Synthesis ToolKit in C++ Version 1.0 May, 1996
SIGGRAPH 1996

Perry R. Cook
Department of Computer Science,
Department of Music, Princeton University
prc@cs.princeton.edu

Abstract

This paper describes a collection of roughly 60 (as of May, 1996) classes in C++, designed for the rapid creation and connection of music synthesis and audio processing systems. Primary attention has been paid to cross-platform functionality, ease of use, instructional code examples, and real-time control. The types of objects can be divided into three categories: 1) basic audio sample sources and manipulators called unit generators, 2) musical instrument and audio signal processing algorithms built from unit generators, and 3) control signal and user interface handlers. Instrument synthesis algorithms include additive (Fourier) synthesis, subtractive synthesis, frequency modulation synthesis of various topologies, modal (resonant filter) synthesis, and a variety of physical models including stringed and wind instruments.

1 The C++ Synthesis Toolkit: Motivations

The Synthesis Toolkit in C++ includes many new algorithms and instruments, but it is also a port of most of the algorithms and musical instrument models I have generated over the last decade. These models ran in diverse environments and languages, such as SmallTalk, Lisp, real-time synthesis in Motorola DSP56001 assembler (and connected using the NeXT MusicKit), Objective C, and ANSI C code. The primary motivations for creating the Synthesis Toolkit were a desire for portability, object oriented design and extensibility, and exploiting the increasing efficiency and power of modern host processors, combined with performance improvements of optimizing C compilers. There was also a desire to establish a better framework for implementing many of the "intelligent player" objects as discussed in [Garton 1992][Jànosy 1994][Cook 1995]. Finally, for future research, teaching, and music composition and performance, there was a desire to create a set of examples of different synthesis techniques which wherever possible share a common interface, but allow unique features of each particular synthesis algorithm to be exploited. Sharing a common interface allows for rapid comparisons of the algorithms, and also allows for synthesis to be accomplished in a scaleable fashion, by selecting the algorithm that accomplishes a desired task in the most efficient and/or expressive manner.

The Synthesis Toolkit in C++ is made available freely for academic and research uses via various ftp servers, including Princeton Computer Science, the Princeton Sound Kitchen, and the Stanford Center for Computer Research in Music and Acoustics (CCRMA).

2 Unit Generators

The master class for the entire Synthesis Toolkit is Object.cpp. Little actual work is done in Object.cpp, but all other classes inherit from it. It is thus a convenient place to centralize machine-specific #defines, switches, and some global variables. For example, by defining SGI, NEXT, INTEL, and/or SGI_REALTIME the class RawWvOut.cpp compiles and links appropriately to generate .snd files, .wav files, or stream in real-time to the audio output DACs. Other features and functionality, specifically real-time audio input and output on other platforms, are planned for support in the future.

Audio samples throughout the sytem are floating point (double or float defined as MY_FLOAT for the entire toolkit in the Object.h file), and thus could use any normalization scheme desired. The base instruments and algorithms are implemented with a general dynamic maximim of approximately +/-1.0, and the RawWvOut.cpp class scales appropriately for DAC or sound file output.

All audio sample based unit generators implement a fundamental tick( ) method, which causes the unit generator to do computation. Some unit generators are only sample sources, like the linearly-interpolating oscillator RawLoop.cpp, the simple envelope generator Envelope.cpp, or the RawWvIn.cpp object which allows for sound input streaming. These source-only objects take no argument in their tick( ) function, and return a MY_FLOAT. Other consumer-only objects like the RawWvOut object take a MY_FLOAT argument and return void. Objects like filters, delay lines, etc. both take and yield a MY_FLOAT sample in their tick( ) function. All objects which are sources of audio samples implement a method lastOut( ), which returns the last computed sample. This allows a single source to feed multiple sample consuming objects without neccessitating an interim storage variable. Further, since each object saves its output state in an internally protected variable, bugs arising from accidentally using a shared non-protected "patchpoint" are avoided. Further, it simplifies the process of vectorization as discussed later in this document.

As a simple example, an algorithm will be constructed which reads an input stream from a file, filters it, multiplies it by a time-varying envelope, and writes it out as a file. Here just the constructor (function which creates and initializes unit generators and object variables), and the tick( ) function are shown. For a good beginning reference on C++, consult [Winston 1994].

```
ExampleClass()          {
  envelope =   new Envelope;
  waveIn =     new RawWvIn("infile.raw");
  filter =     new OnePole;
  output =     new RawWvOut("outfile.snd");
}

MY_FLOAT ExampleClass :: tick(void)  {
  output->tick(  envelope->tick( )
      *filter->tick(waveIn->tick( ) ) );
}
```

The base Synthesis Toolkit 1.0 unit generators implement a single-sample tick, that is, tick( ) functions take and/or yield a single sample value. This allows for minimum memory useage, the ability to modularly build very short (one sample) recursive loops, and guaranteed minimum latency through the system. Single sample unit generator calculation, however, is nearly guaranteed to be sub-optimal in terms of computation speed. To address the efficiency issue, the unit generators have been designed to allow for easy vectorization. Vectorized unit generators take and/or yield pointers to arrays of sample values, and improve performance significantly depending on the processor type and vector size. A set of vectorized ToolKit unit generators is planned to be supported as version 1.1v. The vector size will be determined by a #define in the Object.h file, and can be adjusted for tradeoffs of performance, memory useage, and latency requirements.

3 Music Synthesis Algorithms

Algorithms supported in the Synthesis Toolkit include simple oscillator-based additive synthesis, subtractive synthesis, Frequency Modulation non-linear synthesis, modal synthesis, PCM sampling synthesis, and physical modeling. Consult [Mathews and Pierce 1989][Roads 1996] and [Stieglitz 1996] for more information on digital audio processing and music synthesis. Additive analysis/ synthesis, also called Fourier synthesis, is covered in [McAulay and Quatieri 1986][Smith and Serra 1987], and elsewhere in these proceedings by [Serra 1996]. In subtractive synthesis, a complex sound is filtered to shape the spectrum into a desired pattern. The most popular forms of subtractive synthesis in computer music involve the phase and channel VoCoder (voice coder)[Dudley 1939][Moorer 1978][Dolson 1986], and Linear Predictive Coding (LPC) [Atal 1970] [Makhoul 1975] [Moorer 1979][Steiglitz and Lansky 1981], Frequency Modulation synthesis [Chowning 1973 and 1981] and WaveShaping [LeBrun 1979] employ non-linear warping of basic functions (like sine waves) to create a complex spectrum. Modal synthesis models individual physical resonances of an instrument using resonant filters, excited by parametric or analyzed excitations [Adrien 1988][Wawrzynek 1989] [Larouche 1994]. Physical models endeavor to solve the physics of instruments in the time-domain, typically by numerical solution of the differential traveling wave equation, to synthesize sound [Smith 1987][Karjalainen et. al. 1991][Cook 1991 and 1992][McIntyre et. al. 1983][CMJ 1992-3].

Given the author's legacy in synthesis of the singing voice, the Synthesis Toolkit Version 1.0 provides multiple models of the voice, and more vocal synthesis models are planned for the future. References on voice synthesis using subtractive, FM, and physical modeling include [Kelly and Lochbaum 1962] [Rabiner 1968] [Klatt 1980] [Chowning 1981] [Carlson et. al 1990] [Cook et. al 1991b, 1992b, 1993][Maher 1995].

4 Audio Effects Algorithms

The Synthesis Toolkit includes a few simple delay-based effects such as reverberation (modeling of sound reflections in rooms), chorus effect (simulating the effect of multiple sound sources from a single sound), and flanging (time-varying delay mixed with direct sound). See [Moorer 1979b] and the book by [Roads 96] for more details on reverberation and effects processing.

5 SKINI: Yet Another "Better" MIDI ?

To support a unified control interface across multiple platforms, multiple control signal sources such as GUIs of multiple flavors, MIDI controllers and score files, and to support connection between processes on a single machine and across networks, a simple extension to MIDI was created and imbedded into the Synthesis Toolkit. Other more sophisticated protocols for music control have been proposed and implemented [CMJ 1994], but the Toolkit introduces and uses a simple but extensible protocol called SKINI. SKINI (Synthesis toolKit Instrument Network Interface) extends MIDI in incremental ways, specifically in representation accuracy by allowing for floating point note numbers (microtuning), floating point control values, and double precision time stamps and delta-time values. Further, an easily tokenizable text basis for the control stream is used, to allow for easy creation of SKINI files and debugging of SKINI control consumers and providers. Finally, SKINI goes beyond MIDI in that it allows for parametric control curves to be specified and used. This allows continuous control streams to be potentially lower in bandwidth than MIDI (hence part of the name SKINI), yet higher in resolution and quality because the control functions are "rendered" in the instrument and/or in a performer-expert class which controls the instrument. Expressive figures like trills, drum rolls, characteristic pitch bends, heavy-metal guitar hammer-ons, etc. can all be specified and called up using text symbols. To support SKINI scorefiles, the Toolkit provides MIDIText.cpp, which reads SKINI files and controls instrument synthesis and effects. MIDIInpt.cpp is a real-time MIDI input handler. testMIDI.cpp imbeds a MIDIInpt.cpp object and converts the MIDI stream to SKINI for realtime control. Read the Toolkit documentation file SKINI.txt for information on the SKINI format and new features as they develop.

6 GUIs and JAVA

In keeping with cross-platform support and compatibility, simple Graphical User Intefaces for Synthesis Toolkit instruments have been implemented in Tcl/TK [Welch 1995]. All classes in the Synthesis Toolkit have been ported to JAVA [Flanagan 1996], but the execution speed as of this writing is too slow to be useful. Interfaces which function like the Tcl/TK controllers have also been constructed in JAVA. When JAVA compilers or faster JAVA interpreters become available, and when base audio support at useful musical sampling rates is provided in JAVA, the JAVA Synthesis Toolkit will be made available via ftp.

Acknowledgements

Many ideas for the Synthesis Toolkit and the algorithms came from people at CCRMA, specifically Julius Smith, John Chowning, and Max Mathews. Dexter Morrill and Chris Chafe have been instrumental in forming some of my opinions

on controllers and protocols for real-time synthesis. The NeXT MusicKit and ZIPI have also been highly inspirational and instructive. Ted Huffmire at Princeton University ported the ToolKit to JAVA, and suffered through the experiment of determining that vectorization does little to improve efficiency in current JAVA interpreters.

References

Adrien, J. 1988. *Etude de Structures Complexes Vibrantes, Application-la Synthèse par Modeles Physiques*, Doctoral Dissertation. Paris: Université Paris VI.

Atal, B. 1970. "Speech Analysis and Synthesis by Linear Prediction of the Speech Wave." *Journal of the Acoustical Society of America* 47.65(A).

Carlson, G. and L. Neovius 1990. "Implementations of Synthesis Models for Speech and Singing," STL-Quarterly Progress and Status Report, KTH, Stockholm, Sweden, 2-3: pp. 63-67.

Chowning, J. 1973, "The Synthesis of Complex Audio Spectra by Means of Frequency Modulation," *Journal of the Audio Engineering Society* 21(7): pp. 526-534.

Chowning, J. 1981, "Computer Synthesis of the Singing Voice," in *Research Aspects on Singing*, KTH, Stockholm, Sweden, pp. 4-13.

CMJ 1992-3. *Computer Music Journal* Special Issues on Physical Modeling, 16(4) and 17(1).

CMJ 1994. *Computer Music Journal* Special Issue on ZIPI, 18(4).

Cook, P. 1991. "TBone: An Interactive Waveguide Brass Instrument Synthesis Workbench for the NeXT Machine," *Proc. International Computer Music Conference*, Montreal, pp. 297-299.

Cook, P. 1991b. "LECTOR: An Ecclesiastical Latin Control Language for the SPASM/singer Instrument," *Proc. International Computer Music Conference*, Montreal, pp. 319-321.

Cook, P. 1992. "A Meta-Wind-Instrument Physical Model, and a Meta-Controller for Real-Time Performance Control," *Proc. International Computer Music Conference*, San Jose, pp. 273-276.

Cook, P. 1992b. "SPASM: a Real-Time Vocal Tract Physical Model Editor/Controller and Singer: the Companion Software Synthesis System," Colloque les Modeles Physiques Dans L'Analyse, la Production et la Creation Sonore, ACROE, Grenoble, 1990, published in *Computer Music Journal*, 17: 1, pp 30-44.

Cook, P., D. Kamarotos, T. Diamantopoulos, and G. Philippis, 1993. "IGDIS: A Modern Greek Text to Speech/Singing Program for the SPASM/Singer Instrument," *Proceedings of the International Computer Music Conference*, Tokyo, pp. 387-389.

Cook, P. 1995. "A Hierarchical System for Controlling Synthesis by Physical Modeling," *Proc. International Computer Music Conference*, Banff, pp. 108-109.

Dolson, M. 1986, "The Phase Vocoder: A Tutorial," *Computer Music Journal*, 10 (4), pp. 14 -27.

Dudley, H. 1939, "The Vocoder," *Bell Laboratories Record*, December.

Flanagan, D. 1996. *Java in a Nutshell*, Sebastopol, CA, O'Reilly and Associates.

Garton, B. 1992. "Virtual Performance Modeling," *Proc. International Computer Music Conference*, Montreal, pp. 219-222.

Jànosy, Z., Karjalainen, M. & V. Välimäki 1994, "Intelligent Synthesis Control with Applications to a Physical Model of the Acoustic Guitar," *Proc. International Computer Music Conference*, Aarhus, pp. 402-406.

Karjalainen, M. Laine, U., Laakso, T. and V. Välimäki, 1991. "Transmission Line Modeling and Real-Time Synthesis of String and Wind Instruments," *Proc. International Computer Music Conference*, Montreal, pp. 293-296.

Kelly, J., and C. Lochbaum. 1962. "Speech Synthesis." *Proc . Fourth Intern. Congr. Acoust.* Paper G42: pp. 1-4.

Klatt, D. 1980. "Software for a Cascade/Parallel Formant Synthesizer," *Journal of the Acoustical Society of America* 67(3), pp. 971-995.

Larouche, J. & J. Meillier 1994. "Multichannel Excitation/ Filter Modeling of Percussive Sounds with Application to the Piano," *IEEE Trans. Speech and Audio*, pp. 329-344.

LeBrun, M. 1979. "Digital Waveshaping Synthesis," *Journal of the Audio Engineering Society*, "27(4): 250-266.

Maher, R. 1995, "Tunable Bandpass Filters in Music Synthesis," *Audio Engineering Society Conference.* Paper Number 4098(L2).

Makhoul, J. 1975. "Linear Prediction: A Tutorial Review," *Proc. of the IEEE*, v 63., pp. 561-580.

Mathews, M. and J. Pierce. 1989. *Some Current Directions in Computer Music Research.* Cambridge MA.: MIT Press: 57-63.

McIntyre, M., Schumacher, R. and J. Woodhouse 1983, "On the Oscillations of Musical Instruments," *Journal of the Acoustical Society of America*, 74(5), pp. 1325-1345.

McAulay, R. and T. Quatieri. 1986. "Speech Analysis/Synthesis Based on a Sinusoidal Representation." *IEEE Trans. Acoust. Speech and Sig. Proc*. ASSP-34(4): pp. 744-754.

Moorer, A. 1978. "The Use of the Phase Vocoder in Computer Music Applications." *Journal of the Audio Engineering Society*, 26 (1/2), pp. 42-45.

Moorer, A. 1979, "The Use of Linear Prediction of Speech in Computer Music Applications," *Journal of the Audio Engineering Society* 27(3):134-140.

Moorer, A. 1979b, "About This Reverberation Business," *Computer Music Journal*, 3(2), pp. 13-28.

Rabiner, L. 1968. "Digital Formant Synthesizer" *Journal of the Acoustical Society of America* 43(4), pp. 822-828.

Roads, C. 1976, *the computer music tutorial*, Cambridge, MIT Press.

Serra, X. 1996, Papers, Notes, and References Elsewhere in These Proceedings.

Smith, J. 1987. *Musical Applications of Digital Waveguides.* Stanford University Center For Computer Research in Music and Acoustics. Report STAN-M-39.

Smith, J. and Serra, X. 1987. "PARSHL: Analysis/Synthesis Program for Non-Harmonic Sounds Based on a Sinusoidal Representation." *Proc. International Computer Music Conference*, Urbana, pp. 290 - 297.

Steiglitz, K. and P. Lansky 1981. "Synthesis of Timbral Families by Warped Linear Prediction." *Computer Music Journal* 5(3): 45-49.

Steiglitz, K. 1996 *Digital Signal Processing Primer,* New York, Addison Wesley.

Wawrzynek, J. 1989. "VLSI Models for Sound Synthesis," in *Current Directions in Computer Music Research*, M. Mathews and J. Pierce Eds., Cambridge, MIT Press.

Welch, B. 1995. *Practical Programming in Tcl and Tk*, Saddle River, NJ, Prentice-Hall.

Winston, P. 1994, *On to C++*, New York, Addison-Wesley.

Appendix 1: Unit Generators

Master Object:	Object.cpp	Compatibility with Objective C, and shared global functionality
Source&Sink:	RawWave.cpp	Lin-Interp Wavetable, Looped or 1 Shot
	NIWave1S.cpp	Non-Interp Wavetable, 1 Shot
	RawLoop.cpp	Lin-Interp Wavetable, Looping
	RawWvIn.cpp	Lin-Interp Wave In streaming 'device'
	NIFileIn.cpp	Non-Interp Wave In streamer, closes & opens
	RawWvOut.cpp	Non-Interp Wave Out streaming 'device'
	Envelope.cpp	Linearly Goes to Target by Rate, plus noteOn/Off
	ADSR.cpp	ADSR Flavor of Envelope
	Noise.cpp	Random Number Generator
	SubNoise.cpp	Random Numbers each N samples
Filters:	Filter.cpp	Filter Master Class
	OneZero.cpp	One Zero Filter
	OnePole.cpp	One Pole Filter
	AllPass1.cpp	1st Order All-Pass (phase) Filter
	DCBlock.cpp	DC Blocking 1Pole/1Zero Filter
	TwoZero.cpp	Two Zero Filter
	TwoPole.cpp	Two Pole Filter
	BiQuad.cpp	2Pole/2Zero Filter
	FormSwep.cpp	Sweepable 2Pole filter, go to Target by Rate
	DLineL.cpp	Linearly Interpolating Delay Line
	DLineA.cpp	AllPass Interpolating Delay Line
	DLineN.cpp	Non Interpolating Delay Line
NonLinear:	JetTabl.cpp	Cubic Jet NonLinearity
	BowTabl.cpp	x^(-3) Bow NonLinearity
	ReedTabl.cpp	1 Break Point Saturating Linear Reed NonLinearity
	LipFilt.cpp	Pressure Controlled BiQuad with NonLinearity
Derived:	Modulatr.cpp	Per. and Rnd. Vibrato: RawWave,SubNoise,OnePole
	SingWave.cpp	Looping Wavetable with: Modulatr,Envelope

Appendix 2: Algorithms and Instruments

Each Class will be listed either with all UGs it uses, or the <<Algorithm>> of which it is a flavor. All inherit from Instrmnt, which inherits from Object.

Plucked.cpp	Basic Plucked String	DLineA,OneZero,OnePole,Noise
Plucked2.cpp	Not so Basic Pluck	DLineL,DlineA,OneZero
Mandolin.cpp	Commuted Mandolin	<<flavor of PLUCKED2>>
Bowed.cpp	So So Bowed String	DlineL,BowTabl,OnePole,BiQuad,RawWave,ADSR
Brass.cpp	Not So Bad Brass Inst.	DLineA,LipFilt,DCBlock,ADSR,BiQuad
Clarinet.cpp	Pretty Good Clarinet	DLineL,ReedTabl,OneZero,Envelope,Noise.h
Flute.cpp	Pretty Good Flute	JetTabl,DLineL,OnePole,DCBlock,Noise,ADSR,RawWave
Modal4.cpp	4 Resonances	Envelope,RawWave,BiQuad,OnePole
Marimba.cpp		<<flavor of MODAL4>>
Vibraphn.cpp		<<flavor of MODAL4>>
Agogobel.cpp		<<flavor of MODAL4>>
FM4Op.cpp	4 Operator FM Master	ADSR,RawLoop,TwoZero
FM4Alg3.cpp	3 Cascade w/ FB Mod.	<<flavor of FM4OP>>
FM4Alg4.cpp	Like Alg3 but diff.	<<flavor of FM4OP>>
FM4Alg5.cpp	2 Parallel Simple FMs	<<flavor of FM4OP>>
FM4Alg6.cpp	3 Carriers share 1 Mod.	<<flavor of FM4OP>>
FM4Alg8.cpp	4 Osc. Additive	<<flavor of FM4OP>>
HeavyMtl.cpp	Distorted FM Synth	<<flavor of FM4Alg3>>
PercFlut.cpp	Perc. Flute	<<flavor of FM4Alg4>>
Rhodey.cpp	Rhodes-Like Elec. Piano	<<flavor of FM4Alg5>>
Wurley.cpp	Wurlitz. Elec. Piano	<<flavor of FM4Alg5>>
TubeBell.cpp	Classic FM Bell	<<flavor of FM4Alg5>>
FMVoices.cpp	3 Formant FM Voice	<<flavor of FM4Alg6>>
BeeThree.cpp	Cheezy Additive Organ	<<flavor of FM4Alg8>>
Sampler.cpp	Sampling Synth.	4 each ADSR, RawWave (att), RawWave (loop), OnePole
SamplFlt.cpp	Sampler with Swept Filt.	<<flavor of Sampler>>
Moog1.cpp	Swept filter flavor of	<<flavor of SamplFlt>>
Voicform.cpp	Source/Filter Voice	Envelope,Noise,SingWave,FormSwep,OnePole,OneZero
DrumSynt.cpp	Drum Synthesizer	bunch of NIFileIn, and OnePole

Reverb.cpp	Reverberator Effects Processor	Four DLineN, Used as 2 Allpass and 2 Comb Filters.
Flanger.cpp	Flanger Effects Processor	One DLineL, One RawLoop
Chorus.cpp	Chorus Effects Processor	Two DLineL, Two RawLoop