3 Introduction to Digital Sound Synthesis

with John Strawn

Background: History of Digital Sound Synthesis

Music I and Music II
The Unit Generator Concept

Fixed-waveform Table-lookup Synthesis

Changing the Frequency
Algorithm for a Digital Oscillator

Table-lookup Noise and Interpolating Oscillators

Time-varying Waveform Synthesis

Envelopes, Unit Generators, and Patches
Graphic Notation for Synthesis Instruments
Using Envelopes in Patches

Software Synthesis

Instrument Editors and Synthesis Languages
Computational Demands of Synthesis
Non-real-time Synthesis
Sound Files

Real-time Digital Synthesis

Comparing Non-real-time Synthesis with Real-time Synthesis

Specifying Musical Sounds

Sound Objects
Example of the Specification Problem for Additive Synthesis
The Musician's Interface
Musical Input Devices
Performance Software
Editors
Languages
Algorithmic Composition Programs

Conclusion

This chapter outlines the fundamental methods of digital sound production. Following a brief historical overview, we present the theory of table-lookup synthesis—the core of most synthesis algorithms. We next present strategies for synthesizing sounds that vary over time. This is followed by a practical comparison between "software synthesis" and "hardware synthesis," that is, between computer programs and dedicated synthesizers. Finally, we survey the various means of specifying musical sounds to a computer or synthesizer. The only prerequisite to this chapter is a knowledge of basic concepts of digital audio as explained in chapter 1.

Background: History of Digital Sound Synthesis

The first experiments in synthesis of sound by computer began in 1957 by researchers at Bell Telephone Laboratories in Murray Hill, New Jersey (David, Mathews, and McDonald 1958; Roads 1980; Wood 1991). In the earliest experiments, Max V. Mathews (figure 3.1) and his colleagues proved that a computer could synthesize sounds according to any pitch scale or waveform, including time-varying frequency and amplitude envelopes.

Their first programs were written directly in terms of machine instructions for a giant IBM 704 computer fabricated with vacuum tube circuits (figure 3.2). The 704 was a powerful machine for its day, with a 36-bit wordlength and a built-in floating-point unit for fast numerical operations. It could be loaded with up to 32 Kwords of magnetic core memory. Computers were so rare at that time that the synthesis calculations had to be carried out at IBM World Headquarters in New York City, because Bell Telephone Laboratories did not have a suitable machine. After traveling to Manhattan to compute a sound, Mathews and his associates would return to Bell Telephone Laboratories with a digital magnetic tape. There, a less powerful computer with an attached 12-bit vacuum tube "digital-to-sound converter" transformed the samples on the tape into audible form. This converter, designed by Bernard Gordon, was at that time the only one in the world capable of sound production (Roads 1980).

Music I and Music II

The Music I program developed by Mathews generated a single waveform: an equilateral triangle. A patient user could specify notes only in terms of pitch, waveform, and duration (Roads 1980). The psychologist Newman Guttman made one composition with Music I, a monophonic etude called

Figure 3.1 Max V. Mathews, 1981. (Photograph courtesy of AT&T Bell Laboratories.)

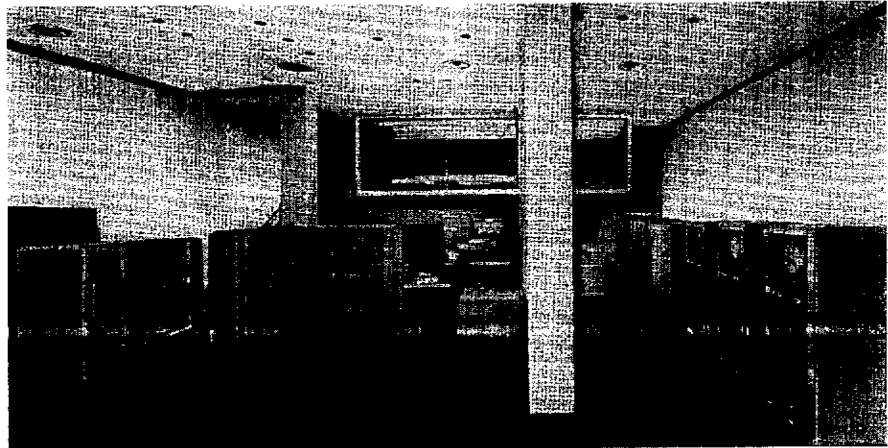

Figure 3.2 IBM 704 computer, 1957. (Photograph courtesy International Business Machines.)

In a Silver Scale written on 17 May 1957 (Guttman 1980). This was the first composition synthesized by the process of digital-to-analog conversion. Even in this first piece, the potential of the computer to generate any frequency precisely was recognized. Guttman was interested in psychoacoustics and used the piece as a test of the contrast between an "equal-beating chromatic scale" described by Silver (1957) and just intonation.

Max Mathews completed Music II in 1958; it was written in assembly language for the IBM 7094 computer, a transistorized and improved computer along the lines of the IBM 704. The 7094 ran several times faster than the older vacuum tube machines. It was thus possible to implement more ambitious synthesis algorithms. Four independent voices of sound were available, with a choice of sixteen waveforms stored in memory. Music II was used by several researchers at Bell Telephone Laboratories, including Max Mathews, John Pierce, and Newman Guttman.

A concert of the new "computer music" was organized in 1958 in New York City, followed by a discussion panel moderated by John Cage. Later that year Guttman played his computer-synthesized composition *Pitch Variations* at Hermann Scherchen's villa in Gravesano, Switzerland, where Iannis Xenakis was in the audience (Guttman 1980).

The Unit Generator Concept

One of the most significant developments in the design of digital sound synthesis languages was the concept of *unit generators* (UGs). UGs are signal processing modules like oscillators, filters, and amplifiers, which can be interconnected to form synthesis *instruments* or *patches* that generate sound signals. (Later in this chapter we discuss UGs in more detail.) The first synthesis language to make use of the unit generator concept was Music III, programmed by Mathews and his colleague Joan Miller in 1960. Music III let users design their own synthesis networks out of UGs. By passing the sound signal through a series of such unit generators, a large variety of synthesis algorithms could be implemented relatively easily.

Music N *Languages*

Since the time of Music III, a family of software synthesis systems—all based on the unit generator concept—have been developed by various researchers. Music IV was a recoding of Music III in a new macro assembly language developed at Bell Laboratories called BEFAP (Tenney 1963, 1969). Music V, developed in 1968, was the culmination of Max Mathews's efforts in software synthesis (Mathews 1969). Written almost exclusively in

Fortran IV—a standard computer language—Music V was exported to several dozen universities and laboratories around the world in the early 1970s. For many musicians, including the author of this book, it served as an introduction to the art of digital sound synthesis.

Taking Music IV or Music V as a model, others have developed synthesis programs such as Music 4BF, Music 360, Music 7, Music 11, Csound, MUS10, Cmusic, Common Lisp Music, and so on. As a general category these programs are often referred to under the rubric of "Music *N*" languages (see chapter 17).

Fixed-Waveform Table-lookup Synthesis

As chapter 1 explains, digital synthesis generates a stream of numbers representing the samples of an audio waveform. We can hear these synthetic sounds only by sending the samples through a *digital-to-analog converter* (DAC), which converts the numbers to a continuously varying voltage that can be amplified and sent to a loudspeaker.

One way of viewing this process is to imagine a computer program that calculates the sample values of a waveform according to a mathematical formula, and sends those samples, one after the other, to the DAC. This process works fine, but it is not the most efficient basis for digital synthesis.

In general, musical sound waves are extremely repetitive, a fact that is reflected in the notions of frequency and pitch. Hence a more efficient technique is to have the hardware calculate the numbers for just one cycle of the waveform and store these numbers in a list stored in memory, as shown in figure 3.3. Such a list is called a *wavetable*. To generate a periodic sound, the computer simply reads through the wavetable again and again, sending the samples it reads to the DAC for conversion to sound.

This process of repeatedly scanning a wavetable in memory is called *table-lookup synthesis*. Since it typically takes only a few nanoseconds for a computer to read a value from memory, table-lookup synthesis is much quicker than calculating the value for each sample from scratch. Table-lookup synthesis is the core operation of a *digital oscillator*—a fundamental sound generator in synthesizers.

Let us now walk through the valley of table lookup. Suppose that the value of the first sample is given by the first number in the wavetable (location 1 in figure 3.3). For each new sample to be produced by this simple synthesizer, take the next sample from the wavetable. At the end of the wavetable, simply go back to the beginning and start reading out the sam-

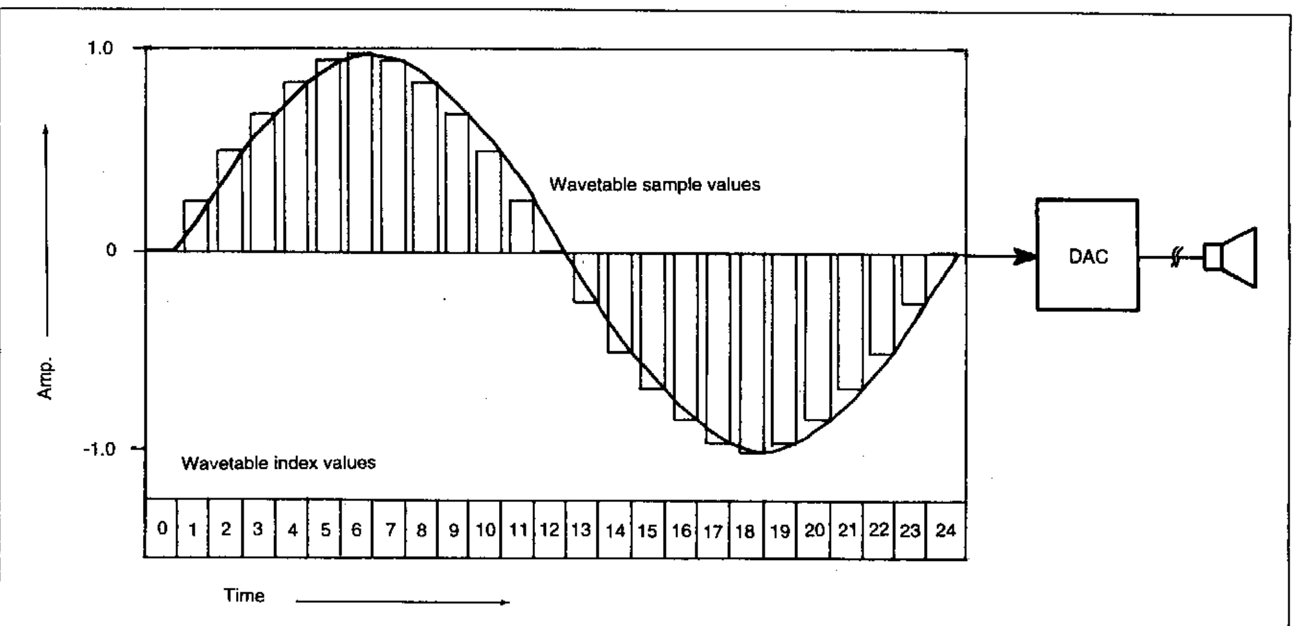

Figure 3.3 Graphical depiction of wavetable-lookup synthesis. The list 0–24 in the lower portion are numbered locations or "table index values." An audio sample value is stored in memory for each index point. The samples are depicted as the rectangles outlining a sine wave in the top portion. For example, Wavetable[0] = 0, and Wavetable[6] = 1. To synthesize the sine wave the computer looks up the sample values stored in successive index locations and sends them to a DAC, looping through the table repetitively.

ples again. The process is also called *fixed-waveform synthesis* because the waveform does not change over the course of a sound event.

For example, let us assume the table contains 1000 entries, each of which is a 16-bit number. The entries are indexed from 0 to 999. We call the current location in the table the *phase_index* value, with reference to the phase of the waveform. To read through the table the oscillator starts at the first entry in the table (*phase_index* = 0) and moves by an *increment* to the end of the table (*phase_index* = 999). At this point the phase index "wraps around" the ending point to the beginning of the wavetable and starts again.

Changing the Frequency

What is the frequency of the sound produced by table-lookup synthesis? It depends on the length of the wavetable and the sampling frequency. If the sampling frequency is 1000 samples per second, and there are 1000 numbers in the table, the result is 1000/1000 : 1 Hz. If the sampling frequency is 100,000 Hz, and the table contains 1000 entries, then the output frequency is 100 Hz, since 100,000/1000 = 100.

How is it possible to change the frequency of the output signal? As we have just seen, one simple way is to change the sampling frequency. But this strategy is limited, particularly when one wants to process or mix signals with different sampling rates. A better solution is to scan the wavetable at different rates, skipping some of the samples in it. This, in effect, shrinks the size of the wavetable in order to generate different frequencies.

For example, if we take only the even-numbered samples, then we go through the table twice as fast. This raises the pitch of the output signal by an octave. If we skip two samples, then the pitch is raised further (by an octave and a fifth, to be exact). In the table-lookup algorithm, the increment determines the number of samples to be skipped. The increment is added to the current phase location in order to find the next location for reading the value of the sample. In the simplest example, where we read every sample from the table, the increment is 1. If we read only the odd- or even-numbered samples in the table, then the increment is 2.

Algorithm for a Digital Oscillator

We could say that the oscillator *resamples* the wavetable in order to generate different frequencies. That is, it skips values in the table by an increment added to the current phase location in the wavetable. Thus the most basic oscillator algorithm can be explained as a two-step program:

1. $phase_index = \mathrm{mod}_L(previous_phase + increment)$
2. $output = amplitude \times wavetable[phase_index]$

Step (1) of the algorithm contains an add and a modulo operation (denoted mod_L). The modulo operation divides the sum by the table length L and keeps only the remainder, which is always less than or equal to L. Step (2) contains a table lookup and a multiply. This is relatively little computation, but it assumes that the wavetables are already filled with waveform values.

If the table length and the sampling frequency are fixed—as is usually the case—then the frequency of the sound emitted by the oscillator depends on the value of the increment. The relationship between a given frequency and an increment is given by the following equation, which is the most important equation in table-lookup synthesis:

$$increment = \frac{L \times frequency}{samplingFrequency}. \tag{1}$$

For example, if tablelength L is 1000 and sampling frequency is 40,000, while the specified *frequency* of the oscillator is 2000 Hz, then the *increment* is 50.

This implies the following equation for frequency:

$$frequency = \frac{increment \times samplingFrequency}{L}. \tag{2}$$

So much for the mathematical theory of digital oscillators. Now we confront the computational realities.

Table-lookup Noise and Interpolating Oscillators

All the variables in the previous example were multiples of 1000, which led to a neat integer result for the value of the phase index increment. However, for most values of the table length, frequency, and sampling frequency in equation 1, the resulting increment is not an integer, but rather a real number with a fractional part after the decimal point. However, the way we look up a value in the wavetable is to locate it by its index, which is an integer. Thus we need to somehow derive an integer value from the real-valued increment.

The real value can be *truncated* to yield an integer value for the table index. This means to delete the part of the number to the right of the decimal point, so a number like 6.99 becomes 6 when it is truncated.

Table 3.1 Phase index values in an oscillator wavetable, calculated and truncated

Phase index	
Calculated	Truncated
1.000	1
2.125	2
3.250	3
4.375	4
5.500	5
6.625	6
7.750	7
8.875	8
10.000	10
11.125	11
12.250	12
13.375	13
14.500	14
15.625	15
16.750	16
17.875	17
19.000	19

Suppose that we use an increment of 1.125. Table 3.1 compares the calculated versus the truncated increments. The imprecision caused by the truncation means that we obtain a waveform value near to, but not precisely the same as, the one we actually need. As a result, small amounts of waveform distortion are introduced, called *table-lookup noise* (Moore 1977; Snell 1977b). Various remedies can reduce this noise. A larger wavetable is one prescription, since a fine-grain table reduces lookup error. Another way is to *round* the value of increment up or down to the nearest integer instead of simply truncating it, in this case, an increment of 6.99 becomes 7, which is more accurate than 6. But the best performance is achieved by an *interpolating oscillator*. This is more costly from a computational standpoint, but it generates very clean signals.

An interpolating oscillator calculates what the value of the wavetable would have been, if it were possible to reference the wavetable at the exact phase specified by the increment. In other words, it interpolates between the entries in the wavetable to find the one that exactly corresponds to the specified phase index increment (figure 3.4).

With interpolating oscillators, smaller wavetables can yield the same audio quality as a larger noninterpolating oscillator. Consider that for a 1024-entry wavetable used by an interpolating oscillator, the signal-to-noise ratio for a sine wave is an excellent 109 dB (worst-case), as compared with the

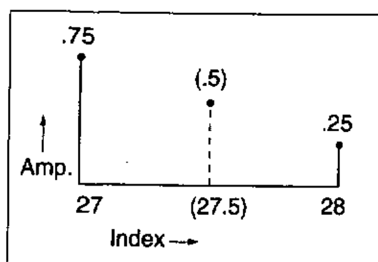

Figure 3.4 Action of an interpolating oscillator. The graph shows two x-points in a wavetable, at positions 27 and 28. The oscillator phase increment indicates that the value should be read from location 27.5, for which there is no entry, so the interpolating oscillator calculates a y-value in between the values for 27 and 28.

abysmal 48 dB for a noninterpolating oscillator using the same size wavetable (Moore 1977). These figures pertain to the case of linear interpolation; even better results are possible with more elaborate interpolation schemes (Chamberlin 1985; Crochiere and Rabiner 1983; Moore 1977; Snell 1977b).

This concludes our introduction to fixed-waveform table-lookup synthesis. The next section shows how aspects of synthesis can be varied over time.

Time-varying Waveform Synthesis

So far we have seen how to produce a sine wave at a fixed frequency: well and good. Since the maximum value of the sine wave does not change in time, the signal has a constant loudness. This is not terribly useful for musical purposes, since one can only control pitch and duration, leaving no control over other sound parameters. Even if the oscillator reads from other wavetables, they repeat ad infinitum. The key to more interesting sounds is *time-varying* waveforms, achieved by changing one or more synthesis parameters over the duration of a sound event.

Envelopes, Unit Generators, and Patches

To create a time-varying waveform, we need a synthesis *instrument* that can be controlled by *envelopes*—functions of time. For example, if the amplitude of the sound changes over its duration, the curve that the amplitude follows is called the *amplitude envelope.* A general way of designing a synthesis instrument is to imagine it as a *modular system,* containing a number of specialized signal-processing units that together create a time-varying sound.

The unit generator is a fundamental concept in digital synthesis. A UG is either a signal generator or a signal modifier. A signal generator (such as an oscillator) synthesizes signals such as musical waveforms and envelopes. A signal modifier, such as a filter, takes a signal as its input, and it transforms that input signal in some way.

To construct an instrument for sound synthesis, the composer connects together UGs into a *patch*. The term "patch" derives from the old modular analog synthesizers in which sound modules were connected via *patch cords*. Of course, when a program is making music, the connections are all done by the software; no physical wires or cables are connected. But if a UG produces a number at its output, that number can become the input to another UG.

Graphic Notation for Synthesis Instruments

Now we introduce the graphic notation that is often used in publications on digital sound synthesis to illustrate patches. This notation was invented to explain the operation of the first modular languages for digital sound synthesis, such as Music 4BF (Howe 1975) and Music V (Mathews 1969), and is still useful today.

The symbol for each unit generator has a unique shape. Figure 3.5 shows the graphic notation for a *table-lookup oscillator* called **osc**, a basic signal generator. It accepts three inputs (amplitude, frequency, waveform) and produces one output (a signal). The oscillator reads from a single wavetable that remains unchanged as long as the oscillator plays. (More complicated oscillators can read through several wavetables over the course of an event; see chapter 5 on *multiple wavetable synthesis*.)

In figure 3.5 the top right input to the oscillator is frequency. The top left input determines the peak amplitude of the signal generated by the oscillator. The box to the left is the wavetable *f1* containing a sine wave. (Note: In

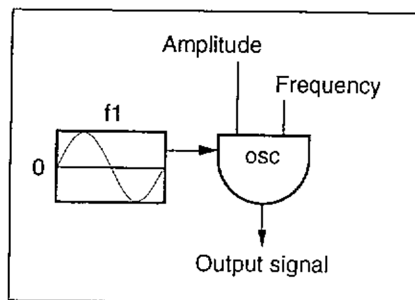

Figure 3.5 Graphical notation for an oscillator. See the text for explanation.

some implementations, instead of frequency, the value fed directly to the oscillator is a raw phase increment. Since phase increment is not a musically intuitive parameter, we assume here that the system automatically takes care of conversions from frequency to phase increment according to equation 1.)

Using Envelopes in Patches

If we supply a constant number (say, 1.0) to the amplitude input of an oscillator, then the overall amplitude of the output waveform is constant over the duration of each event. By contrast, most interesting sounds have an amplitude envelope that varies as a function of time. Typically, a note starts with an amplitude of 0, works its way up to some maximum value (usually *normalized* to be no greater than 1.0), and dies down again more or less slowly to 0. (A normalized wave is one that has been scaled to fall within standard boundaries such as 0 to 1 for amplitude envelopes, or -1 to $+1$ for other waves.) The beginning part of the envelope is called the *attack* portion, while the end of the envelope is called the *release*.

Commercial analog synthesizers used to define amplitude envelopes in four stages: *attack*, (initial) *decay*, *sustain* (a period that depends, for example, on how long a key on a keyboard is depressed), and *release*. The usual acronym for such a four-stage envelope is ADSR (figure 3.6). The ADSR concept is useful for describing verbally the overall shape of an envelope, for example, "Make the attack sharper." But for specifying a musical envelope, a four-stage limit is anachronistic. Amplitude shaping is a delicate operation, so more flexible envelope editors allow musicians to trace arbitrary curves (see chapter 16).

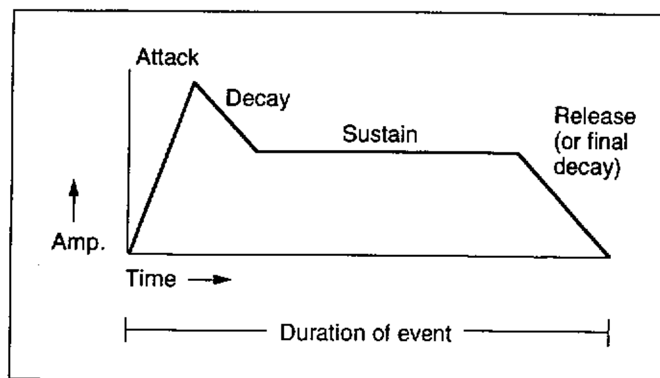

Figure 3.6 Graph of a simple ADSR amplitude envelope, showing the way the amplitude of a note changes over its duration.

The instrument of figure 3.5 can be easily adapted to generate a time-varying amplitude by hooking up an envelope to the amplitude input of the oscillator. We are now closer to controlling the oscillator in musical terms. If we set the duration and the curve of the envelope, then the envelope controls the amplitude of each note.

To design manually an envelope for each and every event in a composition is too tedious. What we seek is a simple procedure for generating an envelope that can scale itself to the duration of diverse events. One solution is to take another table-lookup oscillator (labeled **env_osc** in figure 3.7, but this time fill its wavetable *f1* with values of the amplitude envelope between 0 and 1 instead of a sine wave. Rather than finding the increment from the frequency, the *envelope oscillator* derives the increment from the duration of the note. If the duration of the note is 2 seconds, for example, the "frequency" of envelope oscillator is 1 cycle per 2 seconds, or 0.5 Hz. Thus, the **env_osc** reads through the amplitude table just once over this period. For each sample, **env_osc** produces at its output a value derived from the stored envelope *f1*. This value becomes the left-hand (amplitude) input for the sine wave oscillator, **osc**. After **osc** has looked up a sample in its wavetable *f2*, the value of the sample is scaled inside **osc** by whatever appears at its amplitude input, which in this case comes from **env_osc**.

Figure 3.7a is a typical instrument as defined in a synthesis language of the type described in chapter 17. Figure 3.7b shows another way to characterize the same structure, which is perhaps more common in synthesizers. This figure replaces the envelope oscillator with the simple *envelope generator* **env_gen**. The **env_gen** takes in a duration, peak amplitude, and a wavetable; it reads through the wavetable over the specifed duration, scaling it by the specified peak amplitude.

As the reader might guess, we could also attach an envelope generator to the frequency input of **osc** to obtain a pitch change such as vibrato or glissando. Indeed, we can interconnect oscillators and other unit generators in a wide variety of ways in order to make different sounds. Interconnected oscillators are the basis of many of the synthesis techniques described in chapters 4 through 8.

Software Synthesis

So far we have discussed digital synthesis in abstract terms. The next sections describe synthesis systems in more practical terms. The most precise and flexible approach to digital sound generation is a *software synthesis*

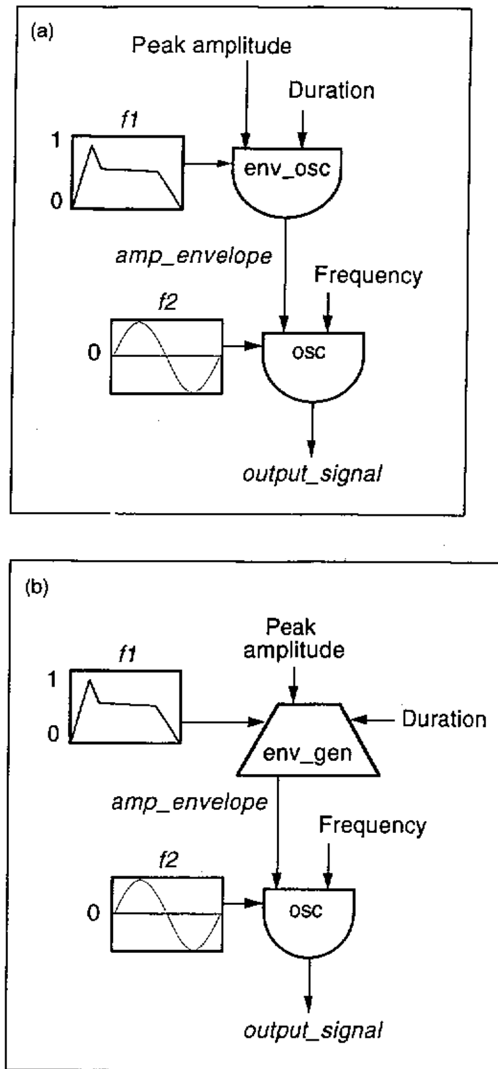

Figure 3.7 Time-varying amplitude control of oscillator. (*a*) Oscillator as envelope generator. The upper oscillator **env_osc** is employed as an envelope generator to control the amplitude of the sine wave generated by the lower oscillator **osc**. **env_osc** assumes that it will complete one cycle. This structure is found in synthesis languages. (*b*) An equivalent structure to (*a*) using a simple envelope generator unit **env_gen**. This unit takes in duration, peak amplitude, and waveform. This structure is more typical of synthesizers.

program running on a general-purpose computer. Software synthesis means that all of the calculations involved in computing a stream of samples are carried out by a program that can be changed in arbitrary ways by the user. A canonical example of software synthesis is the Music V language (Mathews 1969) or its many Music *N* variants.

Software synthesis stands in contrast to *hardware synthesis*, which carries out synthesis calculations using special circuitry. Hardware synthesis has the advantage of high-speed real-time operation, but the flexibility and size of the synthesis algorithm are limited by the fixed design of the hardware. A typical example is a fixed-function commercial keyboard synthesizer. Its internal circuits cannot necessarily be reconfigured to perform a technique developed by a rival manufacturer, for example.

The distinction between software and hardware synthesis blurs in some cases. Consider the case of a system built around a programmable *digital signal processor* (DSP) with a large memory. It may be possible for such a system to run the same type of synthesis software as a general-purpose computer. (See chapter 20 for more on the architecture of DSPs.)

In any case, all of the pioneering work in computer music was carried out via software synthesis. Today a variety of synthesis programs run on inexpensive personal computers. Good-quality ADCs and DACs are either built in or readily available as accessories. A great advantage of software synthesis is that a small computer can realize any synthesis method—even the most computationally intensive—provided that the musician has the patience to wait for results. Thus, with little else needed but musical will, computers are primed and ready for high-quality music synthesis.

Instrument Editors and Synthesis Languages

Contemporary software synthesis programs can be divided into two categories: (1) *graphical instrument editors* and (2) *synthesis languages*. With a graphical instrument editor, the musician interconnects icons on the display screen of a computer, making *patches*. Each icon stands for a UG. (Chapter 16 presents this subject and gives examples.)

With a language, the musician specifies sounds by writing a text that is interpreted by a synthesis program. Figure 3.8a shows a textual representation of the same instrument shown in figure 3.7a. The example uses a simple hypothetical synthesis language that we call Music 0. The symbol ← means "is assigned to the value of." For example, the output of **env_osc** is assigned (routed) to the signal variable *amp_envelope*. Then the value of *amp_envelope*, at each sample period, is fed into the amplitude input of the **osc** module.

(a)

```
Instrument 1
     /* env_osc arguments are wavetable, duration, amplitude */
     amp_envelope ← env_osc  f1  p3  1.0;
     /* osc arguments are wavetable, frequency, amplitude */
     output_signal ← osc  f2  p4  amp_envelope;
     out output_signal;
EndInstrument 1;
```

(b)

```
/*  Score line for Instrument 1        */
/*  p1        p2     p3        p4      */
    i1        0      1.0       440
```

Figure 3.8 Textual representation of the instrument and score. (*a*) Instrument corresponding to figure 3.7. The remarks between the characters "/*" and "*/" are comments. The *parameter fields* (beginning with "p") indicate values that will be derived from an alphanumeric score, as in (*b*). p3 specifies duration, and p4 is frequency. Notice that the third argument to the second oscillator (the amplitude) is supplied by the *amp_envelope* signal generated by the first oscillator. (*b*) Score for instrument in (*a*). The first field is the instrument number. The second parameter field indicates the start time, the third duration, and the fourth frequency.

Figure 3.8b presents a simple score that supplies parameters to this instrument. (Chapter 17 explains the basic syntax and features of synthesis languages.)

Computational Demands of Synthesis

Every step in a synthesis algorithm takes a certain amount of time to execute. For a complicated synthesis algorithm, a computer cannot always complete the calculations necessary for a sample in the interval of a sample period.

To make this point more concrete, see the steps below that are necessary for calculating one sample of sound by the table-lookup method.

1. Add increment to current wavetable lookup location to obtain new location.
2. If the new location is past the end of the wavetable, subtract the wavetable length. (In other words, perform a modulo operation.)
3. Store the new location for use in calculating the next sample. (See step 1.)
4. Look up the value in the wavetable at the new location.
5. Multiply that value by the amplitude input.
6. Send the product to the output.

The important point here is that each step takes some amount of time to perform. For example, it might take a computer one microsecond to perform the calculations above. But if we are using a sampling rate of 50,000 samples per second, the time available per sample is only 1/50,000th of a second, or 20 microseconds (20,000 nanoseconds). This means that it is difficult for the computer to complete the calculations necessary for more than a few simple oscillators in *real time*. If the process is made more complicated, by adding filters, delays, more table lookups, random functions, or the time needed to interact with a musician, even one instrument may become impossible to realize in real time. What do we mean by real time? In this context, real time means that we can complete the calculations for a sample within the duration of one sample period.

Non-Real-time Synthesis

Certain synthesis and signal-processing techniques are costly from a computational standpoint and are therefore inherently difficult to realize in real

time. This means there is a delay of at least a few seconds between the time we start computing a sound and the time that we can listen to it. A system with such a delay is called a *non-real-time* system.

Non-real-time synthesis was the only option in the early days of computer music. For example, a two-minute portion of J. K. Randall's *Lyric Variations for Violin and Computer*, realized between 1965 and 1968 at Princeton University (Cardinal Records VCS 10057), took nine hours to compute. Of course, if a small mistake was made, the entire process would have to be repeated. Even though this was a laborious process, a handful of dedicated composers with access to the proper facilities were able to create lengthy computer-synthesized works of music (see also Tenney 1969; Von Foerster and Beauchamp 1969; Dodge 1985; Risset 1985a).

Sound Files

Because it may longer than one sample period to compute each sample, software synthesis programs generate a *sound file* as their output. A sound file is simply a data file stored on a disk or tape. After all the samples for a composition are calculated, then the sound file can be played through the DAC to be heard.

A sound file contains a *header text* and numbers representing sound samples. The header contains the name of the file and relevant information about the samples in the file (sampling rate, number of bits per sample, number of channels, etc.). The samples are usually organized in data structures called *frames;* if there are *N* channels, each frame contains *N* samples. Thus, the sampling rate really indicates the number of frames per second.

As in other computer applications, different file formats coexist. The need to convert between formats is a practical fact of life in computer music studios.

Real-time Digital Synthesis

Just as computers have become faster, smaller, and cheaper, digital synthesis technology has also become more efficient. As early as the mid-1970s it was practical to build digital synthesizers (albeit bulky ones) that were fast enough to do all of the calculations necessary for a sample within the duration of one sample period. With advances in circuit technology, the bulky synthesizers of the past have been replaced by tiny *integrated circuits* (ICs or *chips*) that can realize multivoice synthesis algorithms in real time.

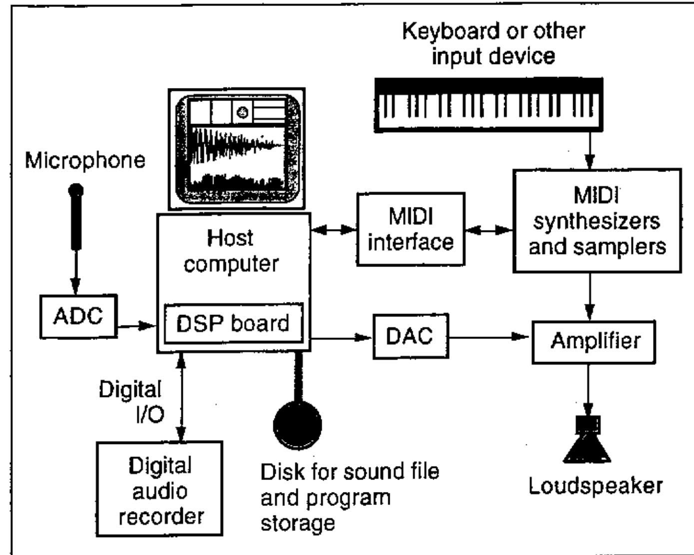

Figure 3.9 Simplified overview of a typical digital recording and synthesis facility. Musicians communicate with the synthesizers using keyboards or other input devices, or through programs running on the host computers. Sound can be recorded via the ADC and stored on disk for later playback through the DACs. In a computer equipped for multimedia production, all of the components except the MIDI keyboard may be built into the computer.

Figure 3.9 shows an overview of a real-time computer music synthesis system. This system actually has three ways of generating digital sound: (1) non-real-time software synthesis calculated on the computer, with sound from the DAC, (2) real-time synthesis calculated on the digital signal processing (DSP) board, with sound from the DAC, and (3) real-time synthesis using a synthesizer controlled via the Musical Instrument Digital Interface (MIDI; see chapter 21).

An obvious advantage of a real-time synthesizer is that *musical input devices* (also called *performance controllers*) such as musical keyboards, footpedals, joysticks, buttons, and knobs can be attached to it, so that the sound can be modified by the musician as it is being produced. *Sequencers* and *score editors* make it possible to record and edit these performances, and *patch editors* running on the computer can change the synthesis and signal-processing patches at any time.

Real-time systems are discussed in more detail throughout this book. In particular, part V discusses the internals of digital synthesizers and the MIDI protocol, and chapters 14 and 15 deal with performance controllers

and performance software (see also Alles 1977a; Buxton et al. 1978; Strawn 1985c; Roads and Strawn 1985; Roads 1989).

Comparing Non-real-time Synthesis with Real-time Synthesis

Non-real-time software synthesis was the original method of digital sound generation, and it still has a place in the studio. As we have stressed before, the advantage of software synthesis using a patchable music language is programmability and therefore musical flexibility. Whereas commercial real-time synthesizers often set factory-supplied limits, software synthesis is open-ended, letting users create personalized instruments or arbitrarily complex synthesis algorithms. Many new and experimental synthesis and signal-processing methods are available only in the form of non-real-time software.

Another strong advantage of software synthesis is the flexibility of a programmed score. Even with a simple synthesis instrument, control via a *score language* (discussed later) can be extremely detailed or complicated, exceeding the range of human performers or the transmission rates of MIDI equipment.

Nonetheless, the disadvantages of non-real-time software synthesis are obvious. Time is wasted waiting for samples to be computed. Sound is disconnected from real-time human gestures—we cannot shape sound as we hear it being generated. The stilted quality of some computer music derives from this predicament. The advantage of programmability becomes a disadvantage when we have to encode simple musical phrases with the same overhead as more complicated ones. Even a trivial envelope may require us to precalculate and type in dozens of numbers. Non-real-time software synthesis is "the hard way" to make music.

Fortunately, dramatic speedups in hardware are pushing more and more synthesis methods into the arena of real-time operation. Commercial synthesizers based on DSP microprocessors circuits allow flexibility in programming synthesis algorithms. Only the most esoteric and complex methods, like some forms of *parameter estimation* and *analysis-resynthesis* (chapters 7 and 13), remain outside the limits of low-cost real-time hardware. So today we can choose between real-time and non-real-time synthesis, depending on the musical application. Besides the time savings, real-time synthesizers have the great advantage that they can be played—animated by a musician's gestures as sound is heard.

Specifying Musical Sounds

Now we turn to the different ways to specify a piece of music to a synthesis system. The traditional way of making a piece of music is to select various instruments and write a paper score that directs the performers to play specified musical events, allowing room for interpretation depending on the performers and the instruments they happen to play. But the possibilities of digital synthesis extend far beyond the ink of traditional scores.

Sound Objects

In traditional music theory, the note is a static, homogenous, unitary event. Modern synthesis techniques suggest a generalization of the concept of musical event called a *sound object* (Schaeffer 1977; Chion and Reibel 1976; Roads 1985f). The notion of sound object is often useful, since it can encompass sounds that are longer than one ordinarily considers a note to be, or more complicated. A sound object may contain hundreds of short subevents (as in vector and granular synthesis). Or it may be controlled by a dozen or more time-varying parameters, causing it to undergo mutations of identity from one pitch/timbre to another.

The burden of controlling the complicated parameter evolutions for sound object synthesis falls to the composer. This begs the question: how can we specify all these time-varying quantities? In the next section we show how much data a common synthesis technique may require. Then the section on the musician's interface presents five strategies for supplying it.

Example of the Specification Problem for Additive Synthesis

Additive synthesis is a venerable method of sound synthesis. Faithful to its name, it sums the output of several sine wave oscillators to form a composite sound waveform.

Figure 3.10 presents a digital synthesis instrument for additive synthesis. The instrument includes a frequency envelope as well as an amplitude envelope for each oscillator. The frequency envelope is a time-varying function with a range $[-1.0, +1.0]$. This envelope scales the *peak deviation* value specified as one input to **env_osc**. If the peak deviation is 100, for example, and the frequency envelope at its lowest point is -0.1, the value coming out of the frequency envelope at that point is -10. The adder (+) sums this with the center frequency of the lower oscillator, causing the frequency to droop from its nominal center point. If the center frequency had been

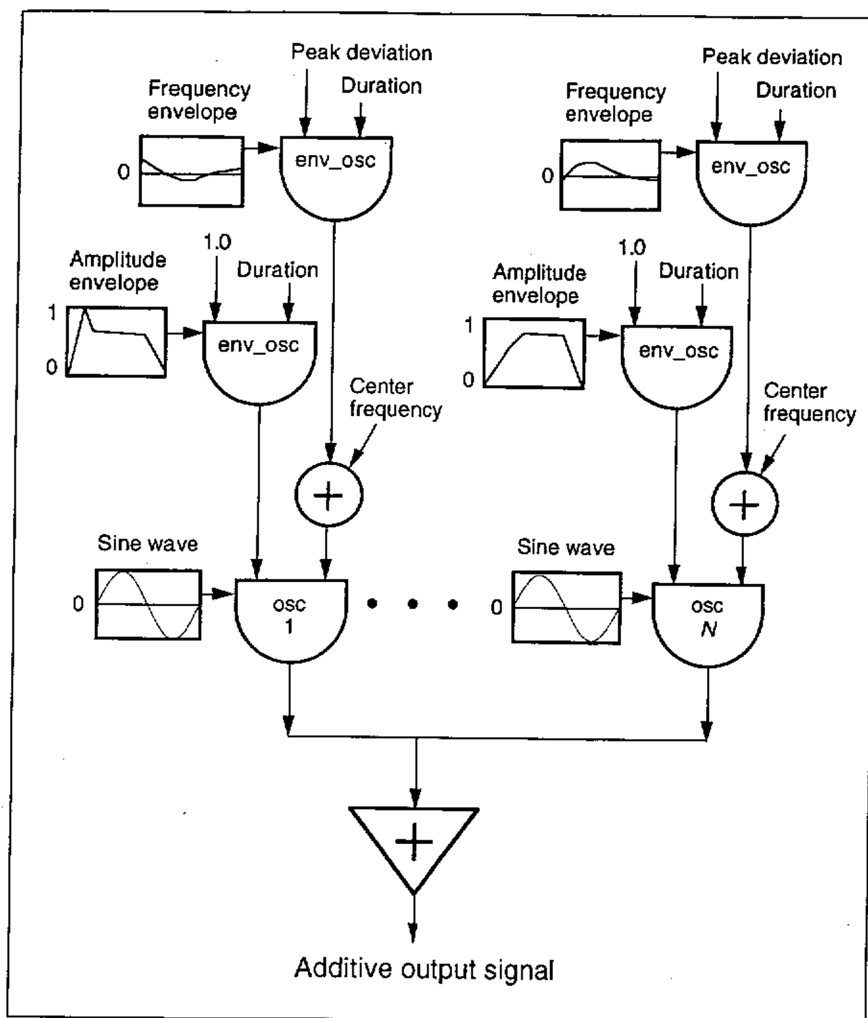

Figure 3.10 The patch shown in figure 3.7 expanded to form a simplified instrument for additive synthesis. Each sine oscillator is modified by an amplitude and frequency envelope. The outputs of many sine oscillators are added together to make one sample. More three-oscillator units might be added to this patch to make more complicated sounds.

specified as 440 Hz, the frequency envelope would cause it to go down to 430 Hz at some point.

Notice how each vertical slice of this instrument includes two envelope generators and an audio oscillator. We will call this unit a *voice*. Only two voices are shown, but the ellipses indicate that other voices are hidden. Such an instrument can generate an extremely wide range of sounds—provided that we can specify the data.

Now we turn to the problem of specifying the parameters for the instrument in figure 3.10. For each voice and each event, the instrument requires the following parameters:

1. Center frequency of audio oscillator **osc**
2. Peak amplitude (set as 1.0 in the figure)
3. Amplitude envelope
4. Begin time of amplitude envelope
5. Duration of amplitude envelope
6. Frequency envelope
7. Begin time of frequency envelope
8. Duration of frequency envelope

If the instrument has fifteen voices, and each voice requires these eight data values, that means 120 data values must be specified for just one event!

Thus no matter how powerful synthesis hardware becomes, the problem of specifying the *control data* remains. In chapter 4 we look in more detail at the data requirements of additive synthesis. The next section presents six general strategies that apply to all synthesis techniques.

The Musician's Interface

The different ways of supplying synthesis data to a computer and synthesizer fall into six categories:

1. Musical input devices
2. Performance software
3. Editors
4. Score languages
5. Algorithmic composition programs
6. Sound analysis programs

Figure 3.11 schematizes these categories. The first five categories correspond to the *musician's interfaces* explored in part V of this book. The last category is covered in part IV. The next six sections explain briefly each category.

Musical Input Devices

Musical input devices are the physical instruments manipulated by musicians (see chapter 14). The instrument directly links the musician's gestures to the production of sound. Electronic input devices decouple the manipulation of sound from the need to power it physically. Hence they are potentially more flexible than traditional instruments. For example, with electronic instruments, a single wind controller can create low bass sounds as easily as high soprano sounds. Indeed, electronic input devices are so easy that one research direction seeks to reinfuse the physical difficulty, to recreate the sense of effort that leads to expressive performances.

The benefits of real-time musical input devices are clear, although the technical problems associated with connecting them to a computer can be formidable. Traditional acoustical instruments developed over hundreds of years, whereas their digital counterparts have just begun their evolution. Musical input devices are best suited for fine control of a few musical parameters. For example, the keys on a keyboard can indicate pitch, while the velocity of key depression determines the amplitude of the higher-frequency oscillators. Most MIDI keyboards have one or more *continuous controllers* (such as footpedals, modulation wheels, or joysticks). These controllers can be assigned to any manipulable parameter, so we might set the foot pedal to control overall amplitude, and a modulation wheel to bend the shape of the fundamental pitch.

Performance Software

The use of real-time *performance software* is expanding due to the proliferation of MIDI-based systems (see chapter 15). Performance software includes such utilities as *sequencers* that can remember and play back keyboard performances. Sequencers record pure control data (such as the onset time of key depressions on a keyboard, signaling the beginnings of notes) rather than samples of audio waveforms. Computer music also provides the opportunity to go beyond traditional solo performance, for example, to provide control at the level of a conductor of an ensemble.

Fitted with eyes (a camera or another type of sensor) and ears (microphones and sound analysis software), computer-based instruments can

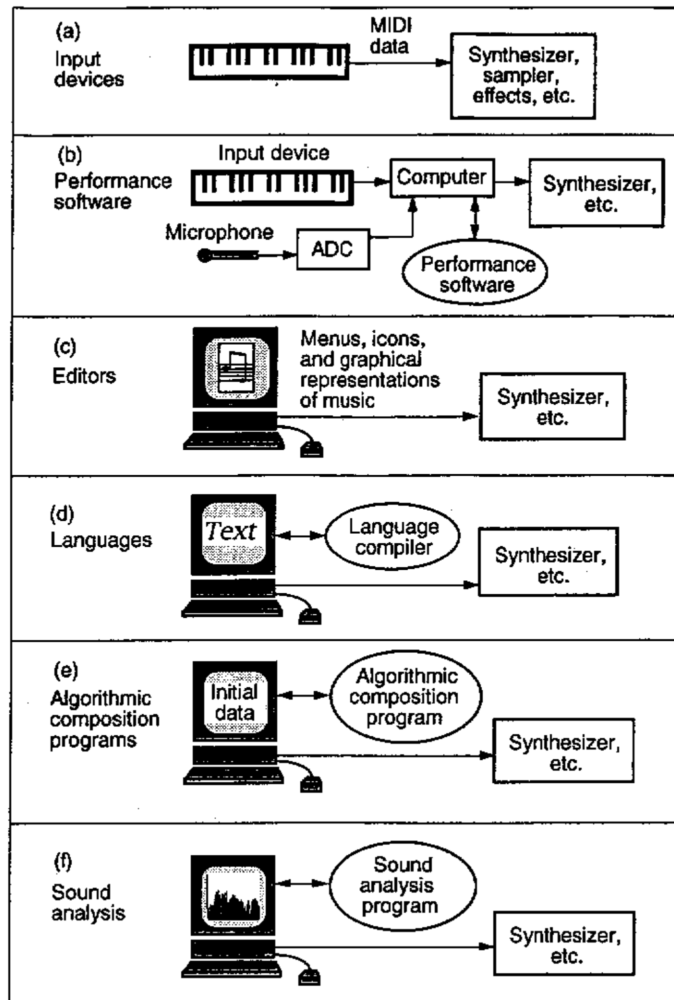

Figure 3.11 The musician's interface: six different ways of specifying synthesis data to a computer or synthesizer. (*a*) An input device can transmit the necessary data directly to a synthesizer, with or without a computer in between. (*b*) Performance software interprets the performer's gestures and may even be able to improvise. (*c*) Editors let the user build up a specification through interactive graphics techniques. (*d*) Languages encode the specification as a precise text. (*e*) Algorithmic composition programs typically require a small amount of initial parameter data from the composer before they generate music. (*f*) Sound analysis automatically derives data for modification and resynthesis from sounds fed into it.

respond to a human gesture in arbitrarily complex ways, through the use of procedures embedded in the performance software. It is increasingly common to see concerts in which a synthesizer controlled by a computer improvises with a human performer. Another application of such a system is a more flexible rendition of a prepared score, replacing the fixed tape recorder mode of performance.

As a simple example of performance software, one might set up a situation whereby a certain passage played on a keyboard triggers the start of a prerecorded score section, while a single high C key stops the sequence. A modulation wheel might determine the tempo of the prerecorded sequence.

Editors

An *editor* program lets a musician create and change a text, sound, or image (see chapter 16). Many interactive editors employ graphics techniques to provide an efficient environment for the musician. The material being edited can be quickly cut, pasted, or changed with simple gestures.

Graphical editors facilitate rapid prototyping of ideas, and hence they are most often found in the individual studio, where there is time for research. Musical ideas can be built up incrementally in an editor, and often the musician can hear the result as the change is being made.

Since music exists on many different levels and perspectives, it makes sense that there should be many types of editors for music. To set up a performance for an additive synthesizer, one uses score, instrument, and function editors. We enter the parameters for each sound object into a text editor, or manipulate a graphic image (such as common music notation or piano-roll notation). The instrument editor editor configures the additive synthesizer from unit generators such as oscillators and envelope generators. At the end of an editing session we tell the program to write the patch to the synthesizer. A function editor provides several ways of defining functions of time (waveforms and envelopes), including graphical methods and mathematical formulas. We apply the function editor to the task of creating the amplitude and frequency envelopes for the various oscillators.

Languages

Perhaps the most precise method of specifying music involves preparing *note lists* or *play lists* that are part of a *score language* (see chapter 17). The score language defines a syntax for the parameters of the instrument, listed in individual *parameter fields* (abbreviated *pfields*).

; p1	p2	p3	p4	p5	p6
; Ins	Start	Dur.	Freq.(Hz)	Amp.(dB)	Waveform
i1	**0**	**1.0**	**440**	**70**	**3**
i2	**1.0**	**.5**	**660**	**80**	**4**

Figure 3.12 Numerical score example. Three lines of comments followed by a two-line score. The first line specifies a note for synthesis instrument 1 (**i1**), while the second specifies a note for **i2**.

Our first example of a score language was the simple score line in figure 3.8b. Traditionally the first parameter after the name of the instrument gives the start time, and the second parameter gives the duration for the event. Subsequent parameters have different meanings, depending on the nature of the instrument. For example, the first line of the score file shown in figure 3.12 says that the event uses instrument 1, starts at 0, plays for 1.0 seconds, has a frequency of 440 Hz, an amplitude of 70 dB, and uses waveform number 3. (The two bottom lines in bold are the score; the other lines are comments.)

Score languages also contain *function table definitions*—the envelope and waveform definitions used by the instruments (see chapter 17).

Traditional score languages are basically numeric: instruments, pitches, and amplitudes are expressed as numbers. Alternative score languages support more "natural" specifications of music, allowing equal-tempered pitch names, for example. (For a discussion of score languages, see Smith 1973; Schottstaedt 1983, 1989a; Jaffe 1989; also Loy 1989a and chapter 17.)

The principal advantage of score languages is also their disadvantage: precision and detail. With a language, musicians are required to enter the score as an alphanumeric text. Not all composers care to specify their music in such minute detail at all times. In the additive synthesis example given above, the musician would be required to type 120 values for each sound object. On the other hand, a score language lets the musician precisely specify a score that is so detailed that it could never be played accurately by a human performer.

Algorithmic Composition Programs

Some of the earliest work in computer music involved *algorithmic composition:* the creation of a music score according to a procedure specified by the composer/programmer (Hiller and Isaacson 1959; Xenakis 1971; Barbaud

1966; Zaripov 1969). For example, the computer can calculate the parameters of sound according to a probability distribution or another type of procedure (see chapters 18 and 19).

For example, suppose that we feed a set of initial data to an algorithmic composition program, and then let it generate a complete score including all parameters needed for additive synthesis. Chapter 19 shows that there are many possible strategies that an algorithmic composition program might take. Hence it is understandable that the nature of the initial data varies from program to program. For a program that computes a score on the basis of probabilities, the composer might specify these general attributes of the score:

1. Number of sections
2. Average duration of sections
3. Minimum and maximum density of notes in a section
4. Grouping of frequency and amplitude envelopes into *timbre classes*
5. Probability for each instrument in a timbre class to play
6. Longest and shortest duration playable by each instrument

In this case, the control is global and statistical in nature. The composer can determine the overall attributes of the score, but all the details are calculated by the program. In other programs, the data might be more detailed and the stylistic constraints more specific.

Sound Analysis

Like music, sound can be dissected in innumerable ways. The established categories of sound analysis target three aspects: pitch, rhythm, and spectrum. We can use the output of these analyzers to drive synthesis, as in a convolver that maps the rhythm of one sound onto the timbre of another (Roads 1993a; chapter 10), a pitch detector tracking a human voice that drives the accompaniment pitch of a digital oscillator (chapter 12), or a spectrum analyzer that extracts the time-varying frequency and amplitude curves for additive resynthesis (chapter 13).

Conclusion

Developments in physical and electronic acoustics have opened the way for numerous experiments in musical tone production. Creations in this category represent the most

avant-garde developments in music today. The new sounds, added to new rhythmic, harmonic, and tonal concepts, make the music extremely difficult to evaluate in terms of normal musico-aesthetic standards.
—H. Miller (1960)

The musical potential of digital sound synthesis has begun to be explored, but much remains poorly understood. For now, digital technology allows precise and repeatable sound generation. With the proper hardware, software, and audio playback system, we can generate musical signals of very high audio quality. Perhaps even more important than precision is programmability, which translates into musical flexibility. Given enough memory and computation time, a computer can realize any synthesis algorithm, no matter how complicated.

While hardware continues to increase in speed, there is a continuing problem of finding the proper control data to drive the synthesis engine. One of the challenges of synthesis is how to imagine and convey to the machine the parameters of the sounds we want to produce. The point of specification is the musician's interface, discussed in the six chapters comprising part V, and sound analysis, presented in part IV.

Music theory lags a half century behind the actual practice of computer music. Synthesis techniques of leading composers are exploring the space of possibilities, leaving behind charts of musical sound geography for future generations to scan. The history of music in times of experimentation like these indicates that the current period is leading to an era of consolidation—when much of the experimentation of today will seem mundane, when the resources that at present seem radical will appear commonplace. Music composition will then enter a new era of refinement, and questions of orchestration can again be addressed within a systemic framework, as they were in the age of the symphony orchestra.