Examples of ZIPI Applications
Author(s): Matthew Wright
Source: *Computer Music Journal*, Vol. 18, No. 4 (Winter, 1994), pp. 81-85
Published by: The MIT Press
Stable URL: http://www.jstor.org/stable/3681360
Accessed: 10-04-2017 04:37 UTC

Matthew Wright
Center for New Music and Audio Technologies (CNMAT)
Department of Music, University of California, Berkeley
1750 Arch St.
Berkeley, California 94720 USA
Matt@CNMAT.Berkeley.edu

Examples of ZIPI Applications

This article lists some sample applications of ZIPI: alternate controllers, transfers of large amounts of data, and common applications from the MIDI world. I show how these applications would be implemented in ZIPI and, where appropriate, evaluate ZIPI's performance in each case. Basic knowledge of ZIPI is assumed.

A ZIPI Guitar

Zeta Music will soon release a combination ZIPI controller and synthesizer. The controller is a general-purpose sound-to-ZIPI converter; it takes arbitrary sound sources as its input and produces ZIPI control data. The synthesizer portion is the opposite; it takes incoming ZIPI control data and synthesizes sounds.

The controller will work with any instrument, but for now assume that it is connected to a guitar with a hexaphonic pickup. (*Hexaphonic* means one output channel for each of the six guitar strings instead of the usual case in which the sound of all six guitar strings comes out of one output.) The controller will track pitch, loudness, even/odd balance, pitched/unpitched balance, and brightness information for each of the six strings of the guitar in real time, updating each parameter every 8 to 10 msec. It will also produce articulation information, noting when trigger and release events occur on each string.

The built-in synthesizer will use sample playback and will be able to vary the pitch, loudness, even/odd balance, pitched/unpitched balance, and brightness of a sound. Thus the sounds produced by the synthesizer will be able to follow all these nuances of the acoustic sound, not just its pitch and volume.

Computer Music Journal, 18:4, pp. 81–85, Winter 1994
© 1994 Massachusetts Institute of Technology.

The intention is to give an instrumentalist such as a guitar player more expressive control over the sounds produced by a synthesizer. When the guitarist picks closer to the bridge, a synthesizer producing an organ sound would make it brighter. When the guitarist plays a 12th-fret harmonic, a trumpet sound would go up an octave and change to a quieter, more delicate timbre. When the guitarist partially mutes the strings with his or her palm, a saxophone sound would have a more breathy, noisy character. When the guitarist completely mutes the strings with the left hand and scratches rhythmically, a piano patch would produce just the sound of hammers hitting strings, with no pitched content.

This instrument will use ZIPI to control an external synthesizer with the same high-bandwidth continuous control information that it uses internally. As the default configuration, the controller will send the information from each guitar string to a separate MPDL note address. It would be useful for these six MPDL notes to be in the same instrument; this would make it possible to control all of the guitar's synthesized sound with a single message, e.g., pan or amplitude.

How much bandwidth does this require? Every 10 msec, the controller will send a ZIPI packet that looks like this:

Address of note 1
Pitch, loudness, brightness, even/odd balance, noise balance
New address: note 2
Pitch, loudness, brightness, even/odd balance, noise balance

and so on for the other four notes. Pitch and loudness are 3-byte messages (including the note descriptor ID); the other three are 2-byte messages. Thus there are 3 + 3 + 2 + 2 + 2 = 12 bytes of data per string, or 6 × 12 = 72 bytes of data altogether. The note address at the beginning of the frame is three bytes; the other five note addresses must be specified with 5-byte "new address" messages, for a total of 3 + (5 × 5) = 28 bytes of addresses. Including the 7 overhead bytes for each ZIPI packet, that is a total of 107 bytes per update.

One hundred and seven 8-bit bytes every 10 msec requires a bandwidth of 85.6 kBaud—just over a third of ZIPI's bandwidth at the slowest speed and more than two and a half times MIDI's bandwidth. This does not include articulation messages for triggering and releasing notes, but these would be much more than 10 msec apart on average and would not take up any appreciable amount of bandwidth.

What happens to the bandwidth if we add in many other ZIPI synthesizers to layer the sound? Nothing. ZIPI synthesis modules typically only listen to the network and do not send their own messages. Thus there could be one or 10 synthesizers connected to this controller, with the same network performance in either case.

A ZIPI Keyboard

ZIPI keyboards, like MIDI keyboards, would send messages whenever a key is pressed or released. A ZIPI keyboard would probably preallocate as many ZIPI notes as it has keys, sending each of them a pitch message only once as part of a setup routine. (For a keyboard, a note's address truly is the same as its pitch, so MIDI's model, in which the address is the same as the pitch, applies.)

A "note-on" packet would have a loudness note descriptor computed from the key's velocity, followed by an articulation note descriptor ("trigger"). A "note-off" packet would just consist of a single articulation note descriptor—"release." Seven overhead bytes plus a 3-byte address, 3 bytes for the loudness note descriptor, and 2 bytes for articulation equals 15 bytes. At 250 kBaud, this takes 480 μsec to transmit.

Like all ZIPI controllers, a ZIPI keyboard should be able to send raw controller measurements instead of mapping those measurements onto parameters like loudness and pitch. In this mode, key number would replace pitch, and key velocity would replace loudness.

It is possible that a ZIPI keyboard controller would want to implement its own note-stealing algorithm rather than rely on that of the synthesizer being controlled. In that case, it could allocate as many notes as the receiving synthesizer has voices of polyphony. In each packet that contains a trigger message, there would also be a pitch message. Now, the keyboard knows that any note it asks the synthesizer to play will be played, and if a note needs to be turned off, the keyboard can choose which one. The keyboard controller could even control multiple synthesizers of the same type, allocating notes among them according to the polyphony capabilities.

On a ZIPI keyboard, alternate tunings would be a feature of the keyboard controller, not the synthesizer. Remember that in ZIPI, pitch is a 16-bit quantity. The keyboard already has a mapping between key numbers and pitches; for example, it knows that the middle C key has the ZIPI pitch with the hexadecimal value 7900. If the musician wants alternate tunings, the keyboard could just use a different mapping, perhaps associating the middle C key with ZIPI pitch hexadecimal 78E2 or hex 792A.

How to Do Multis

Most MIDI timbre modules are "multi-timbral," meaning that the same synthesizer can produce pitches with different timbres on different MIDI channels. Because MIDI has only 16 channels, it is important to choose which 16 timbres to use at a time. Therefore, many synthesizers incorporate the concept of a "multi," which is a collection of up to 16 timbres. Sometimes it is possible to select a whole multi all at once, which is the equivalent of sending program change messages on all 16 MIDI channels. What is the equivalent mechanism in ZIPI?

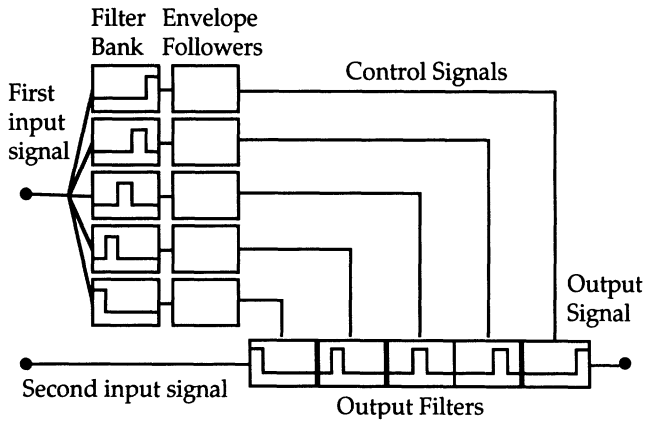
In ZIPI, there are 8,001 instruments in 63 families, so it is easy to set up an instrument with every timbre that might be needed and then choose timbre just by selecting a particular instrument to trigger. However, if you like the idea of "swapping in" a set of instruments (i.e., changing 16 timbres at once), it would still be easy via ZIPI. The ZIPI controller would have a data structure similar to a MIDI multi, but of any size, and possibly spanning multiple synthesizers. At the push of a button, the controller could send program change messages to all the appropriate instruments of all the appropriate synthesizers.

Y-splitters and Mergers

In MIDI, two common tools are the Y-splitter and the merger. The Y-splitter has one MIDI input and multiple MIDI outputs, all of which are copies of the input signal. (Thus, it looks like the letter "Y.") It is useful to send the control information from one computer or musical instrument to multiple synthesizers (e.g., to make them play in unison). The merger performs the opposite function, taking some number of MIDI inputs and combining the MIDI messages logically into one single stream of MIDI data at the output. It is useful for controlling the same synthesizer or computer with two different MIDI instruments.

Neither of these is required in ZIPI. Any collection of ZIPI devices can be on the same network, and any device can send a message to any other. If two devices want to send messages to the same synthesizer, they both send data to the appropriate network device address. Likewise, if a device wants to send a message to two synthesizers, nothing needs to change in the ring configuration. The sending device can just send two messages addressed to the two devices, and both will reach their destinations. If the message is intended for all devices in the ring, it can be broadcast, meaning that every device sees the same message.

Figure 1. Components of a vocoder.



A ZIPI Vocoder

A vocoder is a musical instrument that applies the frequency spectrum of one sound source to a second sound source. The first input signal is analyzed for its frequency content, and the result is used as the parameters of a filter applied to the second input signal. (See, for example, Moore [1990] or Buchla [1974] for an explanation of this technique.)

A vocoder might consist of the components shown in Figure 1. The first input signal passes through the analysis filter bank and envelope followers, which produce a set of continuously varying control signals. These signals represent the amplitude of the control signal in each frequency region. The control signals are then used to control the gains of the components of a filter bank.

One interesting variant on vocoding is applying some transformation to the control signals between the time when they are produced by the analysis portion and when they are used to control the output filter bank (Buchla 1974). For example, one might want to accentuate the effect of the first input signal (e.g., by squaring the control signals). Another interesting transformation would be to have the overall amplitude of the first input signal control the brightness imposed by the output filter bank by selectively increasing the control signals for the higher-frequency filters. Yet another possibility would be to frequency-shift the control signals so that each control signal would control the gain of the next higher filter.

These kinds of transformations are easy if the two halves of the vocoder communicate via ZIPI. Suppose we have a vocoder with 20 filter segments, spaced in some reasonable way across the frequency spectrum. We would then use ZIPI to send the 20 control signals. For the usual vocoder behavior, we would simply connect the control ZIPI stream directly to the synthesis filter bank. For more elaborate behaviors, we would apply some transformation to the ZIPI signal, presumably with a computer program.

Exactly what would we send over ZIPI? In the best case, the analysis filter bank would produce logarithmic amplitudes, and the synthesis filter banks would expect a logarithmic control signal, so we could get by with 8-bit data. Because all 20 of these numbers would be produced at once, there is no need to set them individually, so we would probably send them as a single 20-data-byte note descriptor. We would need to pick an undefined *n*-byte note descriptor (e.g., hexadecimal C9) for this. We might sample the envelope followers in the analysis bank every 5 msec, so we would update this note descriptor every 5 msec. The update packet would have 7 bytes of overhead, a 3-byte address, a 1-byte note descriptor ID, another 2 bytes to indicate 20 data bytes, and the data bytes themselves, for a total of 33 bytes. So at 250 kBaud, it would take just over a millisecond to send this packet, thus using about a fifth of ZIPI's bandwidth.

If the filter banks used linear control signals instead of logarithmic ones, 8 bits would not be enough resolution, so we would switch to 16-bit control signals. Thus, our packets would be 53 data bytes, taking 1.7 msec to send. In the worst case, the two filter banks would use different units, requiring that there be some mapping function, such as linear-to-logarithmic conversion.

Sample Dumps

ZIPI will have a separate application layer for audio samples, with a fixed header format for specifying the sampling rate, number of channels, etc. This header is likely to be the same as an existing sound file standard (van Rossum 1993) but might be specially designed for ZIPI. In most cases, sample dumps are a type of packet that would be sent with "guaranteed delivery," meaning that the receiving device has to acknowledge successful receipt of the packet.

The longest legal ZIPI packet has 4,096 data bytes plus 7 overhead bytes, so samples will have to be broken up into multiple ZIPI packets. After getting one of these packets, the receiving device would have to send back an acknowledgment of around 8 bytes. Therefore, it takes 4,111 bytes over the network to reliably transmit 4,096 data bytes—99.64 percent efficiency. ZIPI's lower network levels do a hardware CRC checksum, so if there is any corruption of the data, the receiving device will know it. (Of course, if there is data corruption, it will be necessary to retransmit those data bytes, slowing down the process. But that is better than a garbled sample file!)

Assuming 1-bit monophonic PCM data, 4,096 bytes is 2,048 samples. At a 44.1-kHz sampling rate, that would be about 46 msec of sound.

In a ZIPI network running at 1 MBaud, it would take about 33 msec to transmit 4,111 bytes, so it would be about 40 percent faster than real time to transmit CD-quality samples. Of course, a ZIPI network running at 20 MBaud would be 20 times faster, transmitting those 4,111 bytes in 1.64 msec—28 times faster than real time.

Real-time Digital Audio

Real-time digital audio has something in common with sample dumps—PCM audio data is sent at high speeds over ZIPI—but there are also other issues that must be resolved. First of all, obviously, the network must be able to send the data at faster than real time!

Variations in network latency also add complication to real-time digital audio. To solve this, the sending device should time-tag all outgoing audio messages. The receiving device can then buffer data before playing it, imposing a small but fixed delay on the audio stream. Anderson and Kuivila (1986) give an explanation of this process.

For example, consider the case of transmitting CD-quality monophonic digital audio over a 1-MBaud ZIPI network. As mentioned above, there is more than enough bandwidth, by 40 percent, for this. What are the expected and worst-case latencies? A maximum-length ZIPI packet and its acknowledgment require 33 msec to transmit, so that is the expected (and best-case) latency, but if a packet is lost, it will require another 33 msec to retransmit. To be conservative, let us say that the worst-case latency is 200 msec under "normal" conditions. Due to network considerations, this is of course theoretically unbounded.

The receiving device would allocate a buffer of 8,820 samples (i.e., 200 msec). When it receives a packet of samples, it writes them into the buffer at the appropriate location, based on the packet's time stamp. It would read data out of the buffer 200 msec after the device sends it, so the buffer will usually be close to full as long as no packets are lost and there is no unexpected network activity. If a packet is lost, the buffer will continue to be emptied by the playing device but will temporarily stop being filled by the sending device. However, we expect that the sending device will get some data through before the 200-msec buffer empties entirely. Because the network can transmit 40 percent faster than real time, the sending synthesizer can use that 40 percent to catch up, eventually nearly filling the buffer again.

Note that the maximum packet size of 4,096 bytes is not the only possible packet size for digital audio. Smaller packet sizes would require more bandwidth (because the amount of overhead per packet is constant) but would give less latency. Packets of 256 bytes would reduce efficiency to 94.5 percent but would reduce the expected latency to 2.168 msec. In an extreme case, 8-data-byte packets would reduce efficiency to less than 35 percent but would give 0.184 msec latency.

References

Anderson, D., and R. Kuivila. 1986. "Accurately Timed Generation of Discrete Musical Events." *Computer Music Journal* 10(3):48–56.

Buchla, D. 1974. *Model 296 Spectral Processor Manual.* Berkeley, California: Buchla and Associates.

Moore, F. R. 1990. *Elements of Computer Music.* Englewood Cliffs, New Jersey: Prentice Hall.

van Rossum, G. 1993. *FAQ: Audio File Formats (version 2.10).* From Internet server mitpress.mit.edu, file /pub/Computer-Music-Journal/Documents/SoundFiles/AudioFormats2.10.t.