Development of a Composer's Sketchbook

Georg BÖNN
School of Electronics
University of Glamorgan
Pontypridd CF37 1DL
Wales, UK
gboenn@glam.ac.uk

Abstract

The goal of this paper is to present the development of an open source and cross-platform application written in C++, which serves as a sketchbook for composers. It describes how to make use of music analysis and object-oriented programming in order to model personal composition techniques. The first aim was to model main parts of my composition techniques for future projects in computer and instrumental music. Then I wanted to investigate them and to develop them towards their full potential.

Keywords

Music Analysis, Algorithmic Composition, Fractals, Notation, Object-Oriented Design

0   Introduction

Computer-Assisted Composition (CAC) plays an important role in computer music production. Many composers nowadays use CAC applications that are able to support and enhance their work. Applications for CAC help composers to manage the manifold of musical ideas, symbolic representations and musical structures that build the basis of their creative work. Maybe it is time to put a flashlight on CAC again and to look at examples where user-friendly interfaces meet efficient computation and interesting musical concepts.

What are the advantages of CAC? For a composer to be able to use the PC as an intelligent assistant that represents a new kind of sketchbook and a well of ideas.

Not only, that it is possible to quickly input and save musical data, the work in CAC results in a great freedom of choice between possible solutions of a given compositional problem. Maybe a network of different algorithms work together, then only little changes of initial parameters could trigger surprising twists within the final result. Moreover, you can take those outcomes and scrutinise their value by direct and immediate comparison. Thus, a composer can take the time and always look for alternatives. The manifold of structures and ideas is going to be manageable through CAC software. Also, one should take into account the thrill of surprise that well designed CAC algorithms might fuel into one's work-flow.

This paper wants to discuss one specific compositional problem that is the invention and modelling of melodic structures. The proposed software that resolves that particular probelm shall represent a germ for an open source (under the GNU Public License) and free CAC application that is planed to grow as the number of compositional algorithms will hopefully increase in the near future. It is intended that this software is easy to learn and to use and that it benefits from proven concepts of object-oriented design and programming. Therefore, the author hopes that the ideas presented will find the interest and also maybe the support of the Linux Audio Developer's community.

Of course, personal preferences and musical experience influence the work of a composer. Those together with intuition, imagination, phantasy and the joy to play, including the joy for intellectual challenges, they are, in my view, the driving forces behind musical creativity. Would it be possible to define a set of algorithms that would match that experience? Is finding an algorithm not also often a creative process?

1   Music Analysis

The fundamental idea in Composer's Sketchbook is the use of a user-defined

database, that contains the building blocks for organic free-tonal musical stuctures. Those building blocks are three-note cells which stem from my personal examination of the major and minor third within their tonal contexts (harmonics, tuning systems, Schenker's Ursatz), as well as within atonal contexts (Schoenberg's 6 kleine Klavierstücke op.19, Ives' 4th violin sonata). Although in my system, tonality is hardly recognisable anymore because of constant modulations, i cannot deny the historic dimension of the third, nor can i neglect or avoid its tonal connotations. I suppose it helped me to create an equilibrium of free tonality where short zones of tonality balance out a constant stream of modulations.

Analysis of scores by Arnold Schoenberg and Charles Ives led me to a very special matrix of three-note cells. Further analysis revealed that it is possible to use a simple logic to concatenate those cells. Algorithms using this logic were created who are able to render an unprecedent variety of results based on a small table of basic musical material. In order to generate the table, a small set of generative rules is applied to a set of primordial note-cells. The general rule followed in this procedure is to start with very simple material, then apply combinatorics unfolding its inner complexity and finally generate a manifold of musical variations by using a combination of fractal and stochastic algorithms.

The first four cells forming the matrix are variations of the simple melodic figure e-d-c. The chromatic variations are e-flat-d-c, e-flat-d-flat-c and e-d-flat-c (see Figure 2, A-D).

One of the reasons why I chose these figures was, because they represent primordial, archetypical melodic material that can be found everywhere in music of all periods and ages. An exceptional example for the use of the Ursatz-melody is Charles Ives' slow movement of the 4th violin sonata. This movement quotes the chorale "Yes, Jesus loves me" and uses its three-note ending e-d-c ("he is strong") as a motive in manifold variations throughout the piece. Analysis reveals that Ives uses the major and minor versions of the motive and all its possible permutations (3-2-1, 2-1-3, 1-3-2).

2 The Matrix

The matrix I developed uses exactly the same techniques: Four different modes (A-D) of the 'Ur'-melody are submitted to all three possible permutations (see Figure 2). This process yields 12 primordial cells. Each one of those 12 cells is then submitted to a transformation called "partial inversion".

Figure 2: The matrix of 36 cells

Partial Inversion means: Invert the first interval but keep the second one untouched. Or, keep the first interval of the cell original

and invert the second one. This process of partial inversion can be found extensively used by Arnold Schönberg in his period of free atonality. As a perfect example, have a look at his Klavierstück op.19/1, bar 6 with up-beat in bar 5.

The reason for using partial inversion is that it produces a greater variety of musical entities than the plain methods of inversion and retrograde. At the same time partial inversion guarantees consistency and coherence within the manifold of musical entities. Applying partial inversion to the 12 cells yields another 24 variants of the original e-d-c-melody.

The final matrix of 36 cells contains an extraordinary complex network of relationships. Almost every cell in the matrix has a partner, which is the very cell in a different form, the cell may be inverted, retrograde, permutated or inverse retrograde. Yet, each one of these is closely related to the original 'Ur'-melody.

Going back to Schönberg's op. 19/1, it came as a surprise that every single note in this piece is part of one or more groups of three notes, which can be identified as a member of the matrix of 36 cells.

The discovery of an element of my own language in the matrix gave me yet another good reason to further investigate its application. I call it "chromatic undermining", breaking into the blank space left behind by a local melodic movement. Cells which belong to that element are the partial inversions of Ursatz B and C, left column.

The matrix of cells forms the basis of all further work with algorithms. It can be regarded as a database of selected primordial music cells. Thus it is clear that a user-interface should make it possible to replace that database by any other set of entities where it is totally in the hands of the user to decide which types of entities should belong to the database. The user-interface should also allow to add, delete or edit the database at runtime and make the database persistent.

3 Algorithms

The algorithms that are used to concatenate cells from the database are as follows:

3.1 Fractal Chaining

Beginning with two notes, the fractal algorithm seeks to insert a new note between them. It detects the original interval between the two notes, then it queries the database whether there exists a cell, which contains that interval between its first and last note. If it is true, then the middle note of the cell is inserted between the two notes of our beginning (see Figure 3 a). We now have a sequence of three notes whose interval strucure is equal to the cell that was chosen from the database. The pitch-classes of our first two notes are preserved. This algorithm, can be applied recusively. Starting now with three notes from our result, the algorithm steps through the intervals and replaces each one of them by two other intervals that were obtained from another cell within our matrix database (see Figure 3 b). Of course, there are multiple solutions for the insertion of a note according to a given interval, because there are several cells within the matrix that would pass the check. The choice made by the program depends on a first-come-first-served basis. In order to make sure that each cell has an equal chance of getting selected, the order of the matrix has always been scrambled the moment before the query was sent to the database. The fractal algorithm needs a minimum input of two notes but it can work on lists of notes of any length. Fractal chains maintain the tendency of the original structure at the beginning. Therefore, this interval structure is the background structure of the final result after the algorithm has been called a few times.
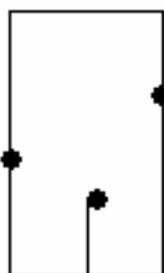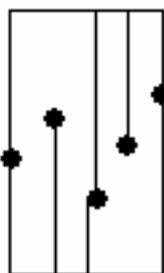
Figure 3: Sequence of Fractal Chaining

3.2 Chain overlapping 2 notes

The chaining algorithms differ from the fractal algorithm because their goal is to add notes to the tail of a given list rather then inserting them between every two notes. The chaining method overlapping two notes looks at the last interval of a melody. It then searches the matrix for a matching three-note cell whose first interval is equal to that last interval of the melody. If a match was found, then the second interval of the cell is added to the melody and so a new note is added, the melody expands (see Figure 4).
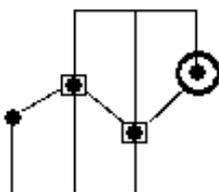
□ = overlap   ◯ = history check

Figure 4: Scheme of Algorithm 3.2

The reason for using overlapping cells was a result of music analysis: One note may belong to more than just one cell, thus providing a high degree of coherence of the inner musical structure.

3.3 Chain overlapping 1 note

This method simply takes a random cell from the database and adds it to the end of the melody by taking its last note as the first note of the cell, thus adding two new notes to the melody (see Figure 5).
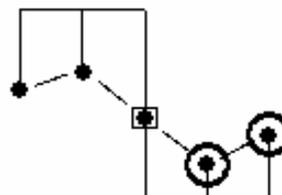
□ = overlap   ◯ = history check

Figure 5: Scheme of Algorithm 3.3

3.4. Chain combining Algorithms 3.2 & 3.3

The algorithm takes the last interval of the melody, finds, if possible, a match with a cell from the database, adds the last note from the cell to the melody, then taking this note as the starting point of a randomly chosen cell from the database (see Figure 6).
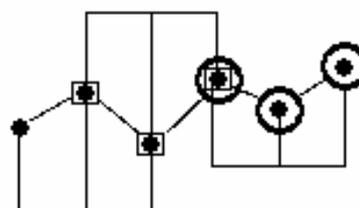
□ = overlap   ◯ = history check

Figure 6: Scheme of Algorithm 3.4

3.5 Option: check history

This option can be switched on or off and triggers a statistical measurment of the pitch-class content of the melody. It ensures that only those cells are chosen from the database whose interval structure generates new pitch-classes when added to the melody. This option leads to the generation of totally chromatic fields and ensures that every pitch-class has an equal chance to occur within the melody. The history-check can be modified in order to meet other requirements, e.g. the algorithm can be told to chose only cells adding pitch-classes to

the melody, which belong to a specific key or mode previously defined by the user.

3.6 Chain with no overlap

The algorithm choses the first interval of a randomly chosen cell and adds the interval to the melody. Then it takes another random cell and adds its content to the melody.

x = random interval

Figure 7: Scheme for Algorithm 3.5

3.7 Chain using different cell contours

The term contour means that each cell has a certain profile of up- or down-movements. These contours are automatically measured within the database. The chaining using contour comparisons takes the last three-note group of the melody and mesures the contour. Then it looks-up the database in order to find a cell which has one of the four possible contours: up/down, down/up, up/up or down/down. The user decides which one of those criteria has to be met and the matching cell is added to the melody.

3.8 Using and extending the program

By using different chaining methods, possibly in a sequence of processes, the user has an enormous freedom of choice and variety in modelling a melodic line and the structure of a sequence.

It will be possible to use within the program an editor for context-free grammar in order to let the user define a set of rules by which musical entites are chosen from the database.

It is easy to imagine, that the matrix of 36 cells can be extended and more variations of the e-d-c-melody could join the matrix. For instance, i added 10 more modes of the 'Ur'-melody resulting in a database of 126 cells.

It is also possible to use the chaining algorithms in order to build-up chord structures. When using a database containig major and minor thirds (or other intervals) it is easy to imagine that the algorithm "Chain overlapping 1 note" with history-check "on" returns all possible chords containing 3 up to 12 notes using nothing but major and minor thirds (or other intervals).

We are also not restricted to the well-tempered 12-note scale. The software is open to more notes per octave as well as it is possible to use micro-intervals.

4 Program Development

The software development uses the C++ language and started as an application for Windows using MFC. In order to port the program to Linux and MacOS X the development switched to the wxWidgets[1] framework.

4.1 User-Interface

The user can input notes via his computer keyboard that automatically mimics the keys of a piano keyboard: 'Q'=c, '2'=c#, 'W'=d, '3'=d#, 'E'=e, 'R'=f, '5'=f# and so on; the lower octave starts with 'Y'=c, 'S'=c# etc.. By using the up- and down-arrow keys the user can switch to the next octave up or down. The notes are being played instantly via MIDI interface and printed directly on screen in western music notation. After a melody has been entered, it is possible to transform the input in a number of ways. For example, rubber-band-style selections can be made, notes can be transposed or deleted just like with a text editor. Series of notes can be inverted and reversed and the aforementioned fractal and chaining algorithms applied. These commands work on both selections or the entire score. A "Refresh" command exists for the fractal algorithms. It automatically reverts the transformation and gives simply "another shot", stepping through all possible alternatives. Of course, selections or the entire score can be played and stopped.

In future versions, all commands described here shall run in their own worker-

---

[1] Formerly known as wxWindows. The name has been changed to wxWidgets. See also www.wxwindows.org

thread, not in the user-interface thread, so the user-interface will not be blocked by any calculations. The program supports export of standard MIDI files. It is planned for future versions to support MIDI, XML and CSV file import in order to give more choices for the replacement of the cell database.

4.2 Classes

Following the above description of the algorithms and the user-interface, it is evident that we had to implement the classic Model-View-Controller paradigm. Frameworks like MFC or wxWidgets are built around this paradigm, so it makes sense to use them. Since wxWidgets supports all popular platforms it was chosen as framework for my development.

The representaion of music data as objects makes it necessary to design fundamental data structures and base classes. The software uses mainly two different container classes: An Array class, which is used to organise and save the note-cell data of our database. The Arrays save notes as midi-note numbers (ints) and they generate automatically information about the intervals they contain, which is a list of delta values. The whole database is kept as an Array of Arrays in memory. In order to facilitate ordering of the cells and random permutations of the database, pointers to the note-cell Arrays are kept in a Double-Linked List. The Double-Linked List is a template-based class which manages pointers to object instances. The melody imput from the user is also kept in an instance of the class Double-Linked List, where the elements consist of pointers to a note-table, that can be shared by other objects as well. The Algorithms described in this paper work on a copy of the melody note-list. Since the note-list is of type Double-Linked List, only pointers are being copied. The Algorithm class intitalises its own Array of notes, which itself creates a delta-list of intervals. Pointers to the delta-list elements are the stored inside a Double-Linked List object, so the Algorithms can easily work on the interval-list from the user input. This is done because there is a lot of comparison of interval sizes going on. The history-check that was described as an option

is implemented as a Visitor of the Algorithm. This object calculates a history table of all pitch-classes that have been used so far. It uses the Double-Linked List containing the intervals from the Algorithm object it is visiting. All Events that are sent to the MIDI interface are also managed by a Linked-List Container.

In order to build the editor for note display and to implement user-interaction, the Composite Design-Pattern will be used. All graphic elements will be either Component or Composite instances. For instance a Staff would be a Composite that contains other Components like Notes, Clefs, Barlines, etc..

5 Conclusion

The use of design-patterns like composite and vistor allows us to achieve a very robust code that is both easy to maintain and to extend. The paper also showed that it is possible to model composition techniques using object-oriented design of musical data. An application has been created that has the flexibility to extend knowledge gained from music analysis and personal experience. The initial goal of creating a sketchbook for composers has been achieved.

6 References

J. Dunsby and A. Whittall. 1988. Music Analysis in Theory and Practice. Faber, London

E. Gamma, R. Helm, R. Johnson and J. Vlissides. 1995. Design Patterns : Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA

Bruno R. Preiss. 1999. Data Structures and Algorithms with Object-Oriented Design Patterns in C++. Wiley, New York

M. Neifer. 2002. Porting MFC applications to Linux. http://www-106.ibm.com/developerworks/library/l-mfc/