The *Creation Station:* An Approach to a Multimedia Workstation

Henry S. Flurry

ABSTRACT

The Center for Performing Arts and Technology at the University of Michigan is developing the *Creation Station*, a workstation-based software package that will provide the artist advanced sound synthesis and graphics capabilities, as well as the tools necessary to create multimedia pieces of art. This paper discusses the design criteria, programming obstacles and implementation details of the *Creation Station*. Because the *Creation Station* is coded in Objective-C, a brief tutorial on Object-Oriented Programming Languages is included.

INTRODUCTION TO THE *CREATION STATION*

The Center for Performing Arts and Technology (CPAT) has been working for the past year on a multimedia electronic arts software product called the *Creation Station*. The team behind the *Creation Station* includes CPAT Director David Gregory, who first envisioned the *Creation Station*, and Associate Directors Hal Brokaw and Henry Flurry, principal architects and programmers of the *Creation Station*. This workstation-based package will provide the artist with advanced sound synthesis and graphics as well as the tools necessary to create multimedia pieces of art. We are attempting to present a unified environment that will reflect an intuitive understanding of multimedia integration while maintaining flexible and sophisticated storage, editing and performance capabilities. We also aim to provide a software foundation that will support different and perhaps yet-undeveloped aspects of computer-aided art, including computer-based art research.

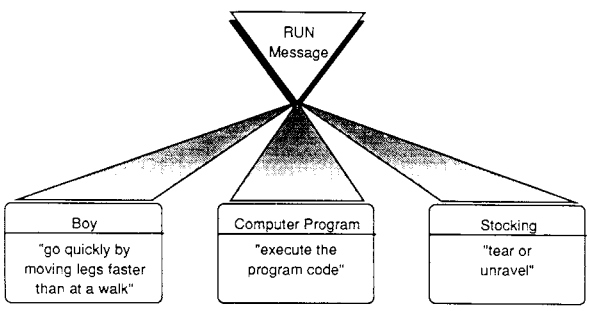
The *Creation Station* will employ a graphics interface similar to that pioneered by Xerox and made famous by the Apple Macintosh. We utilize traditional musical metaphors to present a homogeneous interface that will allow control of many multimedia artforms. For instance, one window will contain the master recorder panel that will govern the performance and recording of all events. From this window, the button labeled 'PLAY' will perform any *Creation Station*–based piece consisting of musical elements, graphic elements or both. Other windows will display events contained in tracks of the hierarchical track structure, allowing the user direct control over the type of event stored within each track and the sequence in which tracks are performed.

Because of the simplicity of this interface, the average user will find the *Creation Station* straightforward. However, to the more experienced user or programmer, the *Creation Station* will reveal many layers of power and flexibility. This paper primarily discusses how this power and flexibility are achieved.

The first version of the *Creation Station*, to be announced during the summer of 1988, will include a composer's application, a choreographer's application and the Instrument Builder. The composer's application will provide a variety of musical output devices and a number of ways to input and edit musical events. The choreographer's application will allow a choreographer to design stage props, input and edit three-dimensional motion paths for objects (presumably, but not necessarily, dancers) and view in three-dimensional perspective the resulting choreography. The Instrument Builder (not discussed in this paper) will be an icon-driven, device-independent, sound design module. One version of the Instrument Builder will allow the user to program sounds on almost any MIDI synthesizer with system exclusive capabilities.

Later versions of the *Creation Station* will provide other artistic tools (such as theater set and lighting design simulation, interactive performance tools, and interactive video editing), educational tools (such as musicianship and other art form lessons), and research tools (such as large sound and video databases and applications supporting interactive testing of theories). Depending upon the complexity of the implementations, the *Creation Station* will run on either super-micro or high-power workstation class computers.

Fig. 1. In Object-Oriented Programming Languages, a single message may effect different actions when applied to different objects. This is called polymorphism.



SOME SPECIFIC DESIGN CRITERIA

We aim to design an environment that will reflect the intuition of most artists but will also adapt to other creative styles. In view of this, we made the following design decisions:

- Tracks are hierarchical. Typically, an artist does not think of his/her creation as a single, indivisible chunk. A hierarchical track system allows the artist to organize

Henry S. Flurry, University of Michigan, School of Music, 1100 Baits Drive, Ann Arbor, MI 48109-2085, U.S.A.

Received 5 April 1988.

Pergamon Press plc. Printed in Great Britain.
0024-094X/88 $3.00+0.00

the creation into progressively smaller and more manageable groups.

- Performance order of tracks is not limited to the hierarchical structure of the tracks. This allows the artist to organize the creation within a structure not based upon time-span division.
- A single track may play in canon with itself, at any timing offset. This includes tracks containing either musical or non-musical events.
- The user may work in many different timing bases when editing events, editing conductor tracks and displaying real-time passage of performance time. This includes measures and beats, minutes and seconds, SMPTE, and user-defined timing bases.
- Tracks can follow whichever conductor track the user desires.
- The *Creation Station* will support third-party development of new types of artistic events, device drivers and editing environments.

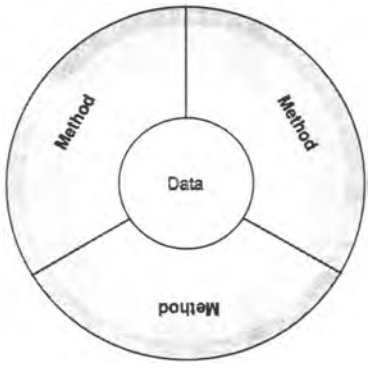
POTENTIAL PROGRAMMING OBSTACLES

Even before coding of the *Creation Station* started, it was clear that these and other criteria would prove challenging. Some of the potential obstacles were:

- How to create a software package that would seamlessly integrate an undefined and potentially large set of event classes (such as music, animation, dance, etc.), their associated device drivers and the event editing environments.
- How to synchronize events of different timing schemes (such as measures and beats, minutes and seconds or SMPTE).
- How to synchronize events of different media—especially if some events take longer to realize than others.
- How to allow a single track to play in canon with itself. We need to prevent interference between two simultaneous performances of a single track.
- How to enable the integration of third-party extensions to the *Creation Station.*

Although solutions to these and other problems could be implemented in many different programming languages, most programmers would agree that the complexity of the necessary code within languages such as C and Pascal would be almost too great to manage. However, in an Object-Oriented Programming Language (OOPL), many of the problems are diminished to the point of becoming trivial, and other programming tasks are conceptually simplified. For this and other reasons, we are developing the *Creation Station* in Objective-C [1].

Fig. 2. An object is presented as a single unified construct, where the methods of the object encapsulate the data of the object. Program code can only access an object's data through the object's methods.



INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING LANGUAGES AND OBJECTIVE-C

In most programming languages, data is passed to a procedure to be acted upon. For example, the calculation of the square root of x might be executed by the procedure call:

sqrt(*x*)

There is, somewhere inside the program code, a unique function accessed by the **sqrt()** call. This function accepts its argument, evaluates the square root of the argument, and returns the resulting value. However, **sqrt()** is expecting a particular type of value—typically a double float number. It is the responsibility of the programmer to make sure that the **sqrt()** function is not passed a character, integer, or user-defined structure. Moreover, the user must make sure that the function's return value is assigned to a variable of the proper *type.*

Because so much of the burden of data typing and procedure management is placed upon the programmer, there is a 'complexity barrier' which prevents economical coding and maintenance of complex systems [Winograd, 1979]. We believe the complexity of the *Creation Station* would have approached this barrier.

OOPLs such as Objective-C encourage a different approach by blurring the distinction between data and code. To explain briefly the concepts behind OOPLs and the terms used in them, I will start by discussing the *object,* the basic construct of an OOPL. You may think of an object as analogous to any entry within the 'real' world. Like anything else around us, an object shows various properties and behaviors.

To command an object to do something, one sends it a *message.* As in the real world, the same message sent to different objects may mean different things. For example, commanding the objects 'boy', 'computer program' and 'stocking' to 'run' would generate three very different actions. In OOPLs this is called *polymorphism*—the use of a single message to evoke different actions from different objects (see Fig. 1).

Objects may be of the same *class* or of different classes. A class is simply a definition of an object's properties and behaviors. Objects of the same class exhibit similar properties and behaviors, whereas objects of different classes will likely display different properties and behaviors. 'Fred', 'Kevin', and 'Nancy' may be elements of the class 'people', and they will all exhibit certain similarities; however, these three people are unique, even if they are of the same class. In OOPLs an *instance* of a class is an object which belongs to that class but has a unique identifier.

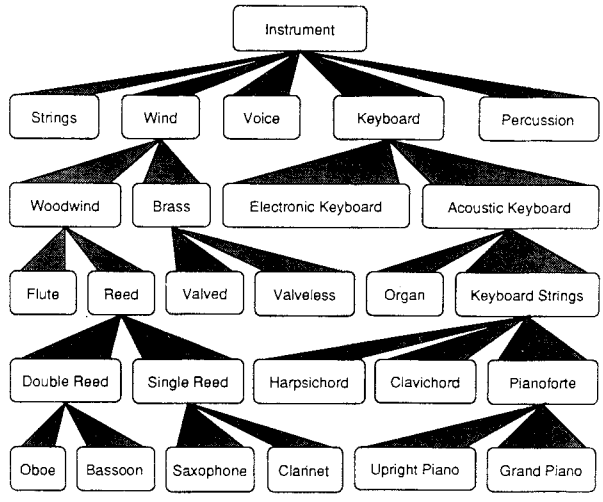
At this point, it becomes necessary to discuss the structure of an object. Within a class definition, the programmer defines an object's *instance variables* and *methods.* Methods are the 'procedures' associated with an object. A message sent to an object invokes a method of the same name, executing the program code of that method and returning a value to the calling routine. Instance variables are variables that are accessible only by the methods associated with a class instance. One may liken the instance variables to the 'properties' of an object, and the messages and methods to

the 'behavior' of an object. Thus, objects of the same class all respond to the same messages, execute the same methods and have the same data format for instance variables. However, different instances of the same class each contain their own copy of the instance variables, and the instance variable values associated with one instance may be changed without affecting the values of any other instances.

There are several other valuable characteristics of OOPLs, most notably encapsulation, inheritance and dynamic binding. The instance variables of an object can be accessed and changed only by the methods associated with that object. This process of information hiding, called *encapsulation*, has some obvious benefits to software design. Because the data of an object can be accessed only by that object's methods, there is less chance of accidental data corruption by outside routines. Once an object has been debugged, it can be used and reused as an entity with little concern for supporting code. In addition, the boundaries of class definitions help clarify the division of labor in group programming projects (see Fig. 2).

Inheritance is a natural by-product of the hierarchy of classes, in which all objects reside. In creating a class, the programmer defines only the differences between the desired class and the *superclass*. Instance variables and methods that are not redefined are inherited by the *subclass*, causing this new class to exhibit properties and behaviors similar to those of its superclass (see Fig. 3). Inheritance is a powerful capability of OOPLs. Valuable programming time is saved by writing only the code that is necessary to differentiate a new class from its superclass. Inheritance also extends the life cycle of an object definition by allowing a class to be tailored to the unique needs of other applications. A third aspect of OOPLs that is often overlooked is *dynamic binding*—the postponement of deciding what method to invoke for a particular message until run-time. Without dynamic binding, the ability to assign any object to a single variable would not exist, for the variable would have to be statically defined in order for the compiler to determine successfully which methods were bound to the coded messages. Dynamic binding also allows the creation of new classes that will work with existing precompiled code. Finally, dynamic binding will enable run-time linking with third-party extensions to the *Creation Station*.

Fig. 3. In this inheritance hierarchy, Flute inherits from Woodwind instrument, which in turn inherits from Wind instrument. Flute would include only those methods and instance variables which differentiate it from Woodwind.



PROGRAMMING IN OOPLS

Many traditional programmers find it difficult at first to program with OOPLs. Two basic concepts useful in designing OOPL software are also useful for understanding many programming decisions presented in this paper:

- The methods of an object should closely relate to the data held within the instance variables of that object.
- Objects with conceptual similarities should be acted upon with a common protocol.

The first concept is fundamental to programming within an OOPL. For example, it might make sense to have an object representing a musical note be responsible for playing itself as staccato, but it would not make sense to have this same note object be responsible for deleting a file from a computer disk. This would be counter-intuitive and would make management of the programming project difficult.

The second concept is not as clear as the first. It does not dictate how to program in an OOPL as much as it displays the power of dynamic binding combined with polymorphism (dynamic polymorphism). Consider the generic class **Event,** which might encompass any type of performable event in which we would be interested. We can look at this group of event classes and begin to formulate a list of several conceptually common methods:

Event Classes:
- musical events
- dance events
- video events
- etc.

Possible Common Methods:
- perform next event
- return to real time of the next event to occur
- compare event attack-point timings of two individual events within the same class

Each event class would more than likely execute each method differently: the performance of a musical event might produce sound, and the realization of a dance event might generate visual images. However, the methods applied to the event classes share the same general concepts. Thus, if these methods carried a common protocol (i.e. respond to the same set of messages), it would be possible to write code that would control, for an object of any event class, many of the desired functions.

For example, to perform an event—be it musical or graphic—we could write the single line of code

```
[an Event performNextEvent]
```

which, in Objective-C, sends the message **performNextEvent** to the event object stored in the variable *an Event*. By changing the value of the variable *an Event*, we could perform any event defined within the *Creation Station*.

In fact, this is how we handle many objects of similar functions. Dynamic polymorphism allows us to define different 'genres' of objects, where each object of a single genre implements a set of messages in conceptually similar but physically different ways. We can then use the same code to control different objects of one genre to produce varying results.

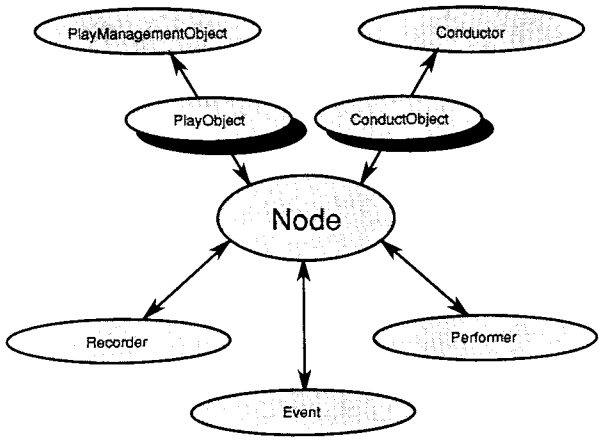


Fig. 4. The *Creation Station* communication channels. To maintain modularity of objects, the Node coordinates much of the inter-object communications. The PlayObject and ConductObject are little more than added levels of indirection of communication between the Node and the PlayManagementObject, and the Node and the Conductor.

THE *CREATION STATION* BASICS

There are six major objects that form the foundation of the *Creation Station.* These objects, to be further discussed later, are:

- Event: Objects of this genre store performable events and contain the code necessary for converting the stored events into a representation acceptable for a Performer.
- Node: A Node is the track of the 'hierarchical track tree'. A Node contains an Event list and provides links to the Nodes below and above it.
- PlayManagementObject: This object is responsible for sequencing during performance all of the events within the *Creation Station.*
- Conductor: An object of this genre is responsible for controlling the performance and recording tempos of any Node or set of Nodes.
- Performer: These objects are responsible for realizing at a specified time individual events received from the PlayManagementObject.
- Recorder: Recorder objects accept external input and translate this input into time-stamped events that may be stored within a Node.

The Event, Conductor, Performer, and Recorder objects all define genres of object classes, and different objects of the same genre may generally be interchanged. Two other objects, the PlayObject and the ConductObject, will be introduced later in this paper.

Time. There will be only one type of timing base used in communication between objects. If one object is to pass a time value to another object, then this time value must be converted to the 'standard timing base' and stored in a variable declared by the macro TIME or in an instance of the class *Time.* The rule is necessary for two reasons:

- All objects must be able to read and understand any timing values received from another object.
- The PlayManagementObject must be able to sort and sequence correctly multimedia events (such as music or video) that may be defined in different timing bases (such as measures and beats, or SMPTE).

The Node

As mentioned above, the Node contains an event list which may be played in synchronization with other event lists within the *Creation Station.* The Node stores the events in a modified AVL tree, a structure that I call the 'Linked Balanced Binary Tree'. This structure was designed to optimize the operations that would most commonly be applied to an event list: insertion, deletion, searching, changing, and sequencing (for performance). A strict AVL tree provides relatively fast insertion and deletion, optimal searching and changing, but slow sequencing [Wirth]. To accelerate sequencing, we modified the AVL structure so that each Node of the binary tree contains not only a link to the Nodes above and below it, but also to the Nodes sequentially before and after it. Sequencing through an event list is quickly accomplished by following the second set of links. Thus, the structure behaves both like an AVL tree and like a linked list.

To maintain maximum modularity, the Node coordinates much of the inter-object communications necessary for event performance and recording. The six objects described above—"*Creation Station* Basics"—must gather information from a number of other objects in order to perform their duties properly. If we allowed each of these objects direct access to the other objects containing pertinent information, we would begin to lose inherent modularity. Experience has shown that a system that loses its modularity becomes more difficult to maintain and more prone to failure. Thus, with the possible exceptions of the PlayObject and ConductObject (discussed later), the Node is the only object of the objects that communicates with more than one other object (see Fig. 4).

So, when the PlayManagementObject needs to know the performance timing of each event, the Node is responsible for communicating with the Conductor and the Event to get this timing. When the PlayManagementObject is ready to perform an event, the Node receives the timing back from the PlayManagementObject, retrieves the event data from an Event, and passes the two to a Performer object. During recording, a Recorder object passes received event data to the Node. The Node then requests its Event class to create an event instance, and the Node inserts this new event instance into its event list.

The Event

The Event is one of the objects that makes the *Creation Station* so versatile. An object of the Event genre is responsible for the following:

- Storing whatever information is necessary to define an event or set of events to be performed.
- Providing both the code necessary to convert the internal representation of an event into a construct acceptable by a Performer object and the code necessary to convert Recorder output into an Event object.
- Providing the code necessary to order two different instances of the particular class of Event (the **compare:** method).

- Sequencing events to be performed in the proper order (the **returnNextEvent:** method).

We can create a variety of events that, by conforming to the above rules and following the same messaging protocol as other event classes, may be used within the *Creation Station.* Beyond these guidelines, there are no restrictions to the event classes.

As an example, it is possible to create an event class where each musical event is timed in measures and beats. In order to have the Node store the events in the proper sequence, the event class implements a method named **compare:**, which examines the measure and beat timings of two event objects and returns a value indicating which object should be sequentially first.

In our choreography application, we implement an event class whose instances store four-dimensional splines (3-D space plus time) defining motion paths of dancers. When an instance of this class receives the message **returnNextEvent:** it calculates and returns a three-dimensional point in space based upon the performance time.

Finally, we could even define an event class which implements an interpreted programming language. Each event object might contain a line of code or a procedure within a user-written program. The **compare:** message of the Event class could sort the procedures or lines of code in the order they were input by the user. When an Event object is issued in the **returnNextEvent:** message, it would execute a segment of the stored program to construct the event to be returned.

The Performer

The Performer objects provide the link between the internal representation of events and the external realization of events. As in the Event genre, there may be many different types of Performer objects, where each of these objects connects the *Creation Station* to some artistic medium.

A Performer object accepts a request from the Node to perform an event at a particular time. The event to be performed is encoded within some construct the Performer object can parse, and the time is passed in the standard timing base. A Performer should make any lengthy calculations necessary for event performance, buffer the results or events for later realization and return control to the Node. There are a variety of ways to buffer events; two of the most common ways are for the Performer to arrange an 'interrupt' to occur at the time the event is to be performed, or, if the *Creation Station* is running on a multitasking machine, for the Performer to send the event to another process which would handle the event realization at the proper time.

Different Performers require various event constructs, and a Performer should not be sent an incompatible event construct. It is easy to imagine accidentally sending a choreographic event construct to a MIDI output performer. On the other hand, a Performer object should be able to accept event constructs from a multitude of similar event classes. For example, both an Event class which simply stores musical events and an Event class which computes musical events should be able to use the same MIDI Performer.

To solve these potential problems, the *Creation Station* implements a system which matches Event classes to compatible Performer objects. Each individual Event class has a 'Performance Format Name' (PFN), a string of ASCII characters which is unique to the type of event construct output by that Event class. Likewise, each Performer object has a PFN which specifies what type of event construct is accepted. When a user wants to choose a Performer for a Node, the Node matches the PFN of its associated Event class with the PFNs of the available performers. The resulting set of Performers is presented to the user, who presumably chooses out of this set a Performer object to realize the events stored within the Node (see Fig. 5).

With this system, it is easy to create new classes of Event objects which will work with pre-existing Performers. New Performers that work with existing Events can likewise be created to support new hardware.

The Recorder

The Recorder object is responsible for accepting input from an external source and converting it to an event construct acceptable to an Event class. It is similar to the Performer in that it needs to be matched with compatible Event classes. Both the Recorder objects and Event classes have 'Recording Format Names' (RFNs) associated with them. As above, the Node is responsible for matching its Event class with potential Recorder objects, and, also as above, it is easy to integrate a new Recorder class into the *Creation Station* if it matches an existing RFN.

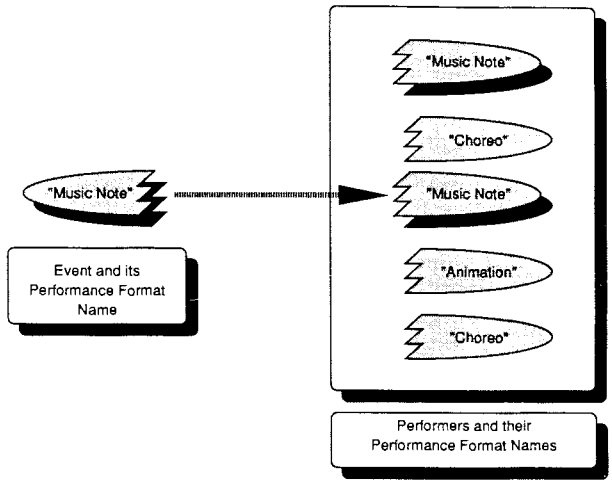
The Recorder is additionally responsible for buffering its external input until the end of a performance is reached. At this time, the Recorder sends each event to its associated Node to be translated into an Event object and stored within that Node.

The Conductor

The Conductor is a very important object to the *Creation Station.* Not only does a Conductor provide a tempo map for performance within the *Creation Station,* it also can be employed by the user to define an arbitrary timing base. The main duties of the Conductor object are:

- To translate virtual time to real time, and vice versa, basing the conversion upon the tempo map stored within the Conductor instance.

Fig. 5. The Performance Format Name is used to match up Events with compatible Performers.



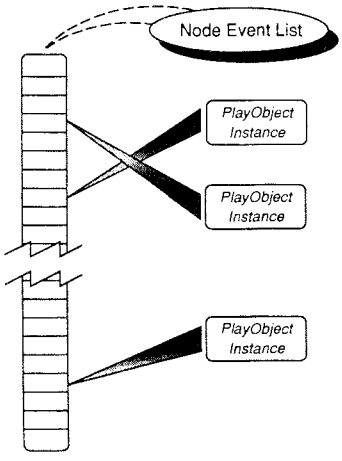


Fig. 6. The PlayObject can be thought of as 'saving a Node's place in an Event list'. Thus, a single Node may have several performances that overlap as long as there are PlayObjects to maintain each performance's place.

- To translate the user-defined timing base to the standard timing base, and vice versa.

There can be any number of Conductor objects active during a performance, with each Conductor controlling a single Node or a set of Nodes. In fact, it is entirely possible to have every Node within a performance following a different Conductor. In any case, the Node is responsible for coordinating the Conductor's translation of an event's virtual time to real time (for performance) and vice versa (for recording).

Before a user can input a tempo map into a conductor, he/she must define the timing base to be used. For example, the user might wish to create a tempo map for a piece of traditional Western music. In this case, the user would probably define the timing base as measures and beats and give the Conductor the number of beats in each measure before constructing the tempo map.

ConductObjects

One aspect of the Conductor and the Node not yet discussed is that of 'offset times'. The timings of events within a Node are all relative to the performance time of the Node. Thus, if a Node starts performing its events 3 seconds into a performance, the event within the Node set to occur at time 2 seconds will actually occur at 3 + 2 = 5 seconds from the start of the performance.

In addition, in order for a Conductor to translate correctly between virtual time and real time, it must keep track of its offset time, the delay between performance start and Conductor start. To help with this, we have created the **ConductObject.** A ConductObject maintains the Conductor's time of start and any memory required by the Conductor to keep track of time translation—everything necessary to re-establish a Conductor's state of conducting. ConductObjects are always used by the Node to communicate with the Conductor, so the ConductObject can forward extra information to the Conductor along with the Node's message.

A Node may request a particular Conductor, or a Node may inherit a Conductor from a previous Node. In the latter case, the actual Conductor is not inherited, but rather the ConductObject, so that time translation remains consistent.

PlayObjects

During performance with the *Creation Station,* it is possible that the user may request that a Node be performed several times. It is further possible that these Node performances may overlap. This presents some potential problems: during a performance, a Node needs to store certain information, such as its performance offset and active ConductObject. Likewise, the Event objects may need to keep track of information. If two or more performances of a Node overlap, conflicts could occur among the data pertinent to each performance.

Similar to the ConductObject, the **PlayObject** resolves any potential conflicts by keeping track of performance states for the Node and its associated Event objects (see Fig. 6). The information stored in the PlayObject needs to be passed to the Node every time the PlayManagementObject wishes to communicate with the Node. This means that the PlayObject must receive and relay the appropriate messages from the PlayManagementObject to the Node.

The PlayManagementObject

The PlayManagementObject is a simple object with two purposes:

- Manage and sequence a list of PlayObjects during performance.
- Update the real-time display during a performance.

There is only one instance of the PlayManagementObject within the *Creation Station.* When a performance begins, the PlayManagementObject creates a PlayObject for the top Node in the hierarchical track system, which in turn creates PlayObjects for other Nodes within the tree. The PlayManagementObject keeps track of the events managed by each PlayObject and correctly sequences the events of the whole hierarchy until either no more events are available or the user interrupts the performance.

If, during the performance, the PlayManagementObject is well ahead of the Performer buffers, the PlayManagementObject will command an object called ScrollWindow to display the performance time. Each active ScrollWindow is associated with a ConductObject and displays the performance time in the user-defined timing base. This allows the user to have performance time displayed in any timing base desired, including measures and beats. If events are not being processed fast enough, the PlayManagementObject may never request the ScrollWindow to update the displayed time.

THE EDITOR AND EDITORUI

These two objects will provide the user with event list editing capabilities. Although these objects have not yet been created, they will follow many of the same ideas as other objects within the *Creation Station.* Objects of the Editor genre will contain the code that edits events of a certain class. Objects of the EditorUI genre (named for 'Editor User Interface') will be responsible for providing the graphics user interface to its associated Editor object. It is possible to have many EditorUIs that will work with a single Editor object, so we will implement a match system that works with the Editor Format Name (EFN). An Editor will have an EFN associated with it, and it will be matched with available EditorUIs of the same EFN. In this way, it will be easy to change the user interface while keeping the same editing functions.

CONCLUSION

As the *Creation Station* nears completion, it becomes increasingly clear how much our project has benefited from our using Objective-C:

- The *Creation Station* is easily expandable, with little or no recompilation of existing code. This includes easy addition of new Event

classes and user interfaces, and easy integration of new hardware.
- The user interface is highly flexible, providing seamless integration and synchronization of a multitude of Event classes, device drivers and editing environments.
- Our program code more closely models how we conceptualize the *Creation Station.* In fact, most of the objects we generate have direct correlations within the user interface. This includes the Node, the Conductor, and the PlayManagementObject, which is directly linked with the master recorder panel.
- We have generated surprisingly little code in very little time to perform the desired tasks.
- Our code is highly reusable. In fact, we have been able to share objects that were originally created to support very specific tasks.
- General program management was much easier than expected.
- We soon expect to support third-party development for the *Creation Station.* This would not be possible without OOPL dynamic binding.

Many people have contributed to the *Creation Station,* and we have been able to implement innumerable ideas into one software package. Because the *Creation Station* is such a flexible system, we expect it to fill countless niches within the artistic world.

Note

1. Objective-C is a registered trademark of the Stepstone Corporation.

Bibliography

Brokaw, Hal, Henry Flurry, Phil Mackenzie and Jeff Stillson, "Center for Performing Arts and Technology Preliminary Documentation of Objects" (University of Michigan, 1988).

Cox, Brad J., *Object-Oriented Programming: An Evolutionary Approach* (Reading, MA: Addison-Wesley, 1986).

Gregory, David, "A Proposal for a Center for Performing Arts and Technology" (University of Michigan, 1987).

Krasner, Glenn, "Machine Tongues VIII: The Design of a Smalltalk Music System", *Computer Music Journal* **4**, No. 4, 4–14 (1980).

Ledbetter, Lamar, and Brad Cox, "Software-ICs", *BYTE* (June 1985) pp. 307–316.

Lieberman, Henry, "Machine Tongues IX: Object-Oriented Programming", *Computer Music Journal* **6**, No. 3, 8–21 (1982).

Pascoe, Geoffrey A., "Elements of Object-Oriented Programming", *BYTE* (August 1986) pp. 139–144.

Rodet, Xavier, and Pierre Cointe, "FORMES: Composition and Scheduling Processes", *Computer Music Journal* **8**, No. 3, 32–50 (1985).

Stepstone Corporation, "Technical Specifications: Objective-C Language Version 3.3" (Connecticut: Stepstone Corporation, n.d.).

Wirth, Niklaus. *Algorithms and Data Structures* (Englewood Cliffs, NJ: Prentice-Hall, 1986).