MEDIACORE: A MULTIMEDIA INTERFACE COMPOSITION TOOLKIT

Gonzalo Navarro*
Universidad de Chile at Santiago, Chile

Jorge Sanz
University of Illinois at Urbana-Champaign, Illinois
IBM Argentina

Abstract

In this paper, the problem of complex interface development when adding multimedia features is analyzed. A model is proposed which focuses mainly in managing the complexity of control, a problem largely ignored in many approaches. This model is integrated into a whole development methodology for composing arbitrarily complex multimedia interfaces. A system implementing this model is presented, which isolates the user from nonrelevant details of the multimedia data he manages and allows him to program full interfaces while thinking only in terms of the model.

Key words and phrases: Multimedia systems, multimedia interfaces, interface composition.

1 Introduction

Human-machine communication has been continuously evolving, from the early days of punchcards to today's graphical environments and windowed interfaces. This evolution has been discontinuous, with big jumps spaced by long periods of stability.

At each step, the achievements in interface friendliness were correlated with hardware advances, new base software support... and more programming complexity for applications [11, 10]. Application interfaces, which began almost nonexistent (in the old Input → Program → Output model), evolved to application-driven (alphanumerical interfaces), and finally to event-driven (graphical windowed interfaces). Each step involved more complex interface logic.

A new step in computer interfaces is at the next corner: technology has evolved so as to allow multimedia data being managed by computers, so the time of multimedia interfaces is coming. The jump from windowed to multimedia interfaces could make interface programming complexity simply the most relevant part of application development.

A multimedia interface will have to take into account difficult problems such as paralellism, synchronization of data streams, real-time constraints, asynchronous events, very large data objects, CPU-demanding activities, etc. [5, 1]. And all those problems not only claim for better hardware and base software support, but for much more complex logic.

The aim of this toolkit is to ease the process of multimedia interface composition in two ways: by isolating the programmer from low-level details of multimedia interaction (hardware, drivers, operating system, etc.), and by providing him a model for expressing the complex logic of such interfaces. The first goal is achieved through a set of *multimedia servers*, which isolate not only the low-level details of each medium, but also its synchronization, timing and data manipulation issues. The second goal is achieved by creating a powerful model of the logic of a multimedia interface, which incorporates communication with the multimedia servers, and giving the necessary support for programming in that model.

The description of the toolkit may be divided in four areas: global model, control agents, multimedia servers and development methodology. Each area will be explained in a separate section.

2 Global model

We will use the term *show* to denote a running program which has a multimedia interface. Our aim is not to develop a special language for shows (as many proposals do, generally losing some expressive power in

the process, see for example [12, 9]), but to enrich a traditional one with the necessary tools to easily write shows.

In our model, a show will be seen as the concurrent execution of a set of *agents*, each one the guardian of a *resource* in the show. Agents are sequential in nature, so this is a flat model in this sense (i.e. agents are not shows). Agents communicate by using asynchronous message passing.

2.1 Agents

There are two main classes of agents: server agents and control agents.

Server agents are simply multimedia servers. They act as guardians of specific multimedia resources, and are predefined: the composer picks them from the pool the toolkit offers. Control agents, instead, are composer-defined, and express the logic of the show. They are what differentiate one show from another. Figure 1 illustrates this idea.

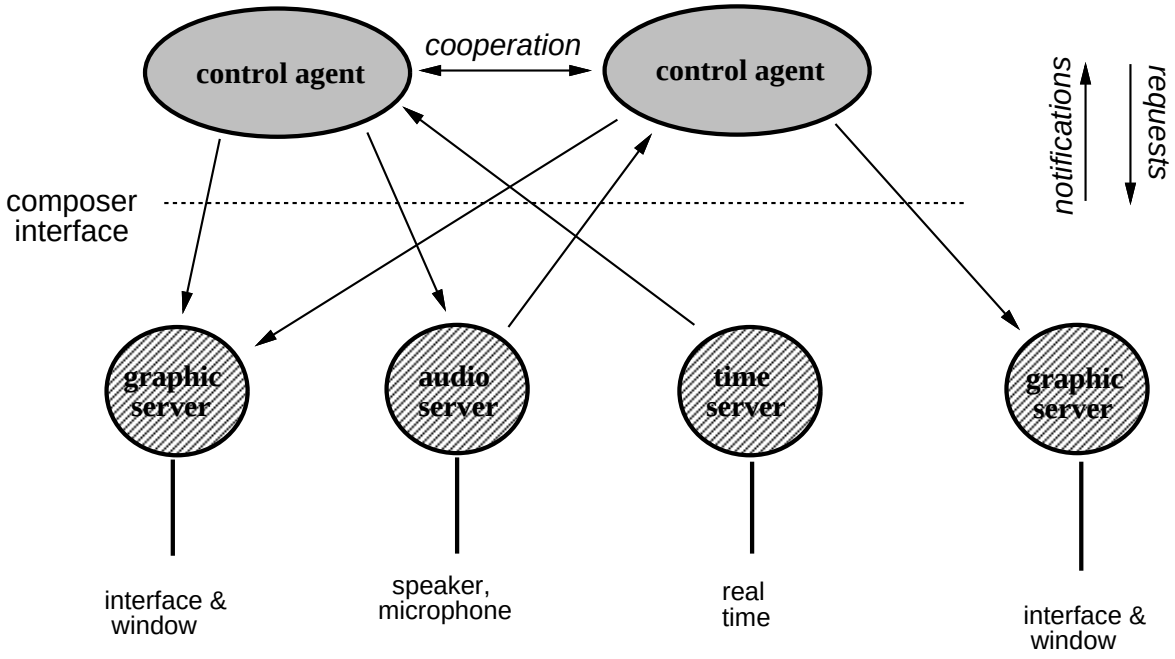

Figure 1: The interface model.

Thus, multimedia servers encapsulate low-level details of the implementation of each medium, and offer a high-level and consistent interface to the composer, which interacts with them from control agents when defining the logic of the show. Each server defines a number of requests it services (e.g. "play audio", "show image", "set alarm"), and a number of notifications (e.g. "mouse click", "time alarm", etc.) it provides to interested control agents. The composer should not be aware of low-level changes. Also, new capabilities would be reflected in more multimedia servers to choose from.

In case of control agents, encapsulation of a "resource" of the show is intended to mean "behavior". If the composer designs wisely a set of control agents, each one could be dedicated to perform a specific task. Control agents communicate among themselves by interchanging arbitrary messages, as opposed to the fixed interface offered by servers.

Control agents may have an arbitrary complex logic, and the toolkit offers a powerful model for expressing it. This is explained in a separate section.

Control agents may be combined (perhaps also with some servers) to form more complex "multimedia servers". For example a video server, and audio server, a graphic server and a control agent could be

combined to create a new, reusable, "annotated movie server".

All agents of a show are started simultaneously, and the show is considered active until the last agent terminates. So, the process model is CSP-like [2].

2.2 Messages

Communication between agents proceeds exclusively by message-passing, in the variant most used by client-server models: the sender specifies the recipient, but the recipient does not specify from which sender to receive [2]. The idea of guardians, combined with message-passing fixed services offered by servers, resemble Ada tasks. The interface of control agents, instead, is arbitrarily flexible.

The client-server model also specifies (in general) an asynchronous message-passing scheme, with FIFO servicing. This is the main method used here; but in a multimedia environment it is also needed a mechanism for immediate servicing of some requests (e.g. pause/resume). Thus, although the FIFO model is the normal convention, there are special types of messages, which are immediately delivered to the front of the message queue of the recipients. The servers guarantee immediate servicing of those messages, and the composer should do the same in control agents.

Another feature is the possibility of broadcasting messages to all agents in a show. This is used, for example, by a time server to synchronously send ticks to all participant agents, if asked to do so.

Synchronization mechanisms are provided on top of the asynchronous model, in two ways. In the first, a special message is defined, which when reaches the front of the receiver's FIFO queue, transparently generates an acknowledgment and dissappears. If the recipient follows a FIFO processing semantics (as all servers do) the ack guarantees that all previous messages have been processed.

The second is a general send-wait mechanism, in which the sender expects an explicit ack from the receiver, only broken by immediate (non-FIFO) requests.

Communication with servers proceeds in two main forms. In the first (*request*), a control agent "asks" something to a server (e.g. to show some image, to a graphic server) and the server replies (if necessary). In the second (*notification*), the servers themselves generate "events", and notify all interested agents of them. Examples of events are clock ticks, special points in an audio stream, mouse clicks, etc. In order to receive events, a control agent must declare its interest to the corresponding server. An agent may also mask messages, in order to not receive them.

As said, server messages (requests and notifications) are predefined. Control agents, instead, must "create" new messages for communicating with one another.

Along with the predefined synchronization messages, there are also a few number of predefined immediate messages. These are *pause*, *resume* and *terminate*. The two first are meant to express the suspension of "real" time, while the last is for immediate termination (e.g. in presence of an error). While the servers obey this semantics, the tool provides mechanisms for the composer to do the same in its control agents.

3 Control agents

As said, control agents are which define the behavior of the show, by interacting with servers, with one another, and by performing whatever actions are desired in the program. Since, as said in the introduction, the logic of multimedia interfaces is harder to program than traditional ones, the tool provides a powerful model for expressing the logic of these agents, by focusing on their response to messages (i.e. notifications from servers and messages from other control agents). This problem is largely ignored in many composition models (see, for example, [12, 9]).

The idea is to extend the classical event loop [10] with *states*. The agent's code is seen as a dynamic automaton [3] (the "dynamic" aspect will be explained shortly). States represent the potential behavior of the agent upon each message. There is an initial state, at which execution begins. At each state and upon receiving each message, the agent performs a different action and passes to a new state. There is also a final state, at which the agent immediately terminates if it passes to it (upon receiving a termination message, the next state is always the final one).

The agent could be modelled, in principle, as an automaton with labeled edges, each edge corresponding to a `(message,action)` pair. But it is really more powerful than that, since it is the action which dictates

which the next state is (if it has to be changed at all), and the term "dynamic" stands for this capability.

Another facility is the attachment of *entry* and *exit* actions to a state, which are to be invoked, respectively, when entering a new state and when exiting the current one. Both are performed after the action of the corresponding "edge" completes. They wouldn't really add nothing to the expressive power, except for the "ghost" transitions to the final state caused by a termination message (this way allowing cleanup actions).

The automata model is actually present in many event-driven programs, implemented in many tortuous forms. In a complex multimedia interface, the complexity could be raised up to the unmanageable. This model, by simplifying the design and implementation of a control agent down to the definition of automata, will greatly ease this task.

4 Multimedia servers

The global model assumes little about multimedia servers, which allows the toolkit to be extended with new servers as new possibilities appear. The clearcut request/notify semantics are common grounds on which to specify the behavior of each medium from a very high-level point of view.

Three multimedia servers are, by now, provided in the toolkit, and this is related to the capabilities available at our operating environment: a real-time server, a graphics and interaction server, and an audio server. In the last section we explore a little the possibility of a video server.

4.1 Real-time server

The time server is designed as a general mechanism for synchronization. It may act as the clock of the whole show, imposing a uniform timestamping on all agents. At most one time server can exist per show.

The most important request this server accepts takes the form "after x seconds, send me/anyone message m, n times, with a gap of d seconds among each of them". n may be specified as infinite. This request, submitted at time t, will generate n notifications (message m) at times $t+x, t+x+d, t+x+2d, ..., t+x+(n-1)d$ (or infinite ones if $n = \infty$). This is called a "notification sequence", and is very useful in multimedia shows.

At any moment, however, the sequence can be cancelled by the requester. Also note that the "ticks" can be sent to the requester or broadcast to all agents (notice that unless an agent unmasks a message, it will not be received).

Other services include: a blocking wait, the current time, converting timestamps to human-readable strings, etc.

Upon a termination request, the time server simply frees up its resources and ends, thereby truncating all notification sequences in progress.

Response to pause and resume is a bit tricky. The problem is how to understand, being the clock of the show, the time elapsed between pause and resume, specially if some client agent is not paused and expects notifications or sends new requests (it was decided not to pause the whole show upon time server pause, this can easily be done at higher levels). The time server acts as if time were stopped at pause time, and continued at resume time. In the while, requests arrived T seconds after pause time are accepted and stored as if they had been sent T seconds after *resume* time. Besides, all other pending requests are also shifted with the pause gap, and all else proceeds normally.

An important problem to be addressed by this server is the *gap* between the moment in which a notification is sent and in which it arrives to the queue of the requester. The times labeled $t + x + id$ are supposed to be the arrival times, not send times. So, the gap must be subtracted from the arrival times to get the send times. The time server uses a mechanism by which an estimation of the gap is being tuned permanently, since it depends on system workload.

4.2 Graphics and interaction server

Since the windowed environment [10] (of which this tool is an extension) imposes a per-window input policy, interaction services could not be separated from graphic services (the precise reasons for this will become apparent later).

This server is the guardian of a window, hiding hardware and low-level problems from the composer, and presenting him a high-level interface to many graphic features. X- and graphic-server-windows may freely

coexist in the parent-child hierarchy, this way easing the inclusion of some multimedia features in an already existent, traditional windowed application.

The response of this server to pause requests is to immediately stop all input/output activities, freezing the window, until receiving a resume request. Response to termination involves cancelling all input/output activity and destroying the window.

4.2.1 Graphic services

Once created, the graphic server maps its window and starts servicing requests of various kinds.

A group of requests allow to display full-RGB 24-bit 3D color images, with the desired *effect*. An effect is a form of showing the image smoothly, for example by fading it in, or by randomly filling squares of the image until it is wholly mapped, or by making it appear from the left side moving it to the right, etc. A large number of effects are provided. Each one may be asked to be performed within a given time, with a given "granularity" (size of the atomic granules that comprise the effect, e.g. the size of the mentioned squares).

Images are stored in a custom format (converters are available). For each pixel, not only its RGB value is stored, but (optionally) its depth and a *transparency* tag. If the pixel is transparent, or its depth is larger than the corresponding point in background, it will not change this background when the image is mapped (images may be mapped at any x, y and *depth* coordinate).

It is possible to "preload" an image, precalculating the final (3D-merged) result and storing it for fast rendering at time-critical parts of the show.

Another group of requests allow to draw interactive 3D graphics [4] in the window, correctly merging with depths of images. Although the ideal protocol would have been to submit display lists, technical limitations imposed to use submission of procedures to execute (which are expected to draw those display lists).

Yet another request allows to show text, in any X-known font, color, size, position and depth.

Finally, it is possible to save the current contents of a window, as a normal image. This provides for multimedia image editors, and allows to record as images the runtime capabilities of the graphic server (e.g. 3D graphics).

Although the required drivers were not available, a placeholder has been left in the protocol that allows handling video camera input as a normal image file (i.e. allowing to show online static shots in a window), this way providing for very interesting interaction possibilities, and combined with the saving capability, a way to capture images.

4.2.2 Interaction services

User input is gathered on a per-window (i.e. per-server) basis, from two devices: keyboard and mouse.

There are two kinds of keyboard services: modal and modeless.

The modal input service consists of a (non-blocking) request an agent makes to the server. When received, the server allows the user to edit a string in the window (a number of features are provided), and notifies the requester of the final input string. Only then does the server proceed with the next graphic or interaction request (as it can be seen, these two services are tightly coupled). Note that, however, the protocol is not blocking from the side of the requester: it makes the input request and the answer notification arrives later.

Modeless keyboard input allows agents to be notified of asynchronous user keystrokes. An agent declares its interest in those notifications, and may further cancel them.

The other supported device is the mouse, which has no "modal" form: agents request or cancel their interest on click notifications.

It is possible to ask simply to be notified about click coordinates, but a more interesting way is through *triggers*, which denotes *sensitive areas* of an image (it is used with the same meaning in [9]). Since these interfaces show mainly natural images, it is better to drive interaction through objects appearing in images, rather than on traditional buttons and the like.

The toolkit's image format also contains (optionally) the definition of triggers. When an image is mapped agents can request to be notified when the user clicks on its triggers. In that case, the name of the trigger will be included in the notification. This stands for a higher-level interaction.

The way such triggers are defined is a subject by itself. The toolkit allows a "gross" delimitation method by specifying a cube, that is, a 3D block. When the user clicks on a pixel, it must be in the 2D square *and* its depth must be in range in order to be considered as part of the trigger.

Sharper object delimitation is possible by using *color*. Color is an important way by which humans separate visual objects, and it was decided to evaluate it as a way of trigger delimitation. Thus a pixel, in order to be considered as belonging to a trigger, must also have its color within a specified cube in color space (this is another dependency between the two types of services).

RGB is not the best color space for this purpose. A desirable property for the needed color coordinate system is that small perceptual changes in color are mapped to small shifts in color space. But a human viewer is most sensitive to color shifts in the blue, moderately in the red, and least sensitive in the green [7].

A color space with the desired characteristics already exists, and is called UCS (Uniform Chromacity Scale). In UCS, numerical distance corresponds to human perceptual color distance, and brightness is a separate coordinate [7]. The color space we use is UCS, with brightness discarded (making it insensitive to illumination changes). The two remaining coordinates form the new color space. Actually, a square of 5×5, centered on the point, is averaged (filtering out artifacts) to determine the pixel's color.

4.3 Audio server

The audio server is responsible for providing a very high-level modelling of an audio stream, isolating the composer from the details.

A very important part of this modelling is to consider that stream as a clock, replacing real time. It is possible to label distinguished points in the audio stream, being the principal notification activity of this server to send appropriate messages to interested agents when those labels are reached during playback.

The most important request this server processes is to play a segment of an audio file (saved in a custom format, along with labels), at some volume and balance. A large number of format options are available: mono/stereo, data format, sampling rate, quantization scale, etc.

A given server can play one segment at a time. Once playing starts, it accepts requests for notification of labels, volume and balance changes (with fading effects, if desired), pause/resume or stop playing, etc.

Any agent may request to be notified of labels, and later cancel these requests. It may ask for a specific label or for all labels in the current segment.

Audio input devices are treated as normal files, thus allowing live audio to take part of a show. It may also be asked to save the current segment in another audio file. Both capabilities combined provide for audio recording.

Response to pause requests involves stopping playback until resume time (thus deferring labels to reach as it corresponds to a pause in time). Response to termination includes truncating playback and label notifications. In order to ease client's development, two special labels are guaranteed to be emited for each segment: `begin` and `end`.

5 Development methodology

This toolkit is still a kernel, that is, although it isolates the composer from all low-level details, it has to be accessed from C: it consists of a C-callable library. The composer, which still has to be a programmer, defines its entities and automata, its actions, requests and notifications, using toolkit-provided C-calls. However, the task of expressing this behavior in C is very easy once the design is done. A future project includes a higher-level graphical interface for avoiding C-coding, making the toolkit available also to nonprogrammers.

Creating a multimedia interface is a process that goes from capturing and generating multimedia data to the delivery of a final product (a simple or complex organization for the presentation of these data).

No unified methodology exists yet for multimedia interface composition. One of the purposes of this toolkit is to enforce a particular style of design.

Roughly speaking, the process can be divided into four stages [9] (although even this division is not universal):

Capture/Generation: In this stage, the basic blocks of multimedia data (images, audio pieces, video segments, graphics, text) may be captured or generated, depending on their nature. Audio and video may be recorded, text is written, graphics may be rendered and stored, etc.

Edition: The captured and generated pieces of data are edited. This stage may be confused with the previous one in the case of generated media, but is clearly distinguished in captured media. Audio pieces may be cut and pasted anywhere, mixed with another audio, filtered in order to reduce noise, etc. Images may be cut and pasted, merged, and filtered too. This is the stage for "special effects" (such as adding graphic overlays to video).

Composition: The many pieces of collected data are assembled and organized in a logical scheme. This scheme defines how the data will be presented along the time, how the interaction with the end user will be, and what the response to his input will be. The result may be as simple as a sequence of shows along a finite time interval, or as complex as an interface for a full system, including database front-ends, hypermedia document presentations, etc.

Run: In the final stage, the interface is run by the end-user, who will interact with it.

In fact, this scheme must be embedded in a prototype/final product process: it is necessary to begin by roughly designing the interface (composition stage), then to obtain "prototype" multimedia data until the composition is completed and it is exactly understood which multimedia data is needed. Only then the high-quality data is obtained and the final layout details are tuned.

This toolkit does not attempt to address the full four-stage process, but only the last two. Although it is possible to think in a full multimedia interface composition system (as [9]), it would be better to allow a great diversity of capture and edition hardware and tools, only focusing on format compatibility issues; and then to use a unique composition tool for composition stage (which, in general, should also include the necessary support for run stage).

The composer should start by identifying the windows and audio tracks which will make up his interface, assigning a server to each of them. Then, the compound behavior of the whole interface should be stated, and expressed in terms of a state machine (or many, if the behavior is better expressed concurrently), where events cause transitions among its states and actions to be performed. If real-time specifications are part of the behavior, a time server should also be included. Finally, events should be mapped to server notifications or messages among control agents, interface actions should be mapped to requests to servers, and automata of control agents should be defined.

Only then the implementation step is performed: the exact layout of windows, audio volumes, tight audio labels, etc. are depicted. Then the show is C-coded: automata are defined, servers are declared, and the show is started. The final process is to fine-tune all these details.

All this is very easy to do, and normally it takes a short while to build simple shows.

6 Implementation details

The toolkit has been implemented in C, on AIX (IBM's Unix) and its standard inter-process communication facilities.

The real-time server is a standard Unix process, whose scheduling policy makes hard to satisfy any kind of time guarantee.

The graphic server is built on XWindows and the IBM's version of GL (Graphics Language). GL allows to display directly RGB images, fast image transferring and graphics rendering, and 3D processing.

The audio server was implemented on IBM's ACPA (Audio Capture & Playback Adapter) card drivers, accessed through AIX facilities.

Our implementation is currently running on a network of IBM Risc System/6000 machines, connected with TCP/IP on a 16 Mbps token-ring. The RISC machines are POWERstation 530 model, with 30 MHz, 15 MFLOPS, 48M RAM. All audio files were stored on other machines and accessed via NFS.

A HP3D-SGP (High-Performance 3D Super Graphics Processor) card is used and accesed through GL. It allows to display 24-bit RGB images, and has a 24-bit zbuffer (to process depths).

Two ACPA cards are used, which allow two simultaneous audio servers to be in operation.

7 A simple case study

Let's now go into an example of the design and implementation process in a simple project. Suppose the show is to resemble an illustrated story tale book. We have stored (in some way) a set of stories, each one consisting of a sequence of images and an audio stream. There is a label in the audio stream for each image, indicating that that image should be shown. We want also to add the label text at the bottom of the image. The scenario is as follows: the user writes a story name, and then the story begins, showing images and playing audio. He has buttons to pause and resume the story, and to stop it at all. He also may turn the page forward or backward at any moment, and the audio should resynchronize with the new page.

The first thing to do is to identify the participant windows (i.e. graphic servers). There is a window at which the input of the story name will take place and will serve as the "title" throughout the story, let's call it `TITLE`. There is another where images will be shown, called `IMAGE`. Both of them will lay on a big container: `BACKGR`, which will have a number of sensitive "buttons" (triggers): `End` (to finish the show), `Stop` (to end the current story), `Pause` and `Cont` (to pause and resume the story), `Back` and `Forw` (to turn pages). Figure 2 shows the rough layout.

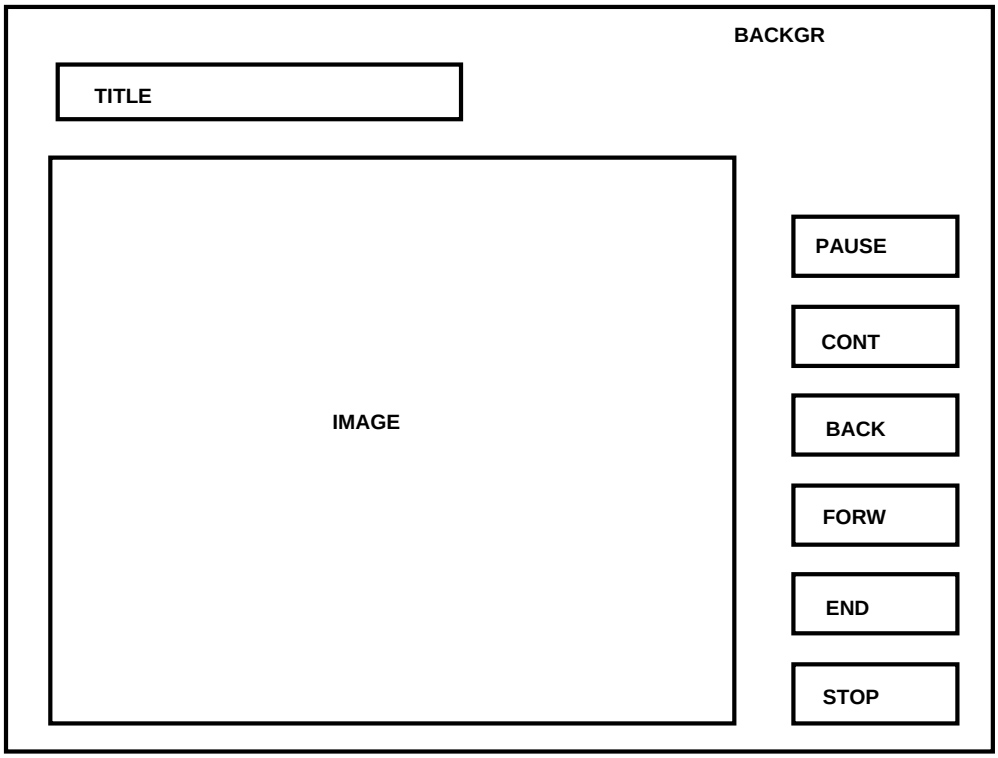

Figure 2: Rough layout of the interface.

We will also need an audio server for the speech, let's call it `AUDIO`. We don't seem to need more servers.

Which states will have our automaton? There is an `idle` state (not telling) and a `story` state (telling a story).

Which events do we expect in `idle` state?

- `End` pressed
- The user entered a new story name

What about `story` state?

- `End`, `Stop`, `Pause`, `Cont`, `Back` or `Forw` pressed

- A new audio label reached

Figure 3 depicts the automaton with all events, actions and transitions. Ovals are states, dotted lines are transitions. The actions on top of states are entry actions. The "when" clause near each state specifies its behavior against messages: arriving messages are indicated as right-to-left solid arrows, and the "code" below them are the actions to carry out in that case. Left-to-right solid arrows indicate messages to emit (they are actions too). Note that in both types of solid arrows, the other agent is identified at the right of the arrow.

Coding the show is a very simple step from the automaton design, being little more than to rephrase the graph as C calls (one call per event, one per emitted message, etc.).

8 Conclusions and Future Work

A number of projects are being carried out on the toolkit: demos; simple image, triggers and audio editors; hypermedias; etc. Much more use of the toolkit is needed in order to evaluate it extensively, so it is still early for clear-cut conclusions. In the tests made so far, the system did fulfill all expectations: interfaces are very easily programmed and debugged, their look and feel is very impressive and the interaction works well. Ideas such as the delimitation of triggers with colors worked fine; the CPU requirements posed by the toolkit can be accommodated by the RISC/6000 without noticeably slowing down the system; audio can be reproduced at very high rates without jitter (two ACPAs playing in parallel at 44.1k mono and 16 bits per sample were tested), performing the playback all the time from a remote file system; and time requests are fulfilled very accurately (error is under the millisecond).

On the other hand, a number of necessary improvements are becoming apparent:

Video: Video was not included in the first release, since we lack at this moment a video-generating or capture tool. There are two possible alternatives to support it in the future. The first one is graphic animations: to generate the frames one by one, as images, using a tool like TDI or the VCA (IBM's Video Capture Adapter) and a high-quality video player (to capture each frame). This approach presents a number of challenging problems (storage and CPU demands, jitter, etc. [8]). The second one is to use the ActionMedia II card [6], which will solve all low-level problems. Regarding the high-level interface of a video server, there are a number of interesting possibilities: video labels, video (temporal) triggers, etc. Indeed, video combines many audio and graphical features.

Trigger delimitation: Although the use of color works well, it is interesting to study alternative ways of "real-object" delimitation in images (such as image-processing techniques).

Jitter: When many graphical effects take place simultaneously, jitter begins to be noticeable. And it will become a real problem if software-rendered video is used. A general mechanism of CPU reservation, overriding Unix's scheduling, will be necessary (see [1]).

VCA and others: Graphic servers will benefit from the incorporation of image capture capabilities from cameras (and possibly scanners); new versions of ACPA drivers may incorporate MIDI; etc.

Editors: A wealth of editors, particular of this toolkit, are urgently needed: depth, transparencies and trigger editors for images; audio and audio labels; a video editor (when it becomes supported); etc.

Composition scripts: A high-level script language for composition is needed (perhaps graph-oriented), in order to make this toolkit available also to nonprogrammers. An integrated framework with script editor, interpreter, debugger and compiler is one of the main projects.

Merging with Motif: The incorporation of the graphic server as a widget class will ease the inclusion of multimedia features in traditional application development.

Acknowledgements

The first author is in deep gratitude with Ricardo Baeza-Yates, who not only pushed him to publish this paper, but also found time to review it and made a number of improvements.

We want also to thank the helpful comments of the referees.

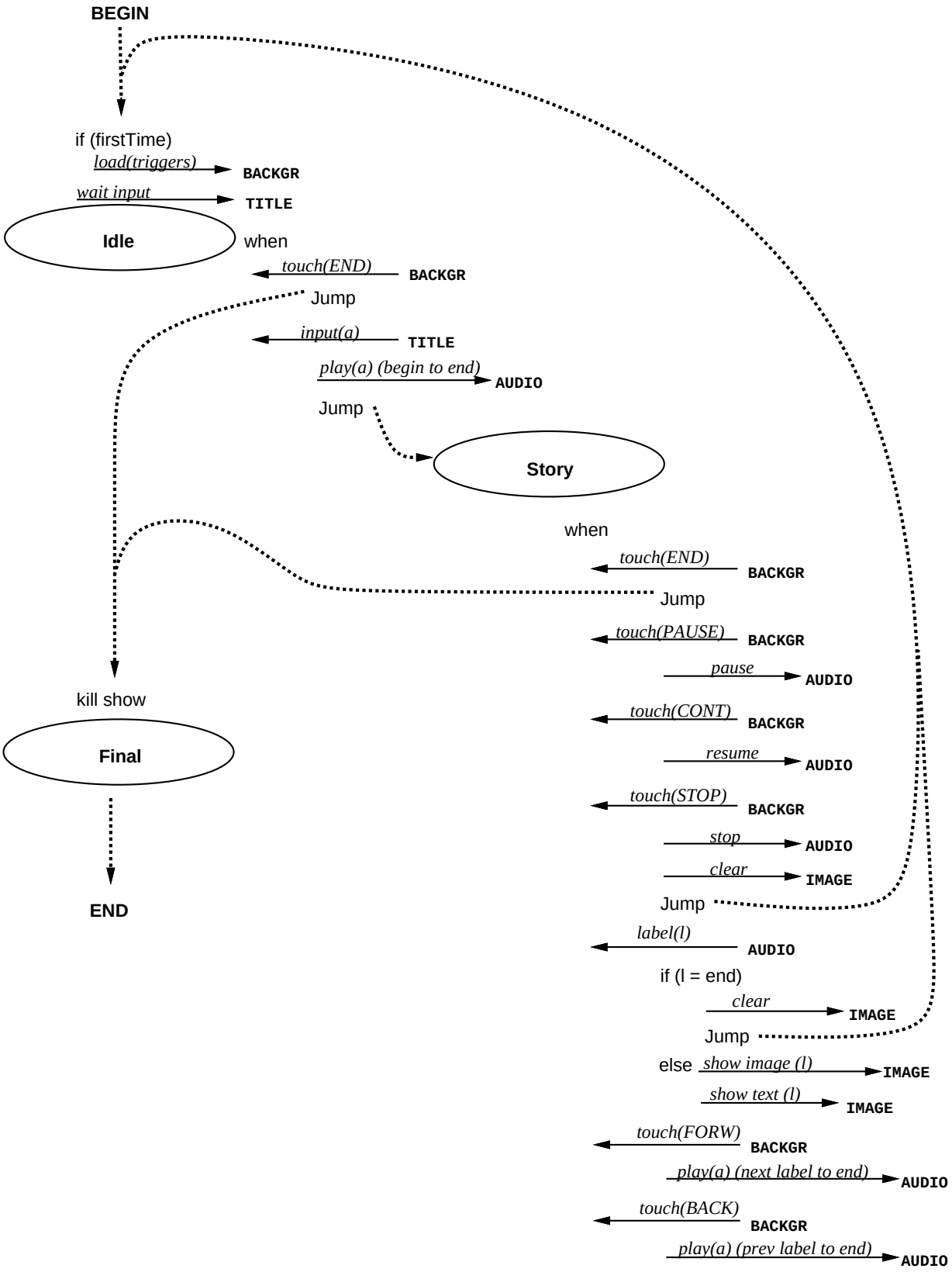

Figure 3: Automaton design for the interface.

References

[1] David P. Anderson. Meta-scheduling for distributed continuous media. Technical report, Computer Science Division, EECS Department, University of California at Berkeley, Berkeley, CA 94720, October 1990.

[2] Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanembaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–357, September 1989.

[3] Martin D. Davis and Elaine J. Weyuker. *Computability, Complexity and Languages.* Academic Press, Inc., Orlando, Florida 32887, 1983.

[4] J. D. Foley and A. van Dam. *Fundamentals of Interactive Computer Graphics.* Addison-Wesley Systems Programming Series. Addison-Wesley, Reading, Massachusetts, 1982.

[5] Edward A. Fox. Advances in interactive digital multimedia systems. *IEEE Computer*, pages 9–21, October 1991.

[6] K. Harney, M. Keith, G. Lavelle, L. D. Ryan, and D. J. Stark. The i750 video processor: A total multimedia solution. *Communications of the ACM*, 34(4):65–78, April 1991.

[7] Anil K. Jain. *Fundamentals of Digital Image Processing.* Prentice-Hall Information and System Series. Prentice-Hall, Englewood Cliffs, NJ 07632, 1989.

[8] M. Liebhold and E. M. Hoffert. Toward an open environment for digital video. *Communications of the ACM*, 34(4):104–112, April 1991.

[9] D. J. Moore. Multimedia presentation development using the Audio Visual Connection. *IBM Systems Journal*, 29(4):494–508, 1990.

[10] Adrian Nye. *Xlib Programming Manual*, volume 1. O'Reilly & Associates, Inc., 632 Petaluma Avenue, Sebastopol CA 95432, 1988.

[11] James Peterson and Abraham Silberschatz. *Operating System Concepts.* Addison-Wesley Series in Computer Science. Addison-Wesley, Reading, Massachusetts, 1984.

[12] Dennis Tsichritzis, Simon Gibbs, and Laurent Dami. Active Media. In Dennis Tsichritzis, editor, *Object Composition*, pages 115–132. Centre Universitaire d'Informatique, Université de Genève, Switzerland, June 1991.