$HOME MOVIE – Tools for Building Demos on a Sparcstation

Stephen A. Uhler – Bellcore

ABSTRACT

$HOME MOVIE is a suite of tools for the capture, editing and playback of window system sessions on a Sun Sparcstation. It includes ISDN voice quality audio, video, and a VCR-like user interface. At any time while the window system is running, a recording may be started, generating a complete log or script that captures the changes to the display. Simultaneously, an audio script is generated, containing any verbal descriptions, or sounds present. Once these *recordings* have been made, they can be re-arranged, edited, annotated or set to music, using the $HOME MOVIE sound and image editing tools. The resulting *movie*, can be played back on the display in real time, and thus provides a convenient way to document and demonstrate interactive software systems.

Introduction

Another demo. The boss walks in with another VIP just as you are starting to get some work done. Now you have to find a working version the program, crate the equipment into the other room, set it up, and pray everything holds together for the next ten minutes.

Presentation of high quality software demonstrations requires not only the appropriate hardware and software environment, but an expert user to manage the interface and often another person to explain what is happening. Existing demonstration methods include video-taped examples, which suffer from poor resolution and require special equipment, or special demonstration software with no audio capability.

$HOME MOVIE is a system for the Sun Sparcstation that solves the problem of preparing demonstrations of interactive software systems. $HOME MOVIE includes audio and video, with a television and VCR-like interface supporting pause, play, slow motion, fast forward, program selection and volume control. It requires no advance setup, and can be turned on at the spur of the moment. It is easy to add background music, or some video special effects and wind up with a snazzy self-contained demo.

How $HOME MOVIE Works

Capturing the Demonstration.

There are two methods that can be used to capture a session. With the first method, *input saving*, the inputs to the application are saved, along with the times between inputs. To replay the session, the saved inputs are re-sent to the program that re-executes the session. With the same inputs as originally used, in the same order and relative timing, the visual results will be identical to the recorded session. In the second method, all changes to the display, a *display list* are saved, along with the appropriate timing information. A stand alone driver program then interprets the display list to recreate the display images.

The *input saving* method has several advantages. For simple programs that require only keyboard and mouse input, the stored representation of the input can be made compact. In addition, capturing input can often be accomplished with no modifications to the program, by intercepting all input before it is sent to the application. Finally, since the demo'ed program is actually running, arrangements can be made for the viewer to take over the execution of the demo, and actually run the program. This capability is quite useful for training and on-line documentation. JYACC [1] is an example of a commercial system that provides this type of capability. Another system Whimsy [2] uses the input saving technique in a windowing environment by capturing an applications inputs to the window system. Whimsy is intended more for testing than for demonstrations.

Unfortunately, the *input saving* technique has some limitations. For many systems it is difficult, or even impossible to capture the entire input to an application. In an network environment, there can be subtle interactions between other programs on the network, as well as non-repeatable interactions with the operating system or file system. To recreate the demo, not only would the demo program need to be re-run, but so would other programs on the network, and all referenced files, network hosts, and machine states; clearly a monumental task. Finally, the playback can't be sped up or slowed down, but can only run at the current speed of the program. Often it is in just this kind of transitory environment, with new software in development, that a demo is required. Recreating the entire state simply isn't possible.

With the *display list* method, which is used by $HOME MOVIE, only the actual changes to the display are saved. None of the program input is required. Consequently, the program to be demo'ed is not needed for playback, and neither is the computing environment required to make the demo program run. An independent display list driver is used to recreate the applications display in real time. The demo is completely self-contained, can be distributed without compromising proprietary software, and can be run on a generic computing platform.

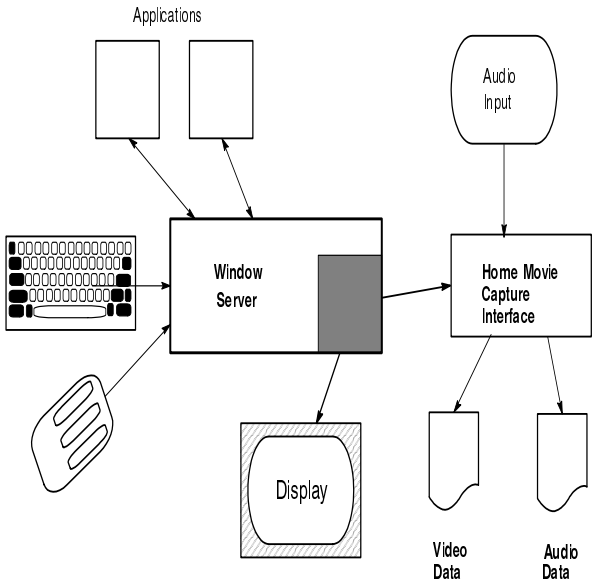

Figure 1: $HOME MOVIE Recording Setup

How the Video Portion is Saved.

To be effective for capturing demos, the demo software needs to be un-obtrusive. The application must be completely unaware that its output is captured, and no changes to the application, or the way it is run can be required. In addition, the user should be able to turn the demo capture on or off at will, with no prior planning.

For $HOME MOVIE to meet this goal, the window system server (not the application) is modified. All changes the window system makes to the display, along with timing information, are written onto a socket to be read by a separate process that saves the data in a file, the *video display list*. Figure 1 is a diagram of the demo capture setup.

The changes to the display are represented as the names of and arguments to the primitives that the window system uses to change the display. When a display primitive is invoked by the window server, a record of the invocation is generated. Recreating the display requires little more than reading the display list, then invoking the display primitives that were used by the window system to create the display in the first place.

The primary display primitive is a *bitblt* [3] (sometimes called a raster-op) which is used to change a rectangular set of pixels on the display. The *bitblt* operations are used to display images, print text, move the cursor, and update windows. The *bitblt* operations used by the window system often involve combining bitmaps (or pixmaps) that reside in memory with bitmaps in the frame buffer memory; the bitmaps that map to pixels on the display. For these operations it is necessary to keep track of not only the prior contents of the display, but to keep track of the contents of the memory bitmaps as well.

Although the entire demo session can be captured strictly with *bitblt* commands, some additional graphics primitives are used for improved efficiency. In addition to *bitblt*, these additional primitives are *points*, *lines*, and *circular arcs*. Still more graphics primitives, such as splines or polygons can be included in the display list as well, but are just as easily constructed out of the above primitives. The complete list of commands currently used by $HOME MOVIE and generated by the window system is shown in Table 1.

The first three items, *Bitcopy*, *bitblt*, and *Point* are various flavors of *bitblt* commands. The *Data* item represents image data. *Arc* and *line* are additional drawing primitives added for efficiency. *Display* and *Free* are used for book keeping, *Time* is for time stamps and *Comment* data is ignored, and can be used by other programs that process the display lists.

Table 1: Display List Commands

name	description
Bitcopy	*bitblt* - without source
Bitblt	*bitblt* - with source
Point	Draw a point
Data	Image data
Arc	Draw an elliptical arc
Line	Draw a line
Display	The display bitmap
Free	Free image data
Time	Time stamp
Comment	Comments

When the source or destination bitmap to a *bitblt* command is first referenced, its size and bit image are saved in the display list. When the playback program reads in the image for that bitmap, the image is cached for later use. The next time that bitmap is referenced, its image is already available in the video playback driver, so the image need not be repeated in the display list. For example, to display text in a window, the first time a character of a given font is referenced, the image of the entire font is saved in the display list. Every other character in the font is displayed by saving the *bitblt*

command required to copy that character from the already saved font image on to the display. The amount of memory required to cache the bitmaps varies with each application, but it is never more that was required of the server in the first place.

Code in the window system server keeps track of bitmap image changes, such as when a client application replaces one image with another, or when a bitmap is destroyed so that its contents are no longer required. In either case, a *bitmap free* command is saved in the display list, indicating a particular bitmap is no longer needed, permitting the display list driver to remove the image from its bitmap cache. The server then resends the new image data to the display list when required.

For most applications, the number of images that need to be saved in the display list is small, usually several fonts, icons, and cursors. All other display changes are made by combining these few images using *bitblt*, or other graphics primitives. When recording starts, the initial display image is saved in the display list, just as the first change to the display is about to occur. Each additional image is saved on the display list just as the first display primitive that references it is invoked. The window system server keeps a table of all images in use by the server, so it can readily find those that are required for the demo.

In one sample $HOME MOVIE session, a 13 minute demo of Superbook [4], a total of 29 images were saved in the display list. The image of the display when recording begins is saved, as well as image data for obscured windows that will be exposed during the demo. The rest of the images consist of cursors, fonts, icons, and graphics images specific to the application. The entire demo is created by performing *bitblt* transformations on the 29 images. Figure 2 shows the initial display of the Superbook demo, with the $HOME MOVIE user interface above the top of the display. Figure 3 contains the images that were required to reproduce the rest of the demo. The first few images are the fonts used by Superbook, each saved in the display list as the first character in the font was referenced. The fonts are followed by various cursors and icons either by Superbook or the window manager. The next image is an illustration presented to the user by Superbook, whereas the last image contains the contents of a window that was obscured on the initial display. Table 2 lists a summary of the images and sizes required for this demo. The total stored size of the images was about 60 kilobytes.

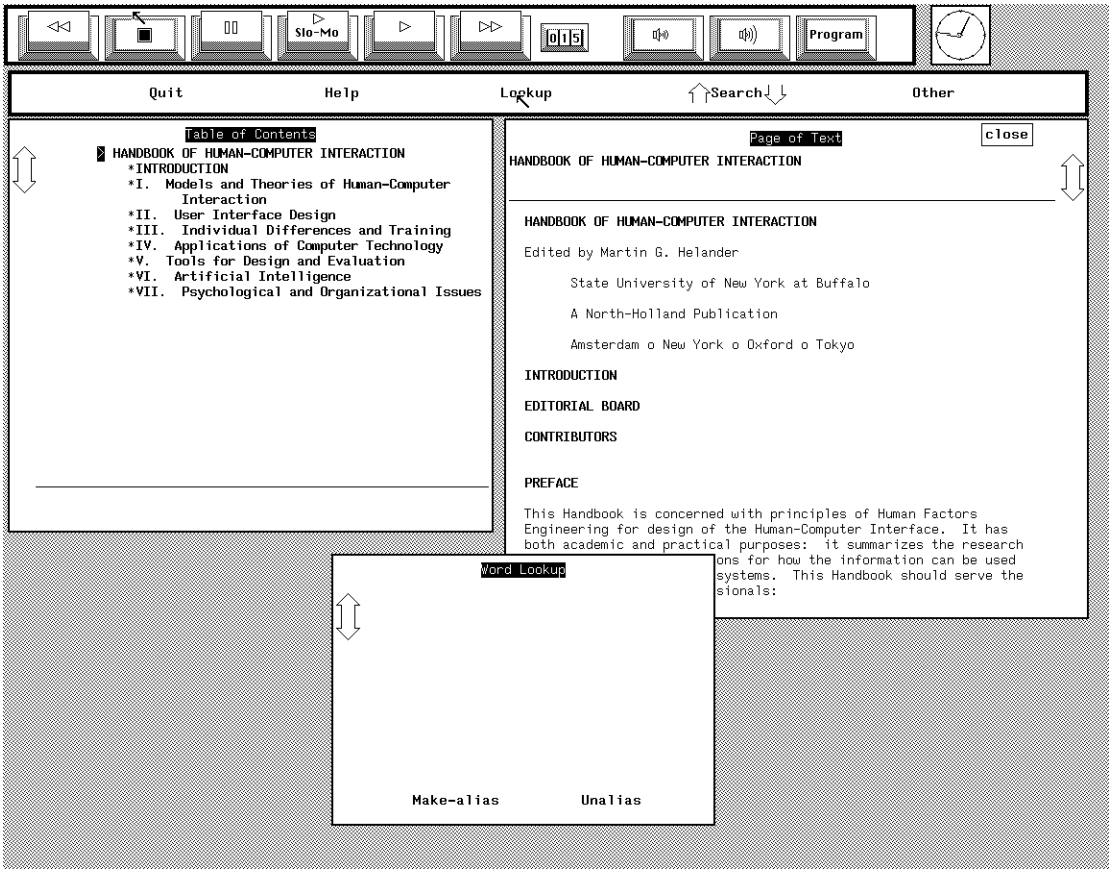

Figure 2: Initial Display of Superbook Demo

Because the display list stores low level *bitblt* and drawing primitives, the video data format is window system independent. It has no notion of fonts, cursors, images or windows; just operations that combine generic images. These are the same operations that are needed by all window systems that use a memory mapped model of the display. Consequently this method of recording demos is applicable to many window management systems, with no particular bias to any one in particular.

Table 2: image breakdown

type	number	size (as displayed KB)
cursors	13	1
fonts	8	18
windows	3	83
images	2	51
initial display	1	130

The window system generates timing information periodically, typically in the main *dispatch* loop in the server. This timing information is saved as a time-stamp in the display list. A time-stamp is a 32 bit quantity that represents 100ths of seconds elapsed since the window system session began. This permits about eight months of display information to be kept in a single display list. The display list driver program detects time-stamp overflow, thus permitting video scripts of almost unlimited duration. Absolute time information is used instead of time differences because it is easier to avoid rounding errors. It is also easy to alter the notion of time and play the display list back either faster or slower than it was originally generated. The video driver program understands a special *time offset* command that can be inserted into a display list at any point to add or subtract a fixed amount of time at any point in the display list without requiring the remaining time-stamps to be adjusted.

An arbitrary format was chosen for the display list data. In this format, each command consists of a 16 bit command identifier, followed by one or more arguments, as indicated by the command type. The display lists are normally stored in compressed form [5], and typically take less than 1000 bytes per second of demo.

In situations where the space consumed by the display list must be minimized, or where the video data needs to be transmitted in real time instead of saved in a file for later use, there are alternate data formats that vastly reduce the space required. A *bitblt* command is often similar to the previously

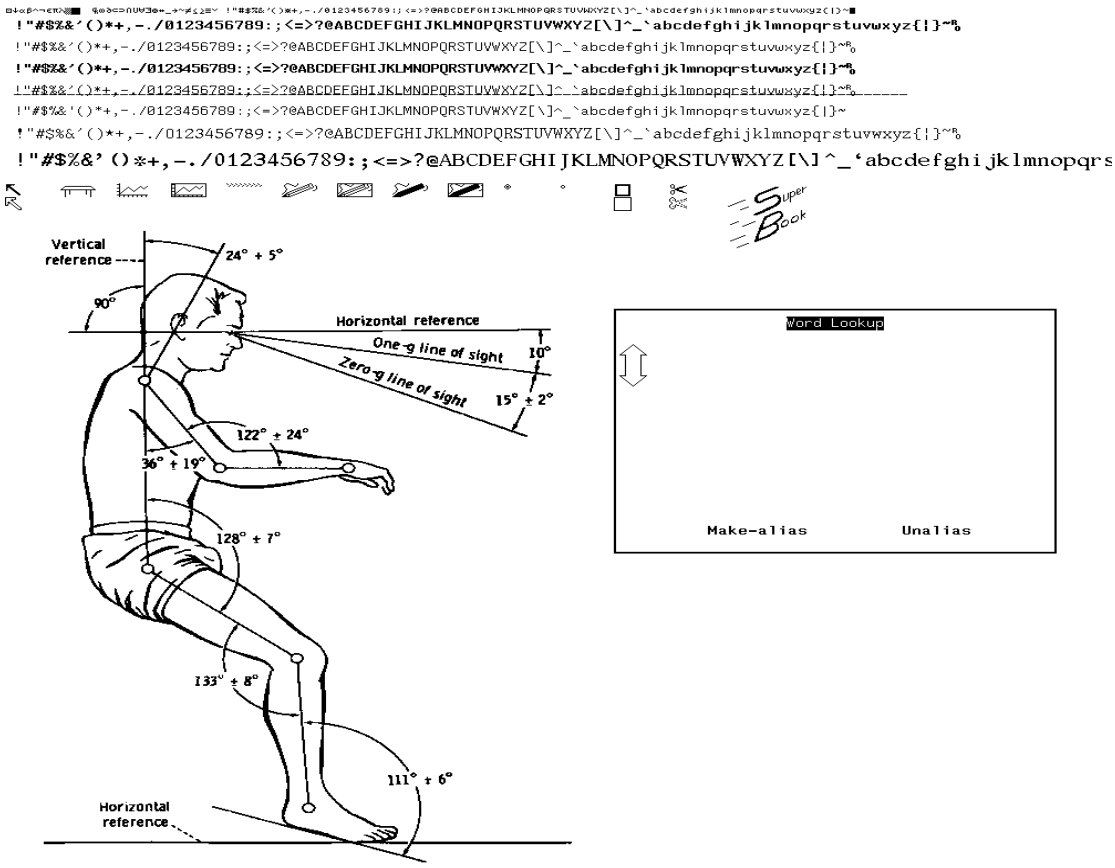

Figure 3: Collection of Saved Images used by The Superbook Demo

issued *bitblt* command. In 72% of the 89000 *bitblt* commands used in the Superbook demo, six or more of the nine arguments to the command were identical to the previous *bitblt* command. By choosing a display list format to encode differences from the previous command, substantial data compression can be achieved. In the extreme case, where ASCII terminal like output is prevalent, most *bitblt* commands can be encoded in a single byte. The usual case on the display in this instance is for the next character on the current line to be displayed. A source offset into the current font bitmap plus a destination offset on the display equal to the previous character width can be represented as a single character.

How the Audio Portion is Saved.

The audio portion of the demonstration is saved separately from the video script and the window system server. It is stored in a file in ISDN style μ-law format [6]. The μ-law format consists of 8000 8-bit samples per second. The audio can be voice, music, special effects, or other noise such as key-clicks or machine noise.

There are several reasons to keep the audio information separate from the video data. First of all, there is a large body of existing tools [7] available to manipulate the audio data. These tools can be used as is.

The large data rate difference between the audio and video is a more compelling reason to keep the audio separate from the video portions of the demo. Whereas the audio portion of the script requires 8000 bytes per second, the average bandwidth for the video display list is only one tenth that. At well under a thousand characters per second, it is possible to transmit the video data in real time over common dial-up lines. Although this might not be beneficial for demos that require sound, it is invaluable for providing remote dial-up window system services, using the same tools as the required by $HOME MOVIE.

How the Video and Audio Data are Synchronized.

Synchronization between the video and audio portions is maintained though the embedded timing information in the video script, and the fixed data rate format of the audio script. At regular intervals - about ten times per second, a timing mark is embedded in the video display list, representing the elapsed time in 100ths of seconds since the beginning of the script. That time, when multiplied by 80, represents the current byte offset in the audio file, thus

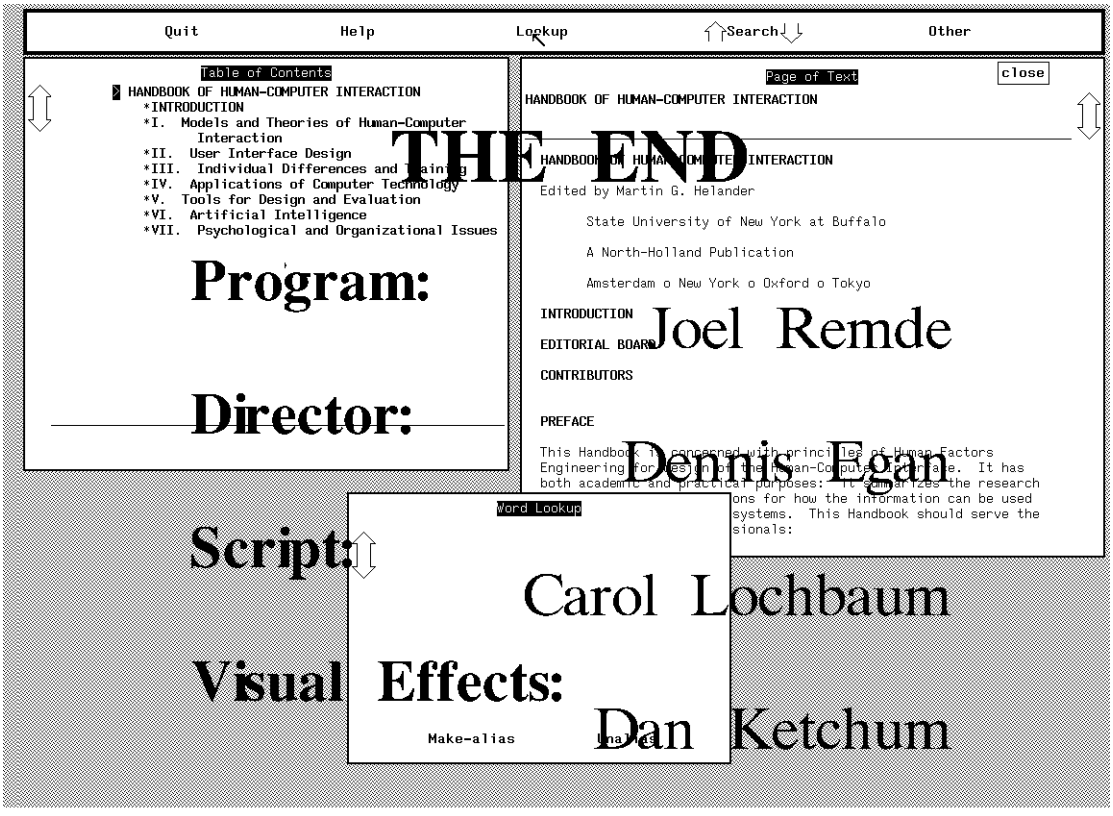

Figure 4: Sample Roll–A–Credit Output

maintaining synchronization to within 10ms, which is sufficient for demos.

Editing A Demonstration Script

Editing The Video portion

A Separate utility *Roll–A–Credit* was developed to generate text annotations, titles, and credits, which can be inserted into the video script files. Roll–A–Credit is a stand alone utility program that generates video display lists in the same format as the modified window system server. Words or phrases in one of several large fonts are animated by scrolling the text slowly on top of the current display. The initial and final position of each text phrase, the speed of scrolling, and the scrolling sequence is all under user control. The Roll–A–Credit display list is then inserted into a demo display list to effect the annotation. The input to Roll–A–Credit consists of one or more lines each containing four fields; 1) a font style and size, 2) text justification or position, 3) vertical offset from the previous line, and 4) the text to be animated. Often several Roll–A–Credit scripts are run consecutively to the same video display list, permitting different groups of text to be animated separately. Here is an excerpt from the Roll–A–Credit script used in the Superbook demo.

```
b-28  c  0     THE END
r-12  c  0  A Cog-Sci Video Production
```

The first line of the script causes *THE END* to be animated in a 28 point bold font, centered horizontally, and separated from the following text by the normal vertical spacing. The rate of animation, size of the drop shadows, and initial and final conditions are specified as arguments to the Roll–A–Credit command. Figure 4 shows a snapshot of Roll–A–Credit, taken from the credits portion of the Superbook demo.

The video script files can be converted to and from an ASCII representation using the program *to_ascii*. Once in ASCII, the display lists can be edited using standard UNIX tools such as *awk*[8]. Table 3 is a sample of the ASCII format. The initial character on the line is the command type; the remaining numbers are arguments to the command. Image data are saved in a hexadecimal representation. In the example above, the command characters *T*, *D*, *B*, and *L* stand for time-stamps, image data definition, *bitblt*s, and lines respectively. Lines beginning with "." define the image data. The command character is followed by the arguments. For time stamps, it is the elapsed time in seconds. For *bitblt*s, the most complex command, it is the destination bitmap number, the offset into the destination bitmap, the size of the rectangle, the *bitblt* function, the source bitmap number, and finally the source bitmap offset. The other command arguments are defined similarly.

Table 3, sample ASCII data format

```
T 5.17
D 13   32      16      1
. 198C0000198C0000198C000039CC00
. 0D9B8000001800000018000000180
. 00000180000000000000000000000
. 00000000000000000000000000000
. 00000000
B 2  557 28  16  16  12  13  0   0
B 13 0   0   16  16  12  2   561 26
B 2  561 26  16  16  14  15  0   0
L 2  49  46  22  76  5
L 2  49  46  19  73  5
T 5.63
```

To illustrate how one might edit a video display list, called *script.z* (the display list is stored compressed), suppose during the course of the demo, debugging output was accidently turned on while displaying a line drawing in a window. The debugging text wrecked our drawing. To fix it on playback, we can delete the non-line drawing commands from the display list that were output on the drawing window. The following command would be run:

```
zcat script.Z ∨ to_ascii ∨
  awk -f fix.awk ∨
  to_binary ∨ compress > new_script.Z
```

The appropriate awk script, *fix.awk* is:

```
{
if ($1=="T" && $2<4 && $2>9)
    print # Not within the proper time range
  else if ($1 != B && $1 != W)
    print # Not a bitblt command
  else if ($2 != 2)
    print # Not destined for the display
  else if ($3<46 ∨∨ $4<150)
    print # Not inside the window
  else if ($3>850 ∨∨ $4>700)
    print # Not inside the window
}
```

Each clause of the awk script examines a line for the ASCII version of the command, and passes it through unaltered unless it is one of the commands targeted for deletion.

Each command in the display list consists of a line containing of the name of the command, followed by its arguments. The playback program, to be described later, has a mechanism to place marks in the display list under user control. The user can watch the demo, and add marks at any point. These marks can be later used to aid in editing the script. The program *to_binary* performs the inverse function, converting the script back to its binary form.

Images stored in the display list are represented in hexadecimal ASCII, similar in format to *od -x*. To facilitate easier editing, The images can be extracted from the display list and stored as separate files in a standard image file format. The images may then be viewed and edited using standard picture editing tools, and later re-combined with the rest of display list.

There is a library of canned video effect scripts that can be used to join demo scripts together. These canned scripts, when sandwiched between two disjoint display lists, provide for smooth transitions between the two. There are canned scripts for fading gradually from one display image to another, or fading to black or white, or pushing the current display image off the screen with the new one.

The canned scripts work by looking at the two scripts to be joined, calculating the final image displayed by the first script, and the initial image displayed by the second script, then constructing the primitives (*bitblt* commands) to generate the desired transition.

Audio Editing

There is a set of audio editing tools to cut, paste, and manipulate sections of audio. These tools include filters for AGC (automatic gain control), squelch, mixing, stretching and shrinking portions of the audio script. The *time_it* utility reads a video display list, and displays the elapsed time to the hundredth of a second at each mark and script merge. The corresponding audio editing tools are then used to extract the proper lengths of sound, to match up with the timings displayed.

The IMG (Incidental Music Generation) system [9] can be used to compose short pieces of music to use either as backgrounds under voice, or to call attention to annotations, titles, or credits. IMG knows how to compose a music in one of several different genres. The exact duration of the piece, as well as its tempo is specified by the user: IMG does the rest, producing a MIDI [10] file containing the composition. The MIDI file is then rendered in software, or fed to a MIDI synthesizer whose output is connected to the Sparcstation's audio input. Either method results in a μ-law rendition of the composition. The *shell* command

```
compose -l21.4 grass ∨
   play_midi > /dev/audio
```

composes and plays a complete 21.4 second bluegrass piece.

To add music to a Roll–A–Credit title or annotation sequence, *time_it* is used to determine the exact duration of the Roll–A–Credit animation. Then IMG is instructed to compose a piece of the proper length, that suits the mood of the demo. After instrumenting and synthesizing the piece, it is inserted into the audio track to accompany the annotation.

Another use for IMG is to dub in background music underneath the narration in some parts of the demo. The demo is previewed, adding marks to the video script to indicate the beginning and ending of sections that need certain types of background accents. Either by using IMG, or clips of prerecorded music, the audio narration can be highlighted by mixing the music into the proper location to fit the audio narration or video display.

Playback

The playback portion of $HOME MOVIE consists of three processes, the user interface, the video driver, and the audio driver. The user interface accepts mouse hits on buttons as commands from the user and translates them into ASCII commands that are sent to both the audio and video display processes. The audio and video display processes read and interpret the demo scripts, under the control of the commands sent by user interface process. A diagram of the playback setup is shown in Figure 5. A short shell script, *movie*, sets up the playback environment and starts the three playback processes.

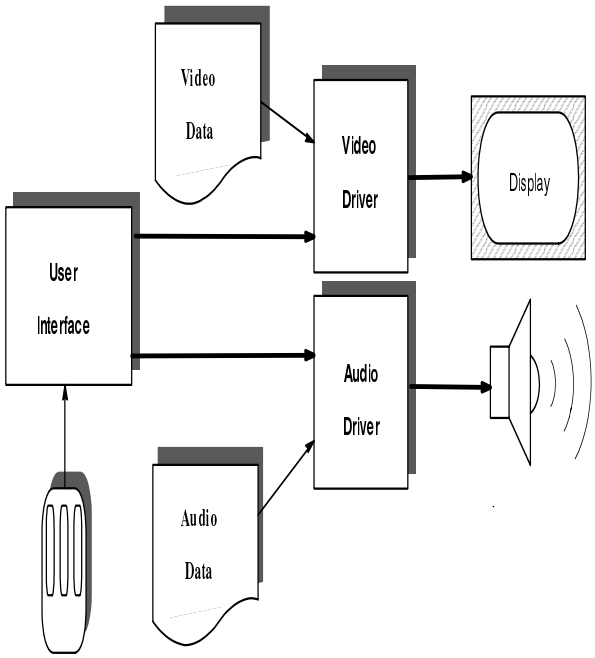

Figure 5: $HOME MOVIE Playback Setup

User Interface

VCR, the primary user interface to $HOME MOVIE, simulates the functions found on a video cassette recorder. Figure 2 shows a picture of the user interface, at the top of a demo in progress. VCR provides a mouse activated button interface to the $HOME MOVIE playback system. From left to right it has buttons for rewind, stop, pause, slow motion, and fast forward. Following fast forward is

a tape counter, volume down and volume up. Finally at the right edge is a program button. Except for the program button, the interface works just like a standard VCR. The playback start up script normally starts in just the top inch of the display, with the VCR program running. The remainder of the display is used by the video driver for the demo.

Using the program button on VCR users can step though a sequence of *tapes*, choosing the one they wish to play. VCR reads a startup script that maps each tape name shown on the *program* button into a list of one or more demo scripts which are played consecutively when that name is selected.

Since the design of the playback system is modular, the various parts can be easily interchanged. By replacing the user interface by a command file, repeated playback of a sequence of demo scripts results in completely unattended demos.

Video Playback

The video driver accepts commands from the user interface and plays back the video script. Normally the video driver writes directly to the display, except for the top inch occupied by the user interface.

During normal *play* operation, the video driver reads and processes display commands from the display list generated when the demo was first created. The time-stamps embedded in the display list are compared against the real time elapsed since the beginning of the script. Whenever the script elapsed time is greater, the video driver sleeps for the difference, thus recreating the pace of the original demo. If the user selects *fast forward* or *slow motion* on the user interface, the script elapsed time is multiplied by a constant other than unity. The effect is to speed up or slow down the notion of time, permitting playback either faster or slower than the original demo.

Audio Playback

The audio driver, like the video driver, also accepts commands from the user interface. In the current implementation, those commands are passed from the user interface through the video driver, so the playback mechanism can easily be started as a pipeline by the shell. the commands tell it to read the audio data from the appropriate point in the audio file, and send it to */dev/audio* to be played out the speaker. As the video playback is stopped, started, speeded up or slowed down, the audio driver

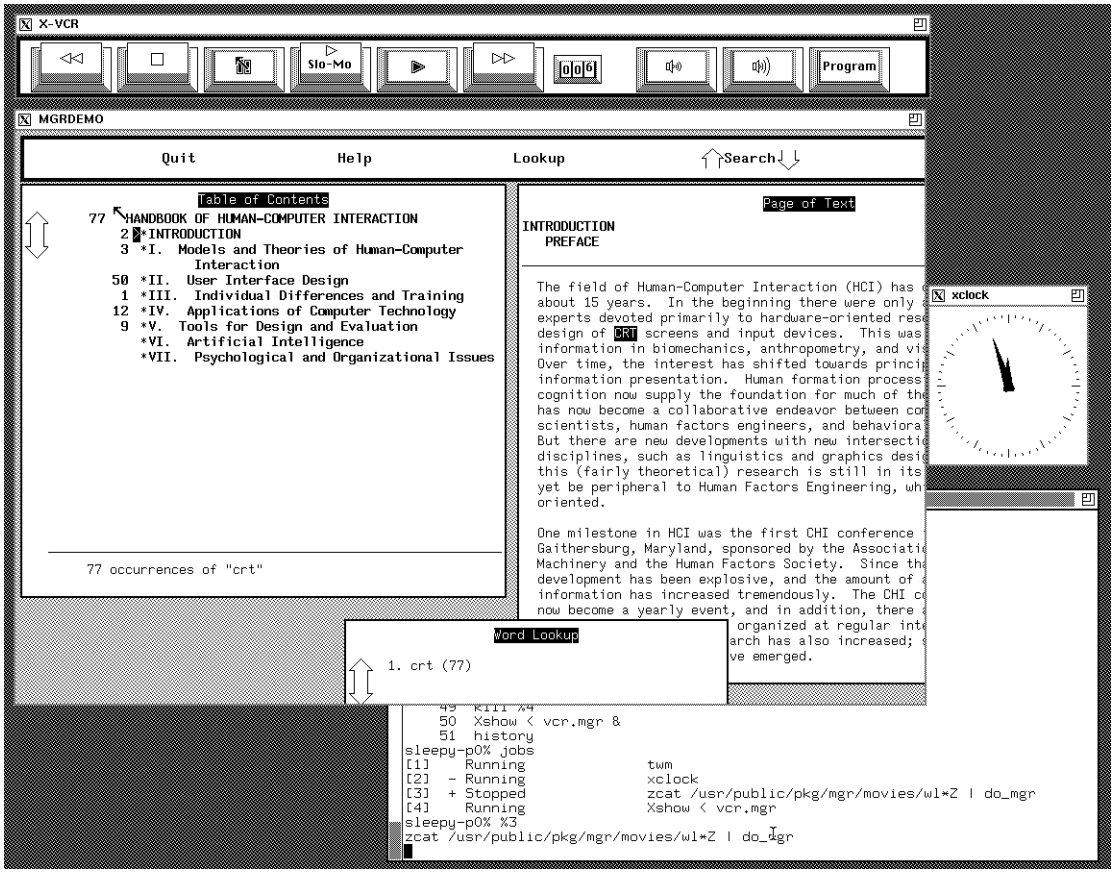

Figure 6: Superbook Demo in an X Window

receives synchronization commands from the user interface so it can determine which byte of the audio file should currently be coming out of the speaker.

When slow motion or fast forward is selected, the audio track is processed to run slower or faster without changing the pitch of the sound track, and can maintain intelligibility over a wide range of playback speeds. This is accomplished by either eliminating or duplicating groups of samples, then smoothing the edges where the groups abut. The fraction of samples removed or duplicated determines how fast the audio track is speeded up or slowed down.

Design Tradeoffs

The low level format for saving the display data was chosen to be window system independent, and requires only a simple driver program to play back the script.

The video and audio tracks are kept separate, in spite of potential synchronization problems and editing difficulties, so the tools that manipulate the data are simpler, and the video portion of the system runs unchanged for systems with no audio capability.

Implementation Considerations

The current version of $HOME MOVIE was produced by modifying the MGR [11] window system to send the video display list information out a socket, about 300 lines of C code. The X11 [12] version of $HOME MOVIE is under development, and uses the same strategy.

There are several ways to play back the demo. Normally, playback is made to the raw display. The video driver program is completely self-contained, and consists of 1600 lines of C code. About 1000 lines comprises the *bitblt* engine, the remainder reads and interprets the display list and user interface commands.

The video scripts can be played back into an X window. The X version of the video driver program, using Xlib calls, is a 1000 lines of C code. For X video driver, the display lists are played back in a window instead of the entire display. This allows the $HOME MOVIE system to be used in the context of another application, such as providing on-line animated help. Figure 6 shows a portion of the Superbook demo playing back in an X window.

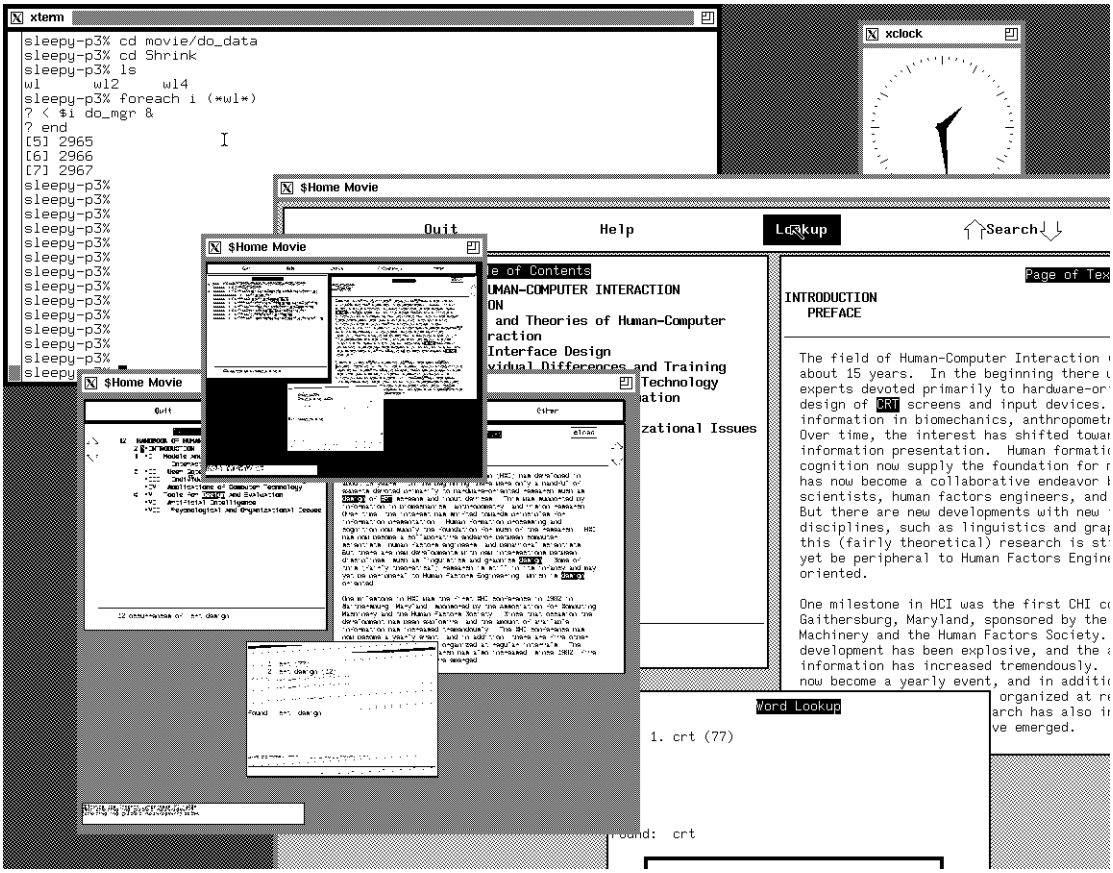

Figure 7: Shrunken Playback in X

The playback can be shrunk to a smaller size than the original recording, permitting playback in less than display-sized windows. This is a direct consequence of the geometrical nature of the video display list format. The drawing commands are scaled simply by scaling their coordinates. Only the images need any significant processing, and are easily reduced in size by integral multiples, although with a corresponding loss of resolution. Figure 7 shows a portion of the Superbook demo playing back in three separate windows; full size, half size, and quarter size.

The audio and video playback portions of $HOME MOVIE are played by separate processes, so even though synchronization data is kept to within a hundredth of a second, due to the granularity of the UNIX scheduler, the audio and video synchronization has considerably greater variance. Packaging the video and audio playback in the same process would ameliorate this problem to some extent, but at the cost of some flexibility.

Conclusions

As a test case, a 13 minute demonstration of the SuperBook hypertext system was prepared. The video display list averages 791 bytes per second, whereas the audio requires a constant 8000 bytes per second. The Superbook movie has been shown dozens of times to hundreds of people and greatly reduces the need for expert users to be present.

The changes made to the window system have a minimal impact on the server performance, and require no changes to either the user or application interfaces. Thus a demonstration can be captured with no prior planning.

The playback interface is familiar, and once pushing buttons with the mouse is mastered, it is obvious and easy to use.

Since the $HOME MOVIE playback portion is small and self-contained, a *demo* diskette can be mailed anywhere, providing a self-contained autonomous demo.

References

[1] *JAM JYACC Application Manager*, Users manual 1988 JYACC Inc.

[2] Cohn, R.J., *Automated Testing of Interactive Programs*, Unpublished memorandum, January, 1987.

[3] Foley, J.D. and VanDam, A., *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, 1982.

[4] Remde, J.R., Gomez, L.M., and Landaur, T.K., *SuperBook: An automatic Tool for Information Exploration -- Hypertext?*, Proceedings of Hypertext '87. University of NC, Chapel Hill, 1987 pp. 307-323

[5] Welch, T.A., *A Technique for High Performance Data Compression*, IEEE Computer, vol. 17, no. 6 (June 1984), pp. 8-19.

[6] Members Technical Staff, Bell Telephone Laboratories, *Transmission Systems for Communications*, 1959.

[7] Langston, P.S., *UNIX Midi Manual, Sections I, III, & V*, Internal Bellcore Technical Memorandum TM ARH-015440, November, 1989

[8] Aho A.V., Kernighan W.K, Weinbeger P.J., *The AWK Programming Language*, Bell Telephone Laboratories, 1988

[9] Langston, P.S., *PellScore, An Incident Music Generator*, Internal Bellcore Technical Memorandum TM-ARH-016281, February, 1990

[10] DeFuria, J.S. and Scacciaferro, *MIDI Programmers Handbook*, M&T Publishing Co., 1989

[11] Uhler, S.A., *MGR - C Language Application Interface*, Bellcore Internal Technical Memorandum TM-ARH-010796, December, 1988

[12] J. Gettys, R. Newman, T. Della Fera, *Xlib - C Language X Interface*, January 1986.

Stephen Uhler joined Bell Communications Research at its inception in 1984, where he is a Member of the Technical Staff in the Computer Systems Research division. He has worked on computing environments and user interfaces for much of that time, and is the author of the MGR window system. Before joining Bellcore, Stephen was a Member of the Technical Staff at Bell Laboratories in Whippany N.J. where he worked on user interface management systems. He received an M.S. degree from Case Western Reserve University. Stephen can be reached via electronic mail at: sau@bellcore.com or uunet!bellcore!sau .