MUSIC GRAPHS FOR ALGORITHMIC COMPOSITION AND SYNTHESIS WITH AN EXTENSIBLE IMPLEMENTATION IN JAVA

Michael Gogins
150 West 95th Street, New York NY 10025
http://www.pipeline.com/~gogins

ABSTRACT: This paper describes Music Modeling Language, a new representation of music, and Silence, a Java program that uses MML for composition and synthesis. MML consists of nodes such as notes, MIDI files, random sequences, generative algorithms, time/frequency analyses, affine transformations, and even groups of nodes nested hierarchically in directed acyclic graphs, as in Virtual Reality Modeling Language. Music graphs are rendered by depth-first traversal to produce scores in 11-dimensional music space, containing notes and/or time/frequency data. Scores are then rendered by Csound to produce soundfiles. Because Silence loads all node classes at run-time, MML is extensible with plugin nodes.

1. Introduction

Music Modeling Language was born out of love for, and frustration with, algorithmic composition. My first explorations of composition using iterated function systems (Gogins 1991, 1992a) and recursive fractal constructions (Gogins 1992b) produced interesting music from modest programming efforts. As time went on, however, the ratio of composition time to programming time dwindled exponentially, as I lost myself, again and again, in the swamps of software development, despite having learned enough programming to make my living at it. Over many cycles of development, some causes of this dilemma became clear... along with some solutions.

- *Inconsistent representations*: Each compositional algorithm is based on some formal representation of music. Every algorithm I implemented seemed to demand a slightly different representation. *Solution*: Use one representation of music for all compositional algorithms, an 11-dimensional linear space I call *music space*.
- *Incommensurate materials*: It became frustrating that material produced by different algorithms could not be assembled into a piece without manual splicing. *Solution*: Adapt the concept of "hierarchical scene graphs" from computer graphics (van Dam and Sklar 1991) to create *music graphs*, replacing 3-dimensional visual space with 11-dimensional music space, replacing visual primitives such as polygons and textures with acoustical primitives such as notes and time/frequency analyses, and using affine transformations to assemble scores by placing the notes and sounds produced by different nodes at their intended locations in music space.
- *Duplication of code*: Compositional algorithms are useless without rapid edit/generate/listen cycles. I've found this easier to provide with visual editors than with computer languages. However, each new algorithm I implemented turned into its own complete application, involving a great deal of repetitive work. *Solution*: Create a single application *framework* with a shared visual editor and utility functions that can be used by many different algorithms, each one being a type of node in music graphs, which are the unifying protocol.
- *Constant relinking*: In computer music, new algorithms appear frequently. For each one I implemented, I had to relink my whole system, often modifying the framework's source code. This became more and more difficult as time went on. *Solution*: Create an abstract interface for nodes in music graphs, `INode`, and give the framework the ability to dynamically identify and load new `INode` classes, or *plugin nodes*, at run-time.
- *Incompatible platforms*: I've always used Unix-based, academically-produced composition and synthesis software such as Cmix (Garton 1993) and Csound (Vercoe 1997, Piché 1998) together with Windows-based MIDI sequencers and soundfile editors such as Cubase (Steinberg 1998) and Cool Edit (Syntrillium 1996). However, I don't like the idea of segregated worlds of music software that don't talk to each other. *Solution:* Enable Silence to read and write standard file formats like MIDI files and WAV soundfiles, and program it in Java (Sun 1998) so it runs on Unix, the Macintosh, and Windows.

2. Music Space

It may be helpful, for a better understanding of the motivation for music space and music graphs, to briefly review some aspects of algorithmic composition and synthesis. Historically, software *synthesis* systems refrained from defining low-level representations of music because they must provide arbitrary arguments, usually variable-length

lists of floating-point numbers, to user-defined software instruments (Pope 1993). Software for *notation* or *analysis* uses representations derived from traditional music theory, all grammar-based (Selfridge-Field 1997). Software *composition* systems have a complex history (see pages 783-909 in Roads 1996), with many algorithms borrowed from mathematics and science. The most impressive systems adapt the λ-calculus to make music by composing functions that implement a whole gamut of algorithms (Taube 1991, Scaletti and Hebel 1991, Tonality Systems 1993), use random variables to generate material (Xenakis 1992), or produce music by matching patterns abstracted from a style (Cope 1991, 1996).

To the best of my knowledge, every one of these systems refrains from fixing the dimensionality of its low-level representation of music. This has the advantage of allowing the composer every liberty, but there is a corresponding disadvantage. Each different algorithm, piece of material, or finished work tends to live in its own peculiar space. They cannot effortlessly be imagined together in the mind, or manipulated together in software.

As I developed compositional algorithms, including random sequences, chaotic dynamical systems, and iterated function systems (Gogins 1991), the value for representing music of linear *spaces* (as opposed to *languages*) became clear. Even implementing a small language, musical Lindenmayer systems (Gogins 1992b), was simpler with spaces. It is just easier to manipulate music with algebra than with procedural code. And if time, pitch, and loudness have infinite gradations, a single space can be used to represent not only "notes," but also "sounds." Then a time/frequency representation of sound, such as a heterodyne filter analysis, can be translated into a large number of elementary notes. This notion was implicit in Xenakis' first intuition of granular synthesis (1992, p. 43).

After I had implemented many algorithms, the dimensions and units of music space stabilized as shown in Table 1, indicating that some sort of fit had been achieved between reality and representation (at least, for geometrically-based algorithmic composition!). All units are more or less psychologically linear; that is, a doubling of octave produces an apparent doubling of pitch, and a doubling of decibels produces an apparent doubling of loudness.

Table 1. Dimensions and Units of Music Space.

ID	Analogous to the high-order nybble of the MIDI status byte (MIDI Manufacturers Association 1996); represents not only notes, but also control information. All notes have ID 144.
Instrument	Analogous to the low-order nybble of MIDI status byte, or MIDI channel; assigns the note to an instrument.
Time	Time in seconds.
Duration	Duration in seconds.
Octave	Pitch in linear octaves, where middle C = 8.0.
Decibels	Loudness in decibels, where 0 = no energy and the maximum for 16-bit samples = 96.
Phase	Phase in units of π; intended for future phase vocoder, wavelet, or granular representations.
X	The horizontal dimension of spatial location in meters, where the listener is at the origin.
Y	The vertical dimension of spatial location in meters, where the listener is at the origin.
Z	The depth dimension of spatial location in meters, where the listener is at the origin.
1	Unity, which makes music space homogeneous.

Each note or grain of sound is a column vector in a homogeneous linear music space, so a set of notes, even a whole score, is a tensor. Here's a four-note theme, $\boldsymbol{A}$:

$$\begin{vmatrix} 144 & 144 & 144 & 144 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 4 & 6 \\ 1 & 2 & 2 & 1 \\ 47/6 & 31/4 & 8 & 95/12 \\ 60 & 70 & 70 & 60 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{vmatrix}$$

Given a tensor ***A*** of points in music space, any affine transformation ***T*** of ***A*** can be accomplished by a single inner product with the homogeneous matrix operator ***B***: ***T***(***A***) ≡ ***BA***. Only those dimensions of the operator whose values are not all identity will affect the notes. For example, to transpose the theme ***A*** up a perfect fourth, delay it 1 second, double the tempo, reduce the loudness by 10 decibels, compress the dynamic range by 3/4, and move it to instrument 2, but without affecting phase or spatial location in any way, the following matrix ***B*** can be used:

1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1/2	0	0	0	0	0	0	0	1
0	0	0	1/2	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0	5/12
0	0	0	0	0	3/4	0	0	0	0	-10
0	0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0	0	1

Then ***BA*** =

144	144	144	144
2	2	2	2
1	3/2	3	4
½	1	1	½
33/4	49/6	101/12	25/3
35	85/2	85/2	35
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
1	1	1	1

Because all discrete operations (such as on pitch-class sets) *can* be represented in a *continuous* space, while continuous operations (for example on spatial position) *cannot* be represented in *discrete* spaces, it makes sense for the general space to be continuous.

In sum, by accommodating both notes and time/frequency representations of sounds in 11 psychologically linear dimensions, music space solves the problem of inconsistent representations.

3. Music Graphs

Once the concept of music space had become clear, its analogy with the visual space of 3-dimensional computer graphics was inescapable, and acquaintance with VRML (International Organization for Standardization 1997, Ames 1997) suggested a solution for the problem of incommensurate materials. Music could be represented not directly as tensors in music space, but, by analogy with *hierarchical scene graphs* in geometrical modeling (van Dam and Sklar 1991), as directed acyclic graphs whose *nodes* are elements of music... *music graphs*.

The elements might of course be individual notes, but they also could be pre-existing pieces or fragments of pieces of music, random sequences, generative algorithms, compositional transformations, or even other nodes, nested to any depth (but no node may contain itself). Then any work of music can be modeled using simpler elements (nodes) that are assembled hierarchically by geometric transformations (other nodes). Because a node might be used by reference in several different locations and under various transformations, music graphs can often be considerably more concise than the music they represent. For example, Figure 1 shows the tune ***Pange Lingua***:

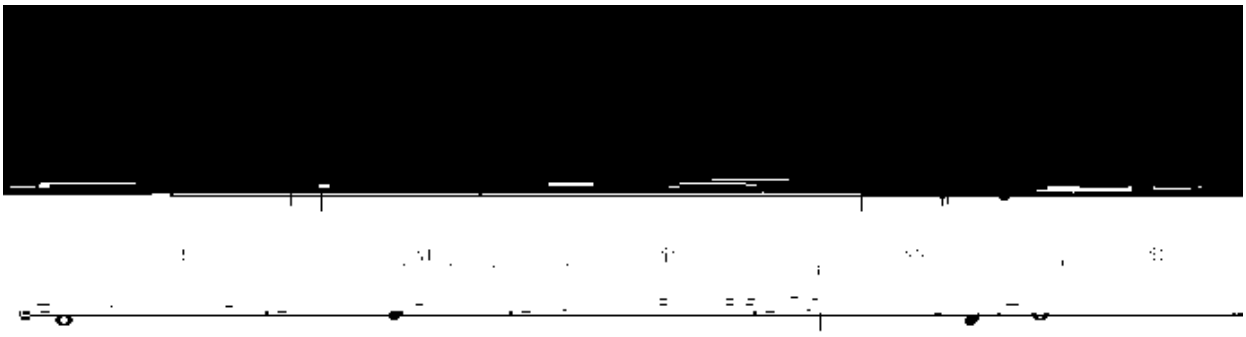

Figure 1. Pange Lingua.

Figure 2 shows the MML editor of Silence, containing a music graph that elaborates this tune.

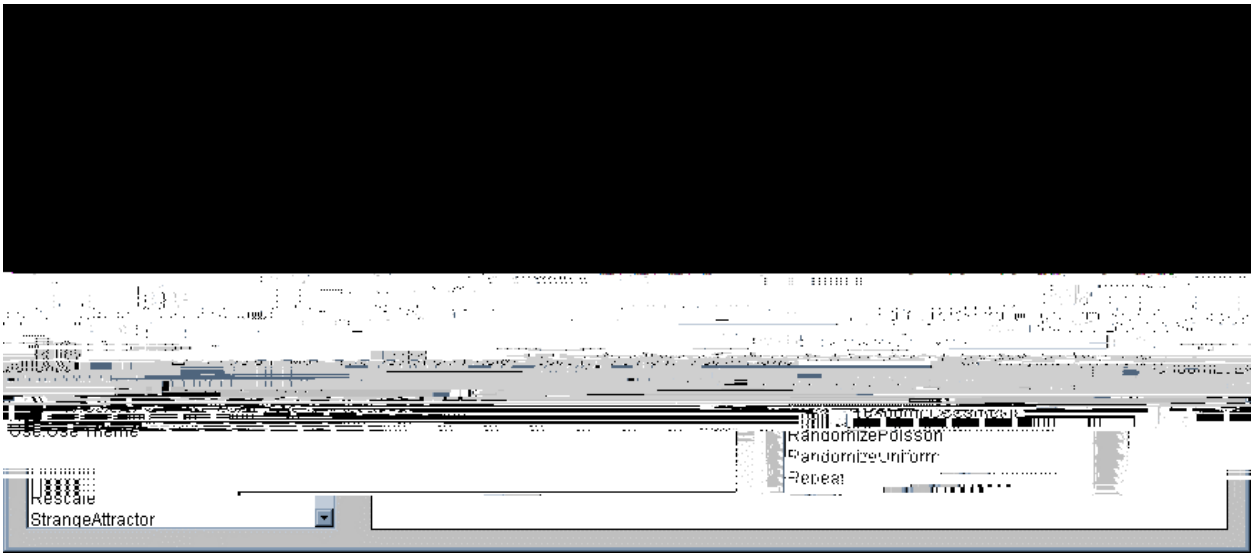

Figure 2. Silence's Music Graph Editor.

The `Definition` node contains a sub-graph that does not itself produce notes, while `Use` nodes produce the contents of the named `Definition`, only at their own locations in music space. These nodes are the same as `DEF` and `USE` in VRML (Ames et al. 1996, pp. 16-17).

`Rescale` nodes perform absolute transformations. `Progression` nodes adjust the pitch-classes of child nodes to fit into a specified pitch-class set, or progression of them. `Repeat` does just that, repeating all notes produced by its child nodes at specified intervals. `Transform` nodes perform transformations of the coordinate systems of their child nodes, in this case copies of the theme, relative to the coordinate system of the parent. The graph in Figure 2 produces a three-part canon in double time over the theme in the bass (Figure 3).

At the time of writing, Silence has nodes for notes, heterodyne filter analyses, all affine transformations, conforming nodes to given pitch-class sets, rescaling nodes, quantizing nodes on any dimension, repeating nodes, and a number of generative nodes. These include random sequences, recurrent iterated function systems (Gogins 1991), Lindenmayer systems (Gogins 1992b), and chaotic dynamical systems with automatic searching for strange attractors using the Lyupanov exponent (Sprott 1994).

Figure 3. Production of a Music Graph.

In syntax and textual representation, MML is a dialect of Extensible Markup Language (XML) (World Wide Web Consortium 1998). Each element of MML is prefixed by a start tag and postfixed by an end tag. However, MML is simpler than XML. Tags never have attributes; all start and end tags are followed by newlines; all other lines of text contain either a property name and value, or a vector of numbers; a matrix is several rows of vectors; and files never have Document Type Definitions, because MML is designed to be dynamically extensible.

The following MML produces, from a dense heterodyne filter analysis of a stainless steel mixing bowl struck with a wooden spoon, three tones. The first is simply a resynthesis of the original sound. The second tone is stretched out in time and lowered in pitch, but a shear transformation (the 0.5 at row 5, column 3) causes it to rise steadily in pitch. The third is shorter in time and higher in pitch.

```
<Group>
name = root
<HeterodyneFilterAnalysis>
name = Bowl
soundFilename = C:\Gogins\Music\Album\FromPangeLingua\Bowl.wav
channel = 1
startSeconds = 0.0
durationSeconds = 0.0
fundamentalHz = 50.0
partialCount = 200
maximumTotalAmplitude = 32767
amplitudeThreshold = 64
initialBreakpointCount = 300
lowpassHz = 0.0
instrumentIndex = 1
useExistingAnalysis = true
</HeterodyneFilterAnalysis>
<Transform>
```

```
name = Move up 3 instruments, delay 0.875 seconds, 4 times longer, 2 octaves lower, glissando
    up 0.5 octave
[1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0]
[0.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 3.017392]
[0.0 0.0 4.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.875]
[0.0 0.0 0.0 4.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0]
[0.0 0.0 0.5 0.0 1.0 0.0 0.0 0.0 0.0 0.0 -2.0]
[0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0 0.0]
[0.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0]
[0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0]
[0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0]
[0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0]
[0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0]
<Use symbol = Bowl>
</Transform>
<Transform>
name = Move up 1 instrument, delay 6 seconds, twice as fast, 1.25 octaves higher
[1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0]
[0.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0]
[0.0 0.0 0.5 0.0 0.0 0.0 0.0 0.0 0.0 0.0 6.0]
[0.0 0.0 0.0 0.5 0.0 0.0 0.0 0.0 0.0 0.0 0.0]
[0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0 0.0 1.25]
[0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0 0.0]
[0.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0]
[0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0]
[0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0]
[0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0]
[0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0]
<Use symbol = Bowl>
</Transform>
</Group>
```

Music graphs are rendered in two passes. In the first pass, the `INode.traverseMusicGraph` function calls itself recursively down to each leaf node of the graph, storing in `score` all notes that are produced:

```
public double[][] traverseMusicGraph(double[][] parentTransformation, INode score)
{
    //  Record the number of notes in the score before traversing any child nodes of this.
    int preTraversalCount = score.getChildNodeCount();
    //  The local transformation of coordinate systems is identity fo r non-transformation
    //  nodes, or some other affine transformation for transformation nodes.
    double[][] compositeTransformation = Matrix.dot(parentTransformation,
        getLocalTransformation());
    //  Iterate over the immediate child nodes of this...
    for(int i = 0, n = getChildNodeCount(); i < n; i++)
    {
        //  ...calling traverseMusicGraph on each one. This has the effect
        //  of performing a recursive, depth-first traversal of the music graph.
        ((INode) getChildNode(i)).traverseMusicGraph(compositeTransformation, score);
    }
    //  Record the number of notes in the score after traversing the child nodes.
    //  This number will have increased by the number of notes produced by all child nodes.
    int postTraversalCount = score.getChildNodeCount();
    //  Finally, either produce new notes, or transform all notes
    //  that were produced by the child nodes of this; each node overrides this function.
    return produceOrTransformNotes(compositeTransformation, score, preTraversalCount,
        postTraversalCount);
}
```

In the second pass, the notes in the `score` are rendered by a software synthesizer, Csound.

Again, how do music graphs compare with functional programming systems like Common Music (Taube 1991) or Kyma (Scaletti and Hebel 1991)? *Both* approaches depend on *functional composition*: of predicates in functional programming, of geometric transformations in MML. However, fixing the dimensionality of the basic representation makes music graphs indefinitely extensible. New nodes can be created at any time that will not break old nodes, because music produced by new nodes will always fit together with music produced by older nodes. Unrolling procedural loops and conditions into nested declarative blocks also makes the musical operations more explicit to the imagination. Finally, it is easier to generate completely declarative code (as opposed to procedural code) using other software, which does not need to maintain any context for the generated code. Although the greater flexibility of procedural languages is lost, the advantages of declarative languages and fixed dimensionality are apparent in virtual reality, whose worlds are created with visual editors that automatically

generate declarative code. Of course, if procedural code is needed for some operation that can't be performed with existing nodes, the thing to do is write it in Java (more widely known than Lisp or Smalltalk) as a new `INode`.

The main difference, then, is that Silence encapsulates abstract models of compositional process within nodes, and is designed for composing quite rapidly via interactive editing of an indefinitely extensible library of nodes.

In sum, because music space can represent all materials of interest, and any manipulation of them can be performed as a hierarchy of linear operations, music graphs solve the problem of incommensurate materials.

4. The Silence Framework

The MML editor of Silence is shown in Figure 1. The left pane lists all the nodes that have been dynamically loaded. The right pane displays the music graph in the form of a expanding or collapsing tree widget. Nodes can be inserted, cut, copied, pasted, and deleted, as well as imported and exported to and from text files. Clicking the right mouse button on a node instance opens a window for editing its attributes.

Each `INode` class must implement certain methods, and provide its own window for editing itself. But the `Silence.Framework` package has a base class for nodes, classes for parsing MML, loading classes, matrix algebra, dimensional conversions, MIDI files, soundfiles, and managing Csound and other external programs. This framework is shared by all nodes, and goes a long way towards solving the problem of constantly rewriting code.

5. Extending MML

The architects of VRML provided a facility for defining whole new types of nodes in terms of existing nodes (the PROTO and EXTERNPROTO nodes (Ames et al. 1991, pp. 603-622). This facility, together with the capabilities of the Java class loader, suggested how to implement *plugin nodes* for music graphs.

The mechanism is simple. Every node in a music graph is represented as an element of XML, with a start tag and an end tag, and each type of node/element is implemented by a Java class with exactly the same name. This convention enables the Silence parser, when it encounters the start tag for an unknown node, to dynamically search for, identify, and load the corresponding class. Once the class is loaded, because it is required to implement the abstract `INode` interface, its `INode.read()` function can be called to parse the remaining text, and its `INode.traverseChildNodes()` and `INode.produceOrTransformNotes()` functions can be called to render the node, even if the class was written and compiled long after the parser. In fact, in Silence, *all* nodes are external, plugin nodes.

The run-time loading of `INode` plugins solves the problem of constant relinking, and enables musicians to exchange independently developed plugins on the basis of a standard yet extensible protocol based on the `INode` interface and MML syntax. It is a *cumulatively extensible* protocol.

Silence could be extended in several directions. The most obvious is to add more generative nodes using new compositional algorithms. To the existing representation of sound, heterodyne filter analyses, might be added a phase vocoder representation, a wavelet representation, or other granular representations. Silence already has MIDI file output, so when Java gains the required capabilities it should be possible to add real-time MIDI output with synchronized input, turning the system into a MIDI sequencer whose sequences are nodes in music graphs. Finally, and perhaps most interestingly, Silence could be extended as a visual composing environment, taking cues from UPIC (Xenakis 1992, pp. 329-334) and the newer Metasynth (Aikin 1998, Wenger 1998). The irreducible quantum of information in sound is the Gabor (1946) logon, the elementary grain. Every logon in an extended work of music could be distinctly pictured in a virtual space the apparent size of a barn interior. It would be fascinating to use generative and transformative nodes to make visual tools, similar to those in Adobe PhotoShop or VRML editors, for creating, sculpting, and filtering those grains in the interactive composition of sound.

6. References

Aikin, Jim. April 1998. "U&I Metasynth (Mac): A Radical New Audio Paintbox for the Mac***," Keyboard Magazine***.

Ames, Andrea L., David R. Nadeau, and John L. Moreland. 1997 [1996]. ***VRML 2.0 Sourcebook, 2nd Edition***, John Wiley & Sons, New York.

Cope, David. 1991. ***Experiments in Musical Style.*** A-R Editions, Madison, Wisconsin.

Cope, David. 1996. ***Experiments in Musical Intelligence.*** A-R Editions, Madison, Wisconsin.

Gabor, Denis. 1946. "Theory of Communication," The Journal of the Institution of Electrical Engineers, Part III, 93: 429-457.

Garton, Brad. 1993. "Using C-Mix," ***Array 13***(2):23-24, (3):14-15, (4):23-27.

Gogins, Michael. 1991. "Iterated Functions Systems Music," ***Computer Music Journal 15***(1):40-48.

Gogins, Michael. 1992a. "How I Became Obsessed with Finding a Mandelbrot Set for Sounds," ***News of Music 13***:129-139.

Gogins, Michael. 1992b. "Fractal Music with String Rewriting Grammars," ***News of Music 13***:146-170.

Gogins, Michael. 1995. "Gabor Synthesis of Recurrent Iterated Function Systems," in ***Proceedings of the 1995 International Computer Music Conference***, ICMA, San Francisco, pp. 349-348

International Organization for Standardization. 1998 [1997]. ***ISO/IEC 14772-1:1997, the Virtual Reality Modeling Language (VRML).*** http://www.vrml.org/Specifications/

MIDI Manufacturers Association. 1996. Complete MIDI 1.0 Detailed Specification.

Piché, Jean (ed.), Barry Vercoe, and contributors. 1998 [1986]. The Csound Manual (version 3.48): A Manual for the Audio Processing System and Supporting Programs with Tutorials by Barry Vercoe, Media Lab, MIT. http://www.leeds.ac.uk/music/Man/c_front.html.

Pope, Stephen Travis. 1993. "Machine Tongues XV: Three Packages for Software Sound Synthesis," ***Computer Music Journal 17***(2):23-54.

Roads, Curtis with John Strawn, Curtis Abbott, John Gordon, and Philip Greenspun. 1996. ***The Computer Music Tutorial***. Cambridge, Massachusetts: The MIT Press.

Scaletti, Carla, and Kurt Hebel. 1991. "An Object-based Representation for Digital Audio Signals," in Giovanni De Poli, Aldo Piccialli, and Curtis Roads (editors), ***Representations of Musical Signals***, The MIT Press, Cambridge, Massachusetts, pp. 371-389.

Selfridge-Field, Eleanor (ed.). 1997. ***Beyond MIDI: The Handbook of Musical Codes***. Cambridge, Massachusetts: The MIT Press.

Sprott, Julien C. 1994. ***Strange Attractors: Creating Patterns in Chaos***. New York: M&T Books.

Steinberg 1998. ***Cubase VST/XT***. http://www.steinberg.net/products/cubasevst24pc.html

Sun Microsystems, Inc. 1998. ***Java Technology Home Page***. http://java.sun.com/

Syntrillium Software. 1996. ***Cool Edit 96***. http://www.syntrillium.com/cool96.htm

Taube, Heinrich. 1991. "Common Music: A Music Composition Language in Common Lisp and CLOS," ***Computer Music Journal 15***(2):21-32.

Tonality Systems. 1998. ***Symbolic Composer***. http://www.xs4all.nl/~psto/

van Dam, Andries, and David Sklar. 1991. "Object Hierarchy and Simple PHIGS (SPHIGS)," in Foley, James D., Andries van Dam, Steven K. Feiner, and John F. Hughes., ***Computer Graphics: Principles and Practice***, Second Edition, Addison-Wesley Publishing Company, Reading, Massachusetts, 1991 [1990], pages 285-346.

Vercoe, Barry. 1997 [1986]. CSound: A Manual for the Audio Processing System and for the Additional Opcodes of Extended Csound. Cambridge, Massachusetts: Media Lab, MIT.

Wenger, Eric. 1998. ***Metasynth***. http://www.ircam.fr/produits-real/logiciels/metasynth-e.html

World Wide Web Consortium. 1998. ***Extensible Markup Language (XML)***. http://www.w3.org/XML/

Xenakis, Iannis. 1992. ***Formalized Music: Thoughts and Mathematics in Music***, Revised Edition. Additional material compiled and edited by Sharon Kanach. Harmonologia Series No. 6. Stuyvesant, New York: Pendragon Press.