Journal of New Music Research, 2013
Vol. 42, No. 1, 97–110, http://dx.doi.org/10.1080/09298215.2013.778890

Routledge
Taylor & Francis Group

The Development of Computer Music Programming Systems

Victor Lazzarini

University of Ireland, Ireland

Abstract

This article traces the history and evolution of Music Programming, from the early off-line synthesis programs of the MUSIC *N* family to modern realtime interactive systems. It explores the main design characteristics of these systems and their impact on Computer Music. In chronological fashion, the article will examine, with code examples, the development of the early systems into the most common modern languages currently in use. In particular, we will focus on Csound, highlighting its main internal aspects and its applications. The text will also explore the various paradigms that have oriented the design and use of music programming systems. This discussion is completed by a consideration of computer music eco-systems and their pervasiveness in today's practice.

1. Introduction

The evolution of Computer Music as an artistic and research discipline has, on one hand, instigated, and on the other, benefitted from, a series of technological developments. One of the most enduring of these is a class of software collectively known as *Music Programming Systems*. As these packages were shaped and developed by composers and researchers, they enabled and supported a host of discoveries and innovations, some of which were later appropriated by the music industry as essential components in the ubiquitous presence of digital audio observed in production and performance.

Computer Music Programming systems, as evoked by the name, are software packages that enable the use of a digital computer to create music programmatically. They provide an environment for defining the sequencing of events that make up a musical performance, both in static and dynamic ways, with great precision. In addition, and quite importantly, the audio signal processing involved in the syntheses or transformations is also programmable to a very fine degree of detail and accuracy.

Some systems only partially offer these capabilities. On one hand, we have packages that allow the implementation of event generation and scheduling algorithms, without offering sound synthesis means. On the other, audio libraries, signal processing specification languages and programmable software synthesizers also only partially offer the elements that make up a music programming system. It is the case, however, that users can benefit from the integration of a variety of tools into a multi-language environment for computer music, as discussed later in this article.

2. Early systems

The first direct digital synthesis program by Max Mathews, in 1957, MUSIC I, and its descendants (Mathews, 1963) paved the path to modern music programming systems. Their development was characterized by an increased generality and flexibility, from simple synthesis programs to fully-fledged programming languages. Some fundamental technologies were proposed along the way: the table-lookup oscillator, which is the single most common component in digital synthesizers, was introduced by Mathews in MUSIC II. The principles of the unit generator, a synthesis module and a compiler for computer instruments, which are still essential to modern systems, were introduced in MUSIC III (Mathews, 1961; Smith, 1991).

The original idea of the unit generator is one that has been applied almost universally in music systems. It figures not only in software synthesis programs, but also in hardware instruments, such as the classic analogue modular systems (which were in fact preceded by MUSIC III), and in the way most synthesizers are structured. Together with this, the *acoustic compiler* was also a breakthrough invention, in that it enabled an unlimited number of sound synthesis structures to be created in the computer. In that sense, the computer became not only a musical instrument, but a musical instrument generator. The principle of the compiler exists today in different forms in all modern music programming systems.

We observe in the software that followed MUSIC III the emergence of a set of design principles which are referred to as the MUSIC N paradigm, which we will examine in more detail later.

Correspondence: Victor Lazzarini, Music Department, National University of Ireland, Maynooth, Co. Kildare, Ireland.
E-mail: victor.lazzarini@nuim.ie

2.1 MUSIC IV

It is accepted that the first general model of a music programming system was provided by MUSIC IV, which ran on the IBM7094 computer, although some of the functionality seen in modern computer music synthesis systems had already been present in MUSIC III (Park, 2009). MUSIC IV was a complex software package, something that can be glimpsed from its programmer's manual (Mathews & Miller, 1964) and Tenney's (1963) tutorial. The software comprised a number of separate programs that were run in three distinct phases or *passes*, producing at the very end the samples of a digital audio stream stored in computer tape or disk file. Playback of such files was not a part of MUSIC IV.

The first pass took control data in the form of a numeric computer *score* and associated function-table generation instructions, in an unordered form and stored in temporary tape files. This is effectively a card-reading stage, with little extra functionality. However, at this point, FORTRAN-language subroutines could be applied to modify the score data before this is saved. Memory for 800 events was made available by the system. The first pass data was then input to the second pass, where the score was sorted in time order and any defined tempo transformations applied, producing another set of temporary files. Finally, a synthesis program was loaded for the third pass, taking the score from the previous stage, and generating the audio samples to be stored in the output file for subsequent playback via a digital-to-analogue converter.

The pass 3 program was created by the MUSIC IV acoustic compiler, written in BE FAP (Bell Fortran Assembly Program, the Bell Telephone Labs IBM7094 assembler). Effectively a block-diagram language compiler, this was used to connect the different unit generators defined in the system and available to the programmer to create an *orchestra*, the synthesis program, made up of *instruments*, the separate code blocks. Some basic conventions governed the MUSIC IV orchestra language, such as the modes of connections allowed, determined by each unit generator and how they were organized syntactically. An incipient type system was present, defining U (unit generator outputs), C (conversion function outputs), P (note parameters), F (function tables) and K (system constants) data types. During performance, the score would be allowed to run a certain number of parallel instances of each instrument, which was fixed in the orchestra definition. Each score event was required to be scheduled for a specific free instrument instance.

The MUSIC IV compiler allowed for fifteen unit generators: three types of oscillators; three addition operators, for two, three and four inputs; a multiplication operator; a table lookup operator; a resonance unit based on ring modulation; a second-order band-pass filter (resonator); two band-limiting noise generators, of sample-hold and interpolating types; linear and table-lookup envelope generators; and an output unit (Table 1).

An example from the MUSIC IV manual is shown in Table 2, where we can see the details of the orchestra language. Unit generator signals are referenced by U names (relating in this case to the order in which they appear), whereas score parameters and constants are denoted by P and C. It is a simple instrument, whose sound is generated by an oscillator (U5) to which an amplitude envelope (U1) and frequency modulation are applied. The latter is a combination of periodic (U2) and random (U3) vibrato signals, mixed together with the fundamental frequency (U4).

Table 1. MUSIC IV Unit generators.

1	OUT	output unit
2	OSCIL	standard table-lookup oscillator
3	COSCIL	table-lookup oscillator with no phase reset
4	VOSCIL	table-lookup oscillator with variable table number
5	ADD2	add two inputs
6	ADD3	add three inputs
7	ADD4	add four inputs
8	RANDI	interpolating bandlimited noise generator
9	RANDH	sample-and-hold bandlimited noise generator
8	MULT	multiply two inputs
10	VFMULT	table-lookup unit
12	RESON	ring-modulation based resonant wave generator
13	FILTER	second-order all-pole bandpass filter
14	LINEN	linear envelope generator (trapezoidal)
15	EXPEN	single-scan table-lookup envelope

Table 2. MUSIC IV instrument example (Mathews & Miller, 1964).

WAIL	INSTR	
	OSCIL	P4, C3, F1
	OSCIL	P6, P7, F1
	RANDI	P8, P9
	ADD3	P5, U2, U3
	OSCIL	U1, U4, F3
	OUT	U5
	END	
WAIL	COUNT	10
	FINE	

It can be argued that MUSIC IV was the first fully-fledged computer music programming environment, as the system allowed a good deal of programmability, which is specially true in terms of synthesis program design. In particular, the system structure models closely the principles of general-purpose programming, in the use of multiple passes for specific data processing tasks. Following its beginnings at Bell Labs, the package was brought over and recreated at Princeton University as MUSIC IVB (Randall, 1965) and then as MUSIC 4BF (Howe, 1996; Roberts, 1965), written in FORTRAN (and using that language for the programming of instrument definitions).

2.2 MUSIC IV variants

MUSIC IV lives on as MUSIC 4C (M4C), which is a C-language port, still available on modern UNIX-derived operating systems. M4C preserves the three passes of earlier versions, albeit with some modifications (Beauchamp, 1996).

Instruments are written in C and compiled in an orchestra as before, which is now built into a single program that does all the processing stages. Pass 1 initializes the output soundfile, reads the score file, set parameters and allocates instances of instruments. Pass 2, as before, does the time sorting of the score. Pass 3 runs the sorted score through a scheduler that instantiates instruments at the correct times.

In M4C, similarly to MUSIC IV, distinct programs will be created for each orchestra, with maximum available instances predefined for each instrument. Such orchestras can be written with instrument template files that combine some basic commands and C code blocks. An example of a very simple instrument is shown in Listing 1, a simple sinewave oscillator whose amplitude is driven by an envelope. The template fields specify what data is expected from the score (.scorecard), and the instrument code is divided into instantiation (.start), initialization (.note) and performance (.sample) sections, for which C code fragments are supplied.

Listing 1. M4C instrument example.

```
 prefix
OSCINS
 filename
OSCins c

 scorecard
pitch  4 00 12 00 octpitch 8 00;
amp  0 32000 amplitude 20000;
att  0 1 sec  05;
dec  0 1 sec  05;
sus1  0 1 level  7;
sus2  0 1 level 1;
rel  0 5 sec  2;
lfrac  0 1 balance 1;

 instrument
float freq, ampi, phase, lfraci;
ADeSR astate;

 globals
 #define FINALAMP  001

 start
phase = 0 ;
astate out = 0 ;

 note
fc = sipitch pitch);
ampi = amp;
adesr_set att,dec,DUR,sus1,sus2,rel,
FINALAMP,&astate);
lfraci = lfrac;

 sample
mono oscili adesr ampi, &astate1, fc, Sine,
 &phase);
```

From these template files, the INSDES translator generates C code that can be used to build a M4C program which includes the orchestra and the three passes outlined above. For this, the user will compile the C code and link to the other object files containing the code for the three passes into a complete command-line application. Each one of these separate compiled M4C programs will feature a given set of instruments available to the composer. To synthesize audio, this program is given a score file and an output soundfile. For orchestra design, M4C will require the C language toolchain to be present (preprocessor, compiler, linker), but as the system was designed for UNIX, this is taken for granted. In addition to the INSDES compiler, the package also includes an orchestra processor (NOTEPRO), which translates a textual score containing traditional staff notation music attributes (12-tone equal temperament pitches, metric rhythms, etc.) into a M4C numerical score. This system is still maintained for modern UNIX-like operating systems.

While strictly not a MUSIC IV variant, but nevertheless, a descendant, CMix (Pope, 1993) also employs similar principles to M4C. Here C-written instruments are compiled and linked into a main program that can be controlled by a score written in a scripting language, Minc. A main difference is that CMix does not employ an intermediary instrument specification as in M4C, but expects its orchestras to be written directly in C. For this, it provides a basic library of commonly-used unit generator functions.

MUSIC 360 was written at Princeton University by Vercoe (1973) for the large IBM 360 computer (Lefford, 1999). It was directly derived from MUSIC IVB and MUSIC IVBF, which were also developed at Princeton, and thus equivalent to those systems as a MUSIC IV variant. This program was taken to other IBM 360 and 370 installations, including for instance the one at MIT. However, as it was tied in to those large computer installations, it did not suit smaller institutions, and was not widely available.

The structure of MUSIC 360 is very similar to its predecessors, utilizing the basic three passes discussed above. Unlike the original MUSIC IV program, all passes are combined into a single 'load module' (the program) after the orchestra is compiled and linked to the library subroutines. Pass I reads the score and implements the carry preprocessing (where values can be 'carried over' from one note statement to another). Pass II sorts the score in its correct temporal order and applies any tempo warping defined in it. Pass III calls the orchestra, initializing its instruments and synthesizing the sound.

MUSIC 360 allowed any number of concomitant instances of instruments to be performed at the same time. An important innovation seen in this system is the clear definition of the initialization-time and performance-time stages, with separate programming routines set for each. The importance of this feature is evident in the separation of 'initialization' and 'performance' descriptions of each unit generator in the reference manual. Many of the modern systems will employ similar principles in their design. The orchestra syntax is very close to assembly language, based on operator codes (opcodes) representing the unit generators, something that will be inherited by MUSIC 11 and Csound.

Another advanced aspect of the language was that arithmetic expressions of up to 12 terms could be employed in the code, with the basic set of operations augmented by a number of conversion functions. Data types included 'alpha'-types, which could hold references to unit generator results (both at I-time and P-time), K-types, used to hold values from a KDATA statement (that was used to hold program constants), P-types for note list p-fields and U-types, which could be used to reference unit generator results (as an alternative to 'alpha' variables). There was scoping control, as symbols could be global or local, which included a facility for accessing variables that were local to a given instrument. The language also supported conditional control of flow, another advanced feature when compared to equivalent systems. Some means of extendability were provided by opcodes that were able to call external FORTRAN IV subroutines at I- or P-time. In addition, to facilitate programming a macro substitution mechanism was provided. MUSIC 360 was quite a formidable system, well ahead of its competitors, including MUSIC V.

An example of a simple orchestra program featuring an instrument based on an oscillator and trapezoidal envelope combination is shown in Listing 2. In this example, we can observe the use of U-types, which refer to the output of unit generators previously defined in the code. In this case U1 is a reference to the unit one line above the current. The PSAVE statement is used to indicate which p-fields from score cards will be required in this instrument. The ISIPCH converter is used to convert 'octave.pitch_class' notation into a suitable sampling increment, operating at initialization time only. OSCIL and LINEN are the truncating oscillator and trapezoidal envelope, respectively, used to generate the audio, the oscillator depending on function Table 1, which is defined as a score statement. Readers familiar with the Csound language will probably recognize the orchestra syntax, as much of it was the basis of MUSIC 11, and consequently Csound.

Listing 2. MUSIC 360 instrument example (Vercoe, 1973).

```
        PRINT NOGEN
        ORCH
        DECLARE SR=10000
SIMPL   INSTR 1
        PSAVE  3,5;
        ISIPCH P5
        OSCIL P4,U1,1
        LINEN U1, 03,P3, 06
        OUT U1
        ENDIN
        ENDORCH
        END
```

At the MIT Experimental Music Studio, Vercoe developed MUSIC 11 (Vercoe, 1981), a version of the MUSIC 360 system for the smaller PDP-11 minicomputer (Vercoe, 1983). As the PDP 11 popularity grew in the 1970s, and with the introduction of the UNIX operating system, the program was used at various institutions both in the USA and elsewhere for over two decades. As we would expect, not only many of the innovative features of MUSIC 360 were carried over to the new system, but also important concepts were pioneered here.

One of the main design aspects first introduced in MUSIC 11 was the concept of control (k-) and audio (a-) computation rates. In this mechanism, for each control sample, a vector of ksmps audio samples was produced. This made the system possibly the most computationally efficient software for audio synthesis of its time. Together with the i-time concept, this established the main operation principle of the orchestra language, dividing instrument action times into initialization and performance at two rates, which was realized in the three basic data types: i,k (scalars) and a (vectors). Global, local and temporary variables were available (marked as g, l or t). MUSIC 11 also featured dynamic memory management, as allocated instrument spaces, when free, could be taken over by new instances.

Listing 3 shows a version of the MUSIC 360 example, now in MUSIC 11 form. Although there are many similarities, some fundamental differences have been introduced in the language. The header declaration now include the definition of a control rate (kr) and the audio vector size (ksmps), as well as the number of output channels to be used (nchnls). The type system has been simplified, we observe the presence of the k- and a-type variables, which now have been defined to hold signals (and not just references to unit generator outputs) of control and audio forms, respectively. Also, taking advantage of the introduction of the control rate concept, and vector-based computation, the oscillator and envelope have had their positions exchanged in the synthesis graph: the envelope now is a control signal generator, rather than an amplitude processor, so the instrument can be run more efficiently.

Listing 3. MUSIC 11 instrument example.

```
        sr = 10000
        kr = 100
        ksmps = 100
        nchnls = 1

        instr 1
k1      linen p4, 03, p3, 06
a1      oscil k1, cpspch p5, 1
        out a1
        endin
```

With the introduction of the concepts of control rate and block-based (as opposed to sample-by-sample) processing of audio, two issues arise. Firstly, control signals are liable to produce audio artifacts such as amplitude envelope zipper noise, which are caused by the staircase nature of these signals, introducing discontinuities in the audio output waveform (and a form of aliasing that results in wideband noise). Secondly, score events are quantized at the control rate, and take place only at sample block (ksmp) boundaries, which can affect the timing precision in some situations. In fact, this issue has been noted as particularly problematic in more recent

realtime-oriented software such as Pure Data (Puckette, 2007), which employs the principle of block-based processing. To mitigate these effects, a balance between efficiency and precision needs to be reached, where the block size is small enough to prevent poor results, but high enough to be efficient.

2.3 MUSIC V

The culmination of Mathews' efforts at Bell Labs was MUSIC V (Mathews, Miller, Moore & Pierce, 1969), a new version of the system, mostly written in FORTRAN, which made it portable to other computer installations (in fact, it still can be run on modern operating systems). It still featured a three-pass process, however, the operation steps were more integrated than in MUSIC IV. The orchestra compilation step was now combined with pass 3, without the need to generate a separate synthesis program. FORTRAN conversion subroutines were also integral to the program code. Also the whole MUSIC V code was written in a single score, which contained both the note lists and the instruments. Unlike MUSIC IV, there was no maximum instance count for each instrument. Unit generators could be written either in FORTRAN or as separate machine-language subroutines.

MUSIC V, as its predecessor, provides simple orchestra data types: P (score parameters), V (scalar values), B (audio signal buffers) and F (function tables). Audio is generated in a series of sample blocks (or vectors), which by default held 512 samples. Vector-based processing became standard in most modern computer music systems (albeit with some notable options). Although no longer maintained, MUSIC V has been ported to modern systems using the gfortran compiler (Boulanger & Lazzarini, 2010). In Listing 4, we can observe a simple MUSIC V score, implementing the well-known Risset–Shepard tones (Risset, 1969a). The instrument employs three interpolating oscillators (IOS), generating an amplitude envelope (from a bell function), a frequency envelope (a decaying exponential) and a sine wave controlled by these two signals (in B3 and B4 respectively). Ten parallel oscillators are started, each with a 10% phase offset relative to the preceding one (tables are 512 samples long). Each NOT in the score defines an oscillator instance, with the first three parameters (P2, P3, P4) defined as start time (0), instrument (1), duration (14). Oscillator frequencies are defined by sampling increments (in P6 and P7). The top frequency of the decaying exponential is P6□ f_s/512, where f_s is the sampling rate. The amplitude and frequency envelopes have a cycle that lasts for 512/(f_s□ P7).

Listing 4. MUSIC V, Risset–Shepard tones, from Risset's (1969a) catalogue.

```
COMMENT -- RISSET CATALOGUE EXAMPLE 513
 -- INS 0 1;

IOS P5 P7 B3 F2 P8 ;
IOS P6 P7 B4 F3 P9 ;
IOS B3 B4 B5 F1 P25 ;
OUT B5 B1 ;
END ;

GEN 0 2 1 512 1 1 ;
GEN 0 7 2 0 ;
GEN 0 7 3 -10;

NOT 0 1 14 100 50  0001 0 0 ;
NOT 0 1 14 100 50  0001 51  51 1 ;
NOT 0 1 14 100 50  0001 102 2 102 2 ;
NOT 0 1 14 100 50  0001 153 3 153 3 ;
NOT 0 1 14 100 50  0001 204 4 204 4 ;
NOT 0 1 14 100 50  0001 255 5 255 5 ;
NOT 0 1 14 100 50  0001 306 6 306 6 ;
NOT 0 1 14 100 50  0001 357 7 357 7 ;
NOT 0 1 14 100 50  0001 408 8 408 8 ;
NOT 0 1 14 100 50  0001 459 9 459 9 ;
TER 16 ;
```

Pass I of MUSIC V scans the score (which includes both the instrument definitions and the note list proper) and produces a completely numeric representation of it. The second pass sorts the note list in time order and applies the CONVT routine, which can be used to convert frequencies to sampling increments etc. Finally, pass III schedules the events, calls the unit generators and writes the output.

Moore's (1990) Cmusic, a component of the CARL computer music package (Loy, 2002), was largely modelled on MUSIC V, inheriting much of its orchestra syntax (for instance, its data types, unit generator names and parameter arrangement, among other things). It also extended the possibilities by offering a C preprocessor, with macro substitutions and include statements, as well as some conversion operators (like hz and db, to covert frequencies into sampling increments and decibels into linear amplitudes, respectively, as well as letters to identify equal-tempered pitches). Moreover, its score offered the possibility of direct expression evaluation in parameter fields, which was not present in any of the earlier systems. Cmusic was designed to work with other CARL programs in the context of a UNIX environment and a medium-size multi-user computer hardware environment (e.g. DEC VAX systems), making heavy use of interprocess communication such as pipes and IO redirection. Audio data synthesized by the software could be used directly as input to other processes, such as a reverb effect or a plotting program, or streamed to output. This enabled the augmentation of the music programming system with UNIX shell scripting. Cmusic is no longer being maintained, although a version of the system has been ported to modern UNIX-like operating systems (Boulanger & Lazzarini, 2010).

Listing 5 shows an equivalent Cmusic instrument to Risset's catalogue example 513. This code is based on the discussion of Shepard tones in Moore (1990), and is somewhat more long winded, due to the use of a separate control instrument and the use of a trans unit generator to control the downward frequency glide. As it does not allow the glissando to wrap-around at the end of its range, each component needs to be

duplicated in the score, splitting the glissando into two stages. Parameters of 'note' statements are similar to MUSIC V. The example, however, demonstrates that, while a number of similarities exist between MUSIC V and Cmusic, in certain cases, scores do not translate in a strictly line-by-line fashion.

Listing 5. Cmusic, Risset–Shepard tones.

```
#include <carl/cmusic.h>

ins 0 control ; { amplitude  b5; and
 frequency  b6; control }
   iosc b5 p6 p5 f3 d ;
   trans b6 d d d 0,p7 p9 1,p8 ;
end ;

ins 0 component ; { single component of
 Risset-Shepard tone }
   mult b2 b6 p5 ;
   lookup b4 f2 b2 A -5; A 5; ;
   mult b3 b5 b4 ;
   iosc b1 b3 b2 f1 d ;
   out b1 ;
end ;

SINE f1; ;
SHEPENV f2; 10 2,
GEN4 f3; 9,0 -1  01,1 0  99,1 -1 1,0 ;

note 0 control 100 p4sec 1/10 A 5; A -5;
   ln p8/p7; ;

note 0 component 100 1 ; { this is a single
 glissando from A 5; to A -5; }
note 0 component 90 1/2 ; { this is one
 octave below }
note 0 component 80 1/4 ; { another
 octave below that one }
note 0 component 70 1/8 ; { etc }
note 0 component 60 1/16 ;
note 0 component 50 1/32 ;
note 0 component 40 1/64 ;
note 0 component 30 1/128 ;
note 0 component 20 1/256 ;
note 0 component 10 1/512 ; { this is the
 lowest component, only 10 secs long }
note 10 component 90 2 ; { it then wraps
 around above the first component }
note 20 component 80 4 ; { etc }
note 30 component 70 8 ;
note 40 component 60 16 ;
note 50 component 50 32 ;
note 60 component 40 64 ;
note 70 component 30 128 ;
note 80 component 20 256 ;
note 90 component 10 512 ;
```

Another notable successor to MUSIC V is Common Lisp Music (CLM) developed by Bill Schottstaedt from 1990 onwards, and still maintained, currently in its version 4 (Schottstaedt, 1996). The system uses the Common Lisp (CL) language as a glue, with a lower-level backend, which is used for audio computation. On its early implementation, the backend was based on assembly-language code for a DSP microprocessor (the Motorola 56000) fitted into a host computer (NeXT), but later, the C language was used, in ports of the language to more commonly-available computer systems (based, for instance, on Linux and Windows operating systems). Instruments are designed in CL, with the critical parts wrapped in a run macro, which compiles the Lisp code into the backend code. In modern systems, this is a C language module that is built as a dynamic library, loaded by CLM when audio processing is requested.

The Common-Lisp aspect of the code is its dominant feature, so in that sense, it does not resemble MUSIC V at all, even though it maintains its core principles. This can be seen in Listing 6, where an equivalent of the original Risset–Shepard tone design is implemented (based on code by Juan Reyes from the CLM sources).

Listing 6. CLM, Risset–Shepard tones.

```
definstrument shepard  beg dur amp &key
 dir 0;  incr  0.001;;
  let*   start  floor  * beg *srate*;;;
     end  + start  floor  * dur *srate*;;;;
     x 0.0;
     arr  make-array 10;;;
     do   i 0  1+ i;;;
     = i 10;;
    setf  aref arr i;  make-oscil
      frequency 0.0;; ;
  run
    loop for i from start to end do
     let   y 0.00;
        oscbank 0.0;;
     do   i 0  1+ i;;;
          = i  length arr;;;
       let   phoffset  + x  / i 10;;;;
         if  > phoffset 1;  setf phoffset
          - phoffset 1;;;
         setf y  - 4.0  * 8.0 phoffset;;;
         incf oscbank  *  exp  * -0.5 y y;;
              oscil  aref arr i;
                 hz->radians  expt 2.0
                  direction dir phoffset;;;
                          ;;;
       ;;
     incf x incr;
     outa i  * amp oscbank;; ;;;
   ;;
```

Unlike MUSIC V and many of the comparable systems, the CLM-compiled sound synthesis code does not employ vector-based processing of audio, but works on a sample (or sample-frame)-by-sample basis. The C-based CLM backend exists also as a complete C library (SndLib), which can be used independently in synthesis programs. CLM provides advanced support for manipulating note lists due to its use of the CL language. It can also be integrated with Snd soundfile editor that can serve as a frontend to the system.

3. Realtime-oriented systems

While music programming systems have traditionally been oriented to offline rendering of audio, the possibilities offered by modern general-purpose computing platforms for realtime audio processing have been explored by a number of systems. One of the earliest of these realtime-oriented systems, Max, was originally designed simply as a scheduler for an outboard synthesizer, the IRCAM 4X (Puckette, 2002). It eventually incorporated its own signal processing capabilities and lives on in two widely-used modern systems, Pure Data (PD) (Puckette, 2007), which is Free software, and MaxMSP (Zicarelli, 2002), which is a closed-source, non-Free package. As it is not possible to examine the source code of the latter system, our discussion here will be limited to the former.

Aside from the superficial aspect of being graphically-edited, a significant difference between PD and the systems discussed in the previous sections is the absence of the concept of a score. Here, the idea of instances of compiled instruments being initiated and controlled by note events does not exist. In fact, instruments, which live in PD patches, are generally single-instance, unless copies of the patch are made. Also, given the realtime orientation, running patches can be directly modified, which is not possible with compiled instruments. This leads to different modes of interaction and of composition, which lean towards user interaction and improvisation.

The MUSIC N heritage lives on in PD with the concept of unit generators as the central objects in the system. In a sense, it embodies the most direct form of the modular synthesizer metaphor, whereby a program is made of various boxes that can be interconnected in various ways, with the possibility of various external controls being applied to them. Its author disputes the application of the concept of programming environment to it (Puckette, 2002), citing the fact that, for economy reasons, the system lacks support for many aspects that are fundamental to creating programs. However, if we are considering the design of synthesizers or effects processors, PD is a legitimate music programming system, which can be used, of course, in context with other programming tools.

One of the essential aspects of operation in PD is its control scheduling. Triggering and passing data are unified in a single mechanism: simple triggers carry no data, while specific data types will work as specific triggers to object actions. This is embodied in PD's messaging system, which is used everywhere in the system to send sporadic control information from one object to another. It features a selector mechanism, where each message is headed by an identifier, that will trigger a method in a receiving object, producing an output.

In addition to these control messages, audio computation employs a completely separate system. This is effected by special unit generators, which react to a DSP message and implement vector-processing callbacks that are scheduled in a signal processing graph. The coexistence of an asynchronous messaging system and this ordered, sample-clock synchronous, audio system is uneasy. However, although it might offer difficulties in some situations, in practice, it can handle many use cases with no significant problems.

Another realtime-oriented music programming system is SuperCollider 3 (SC3) (McCartney, 2002). This is based on two components: an interpreted language (SCLang) and a separate Open Sound Control (OSC)-based software synthesizer (SCSynth). SCLang is a complete smallTalk-like object-oriented language that issue OSC commands over the network to SCSynth. It is designed to break the separation of orchestra (instrument) and score (note-lists), typical of MUSIC-N systems. Here, events and sound processing which can be programmed in an integrated way, and the language features give strong support for algorithmic composition. This is also a feature of another Lisp-based system, Nyquist (Dannenberg, 1997), although it does not share the same realtime orientation as SC3.

Given that precise timing is often an issue in realtime operation, the ChucK language (Wang, 2008) has been designed to incorporate the concept of time and duration as primitives in it. It allows strongly-timed operations, which are sample-level precise, at the expense of some efficiency (as processing of audio is done on a sample-by-sample basis). Earlier versions of the system had a significant number of performance issues, some of which have been addressed in the latest version released in September 2012 after a three-year hiatus. The system shows good potential for future development, although it is at the moment not as robust and complete as SC3 and PureData.

In this vein, it is also worth mentioning the development of extempore (Sorensen, 2012), which aims to provide a Scheme-based music programming language for realtime audio, based on an LLVM (Lattner & Adve, 2004) just-in-time compiler. This system shares similarities with CLM in that it is based on DSP code written in Lisp, but which in this case gets compiled and executed. The use of a just-in-time compiler is also a feature of Pure (Graef, 2009), which is a purely-functional language that allows the scheduling of Faust-based (Orlarey, Fober & Letz, 2009a) unit generators. All of these systems have allowed novel approaches to performance with computers, such as live coding, which is a form user interaction based on the recall and on-the-fly elaboration of code fragments to create and instantiate new instruments dynamically. It is interesting to note that the acoustic compiler, which in MUSIC III and IV was a separate stage (pass) of the software system, is now seamlessly integrated into the language syntax in the case of these realtime systems.

4. Csound

Csound is possibly the longest running heir to the early MUSIC N systems. It was developed alongside a number of equivalent (at the time) systems, such as Cmusic, Cmix and M4C, in the 1980s, all of which employed the C language, that became the standard for systems implementation. In time, it

developed into a much larger and multi-functional music programming environment and with the advent of its version 5 in 2006, it introduced a number of important concepts that were innovative for systems of this kind.

Csound appeared publicly in 1986 (mit-ems Csound), as a C-language port of MUSIC 11, very quickly superseding it. It inherited many aspects of its parent, but now integrating the orchestra compiler and loader into a single program. The original mit-ems Csound was based on three separate commands, scsort, csound and perf. The first command would sort the score; the second would compile and load the orchestra, and run the sorted score on it. The third command was just a convenient tool that called scsort and csound in a sequence. As with all programs of the era, mit-ems Csound was written in Kerninghan & Ritchie (K&R)-style C.

Csound was originally a very faithful port of MUSIC 11, so much that even today many programs for that language can still be run on modern versions of the system. Some small differences existed in terms of a collection of new opcodes, and the removal of some others. Also, the separation between temporary and local variables was removed in Csound, and a means of extending the language with new C-code opcodes was provided. However, beyond these small differences, the central concepts were shared between the systems.

In the 1990s, the centre of development of Csound moved from MIT to University of Bath. Realtime operation had been introduced to the system (Vercoe & Ellis, 1990) in the mit-ems version. From this, the system developed into an offline composition and realtime synthesis language with widespread applications explored in Boulanger (2000). Examining the source code of the 1995 version, we see the system re-written in the established ANSI C89 form of the language (from the original K&R C dialect), supporting realtime audio on a number of hardware platforms running UNIX: Sun, NeXT, DEC, SGI and HP. The program had been ported to PC-DOS, also with realtime audio via soundblaster soundcards. Separately, at Mills College, a version of the system for the Macintosh platform had been developed (Ingalls, 2000). In fact, at the end of the 1990s, the system had been ported to almost all modern general-purpose computing platforms with a C compiler.

In addition, separately it was ported to run on custom DSP hardware, in a closed-source version designated *extended Csound* (Vercoe, 1996). This version eventually became part of a commercial project of Analog Devices, inc., to supply simple synthesizers for embedded applications (Vercoe, 2004). Meanwhile, a large international developer community was involved in expanding the open-source system, which eventually came under the Lesser GNU Public License (and thus Free software). Many new unit generators were developed for it, culminating on the Csound 4.23 version in 2002. The consensus among the community was that the system required a major re-engineering to be able to move forward. A code freeze was established so that the new Csound 5 system could be developed.

4.1 Csound 5

The main goals of the development of Csound 5 were: to provide a clean, re-engineered system, away from the original monolithic program; to support re-entrant instances of the system in a dynamic-loadable library; and to provide a plugin mechanism for various aspects of the system (unit generators, utilities, function tables) (ffitch, 2006). A final goal, only achieved in later versions of the system, was to provide a new orchestra language parser, based on the bison compiler compiler and the flex lexer, which would be maintainable and extendable. The main motivation behind this work was to adopt more up-to-date design paradigms (see Section 5 below) and to allow for a variety of uses for the system. Csound 5.00 was eventually launched in 2006, 20 years after the first mit-ems release in 1986.

With this version, Csound became a programming library, which could be used to embed the system as a synthesis engine for a variety of applications. In fact, since version 4.21, Csound had been released as a library with a public application programming interface (API), however with a number of limitations (for instance, only a single copy of the engine could be used by hosts). In Csound 5, all of these issues were removed and the system became fully re-entrant.

The orchestra language has also undergone some important transformations, although always maintaining backwards compatibility. Facilities for the development of opcodes in the orchestra language itself were provided in the form of user-defined opcodes (UDOs), which allowed for local vector sizes (and single-sample feedback), as well as recursion. New data types were introduced for self-describing frequency-domain data and generic arrays. Another significant change to the system as a whole is the provision of alternative means of instrument instantiation, rendering the need for a separate sorted score optional. Listing 7 shows an example of a Csound program for spectral morphing, which uses some of its newer language features.

Listing 7. Csound 5 spectral processing example.

```
<CsoundSynthesizer>
<CsOptions>
-o dac -i adc
</CsOptions>
<CsInstruments>

nchnls = 2

chnset 0.5, 'ampmorph' ; amp morphing init
chnset 0.5, 'freqmorph' ; freq morphing init

alwayson 1

instr 1
    iws = 2048 ; window size
    ihs = iws/8 ; hopsize
    a1, a2 ins  ; inputs
```

```
    kam  chnget 'ampmorph'  ; amp morphing
    kfm  chnget 'freqmorph' ; freq morphing
    fsig1 pvsanal a1,iws,ihs,iws,1
     ; chn 1 analysis
    fsig2 pvsanal a2,iws,ihs,iws,1
     ; chn 2 analysis
    fmorph pvsmorph fsig1,fsig2,kam,kfm ; morph
    am     pvsynth fmorph ; PV resynthesis
       outs am,am
endin

</CsInstruments>
<CsScore>
</CsScore>
</CsoundSynthesizer>
```

In the wake of these changes, many elements were introduced to provide a flexible use of the engine. The API made it possible to use the system from a variety of programming languages and various third-party host/frontend applications were developed. Listing 8 demonstrates a trivial Python frontend for Csound. Such facilities allow for Csound to be inserted into a variety of Computer Music composition and performance scenarios.

Listing 8. Csound 5 python example.

```
import sys
import csnd
csound = csnd Csound ()
csound Compile sys argv 1[)
 while not csound PeformBuffer ()
    pass
csound Reset ()
```

These developments enabled, amongst other things, the porting of Csound to mobile operating systems (the Mobile Csound Platform, MCP (Yi & Lazzarini, 2012)), and its use as a plugin in a variety of formats, and in the One Laptop Per Child project, running on the XO computer (Lazzarini, 2008). Such applications were, in fact, somewhat forerun by the adoption of a Csound-like language, Structured Audio Orchestra Language (SAOL), in the MPEG4 standard for structured audio encoding (ISO, 1998).

5. Paradigms

The development of modern music programming systems, which was traced in the previous sections, has relied on a number of established paradigms. Some of these are borrowed from general programming practice, such as object orientation, others emerged from the evolution of music systems themselves. In this section, we would like to explore some important paradigms that have influenced these software packages.

5.1 The MUSIC N paradigm

The MUSIC N paradigm is an emergent set of properties from the main music programming systems discussed above. Although the concept has been used widely, there has never been a full definition for it. The essence of the paradigm is that an environment for Computer Music should be programmable. The existence of a compiler in MUSIC III, hailed by Mathews (1961) as its most important breakthrough, is something that survives in one shape or another in all of its descendants. This had the role of translating a symbolic description of a synthesis signal flow into a DSP graph, which would run in the computer. The term 'acoustic compiler', coined by Mathews, defines this element of the paradigm.

In some systems, instead of a compiler, we might find some sort of an interpreter, that would assemble the graph by creating and connecting instances of the processing boxes, the unit generators. These are also important components of the paradigm. Their existence is universal among music programming systems. They have also appeared elsewhere in electronic music instruments, such as the modular synthesizer, such is the pervasiveness of the concept. The table lookup oscillator, sometimes defined as the workhorse of digital synthesis, is an example of unit generator that is ubiquitously present in music programming. Given its importance, it can be argued that it also plays a central role in the paradigm, together with the concept of function tables and their generation (GEN) routines.

A more disputed aspect of the MUSIC N paradigm is the presence of a separate score data specification language. While the most traditional systems, like the original MUSIC IV-V, Cmusic, CMix, MUSIC4C support and depend on the principle, other systems do not espouse it, at least directly (PD, SC3, Nyquist). Systems like CLM try to integrate it by making the score and orchestra language the same (Common Lisp), although maintaining a degree of separation between them. We propose that the essential point of this aspect of the paradigm is not so much the existence of a separate score language, but the principle of instrument classes, from which objects can be instantiated programmatically during performance to produce audio. Most of the systems described here would embrace this principle. The case of PD is, however, particular, in that it does not allow direct instantiation, requiring the user to create copies of instruments manually.

5.2 Object-oriented programming

From the perspective of programming, the MUSIC N paradigm is then realized through the principles of Object-Oriented Programming (OOP). This realization might be partial, with regards to what some view as the complete set of concepts involved in this paradigm. However, the essential ideas are present in all systems.

There is a general consensus that the first example of the OOP paradigm appeared in the Simula 67 language (Nygaard & Dahl 1978). But it can be argued that systems like MUSIC

IV already embodied the central aspects of object orientation, the idea of classes, as a description or template, and objects, the concrete instances of these. MUSIC IV provided these at two levels. Firstly, the unit generator can be thought of as a class, whose objects are instantiated in the instrument definition. Secondly, the instrument, as pointed out above, is also a class that can be instantiated a number of times by score events (the 'note cards' in MUSIC IV).

The OOP paradigm, it seems, serves the work involved in music programming very well, as it does in the case of Simula, the tasks of simulation. This is, in part, because much of the activities of music composition and performance can be modelled in object-oriented terms. For traditional music, we have the abstract idea of a note, of which instances a musical composition is made. This works even in extended concepts of music, where notes are not a valid concept anymore, but where sound objects, textures, gestures, etc., can all be decomposed into objects, which we can describe in terms of types. So it is not a coincidence that the principle was discovered to fit the purposes of Computer Music very early on.

Other aspects of OOP are not supported in general by MUSIC N languages, but make an appearance in various systems. For instance, SC3 has a full set of features that have been modelled on the SmallTalk language, supporting various ancillary aspects of OOP. Kyma was a pioneer system, which offered many features of object-orientation (Scaletti, 2002). Csound 5 is a fully object-oriented programming library, which has both a C and a C++ API, as well as interface extensions and wrappers for various OOP languages. For the development of large music systems, this paradigm is essential. It has been found particularly useful as a means of supporting loadable modules, which use the concept of classes to provide a common interface between host systems and plugins. Such interfaces are found not only in music programming systems, such as PD, SC3 and Csound, but in architectures for various programs (Linux Audio Developer's Simple Plugin API, LADSPA; Virtual Studio Technology, VST; etc.).

5.3 The multi-language paradigm

At the core of music programming lies the multi-language design paradigm, that of the use of a domain-specific language (DSL, in this case, the one provided by the music programming system) and, at different times, a system-implementation language (C/C++, FORTRAN) and/or a scripting language. This arrangement is very similar to what has been advanced by authors such as Ousterhout (1998), where glue code is used to connect components implemented in languages that are closer to hardware. Such glue code can take the form of the instrument design language and any scripting that might be involved in, for instance, score generation.

This approach has been present in Computer Music since MUSIC IV, where the event generation for a given score was completely separated from the design of the synthesis program which would receive the score (Mathews & Miller, 1964). The MUSIC N languages themselves can be seen as glue code connecting unit generators implemented in a lower-level language (Mathews, Moore & Risset, 1974). This concept was extended to be the basis for the CARL system, where components were connected using shell scripting (Loy, 2002). Similarly, it was used in Cmix (Pope, 1993), where synthesis programs written in C were glued by a score code written in MinC. More recently, it is the basis of the meta-programming principles embodied by Faust (Orlarey, Fober & Letz, 2009a). Since the advent of Csound 5, it has become an organic mode of operation for users of that system, where Csound can be part of a system involving a number of other languages. In fact, Csound itself can embed other languages (such as Python and Lua).

A multi-language approach in music programming may also allow users to strike a balance between generality and efficiency. Depending on the task, the point of entry for the programmer can be chosen to be at different grades of complexity. A lower, more general, level requires more involved code design, but allows a wider range of results. On the other hand, if we can operate at a higher, more specialised and specific, plane, the process would be more efficient from the perspective of programming effort.

5.4 The open source dimension

The mechanism that propelled such momentous development in music programming, such as the one described in this paper, is indisputably the availability and exchange of source code. All the MUSIC N-derived systems were *de facto* open-source, even if the term and concept were never current at the time. However, we can see that these were given to any interested parties, with pleasure. These enabled the cross-fertilization of ideas, with the results clearly seen in the evolution of Computer Music as a research and artistic discipline.

The account of the earlier days of computer music perhaps demonstrate how the open source paradigm not only benefits the community of users, but also the evolution of the software itself. A large user base and developer community will facilitate its permanence and continued usefulness. Csound is a good demonstration of this principle, as it has benefitted from a wide adoption to continue to develop over a period of more than 25 years. As with similarly widespread systems, such as PD and SC3, shared use and development has shaped what these packages have become.

As an example of these benefits, we can compare the fate of two major systems of the same era: Cmusic and Csound. In 2006, 20 years after its first release, Csound was going strong with the release of the all-powerful version 5, whereas Cmusic had effectively disappeared. Some of the reasons for this were put forward very well by Richard Dobson, earlier on in 1999: 'I suppose one significant difference between Cmusic and Csound, which might explain the relative lack of "modern" opcodes in the former, is that while Cmusic has largely remained the property and product of F.R. Moore, Csound has

reaped the benefit of a large, skilful, enthusiastic and mostly unrestrained net-wide user and development group' (Dobson, 1999).

The open-source paradigm has emerged from all of the experiences of developers in trying to evolve the sharing of ideas and the building of communities around software systems. Much of it has been put to the fore by the Free Software Foundation and its originator, Stallman (2010). Although his principles go beyond the simple general guarantees for the copying, modification and distribution of source code, there is much that is common between what is Free software and what is open source.

One of the important aspects that guarantees the status of open source software is the use of an appropriate license. In the case of Csound, the licensing issue was a question that took a while to be solved. The software had been originally released under the MIT License, which in the view of many people in the community was problematic and confusing. Csound was moved to the (Free-software) Lesser GNU Public License (GPL), for its version 5 release, after consultation with all its contributors and negotiations with the original copyright holders.

One of the major issues regarding the uncertain license status of certain open-source software was the fact that many developers did not really understand the need for licenses. For some, this was due to ignorance, and for others, like John ffitch himself and so many others, it was because they came from an era where the concept of open-source did not exist. As he explained, 'in those days, the question was not there as everyone shared their software. In fact, when we managed to get a program to work we were so happy that we wanted to give it to everybody' (ffitch, 2005).

With Csound, as with other systems, it is not only the case that the source code for the music programming system is available, but also that the Csound-language code itself is freely distributed. Catalogues of Csound instruments were around for people to use, learn and modify. From Richard Boulanger's Toots to the well-known Amsterdam Catalogue, a wealth of code has been made available for the community. All these things are incredibly useful for people learning computer music, composition, signal processing, etc. For this community, such resources are often more important than the software source code itself.

5.5 Parallel processing

The presence of multicore processors as a standard feature of computing systems, and the availability of parallel graphical processing units (GPUs), has urged the consideration of parallel and concurrent architectures for audio and music. Here, we are not so much speaking of an established paradigm, but of an emerging challenge. Music and audio have a number of eminently parallelizable processes, so for some applications there is quite a good fit (ffitch, Dobson & Bradford, 2009). For instance, DFT-based algorithms, additive synthesis, certain types of filterbanks, some types of physical models, linear time-invariant processing (e.g. convolution), are all examples of good applications for parallel processing, and, indeed, this has been demonstrated in practice (Battenberg, Freed and Wessell, 2010; Battenberg & Avizienis, 2011; Webb & Bilbao, 2012). Algorithms involving feedback/recursion are less prone to these implementations, but as components of a larger process (i.e. a series of instrument instances), would of course benefit from parallelization on a larger level.

Apart from the pioneer effort of porting Csound to transputers (Manning, Berry, Bowler, Bailey & Purvis, 1990), music programming systems have not been customarily implemented in parallel forms. This has changed significantly in the past few years, with efforts on allowing widely-used systems to take advantage of multicore processors. Examples of this are found in parallel Csound (ffitch, 2009), the supernova implementation of SCSynth (Blechmann, 2011), both of which are present in the official releases of these software, and in custom versions of Pure Data, such as the one implemented at the University of Berkeley (Colmenares, Saxton, Battenberg, Avizienis, Peters, & Asanovic, 2011).

Approaches to concurrent performance vary, but there is a consensus that the major problem to be solved by music programming systems is that of communication/synchronization between processes/threads and their granularity level. Also, due to the particular aspect of time dependency in realtime audio, systolic architectures (standard in many other concurrent applications) are in general not very practical, as they introduce undesirable latencies in the process.

Algorithms implemented with a fine level of granularity, which is translated into the size of independent processing blocks, suffer from inefficiencies due to communication overheads, as reported for instance in Orlarey, Letz and Fober (2009b). The level at which parallelism is introduced is therefore crucial. It is generally understood that the best practice is to consolidate unit generators at instrument level (ffitch, 2009), which is also referred to as an aggregated task level in Colmenares, Saxton, Battenberg, Avizienis, Peters, and Asanovic (2011), before splitting these to multiple cores.

A crucial question is also to do with how concurrency is to be presented to the user. In general, two opposing views exist. One states that the implementation of parallelism is a metaprogramming problem and a compiler should resolve it automatically, a strategy adopted by Csound (ffitch, 2009), and the other that users should be responsible for deciding how the processes are to be split between cores, seen in Supercollider (Blechmann, 2011). In the first case, the advantage is that code analysis can also include consideration of load balancing and that sometimes what appears to be an obvious solution from the user perspective does not actually result in an efficient implementation. On the other hand, allowing users to program parallelism directly is perhaps more flexible and allows comparative studies of concurrency in audio processing algorithms.

6. Ecosystems for music programming

A software ecosystem can be defined as a group of packages that bear a certain cooperative and/or complementary relationship and reside in the same or in related environments or platforms. The model for music programming in the twenty-first century is based on such an ecosystem of applications, which will provide support for a variety of tasks that are of value to Computer Music. Two of these can be identified as DSP programming/music application development, and composition. Music programming systems have to be able to be flexible enough to allow for the emergence of this eco-system, either by participating in it, in a multi-application, multi-language environment, or by supporting (i.e. providing the tools for) the development of cooperating applications.

6.1 DSP programming and application development

The translation of signal processing algorithms into running code has been a preoccupation of music programming since MUSIC III. Mathew's principle of unit generators as building blocks for a synthesis program has already been discussed above as a central aspect of computer music environments. Another important feature for the realization of DSP algorithms is support for elementary operations on signals, which was provided in terms of support for expressions in MUSIC360. Although this existed in MUSIC IV and MUSIC V via addition and multiplication, the possibility to use expressions, combined with some simple functions, allowed a compact way to translate formulae. In fact, Vercoe (1983) states as one of the core aspects of a music programming language is the possibility to reproduce the operation of a unit generator using language primitives.

In this sense, the modern systems such as Csound, SC3 and PD enable DSP programming to a fine degree of detail. This is particularly important as it enables the use of these systems in a scientific and research context. Notably, they might be a better option to demonstrate audio and music signal processing ideas than the usual general-purpose modelling software (e.g. octave, or even python with its scientific packages), as they can provide real-world implementations (Lazzarini, Yi & Timoney, 2012) as well as code that can be easy to read.

In some systems, some essential operations are awkward: for instance single-sample delays in PD are possible, but not intuitive to program. However, by taking advantage of the multi-language scenario, in these cases, it is possible to use a system that is designed to translate DSP flowcharts and algorithms, such as Faust, to compile unit generators for a host music programming system (such as PD, Csound or SC3). That way, a large design problem can be broken down into separate components, utilizing the principle of separation of concerns (Damasevicius & Stuikys, 2002). Depending on the task at hand, a number of alternatives exist for DSP programming in a software ecosystem that includes computer music languages.

This principle is even more evident when we come to the task of application development utilizing music programming systems. While environments such as SC3, PD and Csound can provide, to various extents, support for the authoring of music applications on their own, there is more flexibility and scope to using them in a multi-language scenario. Taking the case of Csound, for instance, it is very straightforward to create host applications in a variety of platforms, using either scripting or implementation languages, which use its music programming capabilities. With software developer kits (SDKs) such as the Mobile Csound Platform, Csound becomes the audio/music component in an agile development environment, in combination with the application-authoring languages (Objective-C for iOS; Java for Android and Web apps).

This scenario demonstrates a typical application ecosystem with various cooperating components. Here users can first prototype and develop their signal processing algorithms using Csound through one of its host frontends, using Faust (or C/C++) for creating new unit generators for the language if necessary. In complement, they can author their target application for a given system (desktop, mobile, web-based), add the Csound engine, which will load and run the processing code. This application can in turn be made to cooperate with other apps: for instance, a mobile app might be able to communicate with a desktop app, which can be used to design or make alterations to the Csound code, or share projects between the two in a variety of ways.

6.2 Composition

Composition has always been the prime area of music programming. Early non-realtime systems have been mainly composition environments, although they have also been used in signal processing research (a classic case is Chowning's (1973) development of FM synthesis) and acoustics (e.g. Risset's (1969b) studies of Shepard tones). Thus, all modern music programming systems, continue in this tradition, even though some have been originally designed with performance in mind, as in the case of PD (Puckette, 2002).

The tasks involved in the composition of Computer Music are a mix of instrument development, DSP design (as discussed above), and event programming/scheduling, which in many situations form a single continuum. Facilitating the integration of these tasks is important for some composers, whereas for others, a more clear separation between them is required. Music programming systems should be flexible enough to accommodate these two distinct needs, even if they are designed to give more support to one of them.

Even though some systems are designed to offer powerful support for algorithmic composition on their own as in SC3 and in CLM and Nyquist (due to their underlying language, Lisp), it is quite common to see the multi-language paradigm at play in this scenario. Composers will often feel at home using a well-designed general purpose language, such as for instance, Python, in music creation, which would in some way interface with their preferred music programming system. For

this combination, the often criticized split between score and orchestra programming (which is also a type of separation of concerns as discussed above) is very welcome. It is very unwieldy to combine programming of events in a separate language with a music environment such as SC3, unless we are to discard completely its language component. However, we must of course note that, in some cases the combination of algorithmic composition and sound synthesis, with roots in the practice of composers such as Iannis Xenakis and Barry Truax, can benefit from such integrated environments (see for instance DiScipio, 1994).

It might be fair to consider that the simpler the score language, the easier it is to find means of integrating it within an ecosystem for composition. It allows for substantial customization by composers, who might find it awkward to write directly in any music programming system. For this purpose, Csound not only allows tight integration via its API, but also provides a simple mechanism to allow plugin score processors, written in any language, to be embedded in its code. These only need to output a legal Csound score as ASCII text to the standard output, and can be used to translate any code the composer writes in place of the usual score.

7. Conclusions

Following the discussion in this article, we can reach a more well-defined concept of what constitutes a music programming system. Such a software package should be one that allows users to develop musical applications and compositions of various forms, with full support for a variety of synthesis and processing methods. It needs to be extendable, and allow signal processing algorithms to be expressed in its language, but also capable of integration into an ecosystem of applications and environments. Support for object-oriented programming approaches is also a basic characteristic of these systems. The open source dimension is an important aspect, which facilitates the study and dissemination of ideas, and one which enabled the significant developments to date.

In the 50-odd years of their existence, the software packages discussed here have brought an exceptional richness to all areas of Computer Music, from research to composition and performance. We can only hope that the next fifty years of development can bring just as many new possibilities as the ones provided so far.

References

Battenberg, E., & Avizienis, E. (2011). Implementing real-time partitioned convolution algorithms on conventional operating systems. In Proceedings of the 14th International Conference on Digital Audio Effects DAFx-11, IRCAM, Paris, 313–321.

Battenberg, E., Freed, A., & Wessell, D. (2010). Advances in the parallelization of music and audio applications. In Proceedings of the ICMC 2010, New York, USA, 349–352.

Beauchamp, J. (1996). *Introduction to MUSIC 4C*. Urbana, IL: School of Music, University of Illinois at Urbana-Champaign.

Blechmann, T. (2011). Semantic Aspects of Parallelism for SuperCollider. In Proceedings of the Linux Audio Conference 2011. Maynooth, Ireland, 29–32.

Boulanger, R. (Ed.) (2000). *The Csound Book*. Cambridge, MA: MIT Press.

Boulanger, R., & Lazzarini, V. (Eds.) (2010). *The Audio Programming Book*. Cambridge, MA: MIT Press.

Chowning, J. (1973). The synthesis of complex audio spectra by means of frequency modulation. *Journal of the Audio Engineering Society, 21*, 526–534.

Colmenares, J.A., Saxton, I., Battenberg, E., Avizienis, R., Peters, N., & Asanovic, K. (2011). Real-time musical applications on an experimental operating system for multi-core processors. In Proceedings of ICMC 2011, Huddersfield, UK, 216–223.

Damasevicius, R., & Stuikys, V. (2002). Separation of concerns in multi-language specifications. *Informatica, 13*(3), 255–274.

Dannenberg, R. (1997). Machine Tongues XIX: Nyquist, a language for composition and sound synthesis. *Computer Music Journal, 21*(3), 50–60.

di Scipio, A. (1994). Formal processes of timbre composition hallenging the dualistic paradigm of computer music. In Proceedings of the ICMC 1994, Aarhus, Denmark, 202–208.

Dobson, R. (1999). Computer Music Books (was Re: cmusic). Email to the music-dsp list: http://www.music.columbia.edu/pipermail/music-dsp/1999-June/034758.html

ffitch, J. (2005). On the open-source question. personal communication.

ffitch, J. (2006). On the design of Csound 5. In Proceedings of 4th Linux Audio Developers Conference, Karlsruhe, Germany, 79–85.

ffitch, J., (2009). Parallel execution of Csound (pp. 16–21). Montreal, Canada: In Proceedings of the International Computer Music Conference, Montreal, Canada, 16–21.

ffitch, J., Dobson, R., & Bradford, R. (2009). The imperative for high-performance audio computing. In Proceedings of the Linux Audio Conference 2009, Parma, Italy, 73–80.

Graef, A. (2009). Signal processing in the pure programming language. In Proceedings of the Linux Audio Conference 2009, Parma, Italy, 137–144.

Howe, H. (1966). A report from Princeton. *Perspectives of New Music, 4*, 68–75.

Ingalls, M. (2000). Improving the composer's interface: Recent developments to Csound for the Power Macintosh computer. In: R. Boulanger (Ed.), *The Csound Book*. Cambridge, MA: MIT Press.

ISO/IEC JTC 1/SC 29/WG 11, (1998). *Information Technology - Coding of Audiovisual Objects - Low Bitrate Coding of Multimedia Objects*. London: International Standards Organisation.

Lattner, C., & Adve, V. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. In Proceedings of the (2004). *International Symposium on Code Generation and Optimization (CGO'04)*. CA: Palo Alto.

Lazzarini, V. (2008). A toolkit for audio and music applications in the XO computer. In Proceedings of the International

Computer Music Conference 2008, Belfast, Northern Ireland, 62–65.

Lazzarini, V., Yi, S., & Timoney, J. (2012). Digital audio effects on mobile platforms. In Proceedings of 15th International Conference on Digital Audio Effects (DAFX-12), York, UK, 287–292.

Lefford, N. (1999). An interview with Barry Vercoe. *Computer Music Journal*, *23*, 9–17.

Loy, G. (2002). The CARL system: Premises, history and fate. *Computer Music Journal*, *26*(4), 23–54.

Manning, P., Berry, R., Bowler, I., Bailey, N., & Purvis, A. (1990). Studio report, University of Durham, England. In Proceedings of the ICMC 1990, Glasgow, UK, 415–416.

Mathews, M. (1961). An acoustical compiler for music and psychological stimuli. *Bell System Technical Journal*, *40*, 553–557.

Mathews, M. (1963). The digital computer as a musical instrument. *Science*, *183*, 553–557.

Mathews, M., & Miller, J. E. (1964). *MUSIC IV Programmer's Manual*. Berkeley Heights, NJ: Bell Telephone Labs.

Mathews, M., Miller, J. E., Moore, F. R., & Pierce, J. R. (1969). *The Technology of Computer Music*. Cambridge, MA: MIT Press.

Mathews, M., Moore, F. R., & Risset, J. C. (1974). Computers and future music. *Science*, *183*, 263–268.

McCartney, J. (2002). Rethinking the computer music language: Supercollider. *Computer Music Journal*, *26*(4), 61–68.

Moore, F. R. (1990). *Elements of Computer Music*. Engelwood Cliffs, NJ: Prentice-Hall.

Nygaard, K., & Dahl, O. (1978). The development of the SIMULA languages. *ACM SIGPLAN Notices*, *13*, 245–272.

Orlarey, Y., Fober, D., & Letz, S. (2009a). Faust: An efficient functional approach to DSP programming. New Computational Paradigms for Computer Music. Sampzon, France: Edition Delatour.

Orlarey, Y., Letz, S., & Fober, D. (2009b). Adding automatic parallelization to Faust. In Proceedings of the Linux Audio Conference 2009, Parma, Italy, 81–92.

Ousterhout, J. (1998). Scripting: Higher-level programming for the 21st century. *IEEE Computer*, *31*(3), 23–30.

Park, T. (2009). An interview with Max Mathews. *Computer Music Journal*, *33*, 9–22.

Pope, S. (1993). Machine Tongues XV: Three packages for software sound synthesis. *Computer Music Journal*, *17*(2), 23–54.

Puckette, M. (2002). *Max at seventeen. Computer Music Journal*, *26*(4), 31–43.

Puckette, M. (2007). *The Theory and Technique of Computer Music*. New York: World Scientific.

Randall, J. K. (1965). A report from Princeton. *Perspectives of New Music*, *3*, 84–92.

Risset, J. C. (1969a). *An Introductory Catalogue of Computer Synthesized Sounds*. Berkeley Heights, NJ: Bell Telephone Labs.

Risset, J. C. (1969b). Pitch control and pitch paradoxes demonstrated with computer-synthesized sounds. *Journal of the Acoustical Society of America*, *146*, 88.

Roberts, A. (1965). MUSIC 4BF, an All-FORTRAN music-generating computer program. In Proceedings of the 17th Annual Meeting of the AES. (Preprint 397) Audio Engineering Society.

Scaletti, C. (2002). Computer music languages, Kyma, and the future. *Computer Music Journal*, *26*(4), 69–82.

Schottstaedt, W. (1996). *Common Lisp Music 4 Manual*. Stanford, CA: Centre for Computer Research in Music and Acoustics, Stanford University.

Smith, J. O., & (1991). Viewpoints on the history of digital synthesis. Proceedings, (1991). *International Computer Music Conference, Montreal (pp. 1–10)*. San Francisco, CA: Computer Music Association.

Sorensen, A. (2012). Extempore Sources. Retrieved from: https://github.com/digego/extempore

Stallman, R. (2010). Free Software, Free Society: Selected Essays of Richard M. Stallman. Boston, MA: GNU Press.

Tenney, J. (1963). Sound generation by means of a digital computer. *Journal of Music Theory*, *7*, 24–70.

Vercoe, B. (1973). *Reference Manual for the MUSIC 360 Language for Digital Sound Synthesis*. Cambridge, MA: Studio for Experimental Music, MIT.

Vercoe, B. (1981). *MUSIC 11 Reference Manual*. Cambridge, MA: Studio for Experimental Music, MIT.

Vercoe, B. (1983). Computer system and languages for audio research. The New World of Digital Audio (Audio Engineering Society Special Edition), 245–250.

Vercoe, B. (1996). Extended Csound. Proceedings of the International Computer Music Conference (1996). *Hong Kong (pp. 141–142)*. San Francisco, CA: Computer Music Association.

Vercoe, B. (2004). Audio-Pro with multiple DSPs and dynamic load distribution. British Telecom Technology Journal, 180–186.

Vercoe, B., & Ellis, D. (1990). Real-time Csound, software synthesis with sensing and control. In Proceedings of the International Computer Music Conference 1990, Glasgow. San Francisco, CA: Computer Music Association, 209-211.

Wang, G. (2008). The ChucK audio programming language, a strongly-timed and on-the-fly environ/mentality (PhD thesis). Princeton University, Princeton, NJ.

Webb, C., & Bilbao, S. (2012). Binaural simulations using audio rate FDTD schemes and CUDA. In Proceedings of the 15th International Conference on Digital Audio Effects DAFx-12, York, UK, 97–100.

Yi, S., Lazzarini, V., & (2012). Csound for Android. In Proceedings of Linux Audio Developers Conference, (2012). *Stanford*. CA: Centre for Computer Research in Music and Acoustics, Stanford University.

Zicarelli, D. (2002). How I learned to love a program that does nothing. *Computer Music Journal*, *26*(4), 31–43.