Sound Playback for Art

Art galleries are no longer the hushed temples of times past. Now that the art is liberated from frames and pedestals it is free to make some noise.

An artist may use sound in a variety of ways:

- Background: music or ambiance
- Narrative: voices describing the art or as part of the art
- The piece may make sound on its own or as a performance instrument
- The art may react sonically to stimuli
- The art may react to sonic stimuli
- The art may be derived from sound: from music or ambience
- The sound may be derived from the art.
- Sound and art may be performed together.

Max patches can be the glue that holds all of this together. If we leave synthesis alone, most of these operations are based on playing sound files on demand. The "demand" can come from a multitude of operations as covered in other tutorials—video hotspots, MIDI or Arduino sourced triggers, or just a gallery curator opening up for the day. In addition to playing the files, some processing may be in order, again in response to external stimuli.

For tips on generating images from the sounds iTunes-style, see the tutorials on visual art and Lissajous art. For one way of getting sounds out of images, see the Metasynthesis tutorial.

Simple File Playback

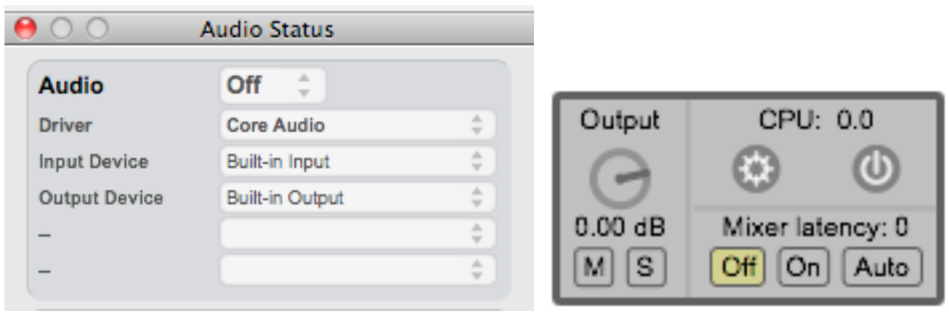
When sound or music is the background to a static installation, there is little point in using a computer. An iPod will do the trick, or even something as basic as a CD player. These are economical, simple to operate and very dependable.

However, if the computer is there anyway, it makes sense to give it the responsibility for playing the background sounds. In Max, all audio[1] runs through MSP, the Max Signal Processing environment.

The DSP Status Window

MSP operations are set up in the Audio (or DSP) status window, which is found under the Options menu. There are a couple of vitally important settings in this window, and several that matter only in high performance situations.

[1] Well, most audio. See the section about QuickTime movies and sound.



Audio	Off
Driver	Core Audio
Input Device	Built-in Input
Output Device	Built-in Output
–	
–	

Output: 0.00 dB

CPU: 0.0

Mixer latency: 0

Figure 1. Audio Status and Window Mixer in Max 6.

The left section in figure 1 shows the crucial items.

- Audio must be turned on for anything to be heard. (This is not the only place audio may be switched.)
- The driver must match the desired audio interface. CoreAudio Built-in is the choice for the internal Macintosh sound system.
- Input device and source will determine where audio comes in.
- Output destination will determine where the sound can be heard.

You can't count on default settings in this window. In fact, the default audio driver is None. If others are to operate your patch, you must leave clear instructions about how to set these options.

The right side of figure 1 shows the "Mini-Mixer" that is attached to each window in Max 6. This has several features that become useful in a complex application:

- There is a master output level for the window.
- There are also mute and solo buttons for the window.
- The CPU load of that window is displayed—this can help track down power hungry sections of a patch.

The on, off and auto buttons affect what happens when you change the patch while audio is running. The on button enables a crossfade from the old patch to the new version—this prevents dropouts and pops. There is inherently some delay when this happens, which is shown as the latency. I'd turn it on only when live coding the audio.

The sfplay~ object

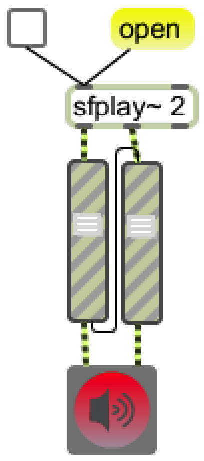


Figure 2.

Figure 2 shows the basic patch for playing an audio file. The open message to sfplay~[2] will bring up a file dialog. You can include a file name in the open message (in quotes if it includes spaces) as long as the file can be found in one of the folders specified in Options>File preferences.

The file will begin playing from the beginning when the toggle is set. The two sliders will adjust the volume. Note that they are connected together by a patch from the right outlet of the left slider to the left inlet of the right slider. If you move the left slider, the right will follow along. These are actually gainslider~ objects, which are distinguished from normal sliders by the stripe patten.

The object with the speaker icon is ezdac~, the connection to sound output. This icon is also a button- click on it to turn audio on or off. The color shown is not the default (which is too subtle for me). Change colors in the Options menu with object defaults.

Notice the yellow and black patch cords. These cords carry signal. Signals are continuous data, as opposed to messages, which only happen when an object is triggered. The continuous data consists of numbers[3] at the sample rate, probably 44,100 per second. The numbers range from -1.0 to 1.0. If the values are allowed to get above 1, they will produce distortion when converted to sound. (Large values are OK for certain other signals.)

Sfplay~ can play at variable speed (the right inlet controls speed, 1.0 is normal) and will respond to pause and resume commands. The command loop 1 sets looping mode, and loop 0 turns it off.

[2] Note that the file opening command is “Open”. Most Max objects that access files do so via “Read”.
[3] They are grouped in batches called “signal vectors”. This makes processing more efficient but delays the sound a bit.

File Management
The toughest problem with simple file playback is finding the files to play. The most convenient method is to have the file names listed in a umenu. That provides mouse based access to the files and a mechanism for stepping through a series of files.

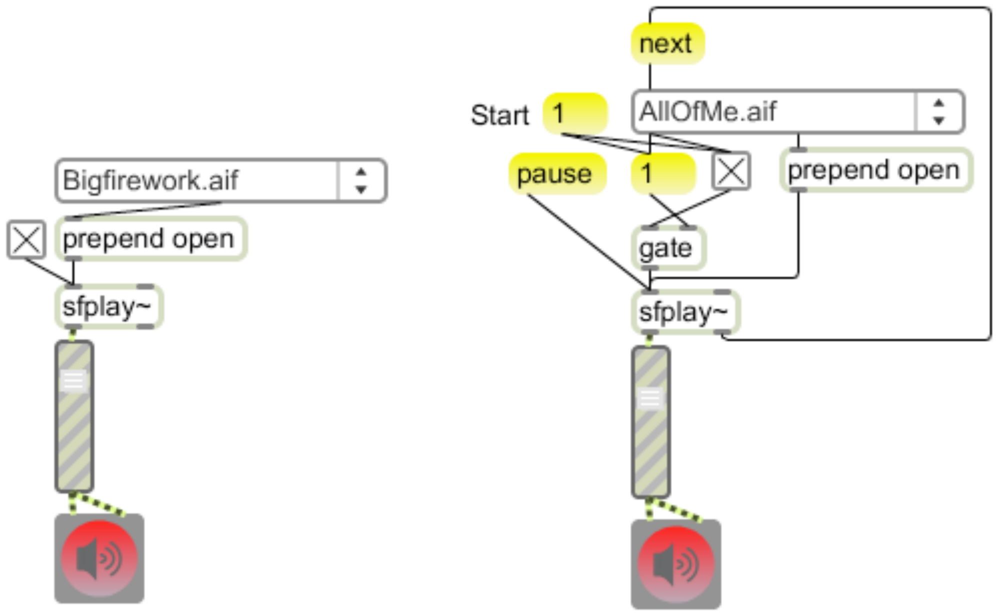


Figure 3.

Figure 3 shows how to use a umenu object to load and play files from a list. The left hand patch simply lets you choose files and play with the toggle. The right hand patch will play all of the files in the folder and stop when the first file comes up for the second time. This takes advantage of the fact that the umenu object sends the name of the menu item from the center, then the number from the left. You must select the first file to load it. (The umenu shows the first item in its list when a patch opens, but nothing is sent out until an item is chosen.) After the file loads, click the message 1 by start. This will open the gate and send a second 1 through to sfplay~. After the file has finished playing, sfplay~ will bang from the right. That sends a next message to the umenu to load the next file on the list. The process continues until all of the umenu items have been played, but when the first one comes back its index of 0 opens the gate and stops the process. The only way to stop at any other time is the pause command, because a stop or 0 directly to sfplay~ produces a bang from the right outlet.

This is just one approach to playing a list of files. The umenu can be activated by item number, so mechanisms could be based on counter, urn (unique random number) or an external input.

You don't necessarily need to copy the filenames into the umenu by hand. Figure 4 shows two ways to automatically fill a umenu.

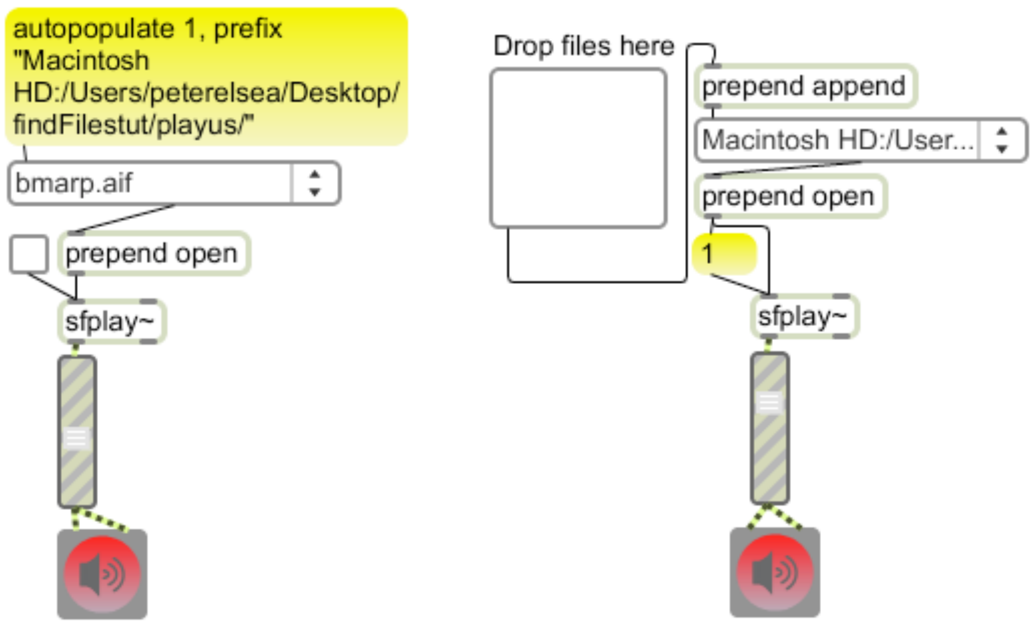


Figure 4.

There are two commands in the big message on the left section. *Autopopulate* 1 sets the umenu to list the files found in a folder sent in via the prefix command. This will list all files in the folder, but these can be restricted with a types message. The command *prefix* sets a symbol to be prefixed to anything the umenu sends. If the prefix is the path to the folder, sfplay~ will find it.

The right section of figure 4 shows how to fill a umenu with dropfile. When a file is dragged from a finder window to dropfile, the path is sent from the left outlet. The append message installs the complete path in umenu. Of course you could also connect the dropfile object directly to the prepend open object. Then files wound be loaded and play immediately.

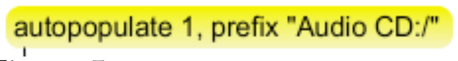


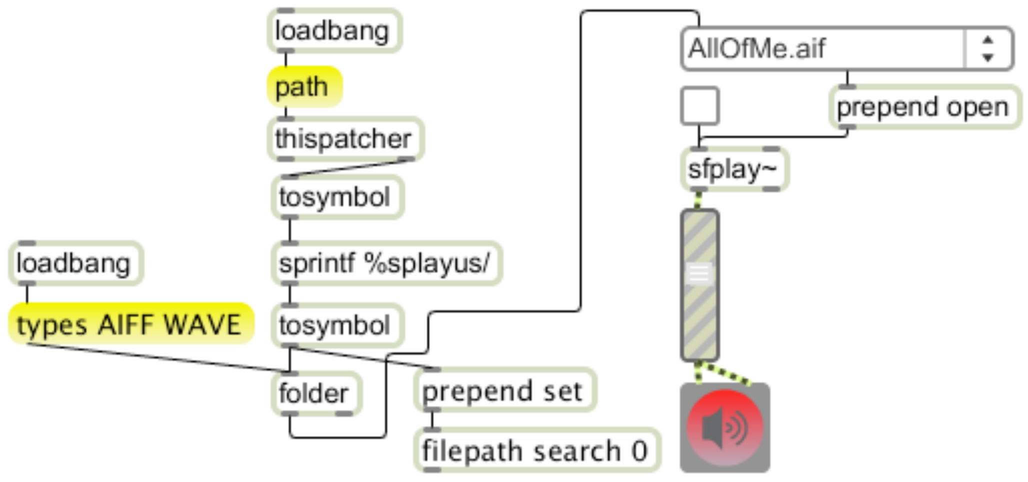
Figure 5

The message in figure 5 will let you play tracks from a CD. It's all the same to sfplay~, but once you have loaded a file from a CD, you will not be able to eject the CD until you send an *fclose* message or open a file from a different location. Opening CD files takes time so copy files you need immediately to the hard drive.

Of course, finding the complete file path to a folder is no trivial matter, especially if the patch is installed by someone else on an unfamiliar machine. Figure 6 is the ultimate in file finding patches. It depends on the keeping the files to play in a folder in the same folder as the patch. This is how it works:

- The path message to the *thispatcher* object reveals the current location of the patch.
- The path is converted into a single symbol to accommodate spaces.

- *Sprintf*[4] concatenates the path and the name of the folder containing files to play. %s is an instruction to *sprintf* meaning "the input symbol". The folder name in the example is "playus".
- *Tosymbol* gain ensures that the complete path is a single symbol.
- Sending this path to *folder* will generate the list of files for *umenu*. The types message tells *folder* what type of files to include.
- Sending the path to *filepath* with the search argument will add the folder to the current File Preferences. The final 0 in *filepath* prevents this from being saved permanently in the preferences.



Courtesy of Luke DuBois

Figure 6.

A word about Loadbang

Note that figure 6 uses loadbang to set some important parameters. Loadbang produces a bang when the patch is loaded (or the object itself is double clicked.) I often omit loadbangs from examples to keep the graphics clear, but you should use them liberally. No patch is complete until it opens up completely ready to play, with all files loaded and all initial parameters set. You should practice this—save the patch, close the patch and open the patch. If it no longer works, there is some parameter that needs to be set by loadbang or loadmess.

[4] Sprintf stands for "String print with Format". There is a thorough discussion in the Max & ASCII tutorial.

Playing Cues

Many projects require quick playback of short snippets of sound. The best way to manage this is to put all of the sources in a single audio file and note their start and end times[5]. Sfplay~ can be instructed to play a section of a loaded file with the *preload* message. Preload takes up to four arguments:

- A cue number.
- The start time in milliseconds.
- The end time in milliseconds.
- A flag (1 or 0) to enable reverse play.
- A speed factor. This is in addition to any speed change done at the inlet.

Figure 7 shows how to preload cues. This particular file is a mono file with percussion samples spaced at 1.2 second intervals. The open message will load the file.

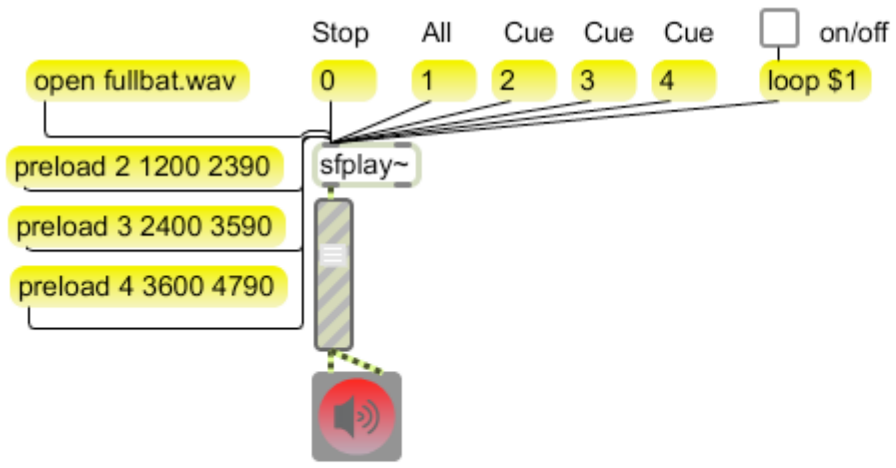


Figure 7.

Cue 2 will start at 1200 milliseconds into the file, then play to the 2390 millisecond point. The other cues are similar. With these set:

- The message 0 stops playback
- The message 1 plays the entire file from the beginning. (You can't use 0 or 1 as a cue number.)
- Other numbers play the cues as set up.
- The loop message works for individual cues as well as the entire file.
- Loopone with a cue number will set only that cue to loop mode.

Only one cue will play at a time. If you require multiple layers of sound from the same file, you need to open it in two independent sfplay~ objects. Several sfplay~ objects can share the same cue list via an sflist~ object. Figure 8 shows how. Each sfplay~ and the sflist~ object are given the same name. The file is

[5] It's essential to include some silence between clips. Making all clips the same length simplifies the patches.

opened in sflist~ and the cues preloaded in sflist~. When cue numbers are sent to each sfplay~, multiple cues will sound together. You cannot play the whole file with a 1 message unless the file is also loaded into an sfplay~.

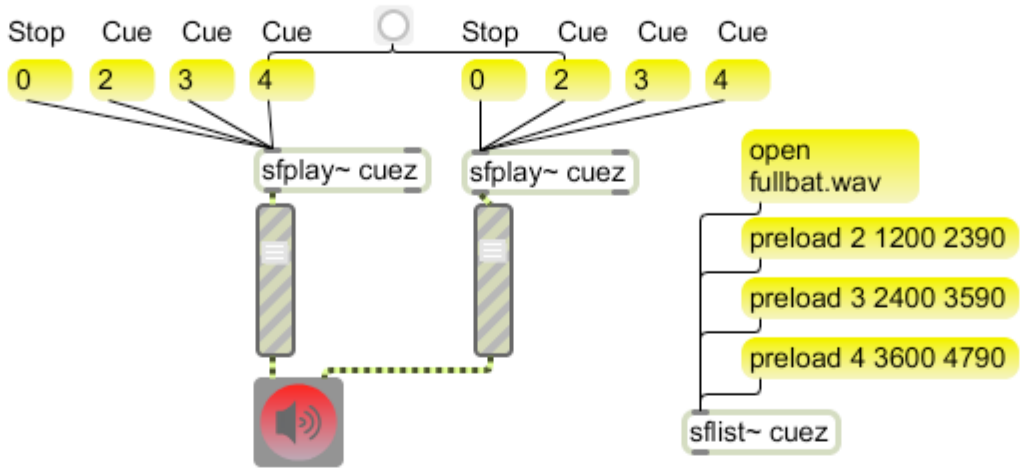


Figure 8.

Groove~

Sfplay~ is good solution if there are relatively few cues and the load on the hard drive is not heavy. If you are trying to play a couple of movies and stream several audio files at the same time, the drive may begin to choke. We can load modest amounts of audio directly into memory with the buffer~ object. Figure 9 shows how this works.

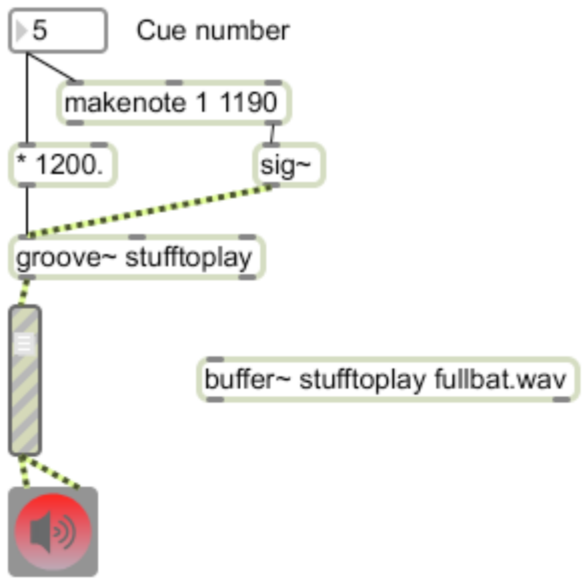
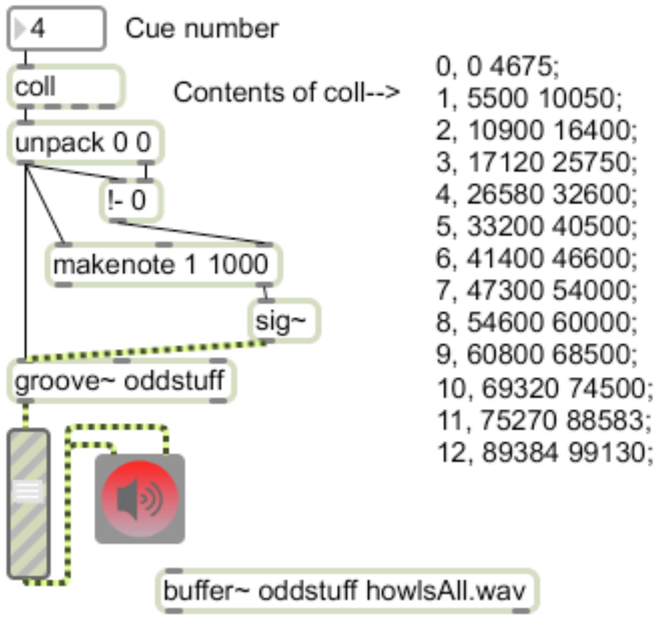


Figure 9.

The source audio file can be specified as an argument to buffer~ then buffer~ will automatically adjust its memory size to the file. Buffer requires a constant signal of 1.0 to run—that's an audio signal that consists of 44,100 1s each second. The sig~ object converts a float value to a constant signal. A signal of 0.0 will stop the buffer, and other values will vary the speed. In figure 9, the source of the value is the makenote object. Usually, makenote is used to produce MIDI notes, sending note number and velocity at the start of a note, followed by the same note number and a velocity of 0.0 at the end of the note. Actually, MIDI note is a quick and easy way to time all sorts of things. Setting the velocity (first argument, the second is duration) to 1 provides the proper signals for groove~.

A plain number will cue groove~ to that position in the file. Makenote is triggered by a number in the left inlet, and this number is passed through the left outlet. The file loaded has equally spaced clips so the duration is always the same and the cue position can be calculated, but how about cueing a file that has unevenly spaced clips?



```
0, 0 4675;
1, 5500 10050;
2, 10900 16400;
3, 17120 25750;
4, 26580 32600;
5, 33200 40500;
6, 41400 46600;
7, 47300 54000;
8, 54600 60000;
9, 60800 68500;
10, 69320 74500;
11, 75270 88583;
12, 89384 99130;
```

Figure 10.

Figure 10 shows how to manage assorted clips with a coll. Coll is an object that organizes data. The contents of this coll are shown in a comment--in use you double-click a coll to view and edit its data. The formatting of coll data is very strict:

- Each line starts with a unique address, which may be a symbol or number.
- The address is followed by a comma.
- The comma is followed by the data at this address, which can be a list.
- Each line must end with a semicolon.

Make a mistake in entering this data and everything after that will get lost when you try to save it. There are two steps to saving the data. When you close the window after editing, a dialog will appear asking if you want to save the changes. This only copies the data to the coll and puts it into effect. The data will not be saved with the patch unless you open the coll inspector and check an option to “save data with patcher”. You may optionally save the data as a text file by choosing the save as option while the editor is open. The data may be loaded into the coll with a read message, or you can use the name as an argument to the coll and the file will be loaded when the patcher is opened. In addition, all colls with that name will share the same data.

The coll in figure 10 has the start and end time (in milliseconds) for each numbered cue. Sending the cue number to the coll prompts the output of a list with the start and end. This is unpacked—the difference between start and end (!-)[6] is applied to the duration inlet of the makenote object, and the start time triggers makenote and is passed through to cue the beginning. The velocity output of makenote drives the signal as before.

Looping cues

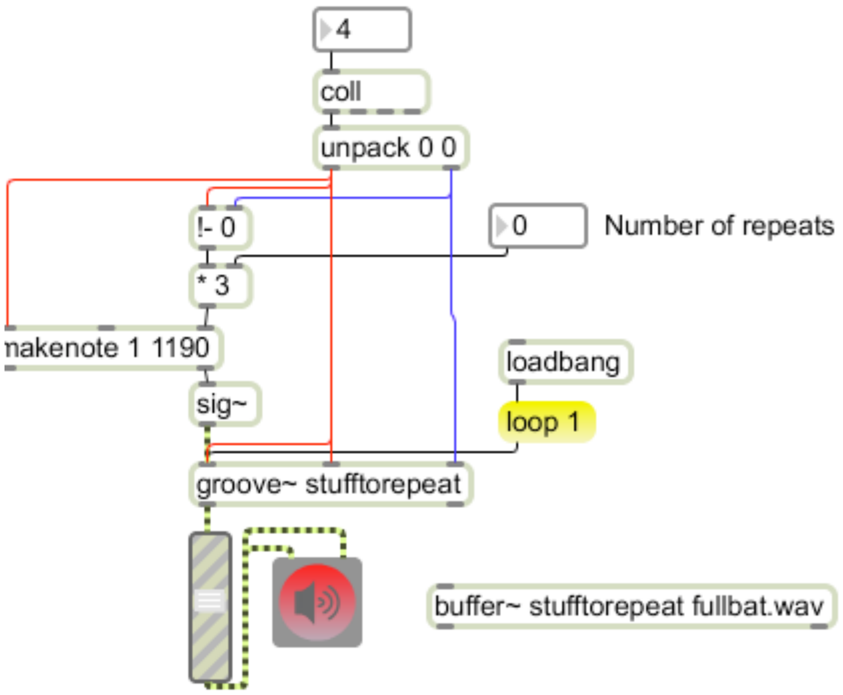


Figure 11.

One of the most popular features of groove~ is its agility in looping. Figure 11 shows how to modify the previous patch for a controlled number of repeats of each cue. The looping process is turned on with the loop 1 message. The second and third inlets of groove~ set loop start and loop end. These values are available in the controlling coll. To set the number of repeats, multiply the number by the duration that is calculated from the start and end times.

[6] The exclamation point reverses the operators in math objects. Thus !/ is divide into instead of divide by, and !- is subtract from instead of subtract.

MP3s

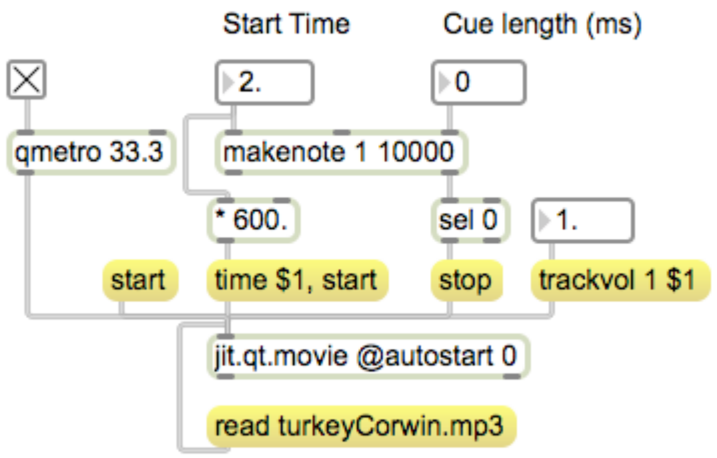


Figure 12.

The sfplay object~ will only open uncompressed audio format files—that is wav and aiff. If you prefer to play an mp3 file, jit.qt.movie is your best option[7]. Since there is no need for matrix output, we can skip the output dimensions but we do need to set the @autostart attribute to 0 so the file will not begin playing as soon as it is loaded. The mechanism for play and cue is not very different from sfplay~. Time is in QuickTime units at 600 per second.

Jit.qt.movie will play loops—in fact loop mode defaults to on. Figure 13 shows the basics. Turning loop on will cue the player to the beginning of the loop. So will adjusting the loop points.

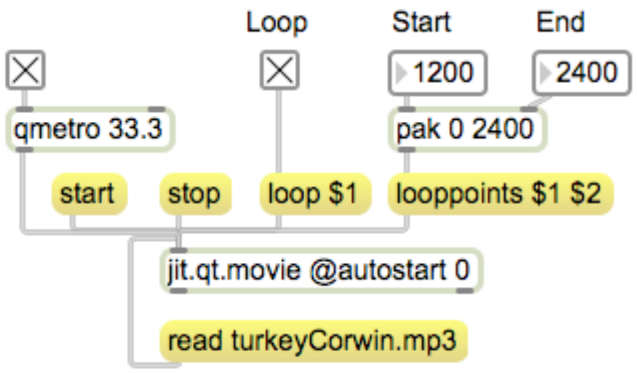


Figure 13.

The audio from jit.qt.movie goes straight to the computer out without involving the Max MSP system. This is not necessarily a bad thing, as there is significant cost to turning MSP on, and that computation may be better used on jitter operations. You still have control of volume via the trackvol 1 message, and that may be all you need. If you do want to pull the movie audio back into Max for further processing, a spigot~ object is required.

[7] Jit.qt.movie is discussed in detail in the Dealing with movies tutorial.

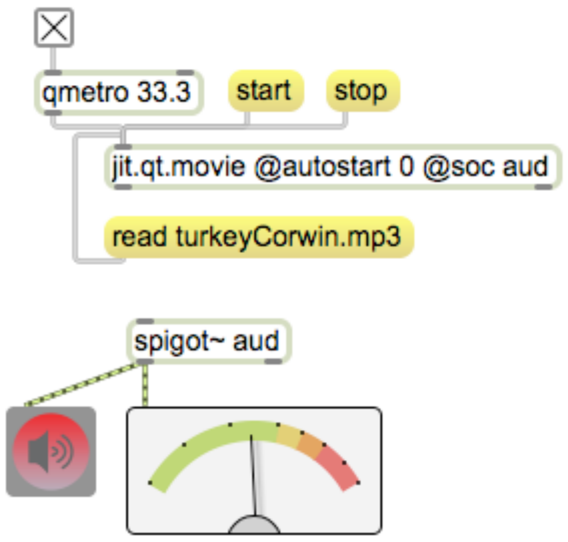


Figure 14.

The connection between jit.qt.movie and spigot~ is made by setting the movie attribute @soc and the spigot~ object to the same name. This connection must be unique—only one jit.qt.movie and one spigot~ may use a name. Once the connection is made, jit.qt.movie controls like trackvol and trackpan are no longer effective. MSP has all responsibility for audio.

Processing Audio

Signal processing is a huge subject, and creative techniques and new applications are being discovered faster than one person can write them down. However, most projects rely on a few old standbys, which I will cover here.

Switching Signals

The most common thing to do with audio is shut it off, or possibly turn it back on. This is not as simple as it seems, because any time an audio signal is interrupted, there will probably be a pop. Restoring the audio will likely produce a worse pop. This is caused by the abrupt transition from the sample value somewhere in the middle of a wave to 0, or from 0 to the middle of a wave.

Figure 15 shows how to cleanly turn a signal on and off. The object that controls signal level in MSP is the *~ (multiply) object. Since audio signals are nothing more than a series of numbers between -1 and 1, multiplying all of the values by a number less than 1.0 will reduce the overall volume. Multiplying by 1.0 will have no effect and multiplying by 0.0 will shut the signal off.

Changing the value in multiply from 1.0 to 0.0 will pop as reliably as any other method. However, if we use a series of numbers that change quickly but smoothly from 1.0 to 0.0, the sound will fade away cleanly. The line~ object will produce that series of numbers.

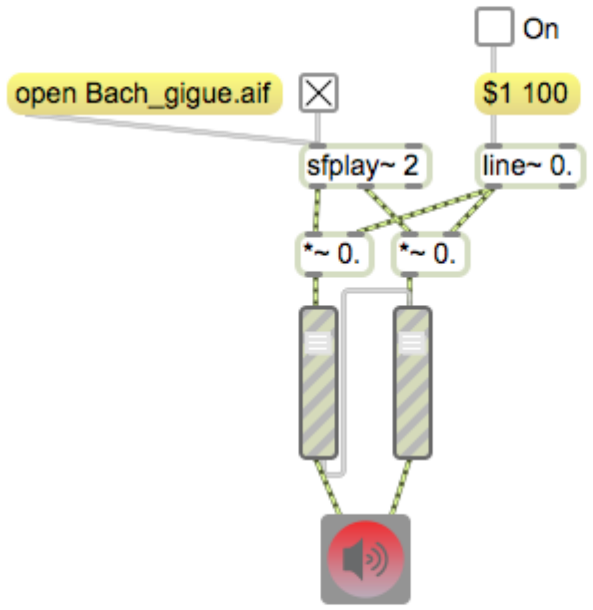


Figure 15.

The line~ object in figure 15 controls two *~ objects to mute the signal. Line~ itself is controlled by a list of two values—a target value and time to target. Given the message [0.0 100], line~ will take 100 milliseconds to transition from the current level to 0.0. The message is constructed as [$1 100]. The token $1 will be replaced by the output of the toggle to switch on or off. Notice that it takes two *~ objects to manage a stereo signal.

Crossfade

The next step after switching a signal is crossfading between different signals. This is nearly the same mechanism. Figure 16 shows how the output of line~ can be inverted by subtracting it from 1 with the !-~ object. This inverted ramp will control the second source of signal. Setting the toggle fades from one source to another over a half minute.

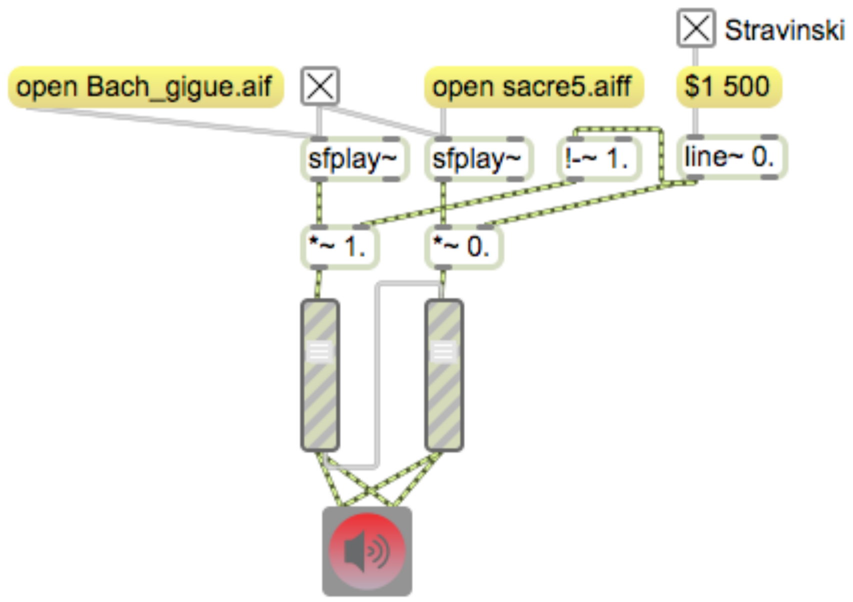


Figure 16.

I have made these mono signals for clarity. The stereo version is in figure 17.

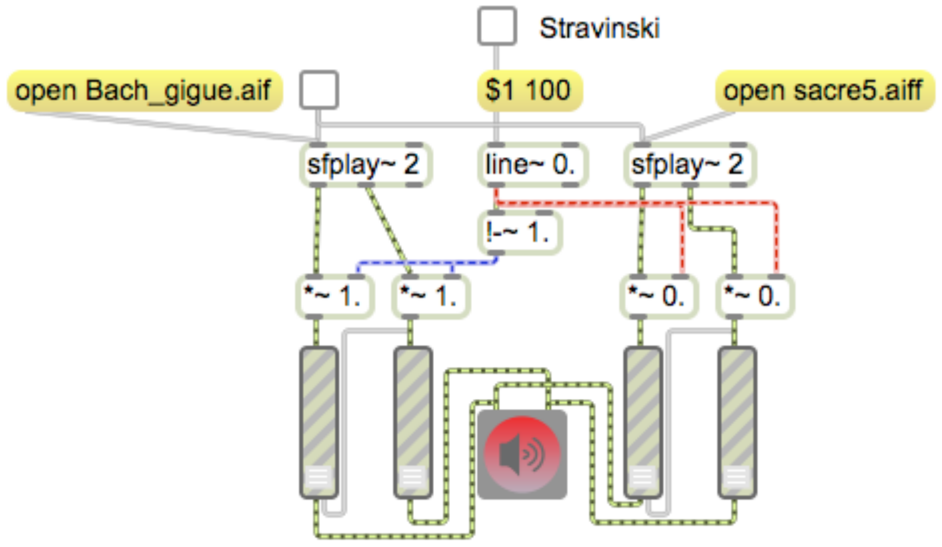


Figure 17.

Pan

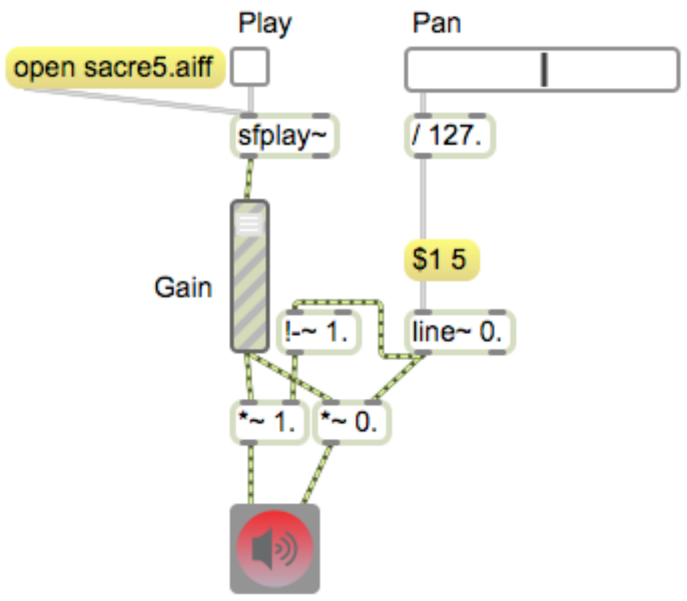


Figure 18.

Stereo panning is a simple variation on crossfade. Instead of choosing one of two sources, panning is choosing one of two destinations, with a user control to set the balance. Note that the pan operation is always a mono source to a stereo (or surround) destination. Moving a stereo source between channels is balance, and can be done in figure 2 by independent gain sliders.

Pan control here is provided by a data slider, which has an output of 0 to 127. This could be driven directly by a MIDI control device. The value must be scaled down to 0-1.0, which is accomplished by the division by 127.0. It is vitally important to remember the dot in the divide object. (The user control for pan is often a dial, but the interaction of a round icon and vertical motion of a mouse is so awkward that I avoid dials entirely.)

The line~ object is required to prevent pops that would occur as the pan slider is moved. The slider only has 127 steps, and can produce large jumps when it is moved quickly. The 5 millisecond time constant fills in between the steps.

Quad Pan

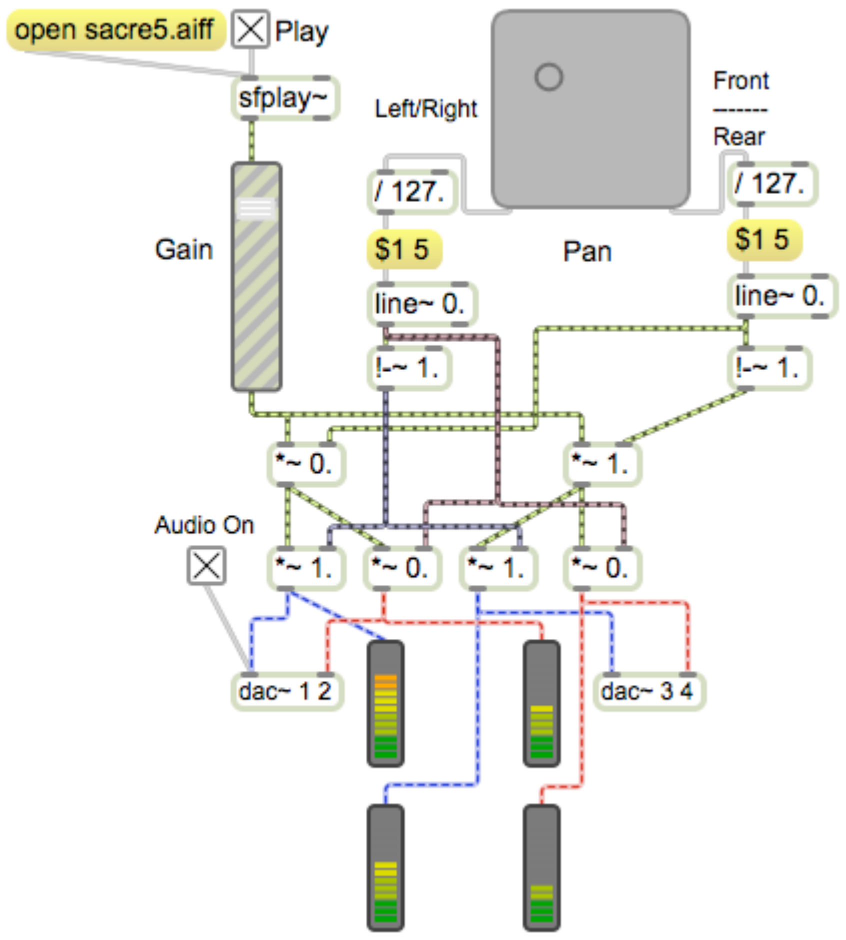


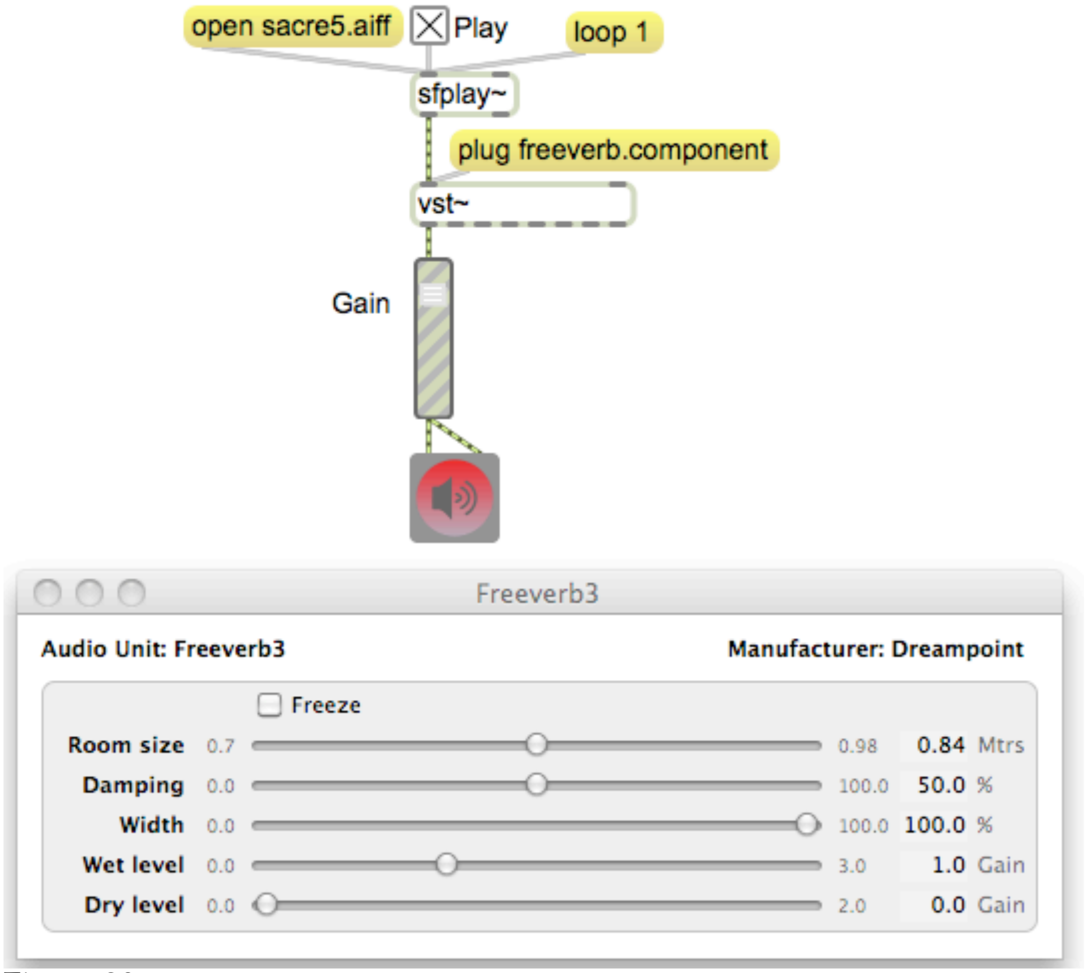
Figure 19.

If you need to distribute audio around a room, the best approach is old fashioned quadrophonic panning. This setup assumes four speakers, each in a corner of the room. We have already seen these mechanisms. Quad panning requires three pan modules, one to control front to back balance, and a left-right pair for both front and rear. The control comes from a pictslider, which provides a value for X and Y positions of the circle. The patch requires a normal and inverted version of each.

Of course you will need a 4 channel audio interface to make this patch work. The dac~ object is needed to reach outputs beyond 1 and 2. It takes arguments for the output channels and provides an inlet for each. Audio is enabled by sending a 1 to dac~ and disabled with a 0.

The colorful objects at the bottom of this patch are meter~ objects. They show the signal level—here the levels indicate the state of the pan mechanism. Each band is a 3 dB step except the second one, which is 2 dB. The top band is bright red and lights up only if the signal exceeds 1.0.

Using Plug-in Processors



Audio Unit: Freeverb3 Manufacturer: Dreampoint

Freeze

Room size	0.7	0.98	0.84	Mtrs
Damping	0.0	100.0	50.0	%
Width	0.0	100.0	100.0	%
Wet level	0.0	3.0	1.0	Gain
Dry level	0.0	2.0	0.0	Gain

Figure 20.

Design of audio processing patches is a whole course in itself. There are a few simple things to do, but the quickest way to get sophisticated processing is with the vst~ object. This lets you install plug-in processors in the MSP audio chain. Despite the name, in Max 6 you can use either VST or Audio Unit style plugs.

You load a plug-in with the plug message. This can take a file name, but only if you add the system plug-in folder to the Max search path. On Macs this is located at "Macintosh HD/Library/Audio/Plug-ins".

Once the plug-in is loaded, a double click on the vst~ object will open the control window. So will an *open* message.

A plug-in's parameters can be controlled without opening the window, but it will take a bit of detective work. Sending the message *params* to the object will cause a list of parameter names to be sent from the third outlet. The easiest way to read these is print them in the Max window with a *print* object. The illustrated freeverb plug-in has parameters Freeze, Room Size, Damping, Width, Wet level, and Dry Level. With some plugs the parameters can be accessed by name, but it is more common to get to them by number. To set parameter, send a list with the parameter number and the new value. The range of allowed values can usually be seen in the fine print of the control window. Figure 21 shows control of freeverb.

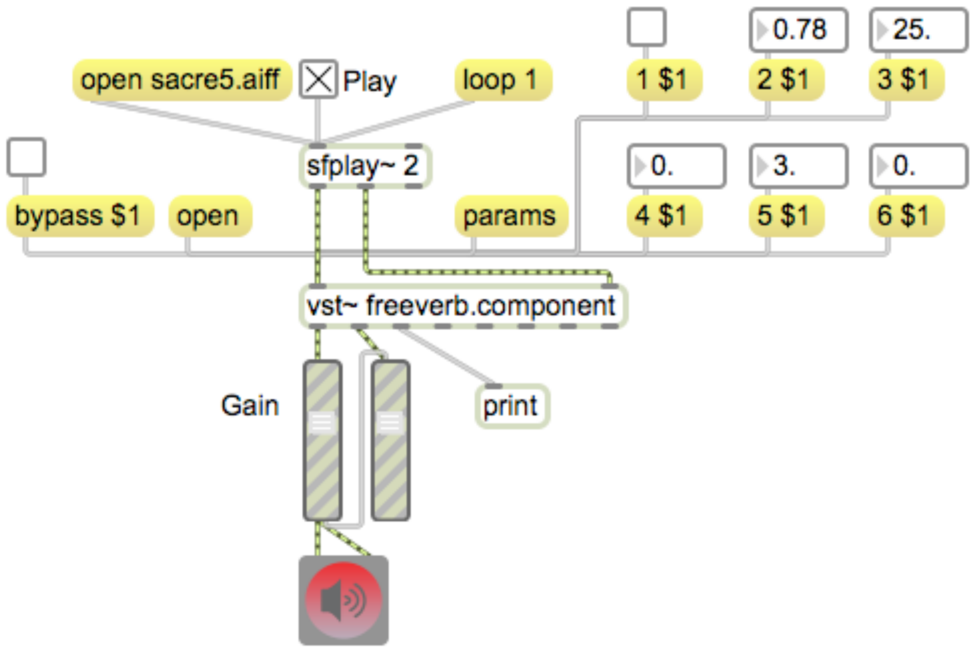


Figure 21.

Audio Input

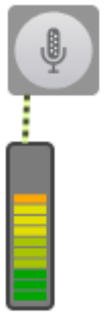


Figure 22.
Audio input gets to Max via the ezadc~ or adc~ objects. Like dac~, adc~ can access any audio input on a device. Ezadc~ is a stereo connection, which is adequate for most uses. A click on the ezadc~ microphone icon will turn audio on.

Simple Delay

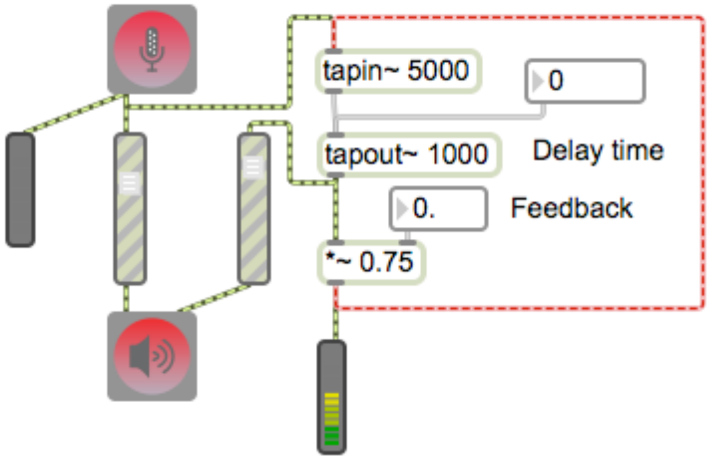


Figure 23.

Figure 23 illustrates a simple live process that is popular for installations. The signal is both sent (via gainslider~) directly to the output and to a tapin~ object which records the signal in memory. The argument in tapin~ sets the maximum record time. The associated tapout~ object reads from this memory. Note the patch cord that connects the two. The argument in tapout~ sets the delay time in milliseconds from write to read. Thus the mechanism as shown delays audio by one second. Simple delays have some interest, but most setups include feedback, where the delayed signal is patched back to the tapin~ for another delay. This will produce dwindling echoes. It is vitally important that the amount of feedback is less than 1.

There is a record~ object that works with the buffer~ object introduced in figure 9 that can produce complex delay effects using groove~. This is covered in detail in a tutorial on looping, Loopsync.pdf.

Audio Trigger

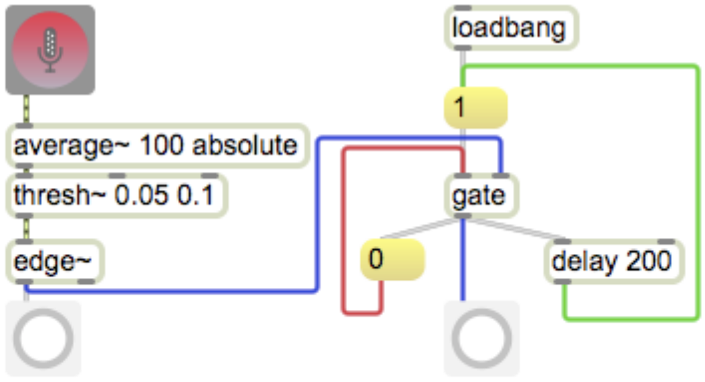


Figure 24.

One of the simplest uses of an audio input is to trigger some reaction from the patch. Figure 24 will react to loud sounds. Since audio is a series of widely varying values that are more or less evenly balanced positive and negative, it's not so easy to get a meaningful measurement. The average~ object measures the average value over the number of samples specified. There are three possible methods for computing that average. Bipolar is a straight average, and will be 0 for most audio signals. Absolute works with absolute values. RMS uses a method that approximates the experience of loudness. Absolute is the best choice here since it gives the widest ranging response.

The thresh~ object will produce a signal of 0.0 when the input is below the threshold set by the second argument value. If that level is exceeded, the output will become 1.0. When the input falls below the first argument, the output will become 0.0. again. The edge~ object will bang when the thresh~ output goes to 1. The right outlet of edge~ will bang when the thres~ falls back to 0.

The left section of the patch will be sufficient in many applications, but it is prone to double triggering. This can be minimized by extending the average~ time and adjusting the values in thresh~, but the nature of audio guarantees that some sounds will produce a flurry of bangs. (This can be tested by attaching a counter object to the button.) The right side of the patch is a mechanism for filtering extra triggers out.

I call this mechanism a mousetrap. The gate is open to begin with, so the first bang will pass through with no problem. This bang has two extra consequences: the message 0 is sent to the gate control, shutting the gate and preventing any more bangs from getting through. The first bang is also delayed, and eventually causes a 1 message to open the gate again.

Level Control

One perennial problem in installations that use sound is level control. Galleries are generally quiet spaces, and there are long periods when only the staff is around. They tend to adjust volume so the sound is not too intrusive when the room is quiet. When the crowd arrives, their chatter drowns out the installation.

The patch in figure 25 monitors the sound level in the room[8] and turns the volume up a bit when the ambience is noisy. (Of course you tell the curator that it turns the volume down when the space is quiet.)

[8] including the installation, so make sure the microphone is not too close to the speakers

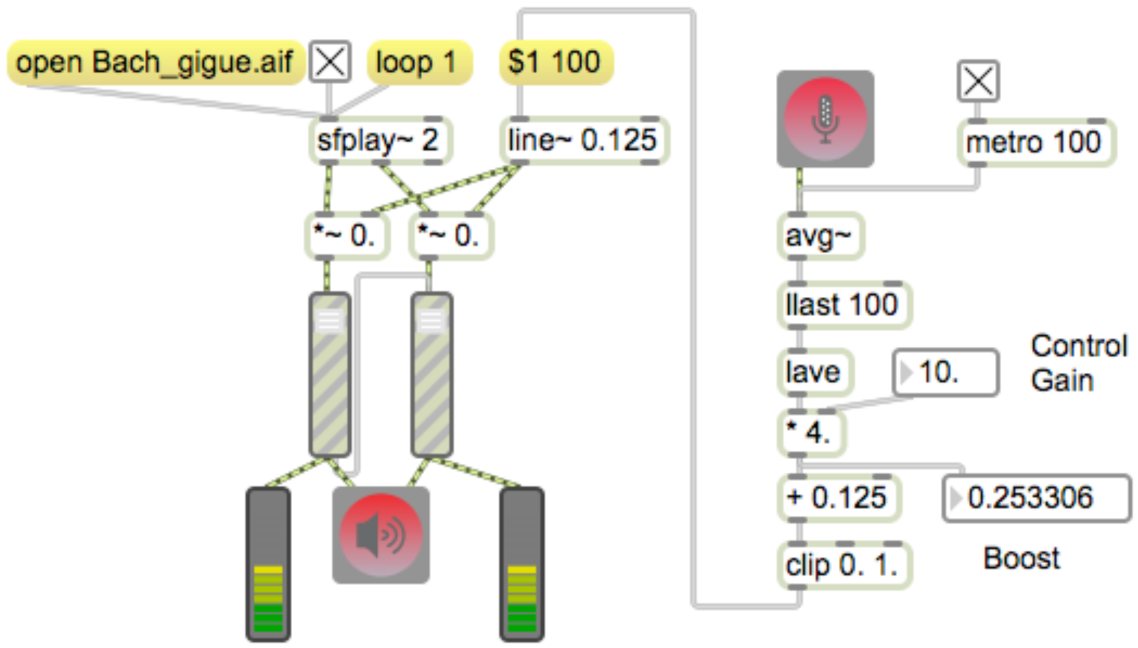


Figure 25.

The playback mechanism on the left is the same as the fade patch in figure 15, including a line~ object to smooth out the changes in level. The sound level in the room is measured by the right side of the patch. Audio from ezadc~ is applied to an avg~ object. This object also requires a steady supply of bangs. When banged, avg~ reports the average absolute signal value since the last bang. Here measurements are taken 10 times a second. The last 100 measurements are gathered into a list by llast (an lobject) and the overall average of the list is calculated by lave. This will prevent sudden jumps in volume—it is taking a period of 10 seconds into account.

The typical measurement will depend on the placement and sensitivity of the microphone. Chances are, the signal will be fairly low, so there is a multiply object to magnify the control effect. This will have to be trimmed for each installation. The enhanced level value is added to a nominal 0.125 that sets the floor volume. This works out to 18 dB of control. The final value is limited to be no more than 1. This is applied to the *~ objects by way of line~.

Ducking

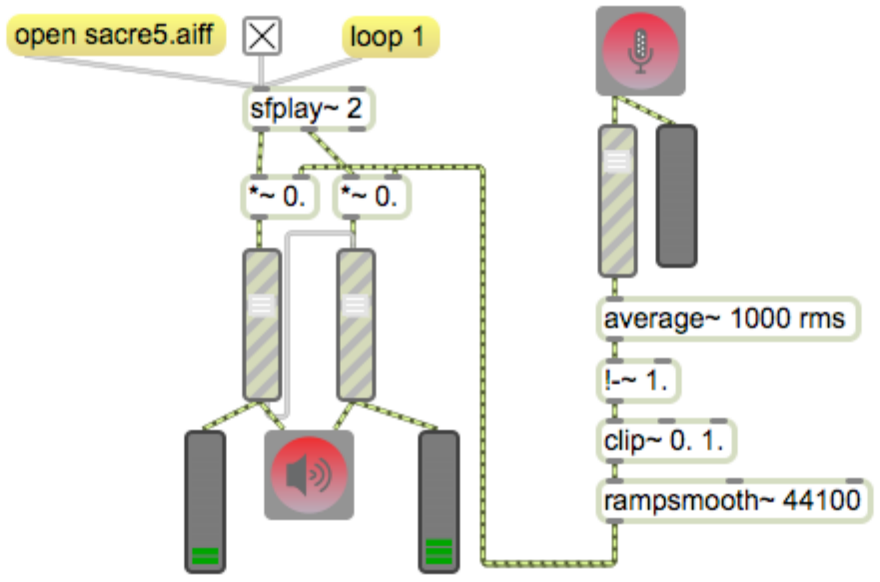


Figure 26.
The patch in figure 26 has the opposite effect. It turns the playback volume down when signal comes in, perhaps when someone wants to make an announcement. Since this should respond quickly, all of the process is done in the audio chain. The average object measures the audio over the given time period with the method (rms) specified. This is inverted by the !-~ object and the result is feed to the *~ objects. Rampsmooth~ works somewhat like line~. The argument is in samples and sets the time to complete a change.