Gama Network Presents:

Gamasutra.com

Audio Prototyping with Pure Data

By ***Leonard J. Paul***

Gamasutra

May 28, 2003

URL: http://www.gamasutra.com/resource_guide/20030528/paul_01.shtml

Creating great audio for games requires successfully combining content and code. Unfortunately, the sound designer/composer (hereafter referred to as "the composer") often has little control over how the code works and thus how the content will actually sound in the final game. This, in turn, reduces the potential of game audio. To remedy this situation, we need a better flow of ideas between the disciplines of the audio programmer and the composer. Even with great communication, however, the coder's time is often at a premium, which causes the audio to suffer as a result.

To solve this problem, we need ways for composers to be able to experiment with interactive audio without needing to wait for the coder's time. This is especially true for smaller game companies which do not have a dedicated audio coder or a refined audio toolset. By putting more control in the composer's hands, the composer can test out ideas independent from the coder. Likewise, the more the composer learns about the control structures and methods to control content, the better informed and effective he will be at creating content. Advanced users of Pure Data could even construct a platform independent prototype sound driver linked to the game via MIDI or TCP/IP.

In this article, we'll examine an audio prototyping tool which can be used to free the composer to experiment with interactive audio ideas on his own.

Benefits and Drawbacks

Before we begin, let us consider two hurdles to having the composer do the prototyping. First, there is a steep learning curve for the composer to learn what is, in essence, a new discipline. Many composers won't find this learning process rewarding enough to continue past the difficult initial learning phase. Second, there is a problem that's widespread in the game development business: audio often plays second fiddle in games to areas including graphics, artificial intelligence and physics. As such, the more complicated the control structures for audio become, the more team members from other areas of the game become concerned about losing precious CPU cycles to audio. Often the most interesting audio possibilities require too much CPU horsepower, so a balance must be struck. If the audio team can demonstrate the prototype to the rest of the team early in the development cycle, they can lobby for more resources before decisions are set.

If the audio team can demonstrate the prototype to the rest of the team early in the development cycle, they can lobby for more resources before decisions are set. In the end, if the composer is unable to learn the techniques for prototyping interactive audio, this task is likely better left to the coder. But even in that case, the goal is to have the composer drive the process.

Since the goal is for the composer to do the bulk of the prototyping, it is important to find musician-friendly prototyping tools. It is possible to use applications such as Reaktor by Native Instruments or AudioMulch by Ross Bencina, but these tools simply aren't flexible enough to implement more complex game audio behaviors.

Audio Examples:

SampleBank.ogg - Plays simulated "footsteps" as generated in the SampleBankTester.pd patch [Figure 3]

Crowd.ogg - Crowd sound as the intensity rises and falls in the CrowdEngine.pd patch [Figure 10]

Speech.ogg - All five sentences are output from the StitchedSpeech.pd patch using synthesized speech as its source samples.

Music.ogg - AdaptiveMusic.pd patch playing example music. The input intensities and volumes can be seen on Chart 1.

Note: To play .ogg files you can use Winamp2.80 or better for PC or see http://www.vorbis.org for a list of more players and related information.

What's Pure Data?

A good balance between flexibility and complexity for audio prototyping lies in graphical, object-oriented audio tools like Max/MSP (available only for the Macintosh) or Pure Data (available for Windows, Linux and Mac OS X). I will concentrate on Pure Data rather than Max/MSP in this article, although they are functionally very similar.

The major drawback to to Pure Data is that it is definitely less user-friendly (due in part to its cross-platform nature) than the commercially refined Max/MSP. However, with some extra effort by the user, I believe Pure Data is a better choice for game audio development. Did I happen to mention it was free?

Pure Data (and its ilk) are interpreted languages which avoid the time spent recompiling when changes are made. They also allow the user to modify parameters and behaviors while the patch is running. This real-time feedback loop between modification and audition makes prototyping very rapid. However, nothing is perfect and there are a number of significant drawbacks which can cause problems during prototyping. One of the main problems with a modular programming environment such as Pure Data is that the order of operations, which one often takes for granted in a more procedural language such as C++, can become quite difficult with larger patches. I spent an inordinate amount of time trying to get the parameters to initialize properly with the "Adaptive Music" patch. Larger program patches also have a tendency to grow incomprehensible quite quickly (which I have tried to avoid in my example prototypes). However, through the diligent use of hiding complexity, splitting patches into logical subpatches and by making a clear user interface, this problem can be overcome.

To download Pure Data, go to http://pure-data.sourceforge.net/. When you begin learning Pure Data, you should have a look at the tool's official web site, http://pure-data.org. The program provides documentation and tutorials in the "pd/doc" directory. Going through the tutorials is useful, but they can sometimes be confusing since they are often geared towards learning electroacoustic principles and the advanced uses of Pure Data instead of being geared towards the novice user. The official website provides links to current developments in Pure Data and to related externals (extensions which others have coded). Another good source for current information is the Pure Data newsgroup at http://www.iem.at/mailinglists/pd-list/.

The goal of this article is not to teach Pure Data, but to show how a knowledgeable user can create prototypes for video games. To become knowledgeable about Pure Data, it's best to read the official documentation, go through some of the tutorials, peruse the newsgroup and then dive in and start modifying the tutorial patches to find out how things really work. Once one is comfortable with the general workings of Pure Data, the example patches here can be explored and modified.

Table 1. Pure Data Tips and Tricks For Newbies

- Make sure to turn on the DSP, otherwise you won't hear anything.
- Use the SHIFT key when dragging on a number box to change it in fractional increments
- When working on a patch, it's often best to stay in edit mode and use the CTRL key to modify any parameters
- I still find symbols and lists confusing, so it's best to try to get these straight early on
- Use the right-mouse button on the canvas and select help to see a list of core Pure Data commands
- Use color-coding and clear arrangements to guide the user to elements which should be modified and hide parameters which shouldn't be modified in sub-patches
- Scan the web for new developments since extensions are being continuously added and modified
- Use comments and labels to keep things clear for yourself and others
- Save frequently and keep your archives organized for easy recovery
- Sequence events by using compound messages
- For easy reuse, make sure you label the inlets and outlets of abstractions and subpatches
- I commonly use an underscore at the start of my abstractions to flag them as being separate regular Pure Data objects

The Four Areas of Game Audio

Like film and television, game audio contains music, speech and sound effects. But games also require interactivity and real-time effects applied to the audio. When this occurs, game audio moves further away from its roots in linear media.

Highly adaptive audio is difficult to create, due to the increasing number of variables created between the content and the code. Adaptive music requires the composer to create content which seamlessly transitions between a multitude of game states. Similarly, complex multi-sample speech stitching can spiral into an almost endless number of possible sentences, and getting such a system to sound good can be overwhelming.

Besides music, speech and sound effects, games also use a fourth category of audio. This category of audio attempts to simulate highly complex sounds, such as the roar of an entire crowd or the sound of a racing car engine. These sounds are made up of many dynamic parameters which modulate the audio content in real-time. With physical modeling and other synthesis methods it may even be possible to synthesize the sound entirely. For the purposes of this article, I will use the term "engine" to refer to this category of audio. The difficulty for the composer, is that the content must react well to many different kinds and combinations of modulation to maintain the illusion of the object it is attempting to simulate.

With game audio platforms supporting audio resolutions similar to film, the question is no longer if we can produce audio which sounds good, but whether we can produce audio which reacts well and sounds good in all of the different states that the game player might trigger. In the following Pure Data section of the article, we'll examine four different prototype patches which address the issues of each of these four categories within game audio.

Using Patches In Pure Data

In the following section I introduce four patches which I put together to aid in audio prototyping. These patches that correspond to the four game audio categories outlined above. The patches are meant to be working prototypes which can be further modified to suit the individual needs of the composer, or just to start learning about Pure Data. The idea is to begin by playing around with the patches and make some sound, understand how they work and eventually start modifying them to suit your own individual needs. Alternatively, you can create new patches using these as a starting point.

Patches for Pure Data
[Click here to download...]

Sample Bank Patch

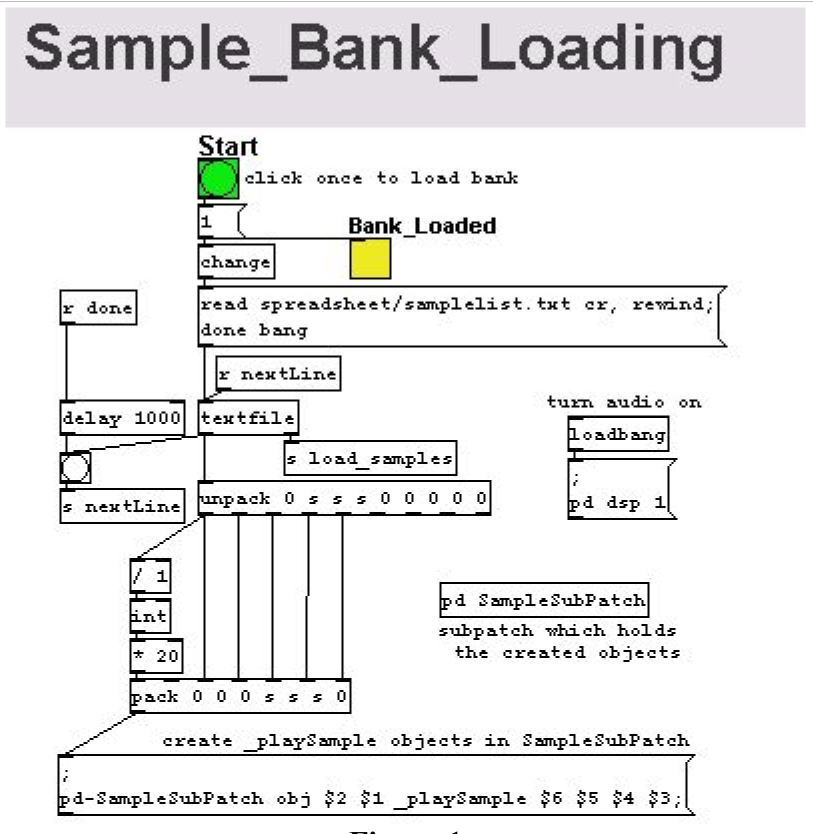
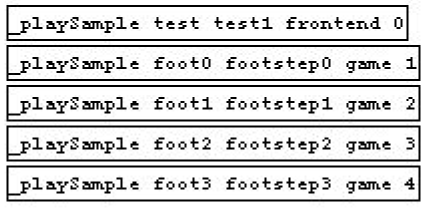


Figure 1

One of the basic concepts when working with game sound effects is the notion of a sample bank. Usually one wants certain sounds to always stay loaded (e.g., short menu sounds), and to have groups of sounds relate to specific locations, levels, or items (e.g., vehicles or weapons). The following Sample Bank patch ,which I have prototyped, reads all the sound effects contained in a spreadsheet and assigns them default values which can be applied to default volume, envelopes and any other desired parameters.



```
_playSample test test1 frontend 0
_playSample foot0 footstep0 game 1
_playSample foot1 footstep1 game 2
_playSample foot2 footstep2 game 3
_playSample foot3 footstep3 game 4
```

Figure 2

For the first patch, I'll describe the Pure Data code in more detail than the other three patches. Rather than construct a large number of array "slots" which the sounds load into (and thus restrict the maximum number of sound effects), this patch creates the sound effects objects (data and associated operations) at runtime using the "; pd-" message at the bottom of the parent window. This method is very similar to the way the sounds might actually be loaded and created in a compiled object-oriented language such as C++ for the game.

Once the sound effect objects are created, they can respond to messages. For instance, they can be told to play as a one-shot sound or as a continuous loop as well as can change their pitch, and so on. Through the use of messaging, multiple effects can be triggered and the resulting mix can be auditioned. This use of triggering multiple sounds with a single message can be thought of as sound macros or soundtags which would often otherwise be difficult to create using a sample editor or sampler. This patch allows the composer to generate and control composite sounds, such as player collisions, breaking glass and other more complex sound events by triggering multiple samples to produce a varied result each time.

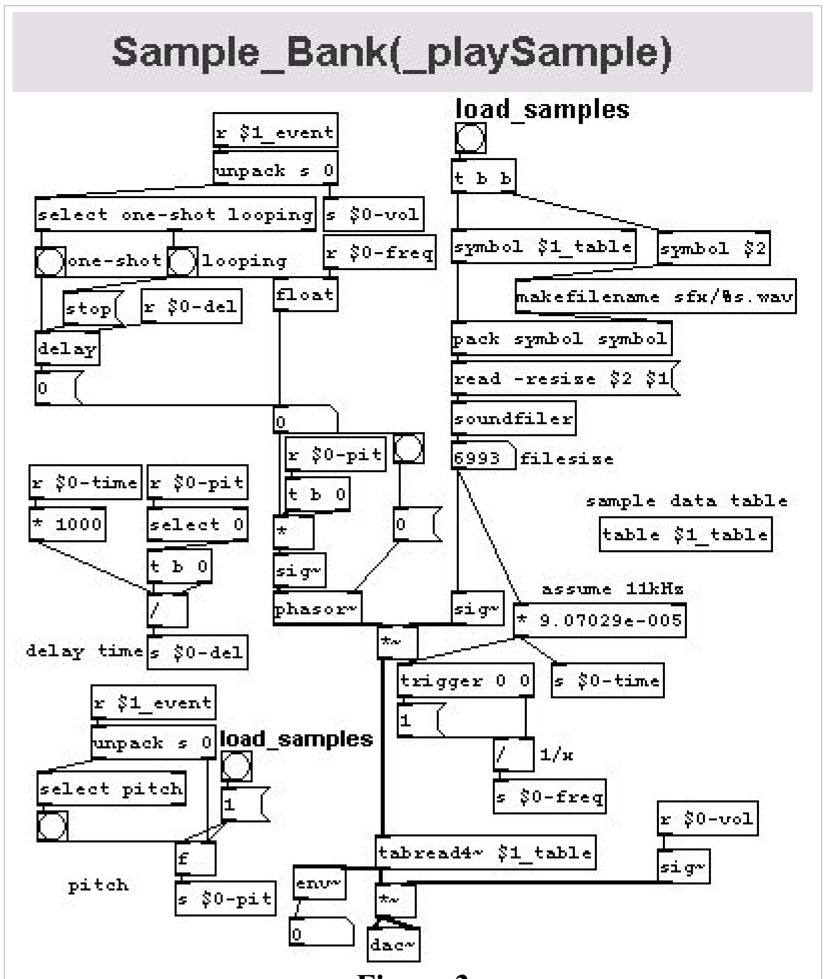


Figure 3

The "_playSample" abstraction begins by loading its respective sample into the sample data table. The size of the file is output from the "soundfiler" object which is converted to a frequency and sent to the "phasor~" object so that it will play back the sound in a loop at the correct rate. When an event is received the abstraction parses the event type and event parameter apart. If the event is "looping" it starts the "phasor~" object which generates a sawtooth wave of the correct frequency to modulate the sample playback position pointer in the "tabread4~" object to play the sample in a loop at the correct frequency scaled by the event parameter. If a "one-shot" event is received, then the playback rate is set to zero after the sample is played once using a "delay" object. If a "pitch" event is received, then the phasor's frequency is scaled by the parameter. Thus, sending "pitch 0.5" will cause the sample to play an octave lower and "pitch 2" will play the sample an octave higher.

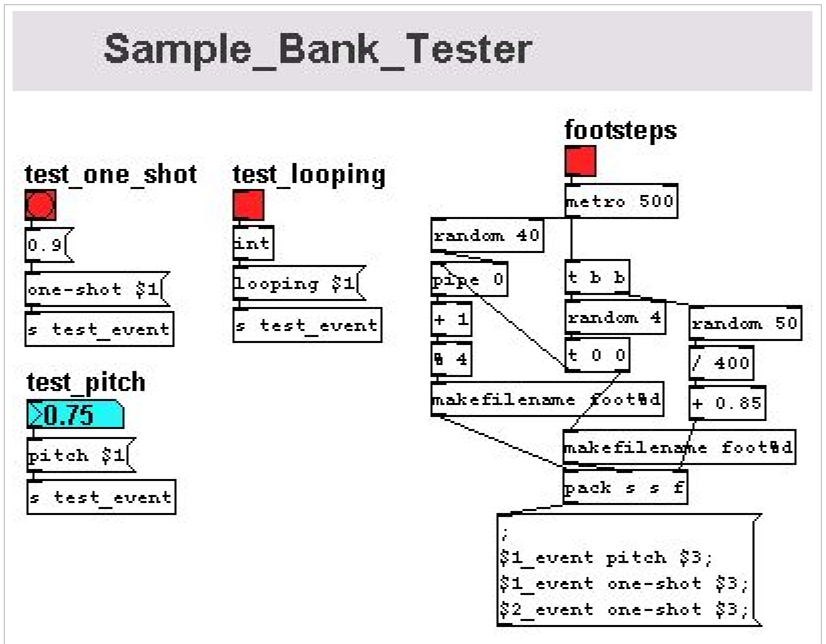
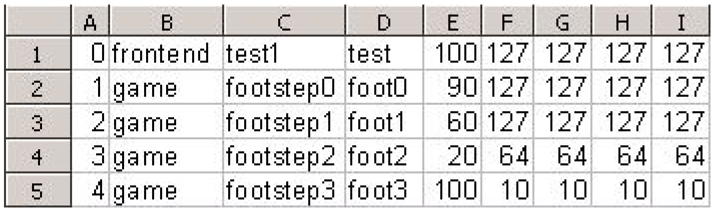


Figure 4

Since the data and its related operations are created as a single unit (or "object" in object-oriented terminology), multiple objects must be created to trigger multiple versions of the same sound. The drawback is that creating multiple versions of the sample generates extra overhead in the spreadsheet. The upside is that the system avoids issues of voices, or confusion over which instance of the sound to modify once the sound is playing. For example, if we trigger multiple tire squeal sounds using the same sample and want to control each sample's playback pitch, we know which sample we are controlling since each sample object has a different name. An improvement could be to define the maximum number of instances of a given object (instance capping) and have Pure Data create the appropriate objects and index them automatically. When implemented, it also wouldn't make much sense to have multiple versions of the same data, so this would need to be optimized as well.

In the "Sample Bank Tester" patch, we can test one-shot, looping and pitch events. The "footsteps" bang causes the simulation of footsteps to be triggered. This is created by randomly triggering footstep samples with a small variation in delay time before the next sample is triggered. This can be heard in the footsteps.ogg file.



	A	B	C	D	E	F	G	H	I
1	0	frontend	test1	test	100	127	127	127	127
2	1	game	footstep0	foot0	90	127	127	127	127
3	2	game	footstep1	foot1	60	127	127	127	127
4	3	game	footstep2	foot2	20	64	64	64	64
5	4	game	footstep3	foot3	100	10	10	10	10

Figure 5

This patch allows the composer to easily change the triggering and modulation of a spreadsheet full of sound effects. Currently, the patch does not use a lot of the supplied parameters, including the "game area" column, which could eventually be used to calculate the memory usage for each bank.

If the game was able to output events that Pure Data could react to, such as TCP/IP, MIDI or some other protocol, this patch could be used to interactively change the behavior of the audio interactively. Likely there would be latentcy issues and other technical concerns but it would allow for multi-platform games to all trigger the same audio to make sure that the sound has been correctly coded on each platform. In essence, it could function as a highly flexible platform independent sound driver.

The Speech Patch

Stitched (or chained speech) is an area of game audio which requires simple tools to playback a generated sentence. Often, however, a fairly complicated set of logic will determine what sentence to generate.

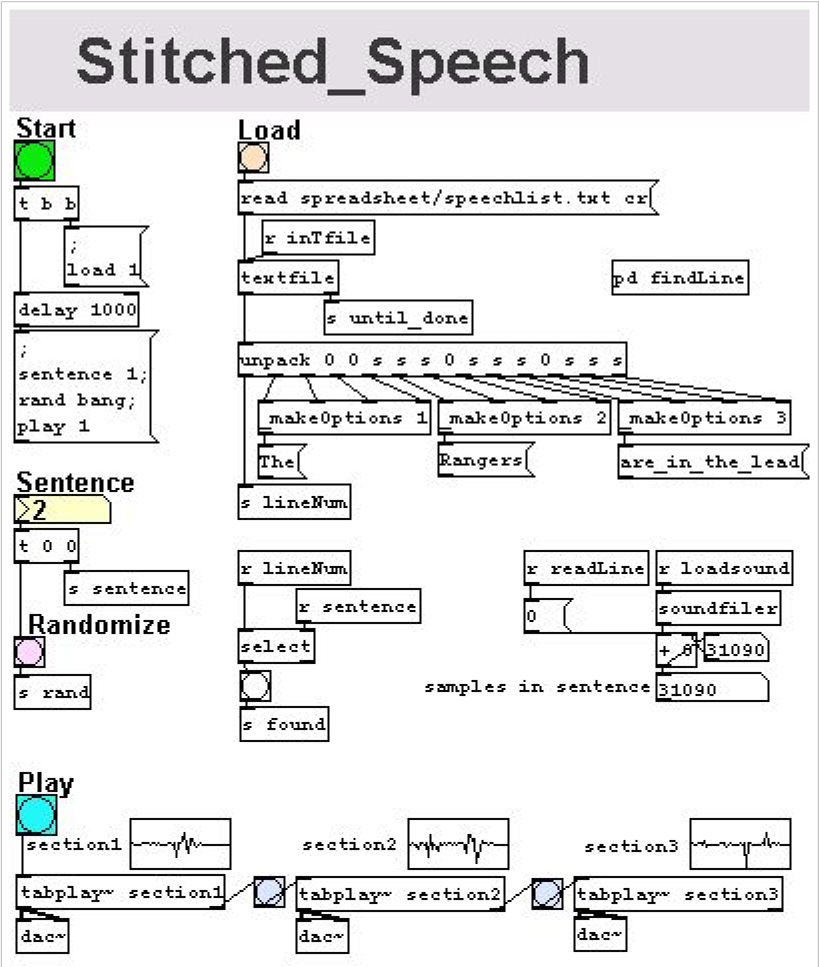


Figure 6

The Speech patch prototype supports a three-section, stitched sentence, using up to three options for each section. This functionality is obviously insufficient for most real-world applications, but it is kept simple here for the purposes of illustration. This patch could be expanded to allow a variable number of options and sections using similar run-time generation methods as the Sample Bank patch.

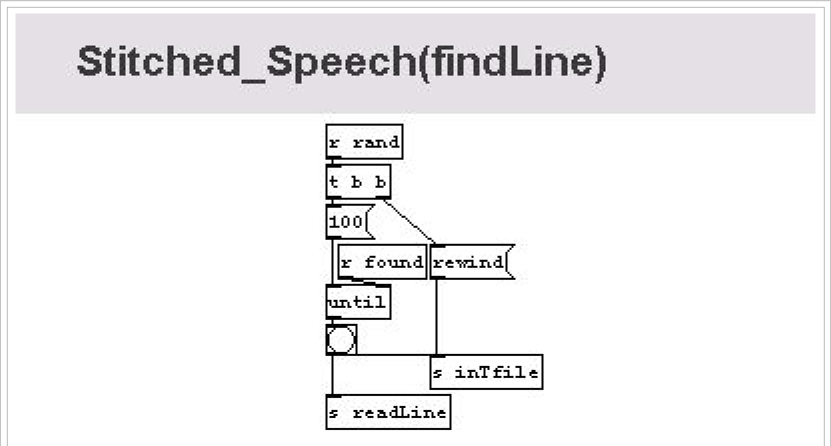


Figure 7

The Speech patch allows a composer to quickly check if a stitched sentence sounds "weird" (possibly an incorrect stitch gap length, uneven emphasis, and so on). It works by randomizing the possibilities for each section and playing back the generated sentence. The sentence can be easily tuned using an external wave editor to change the stitch gap length or to substitute different recorded takes in real time.

One nice feature of the Speech patch that it outputs the name of each phrase, so it is possible to see if file names have the right data and what the entire sentence should be. It also graphically displays the sample data of each part, so the data can be quickly inspected without requiring the file to be opened in an external wave editor.

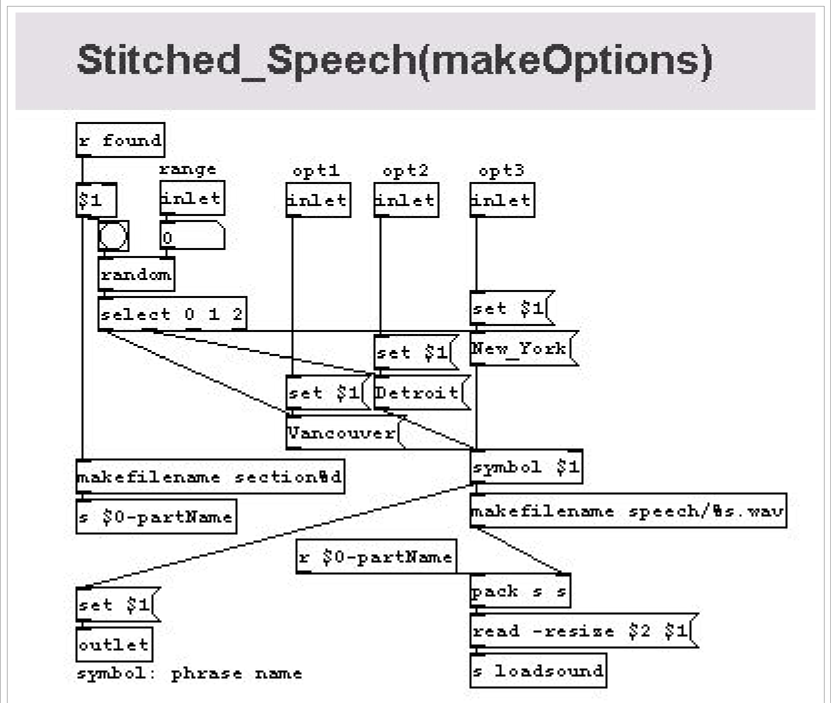
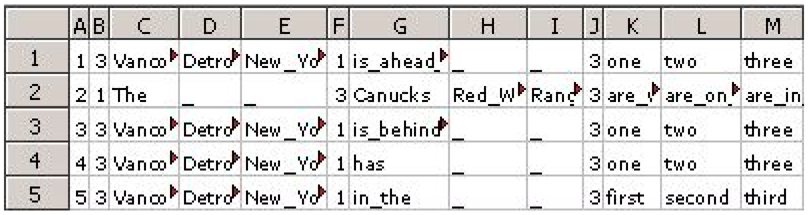


Figure 8

This patch works by filling the patch with the appropriate data. It does this by parsing the spreadsheet, which is in the following format:

<ID> <section 1 number of choices> <section 1 WAV file option 1> <section 1 WAV file option 2> <section 1 WAV file option 3> <section 2 number of choices> ... <section 3 WAV file option 3>



	A	B	C	D	E	F	G	H	I	J	K	L	M
1	1	3	Vanco	Detro	New_Yo	1	is_ahead_	_	_	3	one	two	three
2	2	1	The	_	_	3	Canucks	Red_W	Rang	3	are_	are_on_	are_in
3	3	3	Vanco	Detro	New_Yo	1	is_behind	_	_	3	one	two	three
4	4	3	Vanco	Detro	New_Yo	1	has	_	_	3	one	two	three
5	5	3	Vanco	Detro	New_Yo	1	in_the	_	_	3	first	second	third

Figure 9

In Figure 8, the OpenOffice spreadsheet is converted to the "speechlist.txt" text file as:

```
1 3 Vancouver Detroit New_York 1 is_ahead_by _ _ 3 one two three
2 1 The _ _ 3 Canucks Red_Wings Rangers 3 are_winning_the_game are_on_top
are_in_the_lead
3 3 Vancouver Detroit New_York 1 is_behind_by _ _ 3 one two three
4 3 Vancouver Detroit New_York 1 has _ _ 3 one two three
5 3 Vancouver Detroit New_York 1 in_the _ _ 3 first second third
```

The first sentence first chooses between the three choices of "Vancouver", "Detroit" and "New York". The underscore in "New_York" is necessary for Pure Data to treat it as a single symbol. The second section is fixed at a single choice of the phrase "are ahead by", with underscores being used to fill in the other two blank phrases. The final section has the three choices of "one", "two" or "three".

When a sentence is selected, each section randomly loads in one of its options into an array. When the sentence is played, each section plays in sequence with the bang follows the chain as each gets started, just like the bouncing ball of yesteryear.

Future expansions to this patch could include naming the phrases separate from their file names to adopt a more rigorous file naming convention, but still have human-readable output for the content of each phrase. The ability to test each option of each section in order could also easily be added to allow checking of all the data. Obviously, the number of sections and possible selections for each phrase should be expanded, possibly through the use of multiple spreadsheets to keep track of large repeated lists of options like player names and numbers.

Crowd Engine Patch

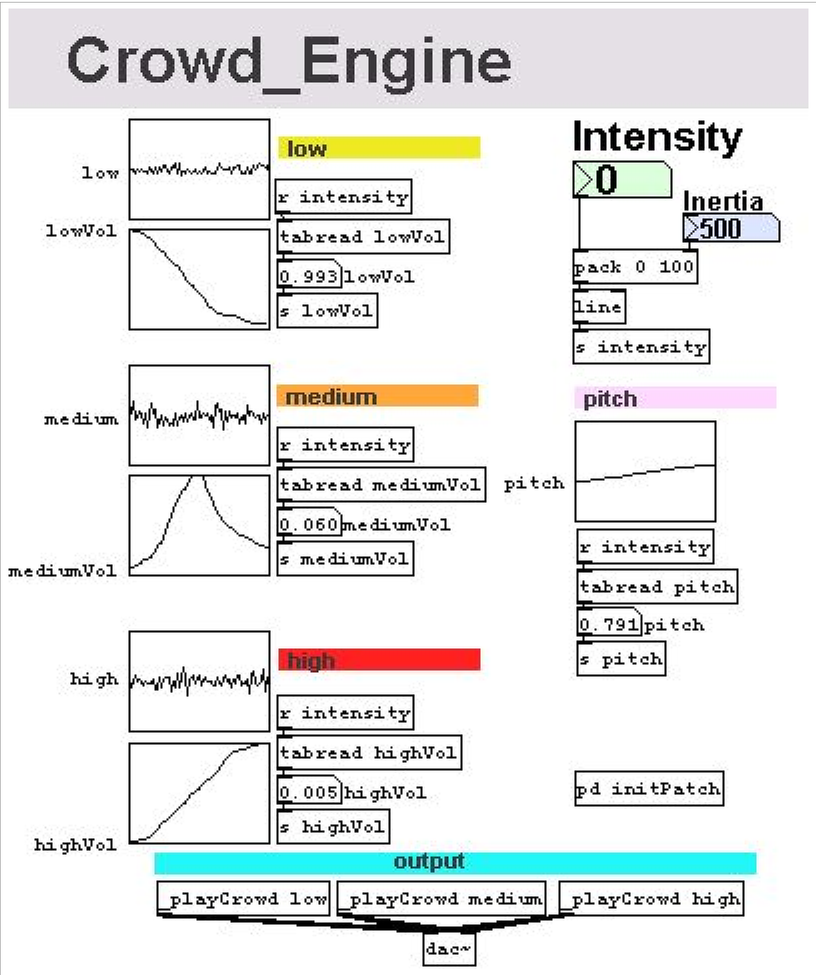


Figure 10

Complex sounds whose behaviors are highly dependent on the current game state (such as crowd noise or vehicle engines) are one of the most difficult areas of sound generation in game audio systems. With the crowd engine prototype patch, I've produced a simple three-sample crossfading crowd model which uses the intensity of the crowd as the modulating input. Synthesis techniques such as subtractive synthesis or physical modeling could also be used to achieve more realistic effects, but will not be covered here. The volume tables simply describe a crossfade such that it smoothly transitions between the low sample at low intensity and the high sample at high intensity. The pitch is also increased as the intensity increases to add more dynamics to the sound.

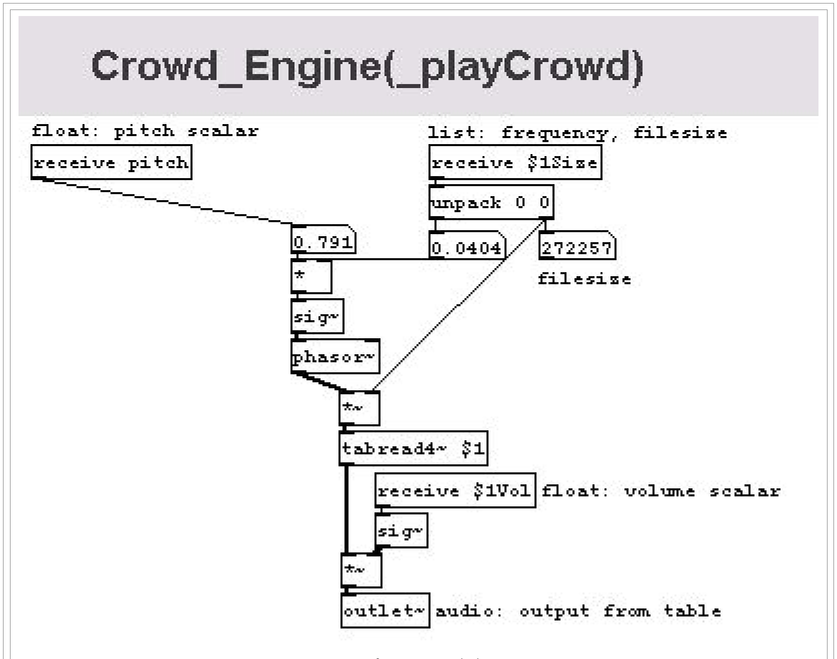


Figure 11

The samples are loaded from the message in the "initPatch" subpatch and all the file-size parameters are set when the patch is first opened. The sample data can be changed interactively by reloading the data by clicking on the "read" message. The crossfade tables, pitch curve tables and their ranges can also easily be changed by opening the table and editing the data or parameters by hand. This allows the composer to interactively tune the crowd samples and their behaviors in real time.

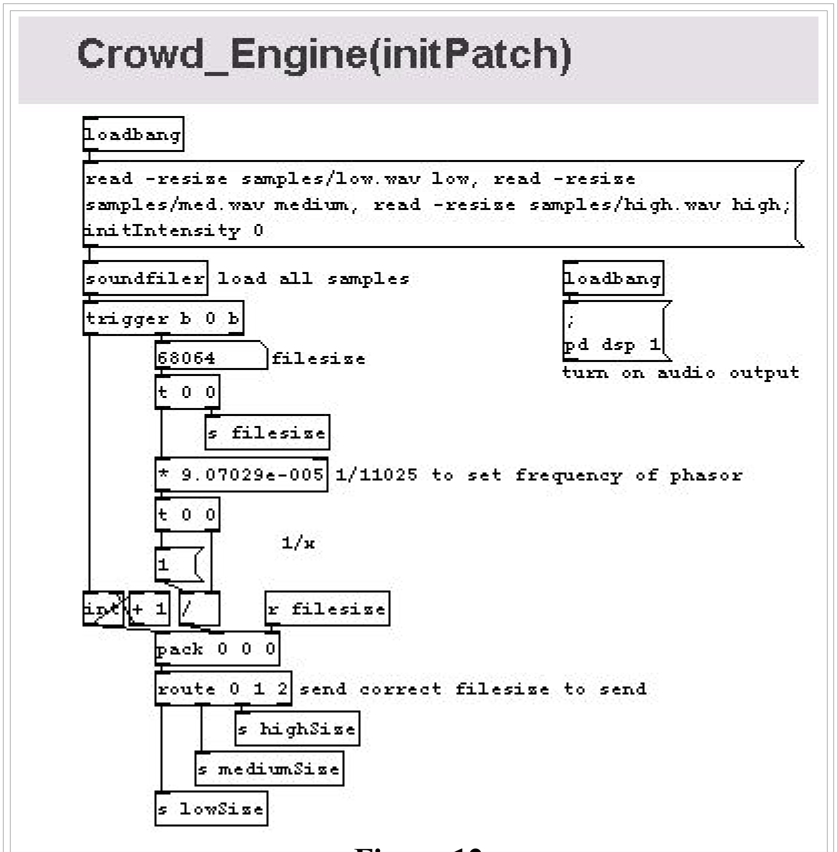


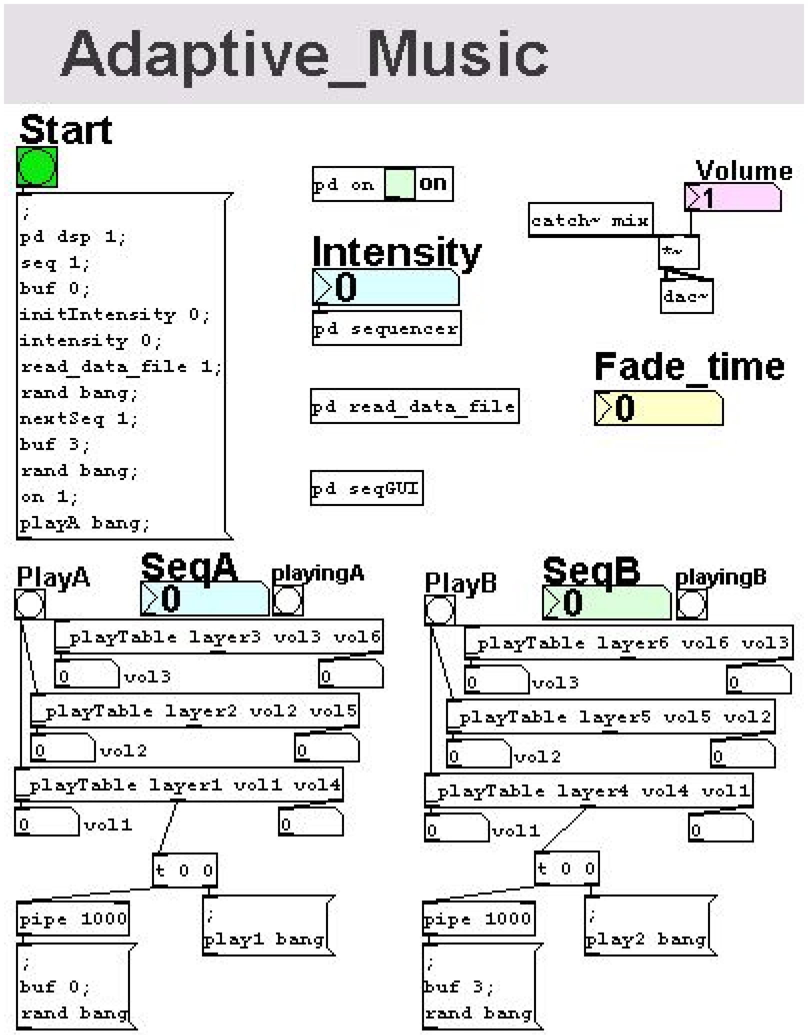
Figure 12

It would be valuable to add the notion of one-shot overlays for whistles, chants and shouts to make the crowd more believable. Another improvement would be to add LFO variation to the volume and pitch to make them less static. Granular effects could also be used to make the samples sound less static. The above model could also be used to generate vehicle engine noise (such as a car or boat engine) by swapping the samples with engine-noise samples and exchanging the notion of intensity for RPM.

A definite problem with this pitch scaling multi-sample crossfade model is the "chipmunk effect." The goal is to only really shift the fundamental pitch and its related overtones as though the crowd was raising the pitch of their voices as the intensity increases. However, when the samples are shifted up in pitch, all other unrelated frequency information such as formants and reverb gets shifted up as well which can make the crowd sound like a bunch of chipmunks. As the audio processors improve in game platforms, they may support more real-time frequency-domain processing techniques to overcome this and similar problems.

Crowd noise is a highly complex sound defined by many individuals in a complex reverberant space. Engine noise is produced by thousands of moving parts which change every fraction of a second. As game audio technology platforms increase in power, the more important prototyping will become to test and refine models which simulate complex acoustic phenomena such as these.

The Adaptive Music Patch



```
;
pd dsp 1;
seq 1;
buf 0;
initIntensity 0;
intensity 0;
read_data_file 1;
rand bang;
nextSeq 1;
buf 3;
rand bang;
on 1;
playA bang;
```

```
;
play1 bang
```

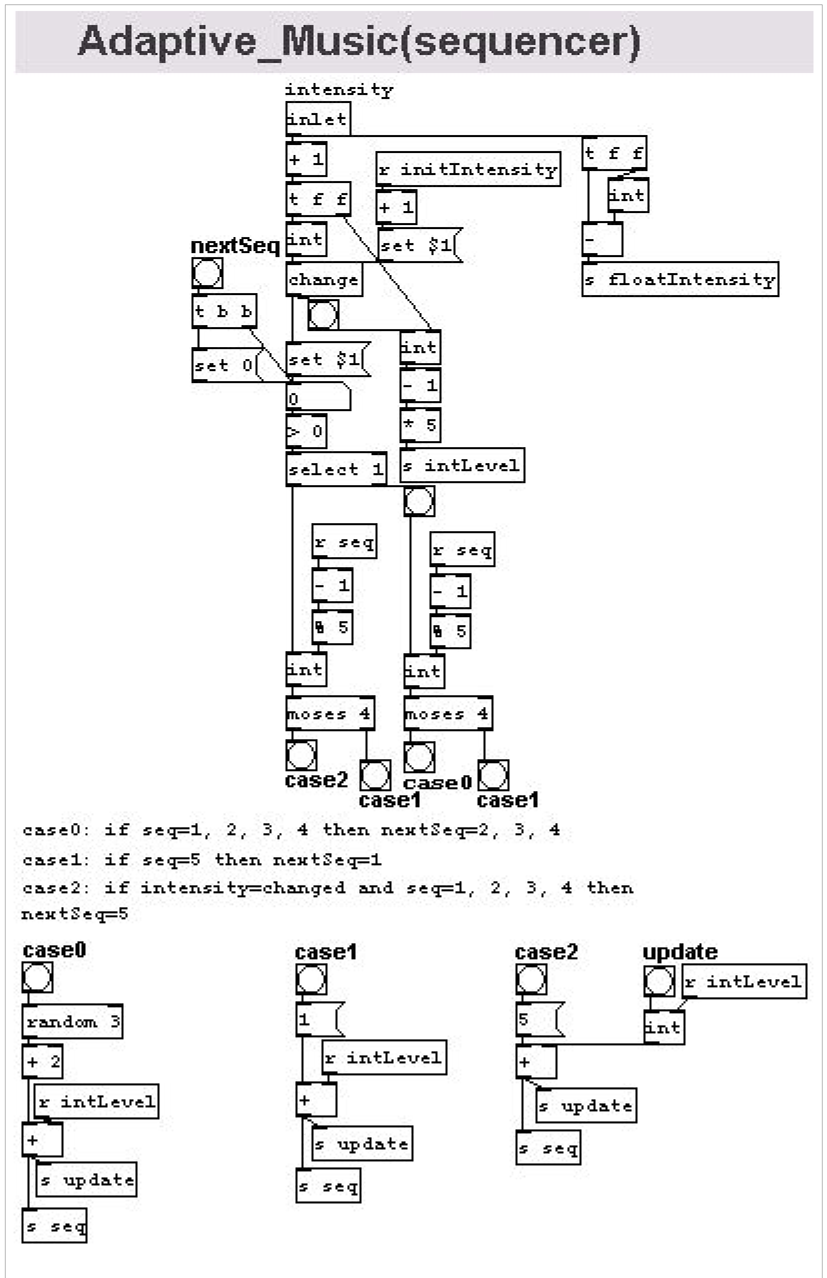
```
;
buf 0;
rand bang
```

```
;
play2 bang
```

```
;
buf 3;
rand bang
```

Figure 13

There are many ways to produce adaptive music control structures. Most streaming audio methods use a combination of branching or layering techniques. Entirely generative adaptive structures are also possible which have roots stretching back to the beginnings of electronic music, but those will not be explored within the scope of this article. Branching consists of playing back sampled music phrases and using the game state to decide what phrase to branch to next. Layering consists of breaking the musical phrases into multiple tracks, which can then be mixed and manipulated separately.



```
case0: if seq=1, 2, 3, 4 then nextSeq=2, 3, 4
case1: if seq=5 then nextSeq=1
case2: if intensity=changed and seq=1, 2, 3, 4 then
nextSeq=5
```

Figure 14

Utilizing branching, the Pure Data "Adaptive Music" prototype patch supports three different intensity levels of musical content. At each level, there is an intro phrase, an outro phrase and three variations on a middle looping phrase. When playback starts, it begins with the intro phrase of the appropriate intensity, branches to a random looping phrase variation and continues to do so until the intensity changes. When the intensity changes to a different value, it plays the outro phrase of the current intensity and branches to the intro phrase of the target intensity. This process is encapsulated in the logic of the "sequencer" subpatch.

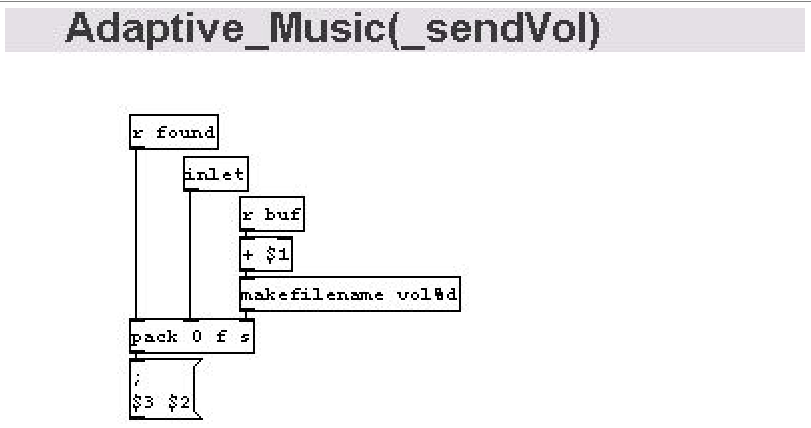
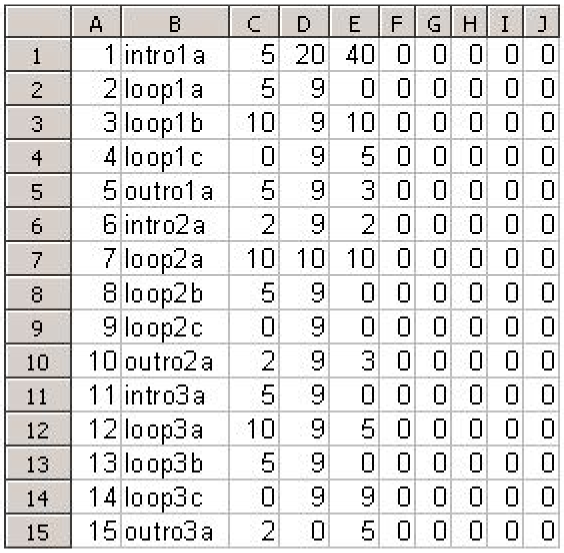


Figure 15

In combination with the branching control structure, this patch also uses layering techniques. Each phrase consists of three layers which can be modified separately. The fractional portion of the intensity is used to add to the general volume variability of each layer. The lower the intensity is within an intensity range, the more possibility there is variation in the volume level of a layer as defined by the spreadsheet. The layers can typically correspond to rhythm, melody and harmony for layers 1 through 3, respectively.

The combination of branching and layering control structures in this patch support logical transitions between intensity levels, layer variation when the intensity level is static, multiple variations on each intensity loop phrase, variable phrase lengths, and the possibility of silence.

A potential drawback of this system is that it could spend a lot of time in the transition sections if the intensity frequently fluctuates between intensity levels. Smoothing out the dynamics of the intensity values would likely reduce this possibility, and more transition variations would also permit more variation. This patch doesn't incorporate the concept of "stingers" which can be used as overlays to react more quickly to game-state changes. Since sections must play out in their entirety, this can also give the impression that the music isn't reactive enough to changes in the game's intensity. However, this can also be an advantage: if the music is too closely tied to events which the player can control, the player may discover how to "play" the music and subvert its normally passive influence.



	A	B	C	D	E	F	G	H	I	J
1	1	intro1a	5	20	40	0	0	0	0	0
2	2	loop1a	5	9	0	0	0	0	0	0
3	3	loop1b	10	9	10	0	0	0	0	0
4	4	loop1c	0	9	5	0	0	0	0	0
5	5	outro1a	5	9	3	0	0	0	0	0
6	6	intro2a	2	9	2	0	0	0	0	0
7	7	loop2a	10	10	10	0	0	0	0	0
8	8	loop2b	5	9	0	0	0	0	0	0
9	9	loop2c	0	9	0	0	0	0	0	0
10	10	outro2a	2	9	3	0	0	0	0	0
11	11	intro3a	5	9	0	0	0	0	0	0
12	12	loop3a	10	9	5	0	0	0	0	0
13	13	loop3b	5	9	0	0	0	0	0	0
14	14	loop3c	0	9	9	0	0	0	0	0
15	15	outro3a	2	0	5	0	0	0	0	0

Figure 16

This patch works by reading each row in a text file generated by a spreadsheet (shown in Figure 16) in the following format:

<Phrase ID#> <filename> <layer1 vol random %> <layer2 vol random %> <layer 3 vol random %> <5 parameters for future expansion>

Each five rows of the spreadsheet are grouped as follows: the first line is the intro phrase, followed by three loop phrases, then the outro phrase. See Figure 16 for an example.

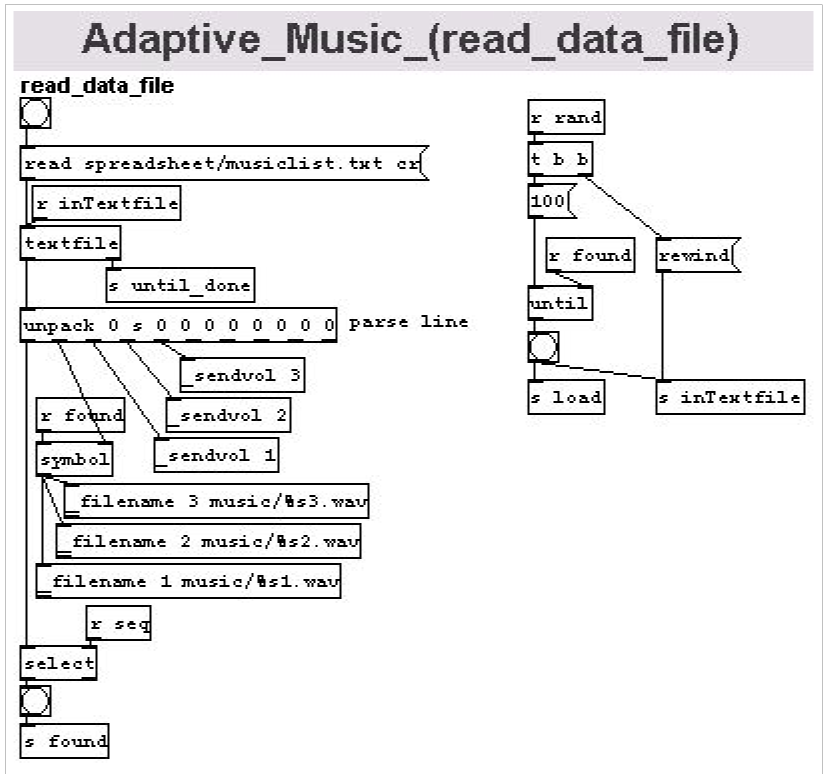


Figure 17

The rightmost five columns on each row are "reserved for future expansion" (i.e., whatever needs might arise). There are three intensity level groupings, which totals 15 lines for all the musical phrases.

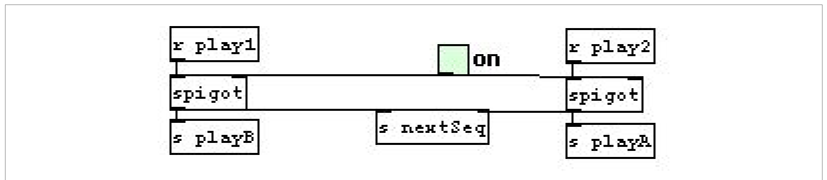


Figure 18

Each line of data is read into the patch's text-file object and split apart by the unpack object. The trick is that the data must be streamed into a double buffer to permit continuous, uninterrupted playback. If overly taxed, sometimes the audio can break up in Pure Data while it is loading the next buffer. The next phrase's layers and associated parameter data are read ahead of time, as the current phrase is playing. This way the data is ready at the instant the current phrase has completed playing, so the audio playback can immediately start playing when the old one is complete. At this point, the "line~" object is used to fade the current layer volume to the new layer's volume to allow smooth volume transitions.

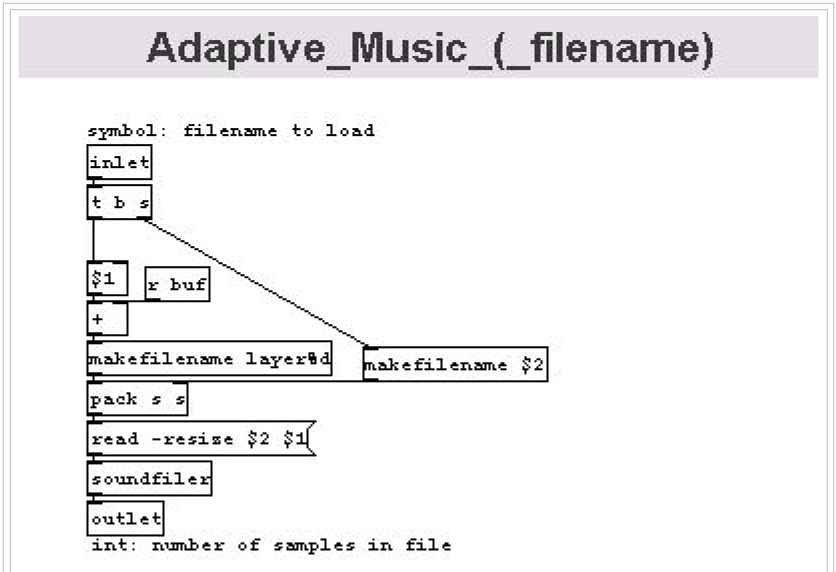


Figure 19

The "on" subpatch uses the graph-on-parent functionality which lets you hide the functions within the patch, but expose the controls such as the toggle switch on the parent patch. Turning the patch off causes playback to stop at the end of the next phrase.

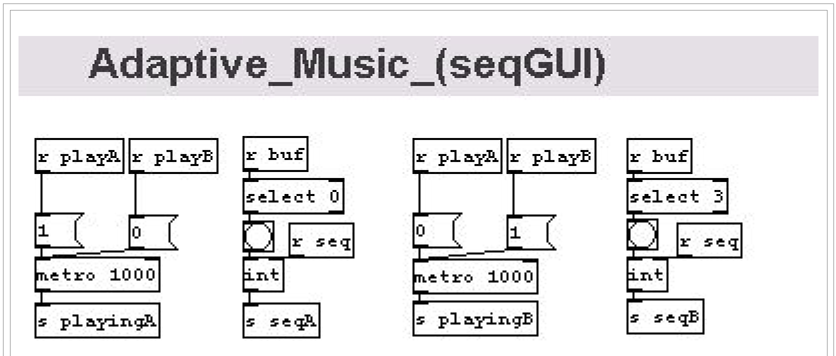


Figure 20

This patch lets you start trying out compositional ideas with phrases and layers, and immediately audition the changes while the patch is running. Since this patch only streams in the audio as needed, it is possible to change the source data while the patch is running and hear the changes in real time.

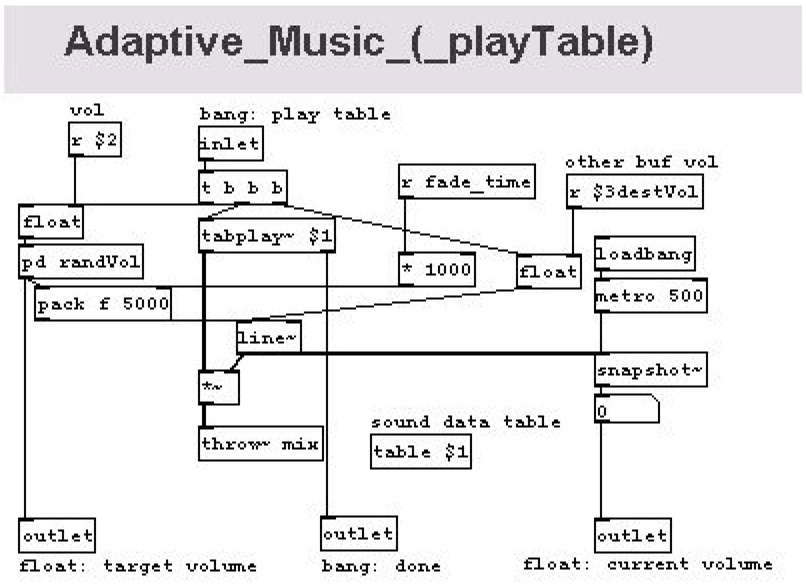


Figure 21

Many improvements could be made to this patch, but one of the first ought to be the creation of a weighting system so that certain phrases are more likely to get played after the current phrase than others. This technique is known as a "simple Markov chain" and could be used to create sequences where a phrase normally plays in a certain order, but sometimes plays differently from time to time. It would also be nice to add the possibility of crossfading, especially between the intro and outro phrases. But this would require extra logic and a crossfade buffer.

Implementing A Prototype

Before a composer gets too far with prototyping, he and the audio programmer should agree which objects are usable, what the maximum CPU utilization should be, the naming conventions for files and patches and so on. The composer might just create partial prototypes for the coder, but even simple ones can help immensely.

During the prototyping process, the coder should aid the composer with implementation issues and technical design decisions. The goal is to have the composer drive the process with support from the coder as needed. Hopefully the composer can stay productive and inspired enough to discover and define new behaviors to apply to his content. Knowing how the content can be controlled may change the manner in which the composer creates the content. The more the composer learns how to formally define interactive behaviors (using Pure Data in this case), the better equipped he will be to when it comes time to describe his goals to the coder.

When the prototype reaches a stage where the composer is pleased with the result, the functionality must be rebuilt by the coder. The audio coder should be familiar enough with Pure Data so that she can understand what the patch is doing and what the important elements are.

The crucial final step in prototyping is when the composer hands the prototype over to the coder for implementation. The composer and coder must trust each other during this process. The coder should feel comfortable enough to make modifications and feel a sense of ownership over the prototype. Many things are difficult to do in Pure Data that are easier in to build in C++ (and vice versa), so it's good to let the coder oversee the technical side of the prototyping process to ensure that the prototype is useful.

A nice benefit to having the coder clean up the prototype in Pure Data is that areas of code which should be reorganized into subpatches or abstractions often become visually obvious. Using software engineering lingo, code cohesion and encapsulation can become visually apparent. This process may even result in the creation of a small library of reusable components that can be applied to future projects. The coder may also find that there are repeated sections which could be better expressed in C++ and custom code objects for the composer to use in the future.

Although the coder will seemingly spend extra time reorganizing and throwing out the prototype code, this time spent will more than likely pay off in the long run with cleaner, easier to maintain code than code which awkwardly evolved through the prototype phase.

Table 2. Pure Data tips & tricks for more advanced users.

- Use "makefilename" to produce a dynamic send. For example to send to receives vol1 through vol4, use "makefilename vol%d" and connect to a message ";$1 $2" where $2 is the data you wish t audio effects, look for free VST plugins on the web and use the "vst~" object.
- Label inlets o send.
- To create objects at run-time use the ";pd-" message.
- For complexand outlets of your abstractions with a data type (bang, int, etc.) and a brief description so its easy to remember how to use them in the future and what they do.
- For network communication, try using the "netsend" and "netreceive" objects to use TCP/IP. This can also be used to communicate with other programs as well.
- With abstractions, use the dollar sign variables to set static parameters.
- To make group volume outputs, use many "throw~" objects to direct your audio at a single "catch~" object which corresponds to a group, modify the audio after the "catch~" and before the "dac~".

Prototype And Conquer

These examples barely scratch the surface of what is possible with prototyping using Pure Data. With advancements in the technology of audio on game platforms, prototyping will become increasingly necessary to harness the new power effectively. The overall goal of this method is to help everyone in the audio team contribute, maximizing their skill sets. The idea is not to turn composers into coders and vice versa; rather, prototyping can help bridge the gap between what can seem like two separate disciplines from the start. Be it technical or creative, everyone should feel open to contribute to the final product and share in the rewards.

Resources

- Miller Puckette's Pure Data site: http://crca.ucsd.edu/~msp/software.html
- Download Pure Data and many extensions: http://pure-data.sourceforge.net/download.php
- Installation instructions: http://www.crca.ucsd.edu/~msp/Pd_documentation/index.htm
- Pure Data documentation: http://crca.ucsd.edu/~msp/Pd_documentation/index.htm
- Official Pure Data website: http://pure-data.org
- Pure Data Newsgroup: http://www.iem.at/mailinglists/pd-list/

Archival images supplied by the Internet Moving Images Archive (at archive.org) in association with Prelinger Archives from "How to Listen To... New Dimensions in Sound."

Contact:
lotusaudio@shaw.ca