PERSISTENCE IN MUSIC DATA STRUCTURES

L. E. Nugroho
James Cook University

A. S. M. Sajeev
Monash University

Abstract

Persistence is the concept of preserving information in their original structure. This allows the design of suitable structures to manipulate data irrespective of their life-span. Much of the current research into persistence is in the area of providing programming and system support. This paper, on the other hand, gives an application of persistence in implementing a flexible data structure for computer music. In one sense, computer music data are like text-documents: they need to be edited, filed and used. On the other hand, unlike ordinary text, they are highly structured. This makes the area interesting to apply the principles of persistence programming. We have designed a persistent music structure and used it in the construction of a music editor and player. We analyze the advantages and disadvantages of using persistence in this context.

Keywords

Persistence, Computer Music, Data Structures

Introduction

Persistence is the concept preserving information in their original form for reuse. Much research has gone into the design of persistent architectures [3], operating systems [1], and programming languages [15]. In a persistent programming environment, the life-span of an object is orthogonal to its type. It has been argued that this would allow a greater flexibility in designing data structures. In a traditional programming environment, the data structure used during the execution of a program can be quite different from the structure (normally a flat sequence) used for long term storage as a file. In cases where data has to be moved back and forth between long and short term storage structures, the data structure must be constructed from scratch each time. The programmer has to write the code to do this. There is a tendency among programmers to design data structures such that the code needed for conversion is minimized. Obviously this results in a compromise because a more appropriate short-term data structure could have been designed, but for the requirement of long term storage. For instance, some C compilers use a linear structure for the symbol table mainly because the table can be

المنارة للاستشارات

www.manaraa.com

preserved as a file to be used by the debugger. Such trade-offs are unnecessary in a persistent programming environment since the life-span of data is independent of its structure. In this paper, we test out this claim in a computer music application. This paper reports how a persistent data structure is designed for computer music editing, and what its consequences are in terms of flexibility and performance.

In the next sections we describe some of the common computer music formats, followed by a brief discussion of our persistent programming environment. This is followed by a discussion of the design of a persistent music data structure and its implementation. Finally we give our conclusions.

Audio/Music File Formats

At present, there are many file formats for storing computer music data, each is designed specifically for a particular system or purpose. It is common to group the different formats into two categories : *self-describing* formats and *raw* formats [17]. A self-describing format is one that incorporates device parameters and encoding into a file header. This means that various settings can be individualized for any file. In a raw format, there is no encoding because the file only contains raw data. A self-describing file is more flexible but complex in structure, while a raw format file has the opposite characteristics. Flexibility is an important factor in choosing a format for music data because music itself is not a static entity.

This work concentrates only on self-describing format as it supports music representation better.

The structure of music

Music is similar to a story in the sense that they are both structured entities. In fact, music itself is a form of story which has its own forms of paragraphs, sentences, words, and even punctuation marks. Musical sounds, like words, build larger structural entities : clauses, sentences, and paragraphs [4]. Each structure part has its own characteristics. There are times when the music should be played softly, loudly, or even with changed tempo (the speed it is played).

The smallest unit in music is a *note*. A sequence of notes forms a *phrase*, the smallest rhythmic unit. Each phrase ends with a *cadence*, and two or more phrases build a *sentence*. Short songs usually have one *paragraph*, which is built from several sentences. Long songs, on the other hand, may have more than one logically-connected paragraphs. There is no strict rule on this structuring, but almost every music has a structure similar to the above. While stories are represented by structuring them from their logical parts, songs should be treated similarly.

www.manaraa.com

Searching for a model format

We start with a search for a model by looking at several popular formats. The MIDI file format [12,6], for example, supports the concept of multiple patterns in a song, but the overall structure is not clearly represented because a MIDI file is basically a stream of MIDI messages from and to various MIDI devices [7]. MIDI emphasizes on communication among its elements, so music structure representation is not crucial.

Other file formats, like the Sound Blaster's VOC and CMF [6], do not support music structure representation elegantly. VOC format is suitable only for voice (non-musical) data because it only holds sample data and does not support musical notes and patterns. CMF format, on the other hand, is designed for music. It consists of a header block, an instrument block, and a music block, but the music block adheres to the Standard MIDI format, so it is similar to the above MIDI format.

Among those formats, only the Amiga's MOD format [17,9] provides a rigid basis for structured music representation. A song (or a MODule in Amiga's term) is always built on three entities : *patterns*, *notes*, and *samples. A* module is a sequence of (not necessarily unique, but a maximum of 128) patterns. A pattern itself is a fixed size array of 64 (possibly blank) notes. Figure 1 and 2 show the structure and layout of a module, respectively.

Each note contains three information : the pitch of the note, an effect command specifying how the note is to be played, and a sample number telling which sample is used to play the note. Figure 3 shows the layout of a note. The note's length is implicitly declared by positioning a non-blank note at the proper place in a pattern. The more the blank notes between a note and its following non-blank note, the longer the previous non-blank note. Some effect commands are supplied to make the music more realistic to some extent; some others are operations on patterns and samples.

In the module term, a sample represents an instrument. A sample contains raw, digitized sound data, which is obtained from a sampling and digitizing process. In this process, analog signals of sound is converted into digital form. Once the signals are digitized, they can be stored in files. To play a sample, the digital data is converted back into its analog form and output to a amplifier. Recent MOD format has a maximum of 31 samples.

المنارة للاستشارات
www.manaraa.com

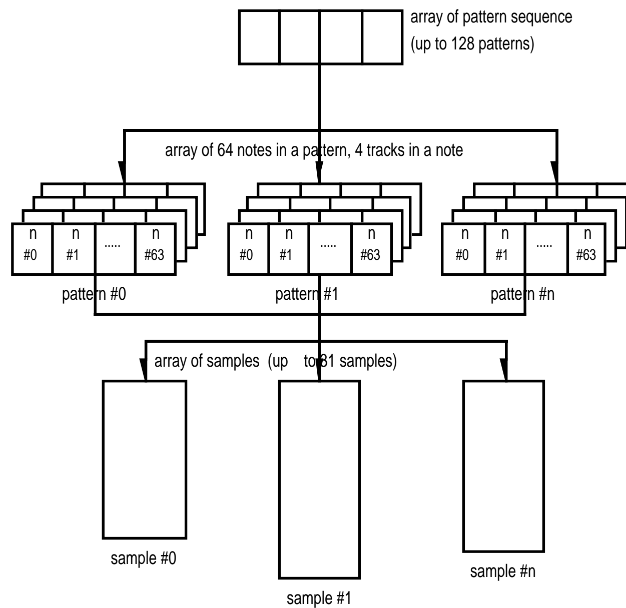

Figure 1 : The structure of a module

It is also common for the MOD format to have more than one *track*. Most modules have 4 tracks, but 6-track and 8-track modules are also available recently. Multiple tracks mean that multiple notes can be played at (almost) the same time.

The MOD format provides a good structure to represent music. Besides, it can be loaded with various samples, thus making the music brighter. With the sample's ability to represent almost any kind of sound, there is no limit on sound variation that can be used in a module. A module size is also usually compact. However, the MOD format has some limitations. It cannot represent all types of music. It cannot handle some musical dynamics, mostly because it is note- and sample-oriented. This limitation makes the MOD format not suitable for classical music, for example. Moreover, the MOD format cannot elegantly handle the same instrument played in different ways. For example, picked and slapped bass guitar samples must be distinguished. It cannot use a common instrument sample and then process it accordingly. However, it is not our research aim to remedy this situation. What we are interested in is the structural deficiencies of music formats.

المنارة للاستشارات
www.manaraa.com

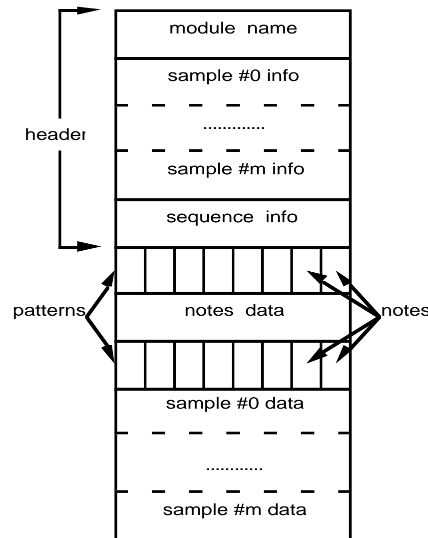

Figure 2 : The layout of a module

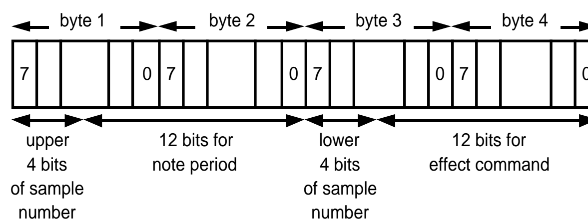

Figure 3 : The layout of a note

The MOD format has some weaknesses in terms of its structure as well. Firstly, a fixed pattern size sometimes makes unnatural splits on musical sentences or phrases. No matter how long or short a phrase is, it is fixed at 64 notes (long phrases need more than one pattern). Secondly, the way of representing note length by positioning notes between blank notes is considered inefficient. Thirdly, some notes are redundant. This, however, is not considered as a major drawback as redundancy overhead is typically much smaller than overall system performance (accessibility and storage space). We address our work to improve the structure in order to overcome these limitations.

The above situation is caused by the need of having two types of existence (short term and long term) of data. This forces programmers to include mapping code in their programs to be able to access both short term and long term media. This is an extra work, and obviously has at least two disadvantages. First, time and effort are spend inefficiently for a distracted task. Second, and this is the case of the MOD format, a good design cannot be implemented perfectly. The designer needs to build a good structure for music representation. However, it is not easy to deal with that structure in long-term storage media. As a result, a compromise is made, which

المنارة للاستشارات
www.manaraa.com

has reduced the quality of the design. A persistent programming environment, on the other hand, will allow us to concentrate on the design without worrying about the life-span of data and its storage media.

Persistent Data Structures

Persistent C and its persistence functions

Persistence is the time period for which data exists and is usable. Atkinson, et. al. [2] defines six persistence categories, but we will only be interested in the broadest persistence range, that is when the data outlives the program that created it.

Much research has gone into the design of persistent systems. Several persistent programming languages have been created, for example Napier88 [10], E [14], and χ [15]. However not much has been done in using such systems in developing applications.

Our research has been performed in the context of a simple extension to the C language [16]. This extension is called **Persistent C (pC)**. Persistent C makes minimal changes to the original language, thus enabling a large community of C programmers to program persistence with minimum efforts.

Persistent C provides two functions to implement persistence : pprintf and pscanf. The syntax is :

```
pprintf(<file_pointer>,<type_string>,<object_address>);
pscanf(<file_pointer>,<type_string>,<object_address>);
```

The pprintf function makes the transitive closure of the object whose address is given as a parameter persist in the form of a file. The pscanf function retrieves back a persistent object to the memory. There is no restriction on the number of pprintf and pscanf calls in a program. There is also no restriction on the data type in a call.

Note that we are using the underlying file system for storing persistent objects, thus reducing the number of constructs to be built into the language.

Type checking

Conventional data files carry no type declaration, so we have to build our own type checking mechanism for pC. Persistent C cannot make use of C's type checking mechanism because it is only useful when data are still transient.

المنارة للاستشارات
www.manaraa.com

To overcome this problem, pC uses a string of characters to describe the type of the data structure being made persistent. This string, called *typestring*, is passed along with the data structure and is stored as a header information. Subsequent access to the data must satisfy a match between the persistent type and the transient type. The typestring approach also plays a very important role in the implementation. This is explained in detail in [11] (in future implementations we plan to automatically deduce the typestring during parsing). Table 1 shows the character codes that form a typestring for the most common cases.

Persistent Music

The structure

The persistent music structure that we design and build is based on the MOD structure, and is called **Persistent Music** or **pMusic** for short. The MOD format is a good design, and our purpose is to add flexibility to it. We replace the fixed pattern size (array of 64 notes) with a dynamic linked-list and assign an identification key to every patterns and notes. This helps to remove unnecessary pattern and note duplications which are present in the original structure.

Character code	Type
c	character
i	integer
f	float
l	long
d	double precision float
a	array
s	structure
u	pointer
0..9	array dimension digit
r	self-referential pointer
-	end of array/structure/union

Table 1 : Character code for typestrings

www.manaraa.com

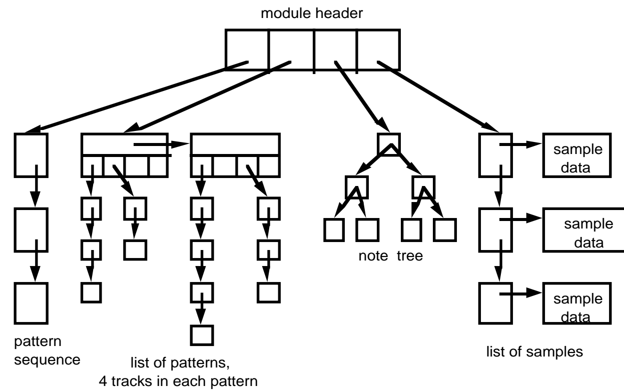

Figure 4 : The Persistent Music structure

In pMusic, access to subordinate levels uses the subordinate's ID key. For example, each node in the song sequence contains a key that tells what pattern to play. The key is the index position of the pattern in the pattern array. Similarly, in each pattern node in the pattern linked-list there is a note key specifying what note to sound. This key is then used to find the note in the note tree. Such an indirect access using keys is unavoidable if pattern and note uniqueness is to be maintained.

The overall structure of a pMusic module is shown as follows :

```
typedef struct sample {
    char sample_num;            /* sample number */
    char sample_name[20];       /* sample name */
    char sample_len;            /* sample length */
    char volume;                /* sample volume */
    int  rep_start;             /* repeat posn */
    int  rep_len;               /* repeat length */
    char *sample_data; /* sample data */
} Sample;

typedef struct note {
    unsigned long key; /* note key */
    unsigned period;            /* note pitch */
    char length;                /* note length */
    char effect;                /* sound effect */
    char param;         /* effect param */
    char sample_num;            /* which sample */
    struct note *left; /* left node */
    struct note *right;         /* right node */
} Note;

typedef struct pat_node {
    unsigned long note_key;     /* which note */
    struct pat_node *next;      /* next node */
} Pattern_Node;

typedef struct a_pattern {
```

المنارة للاستشارات
www.manaraa.com

```
    char key;                  /* pattern key */
    Pattern_Node *head;        /* list head */
} Pattern[4];

typedef struct song {
    char pat_num;              /* which pattern */
    struct song *next; /* next node */
}  Song;

typedef struct pmodule {
    Song *sequence;            /* pattern seq. */
    Pattern patterns[64];      /* arr of patt. */
    Note *root;        /* note tree */
    Sample samples[31];        /* arr of sample */
} PModule;
```

The music editor

We have built a music editor with a graphical user interface to allow the user to compose or edit songs. In entering note values, it uses the common music notation. A translator converts the graphical note symbols into appropriate note values. The translator accepts the note symbol and position on the staff and searches a table for the corresponding note value and length. Screen position information of note and other symbols are also maintained to enable deletion, insertion, and loading operation.

When an editing session ends, the pMusic data structure, along with the screen information, is saved into the disk by calling the pprintf function. The typestring parameter is :

"s*sc*r-a64a4sc*sl*r----*slicccc*r*r-a31sca20c-ccii*c---"

To load the song, the same typestring is passed as the parameter of pscanf, and based on the screen information file, the graphical image of the song is restored on the screen.

The music player

Persistent Music player is a modification of ordinary MOD players [8,5]. It is an interrupt-based program combined with a polling technique that checks a double-buffer for data to be read by the sound card (we use Sound Blaster Pro card) using Direct Memory Access (DMA) mode [18]. From the macro vision, the pMusic player is basically a note tracker driven by interrupt. It tries to scan and play notes in all tracks of the patterns in the order specified by the pattern sequence.

The main part of the player is the tracker. It tracks down the pattern sequence to find out which note and what sample to play. Because each track in the pattern is independent of each other, it is necessary to treat its notes independently, too. This is done by setting a counter to each

المنارة للاستشارات
www.manaraa.com

track, whose value depends on the note length. The longer the note, the greater the counter value.

Note value	Counter value
whole note (1/1)	15
half note (1/2)	7
quarter note (1/4)	3
eighth note (1/8)	1
sixteenth note (1/16)	0

Every time the tracker is called, it decrements all counters for each track. If the counter reaches zero, the subsequent note is tracked down, its sample number is associated with the sample array, and (a portion of) the sample data is placed into the buffer. Otherwise, the tracker keeps playing the current note's sample.

The tracker also maintains a pointer, called Row, in each track to detect the end of the pattern. Row is advanced every time a note is read, following the track linked-list. When the last note is read, it will point to NULL, and it will signal the end of the track. An end of pattern occurs when all Row pointers point to NULL and all counters have zero value. When an end of pattern is encountered, the tracker will determine the next pattern to be played from the pattern sequence list.

Figure 5 shows an example of how the tracker works. Treating each track independently of each other, it decrements every note counter in each track every time it is called. If c=0 it will load a new note, otherwise it keeps playing the current note's sample.

المنارة للاستشارات
www.manaraa.com

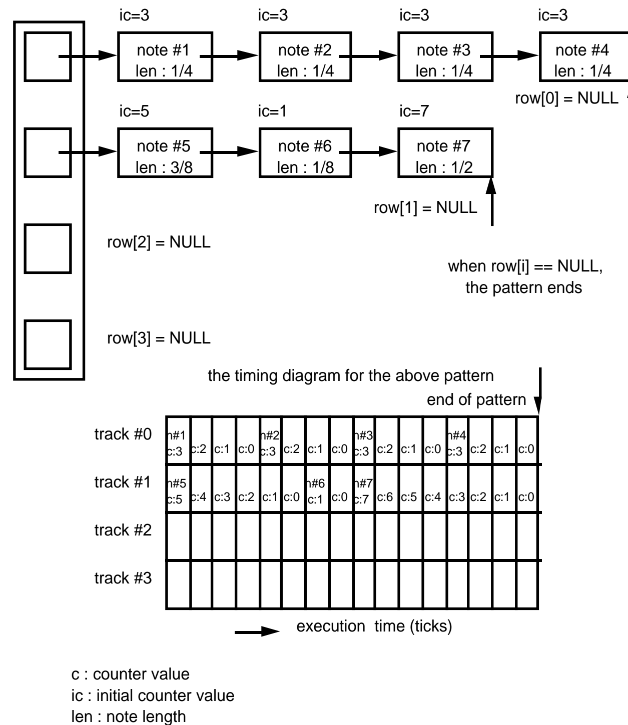

Figure 5 : Timing technique of the pMusic player

Discussion

We have designed a minimal extension to the C language to program persistence. As an application of persistence, we used the language in developing a music editor and player with a persistent data structure. Our approach has several advantages.

- Improved structure.
  In pMusic the structural part of a music is clearly defined; each part is separated from the other. This makes the music more readable. However, we realize that further work is needed to implement full reusability; that is, users should be able to take parts of a music piece and use it in constructing new pieces. We are working to remodel the persistence routines to add this capability.

- Improved flexibility.
  We have removed the restriction which the MOD format has on the pattern size. This gives greater flexibility to the users. A user can now concentrate on the composing task without worrying about the pattern size. The pattern format makes composing even more easier, because a user needs to enter only notes' pitch and length in their normal way, without having to position the notes in the correct place as in a module editor.

المنارة للاستشارات
www.manaraa.com

- Simpler way of storing and loading of song modules.
  For a programmer who wants to build a different pMusic editor and player, this is a great advantage, because the loading ad storing of a module can be performed by calling a single function. Again, the programmer can now concentrate on programming the editor or the player, rather than doing distracting tasks like coding disk i/o or converting data from one structure to another.

This flexibility and structuring, however, comes at a cost. Patterns and notes, while their uniqueness can save space, take longer to edit because of the necessity to check their uniqueness. However, this happens during editing sessions where the human-interaction factor tends to hide the additional internal processing overhead. The structuring of pMusic means that during play back, the player has to traverse a longer path to get a note. Moreover, the dynamic length of each track in a pattern makes the tracker spend extra processing time to check for their ends. The effect of this delay on the tests we conducted was not uniform. Some samples produced slight audible 'clicks', while some did not. We have tried to minimize the delay by using assembly language programming in all time-critical tasks. More work is needed in optimization terms to avoid the delay.

Conclusion

To summarize, the use of persistent data structures for computer music has resulted in a very flexible and structured mechanism with space savings and potential for reusability. However, our experiment has also shown that optimization is needed in some cases to avoid performance deterioration.

Acknowledgments

This research was partly supported by an Australian Research Council grant for the Persistent Reusable Object Environment (PROBE) project.

References

[1] M. Anderson, R. D. Pose and C. S. Wallace, A Password-Capability System, *Computer Journal*, 29(1), 1986, pp. 1-8.

[2] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, P. W. Cockshott, and R. Morrison, An Approach to Persistent Programming, *Computer Journal*, 26(4), 1983, pp. 360-365.

[3] P. W. Cockshott, M. P. Atkinson, K. J. Chisholm, P. J. Bailey, and R. Morrison, Persistent Object Management System, *Software-Practice and Experience,* 14(1), 1984, pp. 49-71.

المنارة للاستشارات

www.manaraa.com

[4] C. T. Davie, *Musical Structure and Design*, Dover Publications Inc, NY, 1966.

[5] C. Hasan, Tiny MOD Player, Available via anonymous ftp from nic.funet.fi in /pub/msdos/sound/sb/tinyplay.zip, December, 1993.

[6] R. Heimlich, D. M. Golden, I. Luk, and P. M. Ridge, *Sound Blaster: The Official Book*, Osborne McGraw-Hill, 1993.

[7] G. Jacobs, and P. Gheorgiades, *Music and New Technology: The MIDI Connection*, Sigma Press, Winslow, England, 1991.

[8] J. Jensen, Protracker for IBM PC, Available via anonymous ftp from wasp.eng.ufl.edu in /pub/msdos/demos/programming/source/pps110.lzh, July, 1993.

[9] N. Lin, MODEDIT v3.01 Documentation, Available via anonymous ftp from simtel-20 or its mirrors.

[10] R. Morrison, F. Brown, R. Connor, and A. Dearle, The Napier88 Reference Manual, *Technical Report PPRR-77-89*, University of St Andrews, 1989.

[11] L. Nugroho, An Implementation of Persistent Data Structures in C, *Working Paper*, James Cook University, May, 1993.

[12] J. Pressing, *Synthesizer Performance and Real-Time Techniques*, Oxford University Press, 1992.

[13] J. W. Ratcliff, Examining PC Audio, *Dr Dobb's Journal*, 18(198), March, 1993.

[14] J. E. Richardson, M. J. Carey, Persistence in the E Language: Issues and Implementation, *Software-Practice and Experience*, 19(12), 1989.

[15] A. S. M. Sajeev, A. J. Hurst, Programming Persistence in χ, *IEEE Computer*, 25(9), 1992, pp. 57-66.

[16] A. S. M. Sajeev, Some Reusability Exercises in Persistent C, *Journal of Computing and Information*, pp. 1160-1175, 1994.

[17] G. van Rossum, FAQ: Audio File Formats, Available via anonymous ftp from ftp.cwi.nl, March, 1993.

[18] R. Watson, DMA controller Programming in C, *The C Users Journal*, 11(11), November, 1993.

المنارة للاستشارات

www.manaraa.com

المنارة للاستشارات
www.manaraa.com