View metadata, citation and similar papers at core.ac.uk brought to you by CORE
provided by Scholarly Materials And Research @ Georgia Tech

Animating User Interfaces Using Animation Servers

Krishna Bharat Piyawadee “Noi” Sukaviriya

Graphics, Visualization and Usability Center
College of Computing, Atlantic Drive
Georgia Institute of Technology
Atlanta, GA 30332
{kb, noi}@cc.gatech.edu

ABSTRACT
Our approach to user interface animation involves simulating the interaction of a user with the interface by synthetically generating the input events that drive the session. The interaction is made explicit by displaying the behavior of input devices audio-visually. Such “animation” is both educational and functional, and has the potential to become a powerful new medium in the graphical user interface domain. We describe the construction of a general purpose tool for animating user interfaces - the *animation server.* Clients drive the server with textual scripts that describe the interaction. These may contain constructs for obtaining application context information at runtime and synchronizing with other media servers. We present a few potential applications for animation servers, including a groupware package for loosely coupled collaboration.

KEYWORDS: User interface animation, animation server, extensible interfaces, multimedia, context-sensitivity, application state, CSCW

INTRODUCTION
In a world where transmission and storage overheads far exceed computational costs, animation could well be the medium of choice for tutorial, help and session-playback systems of the future. *Why draw a picture when there is already one on the screen*? User interfaces of tomorrow will need to be adaptive, flexible and programmable. Animation provides a means of building functionality at a higher level than that provided by the basic interface. This allows systems to be made open and extensible. For instance, collaborating applications can achieve interoperability by using animation to bridge the gap between what is available and what is required by the protocol. On-line help and tutorial systems that use video and graphics are expensive to author and store, and often fail to exploit the presence of the interface on the screen. Shneiderman [15] notes that direct manipulation interfaces are attractive because they are easy to demonstrate and often self-explanatory. Hence the recent interest in animation, and the efforts to evaluate the generality and effectiveness of animated help [10,18].

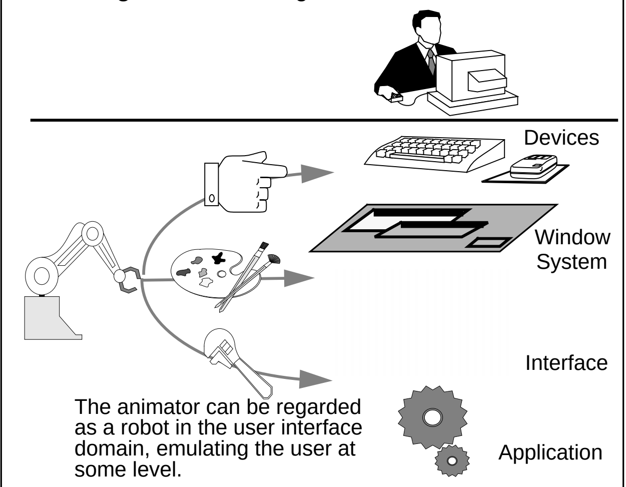
Animated demonstration systems such as MacroMind Director [6] are becoming popular. *Our notion of animation is more restricted.* We define user interface animation as the process of emulating the interaction of a user with the interface. The interaction should be real, in the sense that it should engage the actual application. Such systems are powerful because of their ability to invoke actions, and expressive because they can be used to demonstrate interaction techniques. The effect of the presentation may be enhanced by displaying the behavior of input devices audio-visually.

In this paper we present the architecture of a general purpose tool for user interface animation - the *animation server.* Although designed for the X environment [14] the tool has applicability to other environments as well.

EMULATING THE USER
The emulation of the user could occur at various levels:

1. Application Level: The emulator could invoke application actions directly. Many architectures allow one application to drive another by some internal mechanism such as Apple Events [21]. Since interaction is not being shown, this does not conform to our definition of animation.
2. Interface Level: The emulator could trigger interface actions and make widgets provide appropriate feedback. The emulator may be integrated with the UIMS (user interface management system) that implements the interface.
3. Window System: Input events could be delivered to the interface via the window system. The emulator could be integrated with the window system.

Figure 1. Emulating the user at various levels


Devices
Window System
Interface
Application
The animator can be regarded as a robot in the user interface domain, emulating the user at some level.

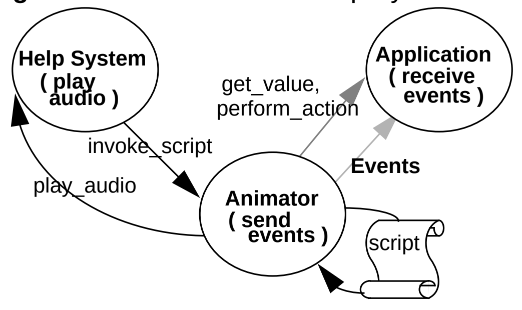
4. Input Device: Input events could be delivered to the window system as if from the device driver. This is tricky because the mouse is being treated as a relative device.

In the Macintosh and PC environments, the UIMS is integrated with the window system. Emulation at the level of the interface provides robustness and control over the presentation. In the X environment, each application has its own UIMS, and it is necessary to operate at the level of the window system to be as general as possible. Mouse and keyboard events are delivered to the X Server which relays them to appropriate clients based on window placement, notification requests, notification masks and grabs. Thus we may create presentations that involve multiple applications and the Window Manager. The disadvantage is that the events being generated are low level, and so is the programmer's interface.

RELATED WORK

Commercial record-and-playback macro packages like Tempo II [11] and Microsoft's Recorder [8] work at the level of window system while most programming by demonstration systems like Metamouse [7], Eager [2] and Chimera [5] work at the interface level. Triggers [13] is a system-wide macro facility on the Macintosh that generates keyboard and mouse events and uses pixel-level pattern matching to obtain context. Cartoonist [17] is a context-sensitive animated help system integrated with the UIMS. Application actions are first mapped to actions in the current context, and then to low-level interface events which are sent to the interface controller. In Metamouse and Eager, an agent within the application infers procedures from interface actions and performs actions autonomously, when a familiar sequence is detected. Kosbie and Myers [4] suggest the use of event hierarchies to relate high-level semantic events with the low-level events that drive the session.

Figure 2. Audio-Animation Help System


Help System (play audio)
Application (receive events)
get_value, perform_action
invoke_script
Events
play_audio
Animator (send events)
script

IMPLEMENTATION

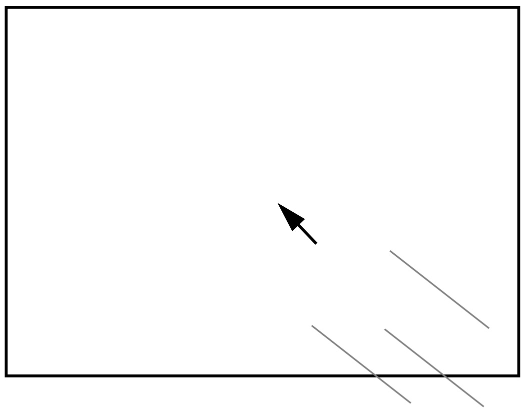
Background

The animation server described in this paper evolved from a tool called the *animator*, implemented in an experiment to compare various forms of multimedia help [18]. Fig. 2 shows the structure of the *Audio-Animation Help System* used in the experiment. Three processes were involved. The initiator of the help sequence which was also responsible for audio playback (hence called the *audio server*), the application being acted upon, and the animator. The help system invoked animation by sending the animator an animation script describing the actions to be performed (e.g. mouse moves, key strokes, mouse clicks). When the location of a widget or a window was needed, or an application level action had to be performed, the animator would interrupt the application using an asynchronous message. The application would respond by returning a value or performing the appropriate action. Synchronous messages travelled between the audio server and the animator to synchronize the animation with audio playback. The audio server played a sequence of audio pieces - each piece being synchronized with a step in the animation. The correlation was specified within the script. The animator showed the manipulation of input devices by the user by moving glyphs across the screen and providing audio feedback for mouse clicks and keystrokes (as shown in Fig. 3).

Architecture

The animation server inherits most of the features of the animator described in the previous section. The purpose of the generalization was to make it a shared resource,

Figure 3. Animation using events, moving icons and sounds effects



encapsulating sufficient functionality to be useful in different scenarios.

The server has a hierarchical, open architecture, with two parts - the *A-shell* and the *kernel*. All device dependent features are absorbed in the kernel. It has a low level command line interface and works in absolute coordinates. The kernel can be thought of as a pseudo device driver for the mouse and the keyboard. There is one kernel per site. Most applications would prefer to interact with the *A-shell*, which is at a higher-level, and has support for symbolic programming, conversion between coordinate frames and synchronization with *media servers* participating in the presentation. Each application has its own A-shell. The dynamic combination of an A-shell with the kernel is the animation server.
The application sends commands or scripts to the A-shell, which translates them to the kernel command language and drives the kernel.

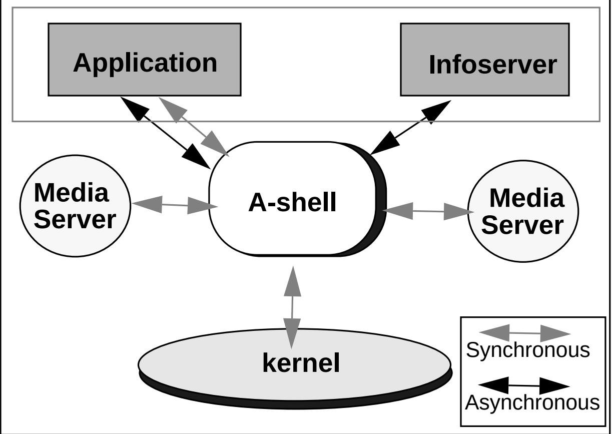
Applications that desire animation often have a lot of contextual information to communicate to the animation server - the location of widgets, selections and even the very composition of the sequence may be unknown till runtime. The acquisition of context information depends on the way the UIMS and the application represent state information and the data exchange mechanisms they support. Since we are building a general purpose tool, it is better to entrust this responsibility to an agent created by the application programmer. We call this process the *infoserver* and it is typically the application itself. When requested by the A-shell, the infoserver gathers data from the window system, the UIMS and the application, and returns it an executable form. The infoserver may in turn, leverage off a high-level data exchange mechanism like Apple PPC [21] or Microsoft DDE [12]. We merely define the communication protocol between the A-shell and the infoserver.

Fig. 4 shows the processes involved. The A-shell is typically spawned by the application, which also appoints any media-servers and infoservers required in the presentation. Every process maintains an input queue for interprocess communication. All communication is in the form of typed messages supported by UNIX[1] System V message queues [1]. The queues allow blocking, non-blocking and selective access to messages.

The programmer's interface to the animation server is a library of routines to drive the A-shell and register callbacks. The application may the drive the A-shell synchronously or asynchronously, but the A-shell always drives the kernel synchronously. Multiple A-shells may share a kernel by yielding control after executing a logically continuous segment of their animation - called a *session*. The programmer may improve concurrency, by making sessions fine-grained.

1. UNIX is a trademark of AT&T.

Figure 4. The Processes Involved



The Kernel

The implementation of the kernel is machine specific. Events for the keyboard and the mouse need to be synthetically manufactured. The kernel command set is low-level, yet comprehensive. The following commands are supported:

1. Appear at *x,y*
2. Disappear
3. Goto *x,y* in n steps
4. Click *b* button after moving to *x,y* in *n* steps
5. Doubleclick *b* button after moving to *x,y* in *n* steps
6. DragAndDrop to *x,y* in *n* steps using *b* button
7. Pause for *s* secs
8. Pause for *m* msecs
9. Type "message"
10. Get Current Position

Where *x, y* are integer coordinates; *n* represents the number of frames in the animation; *b* is either { "left", "middle", "right"} or {"left", "right"} depending on the number of buttons on the mouse *s* and *m* are integer values representing time elapsed, in seconds and milliseconds and *message* may contain codes for special keys, such as <DEL>, <BSPC> and <RETN>. Upper case characters are decomposed into appropriate key-combinations. The kernel is not case-sensitive.

A few low-level commands may also be defined :

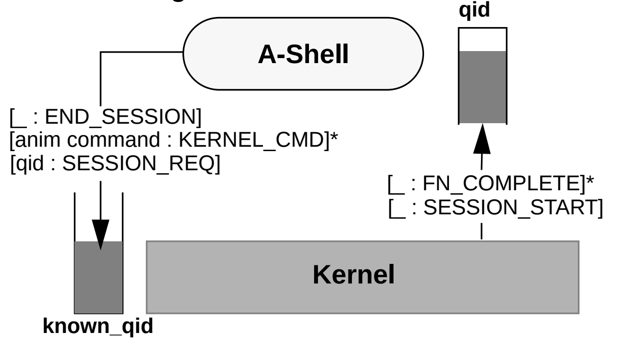
11. Press Button *b*
12. Release Button *b*
13. Press Key *k*

14. Release Key *k*

The command DragAndDrop represents the compound action “Press Button b; Goto x,y in n steps; Release Button b.”

The only command that returns a value is Get Current Position which returns the current location of the cursor. This command is needed by the shell to convert from relative coordinates to absolute coordinates. The Pause command is the only command that involves a reference to time. Other commands merely refer to the granularity of the frames in an animation step. This is not a real-time system. There is no capability to schedule a certain action at a specific instant in time; nor is it possible to estimate how long a certain animation step will take. Nonetheless, the pause command is needed, to prevent the next action from being invoked before the current action takes effect.

Figure 5. Kernel Sessions



Kernel/Shell Session Protocol

A-shells driving the kernel synchronously as shown in the figure. The current implementation permits only one shell to driving the kernel at a given time. The structure of the kernel command execution loop is as follows[1]:

```
Kernel_Session(QueueT inQ)
{ QueueT outQ;
   [outQ:_]=blocking_readQ(inQ, SESSION_REQ);
   writeQ(outQ, [ _ : SESSION_START]);
   [command:code]=blocking_readQ(inQ,
                              KERNEL_CMD | END_SESSION);
   while(code != END_SESSION)
     { switch (command)
          { case “Appear at x,y” :-...
             case “Click ...”:- ...
          // service command appropriately
          }
     writeQ(outQ, [return_value:FN_COMPLETE]);
     [command:code]=blocking_readQ(inQ,
                              KERNEL_CMD | END_SESSION);
 }}
```

1. All interprocess communication is by messages of the form [<command>:<code>]. ‘_’ denotes “Don’t Care.” The ‘|’ symbol is used to “OR” message types in the read command.

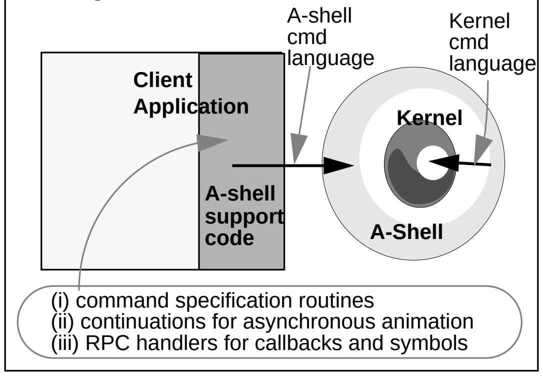
At every site there is exactly one kernel, although multiple A-shell clients may be present. The queue-id of the kernel (inQ in the example) is publicly known. Animation sessions are set up with clients on a first-requested-first-served basis. outQ is the input-queue of the client that is being serviced at any given time.

UNIX message queues allow messages to be typed. This permits the kernel to distinguish between session requests and animation commands. An A-shell initiates a session by placing a SESSION_REQ message on the inQ, along with its own input queue id, and waits for a SESSION_START message. Thereafter it sends commands to the kernel, one at a time, in a synchronous fashion. After each command, the shell blocks on its input queue, waiting for a FN_COMPLETE message. This is necessary because the next command may not be computable until the current one has been executed. At the end of a session, the current client releases the kernel with a END__SESSION message. The kernel is then free to service the next client on the queue.

The A-Shell

The A-shell is a high level, context-sensitive interface to the animation kernel. It runs as a separate process and communicates with the application via message queues, as it does with the kernel. Thus, the A-shell is a layer between the application and the kernel. The client application views the shell and the kernel as a single entity, the *animation server*, although the link gets broken at the end of every animation session, and needs to be re-established when needed again.

Figure 6. Interface to A-Shell



A-shell cmd language
Kernel cmd language
Client Application
Kernel
A-shell support code
A-Shell
(i) command specification routines
(ii) continuations for asynchronous animation
(iii) RPC handlers for callbacks and symbols

The A-shell needs to be a separate process because, in the callback style of programming, the process that invokes the animation cannot block while it is in progress if it is the recipient of input events. The client application typically operates in asynchronous mode. It issues an animation request to the server and returns from the callback. The server signals the client when the

Preprocessor Commands

Command	Usage	Purpose
#invoke	#invoke “filename”	Transfers control to a script (batch mode)
#end	#end	Returns control to caller (A-shell or another script)
#define	#define macro-name macro-expansion	Defines macro, e.g. #define OK_BUTTON 100,50
#undefine	#undefine macro-name	Pops last definition of macro-name from the stack
#infoserver	#infoserver process-id queue-id	Nominates a process as the infoserver
#mediaserver	#mediaserver server-name queue-id	Registers a user defined media server with a given name
#play	#play server1(“msg1”), server2(“...”),...	Invokes server1 with msg1 and server2 with msg2 etc.

operation is completed. Often the command will be a script to be executed by the server in batch mode. The A-shell accepts a superset of the kernel command language. The following extensions are provided:

*Preprocessor Commands***.** These are A-shell directives, beginning with a ‘#’ and are listed in the table. They do not get propagated to the kernel. Macros may be used to make the script more readable or to pass parameters to scripts.

```
e.g. #define IconToMove fileIcon
     #define Destination 100,100
     #invoke “MoveIcon”
```

When executing the play command, the A-shell waits until every server referenced in the command has indicated its readiness to accept input before sending requests.

Addition of Coordinates. Conversion from one coordinate frame to another is done by prefix addition expressions. e.g. +(100,500, 20,60) would evaluate to 120, 560. These may be nested.

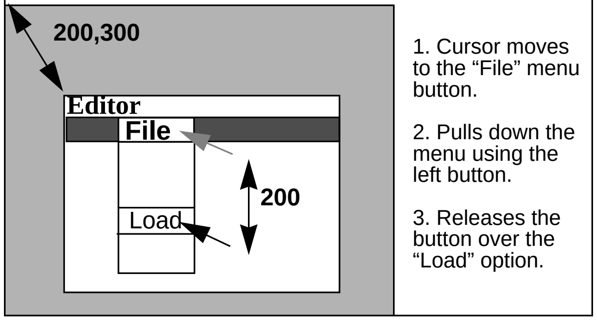
Comments and white-space. Comments start with a ‘;’

*Macro Invocation, Symbols and Callbacks***.** When an identifier preceded by an ‘@’ symbol is encountered, A-shell first checks in the list of known macro-names for a string substitution. Macros may be nested but never recursively.

The following example illustrates the usage of macros:

```
; Macro Definitions
#define editorWindow 200,300
#define pullDown DragAndDrop
#define editorFileButton +(100, 50, @editorWindow)
#define fileClick +(20,20, @editorFileButton)
#define loadFileChoice +(0,200, @fileClick)
...
; Macro invocation.
; Pull down the File menu and select the Load option
Goto @fileClick in 10 step
@pullDown to @loadFileChoice in 30 steps using
Left button.
```

Figure 7. Script for Menu Selection


200,300
Editor
File
Load
200
1. Cursor moves to the “File” menu button.
2. Pulls down the menu using the left button.
3. Releases the button over the “Load” option.

Unfortunately all coordinates are statically defined. This can be avoided using symbols.

When a symbol of the form @symbol-name is encountered, it is sent to the infoserver which provides the expansion for the symbol - typically a value derived from the state of the system (e.g. a coordinate pair representing a location that cannot be computed till runtime). We could simulate a control statement by having the symbol evaluate to a sequence of commands or the name of a script.

Callbacks are of the form @callback-name(arg-string) and are distinguished from symbols by the parentheses enclosing the argument string. Unlike symbols, callbacks are always executed by the A-shell support code in the parent application, which unpacks the arguments and invokes a function whose pointer and signature

have been registered earlier by the application.
In the following example OffsetToOK is a macro representing a static offset, PrintPopup is a symbol that evaluates to the northwest corner of the "Print" window, and loadFile is a callback registered by the application with the A-shell support code.

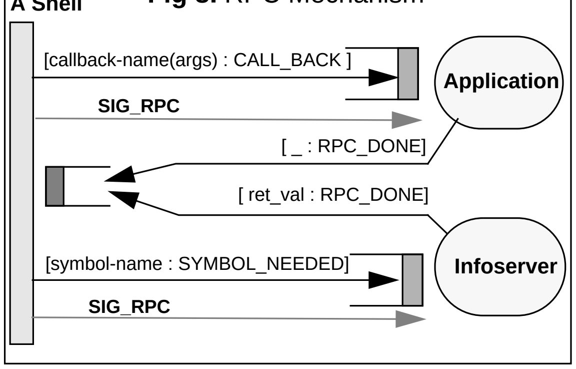
```
#define OffsetToOK 400,50
...
@loadFile("demo1.doc")
; loadFile is an application callback
...
Goto +(@PrintPopup, @OffsetToOK) in 15 steps
; Infoserver provides the current location of
; the window designated PrintPopup
```

@CurrentPosition is a special symbol that causes the kernel to be contacted instead of the infoserver. Relative positions may be specified by expressions of the form +(@CurrentPosition, x_increment, y_increment).

Callbacks are used to invoke actions internally, as opposed to invoking them externally via input events. This gives the animation designer some flexibility in composing the simulation. Typically they are used to perform actions that are either impossible or tedious to perform using normal animation. For example, help systems may use a callback to save the state of the application before performing some actions, and another callback to revert to the saved state.

Symbols and callbacks make it possible for the animation script to access contextual information and invoke application actions.The RPC mechanism is illustrated in the following figure.

Fig 8. RPC Mechanism



When a symbol is encountered:

1. A message, [symbol-name : SYMBOL_NEEDED], is placed on the input queue of the infoserver.
2. A signal, SIG_RPC, is sent to the infoserver.
3. The A-shell blocks on its input queue waiting for a return message of the form [ret_val:RPC_DONE].

The string ret_val is used as the expansion for the symbol. The protocol for callbacks is similar. A message of the form [callback(args):CALL_BACK] is sent to the application. The return value is ignored and the instruction is replaced with a null-string.

In both cases a signal handler is responsible for reading the input queue and carrying out the appropriate action. Note that in the synchronous case, the application may be waiting for an acknowledgment from the server when the RPC request message is received. Since messages are typed there is no collision.

PROGRAMMING THE A-SHELL

The user interacts with the A-shell through a library of routines linked with the application, called the *A-shell support code*. Since the A-shell is sometimes driven asynchronously, it needs to be a separate process and is forked when the application starts up. The functions, *Connect* and *Disconnect*, manage the connectivity of the A-shell to the kernel.

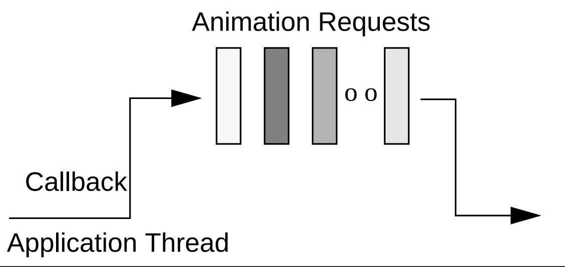
The programmer needs to create the infoserver and all media-servers that participate in the session. The same support code is used in all cases. All processes maintain queues for communication with the A-shell.

Issuing Commands to the A-shell

In most graphical applications there is a single thread of control and all actions, including the action that brings about animation, are invoked within callbacks. In this style of programming, the application will be unable to service events until it exits the animation callback. If a '#invoke' command was sent, then the application will block until the script has been fully executed.If the application needs to return to its event processing loop, a non-blocking approach may be preferred. To support both styles of invocation we have two modes:

(i) Synchronous Mode

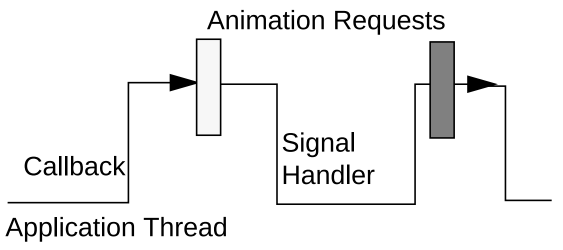
Figure 9. Synchronous Communication



In this protocol the application blocks on its input queue until it receives an acknowledgment from the shell indicating that the action has been completed. Then, it issues the next animation request. This is not very useful because the application may need to process input events while the animation is in progress. This facility helps when the process driving the A-shell is not receiving the events or when no interface actions are being invoked.

ii) Asynchronous Mode

Figure 10. Asynchronous Communication



In this protocol, the application issues a request and then returns from the animation callback immediately. When the request has been serviced, the application is interrupted by a signal, and control is transferred to a signal handler, called a *continuation*, which issues the next request in the sequence. The default continuation does nothing. The programmer may choose to define a continuation if the animated script is to be generated incrementally.

The default behavior is synchronous. Asynchronous animation commands are terminated with a '&' sign.
e.g. DragAndDrop to +(@CurrentPosition, 200,50) in 40 steps &

Media Servers
In this section we describe the architecture of media-servers that the programmer might want to construct. They may be registered with the A-shell at the start of the session using the #mediaserver command. Media-servers process requests in a first-come-first-serviced fashion. Each media-server has a single queue for both input and output messages, and may cater to multiple clients at a given time. The media-server is not aware of the identity of its clients. Any process that obtains the queue-id of a media-server can write to it, and the animation server is just another client.

The media-server loop is as follows:

```
while (1)
    { writeQ( Q, [ _ : CODE_FREE ]);
      [ media_request : _ ] =
                  blocking_readQ(Q,CODE_REQUEST);
      execute(media_request);
    }
```

The client which reads a CODE_FREE message gets to issue the next request to the media-server. When the animation server needs to issue a play request to a set of media-servers, it first obtains the right to issue requests by reading a CODE_FREE message from each server's queue. Then it starts them off together and proceeds with the script. If all clients obtain access to media-servers in a standard order (e.g. the alphabetical order of server names), deadlocks can be avoided.

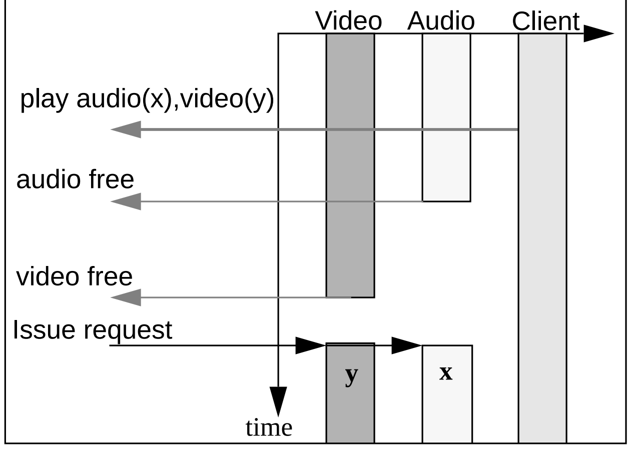
A typical scenario is shown in Fig. 11. The client issues a "play audio(x), video(y)" request to A-shell. The shell needs to wait awhile for the audio-server to become free. On reading a CODE_FREE token from the audio-server's queue the shell waits for a similar token to appear on the video-server's queue. Having read tokens from both queues the shell issues individual requests and proceeds with the animation.This ensures that all media streams are synchronized. For simplicity, propagation delays have not been shown in the figure.

DISCUSSIONS
The animation server can function as both a teacher and a robot in the graphical workspace. Applications may range from purely educational to purely functional.

Knowledge Levels
One of the strengths of our design is that the entity that generates the animation script need not be the application being animated. To create scripts effectively, however, this process must be privy to some information about the state of the application. The level of detail may vary:

i*) No State Information:* Blind access works when an excellent model of the application is available (amounting to a replicated state), or a canned presentation will suffice. This will do in cases where the user does not interact with the interface directly, and uses audio input or types commands

Figure 11. Synchronizing Media Servers



on a terminal to drive the session.

ii) Information about the state of windows and widgets: Interface related information may be obtained from the window system. Often widgets are located at a fixed offset from the window-origin, or at a displacement computable from the geometry of the window. Application level state information has to be inferred from the interaction history.

iii) Application Level Information: The script-generator could obtain state information by sending asynchronous queries to the application. To avoid having to poll, it may request notification when certain parts of the state change. If the script-generator resides within the application itself it will have total access to the application state. If it is a separate process, shared memory or disk storage can be used to share state information, which tends to be expensive and slow. Embedding a script-generator in the application may be intrusive and impractical. Query-based access seems a realistic approach, and provides a generic interface to all clients requiring state information.

Application Support

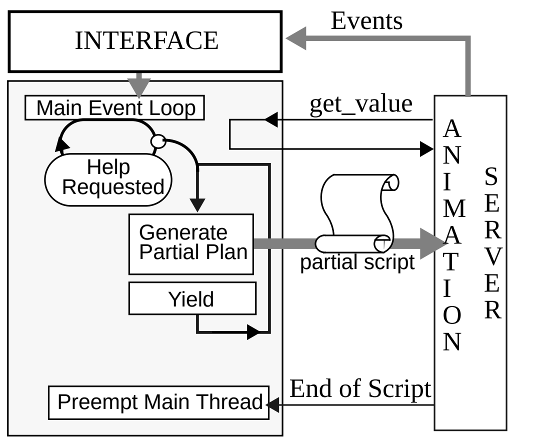
Applications need to be vested with the capability of exporting state information to entities wishing to interact with them. Programs of tomorrow may require "social skills" to survive in a world of collaborating software. Structured design of the application, separation of the interface from the abstraction, and a declarative framework for maintaining state information assist in this enterprise. We hope to integrate animated help with UIDE (User Interface Design Environment [20]), which offers many of these features. In UIDE, pre- and post-conditions are used to describe the partial semantics of application actions. Pre-conditions are used to determine the validity of actions, and post-conditions (or rather post-assignments) register the change in state induced by the action on a global blackboard. Together they provide sequencing control. There is a decomposition of objects into application objects and interface objects, and actions into interaction techniques, interface actions and application actions. All data and control information is available in a declarative form, which would allow an external entity to perceive, analyze and reason with the state of the application. This provides the declarative framework needed to undertake context-sensitive planning and script generation.

APPLICATIONS

Context Sensitive Animated Help

Context sensitive help systems are required to analyze the problem context (the current state of the application) and generate a plan of action that will either carry out the required task or serve as a relevant example [17,19].

Figure 12. Incremental Generation of Help



The above architecture is due to Spaans [16]. Rather than undertake "hypothetical world" planning, the planner is organized so that it generates the plan in chunks. Each chunk is converted into an animation script and executed before the next chunk is generated. This enables the planner to make use of the updated application context in generating the plans that follow. For simplicity the planner and the user interface are implemented as threads (provided by CThreads [9], a non-preemptive threads package). On invocation of animated help, a new thread is forked. This thread is responsible for computing partial plans and dispatching them to the animation server for asynchronous execution. After sending the script to the animation server, the help-thread yields control to the main event-handler of the user interface, so that it may receive input events. When the script has finished executing, the server interrupts the application (as required by the asynchronous protocol), and the help-thread regains control.

Here are some *potential* applications for the animation server:

1. Tutorial / Session Playback Systems

These have a lot in common with Animated Help systems. However, they do not need to analyze the application context and have the advantage of starting with a known or default system state. Tutorial scripts are typically created by a human. The script for session playback may be generated from the application log, provided interface actions have been recorded.

2. Extending Functionality

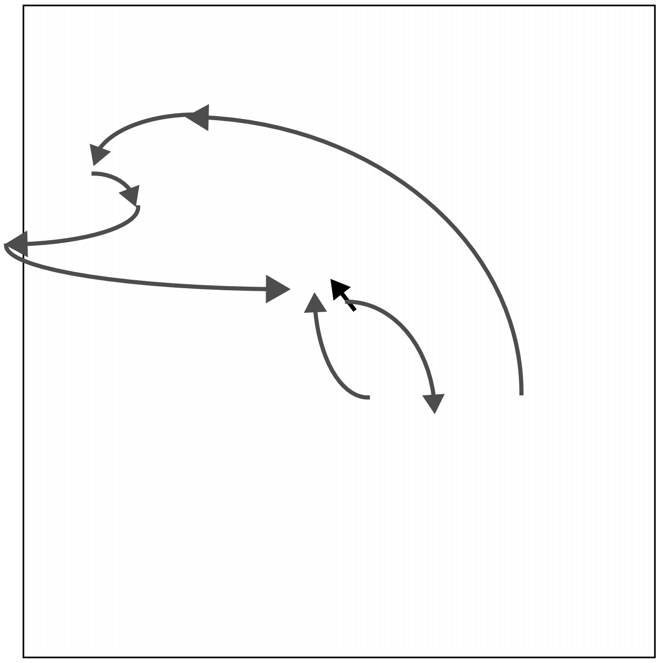
Animation may be used to extend applications by building task-oriented tools at a level above the interface. The creation of such tools is more programming intensive than creating a macro, but may have a pay-off when used to support tasks that are frequently performed. Context sensitivity adds to the power of this facility. Also, animation can assist in the specification of parameters by shifting input focus to appropriate parts of the screen.

The following example (see Fig. 13) illustrates a "Change Format" tool built on top of FrameMaker[1]'s search facility. Note that FrameMaker already has such a utility and this is merely an example.

The extension involves the creation a new window that drives the "Search" and "Character Format" windows of FrameMaker. The tool initializes the "Search" window and locates the first occurrence using the "Search" button. Then the cursor is placed between the "Skip" and "Replace" buttons, allowing the user to specify the next step.

1. FrameMaker is a trademark of Frame Technologies Co.

Figure 13. Task Oriented Extensions



(i) If the user presses the "Replace" button, the animator goes through the process of choosing "Helvetica" in the Font menu, selecting size 12 in the Font Size menu, and Italic in the Angle menu, before pressing the "Apply" button (not shown in the picture) to make the change effective. Then the "Search" button is pressed to locate the next occurrence, and the cursor is returned to its default position to close the loop.

(ii) If the user presses the "Skip" button the animator presses "Search" to locate the next word as before and returns the cursor to its default position.

This is accomplished by attaching code to the "Skip" and "Replace" buttons that causes scripts to be sent to the animation server. These scripts are parametrized by macros derived from the state of the settings in the tool. Note that the tool has no way of detecting when all matches have been found. Nonetheless, it is heartening to observe how much can be accomplished without support from the application.

3. CSCW

Groupware involves replication at some level. If the levels in an application are taken to be the view, the interface and the abstraction, the following configurations are possible:

Replicated View. (Single Interface, Single Abstraction) - Broadcast display events, multiplex input events.

Replicated Interface. (Multiple Interfaces, Single Abstraction) - Broadcast interface actions.

Replicated Application. (Multiple Interfaces, Multiple Abstractions) - Broadcast application actions.

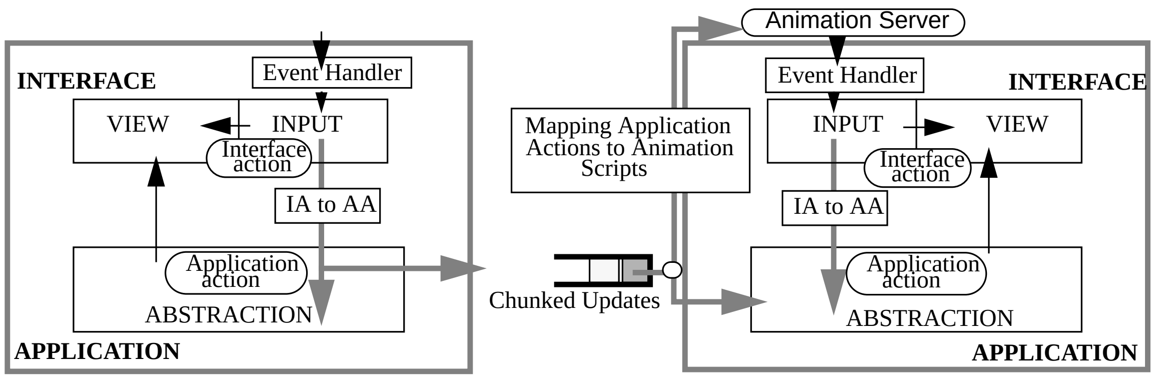
Application actions are typically implementation independent, and hence constitute a generic form of communication. The animation server has all the capabilities needed for tying applications together to form collaborative workgroups:

(i) The ability to invoke actions at the both the interface level and the application level.
(ii) Access to application context and hence the ability to adapt remote actions to local conditions
(iii) Parametrized scripts that may be used to map application actions to interaction techniques. Scripts are used to bridge the gap between various interface styles.
(iv) The ability to reproduce actions at a remote site in a demonstrative fashion, increasing the level of awareness in the group.

The overt reproduction of actions may not always be desirable. Fortunately the option of invoking application actions through callbacks is available.

Fig. 14 shows the design of a system for fully replicated collaboration based on animation servers (only one way communication is shown). Animation servers are present at every site and act as agents for other sites participating in the session. At each site the local interface maps input events to interface actions and then to application actions, which need to be communicated to other participants. If heterogeneous applications are connected together, the application actions may need to be translated to a common language before being broadcast. This is simplified by the fact that application actions tend to be abstract.

Figure 14. Replicated CSCW using Animation Servers



DistEdit [3] is an example of a toolkit that allows heterogeneous applications to collaborate using a common language for application actions. Periodically, each site transmits updates (translated application actions) over the network to other sites, where they get buffered. For maximum flexibility participants should be allowed to import updates into their system at their own convenience. At the same time the system must be aware of the significance of the updates occurring at various sites and maintain consistency. In this design we shall assume that the system will prevent actions that have the potential to interfere with each other. The buffered updates may be reordered, combined and compacted into "chunked" updates for better presentation to the user. The user is presented with a graphical view of the pending updates in a form similar to the "graphical histories" described by Kurlander in [5]. Users may choose to import the updates into their system, either internally as application actions, or externally (and explicitly) by animation. The latter option would be desirable when the graphical history does not explain the changes sufficiently, e.g. in the case of documents that have been edited in numerous places. Application actions are retrieved from the updates using a reverse translation process. The application's structure should allow for the arbitrary introduction of application actions in the system. The change must propagate to the view, providing appropriate feedback.

When actions are to be imported explicitly, they are converted to animation sequences using parametrized scripts. In UIDE, the mapping from application actions to interface actions and then to interaction techniques is present in the system definition. Since there is a one-to-one correspondence between interaction techniques and scripts, UIDE is in a position to generate its own animation sequences.

It should be emphasized that this architecture is suited not only for Same-Time-Different-Place collaboration but also for the more general, Different-Time-Different-Place CSCW. The update buffer is an immutable entity that keeps a log of actions performed at other sites, and provides participants who rejoin a session with a concise enactment of all that transpired in their absence.

4. Audio Interfaces

The animation server may be used in conjunction with a speech-recognition system to assist physically handicapped users to perform mouse based tasks on the workspace.

CONCLUSIONS

This work was born out of an attempt to build systems that provide context-sensitive animated help in a clean, non-intrusive fashion. Although primarily intended to "show" how things may be done on a graphical workspace, we discovered that the design is general enough to provide a variety of services. It is hoped that as the user community starts requiring their interfaces to be self-explanatory, extensible and collaborative, animation will take on the role of robotics in the user interface world. As a medium, animation has a long way to go and a large role to play in demonstration and tutorial systems of the future. In addition to its primary role as a presentation tool, the animation server may be regarded as both a pseudo input-device providing a new input modality (automated tasks, replicated actions in collaborative work), and as an intelligent agent capable of acting on behalf of the user (context-sensitive actions). Future research should be directed towards finding new applications for animation and designing applications that provide easy access to state information. We hope to investigate the use of animation in CSCW and in providing speech-based access to graphical user interfaces.

ACKNOWLEDGEMENTS

This work was supported by the Human Interface Technology group of Sun Microsystems and the Siemens Corporate R&D System Ergonomics and Interaction group of Siemens Central Research Laboratory, Munich.

We appreciate the comments and contributions of Jim Foley, Martin Frank, Mark Gray, Todd Griffith, Ellen Isaacs, Jeyakumar, Thomas Kuehme, Anton Spaans, Jouke Verlinden and Joseph Wehrli.

REFERENCES

1. Bach, M.J., *The Design of the UNIX Operating System*, Prentice-Hall Inc., 1986.

2. Cypher, A. EAGER: Programming Repetitive Tasks by Example, in *CHI ‘91 Conference Proceedings* (May 1991), pp. 33-39.

3. Knister, M.J. and Prakash, A. DistEdit: A Distributed Toolkit for Supporting Multiple Group Editors, *in CSCW ‘90 Conference Proceedings* (1992).

4. Kosbie, D. S. and Myers, B. A., A System-Wide Macro Facility Based on Aggregate Events : A Proposal, in *Watch What I Do - Programming by Demonstration edited by Allan Cypher*, MIT Press (1993), Ch. 22, pp. 433-446.

5. Kurlander, D. and Feiner, S. A History-Based Macro By Example System, in *UIST ‘92 Conference Proceedings* (Nov 1992), pp. 99-106.

6. *MacroMind Director Overview Manual*, MacroMind Inc (Mar 1988).

7. Maulsby, D.L., Witten, I.H., and Kittlitz, K.A. Metamouse: Specifying Graphical Procedures by Example, in *SIGGRAPH ‘89 Conference Proceedings* (Jul-Aug, 1989), pp. 127-136.

8. *Microsoft Windows 3.0 Data Sheet*, Microsoft Corporation (1992).

9. Mukherjee, B. A Portable and Reconfigurable Threads Package, in *Sun User Group Technical Conference Proceedings* (1991), pp. 101-112.

10. Palmiter, S. and Elkerton, J. An Evaluation of Animated Demonstration Systems for Learning Computer-based Tasks, in *CHI ‘91 Conference Proceedings* (May 1991), pp. 257-263.

11. Pence J. and Wakefield C., Tempo II, *Affinity Microsystems*, Boulder, CO 1988.

12. Petzold C., Windows 3.1 - Hello to TrueType, OLE and Easier DDE; Farewell to Real Mode, in *Microsoft Systems Journal*, Vol 6, No. 5, Sep 1991.

13. Potter, R., Triggers: Guiding Automation with Pixels to Achieve Data Access, in *Watch What I Do - Programming by Demonstration edited by Allan Cypher*, MIT Press (1993), Ch. 17, pp. 361-382.

14. Schiefler, R.W. and Gettys, J., *X Window System - Second Edition*, Digital Press, 1990.

15. Shneiderman, B. Direct Manipulation: A Step Beyond Programming Languages, (1983) *Computer*, 16(8), pp. 57-68.

16. Spaans, A. *Generating Context-Sensitive Animated Help*, Master’s Thesis (1993),TU Delft.

17. Sukaviriya, P. *Automated Generation of Context-Sensitive Help*, Ph.D. Thesis (1991), George Washington University.

18. Sukaviriya, P., Isaacs, E., and Bharat, K. Multimedia Help: A Prototype and an Experiment, in *CHI ‘92 Proceedings* (May 1992), pp. 433-434.

19. Sukaviriya, P. and Foley, J. Coupling a UI Framework with Automatic Generation of Context-Sensitive Animated Help, in *UIST ‘90 Conference Proceedings (Oct 1990)*, pp. 152-166.

20. Sukaviriya, P., Foley, J. and Griffith, T. A Second Generation User Interface Design Environment: The Model and the Runtime Architecture, in *CHI ‘93 Conference Proceedings* (Apr 1993).

21. *Technical Introduction to the Macintosh Family - Second Edition*, Apple Computer Inc.