ABSTRACT: Capella is an object–oriented graphical interface for algorithmic composition in Common Music. It defines classes of browsers and worksheets that implement a consistent set of visualization tools and serve as a graphical front end for the system. The interface currently runs on the Macintosh under Macintosh Common Lisp.

Introduction

Algorithmic composition is a complex activity in which both musical and technological issues must be addressed in parallel. This, in turn, places special requirements on a graphical interface that supports the process. Object–oriented composition environments such as Common Music, DMix, and Mode place additional demands on graphical tools due to the breadth of representation and functionality that these kind of systems implement. Smalltalk environments are able to take advantage of a powerful windowing system provided by Smalltalk itself. Since Common Music was designed to be as portable as possible, without the aid of a native windowing system, almost no attempt to address visualization issues was made until recently. Until now, visual output in Common Music was completely text–based, similar to the type of display one sees when working, for example, in a Unix shell window. Common Music’s command–line driven interpreter, Stella, connects to the system’s toolbox similar to the manner in which a shell connects to Unix. Although it allows powerful input expressions to be formulated, Stella does not allow the inner processes to be easily understood. Capella is a response to some of the communication limitations in Stella, while keeping in mind that graphic representation and mouse based gestures are not always the best or most expedient models to choose for interacting with a complex system. Capella has been designed to be a complement, not a replacement, for the two other modes of interaction supported by the system: command processing from Stella and procedure invocation from Lisp. Common Music simply runs all three modes “in parallel” (Figure 1) and the composer is free to choose whatever is most appropriate to a particular situation.

Capella is still in the early stages of development. Its primary goal is to allow a set of flexible visualization tools to be developed, but it also makes interacting with the system as a whole easier and more transparent. The need for transparency is particularly acute in algorithmic composition workshops, where participants must quickly absorb not just new theoretical concepts, but a specific implementation of them as well.

Browsers and Worksheets

Capella provides two basic classes of windows: browsers and worksheets. Browsers provide context sensitive views on musical objects and permit the associated data to be inspected and manipulated in some class specific manner. A basic premise behind the design of Capella is that there is not a single best way to view objects in the system, but rather, that their visualization depends upon context. Browsers in Capella are therefore not identical to the musical structures they display. Worksheets are windows that support compositional activities such as analysis gathering, musical event editing, musical structure definition, and score output processing. Each class of worksheets provides methods for a generic protocol controlling the creation, enabling, activation, and updating of worksheets and their subviews.

Any number of browsers and worksheets may be open on the screen at the same time but only one browser and worksheet are said to be “active” at any given time. The active browser is called the “focus browser” and typically provides selection constraints for the active worksheet. The system currently defines classes of worksheets for data

1 cf. Taube (1994)
2 cf. Oppenheim (1993)
3 cf. Pope (1992)
Score Description Toolboox

- Item Streams
- Classes
- Utilities

Services
- Scheduling
- API

Common Music Lisp Package

Common Music Composition Environment

Figure 1: Capella serves as a graphic “front end” to Common Music. This illustration depicts the three parallel interaction modes possible in Common Music: gesture (Capella), command (Stella) and procedure (Lisp).

Figure 2: A SEE browser with its control pane expanded. Four of the five possible dimensions are depicted for a set of Midi-Note events. The x-axis has been set to time, y-axis to pitch, color to channel, and length to duration. The brightness dimension was not used. The browser’s value access functions take care of mapping back and forth between symbolic, floating point or integer pitch representations.

editing, creation and deletion of musical objects, query processing, global preference setting, loading and saving archives, and score processing. Several of the browsers and worksheets will be discussed in more detail in the remaining sections of this paper.

Information Browser

The information browser provides class specific summary information about an object (its type, status, parents, number of subobjects, and so forth) and a table for displaying and editing the current slot values in the object. This table distinguishes between internal, external and “read only” slots. Normally, only external slots are included, and only “writable” slots have their input buffer active for editing. An “inspect” mode overrides this default behavior and gives full display and editing access to all the slots in the object.

Listing Browser

The listing browser (Figure 3) displays a containing object together with its subobjects. Subobjects are displayed by a printer with class specific methods. Subobjects that are “events” include parameter information in their output displays, so when a listing browser displays event data it provides formatted views for each parameterized sound event.

Listings can be used in conjunction with Edit worksheets for generic sequence editing. Operations on the Edit worksheet apply to the current selection in the focus listing browser. Subobjects in a listing may be selected by mouse gesture, by iterative index referencing, and by the application of musically salient predicate selection expressions.

SEE Browser

The SEE (Structured Event Editor) browser (Figure 2) is a visualization and analysis tool that operates on sequences of parametric, ordered data objects. These sequences may be structures declared in the system or be “virtual” sequences of merged output generated from multiple objects by the scheduler.

The SEE browser provides a programmable graphics output window that supports a number of user customization and display hooks. To use a SEE browser, the composer assigns a maximum of five “data parameters” to five dimensions of visualization supported by the browser. The first two dimensions determine the element’s location on a Cartesian pane (representing its x- and y-axis, respectively). The third dimension is drawn as an element’s “length” in the same direction as either the x- or y-axis. Brightness is used to represent a fourth dimension, and color (ie. hue and saturation) provides a fifth dimension. None of these dimensions are required to participate in the drawing process or be used to display a specific type of data.
Algorithm Browser

The algorithm browser (Figure 4) is a structured editor for displaying and modifying program code associated with musical algorithm and generator objects. The browser maintains two similar but completely independent editors for the object’s initialization and “run time” statements. The editors are separated by a divider; moving the divider up or down controls the percentage of space allocated to each editor in the browser. Each editor manages its program statements using a control menu and an internal code table. Statements are usually indexed in the menu under the name of the parameter they affect. Selecting a statement from the control menu installs it in the associated code table. Once a statement is installed in the table its index in the control menu is “greyed out” and cannot be reselected until the statement is deleted or deinstalled from the table.

The control menu can hold alternate expressions affecting the same parameter. This allows a composer to easily compare or test different expressions affecting the same parameter. Code tables permit their structure as well as their contents to be easily modified. Statements can be moved in the table simply by control–mouse–dragging them to a new position. Code tables also perform automatic syntax checks on their input expressions and will “pretty print” their contents upon request. Once code has been developed, the browser can either redefine the algorithm object associated with it, or else “decompile” the code tables into a Lisp expression that, if evaluated, would redefine the algorithm object. This Lisp expression is automatically dumped to a new window containing a Lisp editing buffer.

Output Browser

Output browsers permit mixtures of sequential and simultaneous arrangements of object references (layouts) to be
“layed out” and then processed to a specific output stream (Figure 5). The Stream and Layout panes allow the user to create, load, save and select layouts and streams from a menu. Once a particular layout has been selected it is graphically depicted in the layout pane itself.

A Layout is defined in terms of one or more “object references”. Each reference may represent a single object (for example an algorithm, thread or merge), or a subset (chunk) of subobjects from some containing object. A reference is graphically depicted as a box (block) in the layout pane. Blocks are mouse–draggable and self–adjust to the width of their textual content. If a block is moved it “snaps” to its horizontal and vertical neighbors after it has been released. All actions—the creation of new object references, duplication of existing ones, moving, selection, and changing the content—may be performed directly on the layout pane using standard command key combinations. Once the layout contains one or more references and an output stream has been selected, the Return key may be used to activate output processing. Layout processing moves from left to right across the pane: objects within a column are scheduled relative to one another and movement across columns represents “sectional” (non overlapping) divisions. Within a column, timeshifts relative to other objects may be specified using the @ <time> qualifier, and block repetition is possible using the repeat qualifier.

Conclusion

Although Capella is a working, functional interface, the project is still in its early stages and much work remains to be done. Short term goals include developing new classes of browsers for musical pattern display and statistical analysis. Intermediate goals include insuring the modularity of native windowing code, and isolating as much graphic behavior as possible in methods on generic windowing operators. Long term goals include porting Capella to other windowing systems and at least one public domain Lisp implementation. The first port will most likely involve X–Windows on the Silicon Graphics line of machines.

References

