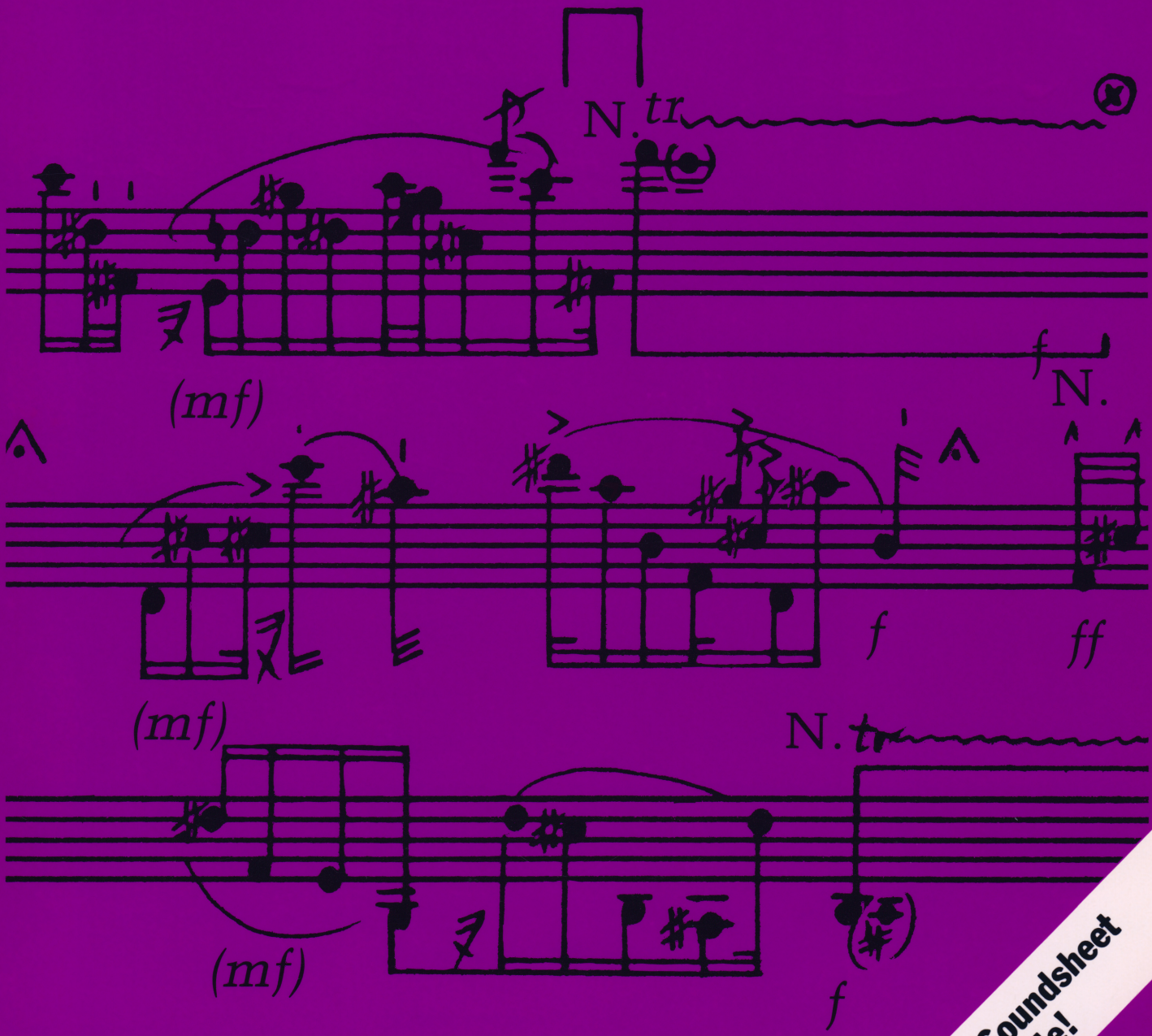
Computer Music Journal

Volume 5, Number 4 ISSN 0148-9267 $5.00 Published by the MIT Press



Special Soundsheet Inside!

Michael McNabb
Center for Computer Research in
Music and Acoustics
Stanford University
Stanford, California 94305

Dreamsong: The Composition

General Description

Sound Elements

Dreamsong was composed and realized during 1977 and 1978 at the Center for Computer Research in Music and Acoustics (CCRMA). The basic intent of the piece was to integrate a set of synthesized sounds with a set of digitally recorded natural sounds to such a degree that they would form a continuum of available sound material. The sounds thus range from the easily recognizable to the totally new, or, more poetically, from the real world to the dream realm of the imagination, with all that that implies with regard to transitions, recurring elements, and the unexpected. The essential sound elements in *Dreamsong* can be divided into five categories: simple frequency modulation (FM), complex FM, sung vocal processing and resynthesis, other additive synthesis, and processed crowd sounds and speech.

The sung vocal sounds were originally those of a single soprano, Marilyn Barber. A total of 10 single held notes and one glissando, on different pitches and syllables, were digitally recorded. Most of the soprano sounds in the work are these tones processed in various ways. In certain cases, however, the tones were synthesized using additive synthesis based on a single Fourier transform of the steady state of the original signal. The resynthesized tones were overlapped with the originals when it was necessary to have individual control over each harmonic.

The other main melodic and drone instrument is an additive synthesis instrument that generates its own time-varying formants using a particular kind of random process. The oscillating chords that appear toward the beginning and again near the end were produced with simple FM, and all the bell sounds are various types of complex FM.

The ambient crowd sounds were recorded from the catwalks of a large auditorium. Two microphones were suspended about 20 ft below the ceiling, and the sound was recorded on tape and later digitized. The speaking voice at the very end of the piece is that of Dylan Thomas.

Musical Material

Musically, *Dreamsong* presents a relatively simple harmonic and melodic structure so as not to obscure the important textural and timbral transitions. There are two major modes from which most of the melodic and harmonic material is derived (Fig. 1). Mode 1 is essentially B-flat mixolydian, although it is not always centered on B-flat. Mode 2 extends through a two-octave range before repeating and is characterized by chromaticism and a division into two regions, one in whole and half steps, the other in thirds.

The primary theme, which builds up over the course of the piece, is from Mode 1 and is the setting of a line taken from a Zen sutra (Fig. 2, top). The secondary theme, the first three notes of which appear at the outset and ending of the work and the whole of which is heard in the middle, derives generally from Mode 2 (Fig. 2, bottom).

As regards duration in general, most of the slower rhythms and section lengths derive from Fibonacci relationships, not because of their numerologic or mystic implications, but because they present a convenient and effective alternative to traditional rhythmic structures. Of course, a little acknowledgment of the gods of mathematics never hurt any computer musician.

Some rhythmic units are based on a suboctave of a central pitch being used at the time. For example, the primary tone in the first oscillating chord at 102 sec is F, 349.23 Hz. Six octaves below that is

Computer Music Journal, Vol. 5, No. 4, Winter 1981,
0148-9267/81/040036-18 $05.00/0

Fig. 1. Two modes from which much of the pitch material in Dreamsong *was derived.*

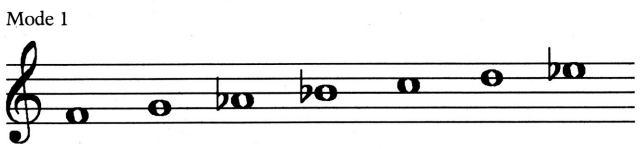


Fig. 2. Themes in Dreamsong. *The primary theme (top),a setting of a Zen sutra derived from Mode 1. Secondary theme (bottom), derived in part from Mode 2.*

5.457 Hz, which is a period of 0.183 sec. This is the rate at which the chord oscillates from channel to channel and is four times the rate of the octave bells on B-flat in the background.

Programs, Machines

The primary program used was MUS10, Leland Smith's version of Tovar's MUSCMP (Tovar 1977; Tovar and Smith 1977), which is an extended version of Stanford's original music program (written by David W. Poole). That program was a descendant of Music IV, written by Max W. Mathews. This music compiler features a subset of Algol with special operations directed toward music synthesis. Thus, in addition to the usual unit generator–type functions, one can program one's own initialization-time and run-time functions, which allows a tremendous degree of flexibility in instrument design—yet to be matched by any hardware synthesizer currently operational. This flexibility was critical to the production of *Dreamsong*.

Sound editing, filtering, and analysis were done using the programs EDSND by Loren Rush (Rush and Mattox, forthcoming) and S by James A. Moorer. In certain cases, note lists were generated with Leland Smith's SCORE program (Smith 1972), although most of the work was generated in small sections to be digitally mixed, so that note lists could usually be typed in directly using the text editor. All computation was carried out on the DEC KL-10 processor of the Stanford Artificial Intelligence Laboratory.

About MUS10

Since much of the discussion in this article will be illustrated with code listings, it will be necessary to digress slightly to give a general introduction to the MUS10 language.

MUS10 code includes a mixture of floating-point variables and arrays, simple Algol statements and functions, and *unit generators*, which are special functions designed to cycle through waveform tables or envelopes or perform other processes such as random-number generation. These unit genera-

tors are very similar to the unit generators in Music V (Mathews 1969).

The code for a particular sound or process is grouped into what is known as an *instrument* block, which begins with the declaration "**instrument** ⟨name⟩;" (an example to be discussed later is given in Code Listing 3). The code in an instrument block is evaluated once per sample, with all current instruments being evaluated in order of definition. The output of each instrument for that sample for each channel is then stored, and the process begins again for the next sample.

An instrument declaration may be followed by the declaration of variables local to that block, using the declaration "**variable** ⟨variable 1⟩, ⟨variable 2⟩, . . . ⟨variable *n*⟩;". If the variables are to change in value during the course of a note (e.g., receive the value of a unit generator), they are known as *run-time variables* and are preceded by a slash in the declaration (the first such example is in the code for RANDF, Code Listing 2). Otherwise they are *initialization-time variables* and will retain the first value that they are assigned. Arrays are declared with "**array** ⟨name1⟩, ⟨name2⟩, . . . ⟨name *n* [*x*]⟩;" where the elements of the array are numbered from 0 to $x - 1$. Variables and arrays may also be declared outside of an instrument block, in which case they are global to all subsequent instruments. These are often used for communication between instruments.

Following the instrument and variable declarations may come a block of code that is to be executed only at the onset of a note. This block is prefaced with "**i_only**" (i.e., initialization only). Either run-time or initialization-time variables may be assigned values in this block.

There are a number of predeclared variables. **SRate** contains the sampling rate in Hertz. **Outa**, **outb**, **outc**, and **outd** store the final values of samples going to each of four possible channels. A set of variables P1, P2, P3, . . . P*n* stores the parameters for each instrument call. **Mag** (for *magic number*) stores a value needed in unit generators and elsewhere to compute the increments. **Mag** is normally set to the standard wavetable length divided by the sampling rate. Multiplying this by the desired frequency gives the increment. For example, if the wavetable is 512 words in length and **sRate** is 25600 Hz, then **mag** is 0.02. If the waveform is to be played at 200 Hz, then the increment becomes **mag** * 200 = 4.

The unit generators referred to in this article are only a subset of those available in MUS10.

Oscil (⟨amplitude⟩, ⟨increment⟩, ⟨array⟩) returns a value from the given waveform array, depending on ⟨increment⟩, and scaled by ⟨amplitude⟩.

Zoscil is similar to **oscil**, but interpolates between adjacent array elements for increased accuracy (i.e., less noise).

Expen (or **zexpen**) is like **oscil** (or **zoscil**), but reads through the array only once, holding the final value thereafter.

Zosca is a form of **zoscil** that allows the waveform index to be initialized to **zosca**'s first argument.

Linen(⟨amp⟩, ⟨attack time⟩, ⟨decay time⟩, ⟨duration⟩, ⟨array⟩, ⟨internal variable⟩) allows independent control over the attack and decay portions of an envelope, like the **LINENS** unit generator of Music IVBF (Howe 1975). The ⟨internal variable⟩ must be declared by the calling routine (an example will be given in Code Listing 3).

Randh(⟨amplitude⟩, ⟨increment⟩) returns a value from a stepwise random function.

Value[n](⟨expression⟩) is not a unit generator as such, but simply causes the given expression to be evaluated every *n* samples.

Algol functions may be declared using the construct "**function** (⟨argument 1⟩, ⟨argument 2⟩, . . . ⟨argument *n*⟩)" followed by a block of code that performs operations on the given arguments or global variables or both. These functions may also **return** a result to the instrument or function from which they are called. A number of predeclared functions exists:

Int(R) returns the integer part of a real number *R*.
Rand returns a random number between −1 and 1.
Power(x,y) returns x^y.
Zero(A) sets all elements in array A to 0.
Sqrt(x) returns the square root of *x*.

Two special operators, **seg** and **synth**, are used to set up envelope and waveform tables. **Seg** is used

Code Listing 1. Code for the ZDELAY and SDELAY functions, which are interpolating variable-length delay units.

```
function ZDELAY(Input, Curlen, BufLen, array Buffer, Inptr);
  begin
  variable Samp1, Samp2;
  Buffer[Inptr] ← Input;                                    ⟨Read in the new sample
  Samp1 ← Inptr − Curlen;                                   ⟨Position readout pointer
  if Samp1 < 0 then Samp1 ← Samp1 + BufLen;                 ⟨Might have to wrap around
  Inptr ← Inptr + 1;                                        ⟨Increment input pointer
  if Inptr ≥ BufLen then Inptr ← 0;                         ⟨Wrap around if at end of array
  if Samp1 < BufLen−1 then Samp2 ← Samp1+1
    else Samp2 ← 0;
  return(Buffer[Samp1] + ((Samp1−int(Samp1)) * (Buffer[Samp2]−Buffer[Samp1])));
  end;

function SDELAY(Curlen, BufLen, array Buffer, Inptr);       ⟨Just reads out of same buffer
  begin
  variable Samp1, Samp2;
  Samp1 ← Inptr − Curlen;                                   ⟨Position readout pointer
  if Samp1 < 0 then Samp1 ← Samp1 + BufLen;                 ⟨Might have to wrap around
  if Samp1 < BufLen−1 then Samp2 ← Samp1+1
    else Samp2 ← 0;                                         ⟨Check for end of array
  return(Buffer[Samp1] + ((Samp1−int(Samp1)) * (Buffer[Samp2]−Buffer[Samp1])));
  end;
```

to fill a table with a line-segment function. If env is declared as an array, then "**seg** (env); y_1 x_1, y_2 x_2, . . . , y_n x_n;" fills the array with a function defined by the given coordinate pairs, where the x_n are between 1 and 100, and the y_n may be any value. The function is normalized in the x domain to fit the bounds of the array. **Synth** is used when a set of harmonic numbers and corresponding amplitudes are to be defined. **Synth** (wave); h_1 a_1, h_2 a_2, . . . , h_n a_n 999;" will fill the array wave with a waveform containing the harmonics humbered h_n at their respective amplitudes a_n. The waveform is normalized to fit within the range [−1,1]. The number 999 is used to mark the end of the definition. If desired, two additional numbers may be added to each pair, representing the initial phase in degrees, and the amplitude ("DC") offset. In this case, "99" must immediately precede the definition (Arnold 1978).

Some Key Functions

ZDELAY, SDELAY

The Algol part of MUS10 allowed the creation of several run-time functions that were used throughout *Dreamsong* for many different purposes. These include ZDELAY (an interpolating delay line), SDELAY (which provides an additional output from ZDELAY), and RANDF (a correlated noise generator).

ZDELAY (see Code Listing 1) consists of a buffer array that can be thought of as a ring. Samples are written sequentially into the array, wrapping around when the end of the array is reached. Samples are then read out from a "delayed" location, some real number of samples behind where the new samples are being written. The length of the delay may be dynamically variable, and a linear in-

terpolation is done to provide the fractional part of the delay length. SDELAY simply provides an additional output tap at a different point on the same delay line.

Randf

MUS10 includes three functions that deal with random numbers: **rand**, **randh**, and **randi**. **Rand** (which requires no argument) returns a new random number whenever it is called. **Randh** generates a random stepwise function at a given frequency. **Randi** does likewise, but linearly interpolates between new values, producing a zig-zag function. All values are scaled to [−1,1].

Attempts to use **randi** as a frequency modulator to produce a natural-sounding random vibrato met with little success and led to extensive experimentation with $1/f$ and other types of noise. $1/f$ noise has a power spectral density that decreases proportionally to 1/frequency, as opposed to white noise, whose power spectral density is flat. The essential difference is that white noise is uncorrelated: each new random value is chosen independently of the preceding one. $1/f$ noise, however, is correlated noise: each new value depends to some extent on what the last one was (Gardner 1978). This kind of randomness is frequently found in nature, for example, in profiles of coastlines, mountain ranges, etc., and it is also probably representative of the natural unevenness of human-produced vocal and instrumental tones. After all, in these cases each physical variation is certainly not independent of the adjacent or preceding ones. When used to synthesize a random vibrato, the $1/f$ variety produces a much more natural-sounding result than white noise. A certain smoothness is heard, resulting from the absence of the occasional large skips between values that occur when there is no correlation.

After several complex and computationally expensive $1/f$ noise algorithms (programmed by Julius Smith) were tried, it was observed that for most purposes almost any moderate amount of correlation applied to the random-number sequence would do the job. RANDF in Code Listing 2 does just that in a very simple manner, producing white noise when Factor is 0, and increasingly correlated noise as Factor approaches 1. The quality of the noise can thus be "tuned." (OldVal keeps track of the previously generated value.) RANDFI produces a linearly interpolating random function at frequency Freq, scaled by amplitude Amp, and correlated according to Factor. Cnt must be initialized to 0, and OldVal and Diff are global variables used to keep track of the preceding value and its difference from the next one. RANDFC is much like RANDFI, but interpolates with the portion of the cosine function taken from 0 to π instead of a straight line. This produces better results in certain applications, such as in controlling the delay time of a ZDELAY unit being used as a choral-effects generator.

Readin

Another special function used frequently is **readin**. This MUS10 function enables input of one or more stored digital sound files into an instrument, where the samples can be modified in any way allowed by the available code. One obvious use of this feature is for processing digitally recorded natural sounds. Another very valuable application is multipass synthesis, in which sounds (phrases, passages) are arrived at gradually in steps. Synthesizing all aspects at once can be very inefficient in a system in which one tiny mistake can cause the recomputation of many minutes of complex sound. For example, one might start by computing a single-voice melody, then process that through a chorus-effect generator, then process that through a panning instrument, and finally add reverberation. Each step of the process may be worked on until it is perfected, without recomputation of the previous steps. For small systems, this method also has the advantage of keeping program core requirements down.

In this fashion, compositions are gradually constructed piecemeal and mixed into larger and larger sections until the work is complete. This is how *Dreamsong* came into being.

Synthesis Techniques

Additive Synthesis: Voice Instrument

As was mentioned earlier, most of the distinctly vocal sounds in *Dreamsong* were produced by pro-

Code Listing 2. Random-number generators used in Dreamsong. RANDF *is a random-number function that uses correlated noise.* RANDFI*, a periodic random-number generator, interpolates linearly between successive values.* RANDFC *is a periodic random-number generator that interpolates using the portion of the cosine function taken from 0 to* π.

```
function RANDF(OldVal, Factor);
  begin
  variable /LowBound, /UprBound, /Range;
  Range ← 2 − 2 * Factor;
  LowBound ← OldVal − Range;
  if LowBound < −1 then LowBound ← −1;
  UprBound ← OldVal + Range;
  if UprBound > 1 then UprBound ← 1;
  return(LowBound + (UprBound−LowBound) * (1+rand)/2);
  end;

function RANDFI(Amp,Freq,Factor,OldVal,Diff,Cnt);
  begin
  variable /Interp;
  Interp ← Cnt/(sRate/Freq);
  if Interp ≥ 1 then Cnt ← Interp ← 0;
  if Cnt = 0 then
    begin
    OldVal ← OldVal + Diff;
    Diff ← RANDF(OldVal, Factor) − OldVal;
    end;
  Cnt ← Cnt + 1;
  return(Amp * (OldVal + Diff * Interp));
  end;
  array Curv,CurvDiff(128);                                  ⟨"s" Curve data
  variable I,CosLen;
  CosLen ← 127;
  for I ← 0 step 1 until CosLen do Curv[I] ← (1−COS(PI * (I/CosLen)))*.5;
  for I ← 0 step 1 until CosLen−1 do CurvDiff[I] ← Curv[I+1] − Curv[I];

  function RANDFC(Amp, Freq, Factor, OldVal, Diff, Cnt);
    begin
    variable Indx, NewVal;
    Indx ← (CosLen/sRate) * Cnt* Freq;
    if Indx ≥ CosLen then Cnt ← Indx ← 0;
    if Cnt = 0 then
      begin
      OldVal ← OldVal + Diff;
      Diff ← RANDF(OldVal, Factor) − OldVal;
      end;
    Cnt ← Cnt + 1;
    return(Amp * (OldVal+Diff * (Curv[Indx]+CurvDiff[Indx] * (Indx−int[Indx]))));
    end;
```

cessing digitally recorded soprano tones. However, if certain kinds of transformations are desired, recorded tones cannot be used. For example, 220 sec into *Dreamsong*, a cluster of bells at random pitches gradually coalesces into a solo singing voice, and soon after that a whole chorus of singers coalesces into a single, low-frequency drone! In these cases, it is necessary to have absolute control over the frequency and amplitude of each harmonic of the voice. If the spectrum is generated by a set of individual sine-wave oscillators, one can easily interpolate from or to other similar sets of data, whether harmonic or inharmonic. The exploitation of *spectral fusion* phenomena (Chowning 1980) becomes relatively simple and is rich with expressive possibilities.

When such control was not needed, it was found that the application of the proper vibrato, overall frequency skew, and overall amplitude envelope to an otherwise steady-state waveform was quite adequate to synthesize a soprano voice convincingly (see Code Listing 3). The spectral information was obtained by doing a Fourier transform of a segment of approximately 200 msec of the original voice. This method of synthesis represents a considerable step in data reduction from complete three-dimensional analysis (time, frequency, amplitude) such as that done by the phase vocoder (Portnoff 1976). All the harmonics remain proportionally the same relative to the fundamental (this would not be the case if an interpolation were taking place). In a complex compositional texture, where the exact reproduction of a specific tone is not needed, this method gives very good results. Essentially, there is only one frequency function and one amplitude function. The frequency function in Code Listing 3 has two parts, vibrato and skew. The *skew* function is a simplified representation of the natural tendency to "home in" on the pitch during the attack of the note. The vibrato is synthesized separately using RANDFI.

Additive Synthesis: Random Formant Instrument

The primary melodic and drone instrument in *Dreamsong*, other than the voice, also uses additive synthesis. It computes its own dynamic spectrum based on a series of randomly generated formant structures. The overall spectral decay shape is supplied as an array of amplitudes or possibly two arrays with a gradual interpolation between them. The formant structure is computed from a random number tree, as shown in Code Listing 4, with the sum of the eight spectral amplitudes normalized to the value N by the GetHarms function. The process is analogous to a binary fractal pattern (Mandelbrot 1977), only each iteration is randomized. The results are quite different from results when eight independent random numbers are selected. For example, with the tree, all eight spectral amplitude values will come out equal only if all the random numbers turn out to be exactly 0.5. This process thus guarantees a much greater variety in the distribution of energy from one function call to the next.

The instrument OMM, given in Code Listing 5, appears many times in *Dreamsong*, in particular between 90 sec and 150 sec, between 265 sec and 415 sec (see Fig. 4), and between 477 sec and 510 sec. At the end of the second low drone (around 400 sec), the controlling spectral shape gradually allows only the 11th and 12th harmonics to be present (in a 12-harmonic version), which become the two pitches around which the following whole section is based, thus effectively blurring the distinction between timbre and normal musical pitch structures in a way possible only with digital synthesis. The instrument as presented here is somewhat simpler than that used in the piece, but all essential elements are present. In addition, envelopes for a sample case are given in the code. Two auxiliary functions are also defined, one that simply copies an array and another that interpolates between two arrays, given the interpolation fraction.

Code Listing 5 contains a sample instrument call that begins generating new formants at 1-sec intervals and gradually speeds up to 0.1-sec intervals. The overall spectral decay shape changes from F1 to F2 (also defined in Code Listing 5) in the first 2.5 sec of the note, which glissandos up from F to G and has 1.2% vibrato at 4.5 Hz.

Frequency Modulation: Two Bells

A complex FM instrument was designed to produce a cathedral-bell-like sound. It uses three carriers,

Code Listing 3. Instrument SING, *used to model a singing voice with additive synthesis and* RANDFI.

```
array F1,F2,F3(512);
synth(F1);1 1.0000,   2 0.7079,   3 0.0126,   4 0.0050,
          5 0.0316,   6 0.0016,   7 0.0022,   8 0.0014,
          9 0.0018,  10 0.0056,  11 0.0010,  12 0.0010,
         13 0.0010,  14 0.0014,  999;                      ⟨soprano "ah" at D5
seg(F2); 0 1,  1 10,  0.7 25,  0.9 37.5,  0.7 50,  0 75,  0 100;   ⟨Amplitude envelope
seg(F3);−1 1,  0.3 12,  0 25,  0 50,  −.5 75,  −.5 100;            ⟨Frequency skew envelope

instrument SING;                                           ⟨Parameters: Beg, Dur, Freq, Amp
variable /X,/Y,/Cnt,/Va1,/Va2,/Vib,/Env,/Sig,/Skew;        ⟨runtime variables
i_only begin X ← Y ← Cnt ← Va1 ← Va2 ← 0; end;             ⟨Always initialize these to zero
Vib ← 1 + RANDFI(.01, 18, 0.6, X, Y, Cnt);                 ⟨random vibrato generator
Skew ← 1 + linen(.12, 0.2, 0.1, P2, F3, Va1);              ⟨frequency skew function
Sig ← oscil(P4, (P3 * mag)* Vib * Skew, F1);               ⟨waveform oscillator
Env ← linen(1, 0.18, 0.15, P2, F2, Va2);                   ⟨envelope generator with attack
outa ← outa + Sig * Env;                                   ⟨  and decay parameters
end;
```

Code Listing 4. Function GetHarms, which generates harmonics used by instrument OMM *(Code Listing 5).*

```
function GetHarms(array Harms, array Shape, N);
  begin
  variable I,Sum; array X(9),R(7);
  for I ← 0 step 1 until 6 do R[I] ← abs(rand);          ⟨Choose seven random
                                                         ⟨  numbers (0 ⇔ 1)
  X[1] ←     R[0] *     R[1] *     R[3] ;                ⟨calculate 8 values from tree.
  X[2] ←     R[0] *     R[1] * (1−R[3]);
  X[3] ←     R[0] * (1−R[1]) *     R[4] ;
  X[4] ←     R[0] * (1−R[1]) * (1−R[4]);
  X[5] ← (1−R[0]) *     R[2] *     R[5] ;
  X[6] ← (1−R[0]) *     R[2] * (1−R[5]);
  X[7] ← (1−R[0]) * (1−R[2]) *     R[6] ;
  X[8] ← (1−R[0]) * (1−R[2]) * (1−R[6]);
  Sum ← 0;
  for I ← 1 step 1 until 8 do
    begin
    Harms[I] ← X[I] * Shape[I];                          ⟨Multiply by spectral envelope
    Sum ← Sum + Harms[I];
    end;
  for I ← 1 step 1 until 8 do
    Harms[I] ← Harms[I] * N / Sum;                       ⟨Normalize to N
  end;
```

Code Listing 5. Instrument OMM *is an additive synthesis instrument that generates its own random series of formants using GetHarms (Code Listing 4). The sample instrument call uses typical parameter values.*

```
Parameters:
  P1:  Begin Time
  P2:  Duration
  P3:  Fundamental frequency
  P4:  Peak Amplitude (arbitrary linear scale from 0 to 2047)
  P5:  Amplitude function
  P6:  Formant shift time, function in P8 = 0
  P7:  Formant shift time, function in P8 = 1
  P8:  Formant shift time function
  P9:  Spectral envelope 1
  P10: Spectral envelope 2
  P11: Duration of interpolation from P9 to P10
  P12: Glissando note, function in P13 = 1
  P13: Glissando function
  P14: Percentage of vibrato (0 ⇔ 1) as percentage of P3
  P15: Vibrato rate
  P16: Amplitude envelope attack time (for linen)
  P17: Amplitude envelope decay time

array Env,Ramp1,Ramp2,Gliss,Syn[512];
array F1,F2,CurShape,Shape1,Shape2,Amps,NewAmp,OldAmp[9];

seg(Env); 0 1, 1 5, 0.7 25, 1 37.5, 0.7 50, 0 75, 0 100;   ⟨Amplitude envelope
seg(Ramp1); 0 1, 1 75, 1 100;                              ⟨Upward ramp
seg(Ramp2); 1 1, 0 100;                                    ⟨Downward ramp
seg(Gliss); 0 1, 0 45, 1 55, 1 100;                        ⟨Upward glissando
synth(Syn); 1 1, 999;                                      ⟨Sine wave
variable I;
for I ← 1 step 1 until 8 do                                ⟨Two spectral envelopes:
  begin
  F1[I] ← 1 / power(I,2);                                  ⟨A = 1/N²
  F2[I] ← 1 / I;                                           ⟨A = 1/N
  end;
F2[2] ← F2[4] ← F2[6] ← F2[8] ← 0;                         ⟨F2 gets odd harmonics only

function ArrTran(array X,array Y,I);                       ⟨Copies one array into another
  begin
  for I ← 1 step 1 until 8 do X[I] ← Y[I];
  end;

function FunIntrp
  (array One, array Two, array New, Fraction);             ⟨Interpolates between two arrays
  begin variable I;
  for I ← 1 step 1 until 8 do
    New[I] ← One[I]+(Two[I]−One[I]) * Fraction;
  end;
  instrument OMM;
  variable /Switch,/Samples,/Amp,/Count,Limit,/Intr,/Vib,Var,
           /AmpScl,/Rate,/Inc,/FormantPeriod,/AttAmp,/Att,/I,/Gliss,
           /H1,/H2,/H3,/H4,/H5,/H6,/H7,/H8;
```

```
i_only begin
   Samples ← Var ← 0;                                  ⟨Initialization code
   Switch ← 1;
   ArrTran(Shape1,P9,I); ArrTran(Shape2,P10,I);        ⟨Get spectral envelopes
   GetHarms(NewAmp, Shape1, Amp ← P4);                 ⟨Initial amplitudes
   FormantPeriod ← sRate * P6;                         ⟨Rate of formant change
   Count ← FormantPeriod + 1;                          ⟨Counter for interpolating
   Limit ← sRate * P11;                                ⟨How quickly to interpolate
   end;
if Samples ≤ Limit then
   begin
   Samples ← Samples+1;
   Intr ← Samples/Limit;                               ⟨Interpolate spectral envelopes
   value[16](FunIntrp(Shape1,Shape2,CurShape,Intr));
   end;
if Count > FormantPeriod then
   begin
   Count ← 0;
   ArrTran(OldAmp, NewAmp, I);                         ⟨Save previous formants
   GetHarms(NewAmp, CurShape, Amp);                    ⟨get new formants
   end;
Rate ← P6 + oscil(P7−P6, mag/P2, P8);                  ⟨Get rate of formant change
value[8](FormantPeriod ← int(sRate * Rate));           ⟨Convert to samples
value[4](Switch ← 1);
if Switch = 1 then
   begin                                               ⟨Calculate current amplitudes
   Intr ← Count / FormantPeriod;
   for I ← 1 step 1 until 8 do
      Amps[I] ← OldAmp[I]+(NewAmp[I]−OldAmp[I]) * Intr;
   Switch ← 0;
   end;
Count ← Count + 1;
AttAmp ← expen(1, mag/.17, Ramp2);                     ⟨Attack noise envelope
if AttAmp > 0 then
   Att ← 1 + RANDH(0.5 * AttAmp, 4000 * mag);          ⟨Attack noise
Gliss ← oscil((P12−P3) * Att * mag, mag/P2, P13);      ⟨glissando factor
Vib   ← oscil((P14*P3) * Att * mag, mag * P15, SYN);   ⟨vibrato factor
Inc ← P3*Att*mag + Gliss + Vib;                        ⟨Calculate frequency increment
H1 ← oscil(Amps[1],     Inc, SYN);                     ⟨generate eight harmonics
H2 ← oscil(Amps[2], 2 * Inc, SYN);
H3 ← oscil(Amps[3], 3 * Inc, SYN);
H4 ← oscil(Amps[4], 4 * Inc, SYN);
H5 ← oscil(Amps[5], 5 * Inc, SYN);
H6 ← oscil(Amps[6], 6 * Inc, SYN);
H7 ← oscil(Amps[7], 7 * Inc, SYN);
H8 ← oscil(Amps[8], 8 * Inc, SYN);
AmpScl ← linen(1, P16, P17, P2, P5, Var);              ⟨Amplitude function
outa ← outa + (H1+H2+H3+H4+H5+H6+H7+H8) * AmpScl;      ⟨Output
end;
play;                                                  ⟨sample instrument call
OMM 0 3 F 2000 Env 1 0.1 Ramp1 F1 F2 2.5 G Gliss 0.012 4.5 0.2 0.2;
finish;
```

Code Listing 6. An oscillator is used to shape a signal read in by the **readin** *function; the shaping envelope name is passed in P4.*

```
instrument SHAPER;                          ⟨Parameters: Beg, Dur, Amp scaler, Envelope
variable /Env;
Env ← zoscil(P3, mag/P2, P4);               ⟨generate envelope
outa ← outa + readin(RD) * Env;             ⟨multiply times input samples
end;
```

each modified by one modulator, one of which contains a complex wave. The second carrier, which produces an inharmonic spectrum, actually contributes relatively little, since its amplitude is kept small. Most of the characteristic sound comes from a combination of one harmonic spectrum, with a second harmonic spectrum having a fundamental a just minor 10th above the first. This instrument is used at the very beginning of *Dreamsong* and again in the middle, at about 350 sec.

A handbell sound was also created, which uses two modulators and two carriers like two simple FM units in parallel. There is no inharmonic ratio in either unit, and again the characteristic sound comes from the combination of one harmonic spectrum with another having a fundamental a just minor 10th above the first.

Processing Techniques

Shaping

Most of the following processing algorithms are constructed around the functions defined in the section entitled "Some Key Functions" and were applied to the digitally recorded sung vocal and crowd sounds and to previously synthesized sounds. They demonstrate in particular the usefulness and versatility of the ZDELAY function.

An introductory example of sound file processing in MUS10, Code Listing 6, is an instrument that reads a sound file, shapes its amplitude by a given function, and writes it out again.

Comb Filtering, Flanging

Perhaps the simplest application of ZDELAY is as a *comb filter*, which results when a delayed signal is added back to itself. Since the peaks thus produced are separated by a constant frequency, they form a harmonic series and can be used to give a pitched effect to an otherwise nonpitched sound. As more feedback is added, the filtering is more severe and the pitched effect is more pronounced. A delay of **sRate**/f samples (used for curLen in Code Listing 1) will place the first peak at frequency f. If the delay is doubled and the delayed signal subtracted from the original signal instead of added, the peaks form a pattern that corresponds only to odd harmonics, which produces an expectedly "hollow" effect. If feedback is used, the output signal must be renormalized down to a reasonable amplitude.

Modulation of the delay time (curLen in Code Listing 1) with a sine wave results in an effect known as *flanging*. This term originated in the pop music recording industry when somebody got drunk and discovered that an interesting phasing effect could be produced by setting up a very short tape delay and leaning on the flange of the reel to change its speed slightly. Of course, functions other than a sine wave may be used as modulators. The maximum delay time should be around one period of the sound being processed. This produces more than just a nice sound when the function used is not just a sine wave; if the modulating function undergoes discrete changes, a sequence of pitches is heard. This occurs in *Dreamsong* 30 sec into the piece, when the initial crowd noise is processed into playing the opening melodic motif.

Choral Effect

A more useful ability of ZDELAY is to provide a choral-effect generator that can be used on any sound, in particular on digitally recorded sounds and synthesis instruments that are too complex to allow for multiple copies. In Code Listing 7, three delayed

Code Listing 7. Instrument CHORUS: RANDFC *and* ZDELAY *functions are used to generate a choral effect from any sound.*

```
array Buffer[512];
zero(Buffer);

instrument CHORUS;
variable /Sig, /De1,/De2,/De3, /Ds1,/Ds2,/Ds3,
  /Inptr,Len,Dev,Rate,/A1,/A2,/A3,/B1,/B2,/B3,/C1,/C2,/C3;
i_only begin
  Inptr ← A1 ← A2 ← A3 ← B1 ← B2 ← B3 ← C1 ← C2 ← C3 ← 0;
  Len ← length(Buffer);
  Rate ← P3;                          ⟨4 Hz is typical
  Dev  ← P4;                          ⟨10 msec worth of samples is usually good for this
  end;
Sig ← readin(RD);                                  ⟨Input the samples
De1 ← Dev + RANDFC(Dev,Rate,.5,A1,A2,A3);          ⟨delays range from 0 to Dev*2;
De2 ← Dev + RANDFC(Dev,1.1*Rate,.5,B1,B2,B3);
De3 ← Dev + RANDFC(Dev,.9*Rate,.5,C1,C2,C3);
Ds1 ← ZDELAY(Sig, De1, Len, Buffer, Inptr);        ⟨three delayed versions of signal
Ds2 ← SDELAY(Sig, De2, Len, Buffer, Inptr);        ⟨SDELAY reads out in a different spot
Ds3 ← SDELAY(Sig, De3, Len, Buffer, Inptr);        ⟨Yet another copy
outa ← outa + (Sig + Ds1 + Ds2 + Ds3) * 0.3;       ⟨combine all with original, rescale
end;
```

copies of the signal are combined with the original. The delay time of each copy is modulated independently by a RANDFC. The frequencies of the three modulators are offset by about 10%.

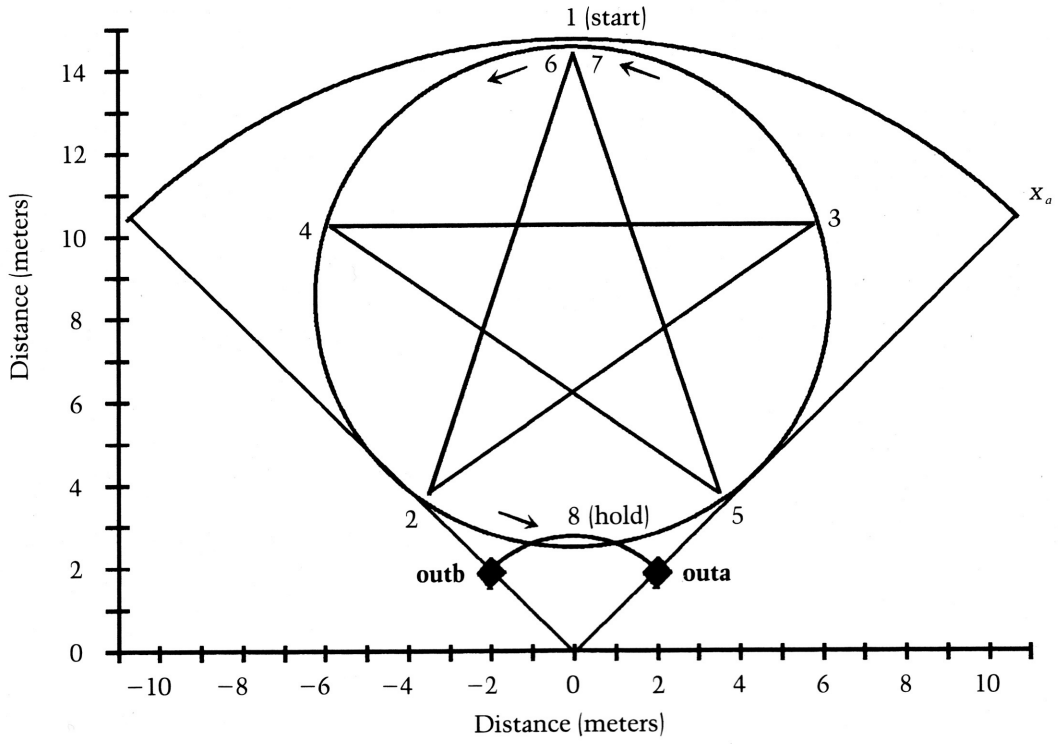
Panning, Doppler Shift

The instruments discussed in this section make up the spatial movement system used in *Dreamsong*. Since the piece is in only two channels, every effort was made to maximize the effect of depth and movement. To this end, there is a panning instrument used to both simulate Doppler shift and modulate the initial reflection times for reverberation. This instrument is set up to move the input signal in a straight line or in an arc through an imaginary space defined by distances given in meters.

In general, the action may be thought of as taking place on an *x-y* coordinate "stage" (see Fig. 3). The amplitude of the direct (nonreverberated) part of the signal is scaled to be inversely proportional to the "distance" from the listener; maximum amplitude occurs when the sound is positioned at the same distance as the speakers. The delay of the direct signal is made dynamically proportional to the distance by use of a ZDELAY. The first parameter to ZDELAY is the scaled direct signal, and the second parameter is the delay time in samples, which is equal to the distance times the sampling rate divided by the speed of sound. Since only the delayed signal output from the ZDELAY is sent to the output, an accurate Doppler effect results as the distance changes. Computation of a Doppler effect in this manner is efficient, and is equally applicable to synthesized and recorded sounds.

Similar scaling was done on each channel of reverberation as well as in the reverberation instrument. Modulation of the amplitude and reflection times of each channel of the reverberation was proportional to the total distance that the sound would travel—from the apparent source, back to the "wall," then to the listener (who is, of course, assumed to be in the ideal listening position; sigh). Three "sources" of reverberation were calculated this way, one coming from each channel and one

Fig. 3. The imaginary spatial environment defined by the panning and reverberation instruments extends beyond the two speakers (black squares) shown at the bottom of the figure. The listener is ideally positioned at 0 along the bottom axis. The first five calls to the panning instrument move the sound in the shape of a five-pointed star. The sixth call continues traveling one and one-half times around a circle, and the final call holds the sound in the final position. This is the first such path that is used in Dreamsong, *beginning 29 sec into the piece.*



from both channels. Thus, if a sound were positioned on the left, the reverberation from the right channel would be delayed and scaled at a lower amplitude relative to the left channel reverberation. Robert Poor came up with the basic idea for this application of ZDELAY. Figure 3 is an example of how the reverberation instruments can be used to set up an imaginary space 15 m deep, with the speakers positioned 2.8 m from the listener in ideal position.

Amplitude Modulation with Complex Wave

From 70 to 100 sec in *Dreamsong*, a choral texture is gradually transformed into a texturé that sounds like strings and brass. This effect was carried out by amplitude modulation (AM) of the digitized vocal tones with a simple FM instrument (see Code Listing 8). The amounts of FM and AM modulation are gradually increased in parallel. For the effect achieved in *Dreamsong*, the frequencies of the FM oscillators were close to the center frequency of the signal being modulated. Due to the random vibrato in the original signal, the subjective density of the sound is also increased due to phasing and beating effects with the steady FM tone.

Pitch and Envelope Follower

Instrument PITCH (Code Listing 9) reads in a file and puts out its pitch and amplitude contours. It does this by looking for the peaks of the waveform within a window that is dynamically readjusted according to the current amplitude. It remembers the precise time each peak occurs and calculates the frequency every period. After checking to see that a reasonable value results, it sends out the average of the last four values calculated (for smoothness). It is a tricky business, and many of the variables need to be adjusted precisely for each individual file to

Code Listing 8. Instrument AMOD *amplitude modulates a signal using an FM-generated waveform.*

```
synth(F1);1 1, 999;                          ⟨sine wave
seg(F2);0 0, 1 100;                          ⟨ramp from (0,0) to (100,0)

instrument AMOD;                             ⟨Beg, Dur, Freq, Deviation
variable /Sig, /FmScl, /Out, /FM, /Am;
Sig ← readin(RD);                            ⟨input signal
FmScl ← oscil(1, mag/P2, F2);                ⟨modulation envelope
Fm ← oscil(FmScl*P4*P3*mag, P3*mag, F1);     ⟨FM modulator; P4 is the index
Am ← 1 + zoscil(FmScl, P3*mag+FM, F1);       ⟨FM carrier
Out ← (Sig * Am) / (1+Dev);                  ⟨amplitude modulate and rescale input signal
outa ← outa + Out;                           ⟨   by the FM deviation (see Chowning 1973)
end;
```

Code Listing 9. Instrument PITCH *puts out an amplitude and pitch contour of the input signal in variables Amp and Freq, which can be read by another instrument (see Code Listing 10). The Squelch removes low-level transients from the more or less silent parts of the input signal.*

```
array Freqs1[16],Freqs2[4],Del1,Del2[2000];
zero(Del1);zero(Del2);

variable /Freq, /Amp, /Samp;                 ⟨Global variables to be read by filter instrument
                                             ⟨   in Code Listing 10

instrument PITCH;
variable /Max,/Min,/MaxLim,/MinLim,/Peak,/Time1,/Time2,/Avg,/AvDev,/I,
  /S1,/S2,/S3,/S4,/Tmp1,/Tmp2,/Tmp3,/Interp,/Inc,/Cnt,/Cnt1,/Cnt2,/Sum,
  /Ampx,/Amp1,/Amp2,/TmpFreq,/CCR1,CCR2,Inc,Peak_Window,Freq_Window,Squelch;

i_only begin
  for I ← 0 step 1 until 15 do Freqs1[I] ← P3;        ⟨P3 = A guess at the average pitch
  for I ← 0 step 1 until 3 do Freqs2[I] ← P3;
  Avg ← Freq ← P3;
  Time1←Amp2←Max←Min←Peak←Interp←Cnt←Cnt1←Cnt2←S2←0;
  S1 ← S3 ← 1;
  CCR1 ← 2 * sRate/P3;                       ⟨controlled calling rates
  CCR2 ← 960;
  Inc ← 1/CCR2;                              ⟨increment for amplitude envelope interpolation
  Peak_Window ← 0.12;                        ⟨these variables need to be readjusted for
  Freq_Window ← 0.3;                         ⟨   different kinds of sounds
  Squelch ← 25;                              ⟨Probably not needed for many sounds
  end;

Samp ← readin(RD);                           ⟨get next sample of input signal
if Samp > Max then Max ← Samp;               ⟨Keep track of local maxamp & minamp
if Samp < Min then Min ← Samp;

value[CCR1](S1 ← 1);
```

Code Listing 9 (cont'd)

```
if S1 = 1 then begin                    {Update limits for finding peaks every CCR1 samples
  Tmp1 ← Max − Min;                     {Current peak-to-peak amplitude
  Tmp2 ← Tmp1 / 2;                      {Peak amplitude
  MaxLim ← Max − Tmp1 * Peak_Window;    {Amplitude window for positive peak
  MinLim ← Min + Tmp1 * Peak_Window;    {Amplitude window for negative peak
  Max ← Min ← S1 ← 0;                   {reset variables
  end;

value[CCR2](S3 ← 1);
if S3 = 1 then begin                    {update amplitude for contour every CCR2 samples
  Amp1 ← Amp2;                          {replace starting amp
  Amp2 ← Tmp2;                          {goal amp ← current peak from above
  if Amp2 < Squelch then Amp2 ← 0;      {may not want very low amplitudes
  Tmp3 ← Amp2 − Amp1;                   {net difference
  S3 ← Interp ← 0;                      {reset variables
  end;

Amp ← Amp1 + Tmp3 * Interp;             {Interp from one amp value to the next
Interp ← Interp + Inc;                  {increment is 1/CCR2

if Samp > MaxLim then                   {Let's look for the peak now
  if S2 = 0 then
    if Samp > Peak then
      begin
      Peak ← Samp;                      {update peak value
      Time2 ← Cnt;                      {note the time (in samples)
      end
    else S2 ← 1;
if Samp < MinLim then                   {good time to figure out current frequency
  if S2 = 1 then
    begin
    TmpFreq ← sRate / (Time2 − Time1);  {new frequency value
    Freqs1[Cnt1] ← TmpFreq;
    if TmpFreq > Avg * 2 then Freqs1[Cnt1] ← Avg;
    if TmpFreq < Avg/2 then Freqs1[Cnt1] ← Avg;
    Sum ← 0;
    for I ← 0 step 1 until 15 do Sum ← Sum + Freqs1[I];   {take average
                                                          {  of last 16 frequencies
    Avg ← Sum/16;
    if abs(Freqs1[Cnt1] − Avg) < Freq_Window * Avg then   {If new value is not too
      begin                                               {  weird then accept it
      Freqs2[Cnt2] ← Freqs1[Cnt1];                        {Add to 4 most current
      Cnt2 ← Cnt2 + 1; if Cnt2 = 4 then Cnt2 ← 0;         {  acceptable values
      Sum ← 0;
      for I ← 0 step 1 until 3 do Sum ← Sum + Freqs2[I];  {average these for smoothness
      Freq ← Sum/4;
      CCR1 ← 2 * sRate/Freq;
      end;
    Cnt1 ← Cnt1 + 1;                    {counter for freqs1 array
    if Cnt1 = 16 then Cnt1 ← 0;
    Time1 ← Time 2;                     {the two peak times
    Peak ← S2 ← 0;
    end;

Cnt ← Cnt + 1;                          {sample counter
end;
```

Code Listing 10. Instrument COMB *uses variables Amp and Freq from the* PITCH *instrument (Code Listing 9) to control amplitude and peak position of the comb filter* ZDELAY.

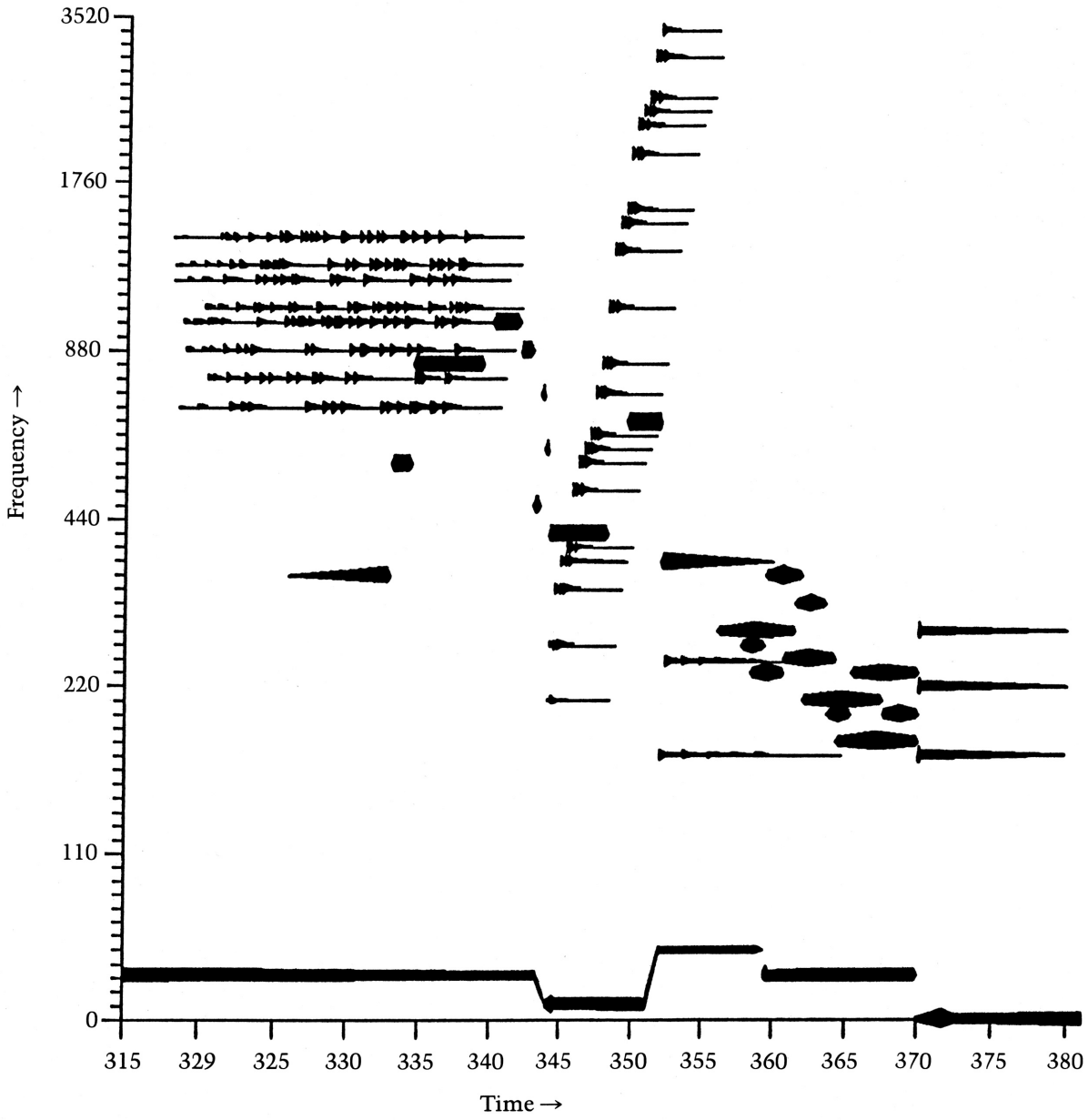
```
array BuffA,BuffB[512],BuffC[800];
seg(F1);0,1 0,18 0.6,34 1,50 1,100;
seg(F2);0,1 0,50 0.1,57 0.3,64 0.6,71 1,100;

instrument COMB;
variable /CrowdR, /CrowdL, /DelSigA, /DelSigB,
  /PtrA, /PtrB, /G, /CrowdAmp, /Del, /Voice;
i_only begin
  DelSigA ← DelSigB ← PtrA ← PtrB ← 0;
  zero(BuffA); zero(BuffB);
  Freq ← 200; Amp ← 500;
  end;
CrowdR ← readin(RD);                    ⟨read in crowd sound in stereo
CrowdL ← readin(RD);
VoiceAmp ← oscil(1, mag/P2, F2);        ⟨controls mix of unfiltered voice
AmpMod ← INTRP(1,Amp/500,F1);           ⟨controls modulation of crowd
CrowdAmp ← AmpMod * (1−VoiceAmp);       ⟨  by amplitude of voice

G ← oscil(1, mag/P2, F1);               ⟨controls gain of filter
Del ← sRate / Freq;                     ⟨Freq comes from PITCH instrument, Code Listing 9
DelSigR ← ZDELAY(CrowdR * G, Del, 512, BuffA,
  PtrA);                                ⟨stereo dynamic comb filters
DelSigL ← ZDELAY(CrowdL * G, Del, 512, BuffB,
  PtrB);

Voice ← DELAY(Samp, 800, BuffC);        ⟨delay voice to match delayed information
                                        ⟨  from PITCH
outa ← outa + (CrowdR+DelSigR) * CrowdAmp + Voice * VoiceAmp;
outb ← outb + (CrowdL+DelSigL) * CrowdAmp + Voice * VoiceAmp;
end;
```

be read in. In *Dreamsong*, this instrument was used (1) to check the pitch of some of the soprano notes and (2) to follow the pitch and amplitude of Dylan Thomas's voice, which was used to modulate a ZDELAY comb filter on the crowd sound at the very end. The filtering/mixing instrument (COMB) used for that is given in the next section.

Stereo Comb Filters Controlled by Pitch

Instrument COMB, given in Code Listing 10, reads in a stereo signal and filters it with a comb filter that has its first peak at the frequency provided by the PITCH instrument. It also modulates the amplitude of the input signal by the amplitude of the sound being read by PITCH and gradually interpolates between the sound being filtered and the sound doing the modulating. In *Dreamsong*, the modulating sound was Dylan Thomas's voice (taken from a recording) and it was used to filter the crowd sounds that serve as one of the motivic elements. The passage begins with the straight crowd sound, which gradually becomes more and more processed until it takes on the pitch and cadence of Thomas's characteristic speech. At this point it is gradually cross-faded with the unaltered voice.

Fig. 4. Graphic representation of central portion of Dreamsong. *The frequency scale is logarithmic, time is in seconds, and the thickness of each line is proportional to amplitude. Instruments* OMM, BELL1, *and* BELL2 *are playing here. This plot was produced by the author with the aid of graphics routines by James A. Moorer and the MUSBOX program of D. Gareth Loy.*



Acknowledgments

The author wishes to thank the many people who wrote and taught him to use effectively the programs used in *Dreamsong*, who helped him over conceptual or mathematical stumbling blocks, and who calmly gave him encouragement when he was mad as hell about some programming bug and wasn't going to take it any more. The many include John Chowning, John Grey, D. Gareth Loy, F. Richard Moore, James A. Moorer, Robert Poor, Loren Rush, Bill Schottstaedt, Ken Shoemake, Julius Orion Smith, Leland Smith, Tovar, and Paul Wieneke.

Thanks also to Stephen Volz, who helped me make that beautiful recording of the crowd, to soprano Marilyn Barber, who provided so much with so few notes, and to John Strawn for his editorial assistance.

References

Arnold, A. 1978. Private communication.

Chowning, J. 1973. "The Synthesis of Complex Audio Spectra by Means of Frequency Modulation." *Journal of the Audio Engineering Society* 21(7):526–534. Reprinted in *Computer Music Journal* 1(2):46–54, 1977.

Chowning, J. 1980. "Computer Synthesis of the Singing Voice." In *Sound Generation in Winds, Strings, Computers*, ed. J. Sundberg. Stockholm: Royal Institute of Technology, pp. 4–13.

Gardner, M. 1978. "Mathematical Games: White and Brown Music, Fractal Curves, and One-over-f Fluctuations." *Scientific American* 238(4):16–31.

Howe, H. S., Jr. 1975. *Electronic Music Synthesis*. New York: Norton.

Mandelbrot, B. 1977. *Fractals: Form, Chance and Dimension*. San Francisco: Freeman.

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

Portnoff, M. R. 1976. "Implementation of the Digital Phase Vocoder Using the Fast Fourier Transform." *IEEE Proceedings on Acoustics, Speech, and Signal Processing* 24:243–248.

Rush, L., and J. Mattox. Forthcoming. "Mama Don't Allow No Tape Machine 'Round Here: The Digital Audio Production Facility." In *Computer Music*, ed. C. Roads and J. Strawn. Cambridge, Massachusetts: MIT Press.

Schroeder, M. R. 1961. "Natural Sounding Artificial Reverberation." *Journal of the Acoustical Society of America* 10(3):219–223.

Smith, L. 1972. "Score: A Musician's Approach to Computer Music." *Journal of the Audio Engineering Society* 20:7–14.

Tovar. 1977. "Music Manual." Unpublished user's manual. Stanford: Center for Computer Research in Music and Acoustics.

Tovar, and L. Smith. 1977. "MUS10 Manual." Unpublished user's manual. Stanford: Center for Computer Research in Music and Acoustics.