Adding digital video to an object-oriented user interface toolkit.

S.M.G. Freeman[†],
Rank Xerox Research Centre, Grenoble, France. Steve.Freeman@xerox.fr

M.S. Manasse,
Digital Equipment Corporation, Systems Research Center, Palo Alto.
msm@src.dec.com

We have integrated digital video into Trestle, an object-oriented user interface toolkit written in Modula-3. The display of video frames is managed within the application process using, where possible, shared memory to transmit images to the window system. We took advantage of Modula-3's type system, lightweight threads and garbage collection to develop a flexible architecture that supports the reuse of image data within an application; the object-oriented features of Modula-3 we found most useful were inheritance, partial revelations, and encapsulation. We then integrated our video extension into several higher-level tools which allow us to dynamically experiment with video applications.

1. Introduction

This paper describes the experiences of adding digital video to Trestle [8], an object-oriented user interface toolkit written in Modula-3 [9]. The video infrastructure, based on hardware JPEG compression and decompression, uses the X shared memory extension [6] to display images at full-motion frame rates. The previous video library was implemented as a Motif widget and had three main problems. First, the single-threaded environment meant that the video pipeline had to be built as a state machine. In effect, the widget contained its own video scheduler and this was reaching the limits of maintainable complexity. Second, the closed structure of the widget inhibited the sharing of image data from the various stages of the video pipeline. A second window could not pick out frames from a video stream without either special code in the widget or a duplicate connection to the video source. Finally, and perhaps most important for us, Motif was (and remains) incompatible

[†] formerly at the Computer Laboratory, University of Cambridge, UK.

with Trestle, the Modula-3 user interface toolkit. We could not take advantage of the substantial body of tools and libraries available in the Modula-3 environment. In particular, the desktop video was to be used within Argo [7], a project to investigate the collaborative use of computers, for which most of the rest of the software is implemented in Modula-3.

This project, then, was to implement digital video within Trestle, with some consideration of the advantages and disadvantages of such Modula-3 features as an object-oriented type system, integrated with threads and garbage collection. We found that, with some care, use of these facilities greatly improved the structure of the software, making it more open and flexible, that integration with other Modula-3 packages was very easy, and that the performance costs were not excessive.

The next section describes the software and hardware infrastructure on which the project was built and the requirements we wished to support. We then describe the two main components of the project, the video library and the alterations to Trestle, and show how the video extensions fitted into other Modula-3 based tools. After that we distinguish those object-oriented features which most benefited the project and discuss related work. Finally, we draw some conclusions and outline further work to improve the software.

2. Background

2.1. A Trestle primer

Trestle is an object-oriented, hierarchical windowing toolkit, designed to support and exploit concurrent threads and garbage collection. The primitive window object is a VBT, or *virtual bitmap terminal*, which represents a share of the screen, mouse and keyboard. Each VBT is responsible for all the user events and painting that occur in a region of the screen. Some VBTs handle their responsibilities by passing them on to a parent or child. For example, an *HVSplit* tiles itself with children, either horizontally or vertically; a *TSplit* gives its space to exactly one of its children at a time; while a *ZSplit* divides its region into possibly overlapping subregions, and is responsible for ensuring that painting gets properly clipped to those regions. A *leaf* VBT, such as a text region or an image, has no children, and deals with input events as appropriate. Another common kind of VBT is a *filter*, which has only one child, and which usually wraps some small behaviour around the child; for example, a filter can be used to turn any VBT into a button or to put a border around a child.

A VBT is defined as an object with its behaviour determined by its methods. A new type of VBT may be created by subclassing an existing VBT type and overriding some of its methods, so a reaction to button clicks, for example, may be implemented by overriding a VBT's `mouse` method. This is a standard approach for many user interface toolkits, but Trestle carries it further than most. For example, there is a `paint` method that defines how a VBT handles paint requests from its children; the *JoinVBT* filter overrides this method to copy the output from its child to multiple parents.

Trestle also includes a sophisticated locking strategy to allow as much parallelism as possible within the toolkit. There is a global lock for the toolkit as a whole and a mutex to protect each VBT. Trestle defines an order on these mutexes so that a thread can lock the resources it needs without risking deadlock.

2.2. A J-Video primer

A J-Video board [4] provides hardware JPEG compression and decompression; it has input ports for audio and video, an output port for audio, and can read and write shared memory segments on a TurboChannel host. In this paper we concentrate on the video aspects of the board as, in our implementation, audio was managed by another software system.

Each host with a J-Video board runs a *jvdriver*, which manages low-level access to the board, and a *jvsource*, which supplies streams of compressed frames to clients. If a machine contains multiple boards, an instance of each of these services is run for each board. When a frame is required by a client, the jvsource asks the jvdriver to ask the J-Video board to grab a frame from the attached video source, digitize it, and compress it into a given shared memory segment. The jvsource server may then send the compressed frame to one or more clients via a socket; this is shown on the left-hand side of Fig. 1.

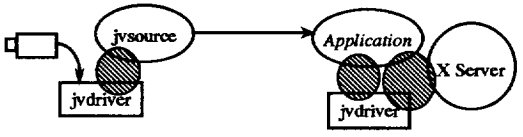


Fig. 1. The components of a J-Video connection. The shaded areas represent shared memory segments. The jvdriver consists of a J-Video board and a server process to manage it.

The client application receives the compressed frame and stores it in a shared memory segment, which it passes to its local jvdriver with the identifier of a second shared memory segment and other decompression parameters. The board decompresses the frame into the destination shared memory segment with the scaling and colour mapping specified by the client. The application then uses the X shared memory extension to pass the decompressed frame to the X server for painting; this is shown on the right-hand side of Fig. 1. The use of shared memory means that image data is copied only at operating system boundaries (to and from socket connections and, possibly, to the frame buffer) and allows near real-time frame rates (performance is discussed in Section 8).

2.3. Performance requirements

Fig. 2 shows a more abstract view of the image processing in a client application; frames are accepted from a server, decompressed and then displayed on the screen.

Our system is intended to support video conversations, so the latency between an application receiving a frame and displaying it is an important performance issue. We want each stage in the pipeline to be kept equally busy, so that stages neither have to wait for the previous stage to complete a frame nor process frames too quickly for the next stage to accept—timeliness requires that a frame must be thrown away once the next frame is ready, so unused frames represent wasted processing.

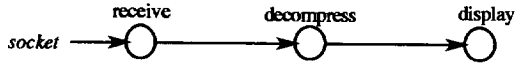


Fig. 2. The video pipeline in a client application. Each node represents part of the processing of an image.

In addition, we want different parts of an application to be able to share buffers. A user may want to watch video in a small-sized window, but also have another window showing a portion of that video magnified; it is more efficient if the two windows share the compressed image stream. A user may also want two windows showing the same image; for example, a small picture-in-picture window might have the same dimensions as a video-channel selector (although the frame rates differ). In this case, we want to share the same decompressed frames.

3. The JVideo library

The integration of video into Trestle involved two main components: a JVideo library to communicate with a jvsource and the local jvdriver and to manage frames as they are received and decompressed, and extensions to Trestle to support the painting of images and a shared memory extension (described in Section 4). A major difference from the previous J-Video client library is Modula-3 environment, with integrated types, user-level threads and garbage collection, which led to a cleaner and more flexible internal structure within the library. We were able to break up the monolithic Motif widget into a set of object types that represented features of the system. The periodic nature of video, consisting of discrete frames, allows its data to be held in a set of fixed-size buffers each of which may be managed by an object. Thus, once the incoming video stream has been broken up into frames, everything in the system is represented by an object.

The JVideo library has three main types of component: buffers, converters and buffer pools; the last two are represented by the circles and lines, respectively, in Fig. 2. Converters may be thought of, loosely, as active objects that receive messages (buffers) via a channel (buffer pool), process them, and then send the results out via another channel. We now describe these components in more detail.

3.1. Buffers and Pools

A *JVBuffer* object provides protected access to an image buffer and may also hold application-level data, such as a timestamp or the image dimensions. JVBuffers also have reference counting facilities so that threads may declare their interest in the contents of the buffer; this allows us to avoid a buffer being freed in one thread while another is still using it.

JVBuffers belong to *JVBufferPools*, which can hold up to some maximum number of buffers; the maximum can be set dynamically. Each pool has one *current* buffer, which holds the most recent frame for that stage in the pipeline. The identity of the current buffer may be changed by a *writer* thread as new frame data arrive from the previous stage. There is normally only one writing thread per buffer pool, but any number of readers may acquire a handle on the current buffer, incrementing its reference count so that it cannot be overwritten whilst still in circulation. A buffer that is replaced as the current buffer may still be in use by reader threads, so it is considered *pending*. The storage cannot be reused until its reference count drops to zero, at which point the buffer is now *free* and can be acquired by the writer thread (Fig. 3).

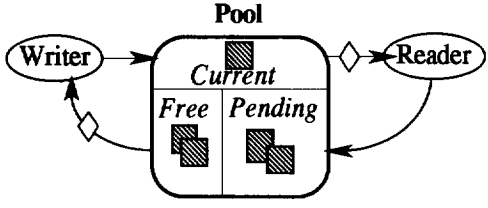


Fig. 3. A buffer pool. The left-hand diamond represents a *wait-for-free-buffer* condition, the right-hand diamond a *current-buffer-changed* condition.

This approach supports some of the features we require. First, the enforced maximum size of a pool means that a writer must wait for a free buffer if all the buffers are still held pending by reader threads. This simple flow control avoids having the writer continuously updating the current buffer when all the readers are still processing previous frames. The pool itself is also reference counted for the number of readers interested in receiving frames—there is no point in writing frames to a pool that has no readers. Second, the use of the current buffer means that a reader can always acquire the most recent frame; readers may also wait to be notified when the contents of the current buffer changes. Finally, the use of reference counting to protect buffers means that they can be shared by arbitrary numbers of readers without explicit synchronisation. In principle, the Modula-3 garbage collector could have been used to return buffers to the pool as the runtime provides access to the finalisation stage, but the shared memory segments are a sufficiently valuable resource to justify this more assertive technique. In particular, the collector we use is not guaranteed to find all garbage as it runs; references whose addresses match a bit pattern on the stack of some thread will not get cleaned. Our kernels

support only a few dozen shared memory segments of the size needed for video frames, so the collector would have to run almost continuously.

3.2. Converters

JVConverters, the nodes in the video pipeline, are objects that repeatedly receive a frame, process it, and pass it on to the next stage. So far we have implemented a *JVSink* type to connect to a source and read images from a socket, and a *JVDecomp* type to decompress images from a JVSink. The basic algorithm for the main loop in a converter, typically executed in a single thread, is simply:

```
LOOP
   outbuf := outpool.getFreeBuffer();
   (* wait for space in output buffer pool *)
   inbuf := inpool.waitForChange();
   (* get frame from previous stage in pipeline *)

   Convert(outbuf, inbuf);              (* do the conversion *)

   outpool.setCurrentBuffer(outbuf);
   (* set the current value of the output pool *)
   inbuf.free();
   (* return the input buffer to its pool *)
END;
```

Converters communicate with each other only via buffer pools (Fig. 4) so it is easy to share the output from each stage. For example, we create a separate JVSink object in an application for each connection to a given server which has different compression parameters, but these connections are cached within the library; this avoids excessive creation and destruction of converters when a user is switching between streams. The procedure to acquire a connection to a server transparently either returns an existing JVSink object or creates a new one; the only requirement for the client is that it can process the type of JVBuffer object generated by the JVSink. Communication via buffer pools also makes it easy to change components in the pipeline, as the only dependancy between converters is the type of JVBuffer the receiving converter can accept. We could, for example, substitute a software decompresser for our existing JVDecomp or insert a thresholding filter between the decompression and the display stages.

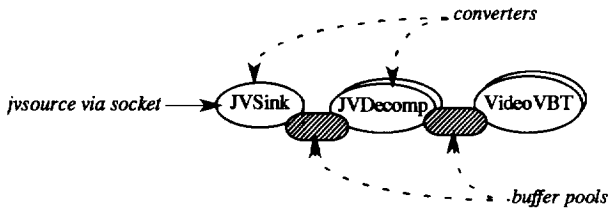


Fig. 4. The video pipeline implemented with JVConverters and JVBufferPools. At each stage there may be multiple readers of a buffer pool.

4. Additions to Trestle

4.1. Trestle on X

To explain our extensions to Trestle, we must first describe how input and output are managed in its X implementation. An input event generated by the X server is received by the *XInput* thread, which unpacks it and appends it to an internal queue. An *XMessenger* thread pulls the event off this queue, interprets it, and dispatches it down the VBT tree; the relevant method of the destination VBT implements the application's response to the event, if any. This is shown in the upper half of Fig. 5.

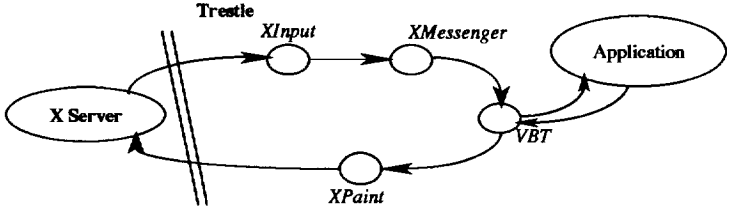


Fig. 5. Input and output paths in the X implementation of Trestle.

Painting operations on a VBT place a request on the VBT's paint batch queue. Some time later, when the *XPaint* thread wakes up or an explicit synchronisation is requested, the batch is passed up through the VBT's antecedents until it is processed by an ancestor—by merging it with the ancestor's paint batch, by changing it in a filter or, eventually, by painting its contents to the screen. A *ZSplit*, for example, manages overlapping child VBTs, so its paint method may break a child's paint request into several smaller requests that represent the exposed parts of the child's area (Fig. 6); Trestle does its own clipping, rather than using X's. The *JoinVBT*, on the other hand, attaches its child to two parents so that an application window can be replicated, possibly on different displays; its paint method replicates its paint batch and passes a copy up to each of its parents. The *XClient* VBT provides the bridge between Trestle and an X server; its paint method translates Trestle paint requests into calls to X painting procedures; this is the lower half of Fig. 5.

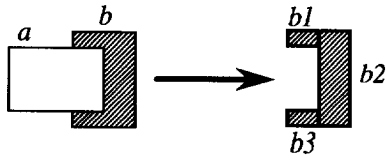


Fig. 6. Splitting a VBT's paint request. The paint request for VBT *b* is broken into three components (*b1*, *b2*, *b3*) to paint an overlapped rectangle.

4.2. Pictures

The previous version of Trestle provided limited support for displaying arbitrary images. Trestle *Pixmaps*, like X Pixmaps, are stored in the display server and suitable for small and long-lived items such as icon symbols. We defined a type *Picture.T*[‡] that allows an image to be constructed in the application and then passed to the window system for painting. A second requirement was that Picture operations should use shared memory if it is available but should still work if it is not; this allows application code to continue to work if the window is moved between displays or has its output replicated to multiple displays. A subgoal of this requirement was that it should be easy to build a version of Trestle without the X shared memory extension.

A Picture object is a handle for a block of memory that contains the image pixel values and information about the image (bearing a remarkable resemblance to an XImage record); we must provide this level of indirection because the memory may be allocated outside the Modula-3 type system. Applications may paint an image to the display by calling the procedure `Picture.Paint` which places the picture in the paint queue and returns when the paint request has been fulfilled; this avoids the application thread releasing the image buffer before it has been processed by the paint thread.

Picture objects are created with respect to a particular screen, so the procedure `Picture.New(screen)`, where `screen` refers to an X display, returns an instance of *XPicture.T*, the X-specific subclass of Picture.T. An XPicture includes a `put` method which the XPaint thread calls to implement the paint request (i.e. display the image on the screen); for normal images, this is simply a call to `XPutImage`. When the X server is on the same host as the client and supports the X shared memory extension, `Picture.New()` returns an instance of *XSharedMem.T*, a shared memory subtype of XPicture.T.

Shared memory images, however, are more complex to implement because, first, shared memory requires that the application and X server are on the same host

[‡] `Picture` is the name of an interface, so `Picture.T` is a type `T` declared in the interface `Picture`. Similarly, `Picture.Paint`, described below, is a procedure `Paint` declared in the same interface.

and, second, the application must wait until the X server has actually painted the image (rather than just receiving it) before reusing the storage. To solve the first problem, the XSharedMem.T object records the X display to which the shared memory segment is attached. Its `put` method compares this display to the destination display specified by the caller and reverts to the parent XPicture method if the two differ. The second problem requires a more sophisticated solution.

4.3. Completion events.

X shared memory extension clients may request the server to return a *completion event* to notify them when it has fulfilled a paint request so it is safe to overwrite the image buffer. As described above, however, Trestle may generate several paint requests on the same image as a VBT is clipped and transformed, so a simple association between images and completion events is inadequate. To resolve this each call to `Picture.Paint` stores a *Completion* object, which contains a reference count and a condition variable, with the paint request. As the paint request is propagated up the VBT tree, each additional reference to the image (such as the subdivision of an overlaid area in a ZSplit) increments the count in the Completion object.

When the Trestle paint request reaches the XClient level and is translated into a call to `XShmPutImage`, the serial number of the X request and the related Completion object are stored in a completion queue; there is an entry in the queue for each `XShmPutImage` call which has not yet completed or returned an error. The server responds to image requests with completion or error events which are received by the XInput thread and matched with records in the completion queue. The record is then removed from the queue and its Completion object decremented. The condition variable of the Completion object is signaled when its count returns to zero so that application-level code can be notified when the image storage is safe to reuse. By default, the `Picture.Paint` procedure waits for the Completion object signal before returning and, thus, hides the Completion mechanism from the application. As Fig. 7 shows, the completion mechanism does not interfere with Trestle's general event handling.

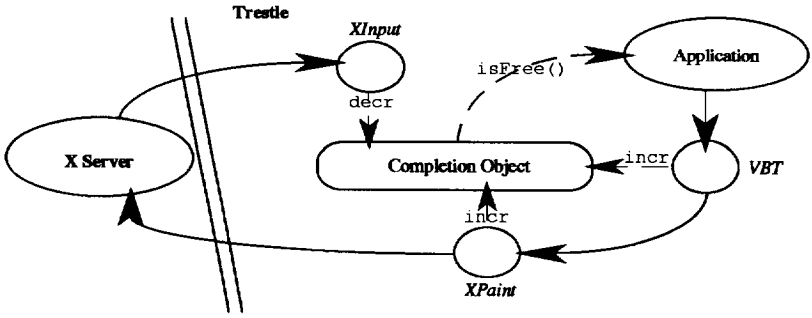


Fig. 7. The path for completion events for shared memory paint requests. When all the paint requests for an image have been fulfilled the Completion object (in the centre) will notify the application that the shared memory segment is now free.

5. Integration with other tools

5.1. VideoVBT

To tie the JVideo library and Trestle together, we implemented a *VideoVBT.T* that provides a convenient interface for the application writer, who need only specify a video source and some parameters when creating the VBT. When a VideoVBT is attached to a display ("realized" in X terminology) it assembles a pipeline to connect it to the video source and starts a thread to receive frames from the pipeline and display them. The VideoVBT handles changes in state automatically: images are scaled to fit the VBT's shape and the display loop is suspended when the VBT is not visible on the screen. The user may also specify a minimum time between frames so that an application may reduce its processing demands where a full frame rate is unnecessary; "active icons," for example, may only refresh an image every few seconds to give the user an idea of the activity in a video stream.

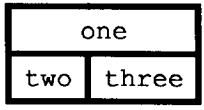
We also wrote an *AudioVBT.T* that provides an interface to the audio subsystem. It is implemented as a Trestle filter, so it is invisible to the user but can intercept window mapping, unmapping and destruction events. When made the ancestor of a VideoVBT, an AudioVBT ensures that an audio connection is live only when its associated video window is visible.

5.2. FormsVBT

We then integrated the audio and video VBTs into *FormsVBT* [3], an interface builder that allows application writers to specify an interface, in a lisp-like language, which is separated from the application. For example, the FormsVBT expression:

```
(Frame       ;; draws a border around its child
  (Vbox      ;; arranges its children vertically
    (Button %button1 (Text "one"))
      ;; creates an interactive button
    (HBox   ;; arranges its children horizontally
       (Button %button2 (Text "two"))
       (Button %button3 (Text "three")))))
```

produces the interface:



where the effects of selecting each button can be specified by attaching callbacks to the named components *button1*, *button2* and *button3*. We can build a simple channel switcher with the expression:

```
(Frame
  (Shape (Width 500) (Height 400)
    ;; fixes the shape of the child windows
    (TSplit
      (TButton %source1 (For source2)
        (Audio "source1" (Video "source1")))
      (TButton %source2 (For source3)
        (Audio "source2" (Video "source2")))
      (TButton %source2 (For source1)
        (Audio "source3" (Video "source3"))))))
```

A *TSplit* displays one of its children at a time and a *TButton* forces the parent TSplit to display the child named in the *For* parameter, so the expression creates an application that cycles between the three video sources each time the user clicks in the window. The audio is also switched with the video stream as only one *Audio* at a time will be mapped to the display and, hence, active. We have built several applications using only FormsVBT expressions without any additional Modula-3 code; these include a channel selector that presents the available channels as live video windows in a pull-down menu, and a magnifier that allows the user to choose the scale to which the video stream is decompressed and pan around the video image through a smaller window.

5.3. Obliq

Finally, we integrated the new libraries with *Obliq* [5], a lightweight interpreted object-oriented language that is implemented over Modula-3. The following fragment, for example, defines a procedure that will be attached to a *TypeInVBT* (which allows a user to edit a single line of text and generates an event when the Enter key is hit) with the name "videoSource". The procedure extracts the name of

the video source from the TypeInVBT and constructs a FormsVBT expression that will be evaluated and the result inserted into the parent VBT called "parentVbt"; the new VBT generated is a VideoVBT that will display the video stream from the source specified in vname.

```
let videoproc = proc(fv)
  let vname = form_getText(topForm, "videoSource", "");
  form_insert(topForm, "parentVbt",
    "(Shape (Width 100) (Height 80)
       (Video \"" & vname & "\"))", 0);
  end;
```

The code to attach the procedure to the TypeInVBT is simply:

```
form_attach(topForm, "videoSource", videoproc);
```

Thus, we can use six lines of code, plus a related FormsVBT expression, to implement a part of an interface that allows users to create arbitrary video windows. This integration with Obliq shows how careful design of our basic components makes it easy for programmers to develop, and experiment with, applications which manage complex data types such as audio and video.

6. Object-oriented techniques

It is difficult to say exactly how much each of the features of Modula-3 helped in the implementation of the system—lightweight threads were essential and garbage collection simplified much of the code—but we can highlight three of its object-oriented techniques that proved most useful: inheritance, partial revelation, and encapsulation.

6.1. Inheritance

Throughout the system, we put some effort into exploiting the class hierarchy to encourage code reuse. The current JVideo library, for example, supports two types of buffer, `JVFromSource.T` for compressed images and `JVFromDecomp.T` for decompressed images, both of which are subclassed from an abstract `JVBuffer.T` type; this allows JVBufferPools to be defined solely in terms of JVBuffers rather than a particular subclass. Reference counting, for example, is independent of the contents of the buffer so it is implemented in the JVBuffer supertype. Modula-3 is strongly typed, so when a converter acquires a buffer from a buffer pool, the method it calls returns a reference to a generic JVBuffer, rather than to a concrete subtype. Modula-3 includes a NARROW function that provides type-safe casting between related types. For example, the JVDecomp loop includes the lines:

```
VAR inbuf : JVFromSource.T := NIL;
(* etc... *)
  inbuf := NARROW(source.waitForChange(), JVFromSource.T);
```

where `inbuf` is a local variable for a reference to a `JVFromSource.T` buffer and `source` is the buffer pool to which the previous converter in the pipeline writes; we can trust this assignment as `NARROW` checks the validity of the conversion.

The only activity in a buffer pool that depends on the type of buffer it holds is the allocation and deallocation of individual buffers. To support this, the interface file for each buffer subclass includes a *Factory* object, subclassed from a common supertype, which is used to "manufacture" and recycle buffers of the given type. During its initialisation, a JVBufferPool is passed a Factory object which it then uses to manage its storage. Furthermore, in this implementation, all Factory objects for a given type of buffer maintain a common free list to hold unused buffers, which allows unused shared memory segments to be shared between buffer pools (Fig. 8). The free list is set up in the initialisation block of the file that implements the buffer type; Modula-3 guarantees to run this block before running any code which uses the buffer type.

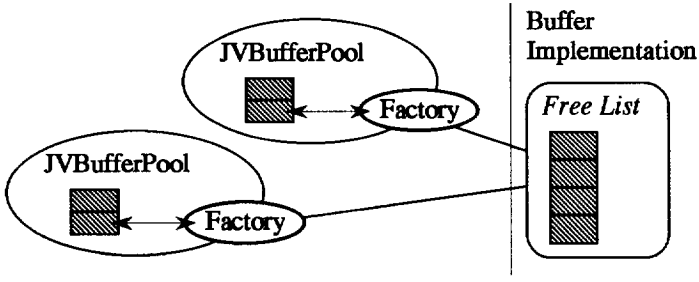


Fig. 8. Two buffer pools are used to store the same type of buffer, so their Factory objects refer to the same free list. Buffers are shown as shaded rectangles.

Similarly, common features of JVConverters, such as management of their threads, are implemented in an abstract JVConverter class, while the specific conversion—from socket to buffer or decompression—is implemented in the concrete JVSink or JVDecomp subclass.

The flexibility of the new JVideo library depends on the careful selection of the basic components and the strict enforcement of the boundaries between them; converters, for example, communicate only through buffer pools. The process of developing an inheritance tree helped us to determine the common features of these components—the infrastructure of the library—which we implemented in the abstract superclasses. We then implemented specific features in the leaf-classes, using inheritance-based polymorphism to share the common code. Constructing a new stage in the pipeline now requires (simply) the creation of a new subclass of JVConverter and, possibly, a new subclass of JVBuffer.

We found similar advantages in the Trestle library. Image handling is defined at the application level in terms of generic Picture objects, while at the X implementation level it is defined in terms of XPicture objects—shared memory or not. We can rely on individual Picture objects to determine the "right thing" to do under the circumstances. The code for determining which Picture subtype to create is localised within one file and all the code that implements each case can be contained within its own file.

Finally, inheritance allowed us incrementally to add video and audio to existing tools. The VideoVBT is subclassed from a leaf VBT and the AudioVBT from a filter VBT, so both can be inserted into any Trestle application. To include the video and audio VBTs in the FormsVBT we had only to add the new types to the forms interpreter—about 100 lines of stereotypical code—and link in the new library; everything else is specified in terms of standard higher-level VBT types. The integration with Obliq required even less effort—just relinking with the new FormsVBT library. Obliq uses the FormsVBT interpreter to create new VBTs, passing forms expressions as text, and its callback procedures are defined in terms of generic VBTs.

6.2. Partial revelations

Modula-3 allows arbitrary components of an object to be partially revealed in multiple source files—unlike C++, which allows only `private`, `protected` and `public` levels of access. Thus, we can declare[§] in the `XClient` interface that:

```
TYPE
  T <: T_Ext;   (* T is a subtype of the type T_Ext *)
  T_Ext <: T_Public;  (* T_Ext is a subtype of T_Public *)
  T_Public = OBJECT   (* the definition of T_Public *)
     public fields...
```

For the X implementation, the `XClientExtension` interface file contains details of all the X extensions which this build of the Trestle library uses. At present, we support only the shared memory extension, so this interface reveals that the type `XClient.T_Ext` is a subtype of `XSharedMem.XClient_T`; if we wish to support other extensions, we can use this interface to insert additional types in the hierarchy. Finally, we define `XSharedMem.XClient_T` by declaring in its *interface* file that:

```
TYPE XClient_T <: XClient.T_Public;
```

and revealing in the *implementation* file that:

[§] The phrase `A <: B` means that `A` is a subtype of `B`, but that this is not the entire definition of `A`; the rest of `A` will be revealed in other places, possibly in other files. This allows us to place a type in a hierarchy while hiding its details from the public interfaces.

```
REVEAL XClient_T = XClient.T_Public OBJECT
  fields specific to the shared memory extension...
```

When declaring an intermediate type in an hierarchy, we can also chose how much detail to specify in each declaration, including its position; we can reveal elsewhere exactly where the type belongs. An application sees the type declarations in the public interfaces for the X extensions (XSharedMem, $X_1..X_n$ in Fig. 9) as independent of each other; to gain access to a particular extension, it imports the relevant interface.

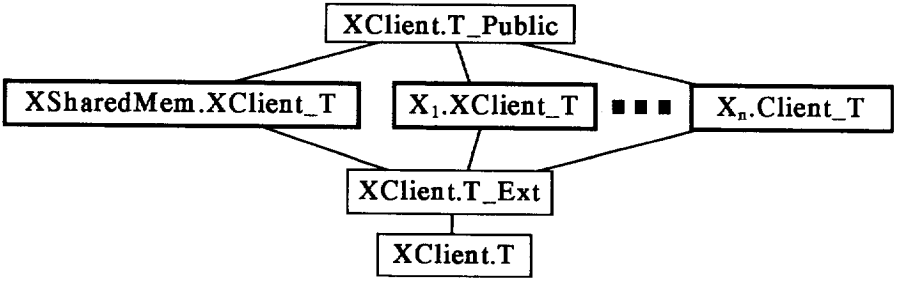


Fig. 9. The XClient type hierarchy as defined in the interfaces; supertypes are shown above subtypes.

The definitive linear hierarchy is then revealed in the implementations (Fig. 10).

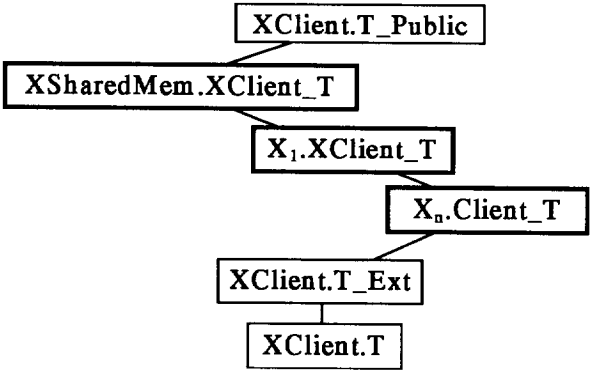


Fig. 10. The XClient type hierarchy as revealed in the implementations.

This approach may seem baroque, but it allows us to split the definition of a complex type into a set of discrete components with control over the scope of the fields in an object. The extension-specific fields revealed in the implementation of `XSharedMem.XClient_T` are visible only in that file, so we can change them without affecting any other source code. This allows us to restrict all the shared memory code for the library to the `XSharedMem` implementation file. Furthermore, any code that uses an X extension must import `XClientExtension` to reveal the

details of the type `XClient.T_Ext`, otherwise the type is opaque, and the relevant extension interface; this makes any dependancies on an X extension easy to identify.

6.3. Encapsulation

We made wide use of implementation encapsulation, hiding the details of how an object works behind an interface which describes what the object does; this helped to focus the design of the components of the system and made maintenance easier. The interface of the VideoVBT, for example, is defined solely in terms of the name of the video source and various video parameters. The application programmer sees nothing of the JVideo or Trestle Picture systems, so we could substitute a different implementation without changing any application code.

Objects, however, also encapsulate state and so can be used by components to communicate with each other. A completion object, for example, provides a single point of access for the state of a paint request; the application thread is notified of a change in a completion object's state (when its reference count drops to zero) when the XPaint thread has finished with the related buffer. Similarly, a JVBufferPool encapsulates the state of a stage in the video pipeline: its current buffer holds the most recent image in the stream, and its reference and free buffer counts describe its flow control. Readers and writers communicate by accessing common information held in a buffer pool—either directly, as when a writer changes the current buffer, or indirectly, as when a reader releases a buffer, which in turn is released to a writer.

Communication via objects, rather than via separate channels or global variables, helped us to enforce the boundaries between components. First, access to the events which affect an object is included with the object itself, and so is limited to that code which is interested in its contents; this avoids unforseen side-effects. Second, remote parts of a system that hold a reference to the same object can use it pass events to each other; this avoids the need for extra communication paths, and hence dependancies, between components. Finally, sharing events via objects means that *de facto* communication paths exist where they are needed, so it is easy to change the senders and receivers when, for example, a new converter is added. Incidentally, we found that the integrated garbage collection encouraged our use of this technique, as we did not have to worry about coordinating deallocation between distant parts of a library.

7. Related Work

The Pandora system [11] uses additional image hardware to mix the desktop and video streams. The output from the host window system is piped through the Pandora unit before being passed to the display. Within the Pandora unit, a mask plane can be set to accept pixels from the window system or a video stream; this plane is set by the window system to describe the shape and location of the current video windows. This approach allows video streams to be drawn directly to the display without affecting the processing on the local host but requires special

hardware and a modified X server to coordinate with this hardware. The Trestle extension, on the other hand, runs on a standard machine and window system, and makes the image data accessible to the application; it requires, however, much greater processing and memory resources from the host.

Apple Computer's QuickTime [1][2] is another general framework for displaying video in a window. It supports multiple compressors and decompressors, fitting to a single interface. It does not attempt to provide any unifying abstraction between the streams of compressed and decompressed video, nor does it provide simple mechanisms for controlling latency. It does provide the parameters that would be necessary for implementing pipelined decompression, but the standard procedures used to display compressed video do not take advantage of this. The lack of thread support causes QuickTime to handle concurrent actions through an event loop; this makes it much harder to implement the control flow needed for greater asynchronous action.

Schnorf's extension to ET++ [10] is closest to our work. It integrates video into an object-oriented toolkit which is based on an abstract window system layer and Schnorf has added video to a number of standard applications. Its video objects, however, are used to control autonomous video processes that draw the video images directly to the display using either hardware or a separate software process. In our approach, video data is handled within the application process, making it available for sharing or further processing, and avoiding issues of co-ordination between video and application processes.

Schnorf's work and ours are based on fundamentally different underlying toolkits. The primary mechanism in ET++ for painting is based on damage repair. Painting is typically not clipped on the way to the screen, but instead the painter's algorithm is used (typically in an off-screen buffer) to cause the appearance of clipping. Much of Schnorf's work was concerned with the difficulties of adding asynchronous painting to ET++. In Trestle, parents are always responsible for performing any necessary clipping for their children. Asynchronous painting has always been supported (although double-buffering and region invalidation are provided for those VBTs that prefer it). This allows us to provide flicker-free integration of video into our user interface. To minimize the complexity of the interface between video hardware, applications, and the window system, J-Video does not attempt to transfer video images directly to display memory. The current design allows DMA transfers of decompressed frames to arbitrary memory locations. This allows us to use built-in capabilities of the window system to perform masked transfers with the same efficiency in bus transfers that direct painting would afford, without modifying the window system or complicating the client. Had we been working with hardware similar to Schnorf's, in which masked transfers to the screen were considerably more efficient than transfers via memory, we might well have chosen a design more similar to his.

8. Conclusions

We have used an object-oriented approach to add digital video to a user interface toolkit and found a number of benefits. We broke the video pipeline down into a set of communicating objects, each of which represents a stage in the processing of an image. This produced an internal structure that is more flexible and easier to understand and, hence, maintain than its predecessor. We also exploited such Modula-3 features as its type system and its distinction between interface and implementation to produce well-structured code with better encapsulation and few implicit side-effects. It is now simple to share the image data at each stage in the pipeline (compare Fig. 2 with Fig. 11) and to change the stages in the pipeline. The new extensions have proved easy to integrate into higher-level toolkits, which we have used to build some simple applications.

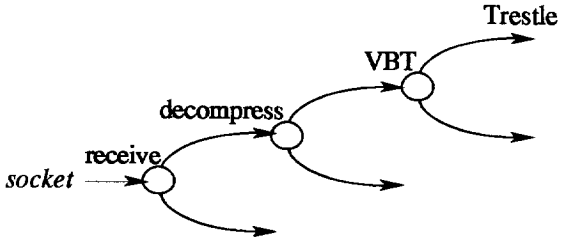


Fig. 11. The new video pipeline. The output from each stage is now shareable by multiple readers.

We ran some basic tests and got the timings in Table 1 for a loop that repeatedly displays the same image (that is, no effort is spent on generating the image data):

	local socket	*shared memory*
320 × 240	42	54
640 × 480	11	22

Table 1. Frame rates (to the nearest frame-per-second) for two image sizes and two types of inter-process communication.

This shows that we can display video streams at respectable frame rates (although not yet full video) for quite large images and that, as one might expect, the use of shared memory is more important for large than for small frames. With a full video pipeline (i.e. accepting images from a remote video source) we achieved between 18 and 19 frames a second for 500 × 400 pixel images. All the implementations were run on a DECStation 5000/240, with 128M of memory, but the compiled code was not optimised, so the timings are only indicative. One curious effect is that, when compared to the Motif-based implementation, frame rates are worse but latency is slightly (but consistently) better; we have not yet analyzed this difference.

There are, of course, a number of unresolved issues we have to address. The first of these is to develop some heuristics for determining how long to cache buffers, pools and converters. At present, converters, once created, are kept in a pool until the program terminates so that connections can be quickly re-established. This improves responsiveness to the user but can lead to an application hogging the workstation, so we need a mechanism, perhaps under user control, which allows unused resources to be freed. We also intend to implement software decompression so that video applications can be run without JPEG hardware, if more slowly. Thirdly, we need to spend time tuning the video software so that we can achieve real-time video frame rates and to reduce the processing load on the host machine.

Our enhancements to the Trestle toolkit show, first, that we can achieve reasonable performance with complex media such as video while retaining such facilities as an object-oriented strong type system with inheritance-based polymorphism, lightweight threads and garbage collection. Second, they show that these facilities provide real benefits for both the development and the internal structure of such systems.

Acknowledgements

Thanks to Hania Gajewska for originally suggesting the project, to Lance Berc, Dave Redell and Luca Cardelli for corrections and suggestions, to the reviewers for their comments, and to Digital's System Research Center, particularly the Argo group, for supporting the work.

References

1. Apple Computer Inc., Inside Macintosh: QuickTime. Addison-Wesley, Reading, MA., 1993.

2. Apple Computer Inc., Inside Macintosh: QuickTime Components. Addison-Wesley, Reading, MA., 1993.

3. Avrahami, Gideon, et al., A Two-View Approach to Constructing User Interfaces. Computer Graphics, 23 (2), July 1989, pp. 137–146.

4. Berc, Lance, et al., J-Video: High Performance Digital Video on Conventional Workstations. In preparation.

5. Cardelli, Luca, Obliq: A Language With Distributed Scope. Technical Report 122, Systems Research Center, Digital Equipment Corp., Palo Alto, CA. In preparation.

6. Corbet, Jonathan, Keith Packard. The MIT Shared Memory Extension. The MIT X Consortium, 1991.

7. Gajewska, Hania et al., Argo: A System for Distributed Collaboration. In preparation.

8. Manasse, Mark, Greg Nelson. Trestle Reference Manual. Technical Report 68, Digital Equipment Corp, System Research Center, Palo Alto, CA, December 1991.

9. Nelson, Greg (ed). Systems Programming with Modula-3. Prentice Hall, Englewood Cliffs, NJ, 1991.
10. Schnorf, Peter. Integrating Video into an Application Framework. Proceedings of ACM Conference on Multimedia, Anaheim, CA, August 1993, pp. 411–418.
11. Wray, Stuart. The Interface to Pandora's Box. Technical Report 89-4, Olivetti Research Ltd, November 1989.