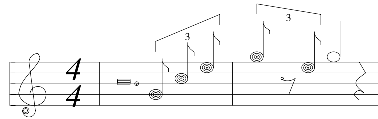
(201) 644-2332
or
Eedie & Eddie on the Wire
An Experiment in Music Generation

Peter S. Langston

Bell Communications Research
Morristown, New Jersey

ABSTRACT

At Bell Communications Research a set of programs running on loosely coupled Unix systems equipped with unusual peripherals forms a setting in which ideas about music may be ‘‘aired’’. This paper describes the hardware and software components of a short automated music concert that is available through the public switched telephone network. Three methods of algorithmic music generation are described.



Introduction

Ten years ago, in order to experiment with computer-generated music, a researcher would have required a large computer and a great deal of special purpose equipment or would have had to settle for orchestrating the serendipitous squeaks and squawks of other equipment.[0] In the last few years, advances in signal processing techniques and large-scale integration, combined with the proliferation of consumer music products have brought the cost of computer-controlled music hardware down to that of conventional musical instruments. Sound processing hardware is now reaching a level of availability comparable to that reached by text processing hardware ten to fifteen years ago. In the next ten years it would not be unreasonable to expect intense activity in the area of sound manipulation software, with a revolutionary impact on industries that depend on sound.

Sound can be used in many ways; the single most important use is as a communications medium. Spoken language is the most obvious of the techniques for communication via sound, and it is has been the subject of intense research for many years. Another use of sound is to create an experience that ‘‘communicates’’ on a non-linguistic level.

‘‘Music’’ incorporates both communication and experience but the line separating them becomes somewhat indistinct. What is it we enjoy in a piece of music? There are convincing arguments to the effect that music has both a vocabulary and a grammar, with different types of music having different vocabularies and grammars. Thus, the enjoyment of music becomes a learned skill, much like reading French or Latin, and new types of music appear to be gibberish until they are learned, e.g. ‘‘Gamelan music is Greek to me.’’ There are many interesting questions to be answered: What is the language of music? Does it have a grammar? What differentiates a ‘‘jumble of notes’’ from ‘‘a piece of music’’? What are the semantics of music? Can you say ‘‘The countryside is very peaceful’’ or even ‘‘I have lost my American Express card’’ in

[0] In 1965 I even wrote a compiler to turn music notated on punched cards into a program that, when run, would produce controlled radio interference and thereby ‘‘play music’’ on a nearby transistor radio.

music?

Before we can hope to answer any of these questions we need to have a lot more data. The present project seeks to provide some data for questions about differentiating ‘‘music’’ from ‘‘a jumble of notes’’ through subjective evaluation of the output of programs that assemble notes by relatively simple rules. Hopefully we will be able to draw some conclusions from these data. For instance: a) If the program outputs are deemed ‘‘musical’’ then the rules used in the program are *sufficient*, b) If the outputs of programs with non-overlapping sets of rules are deemed ‘‘musical’’, then neither set of rules is *necessary*.

The approaches used here for music generation deal only with syntactic (i.e. form) considerations; the programs have no methods for handling semantics. The semantics of music are assumed to involve an immense wealth of cultural and historical information (a.k.a. ‘‘knowledge’’) that does not yet exist in any machine readable form. Understanding the semantics of music is no simpler than understanding the semantics of natural languages; for that matter, it would be easy to argue that music is a natural language, albeit often a non-verbal one.

Concurrent with the project in music generation at Bell Communications Research is a project exploring the benefits of interconnecting computers and telephone equipment [REDMAN85] [REDMAN86]. As a demonstration of both projects, an E&M trunk (a direct audio connection) on our experimental telephone switch was allocated to provide telephone access to the music hardware. Some programs were then written to present, in an entertaining form, examples of the music generated.

(201) 644-2332

A block of 100 telephone numbers (644-2300 through 644-2399) have been allocated to an experimental telephone switch in our lab which has been dubbed ‘‘BerBell’’ (the switch itself is manufactured by Redcom, Inc., which has no direct relationship to B. E. Redman, logname ‘‘ber’’, the principal investigator on the BerBell project). Aside from providing enhanced telephone service for approximately 40 ‘‘people’’ lines, (32 in-house extensions, two remote lines to residences, and several numbers to which participants can forward their home phones to use BerBell’s call management features), a number of experimental services are provided.

Among these services are:

644-2300	BerBell robot operator
644-2311	Touch-Tone Directory Assistance [MORGAN73]
644-2312	games (name the beast, guess your age)
644-2312	miscellaneous information (quote of the day, today’s events in history)
644-2331	recorded music of the day
644-2332	Eedie & Eddie on the wire (the topic of this paper)
644-2335	news summary (derived from the Associated Press news wire)
644-2337	weather report (derived from the National Weather Service wire)

and others.

Figure I shows the various hardware modules that form the system and their interconnections. Lines terminated with square boxes are EIA RS-232 serial connections (bidirectional). Lines terminated with simple arrowheads are audio connections (directional). Lines terminated with pentagonal arrowheads are Musical Instrument Digital Interface (MIDI[1]) connections (directional). The remaining connections are either Ethernet lines (terminated with rectangles) or telephone lines (with no special termination symbol). The E&M trunk mentioned earlier connects the equalizer in the upper left corner of the figure with the Redcom telephone switch.

When a call comes in to 644-2332 a process called ‘‘demo2332’’ is spawned on the VAX 11/750 (‘‘yquem’’ in figure I). This process first connects the Dectalk speech synthesizer (‘‘Eddie’’ in figure I) to the caller’s incoming line and then proceeds to introduce the demo, ask the caller to enter his/her telephone number

[1] MIDI is a standard representation of synthesizer ‘‘events’’ (e.g. note-on, note-off, volume change, etc.) as a serial data stream [MIDI85].

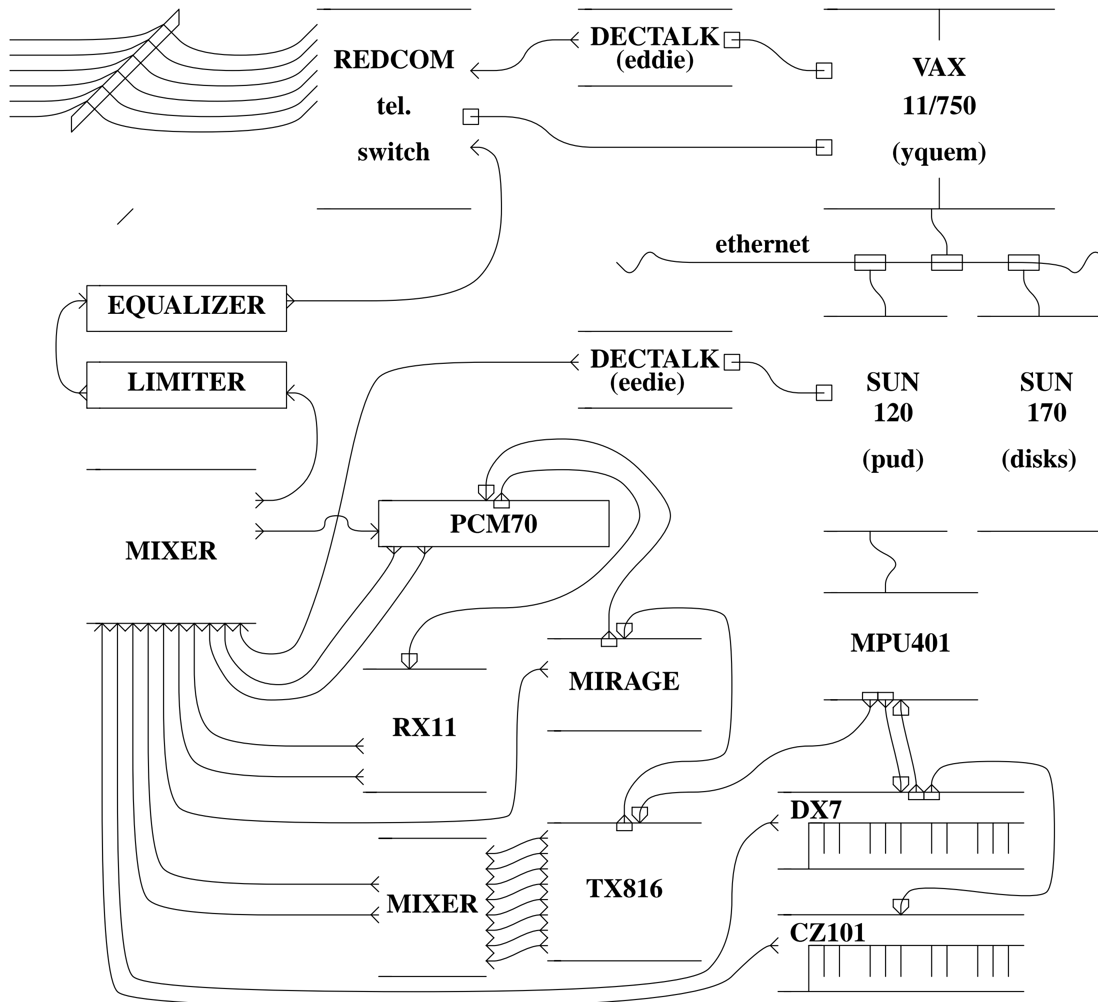


Figure I - Simplified interconnection schematic

(not for billing, but for some idea of the demographics of the participants), and to choose a long, medium, or short demo. Concurrently, ‘‘demo2332’’ spawns another process that probes the Sun workstation (‘‘pud’’ in figure I) to determine whether or not it is running (crashes have not been uncommon). If the probe finds pud ‘‘up’’, a remote process is started on pud to carry out the demo and the caller's incoming line is disconnected from Eddie and is reconnected to the equalizer in the lab where the sound generation hardware is installed. The new process running on pud plays a short fanfare to make sure that the sound generation equipment is functional (it sometimes fails for obscure reasons) and then introduces and plays each piece. The introductions are spoken by another Dectalk voice synthesizer (‘‘Eedie’’ in figure I). Note that the Dectalk does a credible job of transforming ASCII text into speech, but that sometimes creative misspelling is required in order to resolve ambiguous or irregular cases (e.g. ‘‘Eedie’’). The music is played on sound synthesizers communicating via MIDI. At the moment the sound synthesizers in use are:

Yamaha DX7	16 voice polyphonic keyboard synthesizer using FM synthesis [CHOWNING73]
Yamaha RX11	digitally sampled drum synthesizer
Casio CZ-101	8 voice polyphonic keyboard synthesizer using phase distortion synthesis
Ensoniq Mirage	8 voice polyphonic digital sampling synthesizer
Yamaha TX816	128 voice synthesizer using FM synthesis

D.E.C. Dectalk	algorithmic speech synthesizer

In addition, several pieces of sound processing equipment are in use:

Fostex 450	8 input, 4 output mixer
Tapco 8201B	8 input, 2 output mixer
Lexicon PCM70	digital effects processor used for reverberation, MIDI controlled
S-S Complimiter	audio compressor used to decrease dynamic range (for phone lines)
Rane GE-27	graphic equalizer with 1/3 octave controls

The final pieces of special hardware are those used to connect the Sun workstation to the MIDI instruments:

Home-Brew 101	Multibus adapter card to read and write parallel data to the MPU-401 and provide register access on the Sun's Multibus
Roland MPU-401	intelligent MIDI processor used to buffer MIDI data and provide real-time data control

Using these peculiar peripherals and approximately 50 special purpose programs, it is possible to generate, record, play, and edit musical pieces with an ease that astounds conventional musicians.[2] The appendix contains a list of the principal programs used.

The total cost of this particular selection of hardware, not including the VAX, the SUNs, or the REDCOM telephone switch, was about $15,000 (based on list prices). In the last year, many prices have dropped and some items have been replaced by equivalent, less expensive equipment. The telephone demo, as described in this paper, could now be implemented on equipment costing less than $4,000.

Music Generated by Optimized Randomness — Riffology

Schemes for harnessing random events to compose music predate computers by many years. Devised in the eighteenth century, Wolfgang Amadeus Mozart's "Musical Dice Game" gave an algorithm for writing music by rolling dice. Since that time, researchers at the University of Illinois, Harvard University, Bell Telephone Laboratories, and numerous other places have replaced dice with computers and turned disciplines such as signal processing, combinatorics, and probability theory toward the task of composing music [HILLER70].

The idea for the first music generation technique used in the telephone demo did not come from these illustrious predecessors, however. It came from experiences as lead guitarist in several bands that performed improvisational music. One of the popular criticisms that could be levelled at another guitarist was that he or she "just strings a lot of riffs together and plays them real fast". Of course, you were supposed to be pouring out your soul and a little bit of divinely-inspired ecstasy instead.[3] The principal objection to playing an endless succession of "riffs" is that it doesn't involve a great deal of thought, just good technique. That is to say, the syntax is more important than the semantic content. For this reason, an algorithmic implementation of riffology need not be hampered by its inability to manipulate semantics.

The theme music for the video game "*ballblazer*" (tm, Lucasfilm Ltd.), called "Song of the Grid" (tm), is generated by just such an approach [LEVINE84] [LANGSTON85]. The program runs in little memory on a small 8-bit processor (a 6502) connected to a sound chip that can make 4 independent sounds at once using square waves and pink noise. The riffology algorithm makes dynamically weighted random choices for many parameters such as which riff from a repertoire of 32 eight-note melody fragments to play next, how fast to play it, how loud to play it, when to omit or elide notes, when to insert a rhythmic break, and other such choices. These choices are predicated on a model of a facile but unimaginative (and slightly lazy) guitarist. A few examples should illustrate the idea. To choose the next riff to play, the program selects a few possibilities randomly (the ones that "come to mind" in the model). From these it selects the riff that is "easiest" to play, i.e. the riff whose starting note is closest to one scale step away from the previous riff's ending note. To decide whether to skip a note in a riff (by replacing it with a rest or lengthening the previous note's duration) a dynamic probability is generated. That probability starts at a low value, rises to a peak near the middle of the solo, and drops back to a low value at the end. The effect is that solos

[2] These are *tools* for musicians, however, not *replacements* for them.

[3] I was never accused of this form of "riffology" myself; I always stuck to straight soul-pouring.

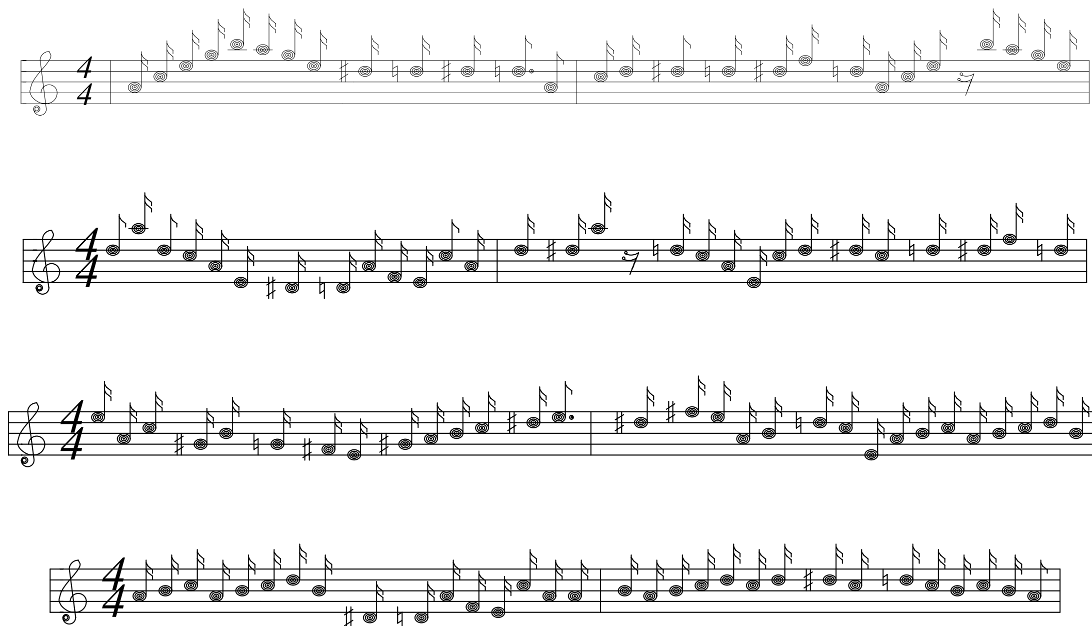


Figure II - Riffology from "Song of the Grid"

start with a blur of notes, get a little lazy toward the middle and then pick up energy again for the ending. The solo is accompanied by a bass line, rhythm pattern, and chords which vary less randomly but with similar choices. The result is an infinite, non-repeating improvisation over a non-repeating, but soon familiar, accompaniment.

A version of "Song of the Grid" forms part of Eedie's telephone demo. This *finite* version has a more standard structure; the accompaniment has a fixed A–A–B–A pattern and a precomposed "head" (melody) is played the first time through the pattern. The following improvisation uses one, two, and finally three interdependent voices. The repertoire of riffs has been increased by about 40% and now contains some famous phrases from early jazz guitarists who, being dead, could not be asked for permission (the original selection contained many riffs contributed by friends and used with their consent).

Figure II is the beginning of one of the improvisations produced. The music generated by this algorithm passes the "is it music?" test; "Song of the Grid" makes perfectly acceptable background music, (better than that heard in most elevators or supermarkets) and even won lavish praise in reviews of "*ballblazer*" in national publications. However it doesn't pass the "is it interesting music?" test after the first ten minutes of close listening, because the rhythmic structure and the large scale melodic structure are boring. It appears that for music to remain *interesting* it must have appropriate structure on many levels.

Music Generated by Formal Grammars — L-Systems

In the nineteen-sixties, Aristid Lindenmayer proposed using parallel graph grammars to model growth in biological systems [LINDENMAYER68]; these grammars are often called "Lindenmayer-systems" or simply "L-systems". Alvy Ray Smith gives a description of L-systems and their application to computer imagery in his Siggraph paper on graftals [SMITH84].

Figure III shows the components of a simple bracketed 0L-system grammar[4] and the first seven strings it generates. This example is one of the simplest grammars that can include two kinds of branching (e.g. to the left for '(' and to the right for '['). The name "FIB" was chosen for this grammar because the number of symbols (as and bs) in each generation grows as the fibonacci series.

[4] A *0L*-system is a context insensitive grammar; a *1L*-system includes the nearest neighbor in the context; a *2L*-system includes 2 neighbors; etc. A *bracketed* L-system is one in which bracketing characters (typically '[' and ']', or '(' and ')', or others) are added to the grammar as placeholders that indicate branching, but are not subject to replacement.

Alphabet:	{a,b}
Axiom:	a
Rules:	a → b
	b → (a)[b]

Generation	String
0	a
1	b
2	(a)[b]
3	(b)[(a)[b]]
4	((a)[b])[(b)[(a)[b]]]
5	((b)[(a)[b]])[((a)[b])[(b)[(a)[b]]]]
6	(((a)[b])[(b)[(a)[b]]])[((b)[(a)[b]])[((a)[b])[(b)[(a)[b]]]]]

Figure III - "FIB", a simple bracketed 0L-system

L-systems exhibit several unusual characteristics and three are of particular interest here. The first is that they can be used to generate graphic entities that really look like plants. Although the grammars themselves are quite mechanistic and have none of the apparent randomness of natural phenomena, the graphic interpretations of the strings generated are quite believably "natural". The second characteristic is structural. The "words" of the grammar (the strings produced by repeated application of the replacement rules) have a fine structure defined by the most recent replacements and a gross structure defined by the early replacements. Since all the replacements are based on the same set of rules, the fine and gross structures are related and produce a loose form of self-similarity reminiscent of the so-called "fractals" (although none of the entities produced by graph grammars are fractal in the strict sense of the word). The third characteristic of L-systems is a property called "database amplification", the ability to generate objects of impressive complexity from simple sets of rules (i.e. generating a lot of output from little input). We would like any composition algorithm to be significantly less complicated than the music it generates.

The generation of the strings (or "words") in 0L-systems is strictly defined; every possible replacement must be performed each generation, or, to put it another way, every symbol that can be replaced will be. This means that the process is deterministic as long as no two replacement rules have the same left side. Note that in cases where the rewriting rules specify replacing a single symbol with many replaceable symbols (the typical case), growth will be exponential.

Although generation of the strings is well defined, an additional *interpretation* step is required to express

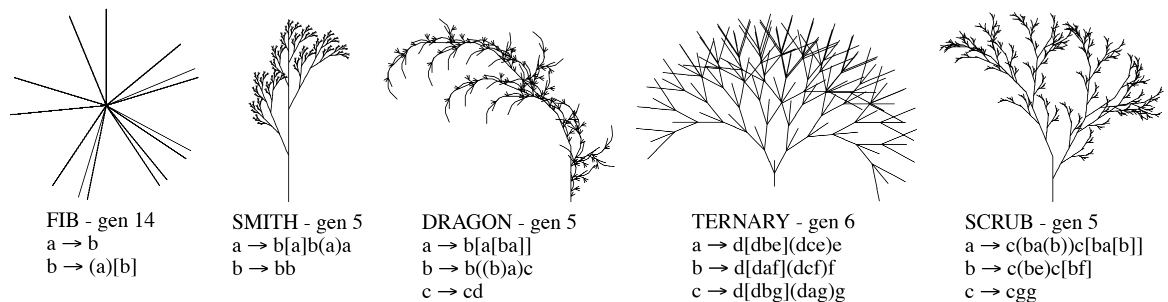


Figure IV - graphic interpretations of some graph grammars

these strings as graphic or auditory entities. Figure IV shows a very simple graphic interpretation of several grammars, including that of FIB.[5] In this interpretation the symbol '(' begins a branch at an angle of 22.5°

[5] Missing from the grammar descriptions are their alphabets ({a,b}, {a,b}, {a,b,c}, {a,b,c,d,e,f,g}, and {a,b,c,e,f,g} respectively) and their axioms (a for all of them).

to the "left" (i.e. counterclockwise) and the symbol '[' begins a branch at an angle of 28.125° to the "right" (i.e. clockwise). The examples are all scaled to make their heights approximately equal; the relative scaling ranges from 1.0 for FIB to 0.026 for SCRUB. In the first example (FIB) 716 separate line segments radiate from the starting point, many overlaid on each other. The resulting figure is 2 line segments tall. In the second example ("SMITH", borrowed from [SMITH84]) 665 line segments branch from each other and produce a figure that is approximately 60 line segments tall. The replacement rules for SMITH are more complicated than those for FIB (although still quite simple), but even when comparing strings of approximately equal length from the two grammars, the graphic interpretation of SMITH is qualitatively more complicated and "natural" looking than that of FIB. Although changing the interpretation scheme could elicit a more ornate graphic for FIB, (e.g. by avoiding the overlaid segments), the extreme simplicity of the underlying structure makes it impossible to achieve the kind of complexity that we expect in a "natural" object. However, adding a little more variety in the structure (as in SMITH) appears to be sufficient to generate that complexity.

Figure V - a musical interpretation of the grammar from figure III

The music in figure V is an interpretation of the seventh generation of FIB. The interpretation algorithm used for this example performed a depth-first traversal of the tree. At each left bracketing symbol {[,{,(} a variable "branch" was incremented. At each right bracketing symbol {],},)} "branch" was decremented. At each alphabet symbol {a,b} a variable "seg" was incremented if the symbol was b, "branch" and "seg" were then used to select one of 50 sets of four notes, and "branch" was used to select one of 13 metric patterns that play some or all of the notes. This is one of the more complicated algorithms that were used.

The fragment in figure V is pleasing and demonstrates a reasonable amount of musical variety. This contrasts sharply with the undeniably boring graphic in figure IV. We could draw any one of several conclusions from this discrepancy:

• The interpretation algorithm is insensitive to its input

While an algorithm that simply ignores its input and emits the Moonlight Sonata (an extreme example) could obviously be written and would meet our criterion for musicality, it would sacrifice the graph grammar benefits of interesting structure (since we would be ignoring it) and database amplification (since the program would have to contain its entire output and would thereby be more complicated than the music it

generates). A reasonable test is to determine how much effect a change to the input has on the output. Obviously there is a matter of degree to consider here; some similarity in the outputs is to be expected (with human composers it might be called "style"), but we also expect easily recognized differences. When this algorithm is applied to other grammars, the output generated has recognizable similarities and differences, so we must conclude it is not insensitive to its input.

• Music requires less complexity than graphics.

Music clearly requires *different* structure characteristics than graphics; the roles of fine and gross structure are almost exactly reversed in them. Music is sequential; it is experienced as a one-dimensional phenomenon that only varies with time (ignoring binaural effects), whereas (static) graphics is holistic; it is experienced as a two-dimensional phenomenon that does not vary with time. This is an important structural difference. When exposed to a graphic work, the first impression the viewer receives is of the gross structure; further examination fills in the finer structure in a sequence determined by the viewer's eye motion.[6] When exposed to a musical work, the first impression is of fine structure in one specific area (the first measure), followed by fine structure in the next area (the next measure), and so on. Only after perceiving the fine structure can the listener know the gross structure of a musical piece.

In his paper "The Complexity of Song" [KNUTH77], Donald Knuth proves the theorem "There exist arbitrarily long songs of complexity O(1)" by showing that a song recorded by Casey and the Sunshine Band has a lyric consisting of eight words arranged in a fourteen word long pattern and repeated arbitrarily many times.[7] Although Knuth's paper dealt with the *text* of songs (and was published as a joke), he touches on an important aspect of melodic structure; reiteration of previous segments. It is commonly accepted that expectation, based on the inductive/deductive process of guessing the overall structure and predicting what will come next, must be satisfied most, but not all, of the time in order for music to be pleasing. This can only be a consideration in a medium in which the structure is revealed sequentially. A simple way to achieve the middle ground between boring predictability and frustrating arbitrariness is to introduce a pattern and then repeat it with variations.

The common implication is that "high" music tends away from predictability and "low" music (as implied in Knuth's example) tends toward it. As one extreme, a long series of notes chosen randomly does not pass the "is it musical?" test. (Some very modern composers seem not to have noticed; is it because they are aiming at a very "high" audience?) Extreme complexity in music is perceived as arbitrariness, i.e. an excessively complex structure is perceived as no structure at all. Similarly, a screen full of many randomly colored pixels appears to be without structure and would fail the "is it a graphic?" test. On the other hand, a small number of randomly chosen notes or randomly colored blotches is often considered "musical" or "graphical". One possible explanation is that since many patterns will match a small part of a random sequence, the small sample of randomness is perceived as a small sample of a "real" (i.e. predictable) pattern. An interesting measure would be how big a random pattern can get before it is recognized as meaningless. A comparison of these measures for graphic and musical entities might help answer the related question, "does music *tolerate* less complexity than graphics?"

• The structure in FIB is, for some reason, more appropriate for music than for graphics.

The requirement that a balance be maintained between the repetition of old material and the introduction of new material in music implies that structures that contain repeats or near-repeats of segments are particularly appropriate. Static graphics has no time dimension in which temporal repetition can occur, and spatial repetition, while not unknown, is not as important in graphics as temporal repetition is in music. Although the syntax of graphics does not require repetition, the syntax of plants (which is the *semantics* of figure IV) does require repetition. Simply stated, plants manifest a structure composed of varied repetitions, just as music typically does. Is this a case of art imitating nature?[8] The structure in FIB, which is almost entirely repetition, satisfies one of the important requirements of music (and plants), but has little to offer for graphics per se.

---

[6] "Progressive" picture transmission techniques take advantage of this observation. Gross details are sent first, followed by successively finer details until the viewer is satisfied [KNOWLTON80] [FRANK80].

[7] "That's the way, uh huh, uh huh, I like it, uh huh, uh huh, ..."

[8] "For when there are no words it is very difficult to recognize the meaning of the harmony and rhythm, or to see that any worthy object is imitated by them." [PLATO55]

192 samples of music were produced by trying twelve different interpretation algorithms on the third generation strings of each of 16 different grammars. The samples ranged from 0.0625 bars long (one note lasting 0.15 seconds) to 46.75 bars long (about 2 minutes). A small group of evaluators listened in randomized order to every sample and rated them on a 0 to 9 scale; 0 for ‘‘awful’’ through 3 for ‘‘almost musical’’ and 6 for ‘‘pleasing’’ to 9 for ‘‘wonderful’’. Of these 192 samples ~89% rated above 3.[9] Some algorithms performed very well and generated not only ‘‘musical’’ but ‘‘pleasing’’ results on the average. Only one of the algorithms (the first one designed) averaged below 3. If that algorithm is discarded the success rate becomes ~95%.

Eedie plays several examples of melodies generated by L-system grammars in her demo, including one in which sequences derived from two different grammars are interpreted by the same algorithm and overlaid. The similarities resulting from the common interpretation scheme give the piece a fugal quality while the differences in the grammars cause the two melodies to diverge in the middle and (by luck) converge at the end.

Music Generated by ‘‘Stochastic Binary Subdivision’’ — DDM

The metric character of western popular music exhibits an extremely strong tendency. It is binary. Whole notes are divided into half notes, quarter notes, eighths, sixteenths, etc. And, while a division into three sometimes happens, as in waltzes or triplets, thirds are almost never subdivided into thirds again in the way that halves are halved again and again.[10] Further, it is rare for notes started on ‘‘small’’ note boundaries to extend past ‘‘larger’’ note boundaries. More precisely, if we group the subdivisions of a measure into ‘‘levels’’, such that the *n*th level contains all the subdivisions which are at odd multiples of 2^{-n} into the measure (e.g. level 3 consists of the temporal *locations* {1/8,3/8,5/8,7/8}), we find that notes started on level n infrequently extend across a level $n-1$ boundary and only rarely extend across a level $n-2$ boundary.

Rhythms, like melodies, must maintain a balance between the expected and the unexpected. As long as, *most of the time*, the primary stress is on the ‘‘one beat’’ with the secondary stress equally spaced between occurrences of the primary stress, and so forth, that balance is maintained. By making note length proportional to the level of subdivision of the note beginnings, emphasis is constantly returned to the primary stress cycle. Simple attempts at generating rhythms ‘‘randomly’’ fail because they ignore this requirement. More sophisticated attempts make special checks to avoid extending past the primary or the secondary stress, but the result is still erratic because we expect to hear music that keeps reiterating the simple binary subdivisions of the measure.

```
divvy(ip, lo, hi)
struct      instr      *ip;
int         lo, hi;
{
            int mid = (lo + hi) >> 1;

            ip->pat[lo] = '1';
            if ((rand() % 100) < ip->density && hi - lo > ip->res) {
                        divvy(ip, lo, mid);
                        divvy(ip, mid, hi);
            }
}
```

Figure VI - Subroutine ‘‘divvy’’ from ddm.c

The program ‘‘ddm’’[11] attempts to make musical rhythms by adhering to the observations made above, and making all other choices randomly. Figure VI is the heart of ddm. The structure instr contains, among other things, density - a probability that, at any level, subdivision to the next level will occur, res - the

[9] The median value was 5.0, but I would have been happy had it been 3.

[10] Waltzes and triplets probably gain much of their effect from the feeling that a beat is missing from a pattern of four, or has been added to a pattern of two.

[11] a.k.a. digital drum madness

shortest note that can be generated, i.e. the deepest level to which it can subdivide, and pat - a character

45:75: 2:0: 96:1 BD 52:75: 4:0: 96:1 SD 57:50: 8:0:120:1 HHC 51:50: 8:0: 64:1 RIM 48:67: 8:0: 80:1 TOM3 54:70: 8:0: 64:1 CLAP 55:75: 8:0: 64:1 COWB 59:50:16:0: 80:1 HHO 53:67:16:0: 72:1 TOM1

Figure VII - Typical Data File for DDM

string in which the generated pattern of beats is stored. Figure VII shows a typical input file for the program. The first column is the code that must be sent to a drum machine to make the intended drum sound; the second column is the percent chance that subdivision will occur at any level; the third column is the smallest subdivision allowed; the fourth column is the maximum duration of a note (0 for drum machines); the fifth column is how loud the sound should be; and the sixth column is which MIDI channel (i.e. which drum machine) should play it. Any further information on the line is comment, in this case the instrument

A2 #...............................#............................... BD E3 |...............#...............|...............#............... SD A3
|...............................|............................... HHC Eb3 |... RIM C3
|.......#.......|...............|.......#.......|............... TOM3 Gb3 |...............................|.......|.......|.......#....... CLAP G3
|...............................|.......|.......|.......|....... COWB B3 |.......|.......|...............|............................... HHO F3
|...............................|...#...|...#...|...#...|...#... TOM1

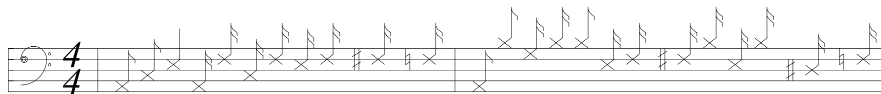


Figure VIII - Sample DDM Output

name.

Figure VIII shows the output of ddm in two forms; the first is one measure of debugging information showing the results of all subdivisions and the second is two bars of drum score showing the final result. Note that only one instrument is played at any one time; '|' indicates a subdivision, and '#' indicates the drum to be played at that time (''pound'' sign seemed appropriate, somehow). Precedence is given to the instruments listed earliest in the input file, thus the bass drum (BD) plays the down beat, even though all the instruments wanted to appear then; similarly, the low tom-tom (TOM3) plays on the next eighth note, having precedence over the open hi-hat (HHO). The drum score in figure VIII starts with the same measure and then goes on for another measure which is quite different although based on the same set of probabilities and precedences.

If we let the lines in the ddm file describe notes and pitch differences instead of drum types, ddm can be used to generate melodies. This is a simple change in the implementation; the drums are already encoded as pitches (45 ≡ A2 ≡ bass drum, 52 ≡ E3 ≡ snare drum, etc.). The only extension that is needed is to encode pitch differences. This we do by defining any value of 12 or smaller to be a pitch difference (by doing so, we make C#0 the lowest note we can specify directly). By adding lines like 1:60:16:31:64:0 and -1:65:16:31:64:0, meaning go up one step and down one step respectively, rising and falling motions can be

Scale 1,2,4,7,9 Limits 48,84 69:33: 8:12:64:0 A4 64:33: 8:12:64:0 E4
1:60:16:28:64:0 up -1:65:16:28:64:0 down
1:55:32: 4:64:0 up -1:50:32: 4:64:0 down

Figure IX - DDM File for Scat

included. Figure IX shows an eight line file used to generate melodies to be ''sung'' by a Dectalk speech synthesizer. The ''Scale'' and ''Limits'' lines constrain the program to stay within a particular musical

scale (a simple pentatonic) and within a certain range (C3, an octave below middle C, through C6, two octaves above middle C). By specifying fairly short notes in the fourth column, the chance of rests appear-

Figure X - Sample Scat Output

ing (for the singer to breathe) is increased. Figure X is a sample of ddm ouput from the file in figure IX. The program ‘‘scat.c’’ converts the MIDI format output of ddm into a sequence of Dectalk commands that produces scat singing. The syllables are generated by combining a randomly chosen consonant phoneme with a randomly chosen vowel phoneme; the result is usually humorous (and occasionally vulgar).

Eedie uses ddm twice during the telephone demo, once to compose a little scat sequence to sing for the caller and once at the end to compose a piece for bass, drums, piano, and clavinet called ‘‘Starchastic X’’, where X is the process id of Eedie’s process. Although no formal testing has been done with ddm, informal testing, (e.g. ‘‘Well, how’d you like it?’’), always elicits compliments with no apparent doubts as to its musicality.

Results & Summary

The telephone demo, despite its high ‘‘down time’’ (due in part to hardware frailties and in part to its dependence on two largely unrelated sets of experimental software), has been a great success. Eedie and Eddie have received over a thousand calls and have given as many as 60 demos a day to callers from 50 different area codes. Although no formal announcements of their demo has ever been made (prior to this paper), word of mouth has brought calls from Belgium, Canada, England, Holland, and even Texas.

Although enough experience has been gained with the music generation algorithms to draw some tentative conclusions, it should be stressed that this is still a ‘‘work in progress’’. From a quick perusal of the ‘‘further work’’ section it should be obvious that many ideas and experiments have yet to be tried, and that, undoubtedly, these tentative results will be modified and refined.

The riffology technique generates ‘‘musical’’ sounds; it is *sufficient* but gets boring with repetition. Greater rhythmic variety and some interesting overall structure are needed.

L-system grammars generate strings that have at least one important musical characteristic — varied repetition. It is not yet clear whether the relationship between their fine structure and their gross structure is a benefit for music. It is clear that the database amplification property is useful in that an arbitrarily long piece can be generated. The majority of cases tried generate ‘‘musical’’ sounds, allowing us to call this technique *sufficient*.

The binary subdivision technique works reliably. A tendency to produce fast, rising or falling sequences must be avoided by careful file specifications, otherwise almost comic sequences result (not unlike Chico Marx at the piano). Given that such sequences are avoided, its results are ‘‘musical’’, so it is *sufficient*.

With all three differing algorithms showing sufficiency we must conclude that none of the three is *necessary*. The interesting possibility remains that the overlap (intersection) of the three algorithms would be sufficient by itself. Unfortunately, the intersection of the three sets of rules is so trivial {use diatonic pitches, use no note shorter than a sixty-fourth note}, that it is unlikely that it is sufficient.

It is interesting to note that many of the pieces generated by these algorithms appear to have semantic content, (some seem to brood, some are happy and energetic, others bring Yuletide street scenes to mind). Since the algorithms themselves have no information about human emotions or the smell of chestnuts roasting on an open fire, we must conclude that any semantically significant constructions that occur are coincidental. Our semantic interpretation of these accidental occurrences, like seeing a man in the moon, testifies to the ability of human consciousness to find meaning in the meaningless.

Further Work

The work presented here just scratches the surface of a very complicated subject area. For every idea implemented during this project, ten were set aside for later consideration.

Among the ideas not yet tried for the ‘‘riffology’’ technique are:

- Expand the repertoire of riffs.
- Allow metric variation within the riffs.
- Enforce ‘‘breathing’’, i.e. limit the length of phrases of rapid notes, inserting rests in the same way a singer must.
- Provide a mechanism for generating large-scale structure, perhaps by using grammar-driven or binary subdivision algorithms.

Among the ideas not yet tried for grammar based techniques are:

- Experiment with much more complex grammars.
- Establish ‘‘controls’’ by feeding the interpretation algorithms random input and constant input. Compare the output from the controls to that from the grammars.
- Interpret the strings as rhythmic entities rather than melodic.
- Interpret the strings in larger chunks instead of symbol by symbol; perhaps by treating pairs or triples of symbols as objects, or by treating entire ‘‘branches’’ as objects.
- Experiment with 1L-systems or other context-sensitive grammars.
- Define analogues in the audio domain for the graphic interpretation of the strings such that the sounds produced are, in some non-trivial sense, equivalent to the graphics produced.
- Choose an audio interpretation of the strings and design some grammars specifically to sound musical with that interpretation; does the graphic interpretation of those grammars communicate the same components (whatever they are) that made the sounds musical?

Among the ideas not yet tried for binary subdivision techniques are:

- Allow multiple simultaneous instruments or notes (currently only one note is played at a time).
- Allow the events being selected to be at a higher level than notes (e.g. entire phrases or repeats of previous output).
- Allow the events being selected to be at a lower level than notes (e.g. slurring of notes or vibrato).
- Allow the events being selected to have more global scope (e.g. key changes or tempo shifts).

Among the ideas not yet tried for demonstrating music over telephone lines are:

- Solicit listener evaluations through the touch tone pad.
- Allow callers to select types of music or specify other parameters that influence the music generated.
- Provide stereo output (via two telephone lines).
- Charge money.

Acknowledgements

Several people have been particularly helpful with this project. Gareth Loy of UCSD did the initial work with connecting the Sun to MIDI instruments, wrote the original *record* and *play* programs, and proved it can be done. Michael Hawley of the Droid Works extended Gareth’s work, provided a flexible programming environment on the Sun, and, best of all, gave me copies of all that. Brian Redman is responsible for

the entire BerBell project and often went out of his way to allow me to do things with the phone system that no civilized person would allow. Stu Feldman (my boss) has provided encouragement, equipment, and proofreading. He has yet to suggest I consider a career in sanitation engineering.

Credit is also due to fellow musicians who were excited by the idea of computers improvising music and contributed riffs: Steve Cantor, Mike Cross, Marty Cutler, Charlie Keagle, David Levine, Lyle Mays, Pat Metheny, and Richie Shulberg.

References

CHOWNING73 John Chowning, ‘‘The Synthesis of Complex Audio Spectra by Means of Frequency Modulation’’ *Journal of the Audio Engineering Society* vol. 21, no. 7, pp. 526−534

FRANK80 Amalie J. Frank, J. D. Daniels, and Diane R. Unangst, ‘‘Progressive Image Transmission Using a Growth Geometry Coding’’ *Proceedings of the I.E.E.E.*, Special Issue on Digital Encoding of Graphics, vol. 68, no. 7, pp. 897−909 (July, 1980)

HILLER70 Lejaren Hiller, ‘‘Music Composed with Computers − A Historical Survey’’, *The Computer and Music*, Cornell University Press, pp. 42−96 (1970)

KNOWLTON80 Ken Knowlton, ‘‘Progressive Transmission of grey-scale & binary pictures by simple, efficient, and lossless encoding schemes’’ *Proceedings of the I.E.E.E.*, Special Issue on Digital Encoding of Graphics, vol. 68, no. 7, pp. 885−896 (July, 1980)

KNUTH77 D. E. Knuth, ‘‘The Complexity of Songs’’ *SIGACT News* vol. 9, no. 2, pp. 17−24 (1977)

LANGSTON85 P. S. Langston, ‘‘The Influence of Unix on the Development of Two Video Games’’, EUUG Spring ’85 Conference Procedings, (1985)

LEVINE84 D. Levine & P.S. Langston, ‘‘*ballblazer*’’, (tm) Lucasfilm Ltd., video game for the Atari 800, & Commodore 64 home computers, (1984)

LINDENMAYER68 Aristid Lindenmayer, ‘‘Mathematical Models for Cellular Interactions in Development, Parts I and II,’’ *Journal of Theoretical Biology* **18**, pp. 280-315 (1968)

MIDI85 ‘‘MIDI 1.0 Detailed Specification’’, The International MIDI Association, 11857 Hartsook St., No. Hollywood, CA 91607, (818) 505-8964, (1985)

MORGAN73 S. P. Morgan, ‘‘Minicomputers in Bell Laboratories Research’’, Bell Laboratories Record, vol. 5, no. 7 p 194−201 (1973).

PLATO55 Plato, *Laws*, Book II, (ca. 355 B.C)

REDMAN85 B. E. Redman, ‘‘Who Answers Your Phone in the Information Age?’’, Usenix Summer ’85 Conference Proceedings, (1985)

REDMAN86 B. E. Redman, ‘‘BerBell’’, Bell Communications Research Technical Memorandum (in preparation)

SMITH84 Alvy Ray Smith, ‘‘Plants, Fractals, and Formal Languages’’ *Computer Graphics* Proceedings of the Siggraph ’84 Conference, vol. 18, no. 3, pp. 1−10 (July 1984).

APPENDIX

The following list contains brief descriptions of programs used to manipulate MIDI data or for some other aspect of the telephone demo.

2332probe – Used by demo2332 to check pud’s status
adjust – perform metric adjustment dynamically
atox – convert hexadecimal ascii to binary
bars – count & select bars of MIDI data
bbriffs – generate improvisation for ‘‘Song of the Grid’’
ccc – convert ascii chord charts to MIDI data
chart – display midi data in PRN (piano roll notation)

ched	– interactive midi data editor (CHart EDitor)
chmap	– select and remap MIDI channel events
cntl	– generate MIDI control change commands
da	– disassemble MIDI data to ASCII listing
dack	– lint for ASCII MIDI listings (da check)
ddm	– stochastic binary subdivision generator
decsquawk	– control a Dectalk voice synthesizer
harm	– harmonize melody lines (MIDI)
inst	– generate MIDI voice change commands
just	– adjust (quantize) note timings (MIDI)
keyvel	– scale, compress, and expand key velocity dynamics
kmap	– remap key values (pitches)
m2midi	– convert m-format scores to MIDI
m2p	– convert m-format scores to PIC macros for typesetting
marteau	– generate and play modern (random) music
marteauslide	– interactive interface for marteau
mc	– C compiler with MIDI libraries
mdemo	– introduce, compose, and play the telephone demo
mecho	– add echo to MIDI data
merge	– combine MIDI data streams
metro	– metronome with graphic interface
midi2m	– convert MIDI data to m-format scores
midimode	– remove running status from MIDI data streams
mpuclean	– insert running status, remove other junk from MIDI data
mudef	– PIC macros for typesetting music scores
muzak	– interpret unsuspecting ASCII text as music
notedur	– change note lengths without changing tempo
p0la	– a musical interpretation of 0L-systems
p0lb	– a musical interpretation of 0L-systems
p0lc	– a musical interpretation of 0L-systems
p0ld	– a musical interpretation of 0L-systems
p0le	– a musical interpretation of 0L-systems
p0lf	– a musical interpretation of 0L-systems
p0lh	– a musical interpretation of 0L-systems
p0li	– a musical interpretation of 0L-systems
p0lpic	– a graphic interpretation of 0L-systems for typesetters
play	– play (and overdub) MIDI data streams
punk	– generate a “soft-punk” melody and accompaniment
ra	– assemble (re-assemble) ASCII data into MIDI
reclock	– regenerate timing commands for MIDI data
record	– record MIDI input
rxkey	– print information about the Yamaha RX/11/15/21 key mappings
scat	– convert MIDI data into vocal scat for the Dectalk
select	– filter MIDI data by various criteria
stretch	– retime MIDI data
transpose	– transpose MIDI key event (pitch) data
trim	– remove “dead air” from MIDI data
tshift	– shift MIDI data temporally
unjust	– add slight random time variations to MIDI data
xtoa	– convert binary data to hexadecimal ASCII