Exploring New Uses of Video with VideoSpace

Nicolas Roussel

DIUF, Université de Fribourg
Chemin du Musée, 3
1700 Fribourg, Switzerland
nicolas.roussel@unifr.ch

Abstract. This paper describes videoSpace, a software toolkit designed to facilitate the integration of image streams into existing or new documents and applications to support new forms of human-computer interaction and collaborative activities. In this perspective, videoSpace is not focused on performance or reliability issues, but rather on the ability to support rapid prototyping and incremental development of video applications. The toolkit is described in extensive details, by showing the architecture and functionalities of its class library and basic tools. Several projects developed with videoSpace are also presented, illustrating its potential and the new uses of video it will allow in the future.

1 Introduction

Although almost forty years have passed since AT&T's first PicturePhone, most commercial systems and applications dealing with video communication today are still based on a phone paradigm. This paradigm usually implies a formal and explicit setup of the link by both parties and the display of the images "as-is" on a monitor or in a window on a computer screen. Over the last twenty years however, the HCI and CSCW research communities have proposed a number of innovative uses of video including support for informal communication [1] as well as highly focused collaborative activities [2], or image processing for human-computer interaction [3].

In this paper, I present videoSpace, a software toolkit designed to facilitate the use of image streams to support such new forms of human-computer interaction and computer-supported collaborative activities. VideoSpace is motivated by the desire to focus on the *uses* of video, rather than the *technologies* it requires. In this perspective, the toolkit is not focused on performance or reliability issues, but rather on the ability to support rapid prototyping and incremental development of video applications. This approach contrasts with many of the research themes usually associated to video in the Multimedia or Network communities such as compression, transport or synchronization. VideoSpace is not aimed at these topics. It is rather intended to help HCI and CSCW researchers who want to explore new uses of the images.

VideoSpace is designed after A. Kay's famous saying: "simple things should be simple, complex things should be possible". It provides users and developers with a set of basic tools and a class library that make it easy to integrate image streams

within existing or new documents and applications. The tools, for example, allow users to display image streams in HTML documents in place of ordinary static images or to embed these streams into existing X Window applications. Creating a video link with the library requires only a few lines of code; managing multiple sources and including video processing is not much more complicated. Since the image streams managed by videoSpace often involve live video of people, the toolkit also provides a flexible mechanism that allows users to monitor and control access to their own image.

The paper is organized as follows. After introducing some related work, I describe videoSpace by showing the architecture and functionalities of its library and basic tools. I then present several projects based on the toolkit that illustrate its potential and the new uses of video it will allow in the future. Finally, I discuss some lessons learned from this work and conclude with directions for future research.

2 Related Work

Prototyping has been recognized as an efficient means of developing interactive applications for some time [4]: iterative design promotes the refinement and optimization of the envisioned interaction techniques through discussion, exploration, testing and iterative revision. However, exploring new uses of video through prototyping is hard. Researchers are faced with multiple difficult problems such as the need for digitizing hardware as well as specialized encoding/decoding algorithms and communication protocols. Moreover, evaluating a solution to any of these problems usually means having some solution to all of them.

Many innovative works on the use of image streams overcome these problems by using specific video hardware. Early Media Spaces, for example, were based on analog audio/video networks [5, 6, 7]. ClearBoard [2] also uses an analog video link and dedicated video overlay boards to superimpose two image streams in real-time. Likewise, Videoplace [8] relies on dedicated hardware for image processing. Although these specific hardware solutions allow researchers to focus on the interactions and not the technology required to implement them, they are usually expensive, hard to setup, hard to maintain and sometimes even hard to reproduce.

Specific video hardware allows to create fully functional, high-fidelity prototypes. But high-fidelity prototyping is not good for identifying conceptual approaches, unless the alternatives have already been narrowed down to two or three, or sometimes even one [4]. Low-fidelity prototypes have proved useful to narrow these alternatives. Pen and paper, painting programs or other simple electronic tools can be used to get feedback from potential users from the very early stages of product development. Software toolkits such as Tcl/Tk or GroupKit [9] have long been used for rapid prototyping and iterative development of graphical interfaces and groupware applications. The more recent advent of Web-based applications also promotes the use of quickly hacked HTML interfaces that can be easily modified to explore alternative designs.

The motivation for creating videoSpace resides in the lack of such flexible software tools for exploring new uses of video through the creation of high-fidelity as

well as low-fidelity prototypes. Most modern operating systems provide software libraries to manipulate image streams, such as Apple QuickTime, Microsoft DirectX or SGI Digital Media Libraries. These libraries have all their own advantages and disadvantages, but they are usually platform-dependent and incompatible with each other. These characteristics make them difficult to use for building the complex - and usually distributed - applications required to explore new uses of video. Although I realize the importance of the low-level services offered by these libraries, I believe that the HCI and CSCW communities would benefit from a higher-level software platform for image streams manipulation.

The Mash streaming media toolkit [10] and some other platforms developed by the Multimedia and Network research communities offer high-level video digitizing and transmission services. However, these platforms naturally tend to focus on the transmission techniques and usually rely on the idea that images are to be displayed "as-is", as big and as fast as possible. Although this conception is well suited to applications such as videoconferencing, tele-teaching or video-on-demand, it is too restrictive for more innovative applications that might involve image processing or composition at any point between its production and final use.

The Java Media Framework (JMF) is probably the existing platform closest to videoSpace. It provides programmers with a set of high-level classes to digitize, store, transmit, process and display images. However, it is a closed product of a commercial organization. At the time of this writing, for example, JMF supports video digitizing on Microsoft Windows and Sun Solaris platforms only. Since its source code is not publicly available, implementing this feature on other platforms requires writing separate extensions and possibly rewriting some existing code. Moreover, correcting bugs or implementing new features is a privilege of a few people whose interests might differ from those of the HCI and CSCW research communities.

VideoSpace grew out of my numerous experiences and frustrations in writing applications that digitize, synthesize, transmit, store, retrieve, display, modify or analyze image streams. By moving the common elements of these applications into a library and providing tools such as a network video server, I can now quickly create video applications whose complexity shrunk from several thousand lines of code to only a few hundred lines. This, in turn, facilitates the exploration of a wider range of uses of the image streams to support human-computer interaction and distant collaboration.

3 The VideoSpace Library

The videoSpace library was initially developed in C++ on SGI workstations. In its current state, it consists of less than a hundred classes and 15000 lines of code. The hardware and system dependent code is clearly separated from the rest of the code, which allowed us to easily port it to Linux ans Sun Solaris and should facilitate port to other systems. The library and several videoSpace applications were indeed successfully ported to Apple MacOS in September 1999, although this development branch was later abandoned. In addition to the primary C++ library, some of the core services of videoSpace are also available from Python scripts, through a dynamic extension, and some others have been re-implemented in pure Java. Integration with

Tcl/Tk is also provided through a specificity of the X Window system that will be detailed in the next section.

3.1 General Overview

The videoSpace library is built around the concept of *image*: it provides developers with classes and functions to *produce*, *process*, *transmit*, *display* and *record* images (Fig. 1) and to *multiplex* these basic operations.

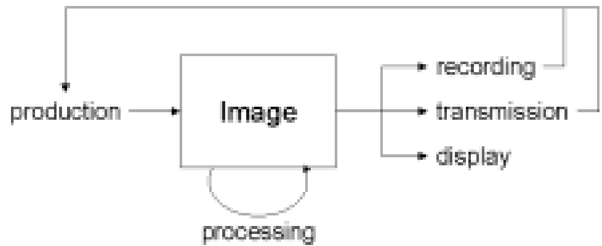

Fig. 1. Typical life-cycle of a videoSpace image

VideoSpace images are produced by local or network-accessible *image sources*. An image source may digitize or synthesize in real-time the images it produces. It may also retrieve them from a pre-recorded stream. All image sources are described by a URL that defines them in a unique way. The operation mode of a particular source can be adjusted through the addition of query string parameters to its URL. Such parameters can be used, for example, to specify the frame-rate, the size or the quality of the images produced.

VideoSpace supports real-time processing of images through *filters* that transform or analyze them. Transformation filters modify the dimensions, the data or the encoding of the images whereas analysis filters only extract valuable information from them. The library provides a number of filters for both transforming and analyzing images. Some of them allow to convert images between the different encodings available and to resize them. Others are used for gamma correction or convolution by a 3x3 kernel. Others to paint rectangular regions, to insert an image in another one or to superimpose two images. More complex filters such as chroma-keying, image difference and basic motion detection are also provided.

Image sinks are used to transmit images on the network, to display them on the computer screen or on an analog video output (e.g., a separate monitor) and to record them into files. Like image sources, image sinks and their operation mode are described by URLs. As we will see in the following sections, this use of text-based descriptions for sources and sinks allows to specify them at run-time, making it possible to create simple yet powerful applications for manipulating the image streams.

Combining an image source with a sink may result in complex execution flows. For example, if an image coming from a remote source has to be displayed on an analog video output, the application has to listen to the network connection to get the image data and, at the same time, it has to make sure that the hardware is ready to

display it when it is completely decoded. The use of multiple image sources or sinks makes this execution flow more complex. Interactive user interfaces add even more event sources to monitor, increasing the complexity further. In order to reduce this complexity, the videoSpace library provides simple mechanisms based on class inheritance and method overriding for multiplexing objects that deal with files, network connections or hardware devices.

3.2 Implementation Details

Image, pixel encodings and memory management. The `Image` class implements simple data structures that describe the width, height and pixel encoding of a rectangular bitmap and point to the memory location of the corresponding data. The following encodings are supported: L - for luminance -, RGB, ABGR, RGBA, Y'CbCr 4:2:0 and JPEG. In order to facilitate memory management between successive manipulations, the memory location of an `Image` can not be modified without explicitly stating what should happen to the new buffer when it is lately replaced. This information, stored along with the pointer, allows a single `Image` object to successively use different memory areas that are automatically de-allocated when not needed anymore.

Network support. The videoSpace library provides several classes for sending and receiving UDP datagrams - unicast or multicast -, creating TCP servers and clients and decoding HTTP messages. These classes allow videoSpace applications to exchange any kind of data, including images. Four network protocols for image streaming have been implemented on top of them: Netscape's HTTP Server-Push extension [11] applied to a series of JPEG images, the client side of the RFB protocol[1], and VSMP and VSTP, two proprietary protocols.

VSMP is based on UDP and can be used for one-to-one (unicast) or one-to-many (multicast) video transmissions. Images are JPEG-encoded so they can fit in a single datagram and a "best effort" strategy is used: lost datagrams are not retransmitted. VSTP was designed for client-type applications that request video from a server. It combines a VSMP transmission of images with an HTTP connection used as a signaling channel: the client sends on the HTTP connection the information required to start the VSMP transmission, i.e. the local host name and a UDP port number, and closes it to stop the transmission.

Although VSMP and VSTP were designed for simplicity and not performance or reliability, their performance level is quite acceptable: over the last three years, they have been used to transmit video streams between France, The Netherlands, Denmark, Germany, Switzerland and Austria at up to 20 QCIF images per second

[1] The Remote Frame Buffer (RFB) protocol was developed by AT&T for their Virtual Network Computing (VNC) project [12]. It allows thin client applications to display real-time images of a remote X Window, Apple MacOS or Microsoft Windows desktop server and to send mouse and keyboard events to this server.

(176x144 pixels) and 10 CIF images per second (320x240 pixels) with a latency of less than half a second and no perceivable image loss.

Image production. The videoSpace library defines an abstract `ImageSource` class with methods for starting the image production, getting the next or the most recent image available[2] and stopping the image production. Several classes derived from `ImageSource` implement the actual source types supported by the library. A factory function allows the run-time creation of a generic `ImageSource` object from the appropriate derived class, given a URL and the desired encoding.

The URL describing a hardware video input on the local machine specifies the device and the actual input used on this device (e.g., analog input number two on digitizing board one). Two generic names, `anydev` and `anynode`, can be used as default values for portability. Three optional query string parameters (`zoom`, `length` and `pause`) control the size of the images, their number and the time between two subsequent ones. A fourth one allows users to indicate whether the hardware resource should be locked (`locked=1`) or whether other applications can preempt it when needed.

URLs corresponding to network sources specify the remote host name or address (or group address, in the case of multicast), possibly followed by a TCP or UDP port number, some path information and some query string parameters. In addition to the four image streaming protocols we mentioned, videoSpace supports to some extent the real-time capture of a window on the computer screen through the X protocol. It also supports two types of pre-recorded image sources, one for single image JPEG files, and the other for files containing JPEG image streams produced by the Server-Push protocol. The following URLs illustrate and summarize the image sources currently implemented:

`videoin:/anydev/camera?zoom=3`	local digitizing hardware
`http://host:5555/push/video`	HTTP server using the Push extension
`rfb://host:1`	VNC server
`vsmp://host:9823`	videoSpace app. using VSMP unicast
`vsmp://225.0.0.252:5557`	videoSpace app. using VSMP multicast
`vstp://host/video?pause=1`	videoSpace app. using VSTP
`xwindow://localhost:0/0x1c0000e`	X window (experimental)
`file:/tmp/test.jpg`	single JPEG image
`file:/tmp/demo.vss`	JPEG stream in the Server-Push format

[2] The most recent image might not be the next one if the source implements some buffering algorithm.

Image processing. Image filters can be implemented as functions, or as classes that associate data and possibly state-based transitions to the processing algorithm. In the latter case, a `SimpleFilter` class can be used as a base class to share a common syntax and allow the run-time specification of filters. Processing algorithms are usually implemented for a subset of the available encodings (e.g., RGB, RGBA and ABGR). Consequently, encoding conversion might be required before and/or after applying a filter.

Image transmission, display and recording. A pointer to the memory location of an `Image` data can be obtained through a `getData` method. Since the encodings supported by videoSpace are also supported by many low-level graphical or video libraries, its images can be easily manipulated with them. OpenGL, GTK or the Linux SVGA library, for example, can be used to display the images on the computer screen. On SGI O2 and Octane workstations with proper hardware, the Digital Media Libraries can also be used to send them to an external analog video device such as a monitor or a VCR. Recent additions to the library also provide some preliminary support for creating MPEG-1 and RealPlayer compatible video streams.

The videoSpace library defines an abstract `ImageSink` class and several derived classes to facilitate the transmission of images using VSMP, their display using several graphical toolkits, and their recording in several formats. As for image sources, a factory function allows the run-time creation of a generic `ImageSink` object from given a URL. The following URLs illustrate and summarize the image sinks currently implemented:

`vsmp://localhost:9823`	videoSpace app. using VSMP unicast
`vsmp://225.0.0.252:5557`	videoSpace app. using VSMP multicast
`glxwindow://localhost:0`	OpenGL display in an X Window
`gtkwindow://localhost:0`	GTK window
`svga:/640x480?centered=1`	SVGA full-screen display (Linux only)
`videoout:/anydev/anynode`	analog video output (SGI only)
`file:/tmp/capture.jpeg`	single JPEG image
`file:/tmp/capture.vss`	JPEG stream in the Server-Push format
`file:/tmp/capture.mpeg`	MPEG-1 stream (experimental, Linux only)
`file:/tmp/capture.rm`	RealPlayer stream (experimental, Linux only)

Multiplexing. The videoSpace library provides a `MultiplexNode` class for multiplexing low-level operations on files, network connections and hardware devices. The `multiplex` method of this class, based on the UNIX `poll` system call, suspends the execution of the application until a timer expires or some descriptor associated to a file, connection or device becomes readable or writable. `multiplex` relies on two other methods: `prepare`, that specifies the set of descriptors to watch and the time limit, and `check`, that defines how the object reacts to low-level state changes. Many classes of the library, including all image sources and sinks, derive from `MultiplexNode` and override these two methods to implement high-level *reactive objects*.

Every `MultiplexNode` object maintains a list of other `MultiplexNode` instances associated to it by the application developer. Calling the `multiplex` method of one object automatically calls the `prepare` and `check` methods of all associated instances in addition to those of the primary object. This allows developers to describe hierarchical structures of high-level reactive objects and to multiplex them in a single call. The following code example illustrates this by showing how to blend a local and a remote image stream and display the resulting images on an analog video output:

```
ImageSource *src1 = createImageSource(Image::RGB,
                    "videoin:/anydev/camera?zoom=2") ;

ImageSource *src2 = createImageSource(Image::RGB,
                    "vstp://remotehost/video") ;

ImageSink *dst = createImageSink("videoout:/anydev/anynode");

dst->addNode(src1) ; // Associate the two image sources
dst->addNode(src2) ; //    to the image sink

src1->start() ;
src2->start() ;
dst->start() ;

Image img1, img2, composite ;
bool newComposite = false ;

while (dst->isActive()) {

     dst.multiplex() ; // Multiplex the sink and the sources

     if (src1->getNextImage(&img1)
          || src2->getNextImage(&img2)) {
        // At least one image has changed
        //   update the composite
        blendImages(&img1, &img2, &composite) ;
        newComposite = true ;
     }

     if (newComposite) {
        // The composite stays "new" until handled
```

```
            newComposite = !dst->handle(&composite) ;
        }

    }
```

4 VideoSpace Basic Tools

In addition to the C++ library, aimed at developers, the videoSpace toolkit provides end-users with a number of tools that can be used off-the-shelf, with no or little programming, and serve as building blocks for more complex applications.

4.1 VideoServer

The main tool provided by videoSpace is videoServer [13]. VideoServer is a personal Web server run by the user of a workstation and dedicated to video: it is the unique point of access to that person's video sources. The three services it provides are:

1. creating a one-way live video connection;
2. retrieving a pre-recorded video file;
3. acting as a relay for another image source.

VideoServer services are mapped to resource names that are accessible through the HTTP protocol and can be described by simple URLs. Video data itself can be transmitted with the HTTP Server-Push or VSTP. The following URL, for example, requests a live video stream with 1 frame every 60 seconds to be sent with HTTP Server-Push (5555 is videoServer's default port number):

```
http://host:5555/push/video?pause=60
```

This URL requests a pre-recorded video file to be sent with VSTP to a client listening on `desthost` on port 9257:

```
http://host:5555/vstp/movie/demo.vss?host=desthost&port=9257
```

This third example illustrates the use of videoServer as a relay for another source, in this case an RFB server. The URL describing the source relayed, `rfb://srchost:1`, has to be encoded in order to be used in the query string of the request:

```
http://host:5555/push/relay?src=rfb%3A%2F%2Fsrchost%3A1
```

Using custom HTTP servers to provide video services is not new. A number of Webcams available on the Internet work this way, and in fact videoSpace users often use them as image sources. However, videoServer differs from these Webcams on a major issue: it provides its user (i.e., the person who runs it) with access control and notification mechanisms to support privacy. For every request it receives, videoServer executes an external program, the *notifier*, with arguments indicating the name of the remote machine, possibly the remote user's login name, the resource that led to the server - the HTTP referrer - and the requested service. In response, the

notifier sends back to the server the description of the service to execute, which can differ from the one the client requested.

The default notifier, a UNIX shell script, allows users to easily define access policies. Low quality or pre-recorded video, for example, can be sent to unidentified users while known people get a high quality live video stream. In addition to computing the service to execute, the notifier can trigger several auditory or graphical notifications to reflect some of the information available, such as the identity of the remote user or the service requested. When a live video request does not specify the number of images, videoServer limits it to 5000, that is, up to three minutes. This ensures that constant monitoring cannot take place without periodically asking permission and thus triggering notifications. All these elements facilitate the acceptance of videoServer by the users and helps finding a trade-off between accessibility and privacy.

4.2 VideoClient

VideoClient started as a simple lightweight application designed to display videoSpace streams on the computer screen, the video being scaled to match the window size [13]. As the class library evolved, it became a more generic tool for easily filtering video streams and displaying, recording, or sending them on the network. The current implementation of videoClient is about 100 lines of C++ and allows to specify at run-time an `ImageSource`, a `SimpleFilter` and an `ImageSink`. For example, videoClient can be used for:

1. recording images from local hardware into a file

```
videoClient -i videoin:/anydev/anynode \
            -o file:demo.vss
```

2. multicasting the recorded sequence after applying a difference filter

```
videoClient -i file:demo.vss \
            -f difference \
            -o vsmp://225.0.0.252:5557
```

3. displaying the multicasted stream in an X window using OpenGL

```
videoClient -i vsmp://225.0.0.252:5557 \
            -o glxwindow://localhost:0
```

In addition to top-level windows which can be manipulated through the window manager, the `glxwindow` image sink can take advantage of the architecture of the X Window system to display video in a new subwindow of an existing one. This can be done by simply specifying the id of the parent window:

```
videoClient -o glxwindow://localhost:0?parent=0x120000e
```

This feature of `glxwindow` can be used to "augment" existing X Window applications with video streams. Since the video window is a child of the host window, it is moved, raised, lowered and iconified with it. This way for example, one can easily add video to a text-based chat application. This approach can also be used with user interface toolkits that explicitly support widgets that host external applications. For

example, the frame widget of the Tcl/Tk toolkit can host a separate application by setting the widget's *container* property to true. The sample code below illustrates the use of videoClient to embed a video stream in a Tk frame:

```
# create a frame widget for the video
frame .video -container true -width 160 -height 120
set id [winfo id .video]
# launch a videoClient inside the frame
exec videoClient \
            -i vstp://host/video \
            -o glxwindow://localhost:0?parent=$id &
# make the frame visible
pack .video
```

4.3 Other Video Tools

In addition to videoServer and videoClient, videoSpace provides users with a number of other tools to manipulate, display or convert video streams. One of these tools, for example, allows to change the width and height of a stream. Another one can combine several streams in a single one using a mosaic placement. A simple UNIX Shell script also allows users to "stick" a videoClient on any existing X Window using the parent parameter of the `glxwindow` image sink.

5 Exploring New Uses of Video

In this section, I present several projects that were developed with videoSpace and benefitted from its flexibility and extensibility. The details of these projects unfortunately fall beyond the scope of this paper. Some have been published and some others will be. What I want to illustrate here is how the variety of image sources provided by videoSpace combined with its facilities for transmitting, displaying and recording video streams support the rapid prototyping and incremental development of video applications. How videoSpace facilitates the exploration of new uses of video.

5.1 Using Documents as Interfaces to Awareness and Coordination Services

Over the last few years, I have been exploring the use of Web-based video environments to provide awareness and coordination services to distributed groups of people [13]. Inspired by previous work on Media Spaces, I have used videoSpace to promote the development of collaborative environments in which video communication facilities are embedded into the existing environment of the users, i.e. their documents and applications, rather than provided as separate applications.

Most Web browsers can display JPEG images and some of them can also display an HTTP Server-Pushed JPEG stream in place of an ordinary image without any plug-in. Combined with these browsers, videoServer allows to include live or pre-recorded video streams in HTML documents by using code such as:

```
<img src="http://host:5555/push/video">
```

By including such references to videoServers, one can easily create dedicated interfaces to the video communication space, such as the awareness view of Fig. 2, but also “augment” existing documents by embedding video services in the existing content. For example, one can include a video link in an e-mail message so the receiver will see the sender's office when he reads the message. When cooperatively editing an HTML document, the authors can also include live views of their offices in the document so they can see if their co-authors are present when they work on it.

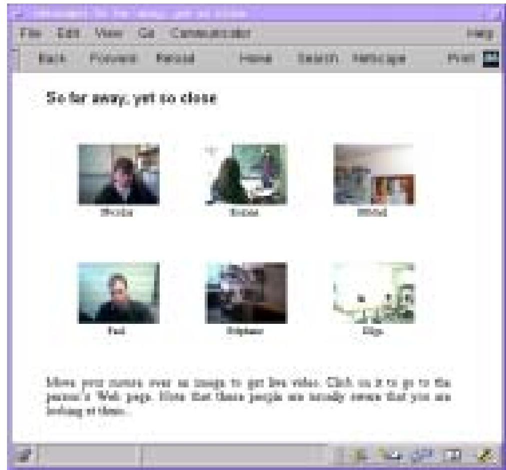

Fig. 2. HTML-based awareness view showing several videoServers

HTML-based interfaces can be easily shared and exchanged. As more and more people are getting familiar with HTML authoring, they are also easily tailorable. Using HTML and videoServers to create a Web-based video communication environment emphasizes the design principle that “simple things should be simple”. VideoServer is indeed in daily use in several places around the world. End users have created HTML interfaces to it and have developed usage patterns without any knowledge of the architecture of the system.

5.2 Designing a New Communication Device to Support Teleconviviality

Videoconferencing systems are getting closer to technical perfection every year, making them more and more pleasant to use. However, the usual setting of these systems favors a face to face between all the participants of each site, which makes cross-site informal conversations difficult: before and after a formal meeting or during pauses, people tend to discuss with some of their co-located partners and often ignore the remote people. I strongly believe that one of the keys to informal communication in these situations is in the ability for people to move away from the central focus point of the formal meeting and gather in small cross-site groups isolated from each other. Starting from this idea, several research partners and I designed a new communica-

tion device, *le puits* [14] (the well), that combines an SGI O2 workstation with microphones, cameras, speakers and a horizontal video projection system to establish an audio and video link between distant groups of up to 6 people (Fig. 3).

Fig. 3. *Le puits*: hardware prototype and sample display configurations. The two rightmost configurations add pre-recorded simulated views to the real views from the prototype

The design of *le puits* required several iterations from the initial sketches to fully functional prototypes. What is interesting to mention here is that videoSpace allowed to develop the software used to compose the projected images several hundred kilometers away from the place where the hardware parts were actually assembled. Prerecorded video streams showing a rough view of what the cameras would see were used to experiment with different composition methods even before the first prototype was built. Network sources were later used to simulate analog video cables and proprietary codecs between two prototypes. The mosaic composer application was also used to simulate its analog equivalent when it was decided to use such a device to put all the cameras of a prototype on a single video stream. In addition to supporting rapid prototyping of the software and simulation of various hardware configurations, videoSpace allowed us to develop the software required for *le puits* on a laptop running Linux and to later recompile it without a single modification and run it on the SGI O2s of the prototypes.

5.3 Using the Hand as a Telepointer

Gesturing is a natural means of communication for humans. Hand gestures in particular are often used to express ideas, to refer to objects, to attract attention or to signal turn taking. To recreate this communication channel over distance, real-time groupware systems usually display telepointers that participants can move over the shared view. However, standard telepointers usually lack semantic information. At best, they are chosen among a predefined set of shapes and/or colors, which makes it hard to draw attention, to designate several objects at the same time or to express an idea. A mouse cursor is a very poor substitute for the hand for gesture communication, and some colleagues and I thought that the image of the hand itself, captured in real-time,

would do a better telepointer. We set up a camera above a desk covered by a large blue sheet of paper and recorded several video sequences showing some hand gestures over this solid-color background. We then developed a chroma-keying filter to extract the image of the hands from these sequences and several prototypes to display them over other image streams or running applications [15].

The first prototype superimposed the chroma-keyed video stream over a screenshot of a supposedly shared application. This first implementation gave us some interesting hints for further tests. We realized, for example, that the chroma-keying process allowed us to annotate the shared view with real world objects. We implemented a second prototype in which the user could control the size, the position and the transparency of the chroma-keyed overlay with the mouse and the keyboard. This second prototype used the same “parent window” trick as videoClient to overlay the chroma-keyed video stream on a running application. This solution, however, couldn't be applied to more than one application at the same time and was too specific to the X Window system. Instead of trying other complex ways to superimpose our chroma-keyed video stream over a traditional computer desktop, we simply used an RFB image stream as the background of a third prototype (Fig. 4).

Fig. 4. Using the hand as a telepointer over an RFB image stream

It is still too early to claim that the image of the hands are indeed better than a traditional telepointer. However, again, this project emphasizes the rapid prototyping and iterative design paradigm supported by videoSpace, from the development of the chroma-keying filter using pre-recorded sequences to the implementation of several alternative designs using different image sources. As the chroma-keying filter was integrated into the videoSpace library, the complexity of the prototypes was reduced to the minimum: each of them is only 200 lines of C++ code.

5.4 Exploring New Desktop Interactions

As illustrated by the previous example, using the images of computer desktops in videoSpace applications offers some interesting perspectives for exploring new interactions or metaphors for these desktops. I am currently developing videoDesktop, a new videoSpace application that acts as a window manager for an X Window server accessible through RFB (an Xvnc server). As a window manager application, videoDesktop knows the dimensions of all the application windows. By using a simple tiling placement algorithm (Fig. 5), it is able to extract the images of individual applications from the RFB image stream and use them as textures to compose a new desktop view.

Fig. 5. Sample view of the tiled desktop image stream

Figure 6 shows an example of graphical re-composition of the applications shown in Fig. 5. This composition scales, rotates and translates every application and adds some transparency and a simple shadow. In its current state, videoDesktop allows to send keyboard events to the applications based on a click-to-focus policy. Mouse event coordinates would have to be transformed before sending the event to the Xvnc server. However, this transformation is not currently implemented.

The technique used by videoDesktop is quite similar to the *redirection mechanism* of the Task Gallery [16]. However, whereas Task Gallery requires a modified version of Windows 2000, videoDesktop is only 800 lines of code and requires only a standard Xvnc server. Although this project is still in a very early stage, it already allows to use a modern graphical toolkit such as OpenGL to experiment new window managing techniques that were until now relying on graphical models from the 80s or 90s.

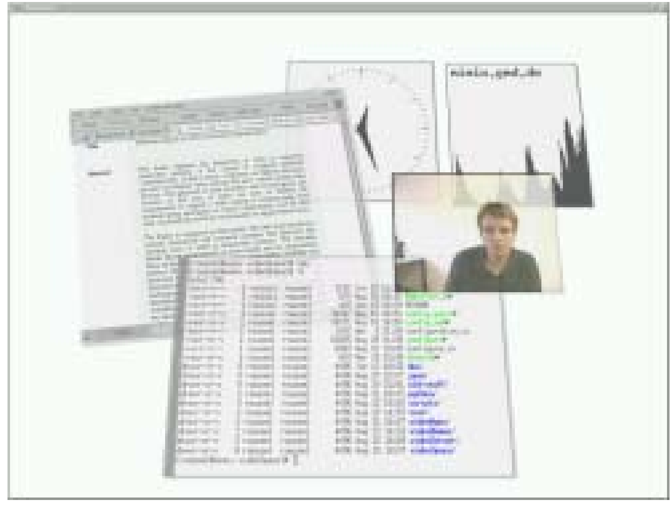

Fig. 6. View composed by videoDesktop from the tiled desktop image stream of Fig. 5

6 Discussion

The library and tools composing videoSpace are the result of an iterating process of design and use. As the library was getting more complex, the tools were getting simpler. As the level of abstraction of the classes was increasing, the number of lines of code of each tool was decreasing. In this section, I discuss several aspects of the toolkit that contributed to this evolution and its success.

One of the early key choices was the separation between the `Image` class, used as a simple data container, and the sources, filters and sinks that operate on the data. The implementation of these image operators as independent classes instead of methods of the `Image` class increases the flexibility of the toolkit and facilitates its extension. Every application can easily define its own operators that can be created and modified at run-time. If these new operators prove useful for other applications, they can then be integrated into the core library.

The JPEG encoding and the HTTP Server-Push protocol were initially chosen for their ease of use and implementation, not their performance. The same desire of simplicity inspired the design of VSMP and VSTP. However, videoSpace is not restricted to these simple choices. New encoding or protocols can be easily added to the library to overcome performance problems or to communicate with other applications. Adding an encoding simply consists in adding a constant to the list of supported encodings in the `Image` class and providing a few conversion filters. Similarly, as illustrated by the `rfb` image source, adding a protocol simply consists in adding classes derived from `ImageSource` and/or `ImageSink` and changing the associated factory functions.

Another early key choice was the use of URLs to name video sources. When a new source type is added to the toolkit, the factory function is modified and all applications benefit from it without any other change. As explained in the previous section, local sources or pre-recorded streams are often used instead of network sources during development to test an application, to compare alternative designs or to evaluate performance. The use of URLs make it possible to switch between sources without even recompiling the application; therefore, applications can be developed on machines that do not have any video hardware or even a network connection. Another advantage of using URLs is the seamless integration with Web applications, most notably Web browsers.

7 Conclusion and Future Work

VideoSpace is a software toolkit designed with one specific goal in mind: facilitate the exploration of new uses of video. In this paper, I have described the class library and basic tools it provides. I have also presented several projects that illustrate its flexibility and its ability to support rapid prototyping and incremental development. VideoSpace has now reached a stable state and some of its applications are in daily use in several research labs. Its source code is publicly available for download[3]. Short-term development plans include the integration of additional video encodings and protocols such as MPEG-4. I would also like to experiment with filters for detecting, identifying and tracking objects or people, based on the techniques presented in [3]. Another direction for future work is the addition of high level functionalities related to session management, such as explicit support for bi-directional or multiple connections.

Acknowledgments

VideoSpace grew out of the many video applications I wrote while I was working at LRI - Université Paris-Sud on the Telemedia project, funded by France Télécom R&D. Thanks are due to Michel Beaudouin-Lafon, who contributed to many ideas presented in this paper. I am also grateful to Jacques Martin, Jean-Dominique Gascuel and the other people from CSTB and iMAGIS who contributed to the design and prototyping of *le puits*. Current work on videoSpace is supported by ERCIM, the European Research Consortium for Informatics and Mathematics, and the Swiss National Science Foundation.

[3] See http://www-iiuf.unifr.ch/~rousseln/projects/videoSpace/

References

1. W. Mackay. Media Spaces: Environments for Informal Multimedia Interaction. In M. Beaudouin-Lafon, editor, Computer-Supported Co-operative Work, Trends in Software Series. John Wiley & Sons Ltd, 1999.
2. H. Ishii, M. Kobayashi, and K. Arita. Iterative Design of Seamless Collaboration Media. Communications of the ACM, 37(8):83–97, August 1994.
3. J.L. Crowley, J. Coutaz, and F. Bérard. Perceptual user interfaces: things that see. Communications of the ACM, 43(3):54–64, March 2000.
4. J. Rudd, K. Stern, and S. Isensee. Low vs. high-fidelity prototyping debate. ACM interactions, 3(1):76–85, 1996.
5. M. Olson and S. Bly. The Portland Experience: a report on a distributed research group. International Journal of Man-Machine Studies, 34:211–228, 1991.
6. C. Cool, R.S. Fish, R.E. Kraut, and C.M. Lowery. Iterative Design of Video Communication Systems. In Proc. of ACM CSCW'92 Conference on Computer-Supported Cooperative Work, Toronto, Ontario, pages 25–32. ACM, New York, November 1992.
7. W. Buxton and T. Moran. EuroPARC's Integrated Interactive Intermedia Facility (IIIF): Early Experiences. In Multi-User Interfaces and Applications, pages 11–34. S. Gibbs and A.A. Verrijn-Stuart, North-Holland, September 1990. Proceedings of IFIP WG8.4 Conference, Heraklion, Greece.
8. M. Krueger. Artificial Reality II. Addison-Wesley, 1991.
9. M. Roseman and S. Greenberg. Building Real Time Groupware with GroupKit, A Groupware Toolkit. ACM Transactions on Computer-Human Interaction, 3(1):66–106, March 1996.
10. S. McCanne, E. Brewer, R. Katz, L. Rowe, E. Amir, Y. Chawathe, A. Coopersmith, K. Mayer-Patel, S. Raman, A. Schuett, D. Simpson, A. Swan, T-L. Tung, D. Wu, and B. Smith. Toward a Common Infrastructure for Multimedia-Networking Middleware. In Proc. of 7th Intl. Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV 97), May 1997.
11. An Exploration of Dynamic Documents. Technical report, Netscape Communications, 1995.
12. T. Richardson, Q. Stafford-Fraser, K.R. Wood, and A. Hopper. Virtual Network Computing. IEEE Internet Computing, 2(1):33–38, Jan-Feb 1998.
13. N. Roussel. Mediascape: a Web-based Mediaspace. IEEE Multimedia, 6(2):64–74, April-June 1999.
14. N. Roussel, M. Beaudouin-Lafon, J. Martin, and G. Buchner. Terminal et système de communication. Patent submitted for approval by France Télécom R&D (INPI n°00-08670), July 2000.
15. N. Roussel and G. Nouvel. La main comme télépointeur. In Tome 2 des actes des onzièmes journées francophones sur l'Interaction Homme Machine (IHM'99), Montpellier, pages 33–36, Novembre 1999.
16. G. Robertson, M. van Dantzich, D. Robbins, M. Czerwinski, K. Hinckley, K. Risden, D. Thiel, and V. Gorokhovsky. The task gallery: a 3D window manager . In Proc. of ACM CHI 2000 Conference on Human Factors in Computing Systems, pages 494–501. ACM, New York, April 2000.