ScoreViews - An Object-Oriented Approach to a Graphical Computer Music Synthesis Environment

Brett Terry
Univ. of Illinois, Urbana-Champaign
Urbana, IL 61801
terry@cnp.ncrl.music.uiuc.edu

Abstract
This paper describes the analysis and design of a C++/Smalltalk-Motif suite of programs for graphical instrument design, function editing, signal processing, and score editing. This environment, in addition to creating an organized framework for Music-N function-generation and sample-generation routines, promotes reusability and inheritance, and provides extendible classes that allow for rapid design of new instruments, sample-producing components, and parameter controllers with a minimum of re-compilation, and a increased amount of reusability achieved through inheritance (e.g. of GUI drawing operations).

Introduction
Embarking on a new software project, in consideration of the effort involved, necessarily proceeds from a requirements analysis--a detailed articulation of the program's intended use and audience, and an exposition of the means by which its creation can be justified, evaluated, and tested. Certainly, many object-oriented approaches to computer music synthesis currently exist, such as MODE (Pope, S. T., 1991), and KYMA (Scaletti, 1989). The aforementioned programs are written in Smalltalk-80, whereas ScoreViews is written in C++, which enforces some differences in design, including the possibility of support for multiple inheritance. KYMA runs on the Macintosh and uses special external hardware, and MODE is specifically intended "to be used by Smalltalk-80 programmers for building music representation languages, real-time performance systems, or new user interfaces for music applications." (Pope, 1991, p. 83)

ScoreViews, in contrast, seeks its niche as a comprehensive graphical compositional environment aiming to support: (1) the ability to make small variations in instrument design and listen to the results within a reasonable amount of time (dependent on the complexity of the instrument), (2) the ability to create a large number of interesting sounds from the basic distribution building blocks, (3) a framework to support additional components as they are developed, (4) a library of basic instruments to be used as models, (5) a flexible utility for designing control functions and waveforms (this is handled by the TableMaker program), (6) support for the most common synthesis algorithms while at the same time providing an interface conducive to experimentation, (7) a well-documented procedure for adding new components and parameter controllers, and (8) a clear delineation of sample-producing and soundSource modifying levels with abstract base classes and generic interfaces to permit common modifications (e.g. temolo, panning, and reverber) to remain distinct from unit-generator-based instruments.

Analysis and Design
Following loosely the methodology elucidated by Coad and Yourdon (1990), I utilized an object-oriented analysis approach by identifying objects, specifying their attributes, defining their operations, and articulating interobject communications. The Sound and SoundHeader objects encapsulate all machine-dependent playback routines (e.g. sound
library routines) and audio file formats. A Score object includes any number of Layer objects (determined by the maximum number of concurrently sounding events used in a piece), each of which contains Events (which possess a duration and some "patch" of Modules). Objects for sample-generation, signal-processing, and parameter control are situated in an inheritance tree with an abstract base class at the root of the tree.

This design permits manager classes to treat any subclass as if it were an instance of the base class (i.e. it need not know exactly what it is, but calling a function such as GetNextSample() will be correctly dispatched to the subclass--this is known as polymorphism), allows all shared operations (e.g. draw(), erase(), select()) to be coded just once at the top level and inherited, and permits slight modifications to be easily derived from other objects (e.g. making an interpolated table lookup oscillator a subclass of a truncating oscillator means only the GetNextSample() function has to be rewritten).

In addition, basic objects exist that can be used by many other objects at different levels, and by developers who might wish to build other objects using them. Examples of these objects include: Table, a hierarchy of Probability objects, the Sound object, Counter objects, Unit objects, and many small chaotic function-based objects.

**The Use of Abstract Base Classes**

The Module base class subsumes objects that must support an operation to return a Sound object as output given one as input. Many modules correspond to signal processing algorithms such as Reverb, Filters, Chorus, Mixers, and Spatial location/panning. The ModuleManager takes care of optimizing the generation and processing of Sound objects, including normalization. Different levels of inheritance provide for flexibility at the expense of speed, so that, for example, one can specify the response type of a Butterworth filter and be prompted for only the cutoff (or center) frequency (and the bandwidth, if necessary), or, one can use a more general class that requires the user to input the coefficients (which can be controlled via any controller type) for maximum time-varying flexibility. A network of connected Modules, which would typically include at least one user-created Instrument or Sound object, comprises an Event. Support for properties such as variable number of inputs can be accomplished using (multiple) inheritance of other base classes such as a varargs class.

The Component class subsumes objects involved with sample-creation. Familiar Music-N style algorithms such as lookup oscillators, FIR filters, and RAND-type objects are all supported within a graphical editing environment, the same one used for Event creation. In this case, however, a network of connected Components comprises an Instrument which can be represented as a Module while editing an Event.

The Controller class subsumes all values and functions to be used as input for parameters to components and modules. The default Controller value is a staticValue, which is just some number, coupled with a Unit object (belonging to the Component or Module). Other Controller types are Ramp, and fromTable which reads a function from any function created by the user. Other controller classes can be read from any combination of the building block objects including the probability and chaotic objects. For the sake of flexibility of time-varying control, any controller can quickly be substituted for any other, so that changing any parameter (e.g. oscillator pitch) from a value to a function requires only a choice in a dialog box, and is not hard-wired into the instrument design. That is to say, instruments contain a set number of controllers and default types, but these can always be changed.
Te arrieve at this level of flexibility, every Component, Module, and Controller must support some interface functions for relating to the editing dialog box. Essentially, each object can be queried for the number and contents of some number of text fields (float values), option menus (e.g. fromTable's list of currently defined tables), and labels.

Support for Common Synthesis Methods and Other Possibilities

Table-lookup synthesis can be accomplished with an Oscil component which includes one Table object. A xFade object is an example of a simple extension to the Oscil object: it uses three table objects: a source waveform, a destination waveform, and a controlling function. Oscillators can also perform random lookup on the same table (the Overtones object), at the expense of not using the Table object's default internal index. Non-linear Synthesis can be accomplished using one Table object as a waveguide. FM synthesis can be implemented with several Oscillator objects, and some Addition objects. Different wavetables can be loaded into any table object, to provide support for a facsimile of waveable synthesis. Work continues on an implementation of granular synthesis with a Matrix and Grain objects.

The real purpose, however, is not to repeat what has gone before, but to provide a rich library of objects that can produce a maximum number of timbres for compositional uses with a minimum amount of development time. Although other environments certainly provide such flexibility, an important feature of ScoreViews is that one can design an object using other objects, a lookup function, and some edit dialog interface functions, and that this object will, by inheritance, quickly and coherently fit into an established GUI environment.

The User Environment

In general, ScoreViews supports the familiar GUI double-click-to-edit paradigm, i.e. double-clicking on an event will open the EventBuilder editor, and double-clicking on an Instrument will open the InstrumentBuilder editor. Events can be created in a score by dragging or by typing into the corresponding eventList view. The duration of events can be altered graphically, and support for cutting and pasting is provided. The EventBuilder and InstrumentBuilder editors support a MAX-like environment with patch cords and objects, permitting slight variations of instruments to be developed and saved quickly. The tableMaker is a separate application that provides a variety of methods for table (function) creation and editing. Sample editing screens are shown below:
Bibliography


