Video Widgets and Video Actors

Simon Gibbs, Christian Breiteneder,
Vicki de Mey and Michael Papathomas

Centre Universitaire d'Informatique, Université de Genève
24 rue Général-Dufour, CH-1211 Genève 4, Switzerland
tel: +41 (22) 705.7770; fax: +41 (22) 320.2927
{simon, chris, vicki, michael}@cui.unige.ch

ABSTRACT
Video widgets are user-interface components rendered with video information. The implementation and several usage examples of a family of video widgets, called video actors, are presented. Video actors rely on two capabilities of digital video: non-linear access, and the layering of video information. Non-linear access allows video frames to be displayed in arbitrary order without loss of continuity, layering allows two or more video streams to be spatially composed. Both capabilities are now becoming available to user-interface designers.

KEYWORDS: digital video, video layering, video widgets, media components

INTRODUCTION
Recent developments in low-data rate digital video formats, such as DVI [10], QuickTime [5], MPEG [8], and JPEG [11], are creating new possibilities for user-interface design. Unfortunately, many applications treat video simply as something which can be played in a window. This view reduces interactivity to choosing when and where to start and stop the video, and ignores the possibility of *processing* the video. Furthermore, by relying on a separate video "playback window", there is little coupling between video material and the material appearing in other application windows. It is difficult, for example, to have a video pointer appear over application buttons or menu items.

This paper shows how traditional video processing techniques, such as keying and layering, can be used to integrate video with the user interface. Currently these forms of video processing are usually performed with studio equipment, however as computer support for digital video becomes more common, video processing is also appearing on the desktop.

In integrating video with the user interface, an essential concept is that of a *video widget* – a user-interface component that is rendered using video information. Like graphics-based widgets, video widgets come in a variety of forms. For example, a video button widget might use a video sequence, instead of an icon or character string, for the face of the button. (Imagine a button that is a burning flame or a fountain of water.) Rather than discuss video widgets in general, we will concentrate on a family of video widgets that we call *video actors*. These widgets are characterized by human-like behavior and are typically used to provide assistance or help information.

In the following we describe how video actors are used in applications and how they can be implemented. We first mention some closely related uses of video in the user interface. We then present a conceptual overview of video actors. Since several pieces of information are associated with a particular video actor, and since different actors have different behaviors, the object-oriented approach is used. The implementation of video actors is then described at two different levels: the application level, and at the level of "multimedia components". We show how a strategy of constructing applications from collections of cooperating components can be used to build video actors. Next we present several examples of applications using video actors and briefly touch on the problem of acquiring suitable video material. Finally we conclude with a discussion of problems related to our implementation of video actors.

RELATED WORK
Composition of graphics and video, as needed by video widgets, is a widely used technique. For example, many computer games and interactive video games overlay video sprites on computer-generated backgrounds or on background video (such as from a videodisc). Video titling systems ("character generators") are another example of graphics and video composition where, in this case, computer-generated titles are overlaid on a video background.

Chroma-keying and other video composition techniques are a mainstay of video production but are little used for computer interaction. (Although, as stated in the introduction, this may well change as digital video becomes more common.) One exception is the *Mandala Machine* where live video, typically of the user shot against a uniform background, is overlaid on a computer-generated scene. Special hardware determines the outline of the user, allowing the video to control the application (e.g., moving a hand over a drum icon causes a drum sound to be produced). Another exception is Krueger's work [7] where live video and real-time gesture and edge recognition are combined to produce highly interactive video worlds. One difference with the work described here is that we use recorded rather than live video. This means that the video can be prepared in advance and devised for specific purposes.

The notion of video actors derives from work directed towards depicting and incorporating human-like agents in the user interface. Several exploratory videos have been produced to help visualize such interfaces (e.g., *Knowledge Navigator* [6]) and systems containing guides or agents have appeared. An early example is *Palenque*, a DVI title where a video guide assists the user in exploring a tropical rain forest (a description can be found in [10]). A second example is the use of guides to help users navigate a hypermedia database [3][9]. Both graphics-based (computer-generated) and video-based guides were developed; this work also experimented with attributing character to the guides so they not only assist in navigation but also supply a point of view on the content of the database. Another example of video-based agents, an application which aids in learning a foreign language, is described in [1] where the roles of "cultural agents" in multimedia interfaces are considered.

Several of the video actors we have implemented explain the operation of software or pieces of equipment. In our case the video explanation is prerecorded. Work has been done, however, on the generation of multimedia explanations. For example, COMET [4] performs the automatic generation of explanations concerning the maintenance and repair of equipment. The explanations are visualized on the computer through graphics and text and generated using an expert system and several knowledge bases.

This brief summary has mentioned only a few of many examples of user interfaces that utilize video. While the design of video-enhanced interfaces is an engaging and creative area, the focus of this paper is more on the design of software abstractions. In particular we are interested in identifying portable, reusable abstractions that enable interface designers to experiment with non-linear access, video layering, and other capabilities of digital video.

VIDEO ACTORS – CONCEPTS

A video actor consists of layered video material that appears on top of application windows. Applications create and manipulate video actors by instantiating a C++ class. Before describing the interface provided by this class we first introduce some of the terms and concepts relevant to video actors:

multiple video sources a video actor is composed from one or more video layers. Layers are separate video values and are accessed by separate players, i.e., there is one player for each video layer and each player can be controlled independently of the others.

layer composition a video actor's layers are composed, using techniques such as chroma-keying and luminance keying, into a single video stream. Application graphics and the single video stream are then overlaid so that the graphics appears *behind* the video. In other words, windows, icons, menus etc. are in the background and the video actor is in the foreground.

non-linear / random-access video Within a layer, individual video frames can be randomly accessed and displayed, in other words playback is "non-linear." Access times are crucial and will depend on the format and size of the video and the performance of the player.

actions, postures and *events* Each video actor has a repertoire of actions and postures. An action is a path through the frames contained within a video layer. Actions need not be sequential and may have accompanying audio. A posture is a particular frame (or set of frames, in order to reduce access times postures can be replicated and scattered through a layer). It is possible for a video actor to perform several actions simultaneously provided each is on a different layer. As video actors perform actions they generate PostureHit and EndOfAction events as pictured in Figure 1. Applications can request notification of these events allowing synchronization of application activities with the actor.

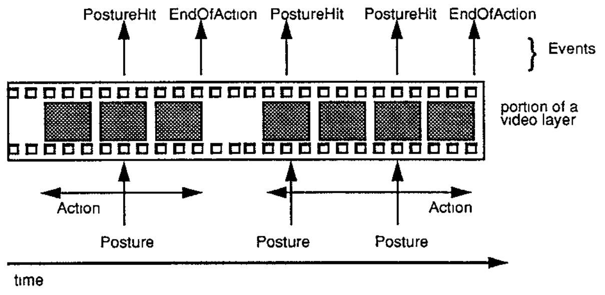

Figure 1: A video layer showing Events, Actions and Postures.

The basic functionality of video actors is provided by a VideoActor class. An outline of this class, in C++, is:

```
class VideoActor {
    int             nlayers;        // how many layers for this actor
    VideoPlayer*  player[ ];     // players for each layer
    LayerInfo      layer[ ];        // state information for each layer
protected:
    // protected instance variables include chroma-key info
    // plus tables defining actions and postures for this actor
    //
    VideoPlayer* Player(LayerId  lay);
public:
            VideoActor(int  n);         // create an actor with n layers
            ~VideoActor( );

    int     LoadLayer(LayerId  lay, char* fileName);
    void    Map(LayerId  lay);          // make a layer visible
    void    UnMap(LayerId  lay);        // make it invisible
    void    Mute(LayerId  lay);         // make a layer inaudible
    void    UnMute(LayerId  lay);       // make it audible

    videoKey BackgroundKey(Layer Id  lay); // identifies
                                            // background
    int     RepertoireSize( );  // how many things can it do?
    void    Repertoire(ActorRepertoire*  ar); // what are they?

    int     Perform(LayerId  lay, ActionId  a, float  speed,
                    bool  block);
    int     Pose(LayerId  lay, PostureId  p);

    void    Register(ActorEvent  e, ActorEventHandler  h);
    void    UnRegister(ActorEvent  e);
};
```

Layers, actions and postures have unique identifiers. When using the VideoActor class directly, applications must be aware

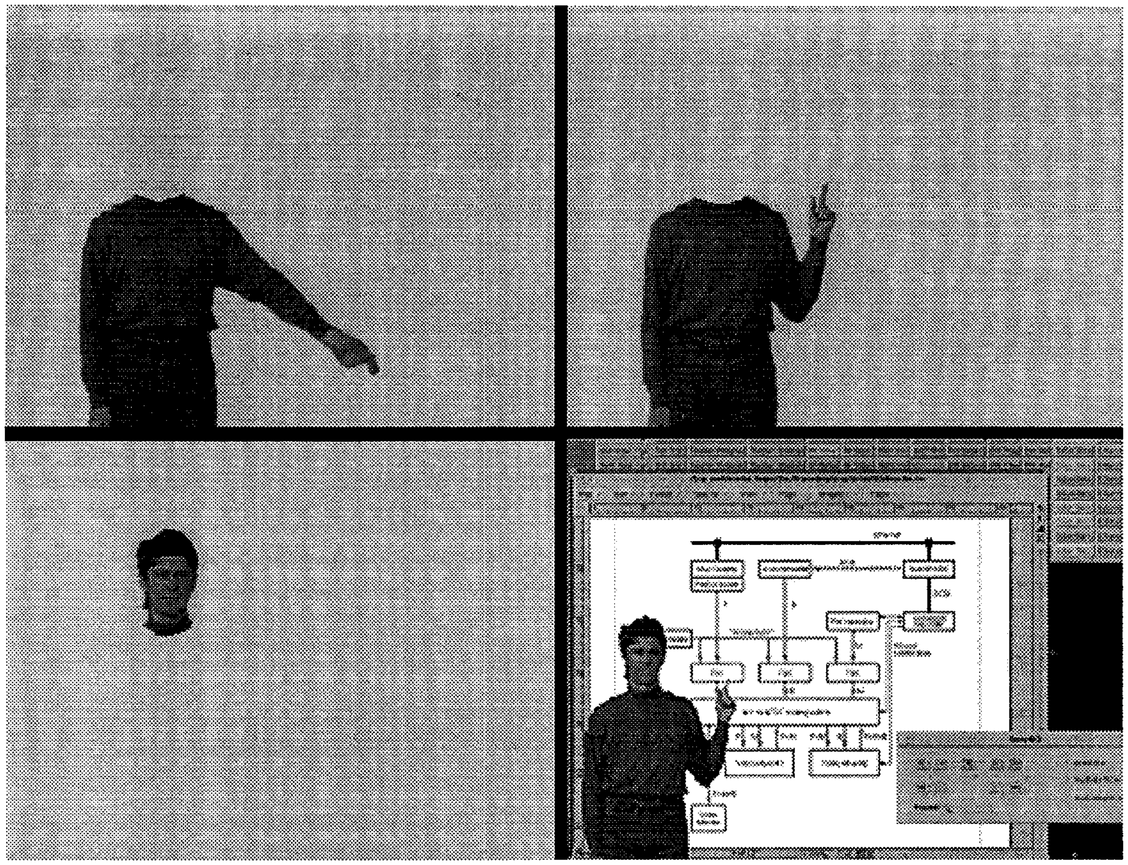

Figure 2: A video actor: two frames from the torso layer (top), a head layer frame and composed video overlaid on an application (below).

of these identifiers. (Our practice, however, is to create subclasses of VideoActor which encapsulate this information.)

After creating an actor, the first step is to bind video values to its layers. This is done by the LoadLayer method which indicates the name of a file containing a video value. The format of the file will depend on the player for the layer in question. For example, one of our players is a network accessible QuickTime server (see Appendix B). In this case fileName will be the name of a Macintosh file containing a QuickTime movie. We also use videodisc players in which case fileName is ignored.

The BackgroundKey method returns a value, videoKey, that separates background from foreground in a video layer. The content of videoKey will depend on how keying is performed. For instance, using luminance keying only a single value, the threshold level, is needed, while chroma-keying requires specification of a key color or color range (e.g., maximum and minimum values for red, green and blue).

The RepertoireSize and Repertoire methods are used to query an actor and determine the actions and postures of which it is capable. In addition to returning valid action and posture identifiers, Repertoire indicates the layers used, and, for actions, whether there is audio, and if so, the language used.

Perform is the crucial method. It directs an actor to start an action on the specified layer (and at the specified speed). Perform takes a boolean parameter indicating whether the method should block. By using non-blocking invocations it is possible for an application to direct an actor to perform more than one action at a time (provided the actions are on different layers). Alternatively, blocking invocations allow applications to synchronize with the end of actions. The Pose method simply requests that the actor assume a specific posture.

Finally, the Register and UnRegister methods are used to set and remove event handlers. An ActorEvent includes an event type (e.g., EndOfAction, PostureHit), a layer identifier, an action or posture identifier, and data to be passed to the application. (To catch all events of a particular type, the application uses wildcards for action or posture identifiers.)

As an example, consider an actor with two layers: one showing the torso of a person and the other the head (see Figure 2). The head layer has an accompanying audio track containing a number of utterances. Actions are defined for particular utterances. The body layer has a number of body positions, such as pointing in a particular direction, which correspond to postures. The following gives an idea of how an application could invoke this actor:

```
vact = new VideoActor(2);        // we want two layers
vact->LoadLayer(TORSO_LAYER, "Torso Track");
vact->LoadLayer(HEAD_LAYER, "Talking Head Track");
vact->Pose(TORSO_LAYER, POSTURE_POINT_LEFT);
vact->Perform(HEAD_LAYER, ACTION_SAY_HELLO,
              someSpeed, FALSE);
```

Since in this example the Perform does not block, it is possible for the application to continue handling user events. These could result in changes of posture, allowing, for example, the actor to follow the movement of a window or cursor.

IMPEMENTATION OF VIDEO ACTORS

This section presents a general software architecture for video actors and describes how the architecture has been realized on a particular multimedia platform. We begin by discussing the application programmer's view of video actors.

Application Level – The Application Programmer's View

Video actors are overlaid on top of the windows, icons, menus etc. used by applications. In our present implementation, the video "floats" over the graphics, i.e., it is not clipped or otherwise hidden. (This is somewhat restrictive and, in cases where actors are interrupted, it seems appropriate for windows to be brought on top of actors.) Assume that graphics rendering is done by an X server. Then an application programmer's view of the relationship between actors, the X server, and the application itself is shown in Figure 3. In this figure, the application

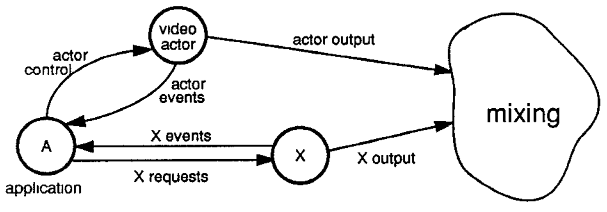

Figure 3: Video actors: application level.

A is an X client, it has a connection to an X server and creates an instance of the class VideoActor. The application programmer is aware that the video actor's output is "mixed" with that of the X server but need not be concerned with how this is done.

Component Level – The Video Actor Implementor's View

There are many ways of implementing the VideoActor class, they differ in the number of actor layers supported and the hardware required. We will describe two implementation approaches. The first is a *possible* (but not realized) digital implementation. In this case the number of actor layers is limited by processor speed; a very fast processor or special-purpose video processing hardware is required. The second implementation approach, and the one we have followed, is an analog-digital hybrid. Our current hardware configuration (see Appendix A) supports two actor layers but is scalable (i.e., more layers can be supported by adding more equipment).

The two implementation approaches can be easily described in terms of *multimedia components* [2]. These are software abstractions representing hardware devices and software processes. Components are connected to form media processing systems, connections carry streams of data (such as digital video frames or X server requests) *or* analog signals.

A possible component network (a group of connected components) for a digital implementation of video actors is shown in Figure 4. Here a Player is a digital video source (e.g, a process which reads compressed video frames from a file and produces decompressed frames) and DVMIX is a digital video mixer (e.g., a process which receives frames from several sources and produces a single composite frame). The other components make explicit the connections between an X server with digital video extensions (the "X+DV" component), the framebuffer, and the display. The modifications needed by the X+DV component are the addition of a port to receive the stream of digital video frames, and the addition of server requests to control compositing of the normal X graphics with the video (e.g., XStartCompositing, XStopCompositing and XSetChromaKey). Since the final mix (that of the actor and the application) occurs in the modified X server, we label Figure 4 a "server-resident mixing" implementation.

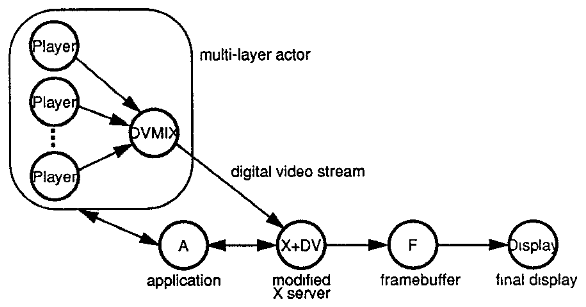

Figure 4: Multi-layer digital actor: component level, server-resident mixing.

In choosing the components to implement video actors we faced two practical constraints. First, our current equipment supports primarily analog video. Second, we were not prepared to modify the X server to incorporate digital video extensions. As a result, our current implementation of the VideoActor class relies on analog video (see Figure 5). Here two

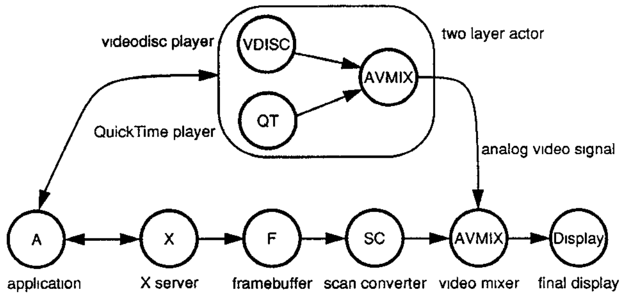

Figure 5: Two layer analog actor: component level, post-server mixing.

sources, such as a videodisc and a QuickTime player, produce two actor layers. These signals pass through AVMIX, an analog video mixer, whose output is in turn mixed with an analog signal coming from a scan converter. (A scan converter is a stand-alone piece of equipment, or a computer board, that converts a framebuffer's RGB video signal to NTSC or some other analog format.) After mixing the result is displayed on an NTSC monitor. Since the final mix occurs after the X server, we label Figure 5 a "post-server mixing" implementation. Although this implementation suffers from the problems of

RGB to NTSC conversion (loss of resolution, color bleeding etc.) it does allow experimentation with the VideoActor class. However, the additional equipment (videodisc player, scan converter, video mixer) leads to an expensive and bulky platform.

EXAMPLES OF VIDEO ACTORS

We have implemented a few examples of video actors using the hardware and software described in the previous sections. The examples fit into two categories: domain-specific "assistants" used to explain particular applications, and generic "gesturers" used to test the basic capabilities of actors, such as random access to video and real-time video display in response to an input device. We have implemented three assistants and one gesturer. The assistants are: a videodisc assistant that provides help to users of a video player application; a museum guide, shown in Figure 6, that supplies information about artwork on display in a virtual museum [2]; and a fax assistant, shown in Figure 7, that explains and demonstrates the use of a fax machine.

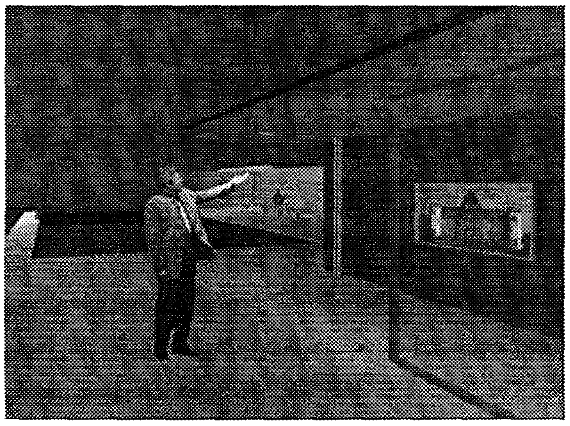

Figure 6: Museum guide.

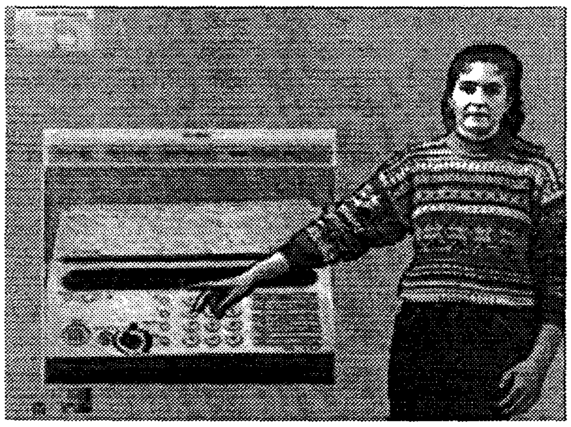

Figure 7: Fax assistant.

In this section we further discuss the videodisc assistant and gesturer actors. The former illustrates the interaction of an application with a single-layered video actor. The latter is an experiment with a multi-layered actor whose different layers are capable of tracking the mouse across the screen and performing a small repertoire of gestures. We also discuss how the video for the examples was acquired.

The Videodisc Assistant – A Domain-Specific Video Actor

In this example a video actor is used to assist the use of an application. The application consists of a panel that controls playback of video from a videodisc in a window on a workstation screen. The large panel appearing near the left side of Figure 8 has video control buttons such as play forward, fast forward and stop. In addition to the buttons used for controlling playback, there is a help button, marked with a "?". If a user pushes this button before pushing one of the other buttons, a video assistant pops up and explains and *demonstrates* the use of the latter button.

Demonstrating the use of a button is the most interesting part of this example. It is accomplished through a callback occurring when the video actor layer reaches the posture where it seems as if it pushes the button (shown in Figure 8). The callback causes execution of the associated command for a short time period, for instance it plays a short part of the video, then the video actor disappears and control is returned to the application.

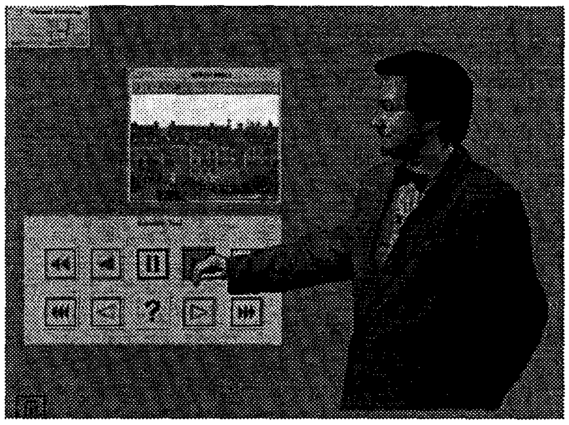

Figure 8: Buttons explained by the "videodisc assistant".

The VideodiscAssistant class implements the behavior of this actor. A skeleton of the definition of this class is:

```
class VideodiscAssistant : public VideoActor {
public:
    bool    CanExplainButton(int buttonId);
    void    ExplainButton(int buttonId);
    void    RegisterHitButtons(ActorEventHandler h);
};
```

The CanExplainButton tests if the video actor has an explanation for the chosen button in its repertoire. The ExplainButton method starts the explanation of some button. The RegisterHitButtons is used for registering a callback to occur when the actor appears to push a button. These methods are implemented using the Repertoire, Perform and Register methods of its superclass, VideoActor.

The Gesturer – a Generic Video Actor

The display of the "gesturer" is made up of two video layers, one layer for the head of the actor and the other for the body.

The final display of the actor is a composite where the head and the body of the actor are free to move independently. For instance, the arm of the gesturer can follow the mouse cursor as it moves around the screen while the head can be positioned to look in different directions or to perform different actions, e.g., nodding. The C++ class definition for the gesturer is as follows:

```
class Gesturer : public VideoActor {
public:
    void   PointTo(int x, int y);
    void   LookAt(int x, int y);
    void   HeadNod( );
    void   HeadShake( );
};
```

The PointTo method finds the posture where the hand is closest to the indicated position, it then invokes the VideoActor::Pose method. Similarly, the LookAt method is used to find the posture with the head looking in the direction of the x, y coordinates. The HeadNod and HeadShake methods request the actor to "perform" (using the VideoActor::Perform method) actions that picture the head either nodding or shaking, respectively. This example is a possible implementation for the video actor pictured in Figure 2.

Video Acquisition

The video material used as layers of an actor must be carefully prepared if the composed video is to appear realistic. In our implementation, video overlay was performed by chroma-keying, an effect that makes certain color values (typically shades of blue) transparent and allows the background to be visible. The use of chroma-keying has two consequences. The color values made transparent have to be as uniform as possible and they must not appear in the foreground. A uniform background was achieved by using paint or cloth in a specific color and most importantly, by lighting that produced little shadow.

The actors appearing in the video required careful rehearsal since they had to look and point in directions where objects appear only after the video is overlaid. We also recorded in a variety of languages (six) so we could experiment with multilingual interfaces.

After shooting, the material was edited to select the best sequences, the exact start and stop points, or to enhance the videos, e.g. by adding graphics into the video. We added, for example, graphic symbols for the fax assistant to highlight fax buttons or regions that are explained. The edited material was copied to a videodisc or stored in digital form. In some cases, however, additional processing of the video material was necessary. This was the case for the gesturer actor, where frames were edited in digital form to separate body and arm positions from head movements.

CONCLUSION

Video widgets offer the possibility of adding dynamism to the user-interface. We have illustrated one type of video widget, the video actor, and have discussed the implementation of video actors and their use in application programs.

One natural question is why use video to implement video widgets when real-time animation would offer more flexibility. Video certainly has fewer degrees of freedom than animation. Nonetheless, video does have some important advantages. First, video is more realistic. We still need video if we want to have real people on our screens rather than animated characters. Second, video players can playback synchronized audio; this may not be the case with "animation players". Third, there are software-based digital video players, while real-time animation often requires expensive hardware. And finally, video is easier to obtain – it's often less work to shoot a few minutes of video than construct a few minutes of animation. Although real-time animation is more interactive and easier to modify than video, we believe that some of the deficiencies of traditional video can be reduced by using keying and other multi-layering techniques.

Our implementation of video actors is still in the experimental stage. We conclude by describing some of the technical problems raised by the initial implementation.

video quality A problem with our particular hardware platform is that the application and video actor are viewed on an NTSC video monitor rather than the workstation monitor. The resulting loss of resolution, and other problems with image quality (flashing when the videodisc seeks, halos around objects due to poor keying) are artifacts of this particular platform. By using digital video sources better quality should be obtained, and the platform can be reduced to a single workstation.

video processing Currently if an application window is moved, the video actor is not translated and the sense of cohesion between the application window and the actor is lost. For example, if the panel shown in Figure 8 is shifted the actor will miss the button. A partial solution is to extend the VideoActor interface and allow spatial positioning of video layers. In other words real-time video processing, in this case 2D translation, is required. Current video mixers and video effects processors are capable of applying 2D transformations (e.g., scaling, rotation and translation), 3D transformations (e.g., perspective viewing and "wrapping" a video layer over a 3D surface) and composition operations (e.g., chroma-keying, fades, wipes, etc.) to multiple video streams in real-time. An intriguing question is how to abstract these capabilities in software allowing the creation of highly visual and fluid user interfaces.

ACKNOWLEDGEMENTS

We would like to acknowledge the support of the Swiss National Research Foundation (Fonds National de la Recherche Scientific) under project number 20-33703.92, the Austrian National Science Foundation (Fonds zur Förderung der wissenschaftlichen Forschung) under contract number J0770-PHY, and Digital Equipment Corporation through EERP contract number CH-023. In addition we thank the anonymous referees for many useful suggestions.

REFERENCES

1. Adelson, B. Evocative Agents and Multi-Media Interface Design. *Proc. CHI'92*, 351-356.
2. de Mey, V. and Gibbs, S. A Multimedia Component Kit: Experiences with Visual Composition of Applications. *Proc. ACM Multimedia Conf.*, 1993.
3. Don, A., Oren, T., and Laurel, B. Guides 3.0. In *CHI'91 Video Proceedings*, 1991.

4. Feiner, S. and McKeown, K. COMET: Generating Coordinated Multimedia Explanations. *Proc. CHI'91*, 449-450.
5. Hoffert, E., et al. QuickTime: an extensible standard for digital multimedia, *Proceedings of the IEEE Computer Conference* (CompCon'92), February 1992.
6. *Project 2000–A Knowledge Navigator*, Apple Computer, 1988 (videotape).
7. Krueger, M.W. *Artificial Reality II*. Addison-Wesley, Reading MA, 1991.
8. Le Gall, D. MPEG: A Video Compression Standard for Multimedia Applications. *Commun. of the ACM*, Vol. 34, No. 4 (April 1991), 46-58.
9. Oren, T., Salomon, G., Kreitman, K. and Don, A. Guides: Characterizing the Interface. In B. Laurel, Ed., *The Art of Human-Computer Interface Design*. Addison-Wesley, Reading MA, 1990, 367-381.
10. Ripley, G.D. DVI – A Digital Multimedia Technology. *Commun. of the ACM*, Vol. 32, No. 7 (July 1989), 811-822.
11. Wallace, G.K. The JPEG Still Picture Compression Standard. *Commun. of the ACM*, Vol. 34, No. 4 (April 1991), 30-44.

Appendix A: Hardware Platform

The components shown in Figure 4 and Figure 5 are abstractions – they encapsulate hardware and software resources. The connections between components are also abstractions and are realized by a variety of mechanisms (e.g., physical cables, message passing). Figure 9 shows the actual hardware and communication paths used to implement the component network of Figure 5. The central part of this layout is a 16x16 video routing switch allowing video equipment to be connected under computer control. The TBCs (time-base correctors) synchronize video signals against some reference signal (coming here from a sync generator) and are needed when video signals are mixed. The video switch and video mixers are configured so that the workstation display is placed at the lowest (background) layer followed by two layers of a video actor. The actor's layers originate from a QuickTime server (see Appendix B) and a videodisc player.

Appendix B: QuickTime Server

QuickTime is an extension to the Macintosh system software providing a standard way of supporting dynamic media such as video and audio [5]. At the highest level of abstraction, a QuickTime movie supports the synchronized presentation of several audio and video tracks. A movie toolbox allows applications to create, playback and edit movies.

We have implemented a QuickTime server and a QuickTimePlayer class which communicates, through TCP/IP, with the server running on a Macintosh. The server uses the movie toolbox, so movies are displayed on a Macintosh screen. A board on the Macintosh is used to obtain a NTSC video signal of the screen, this video signal is then used as input to other video components. The QuickTime player component (QT in Figure 5) encapsulates the server. It is accessed through an instance of the QuickTimePlayer class, a subclass of the abstract VideoPlayer class.

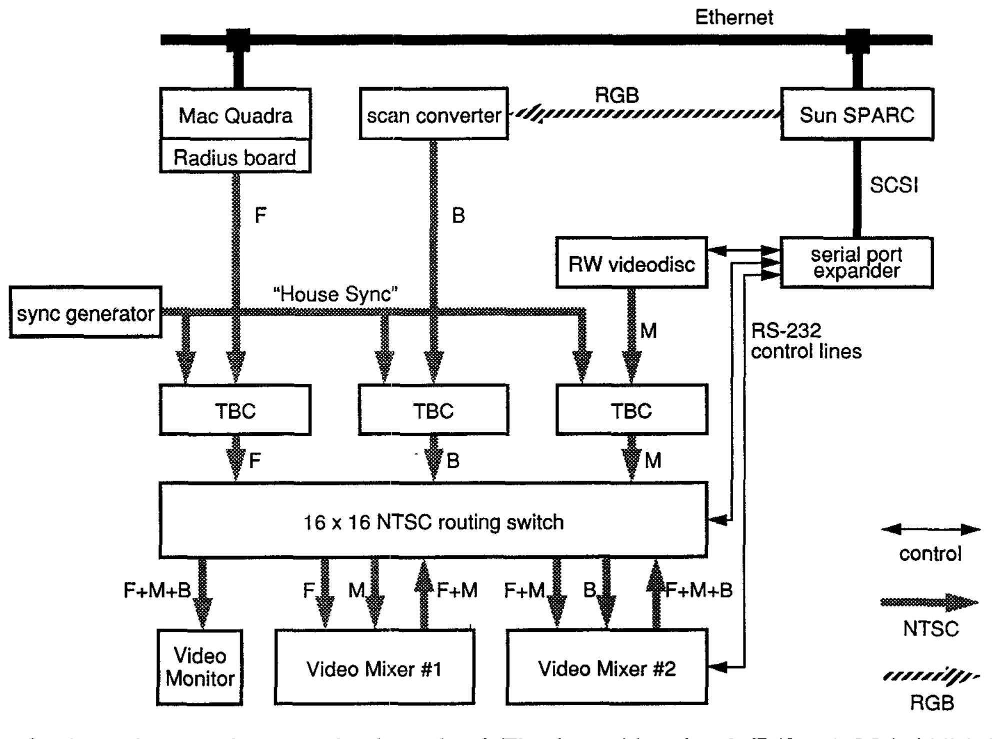

Figure 9: A two layer analog actor: hardware level. The three video signals F (front), M (middle), B (back) come from the two actor layers and the application respectively.