Passive Capture and Structuring of Lectures

Sugata Mukhopadhyay, Brian Smith
Department of Computer Science
Cornell University
Ithaca, NY 14850
{sugata, bsmith}@cs.cornell.edu

ABSTRACT
Despite recent advances in authoring systems and tools, creating multimedia presentations remains a labor-intensive process. This paper describes a system for automatically constructing structured multimedia documents from live presentations. The automatically produced documents contain synchronized and edited audio, video, images, and text. Two essential problems, synchronization of captured data and automatic editing, are identified and solved.

Keywords: Educational technology, audio/video capture, matching.

1. INTRODUCTION
Recent research has lead to advances in software tools for creating multimedia documents. These tools include toolkits and algorithms for synchronization [8], [9], authoring tools [10], [11], [12], and standards for document representation [13]. Despite these advances, creating multimedia presentations remains primarily a manual, labor-intensive process.

Several projects are attempting to automate the entire production process, from the capture of raw footage to the production of a final, synchronized presentation. The Experience-on-Demand (EOD) project at CMU (part of the Informedia project [14], [15]) is one of the most ambitious of these projects. Its goal is to capture and abstract personal experiences, using audio and video, to create a digital form of personal memory. Audio, video, and position data are captured as individuals move through the world. This data is collected and synthesized to create a record of experiences that can be shared.

The goal of the Classroom 2000 (C2K) [7] project at Georgia Tech is also to automate the authoring of multimedia documents from live events, but in the context of a more structured environment: the university lecture. C2K researchers have outfitted classrooms at Georgia Tech with electronic whiteboards, cameras, and other data collection devices. These devices collect data during the lecture and combined it to create a multimedia document that documents the activities of the class.

Both EOD and C2K automatically capture and author multimedia documents based on real world events. One way to understand the difference between them is based on the taxonomy shown in figure 1. One difference is that C2K uses *invasive* capture, while EOD uses *passive* capture. During invasive capture, the presenter is required to take explicit actions to aid the capture process, whereas a passive capture operates without such assistance. In C2K, speakers must load their presentations into the system before class and teach using electronic whiteboards. Although this simplifies capture and production, it constrains the lecturer's teaching style. Speakers must work within the limitations of electronic whiteboards and the C2K software. In EOD, the recording devices do not constrain an individual's actions.

Another difference is that EOD captures *unstructured* environments, whereas C2K captures *structured* environments. This difference allows C2K to produce better documents than EOD because the automatic authoring system can be tailored to a specific environment.

		Environment	
		unstructured	**structured**
Capture	**invasive**		Classroom 2000
	passive	Experience on Demand	Lecture Browser

Figure 1: Automatic Authoring Systems

This paper describes an approach to the automatic authoring problem that combines these approaches. The *Cornell Lecture Browser*, automatically produces high-quality multimedia documents from live lectures, seminars, and other talks. Like C2K, we capture a structured environment (a university lecture), but like EOD we use passive capture. Our ultimate goal is to automatically produce a structured multimedia document from any seminar, talk, or class without extra preparation by the speaker or changes in the speaker's style. Our goal is to allow a speaker to walk into a lecture hall, press a button, and give a presentation using blackboards, whiteboards, 35mm slides, overheads, or computer projection. An hour later, a structured document based on the presentation will be available on the Web for replay on demand.

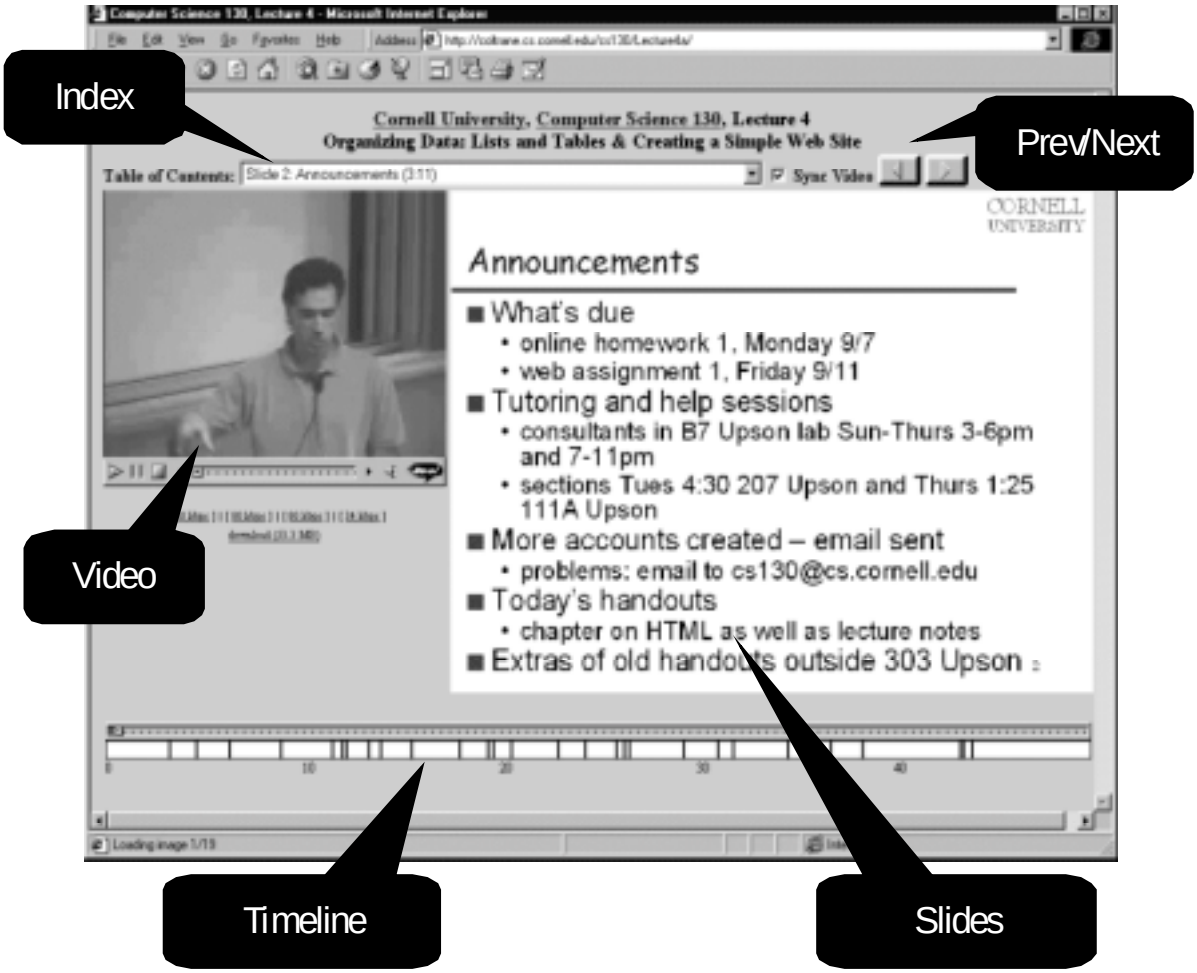

Figure 2: The Lecture Browser user interface

1.1. Overview of operation

Briefly, the system operates as follows[1]. Two cameras capture video footage, which is digitized and encoded in MPEG format. The *overview camera* captures the entire lecture dais from which the presenter lectures. The *tracking camera*, which contains a built-in hardware tracker that "follows" the speaker, captures a head-and-shoulders shot of the presenter. Figures 2 and 6 show typical shots captured by these cameras. At the end of the lecture, the footage is transmitted over a network to a *processing server*. The speaker also uploads the electronic *slides*[2] to this server. The server combines the collected video, slides, and HTML to create a presentation that can be viewed using Netscape Navigator or Internet Explorer with the RealVideo plug-in.

The user interface for viewing the lectures is shown in figure 2. The *video* region uses the RealVideo player's embedded user interface to play a video that alternates between footage captured by the two cameras. We have found that cutting between camera angles gives the viewer a sense of presence and creates the illusion that the video was edited by hand. The resulting presentations are more engaging than those captured using a single camera.

The *slide* region displays high-resolution images uploaded by the speaker, and the *index* displays the title and duration of the current slide in a drop-down list. The slides are synchronized with the video -- when the video shows the speaker flipping a slide, the slide display is updated. Similarly, jumping to a particular slide using the index or the *prev/next* buttons causes the video to jump to the corresponding footage.

The *timeline* provides another way of navigating the presentation. The thin timeline under the video window indicates the current playback point. The thicker, white timeline at the bottom of the browser window reflects the structure of the presentation. Each white box represents a slide, and each black vertical separator represents a slide change. Thus, small boxes represent slides displayed briefly, larger boxes represent slides displayed longer. Moving the mouse over one of these boxes displays the title of the corresponding slide, and clicking the mouse on a box advances the presentation to that slide. Thus, the two timelines provide

[1] We describe the system in the context of capturing a PowerPoint presentation, and discuss extensions to using other presentation media (blackboards, overhead projectors, etc) later.

[2] Using, for example, the "Save As HTML" feature in PowerPoint, which exports both text and GIF versions of the slides.

random access and visually indicate of the structure of the presentation.

1.2. Contributions

While building the Lecture Browser, we identified two essential problems that will be encountered in any system that collects and combines data from multiple independent capture devices. The articulation of the problems, and our solutions to them, comprise the main contributions of this paper. These problems are:

Synchronization. Before the collected data can be combined, the timing relationships between the data items must be established. In other words, the sources must be synchronized. Synchronization, in the context of the Lecture Browser, is tantamount to arranging the collected data on a timeline. Note that synchronization, in this context, is transitive. If we know the position of A on a timeline, and determine the timing relationship between A and B, we clearly know the position of B on the timeline. This property allows data items to be synchronized pair-wise.

Care must be taken when applying pair-wise synchronization because synchronization errors are cumulative. That is, if the uncertainty in the synchronization between A and B is $\varepsilon 1$, and the uncertainty in the synchronization between B and C is $\varepsilon 2$, then the uncertainty in the synchronization between A and C is $\varepsilon 1+\varepsilon 2$.

We can classify the collected data as being *timed* (*T-data*) or *untimed* (*U-data*). T-data contains timing information, U-data does not. Captured video is an example of T-data, since each frame has an associated timestamp. Electronic slides are an example of U-data, since they contain no timing information. With this classification, three types of synchronization problems can be identified (figure 3):

Timed-timed synchronization (TTS) establishes the temporal relationship between two T-data items. In the Lecture Browser, TTS arises because we independently capture two video streams. One might think that the stream could be synchronized by initiating recording simultaneously, but in practice, the MPEG encoders start recording up to five seconds after the command to begin is sent to the encoder. In order to allow frame accurate editing of the two videos, the temporal correspondence between them must be established. We describe a simple and efficient solution to this problem in section 2.

Timed-untimed synchronization (TUS) establishes the temporal relationship between a T-data item and a U-data item. The synchronization of the slides with a video stream is an example of TUS. In section 3, we present an algorithm that matches the slides to the video captured by the overview camera with 97% accuracy.

In *untimed-untimed synchronization* (UUS), both data items are untimed. Synchronization of the slide titles (used to build the index in figure 2) with the slides is an example of UUS. We solve this problem by parsing the HTML produced by PowerPoint. This output is composed of a set of HTML and GIF files. Each slide is represented by an HTML file that displays the GIF file for the slide and interface controls. The synchronization relationship between the slide (GIF file) and the title can be established directly from the HTML file.

When formulating synchronization problems, it is also important to specify the allowable tolerances so the cumulative effect of possible synchronization methods can be evaluated. For example, the video streams in the Lecture Browser must be synchronized accurately enough to allow frame-accurate editing (about 33 ms), whereas the synchronization between the overview video and the slide masters can be looser (we use 0.5 seconds). As we shall see later, the required tolerance can strongly affect the performance and complexity of the synchronization method.

Automatic Editing. Once the two videos, slide titles and slide images are synchronized, rules are used to produce the structured presentation. In the Lecture Browser, the captured MPEG data must be edited to create the finished video stream. We describe the heuristics used to edit the video data in section 4.

The automatic authoring process can now be simply stated. The collected data is synchronized in pairs. The two video streams are synchronized first, as described in the next section. The start of the video captured by the overview camera is arbitrarily set as the beginning of the timeline. Next, the slides are synchronized with

	timed	**untimed**
timed	TTS	TUS
untimed	TUS	UUS

Figure 3: Types of multimedia synchronization

the video captured by the overview camera, as described in section 3, which allows us to position the slides and titles on the timeline. Lastly, the two videos are combined, as described in section 4, and the HTML that combines the media into a presentation is generated.

The rest of this paper describes our methods for TTS, TUS, and automatic editing in sections 2-4, and concludes with a discussion of related work and extensions in sections 5 and 6.

2. TIMED-TIMED SYNCHRONIZATION

When independent capture devices are used to record a presentation, the temporal correspondence between the streams must be computed before the streams can be combined. The situation is illustrated in figure 4. More formally, let a temporal

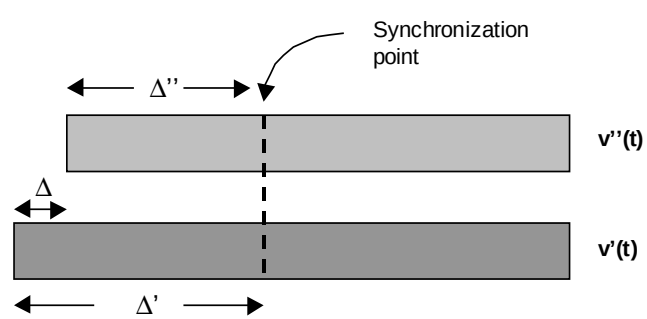

Figure 4: The timed-timed synchronization problem

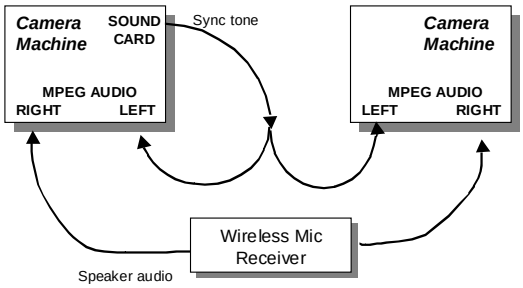

Figure 5: The timed-timed synchronization hardware setup

sequence be denoted by v(t), $t \geq 0$. Given two sequences v'(t) and v''(t), and a tolerance value of ε, compute Δ such that $v'(t + \Delta) \cong v''(t \pm \varepsilon)$, where ≅ denotes the correspondence operator (i.e., the footage was captured at the same instant)

To solve this problem, we establish one or more *synchronization points* within the streams. Synchronization points are events in both streams whose correspondence can be computed. By computing the positions Δ' and Δ'' of the synchronization points in v'(t) and v''(t) respectively, we get $\Delta = \Delta' - \Delta''$. If ε' and ε'' are the errors in Δ' and Δ'', respectively, then the uncertainty in Δ is ε' + ε''.

In the Lecture Browser, we artificially create a synchronization point by generating a one-second *synchronization tone* and recording it in one channel (e.g. the right) of the MPEG streams. A schematic of the hardware connections is shown in figure 5. The output from the lecturer's wireless microphone is fed into the right audio channel of each MPEG encoder. The left channel is connected to the output of a sound card.

After all encoders in the system have begun recording, we generate the synchronization tone using the sound card. This tone arrives simultaneously at the audio input of each MPEG encoder, and is recorded. Later, during processing, we detect the position of the tones in each stream, which gives us Δ' and Δ'' directly.

This brute force approach of detecting the synchronization tone requires fully decoding the appropriate MPEG audio track. Since decoding MPEG audio data is an expensive operation, we use a more efficient compressed domain method to search for the tone.

Our method is as follows. MPEG audio data is encoded as a series of *packets* [17]. Each packet contains *scale factors* that indicate an overall volume scale for the packet. We can estimate the volume of a packet by summing the scale factors. The rising edge of the synchronization tone in the otherwise silent stream, is in a packet where this sum exceeds a fixed threshold. The position of the packet in the stream is taken to be Δ' and Δ''. Assuming the stream is encoded using MPEG layer II, each packet contains 1152 samples. The worst case error in Δ' and Δ'' is therefore 22.5 * 1152 microseconds, or about 26 milliseconds. The total error ε is consequently 52 ms.

For frame accurate editing of video recorded at 30 frames per second, ε is required to be less than 33 milliseconds. To achieve this tighter bound, we decode one of the packets (e.g., from stream v'). The rising edge of the synchronization tone can be located within this packet to the accuracy of a single audio sample. Assuming the stream is recorded at 22.5 kHz, for example, the error in ε' is reduced to about 45 microseconds. The overall error ε is therefore about 26 milliseconds. Note that at a target frame rate of 15 frames per second (commonly used for streaming video), the error tolerance ε becomes 66 ms, so we can forego the extra decoding step. If more accuracy is required (e.g., to bound the error when synchronizing many video sources), the extra decoding step can be performed on both streams.

Our implementation, which is written in Dalí [1], can locate a synchronization tone in an MPEG system stream at over 35 times real-time speed on a Pentium-II 333. Thus, a tone 70 seconds into the stream can be located in less than 2 seconds.

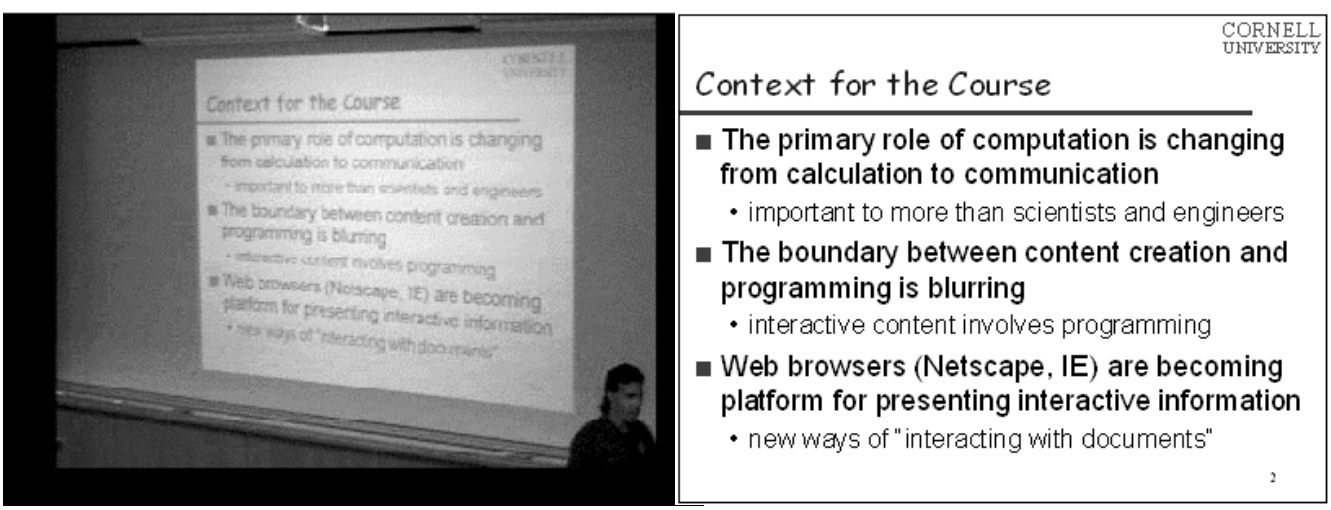

Figure 6: A typical frame from the overview video and the corresponding high-resolution electronic slide

3. TIMED-UNTIMED SYNCHRONIZATION

Having synchronized the video data, our next step is to synchronize the slides with one of the videos. We solve this TUS problem by determining which slide appears in each frame of the video captured by the overview camera. Recall that the overview camera captures the entire stage including the slide, and does not move during the presentation. A typical frame from the overview camera, and the corresponding electronic slide, are shown in figure 6. Notice that the video image of the slide is fuzzy, foreshortened, and has low contrast. Despite these problems, we have developed a method that can synchronize the slides to the overview video with high certainty.

More formally, given a video sequence v(t) and a set of *n* slide images $S = \{ S_1, S_2, \ldots , S_n \}$, we want to compute a function *f(t)*: $[0,d] \rightarrow S$, where d is the duration of v, such that the video is showing slide $f(\tau)$ at time τ. We set the error tolerance ε to be 0.5 seconds, which is sufficient to keep the slide region of the final presentation synchronized with the video.

A brute force approach to computing *f* involves decoding every frame of the overview video and matching it against all slide images. Since matching is an expensive operation, we can do better by observing that slides changes typically occur infrequently. We can reformulate the problem by decomposing it into two steps, *segmentation* and *matching*, as follows (see figure 7):

- *Segmentation*: given a video sequence v(t), compute a set of time values $T = \{t_0, t_1, \ldots t_k\}$, such that the projected slide image does not change in v during the interval t_i to t_{i+1}.
- *Matching*: compute a function *g*: $T \rightarrow S$, such that, for all $i \in \{0,1, .. , k-1\}$, the segment of v from time t_i to t_{i+1} is displaying slide $g(t_i)$.

This formulation is more efficient. The segmentation phase can use scene cut detection algorithms to determine the set T. The matching phase computes *g* by matching a representative frame from each segment in T to one of the slides in S. The cost savings are substantial because cut detection is much faster than matching slides in each frame.

Notice that the definition of the set T does not require that two adjacent segments display different slides, but only that each segment display the same slide. Thus, a segmentation that has "extra" segments is acceptable. The only impact of extra segments is that the matcher is invoked unnecessarily. If the matcher determines the same slide is shown in two consecutive segments, the segments can be combined. Therefore, a segmentation algorithm that signals a slide change where none exists (i.e., false *positives*) is acceptable, but one that misses a slide change (i.e., false *negatives)* is not. A missed slide transition will most likely cause an incorrectly structured presentation.

Although the algorithm is best viewed as operating in two phases, in practice the phases can be executed simultaneously. The pseudo-code for the unified algorithm is shown in figure 8. The remainder of this section describes the segmenting function **`slide_change()`**, and the matching function **`find_slide()`**, used in that figure.

3.1. Segmentation

Our first approach to slide change detection used methods from the literature [5]. The most common approach to scene cut detection is based on comparing the color histogram of successive frames. When a large change in the histogram is detected, a scene change is signaled. Slide changes, however, do not produce large changes in the color histogram, since slides in a presentation often

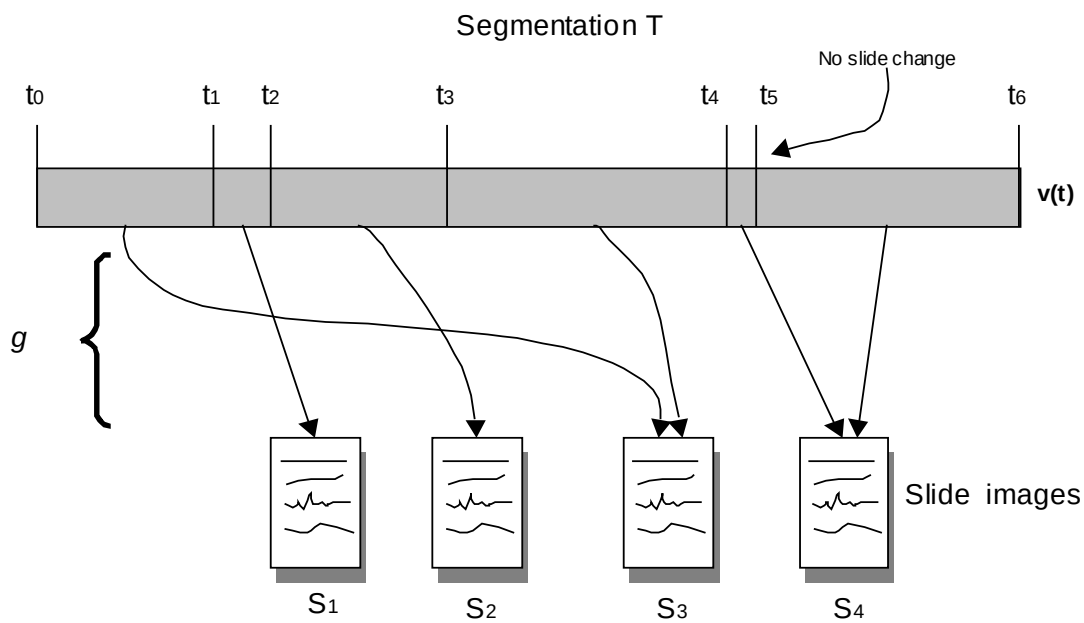

Figure 7: An illustration of segments and the function *g*.

```
Input: sequence v(t) of frames, set of masters M
Output: List T of segment start times
        List g of ordered pairs (t,s), meaning segment starting
        at time t displayed slide s.
Method:

T = {0}
g = {[0, slide_change(v(0),0)]}
lastframe = v(0)
currentslide = 0
for t = 1 .. length(v) {
    frame = v(t)
    if (slide_change(frame, lastframe)) {
        s = find_slide(frame, M, currentslide)
        if (s != currentslide) {
         /* A start of a new segment has been detected */
            append(T, t)
            append(G, [t,s])
            currentslide = s
        } else {
          /* do nothing – we got a false slide change */
        }
     }
     lastframe = frame
} /* end for */
```

Figure 8. Unified segmenting & matching algorithm

use similar colors. Often, the only change between slides is a different text layout. Color histogram techniques cannot detect such changes at the resolution and contrast level of the overview video.

We therefore used a feature-based algorithm to segment the video. The function **slide_change(F_1,F_2)** clips frames $\mathbf{F_1}$ and $\mathbf{F_2}$ to the bounding box of the projected slide. Since the overview camera and the projector are fixed, the corner points of the projected slide image can be determined once, when the camera is installed. Clipping prevents events in the camera's field of view, such as speaker movement, from triggering false slide changes. The clipped frames are low-pass filtered for noise reduction, and then adaptively thresholded [2] to produce binary images, B_1 and B_2. The difference Δ between B_1 and B_2 is computed as follows:

Let b_1 be the number of black pixels in B_1, and let d_1 be the number of black pixels in B_1 whose corresponding pixel in B_2 is not black. Let b_2 and d_2 be similarly defined[1]. Then Δ is defined as:

$$\Delta = (d_1 + d_2) / (b_1 + b_2)$$

[1] This description assumes the slide contains dark text on a light background

The difference Δ is then compared against a threshold. A slide change is signaled whenever Δ exceeds the threshold. Setting the threshold value conservatively, such that no slide changes go undetected, we found that about a tenth of the reported slide changes were false positives. Mostly, the false positives occurred when the lecturer pointed at the projected slide. The motion of the lecturer's arm would trigger a false slide change.

The algorithm's performance can be greatly improved by recalling the synchronization tolerance (ε) to 0.5 seconds. Since I-frames can be quickly located in an MPEG video, and independently decoded, we programmed the MPEG encoder to insert I-frames at half-second intervals and we apply the **slide_change** function to successive I-frames. The modified algorithm only performs 7200 comparisons for each hour of video. Our implementation, based on Dali, can perform this segmentation at 12 times real-time speed on a 333 MHz Pentium-II.

3.2. Matching

We now turn to the function **find_slide(F,M,curr)**. This function computes which slide is displayed in the video frame **F**. Our algorithm has two parts. First, it *unwarps* the foreshortened video frame **F** so that it aligns with the master (see figure 9, lower

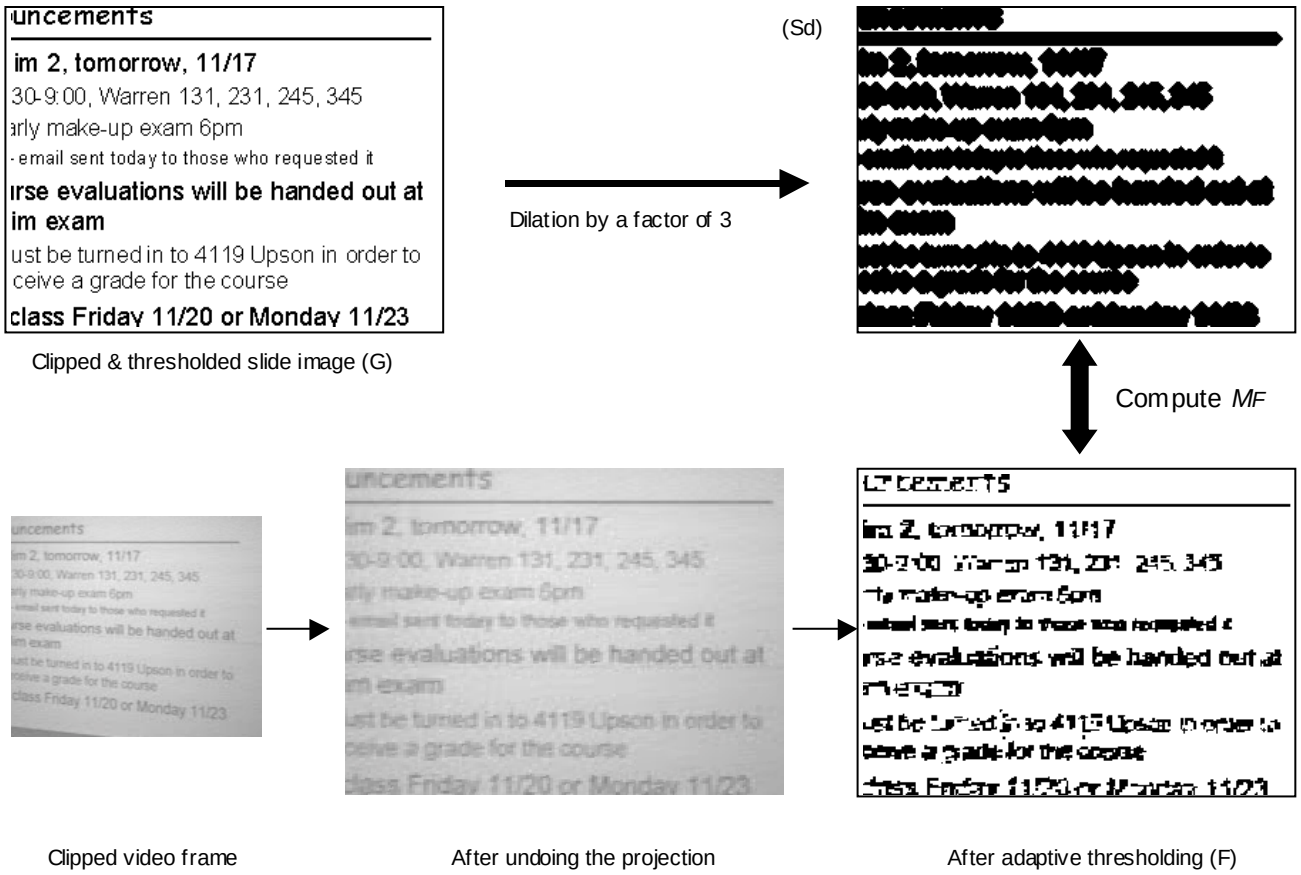

Figure 9: Steps performed during one attempted match

left). Second, it computes the *similarity* between **F** and each slide image in **M**, starting with the last known match (given by **curr**) and then alternately after and before the current slide. During this search, if the similarity exceeds 95%, a match is immediately declared and the search is terminated. After searching all slides, if the maximum similarity was greater than 90%, the slide with the highest similarity is returned. Otherwise, it is assumed that the current frame has too much noise for a good match and the last known matching slide (**curr**) is returned.

We now discuss the two parts of this algorithm, namely unwarping **F** and computing the similarity. Figure 9 illustrates the steps used in both parts.

Unwarping. The video sequence contains a foreshortened version of the slide, as evident from figure 6. Before computing the similarity, we correct this foreshortening by mapping the quadrilateral in *F* that contains the slide to a rectangle of the same size as the slide image. Since the camera and projector are fixed, the corner points of the quadrilateral are constant and can be determined during system installation. Given these coordinates, a projective transform that performs this mapping can easily be computed. The computation, using homogenous coordinates, is straightforward but tedious. The resulting transform is given in the appendix. The upsampled image is computed using bilinear interpolation. Since all slides are the same size, the transform is computed once at initialization and used each time **find_slide** is called.

Similarity. Our similarity metric is based on the Hausdorff Distance [3], [4]. The matching proceeds as follows. The unwarped frame and the slide image are both thresholded to produce binary images. Adaptive thresholding [2] is used on the video frame to account for non-uniform lighting, whereas the slide image is thresholded using a constant value. We call these thresholded binary images *F* and *G* (figure 9, upper left and lower

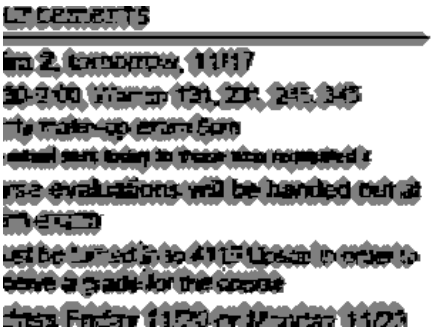

Figure 10: A near perfect forward match

right).

We compute the *forward match* by dilating the slide image G to give another binary image G_d, then counting the percentage of overlapping black pixels. More precisely, dilating a binary image B by a radius r consists of setting all pixels within a radius r of any black pixel in B to black. The quantity r is called the *dilation factor*. Our algorithm uses a dilation factor of 3. To count the overlap, let b be the number of black pixels in F, and b' be the number of black pixels in F whose corresponding pixel in G_d is also black. The forward match $M_F(F,G)$ is simply the ratio b'/b.

Intuitively, the forward match counts the percentage of features in F that are "close" (within the dilation factor) to *some* feature in G. Figure 10 illustrates a case where the forward match is very good. The black colored points correspond to the black pixels in F, and the gray ones are the black pixels of G_d. One can see that this is nearly a 100% forward match because there are very few black points that are not located on top of gray points.

Similarly, the reverse match $M_R(F,G)$ is computed by reversing the roles of F and G. That is, F is dilated to produce F_d, and the ratio of b' (the number of black pixels in G whose corresponding pixel in F_d is also black) to b (the total number of black pixels in G) is computed.

In effect, this matching method compares the shape of the lines of text in the slide. Figures 9 and 10 show that the dilated images reflect the structure of the lines of text in the slide as regions of black pixels (many features) and white pixels (no features). Although the text itself is virtually illegible in the projected video image, the size and shape of the features are easily detected.

Finally, we note that although we used adaptive thresholding to compute the binary images that are compared using the Hausdorff metric, almost any edge detection algorithm can be used.

3.3. Evaluation of the Algorithm

We tested our algorithm on four hours of captured lecture video. The test video contained 106 unique slides and 143 transitions. No transition was missed by the slide change detector, and only four incorrect matches were made by the matcher, translating to an accuracy of 97.2 %. The incorrect matches occurred in slides with fine detail that was lost in the low contrast environment.

The algorithm is sensitive to the ability of adaptive thresholding to adequately distinguish the slide text from the background. The default parameters of the thresholding algorithm performs well on slides that have highly contrasting foreground and background colors, with a light background color. For slides having light text on a dark background, and for those having low contrast, the parameters have to be adjusted to achieve high accuracy. This adjustment currently has to be performed manually.

4. AUTOMATIC EDITING

The final step in processing is to combine the captured video footage into a single stream. Like other functions in the Lecture Browser, this editing is completely automated. After editing the footage by hand, we decided on the following constraints (in decreasing order of importance):

1) Footage from the overview camera must be shown 3 seconds before and 5 seconds after each slide change.
2) No shot may be shorter than 3 seconds.
3) No shot may be longer than 25 seconds.

The first constraint ensures that the viewer notices the slide change. In film and video editing, shots are generally constrained to between 3 and 7 seconds. However, we felt that scene changes that were too rapid would detract from the content of the presentation. Indeed, documentary films often violate these rules. The second and third rules encode the considerations.

We developed a heuristic algorithm for computing an *Edit Decision List (EDL)* that satisfies these constraints. An EDL is an ordered sequence of *shots*. Each shot is a segment taken from one video source, and consecutive shots must come from different sources. A shot is specified by indicating the starting time and duration of the segment, and which video source to use. Concatenating the footage specified by the shots produces us the edited video.

The algorithm is illustrated in figure 11. In the figure, each group of three timelines represents a stage of the algorithm. The timeline for the edited video is first filled with segments from the overview video for 3 seconds before and 5 seconds after every slide change (step 1). The remaining shots are tentatively assigned to the tracking camera (step 2). The resulting EDL is scanned. Shots from the overview camera separated by a tracking camera shot shorter that 3 seconds are merged (step 3). Lastly, another pass is made over the EDL. Shots from the tracking camera longer than 25 seconds are broken up by inserting equally spaced 5-second shots from the overview camera (step 4).

Once the EDL is computed, a Dalí program processes the EDL to compute the resulting video. The video is written directly into RealVideo format. The Dalí program also normalizes the volume level of the audio and brightens the video.

5. RELATED WORK

We described the relationship between the Cornell Lecture Browser and the Classroom 2000, and Experience on Demand systems in the introduction. This section describes other related systems.

The AutoAuditorium [6] project at Bellcore broadcasts live seminars to remote audiences, and simultaneously records them to tape. They use multiple cameras and automatically switch between them based on context. The output of their system is a single taped presentation that must be watched sequentially. No attempt is made to structure the presentation or include non-video data sources.

Minneman et. al. [18] describe a system for recording live meetings and discussions and producing indexed presentations. Like Classroom 2000, their system invasively captures a structured environment. Participants are required to use an electronic whiteboard and the indexed presentation is manually assembled.

STREAMS [19] is another prototype system for recording lectures and creating a presentation to be viewed later. STREAMS uses passive capture, and generates several video and audio streams. The STREAMS authors believe that automatic synthesis of the captured streams would be less effective than having the viewer choose which stream to view. Hence, the system stores all captured streams, and the viewer is presented with low-resolution versions of them. The viewer can then choose a preferred stream and view it in full resolution. We believe that pushing the editing task on the viewer distracts them from the topic of the presentation.

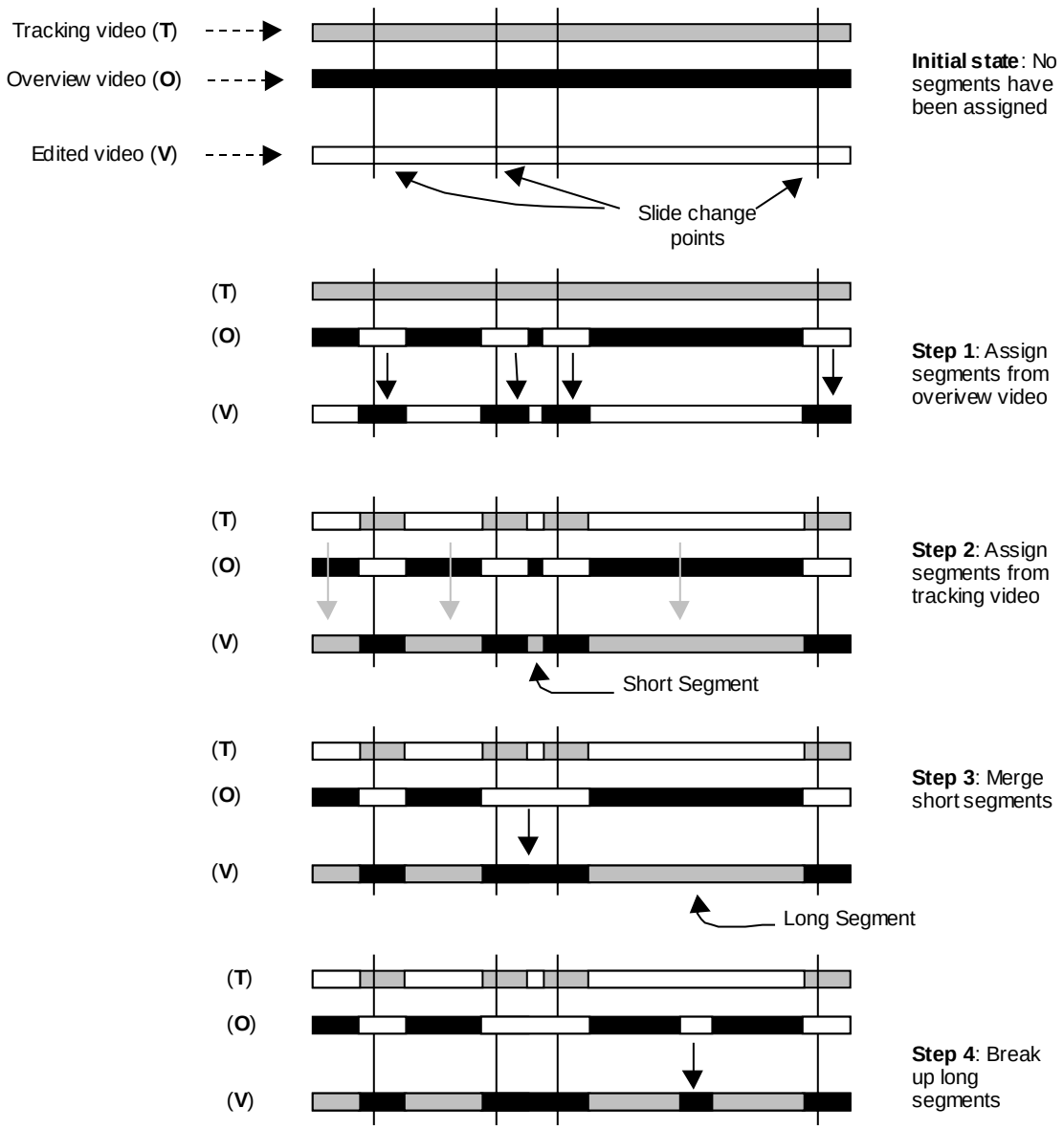

Figure 11: The automatic editing process

Lastly, Eloquent [16] is a commercial service that manually records seminars and lectures and produces presentations similar to the Lecture Browser. As far as we know, they do not use automatic synchronization or editing.

6. CONCLUSIONS

This paper presented an automatic synchronization and editing system that generates structured presentations from lectures and seminars. Such an automated system can produce a structured multimedia document of moderate quality at almost no cost, after the required hardware is purchased and installed.

We identified two important problems arising in such automatic authoring systems, the problem of synchronization and that of automatic editing. We further classified the synchronization problem into three categories, timed-timed synchronization, timed-untimed synchronization and untimed-untimed synchronization, and provided examples of each kind in our application. The advantage of this classification became evident when we showed that the different types of synchronization have different error tolerances, and that knowing the synchronization tolerance can have a strong effect on the performance of algorithms that solve these problems.

We also described a heuristic algorithm for automatic video editing. Other cues can be used to improve the quality of the editing decisions. We are exploring several avenues, including:

- Analyzing the motion vectors of the video to detect scenes with high motion (which would trigger a switch to the overview camera)
- Automatically enhancing the video to correct for poor lighting
- Using the position and orientation of the speaker as cues for editing
- Integrating shots from more cameras
- Inserting the digitized output of scan converters when demonstrations are given.

We also note that the automatic editing algorithms could significantly enhance a traditional video editor, although we are not currently pursuing this idea.

Many challenges remain for this work. In addition to improving the stability of the implementation and the quality of the editing video, we are exploring the use of high-resolution capture devices to capture presentations that use blackboards, whiteboards, 35 mm slides, and transparencies. These media present new challenges in capture, synchronization, and structuring, since electronic masters may not be available. If these efforts are successful, we will be able to capture and structure virtually any classroom presentation with minimal constraints on the speaker.

7. ACKNOWLEDGEMENTS

This work was supported by the Defense Advanced Research Projects Agency (contract N00014-95-1-0799), the National Science Foundation (Research Infrastructure Award CDA-9703470), Intel, Xerox, Microsoft, Kodak and the Alfred P. Sloan foundation.

8. REFERENCES

[1] W. Ooi, B. Smith, et. al., *The Dali Multimedia Software Library*. Proc. SPIE Multimedia Computing and Networking (MMCN ’99), San Jose, Jan 1999.

[2] S. D. Yanowitz, A. M. Bruckstein, *A New Method for Image Segmentation.* Computer Graphics, Vision and Image Processing, 46(1):82–95, 1989

[3] D. P. Huttenlocher, G. A. Klanderman, W. J. Rucklidge, *Comparing images using the Hausdorff Distance*. IEEE Transactions on Pattern Analysis and Machine Intelligence, 15(9):850–863, September 1993.

[4] W. J. Rucklidge, *Efficiently Locating Objects Using the Hausdorff Distance*. International Journal of Computer Vision, 24(3):251–270, September/October 1997.

[5] J.S. Boreczky, L.A. Rowe, *Comparison of Video Shot Boundary Detection Techniques*. Storage and Retrieval for Image and Video Databases IV, Proc. of IS&T/SPIE 1996 Int'l Symposium. on Electronic Imaging: Science and Technology, San Jose, CA, February 1996

[6] M. Bianchi, *AutoAuditorium: a Fully Automatic, Multi-Camera System to Televise Auditorium Presentations.* Joint DARPA/NIST Smart Spaces Workshop, Gaithersburg, MD, July 1998. http://autoauditorium.bellcore.com/overview.html

[7] G. Abowd, et. al., *Teaching and Learning as Multimedia Authoring: The Classroom 2000 Project.* Proc. ACM Multimedia '96, Boston MA, Nov 1996, pp. 187–198.

[8] B. Bailey, et.al., *Nsync - A Toolkit for Building Intractive Multimedia Presentations*. Proc. ACM Multimedia '98, Bristol, England, September 1998, pp 257–266

[9] M. C. Buchanan, P. T. Zellweger, *Automatic temporal layout mechanisms.* Proc. ACM Multimedia '93, Anaheim CA, Aug, 1993, pp 341–350.

[10] D. C. A. Bulterman, L. Hardman, J. Jansen, K. Sjoerd Mullender, L. Rutledge, *GRiNS: A GRaphical INterface for Creating and Playing SMIL Documents*. Computer Networks and ISDN Systems n30 , September 1998.

[11] M. Jourdan, C. Roisin, L. Tardif, *Multiviews Interfaces for Multimedia Authoring Environments*. Multimedia Modeling 98, October 1998.

[12] R. Baecker, A.J. Rosenthal, N. Friedlander, E. Smith and A. Cohen . *A Multimedia System for Authoring Motion Pictures*. Proc. ACM Multimedia '96, Boston MA, Nov 1996, pp 31–42.

[13] T. Meyer-Boudnik, W. Effelsberg *MHEG Explained*. IEEE Multimedia,Vol. 2, No. 1, Spring 1995

[14] M. Christel, T. Kanade, M. Mauldin, R. Reddy, M. Sirbu, S. Stevens, H. Wactlar, *Informedia Digital Video Library.* Communications of the ACM, 38(4):57–58, 1995.

[15] Education on Demand project, http://www.informedia.cs.cmu.edu/eod/

[16] Eloquent Inc. http://www.eloquent.com/

[17] D. Pan, *A Tutorial on MPEG/Audio Compression*. IEEE Multimedia, Vol.2, No. 2, Summer 1995. pp.60–74

[18] S. Minneman et. al., *A Confederation of Tools for Capturing and Accessing Collaborative Activity*. Proc. ACM Multimedia '95, San Francisco, CA, November 1995, pp 523–534

[19] G. Cruz, R. Hill, *Capturing and Playing Multimedia Events with STREAMS.* Proc. ACM Multimedia ’94, San Francisco CA, October 1994, pp 193–200.

APPENDIX

This appendix describes how to compute the two-dimensional homogenous coordinate matrix F that represents a projective transform that maps a quadrilateral **ABCD** to a rectangle of width w x h. The point **A** is mapped to the origin, **B** to (w,0), **C** to (w,h), and **D** to (h,0).

Let

$$B'_x = B_x - A_x$$
$$B'_y = B_y - A_y$$
$$C'_x = C_x - A_x$$
$$C'_y = C_y - A_y$$
$$D'_x = D_x - A_x$$
$$D'_y = D_y - A_y$$

$$\delta = (B'_x - C'_x) \bullet (D'_y - C'_y) - (B'_y - C'_y) \bullet (D'_x - C'_x)$$
$$\alpha = (B'_y \bullet (D'_x - C'_x) - B'_x \bullet (D'_y - C'_y))/\delta$$
$$\beta = (D'_x \bullet (B'_y - C'_y) - D'_y \bullet (B'_x - C'_x))/\delta$$
$$g_1 = (\alpha + 1)\, B'_x/w$$
$$g_2 = (\alpha + 1)\, B'_y/w$$
$$g_3 = (\beta + 1) * D'_x/h$$
$$g_4 = (\beta + 1) * D'_y/h$$
$$g_5 = \alpha/w$$
$$g_6 = \beta/h$$
$$\Delta = g_1 g_4 - g_2 g_3$$

The required projective transform matrix is:

$$F = \begin{vmatrix} g_4/\Delta & -g_3/\Delta & 0 \\ -g_2/\Delta & g_1/\Delta & 0 \\ (g_6 g_2 - g_5 g_4)/\Delta & (g_5 g_3 - g_6 g_1)/\Delta & 1 \end{vmatrix} \bullet \begin{vmatrix} 1 & 0 & -A_x \\ 0 & 1 & -A_y \\ 0 & 0 & 1 \end{vmatrix}$$