Common Practice Notation View: a Score Representation for the Construction of Algorithms.

Author: Donncha O Maidin Centre for Computational Musicology and Computer Music, Department of Computer Science and Information Systems, University of Limerick, Limerick, Ireland. (donncha.omaidin@ul.ie).

Abstract

This paper describes the design of a software representation for music scores. It is intended for use in building algorithms for processing scores. The focus is on representing scores that originate in, or that are generated in common practice notation. Common Practice Notation facilitates the composition, storage, performance and transmission of music. In western music it plays a central role in music literacy, in activities involving the visualisation of music and in music education. Computer-based representations are to be found in composition-oriented systems, in notation-oriented systems, and in some of the standard or semi-standard file-based representations. Composition systems in general are not based on Common Practice Notation. They either ignore it altogether, or embed a simplified subset of it. Most notation systems, on the other hand, have an internal representation that is not externally accessible. Another source of computer scores is found in the various file-based representations that come from corpora-oriented or from interchange-oriented systems or from notation systems. In all cases file-based representations have a level of complexity that gets in the way of their direct use for algorithm construction. The current approach is to provide a software environment that hides such complexity by using an object-oriented approach. The score object is represented in accordance with principles of objectivity and completeness. File-based representations can be converted into C.P.N.View from a number of different file-based representations. Also C.P.N.View score representations can be built algorithmically or interactively. The C.P.N.View environment is designed for the development of algorithms that access, process and create music scores. Application areas include music analysis, performance, multimedia, music education, music corpora, information retrieval and browsing, and computer-assisted composition. The representational model consists of a Score class and a ScoreIterator class. The Score class is used to represent score objects. In some applications the score objects is created in an automatic fashion from a file representation. The ScoreIterator class is used to construct iterator objects, through which the main algorithmic work is done. Iterator objects can be used to extract information, to modify information, or to insert new score components. The ScoreIterator object is responsible for simplifying much of the complexity inherent in score processing. Context resolution is automatic. Existing scores may be traversed in various ways, they may be modified, or be constructed from scratch via the ScoreIterator. The result is an environment in which score algorithms may be constructed that avoids unnecessary complexity and yet preserves generality.

1. Introduction. C.P.N.View provides a software environment that hides such complexity by using an object-oriented approach. The level of abstraction achieved allows the user of the system to construct scores and to retrieve information from them with minimal specification. Nevertheless the score object is represented in accordance with principles of completeness and objectivity. The principle of completeness implies that the basic information content of a score is captured. The working notion of "basic information content" is such that two typesetter would produce versions of a score that would be regarded as musically equivalent by musicians. Inevitably there will be grey areas in what is regarded as "basic". C.P.N.View models the score as an indefinitely long staff, and encodes the type of symbols present and their relative position. Aspects such as font type, exact shape of slurs and exact slope of beams are not represented. Objectivity on the other hand implies that the score representation does not commit the user to a specific interpretation of symbols of the written score in cases where possible ambiguity exists. The distinction between a slur and a phrase mark is an example. There is however a provision for encoding ancillary information in the representation by putting multiple annotations on any score object.

2. Classes. The language of implementation of C.P.N.View is C++. A number of support classes are used in the representation. These include classes for representing strings (String), for rational numbers (Rat), for sets (Set) and for pitch class sets (PitchClasses). Also many enumerated types are used for representing
values of entities such clef names (e.g. TREBLE), note and rest durations (e.g. N4,N8), types of score object (e.g. NOTE, REST, BARLINE). Various objects within the score are represented as classes. These include
KeySig, TimeSig, Clef, Metronome, Tempo, Expression, Duration, Pitch, Rest, Note, Barline, Words and Text. Two classes lie at the heart of the representations. These are classes Score and ScoreIterator.
3. The Score Class. This class represents an entire score. It can be used to construct a score object from an existing file representation. Also it can be used to build a score. The following line causes the Score object called score1 to be built from the contents of the file whose name is given in the parameter.

    Score score1(filename);

Depending on the extension of the filename, the appropriate translator is invoked. Translators are currently available for ALMA, *kern and NIFF formats.
If we want to build a score algorithmically, an empty score called score2 may be first created from the following line

    Score score2;

4. The ScoreIterator Class. The processing of scores is facilitated through the availability of a class called ScoreIterator. A score iterator is a kind of cursor, associated with a score. It is associated at any one time with a single constituent object within the score. A score iterator called si1, is created for the score score1 as follows

    ScoreIterator si1(score1);

The score iterator can be moved around the score in two ways. The first way is by use of the member function locate() which places the iterator at an absolute position in the score. For example, the score iterator si1 may be located at the start of bar 20 in the score as follows

    si1.locate(BAR, 20);

The second way of moving the score iterator is by moving it relative to its current position using the step member function.

    si1.step(NOTE);

will move the iterator to the next note in the score.

The step member function will also take a durational parameter (rational number) to move the iterator to the first objects that lies at the appropriate distance from the current position. For example to move the iterator onwards by a one-eight-note distance.

    si1.step(Rat(1,8));

The following sequence will create the score iterator si1 and locate the iterator at the first note in the score.

    ScoreIterator si1(score1);
    si1.step(NOTE);

The alternate sequence will create the score iterator si1 and locate it at the first note following the start of bar number 20.

    ScoreIterator si1(score1);
    si1.locate(BAR, 20);
    si1.step(NOTE);

These functions return a TRUE/FALSE type result, to indicate the success or failure of the operations. Such would be essential to check in the above case, if, for example the score were less than 20 bars in length, or if there were no notes following the start of the 20th bar.

The above program fragments will work with a score that is either monophonic or polyphonic, and irrespective of how many staves it contains. The "note following the start of bar 20" is subject to various interpretations in all but a single stave monophonic score. The ambiguity here can be resolved by defining the kind of polyphonic traversals available. This will be explained later, and for the present, examples will assume the score in question is single stave and monophonic.

One of the main uses of an iterator is to extract information. All of the basic information content of the score is available via the iterator. If the current object is a note, then the member function getPitch12() returns, in effect, the MIDI note number of the note.

5. Example 1: The following is a complete program that processes the score in a file called "filename" and reports on the highest and lowest note found.
#include "score.h"
using namespace CPN;
void main()
{
    Score s("filename");  // construct the score
    highest = 0;  lowest = 127;  // initialise the limits
    ScoreIterator si(s);  // create iterator
    while( si.step(NOTE))  // step through the notes of the score
    {
        int pitch = si.getPitch12();  // access pitch of the note
        if( pitch > highest) highest = pitch;  // set highest
        if( pitch < lowest ) lowest = pitch;  // set lowest
    }
    cout << "\nHighest pitch = " << highest << ", lowest pitch = " << lowest << "\n";
}

6. Example 2: The following is a recursive function that identifies if a melodic palindrone of length n is present starting at the note at the current location.

int isPalindrome(ScoreIterator si, unsigned int n)  // function checks if note sequence starting at
{  // position si is a melodic palindrome of length n
    if ( n < 2 ) return TRUE;  // return TRUE for less than two note sequence
    ScoreIterator sie = si;  // create new iterator
    for ( int i = 0; i < n-1; i++)  // move iterator n-1 notes forward
    {
        if ( !sie.step(NOTE)) return FALSE;  // return FALSE if no more notes
    }
    return isPalindrome( si, n-2);  // length n-2
}

A practical algorithm will be a little more complex than the above one, as we have not allowed for a number of possibilities, for example, how we should deal with intervening rests.

7. Polyphony. C.P.N.View supports multistave polyphony. If the file held in filename is a multi stave polyphonic score, a single stave score iterator can be created by giving an extra integer parameter to the constructor. The following constructs an iterator score0 that traverses the uppermost stave.

ScoreIterator score0(filename, 0);
A score iterator that traverses all objects on all staves is constructed as follows

ScoreIterator score(filename);
The score iterator will visit all of the objects in a score if its step() member function is called repeatedly. The score iterator starts on the leftmost objects on the first stave, and traverses through all objects on that stave until it reaches the first note or rest. It then repeats this action on the second and subsequent staves. After the lowermost stave is processed in a similar fashion, the iterator is advanced by an amount of time corresponding to the length of the shortest note/rest encountered on its journey through the staves. The subsequent call to step() will position the iterator at the highest object on the uppermost stave that is current at the specified time. When a barline is encountered on any stave except for the lowermost one, the call to step() will cause the iterator to move to the next stave.
8. Example 3: This is a complete program that illustrates the use of a polyphonic iterator to traverse all of the entities in a polyphonic score and to print out the pitch classes of all vertical combinations of notes encountered for each time slice.

```c
#include "score.h"
void main()
{
    String filename;
    // Create Bach score from *kern
    Score score(String("c:\\Mdb\\Ccarh\\Chorales\\BWV0253.KRN"));
    PitchClasses pcs;
    ScoreIterator si(score);
    while (si.step())
    {
        if (si.isA(NOTE)) pcs.pitchClassInc(si);  // add pitch to pitch class set
        // lowest entity reached
        if (si.isSystemLowest() && si.isStaveLowest())
        {
            if (pcs != PitchClasses())  // if note encountered
            {
                pcs.NIAPrimeForm();  // calculate prime form
                cout << "n" << pcs << ",";  // output prime form
                pcs = PitchClasses();  // reset pitch class set
            }
        }
    }
}
```

9. Example 4: This illustrates how a score may be built in CPNView. The '+' operator is used to add entities in sequence. The '^' operator is used to add entities vertically.

```c
#include "score.h"
void main()
{
    Score score;  // create empty score
    score.setMaxStaves(1);  // set maximum number of staves
    ScoreIterator si(score);  // create a score iterator
    si.startEnt();  // set up initial structures
    si += Clef(TREBLE);  // add clef
    si += TimeSig(4,4);  // add time signature
    si += KeySig(SFSC);  // add key signature
    si += Note('C', 5, NOACCID, N4);  // add middle C, quarter note
    si += Note('D', 5, NOACCID, N4);  // add middle D, quarter note
    si += Note('E', 5, NOACCID, N4);  // add middle E, quarter note
    si += Note('F', 5, NOACCID, N4);  // add middle F, quarter note
    si ^= Note('G', 5, NOACCID, N4);  // add middle G, quarter note above C
    si += Barline(H);  // add heavy barline
    si.locate();  // move back to start
    while (si.step()) cout << si << "n";  // print out textual representation of score
}
```

10. Conclusion
C.P.N.View is a developing environment for score representation. It currently supports processing of corpora in ALMA and in *kern. A component for NIFF input is under development and it is intended that components for input from other file types will be developed in future. It currently has a limit capacity for single stave polyphony.