Animal: Graphical Data Definition and Manipulation in Real Time

Article *in* Computer Music Journal · November 1992

DOI: 10.2307/3680768

2 authors, including:

Eric Lindemann
Synful
27 PUBLICATIONS **219** CITATIONS

Eric Lindemann and Maurizio de Cecco
IRCAM: Institut de Recherche et Coordination Acoustique/Musique (IRCAM)
31, rue Saint-Merri
F-75004 Paris, France
elind@ircam.fr

Animal: Graphical Data Definition and Manipulation in Real Time

Animal (Animated Language) is a rapid software prototyping tool designed for experimentation with real-time signal processing and event processing systems. The special emphasis is on computer music applications. Animal is an objected-oriented programming environment with the usual notions of class, method, and instance. A class is defined in Animal by designing its graphical representations. These representations then serve as interfaces for creating and manipulating networks of "live" instances of classes. Animal provides for persistent storage on disk of instance networks, and for reusable libraries of classes, graphic representations, and instances.

Animal is intended as a tool for building "fine-grained" graphic applications—musical event list editors, synthesizer patch editors, and so forth. A graphical interface for a sampling synthesizer system designed with Animal will be discussed in detail and referred to throughout this article.

The IRCAM Musical Workstation Environment

Animal applications are intended to run in the IRCAM Musical Workstation (IMW) environment (Lindemann et al. 1991). The IMW consists of a NeXT host computer, which communicates with a high-speed general-purpose multiprocessor. The multiprocessor, designed at IRCAM, is configured as plug-in boards for the NeXT cube. Each board uses two 40-MHz Intel i860 processors along with a Motorola DSP560001 processor, which is used as input/output processor and direct memory access (DMA) controller. Up to three of these boards may be plugged into one NeXT cube. All real-time event processing and signal processing is carried out by the i860 processors. The host computer is used as graphics and file server as well as a software development platform.

A real-time operating system, CPOS (Viara 1991), and a "toolbox" for supporting real-time musical applications, FTS (Puckette 1991a), have been developed for the IMW multiprocessor. When an Animal application runs in this distributed environment the Animal objects live on the i860 multiprocessor but their graphic representations live on the host computer. The FTS toolbox provides support for creating and deleting objects on the multiprocessor, dispatching messages between objects, and sending and receiving messages from the host.

Existing Systems

Rapid prototyping tools for graphic applications fall into two main categories: database management systems with bundled interface builders, and stand-alone user interface management systems (UIMS). Examples of commercial database systems with interface support are 4th Dimension (Ribardiere 1987), VBASE (Andrews and Harris 1987), and, to a lesser extent, Hypercard (Apple 1987). Research systems include SNAP (Bryce and Hull 1990), ISIS (Goldman et al. 1990), and SIG (Maier and Nordquist 1990).

The second category of tools are stand-alone user interface management systems (UIMS). Examples of these systems are NeXT Interface Builder (NeXT 1989) and UIMX for X Windows/Motif systems (Visual Edge 1989).

Computer Music Journal, Vol. 15, No. 3, Fall 1991,

Database Systems

Animal is more closely related to the database systems mentioned above. These provide a framework for defining complex data objects, for browsing and manipulating dynamic collections of these objects, and for defining links between objects.

Animal differs from most database interface systems in its use of analog representations of primitive class attributes. Size and position of graphic objects are frequently used to represent quantitative data. Animal also encourages the representation of sets of objects by groups of icons arranged on a surface. Here again, size and position represent data.

Animal represents composite class structures by representations within representations. This technique is used in SIG and 4th Dimension. Animal provides for multiple fragmentary representations of classes. This is supported in 4th Dimension in the context of input and output forms for relational schema representation.

Animal is concerned with representation and manipulation of object networks. Structured query does not play a major role. As a result, Animal could not be considered a database system. In addition, database management systems are geared toward efficient search and retrieval of records stored from secondary media. Since Animal is used for building real-time musical applications, object networks are generally required to stay resident in memory. Animal does, however, provide support for disk-resident objects in the form of reusable object libraries.

UIMS

The second category of systems, stand-alone UIMSs, are quite useful for designing control panels and dialogue boxes. These are more or less static interfaces with fixed numbers of control objects—buttons, sliders, and text fields—arranged in static displays. The control objects available in these systems generally come from a predefined set. While there may be support for adding new control objects to this set, there is generally no support for designing new control objects. This exercise is left for the programmer. These systems provide no framework for manipulating dynamic collections of objects.

The Sampler Example

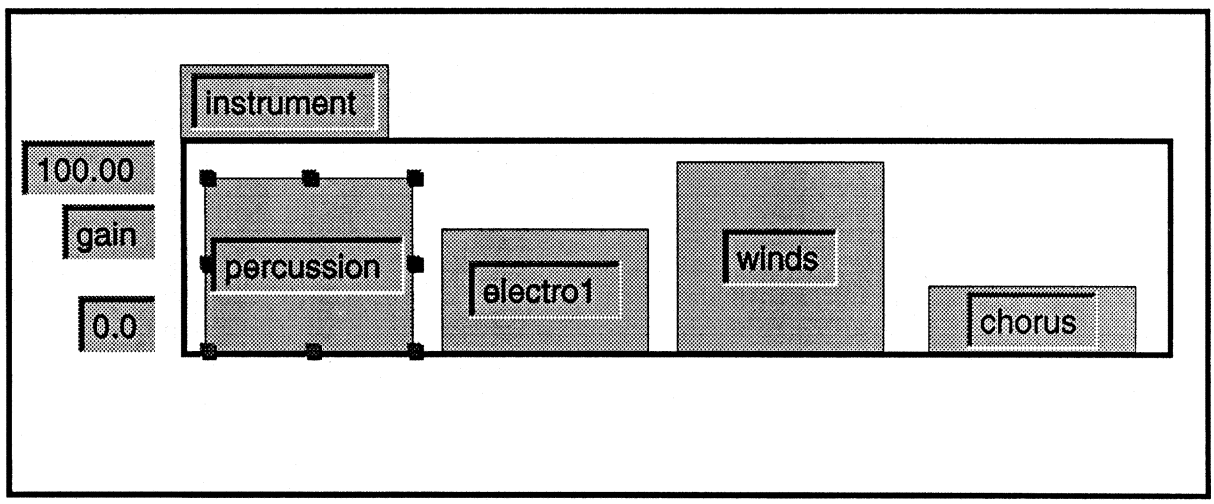
This section describes, from the end-user's point of view, the user interface for a sampling synthesizer designed using Animal. Subsequent sections provide a detailed look at the various issues involved in the construction of this interface.

The interface is hierarchical in nature, appearing as a tree of windows. At the top level of the hierarchy is the orchestra window shown in Fig. 1. The orchestra consists of a collection of virtual instruments represented by rectangular instrument icons. The relative height of the instrument icons represents the relative overall amplitude of the instrument. By resizing the icon using the mouse this relative amplitude can be adjusted. New instruments can be created by "cloning" the prototype instrument icon that appears above and to the left of the instrument collection. Double-clicking the mouse on an instrument icon opens its instrument window, as can be seen in Fig. 2.

The instrument window is dominated by a collection of "samplePatch" icons, which have been mapped onto a rectangular region called the "keymap." The keymap provides a coordinate system with MIDI pitch on the vertical axis and MIDI velocity on the horizontal axis, so that each samplePatch icon covers a region in pitch-velocity space. A sample patch object encapsulates a single sampled sound (e.g., a recording of a gong or chime) and a number of controls associated with that sound. When a note is played on a MIDI keyboard controller, or in general when the instrument object receives a "note-on" message, those sample patches whose mapping covers the point specified by the pitch and velocity of the note-on message will be activated. Overlapping sample patches are permitted, so more than one sample patch may be activated by a given note-on message.

New samplePatches can be cloned from the pro-

Fig. 1. Orchestra window in run mode.



totype found above and to the left of the keymap. Double-clicking on a samplePatch opens the samplePatch window, visible in Fig. 3, with its internal structure consisting of a sound icon, an amplitude envelope editor, a pole plot of a two-pole filter, and a center frequency/bandwidth indicator for the same two-pole filter. The two-pole filter coefficients can be adjusted either by dragging the "X" representations in the pole plot or by dragging and resizing the light gray rectangle in the center frequency/bandwidth display. Several number boxes display in numeric form the same variables that are displayed in analog form by the samplePatch icon in the instrument window. Double-clicking on the sound icon will send a message to the IMW Signal Editor (Eckel 1990), which can display time and frequency domain representations of the sampled sound and provides a rich set of graphical signal editing tools.

The Structure of an Animal Application

An Animal class definition specifies a template data structure consisting of primitive objects (floating-point number, integer, string, etc.), arrays of primitive objects, pointers to other objects which are instances of classes defined in the class table, and lists or sets of pointers to objects. A set of methods is also associated with the class. An Animal application maintains a table of class definitions.

The class system Animal uses is defined by the FTS toolbox, which forms part of the IRCAM Musical Workstation. When Animal generates class definitions it is generating FTS class definitions. The FTS class system is built on top of C and is compatible with C++. In line with C++ compatibility, pointers to objects are typed. Pointers may only refer to objects of the specified type (class) or to one of its subtypes (subclasses). The FTS toolbox provides special support for incremental class definition, run-time type checking, and dynamic linking of methods.

All the objects in an Animal application are instances of classes in the Animal application class table. These objects live in main memory on the real-time multiprocessor. The graphic representations of these classes live on the host computer. The methods for classes generated using Animal are defined by the application designer and written in C or C++. All of these methods run on the real-time multiprocessor. The Animal application designer writes no code that runs on the host.

With their pointer references, the objects in an Animal application form an evolving object network. Animal maintains a separate "proxy network" on the host computer, which is a direct reflection of the object network on the multiprocessor. This relationship is shown in the top part of Fig. 4.

Fig. 2. Instrument window in run mode.

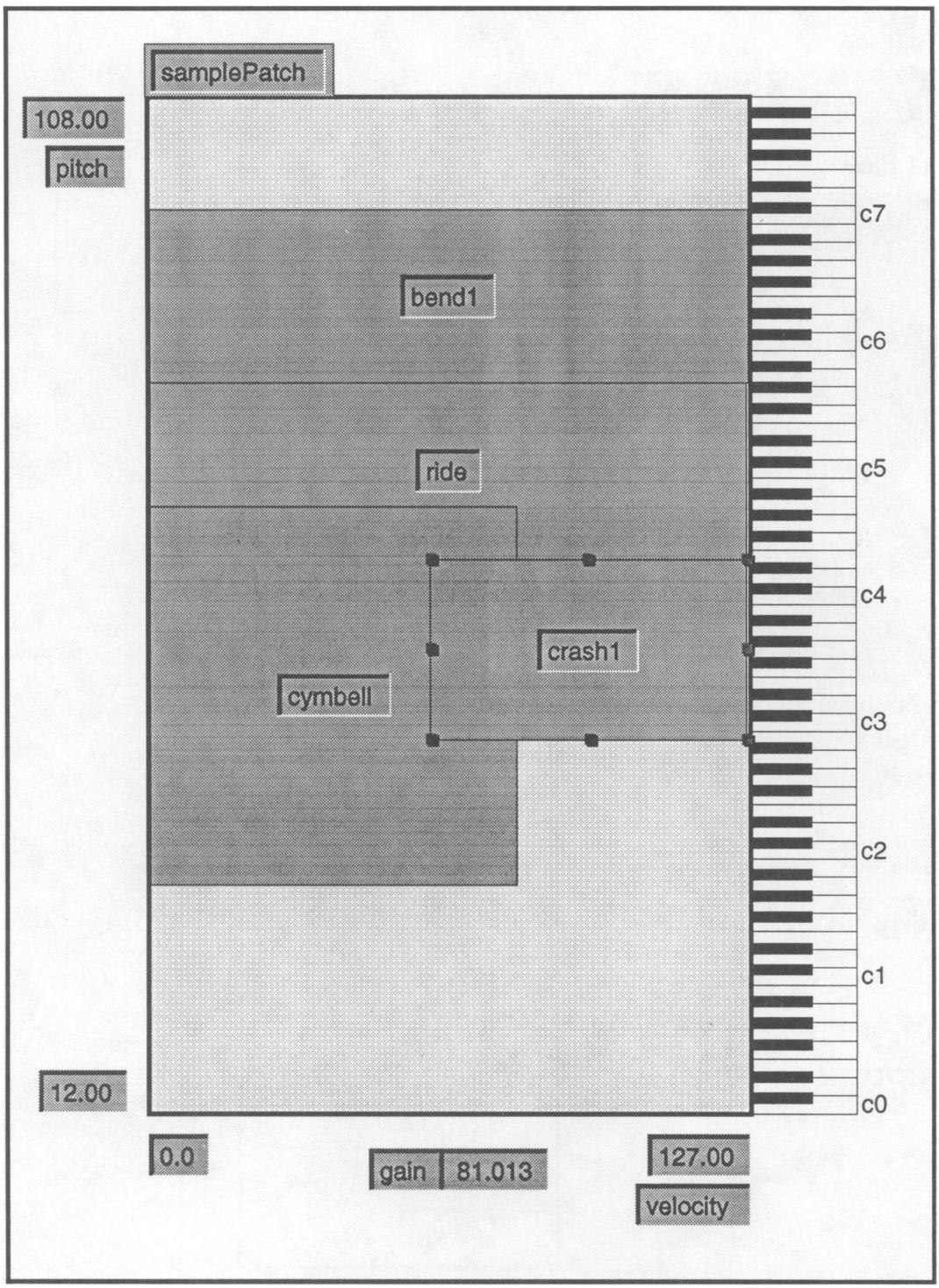


Fig. 3. SamplePatch window in edit mode.

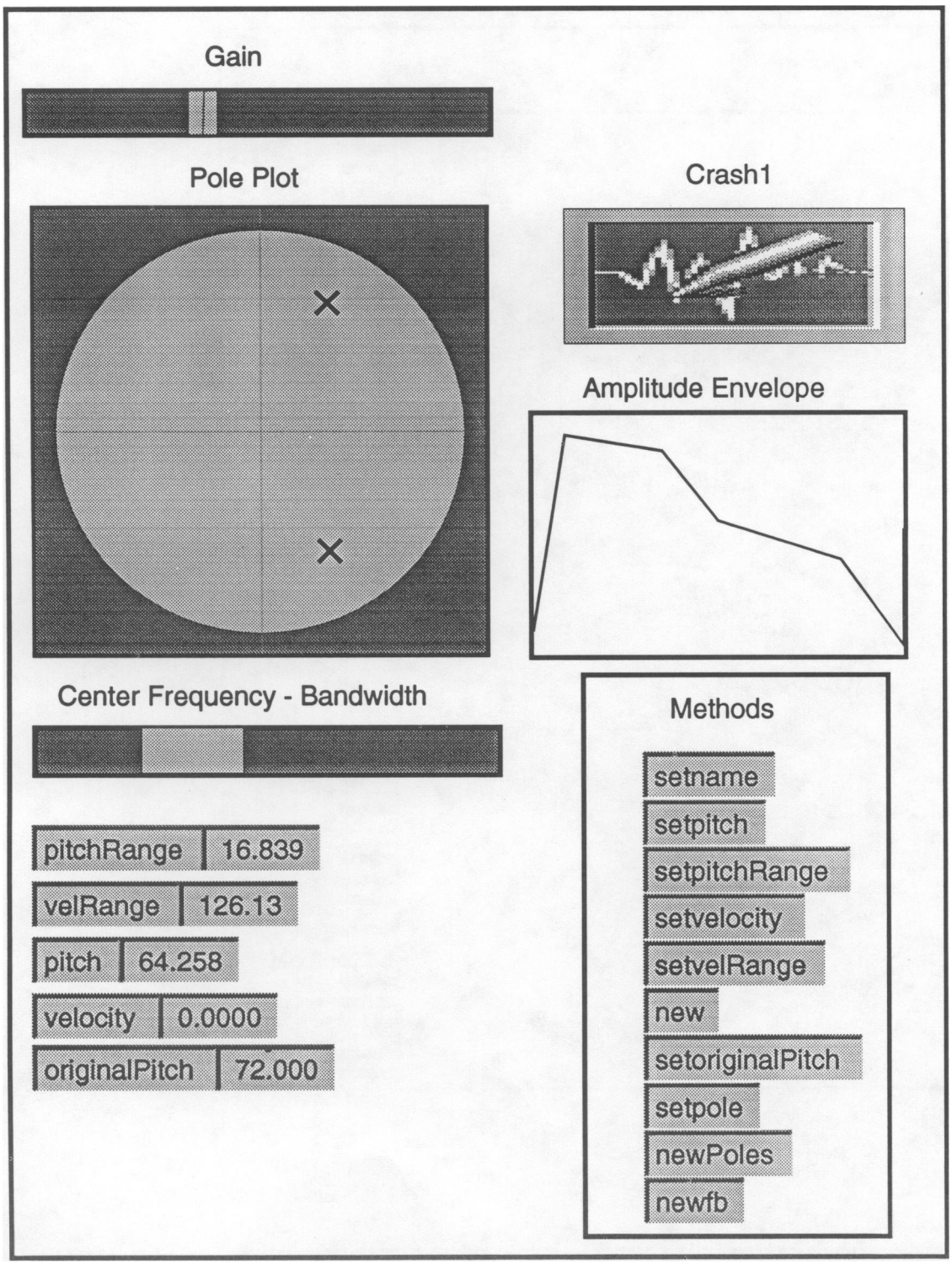
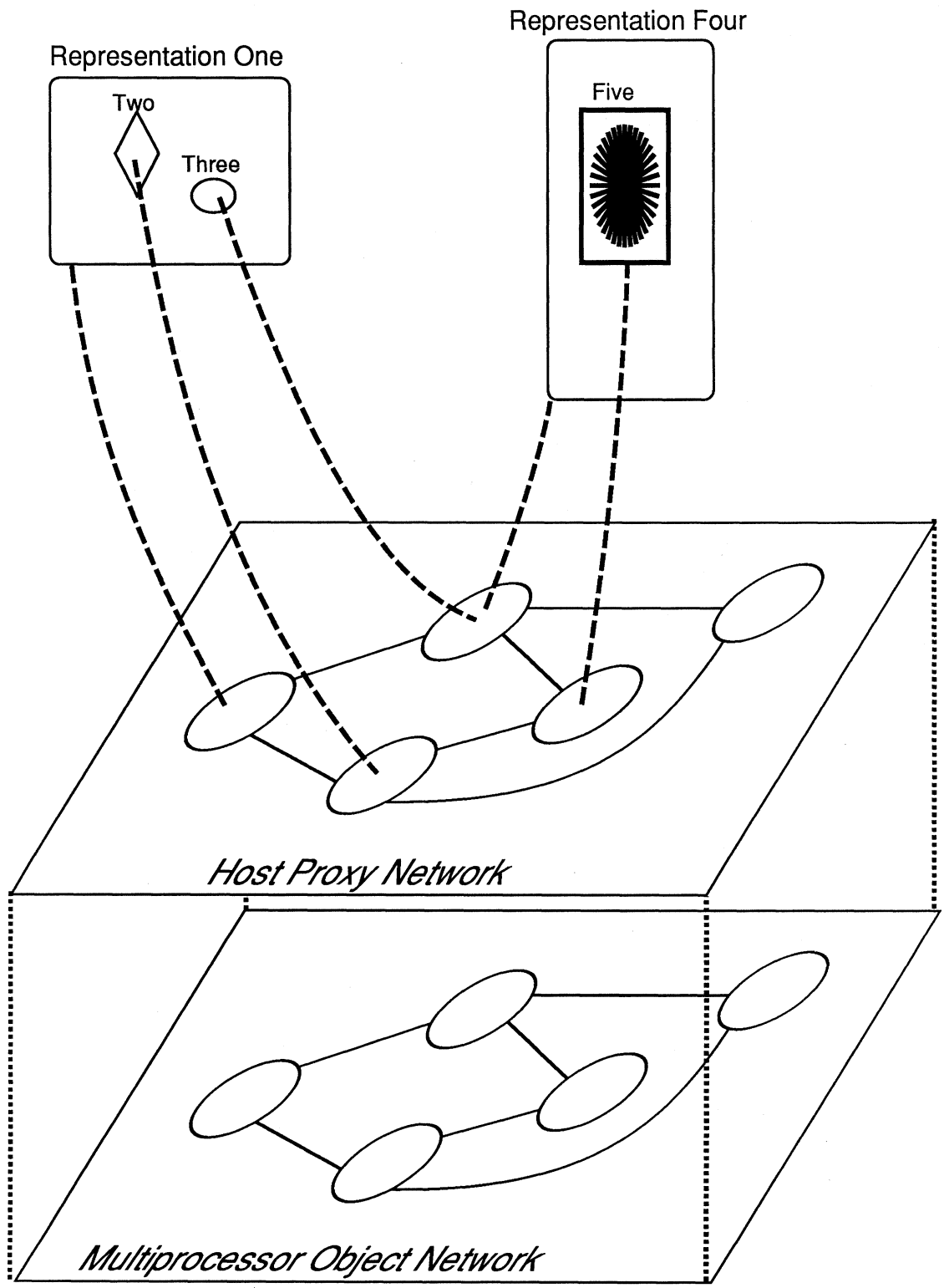


Fig. 4. Object networks.



There may be one or more graphic representations defined for each class in the class table. A graphic representation is said to be bound to the class it represents. A graphic representation maintains slots that are bound to instance variables of the class. There may be more than one slot bound to the same instance variable. Slots, in turn, contain representations of the instance variables they are bound to.

These instance variables may be primitive objects or pointers to complex objects. The representation contained in a slot may be a complex representation of the object pointed to by this instance variable. Representations are thus recursive structures, with representations containing slots containing representations, etc.

Example

The sampler instrument window is a representation of the Instrument class. It has a slot that contains a representation called "keymap." The keymap is a representation of the Set class. The Set representation has a variable number of slots for representations of instances of the SamplePatch class. The representation of instances of the SamplePatch class used in the keymap takes the form of a rectangular icon. The position, width, and height of these icons represent the values of instance variables of the samplePatch class. There is also an independent window representation of the samplePatch class. The instance variables that are represented by the samplePatch icon are also represented as number boxes in the samplePatch window.

At any time there may be many windows visible on screen with representations of instances of different classes, of different instances of the same class, or with multiple representations of the same instance.

Example

In the samplePatch window there are two representations of a single instance of the TwoPoleFilter class. One representation shows a standard Z-plane pole plot. The other is a linear center frequency/bandwidth representation. The graphic representations interact with their multiprocessor instances through the intermediary of the proxy network. This relationship is shown in the bottom part of Fig. 4. Each object in the proxy network maintains a pointer to its counterpart on the multiprocessor. The graphic representations maintain pointers to their respective instances in the proxy networks. As the instance variables of objects are modified on the multiprocessor, update messages are sent to the corresponding proxy objects in the host network. These update messages reflect all the changes of value that have occurred in the object. The update message is sent explicitly during method execution on the multiprocessor.

The application designer defines these methods and, thereby, controls when host updates occur. The update message is always of the same form—**UPDATE**. This is a macro provided by the FTS toolkit that remains the same regardless of the class structure of the object. When a proxy object receives an update message it broadcasts it to all visible representations of that object. These messages are optimized so that only information necessary for each representation is included in each message.

When the user interacts with a representation in a way that is intended to modify some instance variable—dragging one of the poles in the pole plot representation of the two-pole filter, for example—then a message is sent to the proxy object, which in turn sends a message to the multiprocessor object. Update messages are also sent to any other visible representations of the instance, so, for example, the center frequency/bandwidth representation would be updated.

Why couldn't we dispense with the host's proxy network and have the multiprocessor objects interact directly with their graphic representations? This would require that the multiprocessor objects be kept informed of the comings and goings of their various graphic representations and broadcast multiple update messages directly to them. This would weigh down the system, whose main responsibility is to keep up with the real-time calculation load.

The proxy network also maintains certain kinds of instance data which are not sent to the multiprocessor. Instance specific comments typed into a text field are an example. The archival of an Ani-

mal application is accomplished by archival of the proxy network. The multiprocessor system is sent a message to update all host objects. When this is complete, the proxy network is archived along with the class table for the application. The multiprocessor network is not archived directly. While restoring the proxy network on the host, messages are sent to the FTS "object factory" to rebuild the multiprocessor network in the image of the proxy network.

Instance Variable Properties

A number of properties are associated with the instance variables in a class.

Type

This property defines the type or class of the variable. Subclasses qualify as the same type as their ancestors.

Unit

This defines the unit of measurement for a primitive numeric type (centimeters, Hertz, etc.). The significance of these will become apparent when we discuss Animal rulers.

Archive

This property applies to pointer instance variables. When the representation of one object in the object network is copied, generally the intention is not simply to copy that object but to copy a subnetwork of objects emanating from that object. In the instrument window, if one clones a new samplePatch object with the intention of mapping it onto the keymap, one must also clone a new twoPoleFilter and gain slider. The sound pointer in the samplePatch is more complex, however. A new sound should not be cloned, but it is unclear whether the new sound pointer should refer to the old sound or should be void. The `archive` property of a pointer defines this kind of behavior.

It can be seen that `archive` properties define the scope of composite Animal objects. This is because all instance creation in Animal is based on cloning existing instances. This cloning procedure is accomplished by first archiving an instance to a stream—in memory or on disk—and then restoring it as many times as needed.

There are three possible values for the archive property: `Copy`, `Refer_To_Copy_Else_Old` and `Refer_To_Copy_Else_Void`.

A value of `Copy` implies that the system should copy the object pointed to.

Setting the `archive` property to `Refer_To_Copy_Else_Old` will not force a copy, but, if as a result of a different pointer path the object originally pointed to was copied as part of the subnetwork, then the pointer will be set to point to the new copy. Otherwise it will refer to the original object pointed to.

The meaning of the value `Refer_To_Copy_Else_Void` is the same as `Refer_To_Copy_Else_Old` except that the pointer will point to a void address if the object pointed to was not part of the copied sub-network.

Example

Figure 5a shows an object graph for a hypothetical application. Every pointer instance variable corresponds to a directed arc in this graph. Figure 5b shows the result of copying node A. Note that both B and C were copied since the archive property of the instance variables of A which point to B and C are both `Copy`. The `archive` property of the instance variable of B that points to E is `Refer_To_Copy_Else_Void`. E was not copied, so the copy of the instance variable that pointed to E now points to void (i.e., nowhere). D is copied since the instance variable of B pointing to D has `archive` property `Copy`. The instance variable of C pointing to D has `archive` property `Refer_To_Copy_Else_Old` so, C' points to D'. However, when C' is copied by itself when C'' still points to D' as shown in Fig. 5c.

Graphical Class Representations

As described in the previous section, a representation can contain slots that contain other representations. The position and size of a representation

Fig. 5. (a) Original object network. (b) Copy of subnetwork with root at A. (c) Copy of subnetwork with root at C'.

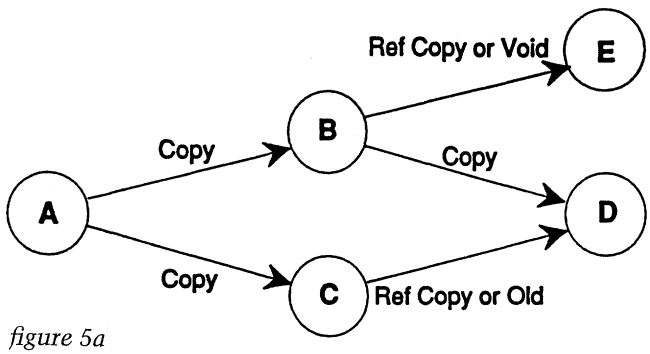


figure 5a

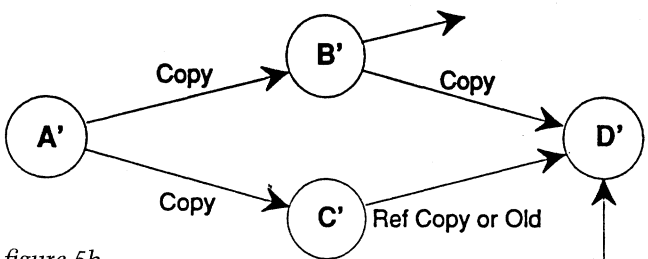


figure 5b

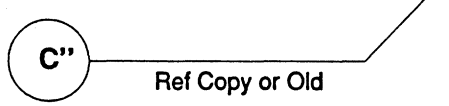


figure 5c

may be used to represent numeric values. Any representation can therefore be considered to have four implicit slots—*x, y, width,* and *height*—containing analog numerical representations. A representation may also contain purely decorative elements that are not considered to occupy slots.

Example

In the full window representation of the Orchestra class, the different instruments are represented by rectangle drawings whose height represents the "gain" instance variable of the Instrument class. In the pole plot representation inside the samplePatch window the center position of the "X" shaped elements is used to represent the complex *magnitude_phase* instance variable of the twoPoleFilter class. A representation can be bound to a class. During the course of designing Animal applications this binding may change. A slot can be bound to an instance variable of a class. There may be more than one slot bound to the same instance variable.

A slot may be left unbound. That is, it may not be associated with any instance variable of a class. The slot may nevertheless contain a representation. This representation will be bound with the first class encountered when climbing up the "slot hierarchy." For example, if the unbound slot is itself a part of a representation which is bound to a class, then the representation contained in the unbound slot will also be bound to this class.

Example

In the samplePatch window there is a representation of a slider labeled "GAIN." This slider representation occupies a slot in the samplePatch window representation. This slot has been left unbound. The slider representation itself has a slot that contains an analog representation of a primitive numeric value—the "knobby." Because the slider representation has been left unbound, the slot containing the knobby representation can be bound directly with the samplePatch class and the position of the knobby can be used to represent a numeric instance variable of this class. It is important to note that the slider representation has not been completely decimated by this binding scheme. It still maintains its identity as an interface object and still occupies a slot. The slider, in fact, plays an active interface role because it contains a Ruler object that measures the position of the knobby and constrains the knobby to move only within certain limits. Rulers and related objects are discussed in detail below.

Some representations may appear in stand-alone windows. There is nothing privileged about these representations. The same class representation may appear in one place as a separate window and in another place imbedded inside a pointer slot of another representation.

Slot Properties

The behavior of slots is defined by a number of properties. The *representation* property selects the graphic representations used by the slot. There may be a menu of possible representations specified from which the end user can choose. If the slot is bound to a pointer instance variable, then the representation property specifies the particular class representation to be used when displaying this instance variable. A representation must be specified for the basic class of the pointer. Additional representations may be specified for any subclasses.

A *null class* representation is used when the pointer is void. If no null representation is specified, then one is automatically derived from the basic class representation. The derived null representation is usually just the image of the basic representation drawn in light gray.

A *selected* representation is used when the slot and its representation are selected. If the representation is selected and no selected representation has been specified, then one is derived by highlighting the standard representation.

A *full window* representation is used if opening a new window on double-click is enabled using the "Open" property. One kind of full window representation involves a call to another host application to open a file name associated with the object. The full window representation of sound in the samplePatch window involves a call to the IMW's Signal Editor application.

Example

In the samplePatch window only one basic sound representation has been specified. If the sound is removed a null representation is derived from this to show that the sound slot is empty.

Assignment and Replacement Properties

For slots bound to primitive instance variables the *assignment* property is ignored and the *replacement* property specifies whether a value can be changed. For slots bound to pointer instance variables, these Boolean properties tell whether a representation of an instance can be placed in the slot, possibly replacing the instance already there. This corresponds to an assignment into the instance variable pointer the slot is bound to. In the case of a newly created instance being assigned to the pointer, the accept/reject assignment policy of the slot also depends on the archive property of the pointer instance variable being assigned.

If the *archive* property is one of the two flavors of reference mentioned in the previous section, then the assignment is rejected since the newly created object would be lost if the object is ever copied or archived to disk. Assignment of *reference copy* representations do not have this problem. When the archive is made the system guarantees that only one copy of a given object is archived even if the object is referred to from two slots, both of which have a `Copy` archive property. If the slot is not bound to an instance variable, then assignment and replacement are disabled.

Example

In the samplePatch window only the Sound slot has assignment and replacement enabled. The archive property of the sound pointer instance variable is `Refer_To_Copy_Else_Void`. This is because sounds are all supposed to be taken from a shared sound library (see the section on reusable libraries below). As a result, only a reference to a sound can be placed in this slot. The other pointer slots—the two-pole filter representations and the gain slider—have assignment and replacement disabled. The number boxes have replacement enabled so the values can be modified.

Copying and Removal Properties

These only apply to slots bound to pointer instance variables. The *remove* property determines whether the representation currently in a slot can be removed from the slot, corresponding to deassignment of the instance variable. There are two copy

properties. *Copy* tells whether a deep copy can be made of the representation and of the instance it represents. *Reference* tells whether a reference copy can be made.

Example

In the Instrument window the slot that the "prototype" samplePatch appears in, which is just above and to the left of the keymap, has its *remove* and *reference* properties disabled, but *copy* is enabled. This allows the prototype to be cloned and new instances placed into the keymap. In general, this approach can be used to set up palettes of clonable objects. The sound slot of the samplePatch window has *remove* and *reference* enabled but *copy* disabled, since copies should only be made explicitly from the sound library.

Drag and Resize Properties

The *drag* property allows the slot and the representation inside of it to be dragged around until they encounter a "wall" (see below). There is an implicit wall along the perimeter of every representation, so that a slot and the representation inside cannot be dragged out of the representation that contains the slot. *Resize* allows the dimensions of the slot and representation inside to be resized as long as the resize does not get blocked by a wall. *Drag* and *resize* may be constrained to be horizontal or vertical only. The *select* and *open* properties make it possible to select a slot or open its full window representation. *Open* works only if a full window representation has been specified.

Properties Associated with Instance Variables

A slot can be considered to have the properties that are associated with the instance variable it is bound to. For example, *type* is associated with an instance variable so the slot can be considered to have a type. Different slots bound to the same pointer may have different slot properties. This allows context-dependent user interface behavior.

Regions, Walls, Rulers, and Sets

A region delimits a rectangular space within a representation. As mentioned above, the perimeter of a representation forms an implicit walled region for the slots inside the representation.

Additional explicit regions can be defined within a representation. A region is a kind of prefabricated representation that may or may not be bound to a class. A region possesses a *region_type* property which may be different from the type of the class the region is bound to. The main function of a region is to generate system messages when a slot or a representation of the specified type enters or exits the region.

Walls, Rulers, and Sets are kinds of Regions. These form a multiple inheritance cluster in the sense that a Region may possess any combination of Wall, Ruler, or Set properties. These properties are presented to the Animal application designer as a number of options for the basic Region representation.

Region Extent

For a simple region the extent is the same as the perimeter. A region may be made scrollable, however, in which case the perimeter is considered to be a window into a larger space. This space may be finite or infinite in any of four directions—up, down, left, or right.

Wall Properties

The wall properties of a region may allow entry and exit for slots of the *region_type* or one of its subtypes. They may allow entry only, exit only, or no entry or exit. Slots not of the *region_type* are always excluded. Since *region_type* can be specified as the root class in the inheritance tree, this means a region could allow any types to enter. Note that entry and exit rights refer to slots, not to representations inside a slot.

A Region representation may be bound to a Region class. This is a real class with instances that live on the multiprocessor. Subclasses of the region class can implement methods that respond to entry and exit of objects. These methods can override the default region behavior and provide their own criteria for accepting or rejecting entry or exit.

A number of things may occur when attempting to drag a slot containing a representation into a region. If the slot is of the correct type, and entry is enabled, then the slot and its representation are allowed to pass into the region. If the slot is of the wrong type or entry is disabled and the slot has its *remove* property disabled, then the slot and its representation are blocked at the region wall. If, on the other hand, the slot's *remove* property is enabled, then the slot will be blocked at the region wall and its representation will be dragged out of the slot and into the region. At this point the slot will display its null representation. If the newly unbound representation is deposited in a slot inside the region, then it will stay there. If it is let go without being accepted by a slot in the region, then it will spring back to the slot from which it came. These same dynamics apply to dragging slots and representations out of regions.

Ruler Properties

A region can define a metric space. Ruler regions have *unit* properties associated with their *x* and *y* axes. Variables associated with the position and size of representations, which have the same *unit* properties as the ruler axes, will be measured by these axes. Rulers have minimum and maximum values specified over their measurement region. A Ruler representation may be bound to a Ruler class. The minimum and maximum values of the ruler representation can be bound to instance variables of this class so that they can be manipulated by application-specific methods running on the multiprocessor. A Ruler that has been bound to a class can have additional variables associated with its own size and position. These variables can be measured by other Rulers, so that a system of nested moving coordinate systems can be set up.

Set Properties

If a region has its *set* property enabled, then unbound representations of the correct type which enter the region will have a slot automatically created for them, permitting them to remain in the region. The slot and the representation inside it become part of the region set. A minimum and maximum *range* property is associated with the set; these specify the minimum and maximum number of objects the set can hold and may be zero and infinity, respectively. A Set is always bound to a real set class. The instances of Set exist as objects on the multiprocessor and can be manipulated by application-specific real-time methods.

Set Slot Properties

These are the slot properties of the slots in the set. They all have the same slot properties. A set slot has all the same properties as a standard slot. If a representation is removed from a set slot and deposited elsewhere, then the original set slot disappears. For this reason the assignment property of set slots is ignored. The set slot properties are not to be confused with the slot properties of the slot that the Set itself occupies. If a Set is moved, all the objects in the Set move with it. Any of the properties of the slot the Set occupies may be enabled: *drag, copy, replace, assignment,* and so forth. This allows copying and moving sets of objects as a unit.

Example

In the instrument window of the sampler example, the keymap is a Region with Wall, Ruler, and Set properties enabled. The *region_type* is samplePatch. Its Wall properties deny entry and exit to all slots, so only unbound objects can be dragged in. If an unbound samplePatch is dragged into the region, then a samplePatch slot is created and the samplePatch becomes part of the keymap Set. This is usually done by "deep-copying" the prototype samplePatch, which appears above and toward the left of the keymap. Representations that are not samplePatches will be rejected. The set slot proper-

ties have *replace, copy, remove, drag,* and *resize* enabled. This permits samplePatches inside the keymap to be copied and mapped anywhere across the region. *Reference* is disabled, so no reference copies of samplePatches can be made. The variables associated with the position and size of the samplePatch representation have pitch and velocity units. These correspond to the pitch and velocity units that are part of the Ruler properties of the keymap region. The keymap region is nonscrollable.

Using Animal Applications

Only a handful of operations are available to the end user of an Animal application. These include the instantiation of objects, where by objects we mean instances of classes from the class table and representations of these instances; placement of these objects in slots; creation of references to objects, which are new representations of old instances; movement of objects between slots; and alteration of the values of primitive variables. The user also controls visibility and arrangement of windows.

To move an object, one simply drags its representation with the mouse. To create an object, an existing object can be cloned by dragging on it with the "alt" key depressed. This makes a deep copy of the object.

Dragging on an existing object with the "control" key depressed creates a reference copy of that object. A new representation (that refers to the same instance) is created and can be stored in some appropriate slot. The ability to move, deep copy, or reference copy depends on the `Copy` and `Drag` properties of the source slot. Objects can also be created by opening the class browser, selecting an object type, and dragging the object that appears in the icon view of the browser while the "alt" key is held down.

When "alt-dragging" or "control-dragging" to create an object or reference, the destination of the drag must be a suitable slot. The slot type must be correct and the slot assignment properties must permit the completion of the assignment. If assignment is rejected, then no new object is created. Graphical feedback is provided when copying and moving objects. When copying, the original representation is left in place and a second one moves with the mouse. When removing a representation from a slot the slot shows the null representation. When a representation is dragged over its destination slot the representation to be replaced will be highlighted if the assignment can be completed. If the mouse is released without the representation being successfully assigned to a slot, then the representation springs back to its original slot.

When an assignment is completed, the representation of the class that appears in the slot depends on the slot representation property, rather than that of the representation that was dragged. A full window representation can be dragged into a tiny slot and will result in a tiny icon representation if that is what is specified by the destination slot. Values of instance variables are modified by resizing or moving objects that have had variables associated with their size or position, or by typing in number or text boxes. One can also click and drag on number boxes to alter their values.

Designing Animal Applications

The designer of an Animal application must create classes, add and delete instance variables from classes, define methods for classes, create graphic representations of classes, bind graphical representations to classes, create slots and regions inside class representations, bind slots to instance variables, and define the "choreography" of the application—that is, which windows and representations appear as a result of which events.

The application designer performs all the user functions of instance creation and assignment as well, so that an application is initialized with a proper default object network with all instance variables set to appropriate values.

The environment for creating and modifying classes is essentially an extension of the user environment. The designer sees all the same graphic class representations that the user sees, and it is by direct manipulation of these representations that classes are created and modified. Animal encourages a flexible approach to application design. One

Fig. 6. (a) Region declared as walled ruler in edit mode. (b) Knobbie added to gain slider. Variables and point of origin will disappear in run mode. (c) The slider has been grouped, but we leave it unbound so that the internal gain variable continues to be bound to the gain instance variable of the surrounding representation.

can start by drawing a picture, then declare this picture to be a representation of a new or existing class. Alternatively, one can start by defining a new class, and then attach some existing representations to it. These representations may be "stolen" from other classes. One can create instances of a class and afterwards modify the class structure. This causes an automatic update of all existing instances of the class. One can add or modify representations of classes that have existing instances. The philosophy is incremental, unconstrained application prototyping through progressive refinement.

Many of the operations involved in application design (the various binding operations, instance variable creation, etc.) are performed implicitly while designing representations. Almost all class definition is accomplished through design of the class representation. The philosophy is, as much as possible, to provide the illusion of "drawing" the application.

Animal has two modes of operation—edit mode and run mode. The user of an application is always in run mode. The designer of an application may switch between the two. When copying, dragging, and resizing representations of instances one uses the standard "Pointer Tool." In edit mode a new tool, the "Designer" tool, becomes available. The Designer tool is used to manipulate instance variables and slots in the same way the Standard tool is used to manipulate instances and representations. There are a number of inspectors that can be used throughout the design process. The representation, slot, class, and region inspectors are among the most important.

To create a new full window class representation one executes the **`New Window`** command in the Representation submenu. A class table browser pops up with options for selecting a class for the representation, creating a new class, or canceling the operation. Note that a full window representation is always bound to a class. In edit mode one can immediately begin drawing in the new window. The standard "Macdraw" interaction style is used. Drawing tools are selected to create lines, circles, and polygons. Line widths, gray levels, and fill patterns can be specified. In addition to the standard draw tools and the Designers tool, special Region and Method tools are available.

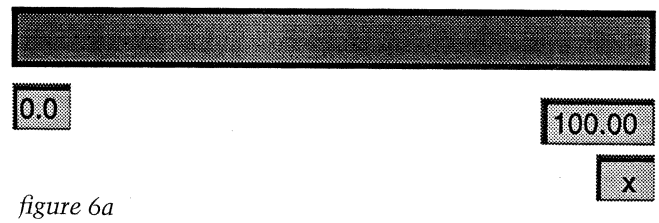


figure 6a

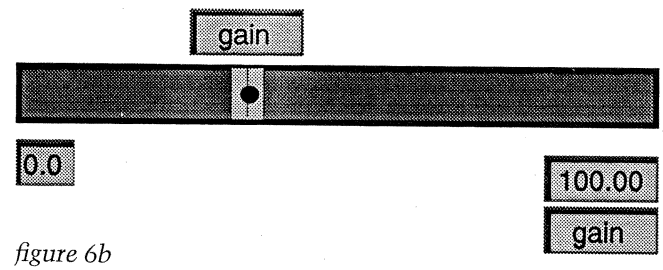


figure 6b

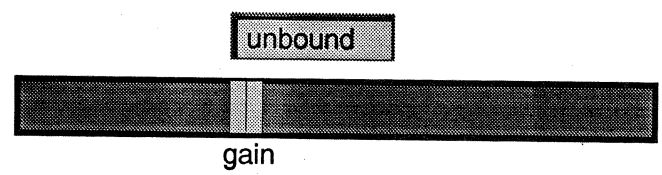


figure 6c

Example: Creating the Gain Slider

Suppose that we want to construct the GAIN slider in the samplePatch window of the sampler example. First, we draw a Region using the Region tool. The region looks like a rectangle the size of the slider. We use the region inspector to declare the region to be a Walled Ruler. This pops up number boxes around the newly declared Ruler region, where we can set the minimum and maximum scale of the ruler as in Fig. 6a. We keep the default of 0 to 100.

Next, we design a knobbie inside the ruler by drawing a filled gray rectangle with a vertical line in the middle and grouping the line with the rectangle. We want to associate the knobbie with a gain variable so we select it and enable an *x* position variable using the representation inspector, which

provides a choice of *x, y, height,* and *width* variables. This is shown in Fig. 6b. Declaring the knobbie as a primitive numeric representation automatically creates a slot for it in the full window representation. A point of origin marker appears in the center of the knobbie. We can move this around if we wish, but we choose not to. A text box appears next to the knobbie with an indication that the slot is currently unbound. We type in "gain" as the name of the instance variable that we wish to bind to. If the samplePatch class represented by the full window has a numeric "gain" instance variable already defined we will bind to it. If not, one will automatically be created with type float. The type can be overridden in the class inspector. We set slot properties of the knobbie using the slot inspector. We want to be able drag it horizontally. Everything else is disabled. We also type gain as the name of the variable the ruler will measure, overriding the ruler's default "x."

Next, we group the ruler and the knobbie to indicate that we want these elements to be part of a single representation so that we can easily reuse it the next time we need a slider. This is shown in Fig. 6c. The grouping action creates a new representation with two slots, one for a ruler and another for a numeric value. The slots that the ruler and knobbie occupied in the samplePatch representation are removed. A new text box appears that allows us to specify a variable we want the slider to bind to and indicates that the slider is currently unbound. Since the slider is unbound the gain knobbie remains bound to the samplePatch gain instance variable. Happily, this is exactly what we want. The variable name text boxes and the point of origin marker appear only when the objects are selected using the Designer tool.

Sometime later we might decide that we do in fact want the slider to represent an instance of a Slider class. We can select it with the Designer tool and type a name of an instance variable in the unbound text box. We type gainSlider. Since there is no gainSlider variable in the samplePatch class, and since it is not a numeric primitive representation, a class browser appears with a field in which we can type a new class name. The field already shows an intelligent-guess "GainSlider". We select New to create the class. A gainSlider instance variable is created in the samplePatch class and the slider representation is bound to it.

Since the slider representation is now bound to a class, the knobbie representation will become unbound from the samplePatch gain instance variable and bound to a new gain variable, which will be automatically created in the GainSlider class.

Grouping and Merging

Grouping representations always creates a new representation with slots for all of its components. "Merging" representations, on the other hand, tries to combine the slots of all representations into a single one. There are rules associated with merging and unmerging representations. If multiple primitive representations are merged, the effect is the same as grouping. Regions and their decedents are considered to be primitive in this context. If primitive representations are merged with a nonprimitive representation, then the primitive representations are added to the nonprimitive one.

Example

In defining the pole plot representation, we may already have defined a representation for the background, merging rulers and decorations. If we then define the "X" objects as primitive numeric representations and merge them with the rest, they will be added to the pole plot representation.

If we merge multiple nonprimitive representations, only one of which is bound to a class, the unbound representations will be added to the bound one and any bound slots in the unbound representations will be bound to the class being represented, adding instance variables are required. Finally, if multiple bound nonprimitive representations are merged, a dialogue box will appear asking which of the bound classes to merge to.

Unmerging a representation moves all the slots of the merged representation to the surrounding representation, preserving the binding of the slots.

Creating and Copying

In the previous gainSlider example we created representations and slots implicitly by declaring primitive representations, binding, and grouping them. We can also select any decorative graphical element and declare it to be a representation by executing the `Choose Class` command from the system menu. This causes the class browser to pop up with options to select an existing class or create a new one. As a shortcut, if an existing bound representation was included in the selection along with the decoration, the browser will show the bound class as being selected.

One can also use the `Choose Class` command to explicitly change the binding of a representation. When the class binding has been changed, all slots within the representation will become unbound and must be rebound to instance variables of the new class. To accelerate binding of slots to instance variables name completion can be used, so that the entire instance variable name need not be typed into a variable text box.

All of the copying and moving options—drag, alt-drag, control-drag—which are performed on representations of instances using the Pointer tool have analogous effects on slots and instance variables when performed using the Designer tool.

"Alt-dragging" on a representation with the Designer tool will make a copy of the representation and the instance variable it is bound to. These copies can be deposited in any representation. The destination representation is defined as the nonprimitive representation that physically surrounds the copied representation. A new slot will be created in this representation. The copied instance variable will be added to the class that the new slots binds to. If there is already an instance variable of the same name, then a new default name will be derived from the old name.

Control-dragging on a representation with the Designer tool will create a new copy of the representation. Depositing this copy in another representation of the same class will create a new slot that is bound to the same instance variable as the original. We have essentially made a reference copy of the instance variable. The copy will be rejected if it is made to a representation that is bound to a different class.

A representation can be replaced with another by dragging the new representation on top of the old using the Designer tool. If the source representation was already bound to a class and had slots that were bound to instance variables of that class, then this operation will try to match up instance variable names to preserve similar slot bindings. A representation can also be changed by selecting it an executing `Choose Representation` from the system menu. A representation browser will appear from which a new representation can be selected.

Choreographing the Interace

Animal provides a number of easy-to-use predefined mechanisms for helping to choreograph an interface. If these are not sufficient, then the event mechanisms described in the next section can be used to build custom user interface behavior. We have already discussed the default, null, and selected representations. If a custom one is defined it can be assigned to the slot by selecting null (selected) in the slot inspector and then dragging the desired representation into the slot using the Designer tool. If we want a particular full window class representation to pop up when we open (double-click) a representation in a slot, then we need only double-click on the slot and a browser will appear allowing us to select from the currently defined representations or to create a new one.

Garbage Collection

Counts of two kinds of references are maintained in Animal: references of slots to instance variables during interface design; and references to graphic representations to proxy network/multiprocessor instances. The latter are always maintained. Instance variables are deleted from a class when there are no longer any slots bound to them. Objects are deleted from the system when there are no longer any possible representations of them (whereby

a representation may exist as a possibility even though it may not be visible).

Multiprocessor methods may create instances of Animal classes. If the pointers to these instances are assigned to pointer instance variables of objects that have representations on the host, then the standard update mechanism will notify the host of their existence and enable reference counts for them. Some objects may not be accessible to the host—objects held in user-defined static variables, for example. No reference counts are kept for these and pointers which these objects maintain to objects which do have reference counts will not affect those reference counts. This can lead to dangling references, so extreme care should be taken by the programmer.

Since Animal instance graphs may be cyclic, the possibility exists of reference counts that never fall to zero. These objects represent "dead" space. They will not be archived with an application, so the memory loss involved is not permanent. If this becomes a serious problem in the Animal system, then a separate manually invoked mark-and-sweep garbage collector will be added with the possibility of manual invocation to avoid pauses at critical moments in the real-time computation.

Events and Actions

Events and actions provide a mechanism for customizing interface behavior and for controlling this behavior from application-specific methods running on the multiprocessor. This customization can override or extend the standard Animal behavior related to copying, moving, selecting, and opening objects.

Events

Events are a way to represent and communicate the fact a particular "something interesting" has happened. Events are coded as ASCII strings that can be intnerpreted as event names. Two events are considered the same if they have the same name.

When an event occurs as a result of a user or system action, the fact is signaled by carrying out an operation known as "raising" or "invoking" the event.

Events are divided into two categories: predefined system events and user-defined events. The predefined events are raised as a consequence of a user action such as typing, clicking, or dragging.

User-defined events have semantics that are defined entirely by the application designer. They are invoked from methods running on the multiprocessor or are specified as arguments to a predefined "action."

An event is raised in a particular context. When the context involves graphical interaction, the event will be raised inside a particular representation or slot. If the context is the internal computation performed by the multiprocessor or by the Animal process, the event will be raised inside an object instance.

Most—but not necessarily all—predefined events will be raised in the context of representation. User events that are invoked from methods on the multiprocessor are raised in the context of an object instance.

Actions

Actions are specific operations that Animal is able to perform. These operations are given names in order to allow the user to customize the Animal behavior—specifying an action to be performed in a particular situation, for example, when a particular event is raised in a given context.

The set of possible actions is predefined and is not user extensible. This is because actions are performed by the Animal process on the host and not in the multiprocessor and the application designer does not write code for the host. However, a set of "meta-actions" is provided that can invoke a multiprocessor method and, as a result, a particular user-defined action as defined by that method. Predefined actions can also invoke another NeXT or UNIX application if required.

Actions can have arguments that are constrained to be constant strings. Typical action arguments might be names of instance variables or names of particular representations. There are special sys-

tem-defined meta names that refer to context-dependent entities such as the currently selected representation or its surrounding representation.

Actions may also use implicit information about the events they are handling. In particular, some actions are representation (object)–specific, so they cannot be used for handling events raised in objects (representations).

Translation Tables and Event/Action Binding Rules

Event handling is specified by a mechanism similar to the one present in various UI toolkits (for example, the X intrinsic toolkit); it is the concept of translation tables.

A translation table maps an event to an action list, and so specifies how to handle a particular event. A translations table is specific to a particular context (i.e., a representation or instance). The action list can be empty, which means that nothing should be done. If there are several actions in an action list all of them will be performed when their event is raised in the context of this translation table.

Translation tables are not generally complete; that is, they do not need to define actions for all possible events. Depending on the context, an event may be looked up in a number of translation tables in order to find a suitable action.

As mentioned above, translation tables may be associated with representations or with object instances. Each object in a class shares the same translation table—they are considered to be properties associated with the class.

When an event is raised, the following rules are used to determine which action to perform. If an event is raised inside an object, the translation table associated with the class of that object is used. If the event is not found in that translation table, then no action will be performed. When an event is raised inside a representation, if an action for that event is specified in the translation table of the representation, that action will be invoked; if not, the event will be passed to the surrounding representation.

A representation thus can declare which events it is interested in, and pass the others up the reprersentation/slot hierarchy. For example, a representation of class Sound could map the *MouseDown* event to the *Play* action, while leaving the handling of the *AltMouseDown* event to the surrounding representation, which might, for example, delete the selected object.

Object translation tables are provided in order to handle events that cannot be related to a particular representation. These events might be generated by user methods coming from the multiprocessor and might, for example, open a new full window representation of a particular object.

System-Defined Actions

The predefined actions can be divided into three categories: user interface choreography and control; event routing; and interapplication communication.

In the choreography category there are actions to open or close representations, highlight or change representations and select or deselect representations.

In the event routing category there are actions to pass an event to another object or representation, to raise another event on the current object/representation or some different ones, and, in particular, to delegate the handling of an event to a multiprocessor object (i.e., to application-specific code). The special action `ignore` will do nothing, but will block the propagation of an event to surrounding representations.

In the interprocess communication class there are actions to invoke external applications using standard NeXT and UNIX-based interprocess communication facilities.

System-Defined Events

A number of events are automatically raised by Animal. These can be roughly divided into three subsets: mouse events, high-level user events, and object maintenance. Mouse events include *MouseUp, MouseDown,* and *MouseMoved.* High-level user events include *ObjectMoving, ObjectMoved,*

ObjectSelected, and *ObjectDoubleClick.* Object maintenance events include *ObjectDearchived, ObjectFreeing,* and *ObjectInitializing.*

System-Defined Translations

Part of the Animal system itself is implemented in the form of a set of predefined system translation tables that are inherited from basic objects and representations. Such translations can be overridden in order to customize the behavior of standard Animal operation.

Libraries and Application Archiving

An Animal application consists of a set of class definitions, with their methods and graphic representations, and an instance graph.

Application Directory Structure

An Animal application is archived using a directory hierarchy. Each application maps onto a directory with the same name as the application, and a ".animal" file name extension. The directory contains several subdirectories and the instance archive file. This file is an archive created with the NeXTStep object archival system.

The "src" subdirectory contains the source files for all the user-written or default methods, ".h" header files defining the class structures, and a "makefile." The ".h" files and the "makefile" are automatically generated by Animal. The "obj" directory contains the object files obtained by compiling the sources in the "src" directory. Other directories can be created when necessary to store user-level help files or other auxiliary files.

Archive File Format

The archive file consists essentially of four parts: the class table, a representation table of all the graphic class representations, the instance graph, and screen information. Each entry in the class table contains the number and type of instance variables, a list of references to representations that are bound to the class. Each entry in the reference table contains the graphical description of the representation. The instance graph is archived in such a way that all instance variables and pointer references are preserved. The screen information keeps track of the windows that were visible at archive time and their locations on the screen Actual windows are not saved, because they are rebuilt using information from the class and representation tables. This structure was chosen to permit an efficient implementation of reusable object libraries.

Reusable Libraries

In an environment oriented toward rapid prototyping it is important to support a high degree of reusability. Animal supports two levels of reusability: reusability of objects inside a project and reusability by easy sharing of objects between projects.

Reusability inside a project is provided by an environment that encourages specification by cloning in every phase of the prototyping cycle, from the interface to the multiprocessor methods, and by providing an inheritance mechanism in the application data model. Sharing objects between projects is supported through centrally maintained libraries for the different entitles in an Animal application.

The Animal architecture has three kinds of entitles that can be considered as modules to be shared between projects: the classes (including method sources), the representations, and the instance graphs or subgraphs.

The objects that are stored in a library are restored from the archive library when an application is read in. An object is never stored in an application and a library. The application always archives a reference to the object in the library. Each entry in a library has a unique name, which serves as a reference key. All applications using a library entity will always restore the most current version.

Basic Libraries

The Animal library system is a specialization of the "basic library" mechanism that creates libraries of arbitrary Objective C objects. A basic library contains objects, which are archived and restored with standard Objective C methods.

The objects are accessed through a library index—a name—which is stored in an index file and loaded during the application start-up. The use of the index file can be expanded in the future to provide a query/search mechanism on libraries.

An application has a user-defined library path associated with it, declaring all the libraries to which the application has access. Objects are loaded by name; this name is interpreted in a name space built by the union of the names defined by the set of libraries listed in the library search path. Loading the object from a library is much the same as restoring it from an application archive file. There are important differences in the archival behavior, however.

Archiving an object obtained from a library will not archive the object itself, but will archive a special delegate object that, at restore time, will automatically substitute itself with the original object reloaded from the library. An object can easily be saved to a library and, from then on, will be loaded from it.

Animal Libraries

As mentioned, the Animal library mechanism builds upon the basic NeXTStep library mechanism. Classes, representations, and graphs are not orthogonal in Animal; the graph nodes are typed, so class information must be stored with each graph. The class information may make references to the representations, so we need to store references to representations with classes. Moreover, class instantiation is realized by copying prototype objects, so we need to store instance graphs with classes.

For these reasons we define a special Animal library mechanism that contains three kinds of entities: classes, representations, and instance subgraphs. On loading a particular entity from the library all the related entities will be loaded. On saving to a library all related entities will be archived. From the user's point of view, an Animal library consists of a class library, with an associated instance graph library, and representation library.

Maintaining a Library

To add an object to a library, a representation of that object is selected and the `Add Object to Library` menu command is executed. A dialogue box appears with a file browser to choose or create a library. The representation, the class information, and the subgraph emanating from the selected instance will be archived to the library. The subgraph is defined by the archive properties of the pointer instance variables of the class. More than one instance subgraph can be stored for a particular class.

New representations and classes can be added to a representation by selecting them from a library. If there is more than one instance to choose from, the library browser will reflect this fact and allow the user to select a particular instance. Both reference copies and deep copies can be made from a library. A deep-copied instance will be stored with the application archive. A reference copy will always be loaded from the library. Library security is provided through the use of UNIX file permissions.

Restoring an Application with References to a Library

Class templates and representation information are loaded using the basic library mechanisms described above. Methods and instance subgraphs are somewhat different. Method source files are simply linked to the application source directory.

Restoring is a more elaborate process. Each instance graph that is stored in a library is analyzed in order to identify its entry and exit points: that is, nodes inside the subgraph which point to nodes outside the subgraph, and nodes inside the subgraph which are pointed to from nodes outside the

subgraph. The subgraph is stored with an entry/exit point table that provides symbolic indices for each subgraph entry and exit point.

In the application archive a "root reference object" is substituted for the subgraph. This object contains the values for all the exit points together with its symbol as found in the entry/exit point table. The reference object also stores the symbol associated with each subgraph entry point. When restoring archives, the subgraph from the library is substituted for the reference objects and all pointers are reestablished using their symbolic representations as stored in the entry/exit point symbol table.

Version Control

All classes, when stored in libraries, have a major and minor version number associated with them. A minor version change does not imply incompatibility with previous versions and is used for documentation purposes. Major version number changes may introduce incompatibility with previous versions. In this case a number of mechanisms are provided to support automatic conversion.

Representation changes are handled semiautomatically. However, a large change in dimension may require a manual user intervention. Class changes can affect the application in two ways: affecting the structure of an instance archived within the application; or changing the structure of the library instance subgraph (the interconnections between objects in the graph).

In the first case the archived instances are restored, mapping the old instance variables to the new instance variables following a simple heuristic—old names are mapped to new names according to type. Then a user-defined method is called for each instance, with the restored instance and a copy of the new subgraph from the library as arguments. This method must perform all additional modifications using the new instance subgraph as a template.

In the second case the library utility provides a way to associate graph entry/exit point table remapper with an instance. This permits the specification of a mapping between the entry and exit points of the old version and those of the new version, thus allowing for the automatic conversion of the application's instance graph to the new format.

The Programming Environment

As mentioned above, all user code is written for the multiprocessor, and none for the host Animal process. The main features of the programming environment are incremental method loading and an automated "make" utility. Programming an Animal application involves writing methods for classes, where the classes have been defined graphically through manipulation of their representations. A method can be declared by placing a method representation—a kind of text box—inside a representation of the class using the Method tool, or declared directly, by using the class inspector. In any event the class inspector will always provide a complete list of methods. The source code for methods can be edited by double-clicking the mouse on the representation of the method, either in a class representation or in the class inspector. This opens an editor on the source file. If a user-defined method has just been declared, then a "do nothing" template method will be automatically generated with the appropriate include files, and so forth.

Program Structure

Animal expects and automatically generates a number of files. In particular, it creates an ".h" header file for each Animal class, containing the class definition. This is generated and maintained by Animal. A ".c" source program file is created with a default "set" and "get" method for each instance variable of a class. This file is generated by Animal but can be written or modified by the programmer. The programmer version is guaranteed to survive subsequent automatic updates. Animal also creates a ".c" file for each user-defined method of the class. This file is generated by Animal (using a "do nothing" template) as soon as a method is declared. The

file must be modified by the programmer. All programmer modifications are guaranteed to be persistent. The user is, of course, free to add application-dependent header or source files.

Compilation and Loading

Animal automatically generates a makefile for the application. This makefile has targets for specifying the compilation and loading of a single file or of all the files based on the standard makefile dependencies.

Depending on the modifications that have taken place since the last compilation, Animal will either call the *make* utility with "all" as the target, using the make program's policy to keep dependencies between files, or will use make to compile the minimal set of files that need to be updated. This is to avoid unnecessary recompilation of all methods when a downward-compatible class modification (such as adding an instance variable) has been made.

Incremental loading is always mediated by the Animal process and requested by the make command through the use of a shell command. The makefile keeps track of which object files have been loaded and will never reload an unmodified object file.

Compiling and Loading from Outside Animal

If the programmer finds it convenient to do so, the make utility can be used from outside Animal to compile and load the application. In order to incrementally load the object files on the multiprocessor, the Animal process must be running. Methods can always be compiled outside of the run-time environment, however.

Debugging

Currently there is no support for debugging multiprocessor methods from inside Animal, but the "gdb" debugger can be used in parallel from another window or from within the GNU emacs text editor.

Current Status of Animal

At the time of this writing, Animal does not fully implement the functionality described in this article. The user-defined event handler has not been implemented yet. This means that a user must be content with the kind of command input and window management that is built into the system. Inheritance, while supported by the underlying data model, is not yet supported by the user interface. The copying and archival strategy described above is fully supported, but the user interface is still based on a cut/paste paradigm rather than on icon dragging.

The Region/Set/Ruler/Wall group of functions is implemented, but without type-checking for instances placed in a set; typed sets are supported in the internal data model, but a user interface for it has not yet been developed. An arbitrary origin for measurement cannot be placed on a graphic in the current version. One must select between use of one of the corners or the center as the origin. The class browser and other system-level browsing windows are not yet implemented. One must determine the structure of a class from its various representations.

Conclusion

Animal implements a general data and user interface model. The hope is that this model will be appropriate for many computer music and real-time control applications. The data model is a free network of objects. The user interface model allows the creation of multiple arbitrary representations of these objects, as well as providing mechanisms for the manipulation of the object network—cloning objects, creating references, copying subnetworks, and managing sets of objects. Animal also supports archiving of the object network as well as shared libraries of objects.

Animal has been used to build the sampling synthesizer example described in this article. A number of other applications are being considered, including a user interface and object management

system for "Mosaic," a physical modeling synthesis program developed at IRCAM, a graphic program for designing interactive musical compositions in terms of state transitions (e.g., State 1: wait for a note in the range C3-A4 and then start sequence five and move to State 2, or if no note is found within 7.25 sec move to State 3).

Animal is also being connected with the MAX graphical programming language (Puckette 1991b). Seen from the point of view of the Animal developer, this will allow the graphic definition of methods for Animal classes, making it possible to build applications without writing any C code. Seen as an enhancement to MAX, Animal will provide richer graphic visualizations as well as management of sets of objects.

References

Andrews, T., and C. Harris. 1987. "Combining Language and Database Advances in an Object-Oriented Development Environment." In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications.* New York: ACM Press.

Apple. 1987. *Hypercard Programming Guide.* Cupertino, California: Apple Computer Inc.

Bryce, D., and R. Hull. 1990. "SNAP. A Graphics-Based Schema Manager." In S. B. Zdonik and D. Maier, eds. *Readings in Object Oriented Data Base Systems.* San Mateo, California: Morgan Kaufman.

Eckel, G. 1990. "A Signal Editor for the IRCAM Musical Workstation." In *Proceedings of the International Computer Music Conference.* San Francisco: Computer Music Association.

Goldman, J., et al. 1990. "ISIS: Interface for a Semantic Information System." In S. B. Zdonik and D. Maier, eds. *Readings in Object Oriented Data Base Systems.* San Mateo, California: Morgan Kaufman.

Lindemann, E., et al. 1991. "The Architecture of the IRCAM Musical Workstation." *Computer Music Journal* (this issue).

Maier, D., and P. Nordquist. 1990. "Displaying Database Objects." In S. B. Zdonik and D. Maier, eds. *Readings in Object Oriented Data Base Systems.* San Mateo, California: Morgan Kaufman.

NeXT. 1989. *Version 1.0 System Reference Manual: Concepts.* Redwood City, California: NeXT Inc.

Puckette, M. 1991a. "FTS: A Real-Time Monitor for Multiprocessor Music Synthesis." *Computer Music Journal* (this issue).

Puckette, M. 1991b. "Combining Event and Signal Processing in the MAX Graphical Programming Environment." *Computer Music Journal* (this issue).

Ribardiere, L. 1987. *4th Dimension User's Guide.* Santa Clara, California: Acius, Inc.

Viara, E. 1991. "CPOS: A Real-Time Operating System for the IRCAM Musical Workstation." *Computer Music Journal* (this issue).

Visual Edge. 1989. *UIMX User Interface Management System—Technical Description.* Visual Edge Software Ltd.