Extension 5: Sound

Text by R. Luke DuBois

The history of music is, in many ways, the history of technology. From developments in the writing and transcription of music (notation) to the design of spaces for the performance of music (acoustics) to the creation of musical instruments, composers and musicians have availed themselves of advances in human understanding to perfect and advance their professions. Unsurprisingly, therefore, we find that in the machine age these same people found themselves first in line to take advantage of the new techniques and possibilities offered by electricity, telecommunications, and, in the last century, digital computers to leverage all of these systems to create new and expressive forms of sonic art. Indeed, the development of phonography (the ability to reproduce sound mechanically) has, by itself, had such a transformative effect on aural culture that it seems inconceivable now to step back to an age where sound could emanate only from its original source.[1] The ability to create, manipulate, and losslessly reproduce sound by digital means is having, at the time of this writing, an equally revolutionary effect on how we listen. As a result, the artist today working with sound has not only a huge array of tools to work with, but also a medium exceptionally well suited to technological experimentation.

Music and sound programming in the arts

Thomas Edison's 1857 invention of the phonograph and Nikola Tesla's wireless radio demonstration of 1893 paved the way for what was to be a century of innovation in the electromechanical transmission and reproduction of sound. Emile Berliner's gramophone record (1887) and the advent of AM radio broadcasting under Guglielmo Marconi (1922) democratized and popularized the consumption of music, initiating a process by which popular music quickly transformed from an art of minstrelsy to a commodified industry worth tens of billions of dollars worldwide.[2] New electronic musical instruments, from the large and impractical telharmonium to the simple and elegant theremin multiplied in tandem with recording and broadcast technologies and prefigured the synthesizers, sequencers, and samplers of today. Many composers of the time were, not unreasonably, entranced by the potential of these new mediums of transcription, transmission, and performance. Luigi Russolo, the futurist composer, wrote in his 1913 manifesto The *Art of Noises* of a futurist orchestra harnessing the power of mechanical noisemaking (and phonographic reproduction) to "liberate" sound from the tyranny of the merely musical. John Cage, in his 1937 monograph *Credo: The Future of Music*, wrote this elliptical doctrine:

> The use of noise to make music will continue and increase until we reach a music produced through the aid of electrical instruments which will make available for musical purposes any and all sounds that can be heard. Photoelectric, film, and mechanical mediums for the synthetic production of music will be explored. Whereas, in the past, the point of disagreement has been between dissonance and consonance, it will be, in the immediate future, between noise and so-called musical sounds.3

The invention and wide adoption of magnetic tape as a medium for the recording of audio signals provided a breakthrough for composers waiting to compose purely with *sound*. In the early postwar period, the first electronic music studios flourished at radio stations in Paris (ORTF) and Cologne (WDR). The composers at the Paris studio, most notably Pierre Henry and Pierre Schaeffer, developed the early compositional technique of *musique concrète*, working directly with recordings of sound on phonographs and magnetic tape to construct compositions through a process akin to what we would now recognize as sampling. Schaeffer's *Étude aux chemins de fer* (1948) and Henry and Schaeffer's *Symphonie pour un homme seul* are classics of the genre. Meanwhile, in Cologne, composers such as Herbert Eimart and Karlheinz Stockhausen were investigating the use of electromechanical oscillators to produce pure sound waves that could be mixed and sequenced with a high degree of precision. This classic *elektronische music* was closely tied to the serial techniques of the contemporary modernist avant-garde, who were particularly well suited aesthetically to become crucial advocates for the formal quantification and automation offered by electronic and, later, computer music.[4] The Columbia-Princeton Electronic Music Center, founded by Vladimir Ussachevsky, Otto Luening, Milton Babbitt, and Roger Sessions in New York in 1957, staked its reputation on the massive RCA Mark II Sound Synthesizer, a room-sized machine capable of producing and sequencing electronically generated tones with an unprecedented degree of precision and control. In the realm of popular music, pioneering steps were taken in the field of recording engineering, such as the invention of multitrack tape recording by the guitarist Les Paul in 1954. This technology, enabling a single performer to "overdub" her/himself onto multiple individual "tracks" that could later be mixed into a composite, filled a crucial gap in the technology of recording and would empower the incredible boom in recording-studio experimentation that permanently cemented the commercial viability of the studio recording in popular music.

Composers adopted digital computers slowly as a creative tool because of their initial lack of real-time responsiveness and intuitive interface. Although the first documented use of the computer to make music occurred in 1951 on the CSIRAC machine in Sydney, Australia, the genesis of most foundational technology in computer music as we know it today came when Max Mathews, a researcher at Bell Labs in the United States, developed a piece of software for the IBM 704 mainframe called MUSIC. In 1957, the MUSIC program rendered a 17-second composition by Newmann Guttmann called "In the Silver Scale". Originally tasked with the development of human-comprehensible synthesized speech, Mathews developed a system for encoding and decoding sound waves digitally, as well as a system for designing and implementing digital audio processes computationally. His assumptions about these representational schemes are still largely in use and will be described later in this text. The advent of faster machines,

computer music programming languages, and digital systems capable of real-time interactivity brought about a rapid transition from analog to computer technology for the creation and manipulation of sound, a process that by the 1990s was largely comprehensive.[5]

Sound programmers (composers, sound artists, etc.) use computers for a variety of tasks in the creative process. Many artists use the computer as a tool for the algorithmic and computer-assisted composition of music that is then realized off-line. For Lejaren Hiller's *Illiac Suite* for string quartet (1957), the composer ran an algorithm on the computer to generate notated instructions for live musicians to read and perform, much like any other piece of notated music. This computational approach to composition dovetails nicely with the aesthetic trends of twentieth-century musical modernism, including the controversial notion of the composer as "researcher," best articulated by serialists such as Milton Babbitt and Pierre Boulez, the founder of IRCAM. This use of the computer to manipulate the symbolic language of music has proven indispensable to many artists, some of whom have successfully adopted techniques from computational research in artificial intelligence to attempt the modeling of preexisting musical styles and forms; for example, David Cope's *5000 works…* and Brad Garton's *Rough Raga Riffs* use stochastic techniques from information theory such as Markov chains to simulate the music of J. S. Bach and the styles of Indian Carnatic sitar music, respectively.

If music can be thought of as a set of informatics to describe an organization of sound, the synthesis and manipulation of sound itself is the second category in which artists can exploit the power of computational systems. The use of the computer as a producer of synthesized sound liberates the artist from preconceived notions of instrumental capabilities and allows her/him to focus directly on the *timbre* of the sonic artifact, leading to the trope that computers allow us to make any sound we can imagine. Composers such as Jean-Claude Risset (*The Bell Labs Catalogue*), Iannis Xenakis (*GENDYN3*), and Barry Truax (*Riverrun*), have seen the computer as a crucial tool in investigating sound itself for compositional possibilities, be they imitative of real instruments (Risset), or formal studies in the stochastic arrangements of synthesized sound masses (Xenakis) using techniques culminating in the principles of granular synthesis (Truax). The computer also offers extensive possibilities for the assembly and manipulation of preexisting sound along the musique concrète model, though with all the alternatives a digital computer can offer. The compositional process of digital sampling, whether used in pop recordings (Brian Eno and David Byrne's *My Life in the Bush of Ghosts*, Public Enemy's *Fear of a Black Planet*) or conceptual compositions (John Oswald's *Plunderphonics*, Chris Bailey's *Ow, My Head*), is aided tremendously by the digital form sound can now take. Computers also enable the transcoding of an audio signal into representations that allow for radical reinvestigation, as in the time-stretching works of Leif Inge (*9 Beet Stretch*, a 24-hour "stretching" of Beethoven's Ninth Symphony) and the time-lapse phonography of this text's author (*Messiah*, a 5-minute "compression" of Handel's *Messiah*).

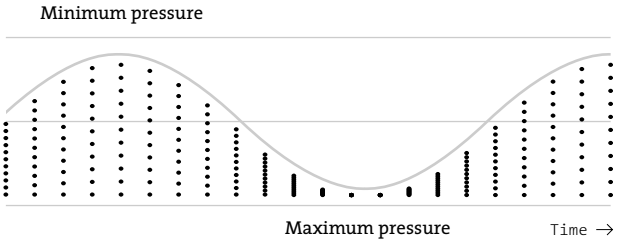
Artists working with sound will often combine the two approaches, allowing for the creation of generative works of sound art where the underlying structural system, as well as the sound generation and delivery, are computationally determined. Artists

such as Michael Schumacher, Stephen Vitiello, Carl Stone, and Richard James (the Aphex Twin) all use this approach. Most excitingly, computers offer immense possibilities as actors and interactive agents in sonic *performance*, allowing performers to integrate algorithmic accompaniment (George Lewis), hyperinstrument design (Laetitia Sonami, *Interface*), and digital effects processing (Pauline Oliveros, Mari Kimura) into their performance repertoire.

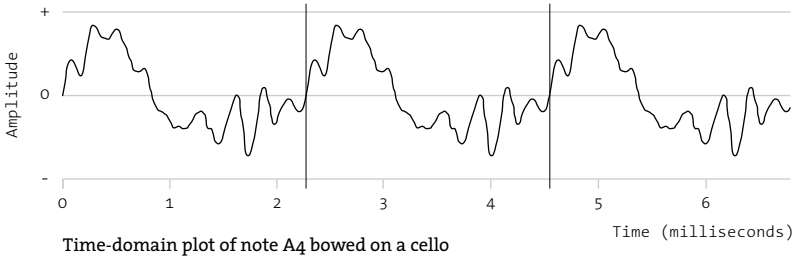
Now that we've talked a bit about the potential for sonic arts on the computer, we'll investigate some of the specific underlying technologies that enable us to work with sound in the digital domain.

Sound and musical informatics

Simply put, we define sound as a vibration traveling through a medium (typically air) that we can perceive through our sense of hearing. Sound propagates as a longitudinal wave that alternately compresses and decompresses the molecules in the matter (e.g., air) through which it travels. As a result, we typically represent sound as a plot of pressure over time:



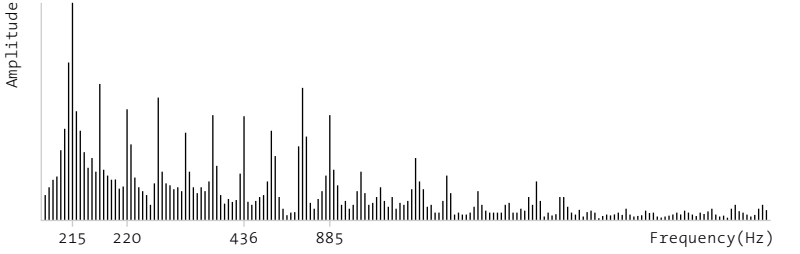
This time-domain representation of sound provides an accurate portrayal of how sound works in the real world, and, as we shall see shortly, it is the most common representation of sound used in work with digitized audio. When we attempt a technical description of a sound wave, we can easily derive a few metrics to help us better understand what's going on. In the first instance, by looking at the amount of displacement caused by the sound pressure wave, we can measure the amplitude of the sound. This can be measured on a scientific scale in *pascals* of pressure, but it is more typically quantified along a logarithmic scale of *decibels*. If the sound pressure wave repeats in a regular or *periodic* pattern, we can look at the wavelength of a single iteration of that pattern and from there derive the frequency of that wave. For example, if a sound traveling in a medium at 343 meters per second (the speed of sound in air at room temperature) contains a wave that repeats every half-meter, that sound has a frequency of 686 hertz, or cycles per second. The figure below shows a plot of a cello note sounding at 440 Hz; as a result, the periodic pattern of the waveform (demarcated with vertical lines) repeats every 2.27 ms:



Time-domain plot of note A4 bowed on a cello

Typically, sounds occurring in the natural world contain many discrete frequency components. In noisy sounds, these frequencies may be completely unrelated to one another or grouped by a typology of boundaries (e.g., a snare drum may produce frequencies randomly spread between 200 and 800 hertz). In *harmonic* sounds, however, these frequencies are often spaced in integer ratios, such that a cello playing a note at 200 hertz will produce frequencies not only at the *fundamental* of 200, but at multiples of 200 up the *harmonic series*, i.e., at 400, 800, 1200, 1600, 2000, and so on. A male singer producing the same note will have the same frequency components in his voice, though in different proportions to the cello. The presence, absence, and relative strength of these harmonics (also called *partials* or *overtones*) provide what we perceive as the timbre of a sound.

When a sound reaches our ears, an important sensory translation happens which is important to understand when working with audio. Just as light of different wavelengths and brightness excites different retinal receptors in your eyes to produce a color image, the cochlea of your inner ear contains an array of hair cells on the basilar membrane that are tuned to respond to different frequencies of sound. The inner ear contains hair cells that respond to frequencies spaced roughly between 20 and 20,000 hertz, though many of these hairs will gradually become desensitized with age or exposure to loud noise. These cells in turn send electrical signals via your auditory nerve into the auditory cortex of your brain, where they are parsed to create a frequency-domain image of the sound arriving in your ears:



Frequency-domain plot of a sustained note (220 Hz) bowed on a cello

This representation of sound, as a discrete "frame" of frequencies and amplitudes independent of time, is more akin to the way in which we perceive our sonic environment than the raw pressure wave of the time domain. Jean-Baptiste-Joseph

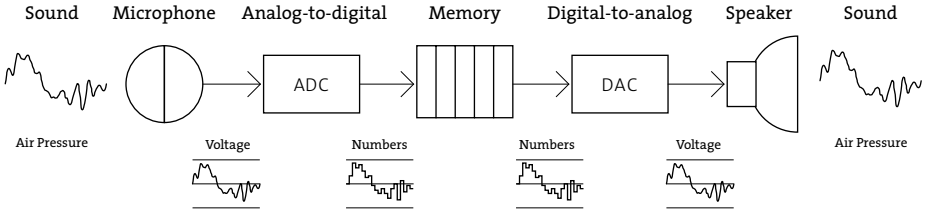
Fourier, a nineteenth-century French mathematician, developed the equations that allow us to translate a sound pressure wave (no matter how complex) into its constituent frequencies and amplitudes. This *Fourier transform* is an important tool in working with sound in the computer.

Our auditory system takes these streams of frequency and amplitude information from our two ears and uses them to construct an auditory "scene," akin to the visual scene derived from the light reaching our retinas.[6] Our brain analyzes the acoustic information based on a number of parameters such as onset time, stereo correlation, harmonic ratio, and complexity to parse out a number of acoustic sources that are then placed in a three-dimensional image representing what we hear. Many of the parameters that psychoacousticians believe we use to comprehend our sonic environment are similar to the grouping principles defined in Gestalt psychology.

If we loosely define music as the organization and performance of sound, a new set of metrics reveals itself. While a comprehensive overview of music theory, Western or otherwise, is well beyond the scope of this text, it's worth noting that there is a vocabulary for the description of music, akin to how we describe sound. Our system for perceiving *loudness* and *pitch* (useful "musical" equivalents to amplitude and frequency) work along a logarithmic scale, such that a tone at 100 hertz and a tone at 200 hertz are considered to be the same *distance* apart in terms of pitch as tones at 2000 and 4000 hertz. The distance between two sounds of doubling frequency is called the *octave*, and is a foundational principle upon which most culturally evolved theories of music rely. Most musical cultures then subdivide the octave into a set of pitches (e.g., 12 in the Western chromatic scale, 7 in the Indonesian *pelog* scale) that are then used in various collections (*modes* or *keys*). These pitches typically reflect some system of *temperament* or *tuning*, so that multiple musicians can play together; for example, the note A4 (the A above middle C) on the Western scale is usually calibrated to sound at 440 hertz in contemporary music.

Digital representation of sound and music

Sound typically enters the computer from the outside world (and vice versa) according to the time-domain representation explained earlier. Before it is digitized, the acoustic pressure wave of sound is first converted into an electromagnetic wave of sound that is a direct analog of the acoustic wave. This electrical signal is then fed to a piece of computer hardware called an analog-to-digital converter (ADC or A/D), which then digitizes the sound by sampling the amplitude of the pressure wave at a regular interval and quantifying the pressure readings numerically, passing them upstream in small packets, or vectors, to the main processor, where they can be stored or processed. Similarly, vectors of digital samples can be sent downstream from the computer to a hardware device called a digital-to-analog converter (DAC or D/A) which takes the numeric values and uses them to construct a smoothed-out electromagnetic pressure wave that can then be fed to a speaker or other device for playback:



Most contemporary digital audio systems (soundcards, etc.) contain both A/D and D/A converters (often more than one of each, for stereo or multichannel sound recording and playback) and can use both simultaneously (so-called full duplex audio). The specific system of encoding and decoding audio using this methodology is called PCM (or pulse-code modulation); developed in 1937 by Alec Reeves, it is by far the most prevalent scheme in use today.

The speed at which audio signals are digitized is referred to as the *sampling rate*; it is the resolution that determines the highest frequency of sound that can be measured (equal to half the sampling rate, according to the *Nyquist theorem*). The numeric resolution of each sample in terms of computer storage space is called the *bit depth*; this value determines how many discrete levels of amplitude can be described by the digitized signal. The digital audio on a compact disc, for example, is digitized at 44,100 hertz with a 16-bit resolution, allowing for frequencies up to 22,050 hertz (i.e., just above the range of human hearing) with 65,536 (216) different levels of amplitude possible for each sample. Professional audio systems will go higher (96 or 192 kHz at 24- or 32-bit resolution) while industry telephony systems will go lower (e.g., 8,192 Hz at 8-bit). Digitized sound representing multiple acoustic sources (e.g., instruments) or destinations (e.g., speakers) is referred to as multi-channel audio. Monaural sound consists of, naturally, only one stream; stereo (two-stream) audio is standard on all contemporary computer audio hardware, and various types of surround-sound (five or seven streams of audio with one or two special channels for low frequencies) are becoming more and more common.

Once in the computer, sound is stored using a variety of formats, both as sequences of PCM samples and in other representations. The two most common PCM sound file formats are the Audio Interchange File Format (AIFF) developed by Apple Computer and Electronic Arts and the WAV file format developed by Microsoft and IBM. Both formats are effectively equivalent in terms of quality and interoperability, and both are inherently *lossless* formats, containing the uncompressed PCM data from the digitized source. In recent years, compressed audio file formats have received a great deal of attention, most notably the MP3 (MPEG-1 Audio Layer 3), the Vorbis codec, and the Advanced Audio Coding (AAC) codec. Many of these "lossy" audio formats translate the sound into the frequency domain (using the Fourier transform or a related technique called Linear Predictive Coding) to package the sound in a way that allows compression choices to be made based on the human hearing model, by discarding perceptually irrelevant frequencies in the sound. Unlike the PCM formats outlined above, MP3 files are much harder to encode, manipulate, and process in real time, because of the extra step required to decompress and compress the audio into and out of the time domain.

Synthesis

Digital audio systems typically perform a variety of tasks by running processes in *signal processing networks*. Each node in the network typically performs a simple task that either generates or processes an audio signal. Most software for generating and manipulating sound on the computer follows this paradigm, originally outlined by Max Mathews as the *unit generator* model of computer music, where a map or function graph of a signal processing chain is executed for every sample (or vector of samples) passing through the system. A simple algorithm for synthesizing sound with a computer could be implemented using this paradigm with only three unit generators, described as follows.

First, let's assume we have a unit generator that generates a repeating sound waveform and has a controllable parameter for the frequency at which it repeats. We refer to this piece of code as an oscillator. Most typical digital oscillators work by playing back small tables or arrays of PCM audio data that outlines a specific waveform. These *wavetables* can contain incredibly simple patterns (e.g., a sine or square wave) or complex patterns from the outside world (e.g., a professionally recorded segment of a piano playing a single note).

If we play our oscillator directly (i.e., set its frequency to an audible value and route it directly to the D/A) we will hear a constant tone as the wavetable repeats over and over again. In order to attain a more nuanced and articulate sound, we may want to vary the volume of the oscillator over time so that it remains silent until we want a sound to occur. The oscillator will then increase in volume so that we can hear it. When we want the sound to silence again, we fade the oscillator down. Rather than rewriting the oscillator itself to accommodate instructions for volume control, we could design a second unit generator that takes a list of time and amplitude instructions and uses those to generate a so-called *envelope*, or ramp that changes over time. Our *envelope generator* generates an audio signal in the range of 0 to 1, though the sound from it is never experienced directly. Our third unit generator simply multiplies, sample per sample, the output of our oscillator with the output of our envelope generator. This *amplifier* code allows us to use our envelope ramp to dynamically change the volume of the oscillator, allowing the sound to fade in and out as we like.

In a commercial synthesizer, further algorithms could be inserted into the signal network, for example a filter that could shape the frequency content of the oscillator before it gets to the amplifier. Many synthesis algorithms depend on more than one oscillator, either in parallel (e.g., additive synthesis, in which you create a rich sound by adding many simple waveforms) or through *modulation* (e.g., frequency modulation, where one oscillator modulates the pitch of another).

Sampling

Rather than using a small waveform in computer memory as an oscillator, we could use a longer piece of recorded audio stored as an AIFF or WAV file on our computer's hard disk. This *sample* could then be played back at varying rates, affecting its pitch. For example, playing back a sound at twice the speed at which it was recorded will result in

its rising in pitch by an octave. Similarly, playing a sound at half speed will cause it to drop in pitch by an octave.

Most samplers (i.e., musical instruments based on playing back audio recordings as sound sources) work by assuming that a recording has a *base frequency* that, though often linked to the real pitch of an instrument in the recording, is ultimately arbitrary and simply signifies the frequency at which the sampler will play back the recording at normal speed. For example, if we record a cellist playing a sound at 220 hertz (the musical note A below middle C in the Western scale), we would want that recording to play back normally when we ask our sampler to play us a sound at 220 hertz. If we ask our sampler for a sound at a different frequency, our sampler will divide the requested frequency by the base frequency and use that ratio to determine the playback speed of the sampler. For example, if we want to hear a 440 hertz sound from our cello sample, we play it back at double speed. If we want to hear a sound at middle C (261.62558 hertz), we play back our sample at 1.189207136 times the original speed.

Many samplers use recordings that have meta-data associated with them to help give the sampler algorithm information that it needs to play back the sound correctly. The base frequency is often one of these pieces of information, as are *loop points* within the recording that the sampler can safely use to make the sound repeat for longer than the length of the original recording. For example, an orchestral string sample loaded into a commercial sampler may last for only a few seconds, but a record producer or keyboard player may need the sound to last much longer; in this case, the recording is designed so that in the middle of the recording there is a region that can be safely repeated, ad infinitum if need be, to create a sense of a much longer recording.

Effects processing

In addition to serving as a generator of sound, computers are used increasingly as machines for *processing* audio. The field of digital audio processing (DAP) is one of the most extensive areas for research in both the academic computer music communities and the commercial music industry. Faster computing speeds and the increased standardization of digital audio processing systems has allowed most techniques for sound processing to happen in real time, either using software algorithms or audio DSP coprocessors such as the Digidesign TDM and T|C Electronics Powercore cards.

As we saw with audio representation, audio effects processing is typically done using either time- or frequency-domain algorithms that process a stream of audio vectors. An echo effect, for example, can be easily implemented by creating a buffer of sample memory to delay a sound and play it back later, mixing it in with the original. Extremely short delays (of one or two samples) can be used to implement digital filters, which attenuate or boost different frequency ranges in the sound. Slightly longer delays create resonation points called *comb filters* that form an important building block in simulating the short echoes in room reverberation. A variable-delay comb filter creates the resonant swooshing effect called *flanging*. Longer delays are used to create a variety of echo, reverberation, and looping systems, and can also be used to create *pitch shifters* (by varying the playback speed of a slightly delayed sound).

Audio analysis

A final important area of research, especially in interactive sound environments, is the derivation of information from audio analysis. Speech recognition is perhaps the most obvious application of this, and a variety of paradigms for recognizing speech exist today, largely divided between "trained" systems (which accept a wide vocabulary from a single user) and "untrained" systems (which attempt to understand a small set of words spoken by anyone). Many of the tools implemented in speech recognition systems can be abstracted to derive a wealth of information from virtually any sound source.

Interactive systems that "listen" to an audio input typically use a few simple techniques to abstract a complex sound source into a control source that can be mapped as a parameter in interaction design. For example, a plot of average amplitude of an audio signal over time can be used to modulate a variable continuously through a technique called *envelope following*. Similarly, a threshold of amplitude can be set to trigger an event when the sound reaches a certain level; this technique of *attack detection* ("attack" is a common term for the onset of a sound) can be used, for example, to create a visual action synchronized with percussive sounds coming into the computer.

The technique of pitch tracking, which uses a variety of analysis techniques to attempt to discern the fundamental frequency of an input sound that is reasonably harmonic, is often used in interactive computer music to track a musician in real time, comparing her/his notes against a "score" in the computer's memory. This technology of score-following can be used to sequence interactive events in a computer program without having to rely on absolute timing information, allowing musicians to deviate from a strict tempo, improvise, or otherwise inject a more fluid musicianship into a performance.

A wide variety of timbral analysis tools also exist to transform an audio signal into data that can be mapped to computer-mediated interactive events. Simple algorithms such as *zero-crossing counters*, which tabulate the number of times a time-domain audio signal crosses from positive to negative polarity, can be used to derive the amount of noise in an audio signal. Fourier analysis can also be used to find, for example, the five loudest frequency components in a sound, allowing the sound to be examined for harmonicity or timbral brightness. Filter banks and envelope followers can be combined to split a sound into overlapping frequency ranges that can then be used to drive another process. This technique is used in a common piece of effects hardware called the *vocoder*, in which a harmonic signal (such as a synthesizer) has different frequency ranges boosted or attenuated by a noisy signal (usually speech). The effect is that of one sound "talking" through another sound; it is among a family of techniques called cross-synthesis.

Music as information

Digital representations of music, as opposed to sound, vary widely in scope and character. By far the most common system for representing real-time musical performance data is the Musical Instrument Digital Interface (MIDI) specification,

released in 1983 by a consortium of synthesizer manufacturers to encourage interoperability between different brands of digital music equipment. Based on a unidirectional, low-speed serial specification, MIDI represents different categories of musical *events* (notes, continuous changes, tempo and synchronization information) as abstract numerical values, nearly always with a 7-bit (0–127) numeric resolution.

Over the years, the increasing complexity of synthesizers and computer music systems began to draw attention to the drawbacks of the simple MIDI specification. In particular, the lack of support for the fast transmission of digital audio and high-precision, syntactic synthesizer control specifications along the same cable led to a number of alternative systems. Open Sound Control, developed by a research team at the University of California, Berkeley, makes the interesting assumption that the recording studio (or computer music studio) of the future will use standard network interfaces (Ethernet or wireless TCP/IP communication) as the medium for communication. OSC allows a client-server model of communication between controllers (keyboards, touch screens) and digital audio devices (synthesizers, effects processors, or general-purpose computers), all through UDP packets transmitted on the network.

The following code examples are written in Processing using Krister Olsson's Ess library (*www.processing.org/reference/libraries*) to facilitate sound synthesis and playback. The Ess library includes classes for audio playback in timed units (`AudioChannel`), playback as a continuous process (`AudioStream`), use of real-time input from the computer's audio hardware (`AudioInput`), and writing of audio output to disk (`AudioFile`). In addition, two classes of unit generator-style functions are available: `AudioGenerators`, which synthesize sound (e.g., `SineWave`, a class that generates a sine waveform), and `AudioFilters`, which process previously generated audio (e.g., `Reverb`, a class to apply reverberation). An `FFT` class (for audio analysis) is also provided.

Example 1, 2: Synthesizer (p. 593, 594)
These two examples show two different methodologies for synthesizing sound. The first example fills an `AudioStream` with the output of a bank of sine waves (represented as an array of `SineWave` generators). The `audioStreamWrite()` function behaves in a manner analogous to the `draw()` function in the main Processing language, in that it repeats indefinitely to generate the audio by updating the state of the different `SineWave` oscillators and writing them (through the `generate()` method) into the `AudioStream`. The frequency properties of the different `SineWave` generators are set based on the mouse position, which also determines where a snapshot of the audio waveform being generated is drawn to the canvas. The second example shows the use of an `AudioChannel` class to generate a sequence of algorithmically generated synthesized events, which are created by a `TriangleWave` generator filtered through an `Envelope` that fades in and out each "note." The notes themselves are generated entirely in the `setup()` function (i.e., the program is noninteractive), based on a sequence of frequencies provided in a `rawSequence[]` array.

Example 3: Sample playback (p. 595)
The playback of an audio sample can be achieved by instantiating an instance of the `AudioChannel` class with a filename of a sample to read in. This example uses an array of six `AudioChannel` objects, each with the same short sample of a cello (*cela3.aif*). By varying the effective `SamplingRate` of each channel, we can change the playback speed (and as a result, the pitch) of the cello sample when it is sounded (by the `play()` method to the `AudioChannel`). The example shows a simple Pong-like simulation where a sound is triggered at each end of the ball's trajectory as well as when it crosses the center of the canvas. Because the `AudioChannel` playback routine will last for different durations depending on the `SamplingRate` we randomly assign to it, we have no way of guaranteeing that a given `AudioChannel` will be finished playing when the next sound is called for. As a result, a `while()` loop in the code searches through the array of `AudioChannel` objects whenever a sound is called for, querying their state property to see if they are available to play a sound. This demonstrates a simple form of *voice allocation*, an important technique in managing polyphony in systems where we have a finite number of "voices" or sound-generating engines to draw from to make multiple sounds at the same time. An `Envelope` filter is also used to fade the `AudioChannel` in or out as it plays to prevent clicks in the sound.

Example 4: Effects processor (p. 597)
Ess (like many other computer music toolkits) allows for the processing of audio to occur *in-place*; that is, it lets us take a sound, change it in some way, and store it in the same block of memory so that we can continue to process it without having to create a duplicate copy of the audio. This allows us to use effects that rely on some degree of feedback. In this example, we take a sample of electric guitar chords (played by an `AudioChannel` class) and process it through a `Reverb` filter. We then take this (already reverberated) sound and process it again, gradually degenerating the original sound by adding more and more reverberation. A similar technique in the analog domain provides the basis for a well-known piece of the electroacoustic repertoire, Alvin Lucier's 1969 masterpiece "I Am Sitting in a Room."

Example 5: Audio analysis (p. 598)
In addition to classes that provide for the generation and manipulation of audio streams and events, Ess provides an `FFT` class to analyze an `AudioChannel` using the Fast Fourier Transform, filling an array with the spectrum of a particular sound. This allows us to look at the frequency content of the sound we're providing, which we can then use to make decisions in our program or, as in this example, visualize during the `draw()` function as a graph. The code draws two versions of a spectrogram for an `AudioChannel` containing a sound file of a sine sweep: the first (drawn in black) shows the spectrum for the current `FFT` frame (i.e., the sound as we're hearing it now); the second (in white) shows the maximum amplitude achieved in each frequency band so far, gradually decaying over time. This second graph is an example of a *peak hold*, a feature that exists on many audio analysis tools (level meters, etc.) to give an analyst a sense of how the current signal compares to what has come before. In the `draw()`

routine, we plot the FFT channels along a logarithmic space, so that the channels representing lower frequencies are farther apart than the ones representing high frequencies on the right of the canvas; this appropriately approximates our perception of frequency as pitch.

Tools for sound programming

A wide variety of tools are available to the digital artist working with sound. Sound recording, editing, mixing, and playback are typically accomplished through digital sound editors and so-called digital audio workstation (DAW) environments. Sound editors range from open source and free software (MixViews, Audacity) to professional-level two-track mastering programs (BIAS Software's Peak application, Digidesign's Sound Designer). These programs typically allow you to import and record sounds, edit them with clipboard functionality (copy, paste, etc.), and perform a variety of simple digital sound processing (DSP) tasks nondestructively on the sound file itself, such as signal normalization, fading edits, and sample-rate conversion. Often these programs will act as hosts for software plug-ins originally designed for working inside of DAW software.

Digital audio workstation suites offer a full range of multitrack recording, playback, processing, and mixing tools, allowing for the production of large-scale, highly layered projects. DAW software is now considered standard in the music recording and production industry, gradually replacing reel-to-reel tape as the medium for producing commercial recordings. The Avid/Digidesign Pro Tools software, considered the industry standard, allows for the recording and mixing of many tracks of audio in real time along a timeline roughly similar to that in a video NLE (nonlinear editing) environment . Automation curves can be drawn to specify different parameters (volume, pan) of these tracks, which contain clips of audio ("regions" or "soundbites") that can be assembled and edited nondestructively. The Pro Tools system uses hardware-accelerated DSP cards to facilitate mixing as well as to host plug-ins that allow for the high-quality processing of audio tracks in real time. Other DAW software applications, such as Apple's Logic Audio, Mark of the Unicorn's Digital Performer, Steinberg's Nuendo, and Cakewalk's Sonar, perform many of the same tasks using software-only platforms. All of these platforms also support third-party audio plug-ins written in a variety of formats, such as Apple's AudioUnits (AU), Steinberg's Virtual Studio Technology (VST), or Microsoft's DirectX format. Most DAW programs also include extensive support for MIDI, allowing the package to control and sequence external synthesizers, samplers, and drum machines; as well as software plug-in "instruments" that run inside the DAW itself as sound generators.

Classic computer music "languages," most of which are derived from Max Mathews' MUSIC program, are still in wide use today. Some of these, such as CSound (developed by Barry Vercoe at MIT) have wide followings and are taught in computer music studios as standard tools for electroacoustic composition. The majority of these MUSIC-N programs use text files for input, though they are increasingly available with graphical editors for

many tasks. Typically, two text files are used; the first contains a description of the sound to be generated using a specification language that defines one or more "instruments" made by combining simple unit generators. A second file contains the "score," a list of instructions specifying which instrument in the first file plays what event, when, for how long, and with what variable parameters. Most of these programs go beyond simple task-based synthesis and audio processing to facilitate algorithmic composition, often by building on top of a standard programming language; F. Richard Moore's CLM package, for example, is built on top of Common LISP. Some of these languages have been retrofitted in recent years to work in real time (as opposed to rendering a sound file to disk); Real-Time Cmix, for example, contains a C-style parser as well as support for connectivity from clients over network sockets and MIDI.

A number of computer music environments were begun with the premise of real-time interaction as a foundational principle of the system. The Max development environment for real-time media, first developed at IRCAM in the 1980s and currently developed by Cycling'74, is a visual programming system based on a control graph of "objects" that execute and pass messages to one another in real time. The MSP extensions to Max allow for the design of customizable synthesis and signal-processing systems, all of which run in real time. A variety of sibling languages to Max exist, including Pure Data (developed by the original author of Max, Miller Puckette) and jMax (a Java-based version of Max still maintained at IRCAM). James McCartney's SuperCollider program and Ge Wang and Perry Cook's ChucK software are both textual languages designed to execute real-time interactive sound algorithms.

Finally, standard computer languages have a variety of APIs to choose from when working with sound. Phil Burke's JSyn (Java Synthesis) provides a unit generator-based API for doing real-time sound synthesis and processing in Java. The CCRMA Synthesis ToolKit (STK) is a C++ library of routines aimed at low-level synthesizer design and, centered on physical modeling synthesis technology.

Ess, a sound library for Processing that has many features in common with the above-mentioned languages, is used in the examples for this text. Because of the overhead of doing real-time signal processing in the Java language, it will typically be more efficient to work in one of the other environments listed above if your needs require substantial real-time audio performance.

Conclusion

A wide variety of tools and techniques are available for working computationally with sound, due to the close integration of digital technology and sound creation over the last half-century. Whether your goal is to implement a complex reactive synthesis environment or simply to mix some audio recordings, software exists to help you fill your needs. Furthermore, sound-friendly visual development environments (such as Max) allow for you to create custom software from scratch. A basic understanding of the principles behind digital audio recording, manipulation, and synthesis can be indispensable in order to better translate your creative ideas into the sonic medium.

As the tools improve and the discourse of multimedia becomes more interdisciplinary, sound will become even better integrated into digital arts education and practice.

Notes

1. Douglas Kahn, *Noise, Water, Meat: A History of Sound in the Arts* (MIT Press, 2001), p. 10.
2. Paul Théberge, *Any Sound You Can Imagine: Making Music / Consuming Technology* (Wesleyan University Press, 1997), p. 105.
3. John Cage, "Credo: The Future of Music (1937)," in *John Cage: An Anthology*, edited by Richard Kostelanetz (Praeger, 1970), p. 52.
4. Joel Chadabe, *Electric Sound: The Past and Promise of Electronic Music* (Prentice Hall, 1996), p. 145.
5. Curtis Roads, *The Computer Music Tutorial* (MIT Press, 1996), p. 43.
6. Albert Bregman, *Auditory Scene Analysis* (MIT Press, 1994), p. 213.

Code

Example 1: Synthesizer

```
/**
  Sound is generated in real time by summing together harmonically related
  sine tones. Overall pitch and harmonic detuning is controlled by the mouse.
  Based on the Spooky Stream Save Ess example
*/

import krister.Ess.*;

int numSines = 5;                   // Number of oscillators to use
AudioStream myStream;               // Audio stream to write into
SineWave[] myWave;                  // Array of sines
FadeOut myFadeOut;                  // Amplitude ramp function
FadeIn myFadeIn;                    // Amplitude ramp function

void setup() {
  size(256, 200);
  Ess.start(this);                  // Start Ess
  myStream = new AudioStream();     // Create a new AudioStream
  myStream.smoothPan = true;
  myWave = new SineWave[numSines];  // Initialize the oscillators
  for (int i = 0; i < myWave.length; i++) {
    float sinVolume = (1.0 / myWave.length) / (i + 1);
    myWave[i] = new SineWave(0, sinVolume);
  }
  myFadeOut = new FadeOut();        // Create amplitude ramp
  myFadeIn = new FadeIn();          // Create amplitude ramp
  myStream.start();                 // Start audio
}

void draw() {
  noStroke();
```

```
  fill(0, 20);
  rect(0, 0, width, height);  // Draw the background
  float offset = millis() - myStream.bufferStartTime;
  int interp = int((offset / myStream.duration) * myStream.size);
  stroke(255);
  for (int i = 0; i < width; i++) {
    float y1 = mouseY;
    float y2 = y1;
    if (i+interp+1 < myStream.buffer2.length) {
      y1 -= myStream.buffer2[i+interp] * height/2;
      y2 -= myStream.buffer2[i+interp+1] * height/2;
    }
    line(i, y1, i+1, y2);  // Draw the waves
  }
}

void audioStreamWrite(AudioStream s) {
  // Figure out frequencies and detune amounts from the mouse
  // using exponential scaling to approximate pitch perception
  float yoffset = (height-mouseY) / float(height);
  float frequency = pow(1000, yoffset)+150;
  float detune = float(mouseX)/width-0.5;
  myWave[0].generate(myStream);              // Generate first sine, replace Stream
  myWave[0].phase += myStream.size;          // Increment the phase
  myWave[0].phase %= myStream.sampleRate;
  for (int i = 1; i < myWave.length; i++) {  // Add remaining sines into the Stream
    myWave[i].generate(myStream, Ess.ADD);
    myWave[i].phase = myWave[0].phase;
  }
  myFadeOut.filter(myStream);                // Fade down the audio
  for (int i = 0; i < myWave.length; i++) {  // Set the frequencies
    myWave[i].frequency = round(frequency * (i+1 + i*detune));
    myWave[i].phase = 0;
  }
  myFadeIn.filter(myStream);                 // Fade up the audio
}
```

Example 2: Synthesizer

```
/**
  Sound is generated at setup with a triangle waveform and a simple envelope
  generator. Insert your own array of notes as 'rawSequence' and let it roll.
*/

import krister.Ess.*;

AudioChannel myChannel;   // Create channel
TriangleWave myWave;      // Create triangle waveform
Envelope myEnvelope;      // Create envelope
int numNotes = 200;       // Number of notes
int noteDuration = 300;   // Duration of each note in milliseconds
float[] rawSequence = {
  293.6648, 293.6648, 329.62756, 329.62756, 391.9955, 369.99445, 293.6648, 293.6648,
```

```
  329.62756, 293.6648, 439.99997, 391.9955, 293.6648, 293.6648, 587.3294, 493.8834,
  391.9955, 369.99445, 329.62756, 523.25116, 523.25116, 493.8834, 391.9955,
  439.99997, 391.9955 };  // Happy birthday

void setup() {
  size(100, 100);
  Ess.start(this);                      // Start Ess
  myChannel = new AudioChannel();       // Create a new AudioChannel
  myChannel.initChannel(myChannel.frames(rawSequence.length * noteDuration));
  int current = 0;
  myWave = new TriangleWave(480, 0.3);  // Create triangle wave
  EPoint[] myEnv = new EPoint[3];       // Three-step breakpoint function
  myEnv[0] = new EPoint(0, 0);          // Start at 0
  myEnv[1] = new EPoint(0.25, 1);       // Attack
  myEnv[2] = new EPoint(2, 0);          // Release
  myEnvelope = new Envelope(myEnv);     // Bind Envelope to the breakpoint function
  int time = 0;
  for (int i = 0; i < rawSequence.length; i++) {
    myWave.frequency = rawSequence[current];    // Update waveform frequency
    int begin = myChannel.frames(time);         // Starting position within Channel
    int e = int(noteDuration*0.8);
    int end = myChannel.frames(e);              // Ending position with Channel
    myWave.generate(myChannel, begin, end);     // Render triangle wave
    myEnvelope.filter(myChannel, begin, end);   // Apply envelope
    current++;                                  // Move to next note
    time += noteDuration;                       // Increment the Channel output point
  }
  myChannel.play();                             // Play the sound!
}

void draw() { }  // Empty draw() keeps the program running

public void stop() {
  Ess.stop();    // When program stops, stop Ess too
  super.stop();
}
```

Example 3: Sample playback

```
/**
  Loads a sound file off disk and plays it in multiple voices at multiple sampling
  increments (demonstrating voice allocation), panning it back and forth between
  the speakers. Based on Ping Pong by Krister Olsson <http://tree-axis.com>
*/

import krister.Ess.*;

AudioChannel[] mySound = new AudioChannel[6];  // Six channels of audio playback
Envelope myEnvelope;                           // Create Envelope

boolean left = true;
boolean middle = false;
boolean right = false;
```

```
// Sampling rates to choose from
int[] rates = { 44100, 22050, 29433, 49500, 11025, 37083 };

void setup() {
  size(256,200);
  stroke(255);
  Ess.start(this);  // Start Ess
  // Load sounds and set initial panning
  // Sounds must be located in the sketch's "data" folder
  for (int i = 0; i < 6; i++) {
    mySound[i] = new AudioChannel("cela3.aif");
    mySound[i].smoothPan=true;
    mySound[i].pan(Ess.LEFT);
    mySound[i].panTo(1,4000);
  }
  EPoint[] myEnv = new EPoint[3];     // Three-step breakpoint function
  myEnv[0] = new EPoint(0, 0);        // Start at 0
  myEnv[1] = new EPoint(0.25, 1);     // Attack
  myEnv[2] = new EPoint(2, 0);        // Release
  myEnvelope = new Envelope(myEnv);   // Bind an Envelope to the breakpoint function
}

void draw() {
  int playSound = 0;  // How many sounds do we play on this frame?
  int which = -1;     // If so, on which voice?

  noStroke();
  fill(0, 15);
  rect(0, 0, width, height);          // Fade background
  stroke(102);
  line(width/2, 0, width/2, height);  // Center line
  float interp = lerp(0, width, (mySound[0].pan+1) / 2.0 );
  stroke(255);
  line(interp, 0, interp, height);    // Moving line

  // Trigger 1-3 samples when the line passes the center line or hits an edge
  if ((mySound[0].pan < 0) && (middle == true)) {
    playSound = int(random(1,3));
    middle = false;
  } else if ((mySound[0].pan > 0) && (middle == false)) {
    playSound = int(random(1,3));
    middle = true;
  } else if ((mySound[0].pan < -0.9) && (left == true)) {
    playSound = int(random(1,3));
    left = false;
  } else if ((mySound[0].pan > -0.9) && (left == false)) {
    left = true;
  } else if ((mySound[0].pan > 0.9) && (right == true)) {
    playSound = int(random(1,3));
    right = false;
  } else if ((mySound[0].pan < 0.9) && (right == false)) {
    right = true;
  }
```

```
  // Voice allocation block, figure out which AudioChannels are free
  while (playSound > 0) {
    for (int i = 0; i < mySound.length; i++) {
      if (mySound[i].state == Ess.STOPPED) {
        which = i;  // Find a free voice
      }
    }
    // If a voice is available and selected, play it
    if (which != -1) {
      mySound[which].sampleRate(rates[int(random(0,6))], false);
      mySound[which].play();
      myEnvelope.filter(mySound[which]);  // Apply envelope
    }
    playSound--;
  }

}

public void stop() {
  Ess.stop();  // When program stops, stop Ess too
  super.stop();
}

void audioOutputPan(AudioOutput c) {
  c.panTo(-c.pan, 4000);  // Reverse pan direction
}
```

Example 4: Effects processor

```
/**
  Applies reverb 10 times to a succession of guitar chords.
  Inspired by Alvin Lucier's "I am Sitting in a Room."
  Based on Reverb by Krister Olsson <http://www.tree-axis.com>
*/

import krister.Ess.*;

AudioChannel myChannel;
Reverb myReverb;
Normalize myNormalize;

int numRepeats = 9;
int repeats = 0;
float rectWidth;

void setup() {
  size(256, 200);
  noStroke();
  background(0);
  rectWidth = width / (numRepeats + 1.0);
  Ess.start(this);  // Start Ess
  // Load audio file into a AudioChannel, file must be in the sketch's "data" folder
  myChannel = new AudioChannel("guitar.aif");
```

```
  myReverb = new Reverb();
  myNormalize = new Normalize();
  myNormalize.filter(myChannel);  // Normalize the audio
  myChannel.play(1);
}

void draw() {
  if (repeats < numRepeats) {
    if (myChannel.state == Ess.STOPPED) {  // If the audio isn't playing
      myChannel.adjustChannel(myChannel.size/16, Ess.END);
      myChannel.out(myChannel.size);
      // Apply reverberation "in place" to the audio in the channel
      myReverb.filter(myChannel);
      // Normalize the signal
      myNormalize.filter(myChannel);
      myChannel.play(1);
      repeats++;
    }
  } else {
    exit();  // Quit the program
  }
  // Draw rectangle to show the current repeat (1 of 9)
  rect(rectWidth * repeats, 0, rectWidth-1, height);
}

public void stop() {
  Ess.stop();  // When program stops, stop Ess too
  super.stop();
}
```

Example 5: Audio analysis

```
/**
  Analyzes a sound file using a Fast Fourier Transform, and plots both the current
  spectral frame and a "peak-hold" plot of the maximum over time using logarithmic
  scaling. Based on examples by Krister Olsson <http://tree-axis.com>
*/

import krister.Ess.*;

AudioChannel myChannel;
FFT myFFT;
int bands = 256;  // Number of FFT frequency bands to calculate

void setup() {
  size(1024, 200);

  Ess.start(this);  // Start Ess
  // Load "test.aif" into a new AudioChannel, file must be in the "data" folder
  myChannel = new AudioChannel("test.aif");
  myChannel.play(Ess.FOREVER);
  myFFT = new FFT(bands * 2);  // We want 256 frequency bands, so we pass in 512
}
```

```
void draw() {
  background(176);
  // Get spectrum
  myFFT.getSpectrum(myChannel);
  // Draw FFT data
  stroke(255);
  for (int i = 0; i < bands; i++) {
    float x = width - pow(1024, (255.0-i)/bands);
    float maxY = max(0, myFFT.maxSpectrum[i] * height*2);
    float freY = max(0, myFFT.spectrum[i] * height*2);
    // Draw maximum lines
    stroke(255);
    line(x, height, x, height-maxY);
    // Draw frequency lines
    stroke(0);
    line(x, height, x, height-freY);
  }
}

public void stop() {
  Ess.stop();  // When program stops, stop Ess too
  super.stop();
}
```

Resources

Sound toolkits and resources

Vercoe, Barry, et al. CSound. Synthesis and signal processing language, 1984. http://www.csounds.com.

Garton, Brad, David Topper, et al. Real-Time Cmix. Synthesis and signal processing language, 1995. http://rtcmix.org.

Wang, Ge, and Perry Cook. ChucK. Real-time audio programming language, 2002. http://chuck.cs.princeton.edu.

McCartney, James, et al. SuperCollider. Real-time audio programming language, 1996. http://www.audiosynth.com.

Puckette, Miller, David Zicarelli, et al. Max/MSP. Graphical development environment for music and multimedia, 1986. http://www.cycling74.com.

Puckette, Miller, et al. Pure Data (Pd). Graphical development environment for music and multimedia, 1996. http://www-crca.ucsd.edu/~msp/software.html.

Burke, Phil. JSyn. Java API for real-time audio, 1997. http://www.softsynth.com/jsyn.

Cook, Perry, and Gary Scavone. STK. C++ synthesis toolkit, 1996. http://ccrma.stanford.edu/software/stk.

Lopez-Lezcano, Fernando, maintainer. Planet CCRMA. Collection of open source audio software for Linux, 2005. http://ccrma.stanford.edu/planetccrma/software.

Klingbeil, Michael. SPEAR. Spectral editor, 2004. http://www.klingbeil.com/spear.

Waveform Software. Sox, PVCX, AmberX. Freeware sound conversion / spectral processing / granular synthesis software, 2005. http://www.waveformsoftware.com.

Audacity. Open source audio waveform editor, 2002. http://audacity.sourceforge.net.

Texts

Bregman, Albert. *Auditory Scene Analysis*. MIT Press, 1994.

Chadabe, Joel. *Electric Sound: The Past and Promise of Electronic Music*. Prentice Hall, 1996.
Garnett, Guy E. "The Aesthetics of Interactive Computer Music." *Computer Music Journal* 25:1, (2001).
Kahn, Douglas. *Noise, Water, Meat: A History of Sound in the Arts*. MIT Press, 2001.
Lysloff, Rene, and Leslie Gay, eds. *Music and Technoculture*. Wesleyan University Press, 2003.
Maurer, John. "A Brief History of Algorithmic Composition." Stanford University. http://ccrma-www.stanford.edu/~blackrse/algorithm.html.
Paradiso, Joseph A. "American Innovations in Electronic Musical Instruments." *New Music Box* 6. http://www.newmusicbox.org/third-person/oct99.
Prendergast, Mark. *The Ambient Century: From Mahler to Moby—The Evolution of Sound in the Electronic Age*. Bloomsbury, 2003.
Puckette, Miller. *Theory and Techniques of Electronic Music*. University of California, San Diego, 2006. http://crca.ucsd.edu/~msp/techniques.htm.
Rowe, Robert. *Interactive Music Systems*. MIT Press, 1993.
Rowe, Robert. *Machine Musicianship*. MIT Press, 2001.
Theberge, Paul. *Any Sound You Can Imagine: Making Music / Consuming Technology*. Wesleyan University Press, 1997.
Supper, Martin. "A Few Remarks on Algorithmic Composition." *Computer Music Journal* 25:1 (2001).
Winkler, Todd. *Composing Interactive Music: Techniques and Ideas Using Max*. MIT Press, 1998.
Roads, Curtis. *The Computer Music Tutorial*. MIT Press, 1996.
Roads, Curtis. *Microsound*. MIT Press, 2002.

Artists

Aphex Twin (Richard James). http://www.drukqs.net.
Bailey, Chris. http://www.music.columbia.edu/~chris.
Cope, David. http://arts.ucsc.edu/faculty/cope.
DuBois, R. Luke. http://lukedubois.com.
Eno, Brian. http://www.enoweb.co.uk.
Garton, Brad. http://music.columbia.edu/~brad.
Inge, Leif. http://www.notam02.no/9.
Interface (Dan Trueman, Curtis Bahn, Tomie Hahn). http://www.arts.rpi.edu/crb/interface.
Lewis, George. http://kalvos.org/lewisge.html.
Kimura, Mari. http://homepages.nyu.edu/~mk4.
Oliveros, Pauline. http://www.deeplistening.org/pauline.
Oswald, John. http://www.plunderphonics.com.
Risset, Jean-Claude. http://www.cdemusic.org/artists/risset.html.
Schumacher, Michael J. http://www.diapasongallery.org/mjs.page.html.
Sonami, Laetitia. http://www.sonami.net.
Stone, Carl. http://www.sukothai.com.
Truax, Barry. http://www.sfu.ca/~truax.
Vitiello, Stephen. http://www.stephenvitiello.com.
Xenakis, Iannis. http://www.iannis-xenakis.org.