VideoScheme: A Programmable Video Editing System for Automation and Media Recognition

James Matthews
Dartmouth College[1]

Peter Gloor
Massachusetts Institute of Technology[2]

Fillia Makedon
Dartmouth College[3]

Abstract

The recent development of powerful, inexpensive hardware and software support has made digital video editing possible on personal computers and workstations. To date the video editing application category has been dominated by visual, easy-to-use, direct manipulation interfaces. These systems bring high-bandwidth human-computer interaction to a task formerly characterized by slow, inflexible, indirectly-operated machines. However, the direct manipulation computer interfaces are limited by their manual nature, and can not easily accomodate algorithmically-defined operations. This paper proposes a melding of the common direct manipulation interfaces with a programming language which we have enhanced to manipulate digital audio and video. The result is a system which can automate routine tasks as well as perform tasks based on sophisticated media recognition algorithms.

1. Introduction: Digital Video Editing with Direct Manipulation

In "Virtual Video Editing in Interactive Multimedia Applications" Mackay and Davenport described a new reality for the producers and consumers of moving picture information: "video becomes an information stream, a data type that can be tagged and edited, analyzed and annotated"[3]. At the same time, the value of moving picture information has become increasingly clear in a wide range of fields, from education to business to scientific visualization. Software developers have seized this opportunity to produce software for manipulating this new data type: examples include Adobe's Premiere [5] and DiVA's VideoShop [1]. Premiere and VideoShop allow users to edit video data in much the same way that popular word processors allow users to edit text. Video is represented by visual proxies, typically thumbnail images and graphic representations of audio waveforms. Users can use a mouse to click on desired movie clips and drag them into place. VCR-style buttons can be used for playback, and a collection of other metaphorical tools (e.g. scissors, magnifying glass, trash can) are available.

In one sense these tools take the video editor back to the time before videotape, when film editors held their media in their hands and edited it without the intermediate presence of video decks and time codes. But the new digital systems offer advantages beyond a more direct interaction with the media. Thanks to random-access storage devices these systems let the editor manipulate many sections of video simultaneously, with quick jumps to any point in the source material. There is no penalty to repeated editing, since the digital information does not degrade. The user interface can be tailored to use common metaphors and standard commands for the computer platform in question. The result is an environment where casual experimentation is encouraged, and beginners can quickly produce acceptable results.

The direct manipulation nature of these systems, however, also limits user options. Some repetitive or complex functions can not be expressed with the provided tools. A user can visually select and delete a period of silence in an audio track, but in a pure direct manipulation interface there is no way to abstract that specific operation into a more general command ("if there is silence, delete audio data") that can be applied repetitively. The ability to evaluate conditions (e.g. "is this audio data silent", "is this a scene transition") is left to human eyes and ears, when the computer might be able to do the job more quickly or accurately. And the user is limited to the operations that the system designer considered important; an unusual function, or combination of functions may be completely out of reach, and no designer can imagine or implement all the functions that might prove to be useful.

---

[1]Department of Mathematics and Computer Science, Dartmouth College, Hanover, NH 03755. Jim.Matthews@dartmouth.edu.

[2]Laboratory for Computer Science, MIT, 545 Technology Sq., NE43-506, Cambridge, MA 02139. gloor@lcs.mit.edu.

[3]Department of Mathematics and Computer Science, Dartmouth College, Hanover, NH 03755. Fillia.Makedon@dartmouth.edu.

This work was partially supported by the Dartmouth Experimental Visualization Laboratory.

2. Related Work

Researchers have attempted to address the shortcoming described above, both in the wider domain of direct manipulation software and in the specific area of video editing systems. Eisenberg's SchemePaint system combines a simple painting program with a Scheme interpreter [2]. The result is an immediately useable system that can also be programmed to produce results that are not possible with a strictly manual interface. Eisenberg argues for the incorporation of domain-enriched programming languages into a wide variety of direct manipulation systems, including video editing systems.

Ueda, Miyatake, and Yoshizawa's IMPACT system attempts to enrich the video editing interface from another direction. Rather than making it programmable in a general way IMPACT's designers supplement the direct manipulation interface with powerful functions that exploit image processing and analysis algorithms [3]. The IMPACT system includes functions to identify "cuts" in a stream of video, to classify cuts (for example as a zoom-in, or pan-left), and to extract moving objects from scenes.

Our effort, called VideoScheme, borrows from both of these examples: we have built an extensible video editing system which provides the user with programming capabilities. By embedding a program interpreter into a direct manipulation video editing system we hope to achieve the flexibility and expressiveness demonstrated by SchemePaint. In particular, we hope to make it possible to implement the dedicated media-analysis functions of the IMPACT system in this programming language, yielding advantages in both power and flexibility.

3. VideoScheme – The System

The core of the VideoScheme system is a simple direct manipulation video editor. It is implemented on the Apple Macintosh, using Apple's QuickTime software to handle video storage, compression, and decompression [6]. Video files are opened into windows that display the movie in a timeline format. Video tracks are represented by a sequence of selected frames, and audio tracks are represented by a graphical view of the sound waveform. The user can scroll back and forth in time, and change the display scale to view anything from a fraction of a second to several minutes of video.

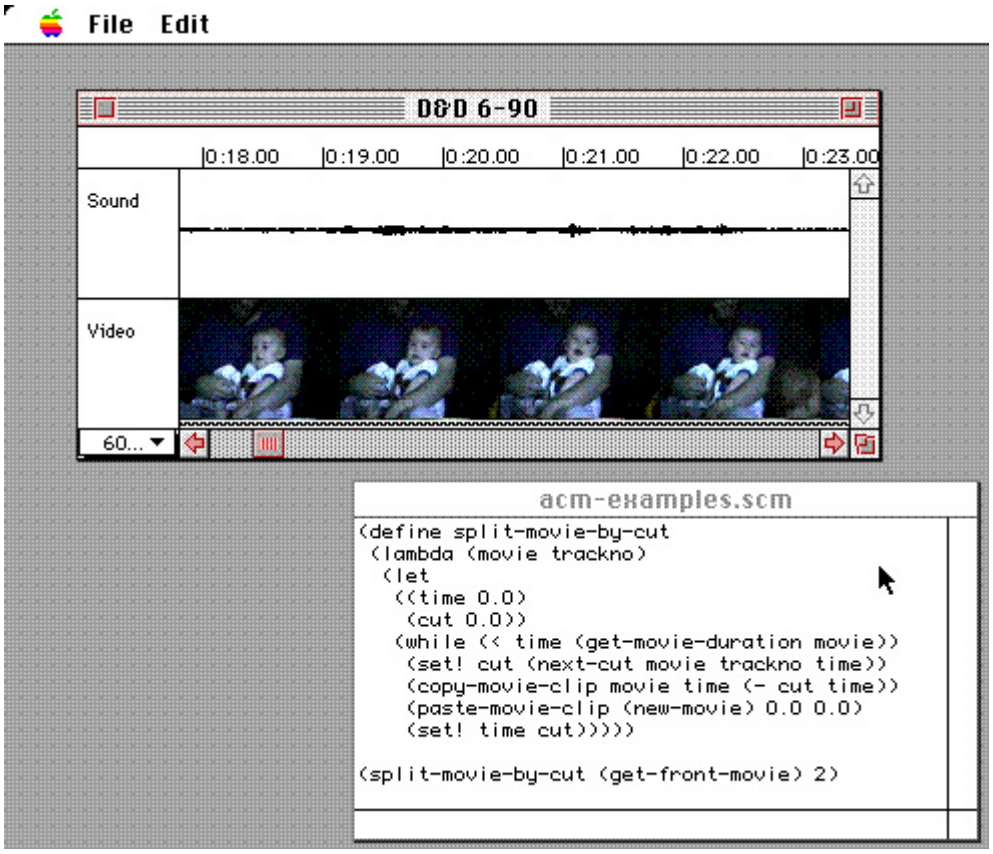

Figure 1: The VideoScheme System

Embedded in this interactive program is a simple Scheme programming environment, built around the publicly-available SIOD interpreter [7]. Expressions are edited and evaluated in text windows, which also collect program output. These text windows co-exist with the video windows, allowing very quick switches between manual editing operations and programming (see Figure 1).

We chose the SIOD Scheme interpreter for its small size, support of array data types, and its extensibility. This last feature made it possible to add new built-in functions which bridge the gap between the Scheme environment and video editing. A partial list of the provided functions appears in Figure 3. The functions are designed to be independent of the lower-level QuickTime-based implementation; they could be re-implemented on another platform, to allow for portability of VideoScheme programs.

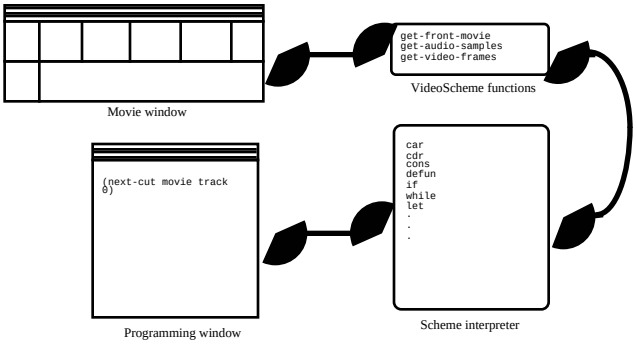

Figure 2: VideoScheme's functional layout

```
(new-movie)                          -- return the id of a new movie

(open-movie movie-file)              -- open and return the id of a movie file

(get-front-movie)                    -- return the front movie's identifier

(get-movies)                         -- return a list of open movies

(get-movie-duration movie)           -- return a movie's length in seconds

(get-next-frame-time movie trackno time)
                                     -- return the time stamp of the next frame

(get-audio-samples movie trackno time duration samples)
                                     -- fill in samples array with audio data

(get-video-frame movie trackno time pixels)
                                     -- fill in pixels array with image data

(get-color-histogram64 pixels histogram)
                                     --  fill  in  histogram  array  with  pixels's
                                        64-element color histogram

(clear-movie-clip movie time duration)
                                     -- delete the specified movie segment

(cut-movie-clip movie time duration)
                                     -- move the segment to the clipboard

(copy-movie-clip movie time duration)
                                     -- copy the segment to the clipboard

(paste-movie-clip movie time duration)
                                     --  replace  the  segment  with  the  clipboard
                                        contents
```

Figure 3: VideoScheme functions for manipulating video data

4. Automating Repetitive Tasks

The automation of repetitive, well-defined tasks can be accomplished with simple programming language constructs. A video editor may want to divide a long video sequence into smaller, equal-sized chunks. By hand this would be a tedious, error-prone process, but in VideoScheme it can be performed with a simple program:

```
(define split-movie
 (lambda (movie chunk-size)
  (let
   ((time 0.0))
   (while (< time (get-movie-duration movie))

    ; copy the next chunk of the movie
    (copy-movie-clip movie time chunk-size)

    ; paste it into a new movie
    (paste-movie-clip (new-movie) 0.0 0.0)

    (set! time (+ time chunk-size))))))
```

Function 1: split-movie

With the function thus defined, splitting the frontmost movie into one-minute-sections is as simple as evaluating the expression `(split-movie (get-front-movie) 60)`.

Similarly, a single new movie can be created by choosing multiple excerpts from an existing movie. Indeed, by this simple formula we can achieve the effect of speeding up the movie:

```
(define speedup
 (lambda (movie factor)
  (let
   ((time 0.0)
    (new (new-movie)))
   (while (< time (get-movie-duration movie))

    ; copy a fraction of the next tenth
    ; of a second
    (copy-movie-clip movie time (/ 0.1 factor))

    ; paste it at the end of the new movie
    (paste-movie-clip new
     (get-movie-duration new)
     (get-movie-duration new))

    (set! time (+ time 0.1)))
   new)))
```

Function 2: speedup

So the expression `(speedup (get-front-movie) 2)` returns a version of the frontmost movie that appears to run at double speed, since every-other twentieth of a second has been removed from it. An analogous function could be written to duplicate information in a movie rather than removing it, yielding a slow-motion effect.

We can make these mechanical functions sensitive to the structure of the movie with the aid of VideoScheme's built-in functions. One of these, `get-next-frame-time`, returns a list consisting of the timestamp and duration of the next frame in a video track. With this function we can copy a movie on a frame by frame basis, reconstructing it in reverse order:

```
(define reverse
 (lambda (movie trackno)
  (let
   ((time 0.0)
    (frame-info nil)
    (duration 0.0)
    (new (new-movie)))
   (while (< time (get-movie-duration movie))

    ; find out when the next frame starts,
    ; and how long it lasts
    (set! frame-info
     (get-next-frame-time movie trackno time))
    (set! time (car frame-info))
    (set! duration (car (cdr frame-info)))

    ; copy the next frame
    (copy-movie-clip movie time duration)

    ; paste it at the beginning of the new movie
    (paste-movie-clip new 0.0 0.0)

    (set! time (+ time duration)))
  new)))
```

Function 3: reverse

5. Media Recognition

Simple, repetitive functions are called for in some circumstances, but their power is clearly limited by their simplicity. Most video editing decisions must take the content of the video data into account, and likewise more powerful VideoScheme functions can be created when the media itself is consulted. VideoScheme includes two built-in functions for accessing the movie data: `get-audio-samples` and `get-video-frame`. Each of these returns arrays of integers: 8-bit sound samples in the case of `get-audio-samples`, and 24-bit color pixel values in the case of `get-video-frame`. These arrays can then be analyzed and the results used to create new editing functions.

One straight-forward application is to search through video data for periods of silence. We can characterize silence as a period where none of the audio samples has an amplitude greater than 10 (out of a maximum amplitude of 128). The returned samples are in the range 0-255, where the sample value 128 an amplitude of zero. Therefore our silence predicate looks like the following:

```
(define silence?
 (lambda (movie trackno time interval)
  (let
   ((samples (cons-array 0 'long)))

   ; get an array of audio samples
   (get-audio-samples movie trackno time
    interval samples)

   ; compute their absolute amplitudes
   (adiff samples 128 samples)
   (aabs samples samples)

   ; is the loudest sample less than 10?
   (< (amax samples) 10))))
```

Function 4: silence?

This predicate may prove unreliable with noisy audio sources; in that case examining the median amplitude, or a certain percentile might prove more effective. These possibilities can be easily explored with VideoScheme.

Periods of silence may be interesting to a video editor, as they often represent transitions. Visual transitions, or "cuts," are also likely to be of interest, and VideoScheme can be programmed to automate the process of finding these transitions. Nagasaka and Tanaka have investigated automatic cut detection algorithms, obtaining the best results with a test that measures the differences in color distributions between adjacent frames [4]. Following their algorithm we can write a function to compute the normalized difference between two histograms:

```
(define histogram-difference
 (lambda (hist1 hist2)
  (let
   ((hist-diff (cons-array 0 'long)))

   ; subtract the two histograms
   (adiff hist1 hist2 hist-diff)

   ; square the difference
   (atimes hist-diff hist-diff hist-diff)

   ; normalize by one of the histogram arrays
   (aquotient hist-diff hist1 hist-diff)

   ; sum the squared, normalized differences
   (atotal hist-diff))))
```

Function 5: histogram-difference

This function makes use of VideoScheme's built-in array functions to subtract, square, normalize, and sum the histogram differences. We can compute the histograms themselves using a built-in function, making it a simple matter to compute the visual continuity at any point:

```
(define full-frame-diff
 (lambda (movie trackno time1 time2)
  (let
   ((pixels (cons-array 0 'long))
    (hist1 (cons-array 64 'long))
    (hist2 (cons-array 64 'long)))

    ; get the histogram for one frame
    (get-video-frame movie trackno time1 pixels)
    (get-color-histogram64 pixels hist1)

    ; get the histogram for another
    (get-video-frame movie trackno time2 pixels)
    (get-color-histogram64 pixels hist2)

    ; compare the histograms
    (histogram-difference hist1 hist2))))
```

Function 6: full-frame-diff

Nagasaka and Tanaka found this function to be sensitive to momentary image noise, which typically affected only parts of the image but created undesirable spikes in the color continuity. They eliminated this effect by dividing the frames into 16 subframes, comparing the subframe histograms, and discarding the 8 highest difference totals. We can implement this improved algorithm in VideoScheme:

```
(define nagasaka-tanaka-diff
 (lambda (movie trackno time1 time2)
  (let
   ((pixels1 (cons-array 0 'long))
    (pixels2 (cons-array 0 'long))
    (sub-pixels (cons-array 0 'long))
    (hist1 (cons-array 64 'long))
    (hist2 (cons-array 64 'long))
    (diffs (cons-array 16 'long))
    (frame1 nil)
    (frame2 nil)
    (index 0))

    ; get the two frames in question
    (set! frame1 (get-video-frame movie trackno time1 pixels1))
    (set! frame2 (get-video-frame movie trackno time2 pixels2))

    (set! index 0)
    (while (< index 16)
     ; histogram one 16th of frame1
     (get-sub-frame16 frame1 index sub-pixels)
     (get-color-histogram64 sub-pixels hist1)

     ; histogram one 16th of frame2
     (get-sub-frame16 frame2 index sub-pixels)
     (get-color-histogram64 sub-pixels hist2)

     ; remember the difference
     (aset diffs index (histogram-difference hist1 hist2))
     (set! index (+ index 1)))

    ; order the subframe differences and discard the 8 highest ones
    (asort diffs)
    (asetdim diffs 8)

    ; total the remaining differences
    (atotal diffs))))
```

Function 7: nagasaka-tanaka-diff

A number of applications can be built using this measurement of visual continuity. A simple function can search a movie for the beginning of the next cut:

```
(define next-cut
 (lambda (movie trackno time)
  (let
   ((diff 0))
   (while
    (and
     (< diff 10000)
     (< time (get-movie-duration movie)))

    (set! diff
     (nagasaka-tanaka-diff movie trackno time
      (+ time 0.1)))
    (set! time (+ time 0.1)))
   time)))
```

Function 8: next-cut

We can modify the split-movie function presented earlier to split a movie on scene boundaries rather at a fixed interval:

```
(define split-movie-by-cut
 (lambda (movie trackno)
  (let
   ((time 0.0)
    (cut 0.0))
   (while (< time (get-movie-duration movie))

    ; find the next cut
    (set! cut (next-cut movie trackno time))

    ; copy up to the next cut
    (copy-movie-clip movie time (- cut time))

    ; paste the segment into a new movie
    (paste-movie-clip (new-movie) 0.0 0.0)

    (set! time cut)))))
```

Function 9: split-movie-by-cut

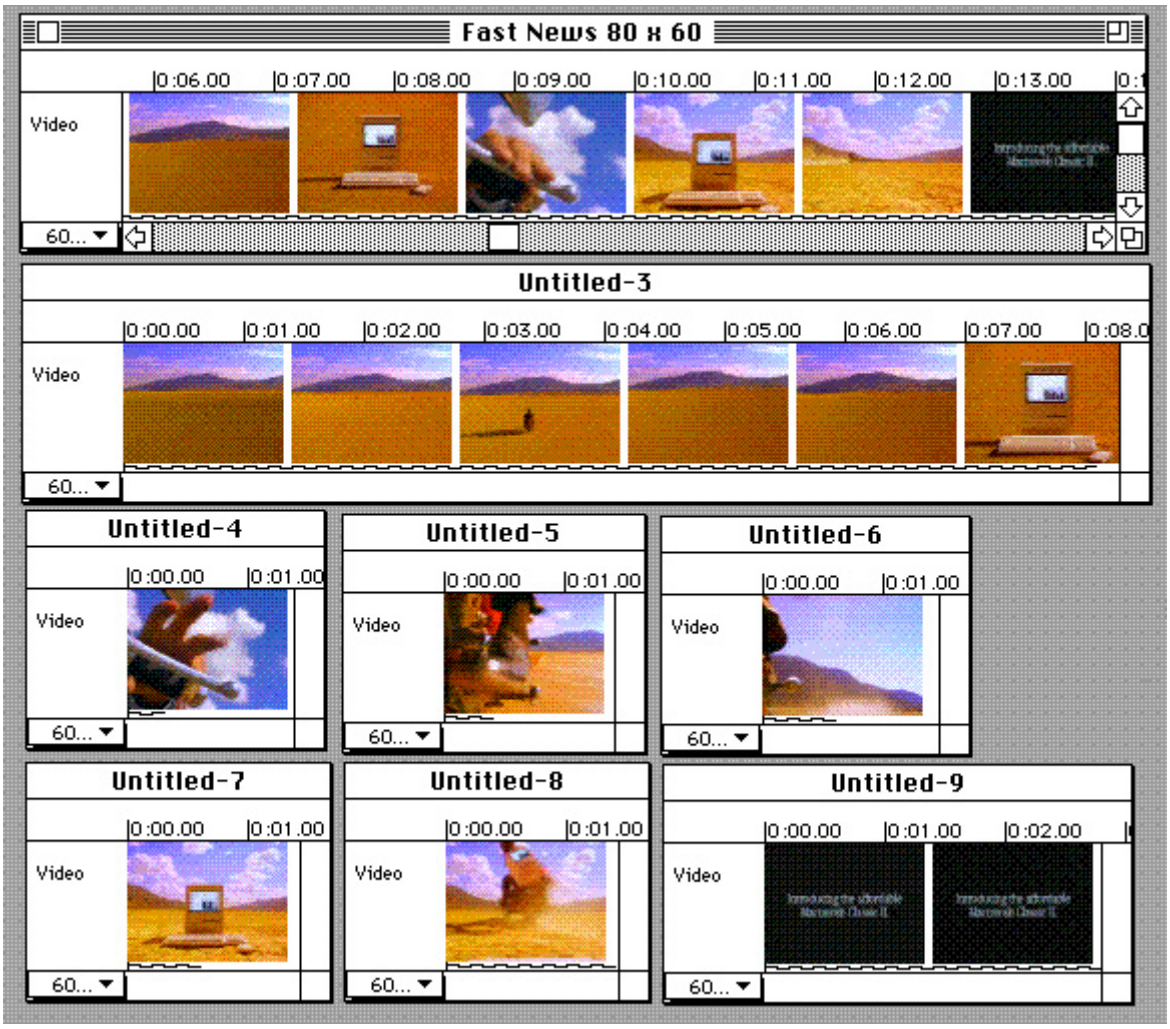

Figure 4: Results of `split-movie-by-cut`

The results of executing the `split-movie-by-cut` function on a fifteen second TV commercial are shown in Figure 4. The movie "Fast News 80 x 60" has been split into nine segments, seven of which are shown. In one case the cut-detection algorithm has performed better than the naked eye: the cut between segment "Untitled-7" and "Untitled-8" is almost undetectible when the movie is viewed at normal speed, but close examination and the Nagasaka-Tanaka algorithm reveal the cut.

Using other knowledge of how video is sometimes structured, we can detect even higher level boundaries, such as television commercials (which can be characterized by scene changes exactly 15, 30, or 60 seconds apart). We can also detect common editing idioms: the expression `(nagasaka-tanaka-diff movie track time (next-cut movie track time))` evaluates the visual continuity between the frames that bracket a cut. A high degree of continuity suggests that the editor is cutting back and forth between two video segments, for example footage of two different characters speaking.

6. Conclusions

We have implemented a first prototype of VideoScheme. Our short term goal is to evaluate the usefulness of our system on some real-world editing tasks. We hope that this experience will suggest new operations to implement in VideoScheme, and clarify the limitations of the prototype.

In addition we plan to investigate new primitives, and thereby to expand the range of functions that can be efficiently implemented in VideoScheme. With these primitives we hope to tackle even more complex algorithms, such as Nagasaka and Tanaka's object search algorithm, cut-classification algorithms, and algorithms that edit the audio samples and video frames to produce special effects.

We have seen that it is possible to achieve some of the results of dedicated video authoring systems such as IMPACT in a programmable editing system with a small number of special-purpose video functions. Our VideoScheme system offers the further advantage of flexibility: it is a simple matter to experiment with new algorithms, and to build new operations by combining previously written functions. These are the classic advantages of programming, and we believe that we have shown them to be equally valid in the domain of interactive video editing.

We believe that VideoScheme is a unique attempt at achieving the best of both worlds: the ease of use of direct manipulation and the flexibility and expandability of an interpreted programming language.

References

[1] *DiVAVideoShop*. DiVA Corporation. Cambridge, MA.

[2] Eisenberg, M. "Programmable Applications: Interpreter Meets Interface." MIT Artificial Intelligence Memo 1325, October 1991.

[3] Mackay, W. and Davenport, G. "Virtual Video Editing in Interactive Multimedia Applications." *Communications of the ACM*, 32:7, July 1989.

[4] Nagasaka, A. and Tanaka, Y. "Automatic Video Indexing and Full-Video Search for Object Appearances." *IFIP Transactions A (Computer Science and Technology)*, vol. A-7, 1992.

[5] *Premiere*. Adobe Systems Incorporated. Mountain View, CA.

[6] *QuickTime*. Apple Computer, Inc. Cupertino, CA.

[7] *SIOD (Scheme-in-one-Defun)*. Paradigm Associates, Inc. Cambridge, MA.

[8] Ueda, H., Miyatake, T., and Yoshizawa, S. "IMPACT: An Interactive Natural-Motion-Picture Dedicated Multimedia Authoring System." *CHI'91 Conference Proceedings*, 1991.