Nick Porcaro, David Jaffe, Pat Scandalis, Julius Smith, Tim Stilson, and Scott Van Duyne
Center for Computer Research in Music and Acoustics (CCRMA)
Department of Music
Stanford University
Stanford, California 94305, USA
{nick, daj, gps, jos, stilti, savd}@ccrma.stanford.edu

SynthBuilder: A Graphical Rapid-Prototyping Tool for the Development of Music Synthesis and Effects Patches on Multiple Platforms

SynthBuilder is a graphical, extensible, object-oriented application for interactive, real-time design of audio signal-processing patches. It played a major role in the development of coupled-mode synthesis (Van Duyne 1997), virtual analog synthesis (Stilson and Smith 1996), and enhancements to physical-modeling synthesis such as the lossless, click-free, pitch-bendable delay line (Van Duyne et al. 1997). It was created primarily for prototyping physical models, as part of the Stanford Sondius physical-modeling project (see http://www.sondius.com), although it can also be used for effects processing as well as other types of synthesis algorithms.

SynthBuilder was written by Nick Porcaro, with significant contributions by David Jaffe, Pat Scandalis, Julius Smith, Tim Stilson, and Scott Van Duyne. It derives much of its functionality from the Music Kit (developed by NeXT Computer, Inc. and later by CCRMA) (Smith, Jaffe, and Boynton 1989) and the NeXT Draw application, both written in the Objective-C language. The first version of SynthBuilder was inspired by GraSP, a prototype for a graphical Music Kit editor by Eric Jordan (David Jaffe, advisor) at Princeton University in 1992. GraSP was in turn preceded by SynthEdit, another early prototype at NeXT (Minnick 1990).

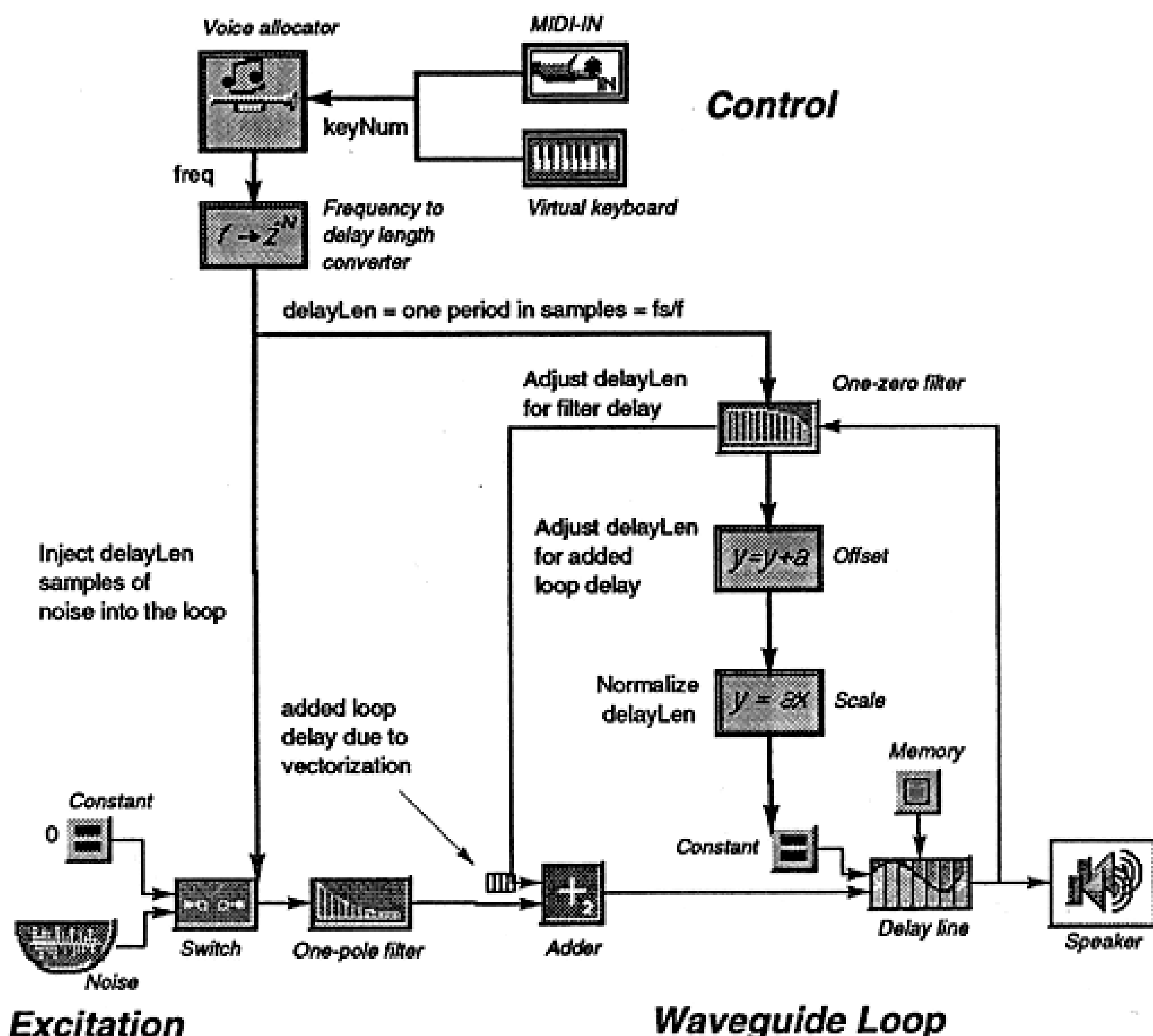
In 1997, development of SynthBuilder moved to Staccato Systems, where it was ported from its original version (which ran only on the NEXTSTEP operating system, with the signal processing running on Motorola DSP5600x signal processors) to a more portable implementation that runs on Windows and other platforms. As of November 1997, SynthBuilder runs on NEXTSTEP for Motorola and Intel hardware (release 3.2 or later) using DSP5600x processors, and OpenStep 4.2 for Windows 95 using the host processor. See the World Wide Web site http://www.staccatosys.com/ for information on how to obtain a free version of SynthBuilder, including detailed on-line documentation and examples.

Computer Music Journal, 22:2, pp. 35–43, Summer 1998
© 1998 Massachusetts Institute of Technology.

Overview

SynthBuilder allows the user to graphically create sound-synthesis programs (*patches*), consisting of interconnections (called *audio patch cords* or *midi cables*, after their physical counterparts) between digital audio signal-processing elements (*unit generators*) and event-processing elements (*note filters*). Events are represented in a flexible representation (*note*) capable of containing arbitrary groups of parameters. (Unlike the Music Kit, SynthBuilder uses lowercase names for its objects.) A MIDI "front end" converts MIDI messages into notes. SynthBuilder makes a patch active by creating a Music Kit "shadow" of the patch and its connections, with each graphical object shadowed by a Music Kit synthesis or event-processing object. The patch then runs in real time and is playable and adjustable from real or virtual MIDI devices, as well as from the graphical interface that the user creates. No "compute-then-listen" cycle is required, and no code need be written. However, advanced users can augment SynthBuilder's functionality by writing their own dynamically loadable unit generators and note filters.

Figure 1. SynthBuilder patch for a simple plucked-string algorithm. The thick lines are midi cables, the thin lines are audio patch cords. Note that both unit generators and note filters can have midi cable connections, but only unit generators can have audio patch cord connections.



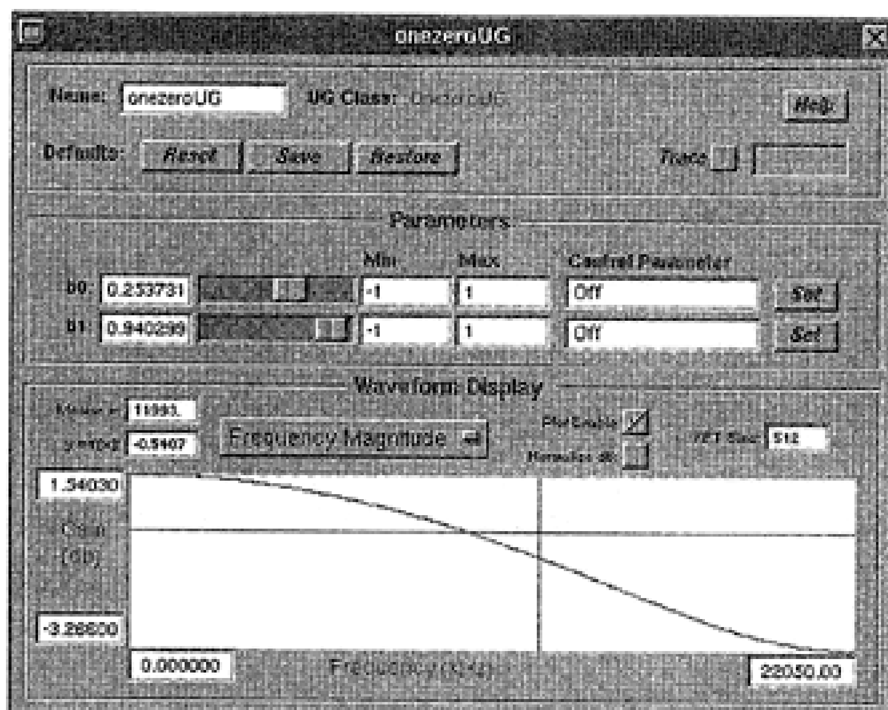
This article will discuss SynthBuilder from a user's standpoint, and will also supply some important implementation details.

Graphical Interface

The user sees note filters, unit generators, and user-interface objects as graphical icons in a SynthBuilder patch, along with text, lines, and other graphics. Figure 1 shows a SynthBuilder patch for a simple plucked-string algorithm. NeXT's Draw application (whose source code is publicly available) served as a good starting point for SynthBuilder. Draw already includes basic drawing features, such as text fonts, lines, polygons, curves, grouping, colors, rulers, an alignment grid, multiple documents, copy/paste, drop/drag of images, and export to two graphic formats: Tagged Image File Format (TIFF) and Encapsulated PostScript (EPS). SynthBuilder augments Draw with a browser that contains two elements: a hierarchical catalog of unit generators and note-filter classes, and an icon well from which users can drag instances to drop into a patch. SynthBuilder also adds tools for drawing connections between these elements: a line-routing algorithm (Rosati 1992) for clearly and legibly depicting the connections, a hierarchical *subpatch* mechanism, and numerous other tools for designing synthesis instruments.

The Draw user-interface paradigm is carried forward to all elements of a patch. For example, text annotations, graphics, unit generators, note filters, and subpatches can be freely combined, grouped,

Figure 2. One-zero filter inspector. When the sliders are moved, changes in the state of the filter are made on the waveform display and on the DSP (or host, depending on the platform) in real time.



formatted, and copied and pasted within a document or between documents. In addition, any of these elements can be inspected. When the user double-clicks on a graphical item, the inspector panel for that item displays itself. For example, the line inspector allows setting of line width, whether there are arrowheads at the ends of the line, and various other attributes. In the case of unit generators and note filters, the inspector allows the user to adjust variables that affect the way the element processes audio or notes. The user makes these adjustments by moving sliders or typing into a text field in real time during synthesis. The changes are heard immediately. Figure 2 shows the inspector for a one-zero filter.

Subpatches—Voices, Instruments, and Processors

Hierarchical subpatches serve a number of purposes. While they can be used for any arbitrary encapsulation, they are typically used in three common situations. First, they can represent individual voices in a polyphonic instrument. Each of these voices is managed by a single *voice allocator* note filter (see "Voice Allocation," below). Second, subpatches can represent individual instruments in a polytimbral context. In this case, each instrument has its own voice allocator. Finally, subpatches allow the user to specify how a patch is distributed across multiple processors (see "Multiprocessor Support," below). The processor assignment can be made for the instrument as a whole, or even divided up within an instrument. A subpatch can optionally inherit the processor assignment from the subpatch above it. For example, a typical synthesis-oriented (as opposed to effects-oriented) patch has subpatches containing several musical instrument algorithms driven by MIDI and/or a score. Each polyphonic instrument algorithm in turn has a subpatch for control, as well as multiple subpatches that correspond to voices, as shown in Figure 3. A typical voice subpatch is shown in Figure 4.

Subpatches support several mechanisms to aid code-sharing and work by development teams. These include *variations, linked subpatches,* and *subscribe/publish.* Variations are different versions of the same subpatch, only one of which is active at a given time. The user selects the active variation from a configuration panel in the top-level patch. This provides a convenient mechanism for maintaining a single top-level patch that will run on many different digital signal-processor (DSP) card configurations. Linked subpatches support consistency between copies of a patch; if you change one of the subpatches in a linked set of subpatches, the others are automatically changed along with it. Subscribe/publish facilitates sharing and version control of subpatches among a development team.

Implementation of the Graphic Representation

The graphical representation of unit generators and note filters is implemented in a subclass of the Draw application's Graphic class, called ElementGraphic. ElementGraphic contains functionality for operations such as drawing, moving, and pin rotation. It also adds support for an inspector. An early implementation of SynthBuilder contained a generic inspector mechanism, but this proved inadequate for describing the behavior of complex unit generators and note filters. For example, when adjusting a filter, the user would like to see a graph of phase and frequency response (see Figure 2). To make this

Figure 3. Typical synthesis-oriented patch. Performance data comes from prerecorded scores, live MIDI, or user-interface actions such as dragging a slider or clicking on the virtual keyboard. Instrument subpatches (such as the flute, harp, clarinet, and horn shown above) are polyphonic, as shown in Figure 4.

Figure 4. Typical instrument voice. The voice allocator feeds three linked instances of a subpatch, each containing a single clarinet voice. Another subpatch containing controls specific to the voice feeds the voice allocator.

IN
score recorder
global
Master Volume
global
Master Tuning
Pitch bend
sensitivity
Intra-patch MIDI pin (typical)
global
score1
flute
flute
harp
flute
clarinet
horn
violin
harp
score2
cello
clarinet
flute
harp
clarinet
horn
horn
horn

Figure 3

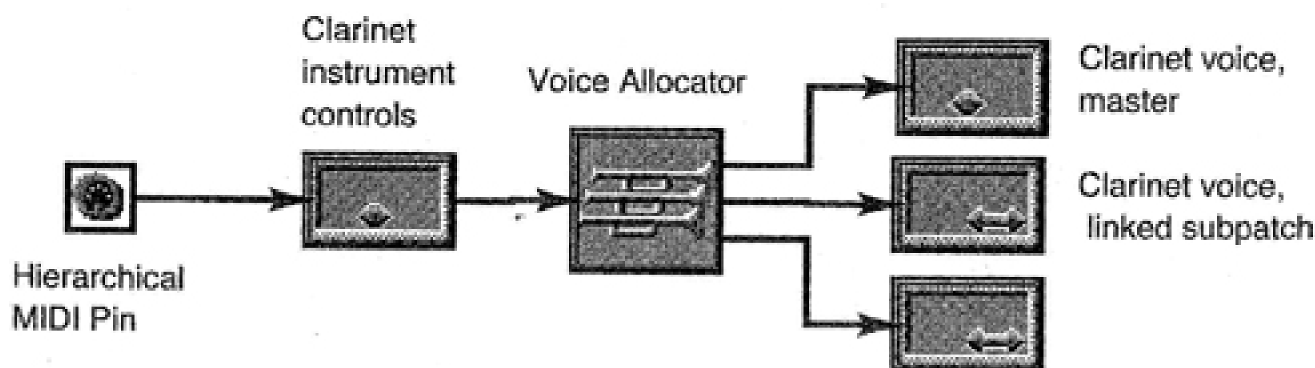


Figure 4

possible, SynthBuilder provides a system of *dynamically loadable bundles* that allows a custom inspector to be specified and dynamically loaded for each class of unit generator or note filter.

Each note-filter or unit-generator bundle has, as its primary class, a subclass of ElementGraphic that is responsible for adding appropriate user-interface controls for that unit generator or note filter. Other resources in the bundle include a TIFF (image) file for its representation in the document, an example patch, and a usage document that is accessible from the help system.

The bundle architecture also has the advantage of allowing each unit generator or note filter to be developed and tested independently from the main application. This facilitates integration with existing NEXTSTEP interfaces developed by others at CCRMA and elsewhere. This has helped SynthBuilder to evolve into a common platform for the work of a number of independent researchers. For example, the filter unit generators all use Gary Scavone's waveform display tool, which shows the filter's amplitude, phase, group delay, and impulse response. The envelope handler uses a graphical view derived from Fernando Lopez-Lezcano's envelope editor. The memory module allows viewing and/or playing of its contents with an external tool, such as Perry Cook's Spectro application. (For information about these three tools, see CCRMA's World Wide Web site, http://www-ccrma.stanford.edu.)

Bundles make it easy for sophisticated users to develop new unit generators and add them to SynthBuilder. In the course of our physical-modeling development, several such unit generators were created, including a periodically triggered excitation table (PET) unit generator for bowed strings (Jaffe and Smith 1995), a special delay line for accurate handling of high pitches in guitars (Van Duyne et al. 1997), and a Moog-style voltage-controlled filter (VCF) (Stilson and Smith 1996).

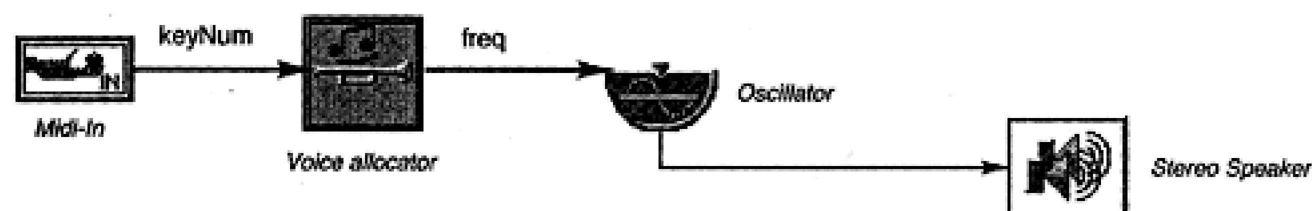
Implementation of Synthesis-Algorithm Control

To control synthesis algorithms, SynthBuilder makes extensive use of the Music Kit's support for events and scheduling, especially the Music Kit's NoteFilter class. While the concept of unit generators is very familiar, dating back to Max Mathews' Music languages (Mathews 1969), note filters are a more recent innovation. Each note filter is event-activated; that is, a note filter performs an action when it receives an event, either from MIDI or from the user interface. Typically, a note filter performs some action on each event it receives via its inputs, then passes the modified event on to its outputs. Note filters also have access to the Music Kit scheduler, and can schedule callbacks for particular times in the future. The name "note filter" derives from the use of Note objects (packages of parameters) to represent the events handled by note filters. The Music Kit Midi object converts each incoming MIDI message to a Music Kit Note object, which is then sent to the first of a chain or network of note filters. After being handled by a note filter, the note travels to the next note filter in the network, and so forth.

Note filters also provide the means for mapping note parameters to low-level synthesis control. When drawing a midi cable from a note filter to a unit generator, the user is actually connecting to a hidden note filter that is associated with the particular unit generator. When a note reaches it, the note filter translates the note's parameter values into calls to the unit generator's instance methods, which in turn set the unit generator's instance variables. The user sets up this mapping from the SynthBuilder unit-generator inspector, which lists the unit generator's instance methods and allows an association to be made with a note parameter.

An example will help clarify how all this works: First, the user draws a simple patch by dragging and dropping a Midi-In note filter, a voice-allocator note filter, an oscillator unit generator, and a stereo speaker unit generator which represents the audio output. One then connects the oscillator output to the input of the stereo speaker, using the audio-patch-cord connection tool. Next, one connects the output of the Midi-In note filter to the input of the voice allocator, and the output of the voice allocator to the MIDI input of the oscillator, using the midi cable connection tool. One completes this simple patch by double-clicking on the oscillator, bringing up its inspector, and specifying

Figure 5. Simple oscillator patch. Notes containing a freq parameter coming out of the voice allocator get mapped to the setFrequency instance method of the oscillator.



that the "freq" parameter in incoming notes be mapped to the "setFrequency" instance method. Figure 5 shows this simple patch.

One then proceeds to play the patch. Depressing an A above middle C on a MIDI keyboard has the following effect: the Midi-In note filter (using its Music Kit shadow object, the Midi object) creates a note in response to the MIDI note-on message, giving it the ordinary MIDI note-on parameters (key number, velocity, and channel). The Midi-In note filter sends this note to its output, which is then sent to the voice allocator. The voice allocator inserts a parameter called "freq" into the note, containing the frequency (in Hertz) corresponding to the key number. Next, the note leaves the voice allocator and travels to the oscillator, which has the effect of invoking the "processNote" method of the note filter associated with the oscillator. This note filter sees that the "freq" parameter is present in the note, and invokes the unit-generator method "setFrequency," which changes the oscillator frequency on the Music Kit shadow object (which is running on the DSP or the host, depending on the platform) to 440 Hz.

Voice Allocation

Voice allocation is managed in SynthBuilder by a note filter called the voice allocator, based on the design of the Music Kit's SynthInstrument class. The voice allocator manages a set of voices, usually encapsulated as subpatches, with each representing a monophonic sound-producing entity. The voice allocator keeps track of which voices are currently sounding. Each voice allocator represents a single MIDI channel. For example, controller information is forwarded to all voices, while polyphonic aftertouch is routed to the appropriate voice. For creating multichannel synthesizers, multiple voice allocators are used. See Figures 1, 3, and 4 for an illustration of this.

In SynthBuilder, one is in effect designing synthesizers or effects boxes. While MIDI arguably does an adequate job of depicting information passed from a keyboard to a synthesizer, it is inadequate for describing all that happens within a synthesizer. For example, since most synthesizers support envelopes that gradually decay after a note-off is received, the MIDI note-off does not mean that a voice is available for reallocation. It merely signals the beginning of the final portion of the note.

When designing a synthesis algorithm, you often need to know when a note is truly over. For example, you might want to produce a different kind of attack, depending on whether a new note-on occurs before or after the preceding note has fully died away. However, the knowledge as to how much time is needed to finish the note may be local to a unit generator or note filter deep within the voice. Therefore, a mechanism is provided for communicating back from the unit generators to the voice allocator.

The voice allocator handles computing frequencies for each voice based on pitch-bend parameters, as well as managing the damper pedal. In addition, if the voice allocator receives a note for which no voice is available, it preempts the oldest running voice in such a way as to guarantee no click. Finally, the voice allocator also handles the case of a monophonic voice, such as a flute, in which connections between notes are important.

Unit Generator Running Order

The set of unit generators in a patch can be viewed as a list of subroutines that are called once for each output sample. (More precisely, each unit

generator reads and writes a vector of samples each time it is run; see "Optimizations," below.) The order in which these unit generators are run is significant in some cases. For example, if unit generator A writes to unit generator B and unit generator B writes to unit generator C, there will be no delay if the running order is A, B, C; but there will be two "hidden" delays if the order is C, B, A. Hidden delays are particularly important in algorithms that include feedback, such as delay-line loops. Designers of feedback systems need to account for these delays, or the effective loop lengths will be slightly longer than desired. SynthBuilder indicates such hidden delays by displaying a small delay symbol on affected audio patch cords. Figure 1 shows such a delay symbol on the top input of the adder. While it is impossible to completely eliminate hidden delays from feedback systems, they can be reduced to one hidden delay per feedback loop. As an aid in minimizing hidden delays, SynthBuilder provides a tool that analyzes the signal graph and then reorders the unit generators to an optimal order.

Debugging a Patch

SynthBuilder has extensive support for tracing the flow of notes through a patch. Individual note filters and unit generators can be selectively traced. Certain note types (e.g., note-on) can be selectively traced. The format of the tracing is also controllable; brief or verbose information about the contents of the note, timing information, and low-level DSP information can selectively be enabled or disabled.

In addition, the ease of modifying patches in SynthBuilder makes it possible to debug by isolating various portions of the patch and examining their behavior in a simpler context. An audio patch cord "probe" tool allows the user to listen to a particular audio patch cord and to record its signal to a sound file, which can then be studied. Similarly, midi cables can be selectively muted or unmuted, allowing or disallowing the flow of notes.

Multiprocessor Support

SynthBuilder uses the Music Kit's multiprocessor architecture to distribute synthesis resources among a set of processors. Multiprocessor support was built into the Music Kit from its inception in 1987. However, it was not until 1990 that the first actual multi-DSP hardware port was made, to the Ariel QuintProcessor (Jaffe 1990). Then, with the release of the Music Kit for Intel (Jaffe, Smith, and Porcaro 1994), it became possible to use multiple low-cost DSP cards in a PC.

In 1995, the Sondius Project sponsored the building of a low-cost multiprocessor DSP engine with faster DSPs than had been available for the QuintProcessor. This engine is known as Frankenstein (Putnam and Stilson 1996). Eight Motorola 56002 evaluation modules were connected via their host ports to an ISA interface that allowed the Music Kit to address each DSP as a separate computing resource. In addition, the modules were over-clocked at 76 MHz, giving the Frankenstein a computation power of approximately 300 DSP MIPS (millions of DSP instructions per second).

Frankenstein's main purpose was to test complex patches, such as the commuted-synthesis piano (Smith and Van Duyne 1995; Van Duyne and Smith 1995). Several multi-instrument scores were developed to exploit Frankenstein's full potential, such as the *Jazz Trio* sound example that accompanies this article. [Sound examples for this article will appear on the forthcoming *Computer Music Journal Sound Anthology CD*, Volume 22. —Ed.]

Optimizations

For the sake of efficiency, SynthBuilder computes vectors of samples, rather than one sample at a time (Smith 1989). This serves to amortize the price of the preamble and "postamble" code for each unit generator. The larger the tick (vector) size, the greater the amortization. However, the tick size also determines the smallest possible delay in the recursive structures that often arise in physical-modeling implementations. For example,

if the tick size is 44 and the sampling rate is 44 kHz, then the highest pitch possible from a Karplus-Strong plucked string is 1,000 Hz. SynthBuilder currently uses the Music Kit's tick size of 16.

Having a tick size greater than one implies that it is not possible in SynthBuilder to create arbitrary filter structures composed of unit delays, scales, and adds. To solve this problem, we are considering providing a set of nonoptimized unit generators based on a one-sample tick size. These can be used to develop arbitrary structures, which may then be encapsulated in custom, optimized unit generators.

Of course, with any real-time system there is a limit to how much the system can accomplish in real time. For patches that exceed the computational resources, SynthBuilder allows you to record a MIDI stream, then play it back, while computing the samples and writing them to a sound file. In this case, the processor need not keep up with real time, and memory is the only limitation.

Recent Work

As of this writing, SynthBuilder has been ported to Windows 95/NT, using OpenStep 4.2, also known as the Yellow Box of the Apple Rhapsody operating system. As of November 1997, Apple's position is that Rhapsody will work on Intel platforms running Windows 95/NT, as well as on Mac OS 8.0 systems using the PowerPC processor. The new version of SynthBuilder is very similar to the NEXTSTEP version, except that it follows Windows user-interface conventions.

In addition, the Music Kit has been replaced by a new low-level system, called SynthServer, that can be easily ported to a variety of systems. SynthServer is a synthesis engine in that it consists of core synthesizer code without the user interface. It contains floating-point implementations of the SynthBuilder unit generators and note filters, as well as facilities for handling MIDI and scheduling notes, reading and writing sound files, and performing real-time audio. Unlike the DSP version of SynthBuilder, SynthBuilder and SynthServer run as two separate processes.

SynthBuilder communicates with SynthServer using an interchange language called SynthScript. A SynthScript patch contains a list of unit generators and note filters, along with interconnection information, initial parameter values, and mappings. When SynthBuilder runs a patch, it generates a SynthScript representation and sends it to SynthServer via an inter-process communication (IPC) mechanism. User-interface actions, such as moving a slider or playing a note on a virtual keyboard, also send SynthScript to the server. SynthScript also contains a specification for high-level performance controls, which are exported to the SynthServer control panel.

The separation of the user interface (SynthBuilder) from the engine (SynthServer) makes it possible to run the synthesis part of the patch in a stand-alone fashion without SynthBuilder. In fact, any application that generates SynthScript can communicate with SynthServer. In addition, SynthServer can read patches from a file (specified from its control panel), and perhaps in the future from a MIDI message.

Finally, we are augmenting the note-filter mechanism with a text-based interpreted language, containing scheduling primitives and operators for manipulating notes. Thus, users will have three options, depending on their needs and level of expertise: they can use existing note filters; they can write code in the note-filter language; or, for maximum efficiency, they can write their own dynamically loadable, precompiled note filters.

References

Jaffe, D. A. 1990. "Efficient Dynamic Resource Management on Multiple DSPs, as Implemented in the NeXT Music Kit." In *Proceedings of the 1990 International Computer Music Conference*. San Francisco: International Computer Music Association, pp. 188–190.

Jaffe, D. A., and J. O. Smith. 1995. "Performance Expression in Commuted Waveguide Synthesis of

Bowed Strings." In *Proceedings of the 1995 International Computer Music Conference.* San Francisco: International Computer Music Association, pp. 343–346.

Jaffe, D. A., J. O. Smith, and N. Porcaro. 1994. "The Music Kit on a PC." In *Proceedings of the First Brazilian Symposium on Computer Music, XIV Congress of the Brazilian Society of Computation.* Canela, Brazil: Informática UFRGS, pp. 63–69.

Mathews, M. V. 1969. *The Technology of Computer Music.* Cambridge, Massachusetts: MIT Press.

Minnick, M. 1990. "A Graphical Editor for Building Unit Generator Patches." In *Proceedings of the 1990 International Computer Music Conference.* San Francisco: International Computer Music Association, pp. 253–255.

Putnam, W., and T. Stilson. 1996. "Frankenstein: A Low Cost Multi-DSP Computer Engine for the Music Kit." In *Proceedings of the 1996 International Computer Music Conference.* San Francisco: International Computer Music Association, pp. 45–46.

Rosati, C. 1992. "Simple Connection Algorithm for 2-D Drawing." In D. Kirk, ed. *Graphics Gems III.* New York: Academic Press, pp. 182–187.

Smith, J. O. 1989. "Unit-Generator Implementation on the NeXT DSP Chip." In *Proceedings of the 1989 International Computer Music Conference.* San Francisco: International Computer Music Association, pp. 303–306.

Smith, J. O., D. A. Jaffe, and L. Boynton. 1989. "Music System Architecture on the NeXT Computer." In *Proceedings of the AES Seventh International Conference: Audio in Digital Times.* New York: Audio Engineering Society, pp. 301–312.

Smith, J. O., and S. A. Van Duyne. 1995. "Commuted Piano Synthesis." In *Proceedings of the 1995 International Computer Music Conference.* San Francisco: International Computer Music Association, pp. 319–326.

Stilson, T., and J. O. Smith. 1996. "Analyzing the Moog VCF with Considerations for Digital Implementation." In *Proceedings of the 1996 International Computer Music Conference.* San Francisco: International Computer Music Association, pp. 398–401. Also available on the World Wide Web at http://www-ccrma.stanford.edu/~stilti.

Van Duyne, S. A. 1997. "Coupled Mode Synthesis." In *Proceedings of the 1997 International Computer Music Conference.* San Francisco: International Computer Music Association, pp. 248–251.

Van Duyne, S. A., D. A. Jaffe, P. Scandalis, and T. Stilson. 1997. "A Lossless, Click-free, Pitch Bendable Delay Line Loop Interpolation Scheme." In *Proceedings of the 1997 International Computer Music Conference.* San Francisco: International Computer Music Association, pp. 252–255.

Van Duyne, S. A., and J. O. Smith. 1995. "Developments for the Commuted Piano." In *Proceedings of the 1995 International Computer Music Conference.* San Francisco: International Computer Music Association, pp. 335–343.