that construct and maintain the state of all soundNodes in the virtual world. With this architecture, users can develop immersive, multi-screen deployments that graphically show what is happening within their virtual audio scenes.

6. CONCLUSIONS & DISCUSSION

We have described a framework for interacting with sound in 3-D and for creating spatially organized audio processing. This framework is supported by a software library written using OpenSceneGraph and the Pure Data API. It allows users to create virtual worlds that function as musical instruments and interactive sonic applications. We have shown that a user can be modelled within such a virtual scene using soundNode and soundConnection objects. DSP and mapping plugins provide mechanisms for creating arbitrary interactions between the various compenents in the audio scene, and even input from external devices. When all of these features work together, they provide a powerful multimodal environment that can lead to many novel and interesting applications.

The power of the framework is realized by the sophisticated representation of soundNodes and their acoustic propagation model. Providing users with the ability to control directivity of sound with high accuracy, and allowing the rules of physics to be bent are both essential factors that are necessary for using virtual environments in a musical context. The traditional APIs and toolkits mentioned earlier are too focused on simple spatialization tasks, and cannot be used as effectively for musical purposes. Furthermore, the fact that soundNodes can act as both sources and sinks allows for localized DSP within a 3-D scene, which can accomplish extremely complicated audio processing tasks. This enables users to create virtual worlds that simulate almost any piece of equipment that can be found in a sound studio (effects processors, synthesizers, mixing consoles, etc.).

Typical musical tasks can thus be accomplished using completely new metaphors, since they take place in continuous 3-D space, and do not need to be logically organized with patch cords and sound modules. Users will tend to think of spatial relations as the factor that influences the sound rather than the positions of sliders or knobs. This is powerful idea, since humans have a great sense of spatial understanding that they have acquired from the real world. Feedback from the system is thus more natural and subconscious, and less cognitive load is imposed on the performer when trying to keep track of the state of an application.

The library along with supporting patches has been made available for public download and use. A suite of editors and sample applications, packaged as "AudioTwist", is also available [5]. The project will hopefully expand as new developers begin to add their own ideas and creations. In time, we expect that several artists will be able to create novel works of art, cutting-edge performances, and interesting installations using the framework. We are eager to see how this new way of thinking about music and sound in 3-D will evolve.

7. ACKNOWLEDGEMENTS

The authors wish to acknowledge the generous support of NSERC and Canada Council for the Arts, which have funded the research and artistic development described in this paper through their New Media Initiative. In addition, the authors are most grateful for the resources provided to the project by McGill University's Centre For Intelligent Machines, La Société Des Arts Technologiques, and The Banff Centre for the Arts. These research centres have provided productive and stimulating environments allowing for the development and prototyping of this project.

8. REFERENCES

[1] Extensible 3D (X3D) ISO standard (ISO/IEC 19775:2004). http://www.web3d.org/x3d/specifications/ISO-IEC-19775-X3DAbstractSpecification/.

[2] OpenSceneGraph. www.openscenegraph.org.

[3] M. Danks and IOhannes m zmölnig. Gem. http://gem.iem.at/.

[4] D. Henry. Pd internal messages. http://puredata.info/community/pdwiki/PdInternalMessages.

[5] La Société des arts technologiques. The Open Territories project. http://tot.sat.qc.ca.

[6] E. Scheirer, R. Väänänen, and J. Huopaniemi. AudioBIFS: Describing audio scenes with the MPEG-4 multimedia standard. In *IEEE Trans. Multimedia*, volume 1, pages 237–250, 1999.

[7] M. Wozniewski, Z. Settel, and J. R. Cooperstock. A framework for immersive spatial audio performance. In *New Interfaces for Musical Expression (NIME), Paris*, pages 144–149, 2006.

[8] M. Wozniewski, Z. Settel, and J. R. Cooperstock. A paradigm for physical interaction with sound in 3-D audio space. In *Proceedings of International Computer Music Conference (ICMC)*, 2006.