ANIMAL - A RAPID PROTOTYPING ENVIRONMENT FOR COMPUTER MUSIC SYSTEMS

Eric Lindemann
IRCAM
31, rue Saint-Merri
F-75004 Paris, France
elind@ircam.ircam.fr

ABSTRACT: An overview of the ANIMAL rapid prototyping system is presented. ANIMAL is designed to support research in real-time musical signal-processing and event processing. ANIMAL allows users to draw pictures which define complex data objects. These pictures can function as animated real-time controls. Dynamic collections of these graphical objects can be created, making it relatively easy to design graphical event list editors, synthesizer interfaces and, in general, any interface involving visualization and manipulation of quantitative data applied to collections of discrete objects.

1. ANIMAL - A NEXT GENERATION UIMS

ANIMAL (ANIMAted Language) combines elements of a graphic user interface management system (GUIMS) and a graphic programming language (GPL). It is intended as one of the main tools for user interface development in the IRCAM Musical Workstation (MW) environment.

User interface programming is often the most time consuming aspect of program development. Several direct manipulation GUIs have been developed to simplify this process (NeXT Inc. [1], Apple Computer [2], Visual Edge Software [3]). These systems tend to emphasize the interconnection of predefined interface objects (sliders, text fields, buttons, etc.) in static configurations. Objects are not intended to be added or deleted from the interface once it is running. ANIMAL attempts to go further by providing several important capabilities.

In ANIMAL, the user can create custom graphic interface objects by drawing them directly on the screen and then specifying, by direct manipulation, which parts of the drawing (the angle of a line, the size or color of a circle, the x-y position of an entire complex object) are to be adjustable in the functioning interface. The user specifies which parts of the drawing are to be controllable by mousing and which parts (often the same) are to be accessible as variables in an underlying program. Complex objects are created from simple graphic primitives (polygon, line, etc.) by grouping techniques similar to those used in standard draw programs. In defining a complex graphic interface object the user is defining a class structure which can be referred to when methods are defined for the class. The complex object serves as both visualization and multidimensional controller for the fields of the object. As a simple example, a musical note may be represented by an object with adjustable x_position (onset time), y_position (pitch), width (duration), color (intensity), shape (envelope). The user provides names for these fields by typing directly on the drawing. These variable names disappear when the interface is activated.

A functioning ANIMAL interface can support the manipulation of dynamic collections of custom interface objects. This is done by creating palettes of objects and associating these palettes with windows. When the interface is activated, instances of the object prototypes specified in the user defined palette can be placed in the associated window. The objects can be selected, moved, cut, copied, pasted, etc. As an example, a musical event list editor can be created by arranging collections of note objects in relation to one another in a scrollable window.

ICMC GLASGOW 1990 PROCEEDINGS
As the position and size of the fields of a graphic object are adjusted, the underlying variables of the associated data structure take on different values. These values are determined by the position of the "graphic variables" with respect to special graphic objects called "rulers". Rulers can be placed anywhere in a window. Rulers can be open-ended or flinate, one or two dimensional. Flinate rulers restrict the extent of motion of the graphics they measure.

An ANIMAL application takes the form of a tree of windows. Each window contains one or more complex graphic objects. Double clicking on a graphic opens its "child" window. The combination of a "parent" graphic and its child window defines an "animal". An animal consists of a number of variables corresponding to the position and size of the moving parts of the parent graphic and a number of pointers to other animals corresponding to the graphics contained in the child window. Additional simple numerical fields corresponding to float or int may also be placed in the child window. The variables associated with the position and size of the parent graphic also appear as numerical fields in the child window. This is convenient since when both windows are visible (the window containing the parent graphic and its child window) precise numerical readouts of the size and position of the parent graphic can be had by looking at the child window.

A C++ class definition in the form of a "h file" is generated by directly parsing the child window of an animal. The variables which are associated with the parent graphic are declared as public. All others are private. Methods can be associated with this class by typing the method names in the child window. Double clicking on the method name opens a text editor where the body of the method can be defined. The method can refer to any variables associated with the parent or child graphic and any animal objects which appear in the child window. Assigning a graphic variable within a method will cause the corresponding graphic to move appropriately. This provides the basis for animation in the environment.

ANIMAL has two modes of operation: edit mode, in which an interface is defined and run mode, in which it is active. Method names and most variable names disappear when the interface enters run mode so that clean uncluttered interface can be provided.

ANIMAL is designed to run in the real-time multiprocessing environment of the IMW. This environment is based on a host computer with an attached high-speed multiprocessor. The functional body of all user defined animal objects resides in the local memory of the IMW's multiprocessor calculation engine. Only the graphic image of these objects is maintained on the host computer. All user defined animal methods run on the calculation engine. No user defined code runs on the host. This simplifies the development loop and provides for increased portability to other host platforms. A real-time operating system, COS/FTS, has been developed to run on the IMW calculation engine. This operating system provides for incremental loading of ANIMAL classes onto the calculation engine and incremental linking of new or redefined methods into the running system as well as the update of existing instances when classes are modified.

2. AN EXAMPLE ANIMAL INTERFACE

This section describes the user interface for a sampling synthesizer designed using ANIMAL. The top level (Figure 1) is dominated by a "keymap" subwindow in which a collection of "sample patches" are mapped onto a two dimensional pitch-velocity ruler. Each sample patch encapsulates a single sampled sound (a recording of a gong, chime, etc.) and a number of controls associated with that sound. The idea is that when a note is played on a MIDI keyboard controller whose sample patches whose mapping covers the note's pitch and velocity will be activated.

Above the window is a user defined palette holding the prototype sample patch. The palette gives the user the right to instantiate new sample patches in the subwindow. Figure 3 shows the design of the sample patch parent graphic prototype. The sample patch parent graphic consists of two ICIMC GLASGOW 1991 PROCEEDINGS
neued rectangles. The inner one defines a pitch/velocity plateau and the outer the pitch/velocity extent of the patch. The amplitude of the activated sample patch is linear with respect to MIDI velocity across the plateau region. From the edge of the plateau to the edge of the extent the amplitude falls to zero. This allows overlapping of sample patches to achieve amplitude crossfades between patches. In addition to the "public" variables associated with the sample patch parent graphic there are two public methods playPatch and stopPatch. The significance of these will be discussed in the next section.

Double clicking on a sample patch reveals its internal structure (Figure 2) consisting of sound file reference, amplitude envelope editor, 2 pole filter editor with frequency response, a number of numerical fields corresponding to the parent graphic variables, and various simple controls. Double clicking on the sound file reference will message the IMW Signal Editor(s) which can display time and frequency domain representations of the sampled sound and provides a rich set of signal processing tools.

The DSP POOL object which is shown in Figure 2 is intended to support one of the special real-time features which will be included with the IMW in the future. The user will be able to preallocate a number of identical instances of a signal processing object (in this case a sampler voice). When a sample patch is activated it acquires a sample voice from the DSP POOL. If none are available a user defined "stealing" algorithm will be invoked to free one of the voices. The DSP POOL facility will be one of the key elements involved in allocating the calculation resources of the IMW.

3. GRAPHICAL PROGRAMMING

As described, ANIMAL provides a means of graphic class definition but provides no means for graphic method definition. In the future dataflow style graphic method definition will be provided. The intention is to allow methods to be declared public so that their names appear next to a parent graphic when it is selected in edit mode. The playPatch and stopPatch methods of the prototype samplePatch (Fig 3) are examples. The user will then be able to define graphic methods by connecting extended methods of subanimali in a child window in a dataflow style. An example is shown in Figure 4 for the samplePatch child window. The traces associated with these graphic methods will disappear in run mode. For methods with multiple arguments a trace merging graphical syntax will be used. Order of argument evaluation will be determined by the order of merging of traces with the straight trace corresponding to the first argument. For some interfaces such as signal processing graph editors, it is desirable for dataflow traces to remain visible in run mode. A node option will be provided. When a node is associated with a variable or the argument of a method, then traces hooked to that node will remain visible in run mode.


ICMC GLASGOW 1990 PROCEEDINGS