Multimedia Systems 8: 201–211 (2000)

Multimedia Systems

Zodiac: A history-based interactive video authoring system

Tzi-cker Chiueh, Tulika Mitra, Anindya Neogi, Chuan-Kai Yang

Computer Science Department, State University of New York at Stony Brook, Stony Brook, NY 11794-4400, USA;
e-mail: {chiueh, mitra, neogi, ckyang}@cs.sunysb.edu

Abstract. Easy-to-use audio/video authoring tools play a crucial role in moving multimedia software from research curiosity to mainstream applications. However, research in multimedia authoring systems has rarely been documented in the literature. This paper describes the design and implementation of an interactive video authoring system called *Zodiac*, which employs an innovative edit history abstraction to support several unique editing features not found in existing commercial and research video editing systems. *Zodiac* provides users a conceptually clean and semantically powerful *branching history* model of edit operations to organize the authoring process, and to navigate among versions of authored documents. In addition, by analyzing the edit history, *Zodiac* is able to reliably detect a composed video stream's shot and scene boundaries, which facilitates interactive video browsing. *Zodiac* also features a *video object annotation* capability that allows users to associate annotations to moving objects in a video sequence. The annotations themselves could be text, image, audio, or video. *Zodiac* is built on top of *MMFS*, a file system specifically designed for interactive multimedia development environments, and implements an internal buffer manager that supports transparent lossless compression/decompression. Shot/scene detection, video object annotation, and buffer management all exploit the edit history information for performance optimization.

Key words: Video editor – Shot/scene detection – Object tracking – Video annotation – Multimedia file system

1 Introduction

A complete digital video authoring environment must support two fundamental functions: capturing and generation of raw video clips, and temporal arrangement of video segments with special inter-segment transition effects. In addition, the ability to synchronize video streams with other media types such as audio is an essential component of nonlinear video editors. Existing video editors are based on a data structure called *edit decision list*, which is a logical representation of the composed video stream in terms of video segments from raw clips. Unfortunately, the information in the edit decision lists are typically thrown away once the composite streams are instantiated, i.e., when the underlying representation is converted from edit decisions to physical files. The thesis of this paper is that edit history contains a wealth of useful information and the video authoring system should exploit it to simplify or even bypass subsequent complicated pixel-level processing for advanced functionalities.

The primary goal of the *Zodiac* project is to demonstrate the hypothesis that recording and analyzing edit operation history could provide useful information about the composed video streams that would otherwise be difficult to come by. In addition, *Zodiac* is meant to verify the usefulness of a home-grown multimedia file system called *MMFS* [16] in supporting multimedia application development. Like existing video editors, *Zodiac* is an interactive video authoring system that focuses mainly on the support for temporal arrangement of video segments extracted from pre-captured raw video clips. The audio support of *Zodiac* is beyond the scope of this paper. *Zodiac* is based on the popular timeline editing paradigm, where the thumbnail images of raw video clips are aligned on a time axis, and frame-accurate cut, copy, and paste operations are supported. The video segments being manipulated are always in an uncompressed representation to avoid quality degradation due to repeated lossy compression/decompression.

Zodiac is built upon a novel *edit history abstraction*, which enables several innovative editing features that do not exist in most commercial or research video editing systems. The edit history abstraction is a first-class object that is persistent across edit sessions and is visible to end users. Unlike edit decision lists, the edit history abstraction is based on a *branching history* model, which keeps track of the development paths associated with versions of video documents. The model greatly simplifies the task of tracking document versions, and thus facilitates interactive exploration of the design space during the authoring process. Moreover, the edit history, coupled with suitable analysis tools, provides a wealth of useful information to other *Zodiac* modules such as shot/scene detection, video object annotation, etc. The major

Correspondence to: Tzi-cker Chiueh

contribution of this paper is the development of the unique edit history abstraction and the demonstration of its power through the implementation of a fully operational video authoring system.

In Sect. 2, we review previous work on digital video editors to set the contribution of this work in perspective. In Sect. 3, we present the edit history abstraction and the system architecture of *Zodiac*. Section 4 describes the storage management support for *Zodiac* at both user and file system levels. Section 5 presents the techniques used in *Zodiac* to detect video shots and scenes from the edit history. In Sect. 6, we describe how a unique video object annotation capability is implemented in *Zodiac* and how it benefits from the edit history abstraction. Section 7 concludes this paper with an outline of ongoing work to improve the *Zodiac* system.

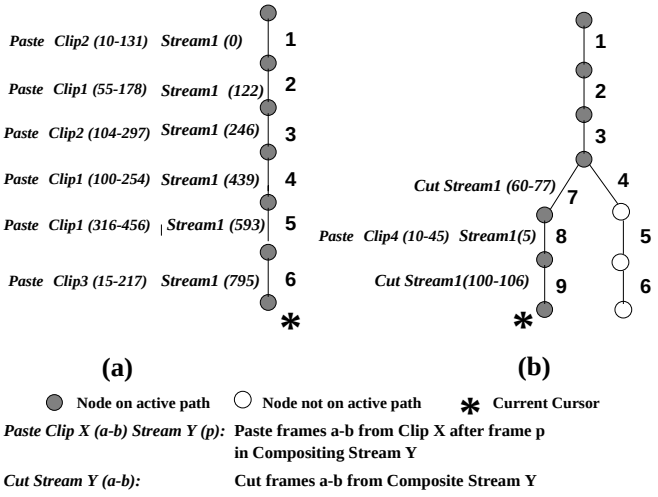

Fig. 1. An example branching history tree for the edit history abstraction. Edges represent the edit operations or steps and are labeled with the corresponding operations. Nodes represent edit states. One of the edit states is called the current cursor, from which the edit history grows

2 Related work

Matthews et al. [13] proposed a melding of the common direct-manipulation interfaces with a programming language designed to manipulate digital audio and video, so that both interactive editing and algorithm-based media-processing operations can be performed. Meng and coworkers [14, 15] described a network-based digital video editor that featured compressed-domain video manipulation, and content-based video browsing/retrieval. Baldeschwieler and Row [3] presented extensions to the Berkeley Continuous Media Toolkit to support network-based media editors, which the authors claimed would work over Internet. The MediaWeaver [18] is a distributed hypermedia framework for teams of authors to compose networked media and video streams in rich models. It is really a middleware that sits between users and various network multimedia services such as video databases and filters. WAVESworld [9] is an object-oriented framework for designing, developing, debugging and delivering three-dimensional, semi-autonomous animated characters. The focus of this work was more on 3D animations than digital video. Commercial video editors include Adobe's Premier [1] and AVID's Media Suite [2], which do not support the advanced shot/scene detection and video object annotation capability in *Zodiac*. A list of existing video editors on various platforms can be found in [17].

Zodiac is different from all previous works on video editors in that it treats edit history as first-class objects and uses edit history to support various important functions such as easy navigation through the version space, accurate shot/scene detection, and video object annotation. The idea of recording design operation history to facilitate design space exploration, and to infer useful semantic information about data objects from design history was first proposed and developed in [4, 5]. In that work, the focus was on supporting VLSI design rather than on video editing. In particular, the types of metadata analysis on operation history are completely different in two cases. Hampapur [7] proposed a model-based approach for video segmentation that exploits the semantics of high-level edit operations such as fade and dissolve. However, this work attempts to uncover these high-level operations, rather than to record them directly, as in *Zodiac*. The ability of annotating moving objects in video streams is available in IBM's HotVideo [8] and V-Active from Ephyx [6]. However, these systems are not integrated with a video editor like *Zodiac*. Also, no literature on the design and implementation of these systems is publicly available.

3 Overview of Zodiac

3.1 Edit history abstraction

3.1.1 Conceptual model

The most innovative aspect of *Zodiac* is its *edit history abstraction*, which maintains a persistent copy of the complete edit operation history associated with the composition of a video document. The edit history abstraction proves extremely powerful in two contexts. First, it provides a conceptually clean interface paradigm for *Zodiac* users to keep track of and navigate through multiple versions of a composed video stream, which correspond to various design alternatives during the authoring process. Second, useful metadata about composed video streams such as shot/scene boundaries can be easily extracted from the edit operation history without resorting to time-consuming and often unreliable pixel-level processing algorithms.

Zodiac's edit history abstraction is a generalization of the edit decision list data structure used in existing video editors because it is based on a *branching history* model of edit operations. Initially, the editing process proceeds linearly, whose corresponding edit history is shown in Fig. 1(a). Each edit operation, such as an insert, append, and delete of a video segment, is called an *edit step*. Between consecutive edit steps we define an *edit point* with a corresponding *edit state*, which is the cumulative effect of applying all the edit steps from the beginning of the edit history up to the preceding step. For example, the edit state immediately after Step 6 corresponds to a video stream that is a concatenation of video segments from Clip 2, Clip 1, Clip 2, Clip 1, Clip

1, and Clip 3. The edit point in the edit history to which records for new edit steps are appended is called the *current cursor*. In Fig. 1(a), the current cursor is the edit point immediately after Step 6.

Assume that at this time the author decides to try out another composition idea starting from the edit state after Step 3. She/he can change the current cursor to the edit point after Step 3. The record for the next new step will start a separate branch from the current cursor. Therefore, the edit history starts to exhibit a branching-tree structure. The records for new steps are appended to the new branch until another change of the current cursor, as shown in Fig. 1(b). Note that in this case, the edit state corresponding to the current cursor is the result of applying Step 1, 2, 3, 7, 8, and 9.

Each edit state in the edit history essentially corresponds to one version of the video stream being composed. Keeping track of the correspondence between document versions and high-level design alternatives during the authoring process is tedious and error-prone. Moving the current cursor among edit points is a powerful and intuitive paradigm to explore the design space, because it greatly simplifies the task of tracking versions and their corresponding design alternatives. Since the document development history is explicitly represented and directly manipulable, users have a clearer picture of the mapping between a particular design version and the context in which it comes to existence. Coupled with appropriate annotations, the edit operation history could even be archived as a design rationale document for future reviews.

The proposed paradigm can also be thought of as a generalization of undo/redo support, since all undo and redo operations required to move the current cursor from one edit point to another are implicitly defined without being explicitly issued by the user. The edit operation history abstraction, although originally developed for VLSI design process management [5], is an even better match to the video document editing process, because intermediate video document versions are inherently represented in a differential form, i.e., the edit decision list.

Because the edit operation history is expressed in terms of high-level edit operations supported by the underlying video editors, the edit history abstraction is applicable to video editors that represent video sequences internally in the compressed or uncompressed form. That is, the underlying video encoding issue is completely hidden from the edit history abstraction and the analysis built on this abstraction. Although this work focuses only on video editing, the same edit history abstraction is equally useful for audio editing. As long as audio and video are properly synchronized, media editors that support both audio and video editing do not add any complications to the edit history abstraction as presented.

3.1.2 Internal implementation

The edit operation history is physically represented as a bidirectional tree to facilitate the traversal up and down the tree. As edit operations are performed, the tree grows accordingly. Reconstructing the state of an edit point involves interpreting the edit operations on the path from the tree's root to the edit point. Traversing the tree from the edit point of interest back to the root uniquely identifies the path, and edit operation interpretation proceeds from the root to the edit point after the path is identified.

All intermediate edit states, except that of the current cursor, are physically represented as a sequence of edit operations, and thus form an *operation log* of the edit history. While this approach reduces the storage requirements for edit states, it also incurs run-time performance overheads for reconstructing the corresponding video document versions. To minimize this overhead, *Zodiac* selectively instantiates those edit states that are commonly used in reconstructing video document versions. Specifically, a *usage counter* is associated with each edit state, and is incremented every time its corresponding state is used in video document reconstruction. When an edit state's usage counter exceeds a threshold, *Zodiac* instantiates the state by explicitly performing the associated edit operations and storing the resulting video stream into a separate physical file. All video document reconstructions that use an instantiated edit state do not need to traverse beyond this state into the history, and thus reduce the reconstruction overhead. This approach corresponds to a *value log* of the edit history.

The state of the current cursor is represented as a *data map*, which is similar to edit decision lists in that its logical composition is explicitly represented with `(raw_clip_ID, start_offset in raw clip, length)` triples.

In summary, *Zodiac* uses the operation log as the main edit state representation, and instantiates only selected edit states with their value logs for performance optimization. That is, a value-log representation is treated as a cache to the operator-log representation of an edit state. The instantiated states are stored on disks as distinct files. Zodiac's instantiation policy uses the edit states' usage counters to determine the instantiation priority among edit states in the edit history. As a result, the states of the edit points that are roots of a larger edit history subtree, and whose states are thus more likely to be involved in the reconstruction of more document versions, tend to have a higher priority of being instantiated. The same usage counters are also used to reclaim storage space from older and less popular instantiated edit states to instantiate newer and more popular edit states.

In *Zodiac*, the physical representation of a video document thus consists of the set of raw video segments, the edit operation history as a tree data structure, instantiated states as separate video files, and the current cursor as a data map expressed in terms of triples. All these representations are stored on persistent storage.

3.2 System architecture

The software system architecture of *Zodiac* is shown in Fig. 2. At the bottom is a buffer manager that manages various data structures used internally in the video editor, including thumbnails, video segments, and object contours used in video annotations. All other modules of *Zodiac*, except the history manager, delegate the storage access service to the buffer manager, which in turn exploits special service features provided by the underlying file system, MMFS. The

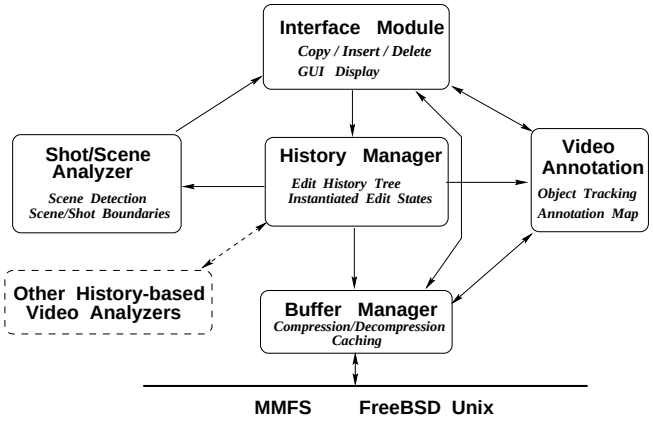

Fig. 2. The system architecture of Zodiac, which consists of a front-end interface module, an edit history manager, a shot/scene boundary analyzer, a video object annotation module, and a buffer manager. The functions performed and data structures maintained are labeled on each block. The *directed edges* among the blocks indicate the data flows

history manager maintains the branching edit history representation and associated data structures such as the current cursor's data map and the instantiated intermediate edit states.

The interface module is responsible for the display of video objects and the interpretation of user inputs. It transparently captures all edit operations and relays them to the history manager. The shot/scene analyzer module analyzes the edit history and determines the shot and scene boundaries *without accessing the pixel values of the associated video segments.* The video object annotation module allows users to annotate a video object in any frame and automatically associates annotations with the object in every frame it appears. This video object annotation capability greatly simplifies the task of annotating digital video and is the enabling technology for the concept of digital video-based knowledge acquisition and transfer [11]. The video object annotation module exploits the edit history information to determine when to stop the object-tracking algorithm.

The current *Zodiac* prototype, which has been operational for a year, is implemented on top of FreeBSD Unix 2.1 and MMFS, running on 200-MHz PentiumPro machines. We are in the process of implementing the audio support of *Zodiac* for public release.

4 Storage management

Because digital video takes a substantial amount of storage space, efficient storage management is essential for *Zodiac* to provide interactive editing performance. *Zodiac* relies on MMFS to improve the performance of individual in-place file insert and delete operations, which are used extensively in video editing. In addition, *Zodiac* includes an internal buffer manager that manages the use and representation of intermediate video objects in the user-level buffer area.

4.1 MMFS

General-purpose file systems such as the Unix file system are oriented towards textual data storage and retrieval, where files are typically small in size. Since rewrite of entire small files takes a relatively short amount of time, support for editing operations such as insert, delete, and update were not considered necessary. Because multimedia editing operations require frequent inserts and deletes and the data segments being manipulated are large, special support from the file system is essential.

To illustrate the need, consider the case in which a new frame is to be inserted between the i-th and $i+1$-th frames of a video clip that has a total of j frames ($j >$ i). On generic Unix file systems, such an insertion entails reading the frames from the $i+1$-th frame to the j-th frame, writing the new frame as the $i+1$-th frame, and then appending the frames just read in to form the rest of the file. In the worst case, this leads to the whole file being read and re-written. For large video files, the response time for this simple editing operation is unacceptably long.

MMFS [16] is derived from FreeBSD 2.1, and is designed to support interactive multimedia application development. One of the main features in MMFS is an efficient support for inserts and deletes at a small cost of storage space overhead, which could be further reduced by periodic compaction. Two new system calls `mminsert` and `mmdelete` are provided for applications to access this service.

The key idea behind MMFS's fast inserts and deletes is to eliminate the constraint that all the blocks of a file, except the last one, must be completely filled with valid data. With this constraint, it is relatively straightforward to compute physical block ID from logical file offsets using the Inode data structure. However, this constraint can no longer be upheld if inserts and deletes are to be performed with only metadata updates, because the segments to be deleted or inserted do not necessarily occupy an integral number of disk blocks. Therefore, MMFS allows file blocks to be partially full. The mnode data structure, an Inode counterpart in MMFS, contains an additional field to record the number of valid bytes in the file block. As a result, a file in MMFS is physically represented as an ordered list of logical blocks, each of which could be partially full.

Deleting data involves only a modification of the mappings in the mnode, followed by a release of the corresponding disk blocks to the free list. An insertion would lead to a write of the data in a freshly allocated block at the end of the file, followed by a modification of the mnode block mappings to indicate the actual position of the newly inserted block. Figures 3 and 4 show the evolution of the file system metadata in the process of inserting and deleting file blocks in MMFS.

4.2 History-conscious buffering

Zodiac's internal buffer manager exploits efficient insert and delete operation of MMFS to support direct updates to instantiated edit states. In addition, it exploits the edit operation history information to make better buffer replacement decisions. Specifically, the buffer manager determines the importance of video segments currently resident in main memory according to the following rules.

– Those video segments that are part of the current cursor's data map have a higher priority over others that are not.

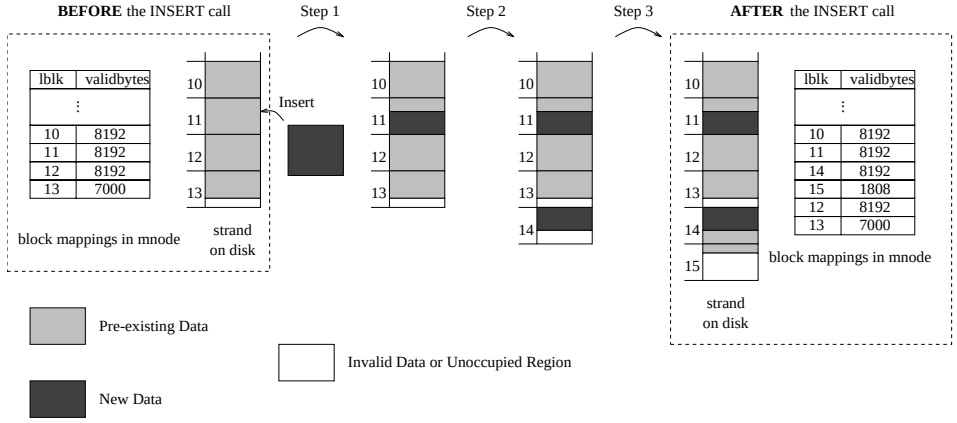

Fig. 3. File data and metadata before, during and after an `mminsert` system call: 10,000 bytes of new data are to be inserted at an offset corresponding to the 3000st byte of Block 11, where the block size is 8 KB. The mnode mappings and the file at the left end of the figure describe the state of the file before the call. *Step 1* inserts 5192 bytes of new data in the freshly created hole. *Step 2* appends the rest (4808 bytes) of the new data at the end of the file in a fresh block (Block 14). *Step 3* appends the old data that was removed from Block 11 to the file, thus filling up Block 14 with 3384 bytes, and partially filling up Block 15 with 1808 bytes. The mnode mappings and the file at the right end of the figure reflect the state of the file after the call

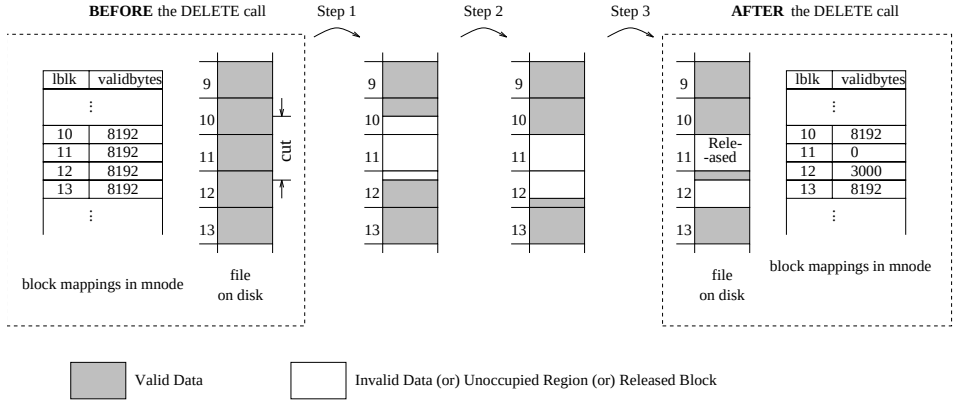

Fig. 4. File data and metadata before, during and after a `mmdelete` system call. The segment to be deleted is from the 4000st byte of Block 10 to the 1000st byte of Block 12, where the block size is 8 KB. The mnode mappings and the file at the left end of the figure define the state of the file before the call. *Step 1* identifies the "invalid" areas of the file. *Step 2* fills the hole in Block 10 with data from Block 12. *Step 3* moves data in the Block 12 to the beginning of the block. The mnode mappings and the file at the right end of the figure reflect the state of the file after the call

– For video segments in the current cursor's data map, those that are closer to the current cursor have a higher priority over those that are farther away.
– For video segments not in the current cursor's data map, they are ordered according to the least recently used (LRU) policy.

Video segments of more importance are more likely to stay memory-resident. The rationale behind these rules is that video editor authors tend to access the parts of the video stream being composed that they were working on recently. By consulting the edit history, the buffer manager has a better grip on the user's current working set, which may be very different from the estimate from the LRU policy, for example, when users move the current cursor. *Zodiac* also exploits the information in the data map to quickly bring in the current working set when users move the current cursor.

Zodiac's buffer manager also supports transparent lossless compression and decompression to make more efficient use of the limited buffer space. Lossless rather than lossy compression algorithms are chosen to prevent quality degradation due to repeated compression and decompression. As far as other *Zodiac* modules are concerned, they only see uncompressed data. That is, the fact that video segments are stored in compressed form in the user buffer during the editing process is completely hidden from other *Zodiac* modules. The current *Zodiac* implementation is based on the GNU gzip library [22]. Compressed frame size is 38%–65% of the original full-size frame for typical video clips of 360×240 resolution. For thumbnail frames of resolution 64×64, compressed frame size is 67% to 87% of the original.

5 Shot/scene boundary detection

Much research effort has been invested in the detection of shot/scene boundaries of digital video sequences. Zhang et al. [21] has a nice summary on the recent results using image-processing techniques. *Zodiac* takes a completely different approach in that it attempts to uncover high-level structures of digital video sequences by analyzing their associated edit history. The key observation underlying this approach is that, since video authors arrange the temporal layout of video segments to express semantic intention, it should be possible to recover the high-level structure of com-

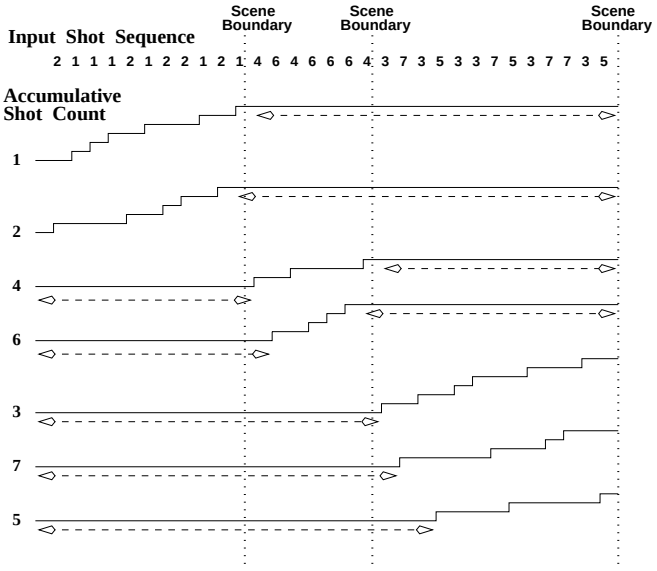

Fig. 5. An illustrating example for Zodiac's scene detection algorithm. The input is a shot sequence shown above, where each number indicates which clip the corresponding shot comes from. Below are the accumulative shot count curves for the participating clips. The *double-arrowed dashed lines* indicate the flat regions of the corresponding clip's accumulative shot count curve

posite video streams by analyzing the edit operations performed in the authoring process. This approach has two important advantages. First, the computational effort required is much lower, because *it does not require pixel-value manipulation.* Second, the accuracy of the analysis results is expected to be better, because the edit history data it uses inherently provides richer information. Of course, the proposed approach only applies to video documents whose edit operation history is available. Pixel-level video-processing techniques are still needed for non-*Zodiac* video documents.

Essentially, the recorded edit history is treated as a computer program. The hypothesis is that authors follow conventions or rules for video editing, if not grammars. These rules can be used to parse the edit operation history to identify higher level semantic constructs rather than individual video frames, just as syntactic analysis of computer codes could uncover high-level program behaviors. The grammars of film [20] is much less rigorous than formal grammars used to define computer languages, and actually allow for exceptions, which typically are deemed as "novel creations." As a result, recognition based on film grammars would not be 100% accurate.

Zodiac currently applies this approach to only shot and scene boundary detection. Other more sophisticated edit history analysis modules using techniques such as those proposed in [19] can be added in a modular fashion, as shown in Fig. 2. Given a composite video stream's data map, which is a sequence of `(raw_clip ID, start-offset in raw clip, length)` triples, shot boundary detection is straightforward. By definition, whenever there is a discontinuity between adjacent video frames, there is a shot boundary between the two. Therefore, by comparing each neighboring pair of triples in the data map *Zodiac* can determine with 100% accuracy whether they come from the same raw clip, and thus the shot boundaries. Some video segments of a composite stream are not copied directly from raw video clips, but result from manipulation of multiple video segments, e.g., fading or dissolving. In this case, a special flag in the data map representation is used to indicate specific transition effects. These flags automatically signify the presence of shot boundaries.

Because there is no guarantee that clip IDs are globally unique, confusion may arise because distinct composite streams manipulated in the same edit session happen to use the same clip ID to refer to different raw clips. In particular, there may be a portability issue about *Zodiac* documents when a composite video stream is edited at one site and transferred to another site for further processing. To ensure the uniqueness of raw clip ID, *Zodiac* relies on the message digest algorithm MD5 to generate the clip ID, which uses the concatenation of the editing machine name and the clip's file name as the input key.

The current scene detection algorithm used in *Zodiac* can only identify scenes that comprise video segments extracted from multiple video clips, each of which corresponds to a particular viewpoint or camera angle towards a given physical world. This type of scenes are characterized as a sequence of shots that are extracted from only a subset of video clips.

From a video shot sequence derived from the edit operation history representation of a composite stream, the scene detection algorithm scans through the shot sequence and counts, for each clip, the cumulative number of shots extracted from that clip versus the number of shots in the sequence examined so far. During such a scene-bearing shot subsequence, the shot counts of those other than active clips should remain unchanged. That is, the accumulative shot count curve of a clip should be flat during a scene to which it does not contribute. Based on this observation, the algorithm to identify scene boundaries is as follows.

1. Set the current shot to be the first shot in the input sequence.
2. Identify all video clips that have a contiguous flat region in their accumulative shot count curves, starting from the current shot. The length of a contiguous flat region, in terms of the number of shots, must exceed a minimum threshold.
3. Intersect the flat regions identified in Step 2, and the intersecting region corresponds to a scene.
4. Set the current shot to be the end of the scene just identified plus 1. If this is the end of the shot sequence, exit; otherwise go to Step 2.

Figure 5 gives an example of how scenes are detected using the above algorithm. In the beginning, the clips that have a flat region are Clips 3, 4, 5, 6, and 7. The intersection of these flat regions gives the first scene boundary, between the 11th and 12th shots. Starting from the 12th shot, the clips that have a flat region are Clips 1, 2, 3, 5,and 7. The intersection of these flat regions gives the second scene boundary, between the 18th and 19th shots. Finally, the intersection of the flat regions from Clips 1, 2, 4, and 6 gives the third and final scene boundary.

We have tested the above scene detection algorithm on a set of composite video documents generated using *Zodiac*

and found that the algorithm identifies *all* scene boundaries correctly. The excellent performance of this scene detection algorithm comes from that fact that it uses not only exact shot boundary information but also the information about which clip each shot is extracted from. Once scenes are identified, *Zodiac* stores them as metadata, together with the original video documents. *Zodiac* uses the scene information associated with video documents to support scene-based rather than frame-based browsing, i.e., jumping from scenes to scenes.

6 Video object annotation

6.1 Functional overview

In addition to standard video edit operations such as cut and paste, and special-effect transformations, *Zodiac* also supports a novel video object annotation capability to create hyper-video documents. *Zodiac*'s video object annotation facility allows users to annotate moving objects, such as vehicles or humans, in a video stream. Users are required to pick a frame in which the annotated object appears, trace the object's contour on the screen, and provide the intended annotation. *Zodiac* automatically tracks the movement of the annotated object in BOTH directions, starting from the selected frame, and associates the annotation with the object's estimated contours in other frames. The result is that when users click on the annotated object in any frame in which it appears, the associated annotation would pop up accordingly. Video object annotation is essentially a generalization of the image map construct in HTML. However, it greatly simplifies the process for integrating annotations with video streams, because users only need to manually describe the contour of the video object to be annotated exactly once.

Video object annotation is expected to play an important role in applications such as electronic commerce, knowledge transfer [11], and so on, where digital video is a much more expressive medium than pure text and images. For example, it is conceivable that an entire Christmas sale catalog is put on a DVD CD, and short video clips are used to to demonstrate the functionality of certain product items, with annotations showing their prices, availability, or other related products.

In *Zodiac*, the annotations for video objects can be in the form of text, image, audio, or even video, as shown in Fig. 6. In the case of video, users can choose to stop or continue to play the annotated video while playing the annotating video. In addition, it is possible to control the relative playback rates of the annotating and annotated video sequences to bring out certain contrasting effects.

6.2 Object-tracking algorithm

The key technology underlying the video object annotation capability is object tracking, which, in this context, means estimating the annotated object's contour in other frames from the user-provided contour in the starting frame. However, unlike computer vision applications, video annotation does not require 100% object-tracking accuracy, because the results of object tracking are contours to be associated with annotations. As long as the estimated contour of the annotated object is sufficiently close to the actual contour, users are likely to click a point inside the estimated contour and thus be able to retrieve the associated annotation. *Zodiac* uses a variant of the algorithm described in [12]. The basic idea is to use motion estimation to arrive at a good guess of the initial contour from one frame to the next, and rely on active contour (*snake*) [10], to relax to the exact object contour.

Assume the object contour is represented as $V = [v_1, v_2, ..., v_n]$, where $v_i = (x_i, y_i)$ is the coordinate of the i-th vertex. In addition, let $v_i(t)$ be the i-th vertex at the t-th frame. The object-tracking algorithm estimates the object contour at frame t from the contour at frame $t-1$ by minimizing the following objective function:

$$E_{snake} = \sum_i \alpha * E_{cont}(v_i(t)) + \beta * E_{curv}(v_i(t)) + \gamma * E_{img}(v_i(t)) + \eta * E_{match}(v_i(t)) . \quad (1)$$

The first three terms are used by the original snake algorithm while the fourth term is introduced to account for motion. α, β, γ and η are weighting constants, to be determined empirically to adjust these terms' relative importance.

Minimization of $E_{cont}(v_i(t))$, defined as $|\overline{d}(t) - |v_i(t) - v_{i-1}(t)||$, where $\overline{d}(t)$ is the average distance between each pair of neighboring vertices, encourages the vertices on the resulting contour to be equally spaced and contracting. Minimization of $E_{curv}(v_i(t))$, defined as $|(v_{i-1}(t) - v_i(t)) - (v_i(t) - v_{i+1}(t))|$, forces the resulting contour to be smoother by reducing the second-order derivative or curvature value at each vertex.

The first two terms in Eq. 1 are related to the structure of the resulting contour, but have nothing to do with the inherent image property of the contoured object. The third term is defined as

$$E_{img}(v_i(t)) = -\frac{g_i(t) - g_{min}(t)}{g_{max}(t) - g_{min}(t)} , \quad (2)$$

where $g_i(t)$ is the magnitude of the gradient vector computed at the i-the vertex in the t-th frame, and $g_{min}(t)$ and $g_{max}(t)$ are the maximum and minimum gradient values computed so far. We assume that the object to be annotated has a very different image profile compared to its surroundings. Therefore, minimization of $E_{img}(v_i(t))$ causes the contour points to be placed on where largest gradient values occur.

The final term, $E_{match}(v_i(t))$, evaluates the likelihood of a candidate point of $v_i(t)$ from the standpoint of its compatibility with global and local motion estimation. The global motion of the tracked object from the t-th frame to the t+1-th frame, $m(t)$, is determined by

$$m(t) = \frac{1}{n}\left(\sum_i v_i(t) - \sum_i v_i(t-1)\right); \quad m(1) = 0 , \quad (3)$$

which is the average object motion computed from all the contour vertices. Given $m(t)$, the purpose of local motion estimation is to search a 9×9 window around $v_i(t)$ to identify the best local motion vector $\Delta v_i^*(t) = (\Delta x_i^*(t), \Delta y_i^*(t))$ for each vertex and arrive at the best estimate of $v_i(t+1)$, $\hat{v}_i(t+1) = v_i(t) + m(t) + \Delta v_i^*(t)$.

Each point in the 9×9 window of given contour point represents a candidate for local motion for that point, and is denoted as $\Delta v_i^{jk}(t)$, where j and k both range from -4 to 4. Each such candidate is assigned a probability for being the best local motion vector for the i-th vertex. These probabilities are updated iteratively, and eventually the one with the highest probability is the best local motion vector. The initial probability for each of these candidates, $\hat{P}^0(\Delta v_i^{jk}(t))$, is calculated based on the difference between their corresponding blocks to the original block.

$$P^0(\Delta v_i^{jk}(t)) = \frac{\frac{1}{1+c_0 * r_i^{jk}(t)}}{\sum_{m=-4}^{m=4} \sum_{n=-4}^{n=4} \frac{1}{1+c_0 * r_i^{mn}(t)}}, \quad (4)$$

where $r_i^{jk}(t)$ is the block difference between the 9×9 block centering at $v_i(t)$ and that at $v_i(t)+\Delta v_i^{jk}(t)$ in the next frame, and c_0 is a constant.

At each iteration, the probabilities of these candidate points are updated according to the probabilities of the current vertex in the previous iteration, and of neighboring vertices in the contour. That is,

$$\hat{P}^{l+1}\left(\Delta v_i^{jk}(t)\right) = P^l\left(\Delta v_i^{jk}(t)\right) * (c_1 + c_2 * \sum_{s=i-1}^{i+1} \sum_{m=j-1}^{m=j+1} \sum_{n=k-1}^{n=k+1} P^l(\Delta v_s^{mn}(t)). \quad (5)$$

The second term inside the parentheses ensures that the probabilities for neighboring $\Delta v_i^{jk}(t)$'s to be similar, and neighboring contour points have similar local motion vectors. c_1 and c_2 are constants that control the convergence rate. Finally, a normalization is performed to ensure the estimated probability values are between 0 and 1.

$$P^{l+1}\left(\Delta v_i^{jk}(t)\right) = \frac{\hat{P}^{l+1}(\Delta v_i^{jk}(t))}{\sum_{m=-4}^{m=4} \sum_{n=-4}^{n=4} \hat{P}^{l+1}(\Delta v_i^{mn}(t))}. \quad (6)$$

In the current implementation, we use $c_0 = 10, c_1 = 0.3, c_2 = 3$ and run for ten iterations to obtain all $P(\Delta v_i^{jk}(t))$, and choose the candidate point with the highest probability as the local motion vector for a given contour point.

Given the object contour from the previous frame, $v_j(t-1)$'s, $j = 1, \ldots, n$, the previous frame and the current frame, the object-tracking algorithm first applies local motion estimation to each contour vertex. Then it applies both local and global motion estimation results to arrive at the best estimate of each contour vertex $v_i(t)$ in the current frame, $\hat{v}_i(t)$. Finally, the algorithm searches a 3×3 window around $\hat{v}_i(t)$ to derive the best estimate for $v_i(t)$ that minimizes Eq. 1, where $E_{match}(v_i(t))$ is defined to be $-P(v_i(t)-v_i(t-1))$. By choosing the local motion vector with the highest probability for each contour point, $E_{match}(v_i(t))$ is automatically minimized.

All terms in Eq. 1 are first normalized to [0, 1], and the weighting coefficients are all set to 1. Multiple iterations through each of the contour vertices are required until the best estimates for all vertices stabilize, i.e., stop changing. In each iteration, the best estimates from the previous iteration are used as the starting points. Finally, the stabilized best estimates for the vertices form the contour of the tracked object in the current frame.

6.3 Implementation

Based on the object-tracking algorithm, *Zodiac* derives the contour of the annotated object in all the other frames in the video sequence, both in the forward and backward directions. *Zodiac* stops tracking objects when the tracking error exceeds a certain threshold. The tracking error of an estimated object contour is defined as the mean square error (MSE) between the pixel-value histograms of the estimated contour and the contour specified in the starting frame, i.e., the frame with which users describe the tracked object's contour. The tracking error of an estimated contour is considered unacceptable if it is more than B times the MSE between the histograms associated with the contours of the start frame and of its next frame. B is an empirical constant and is set to 3 currently. The rationale is that the tracking error between the starting frame and its next frame should be a reliable indicator of the true changing rate of the object contour.

A unique aspect of *Zodiac*'s object-tracking algorithm implementation is that it further exploits the shot information derived from the edit history to perform object-tracking across discontinuous shots. That is, *Zodiac* only runs the object tracking algorithm against video subsequences that are contiguous segments from the same raw clip. As a result, the object-tracking algorithm is not only more efficient because it does not waste time on irrelevant frames, but also more powerful and accurate in that it can track objects between frames that are separated by one or multiple unrelated video shots! To the best of our knowledge, no existing video-object-tracking system can track objects across video shots.

Once the contour of the annotated object in other frames is estimated, *Zodiac* stores the object contour information with the frame it appears. At run time, when users click on the annotated object, *Zodiac* performs a *containment* check to determine whether the point coordinate of a user click falls within the object's contour, and if so, invokes the corresponding annotation (see Fig. 7). The containment check algorithm is based on the assumption that the vertices of the contour are ordered in a *clockwise* fashion. Given a click point p, for each pair of consecutive contour vertices, v_i, v_{i+1}, *Zodiac* computes the cross-product of the two vectors, $v_{i+1} - v_i$ and $p - v_i$. If the object contour is convex and p is inside it, the cross-product for each pair of consecutive vertices should be positive or zero. In practice, the object contour may not be convex everywhere or the tracking errors may introduce concavity. The current implementation accepts p to be inside the estimated contour, if more than 85% of the cross-product tests for all consecutive vertex pairs are positive or zero. The current implementation of the containment check algorithm is sufficiently fast for invoking video annotations interactively.

The total computation time required for object tracking is proportional to the number of contour vertices. On a 32-MB Pentium-200 MHz machine, the measurement is 0.025 s per contour point. In other words, it takes 50 s to track a 20-vertex contour over a 100-frame period. For all the video

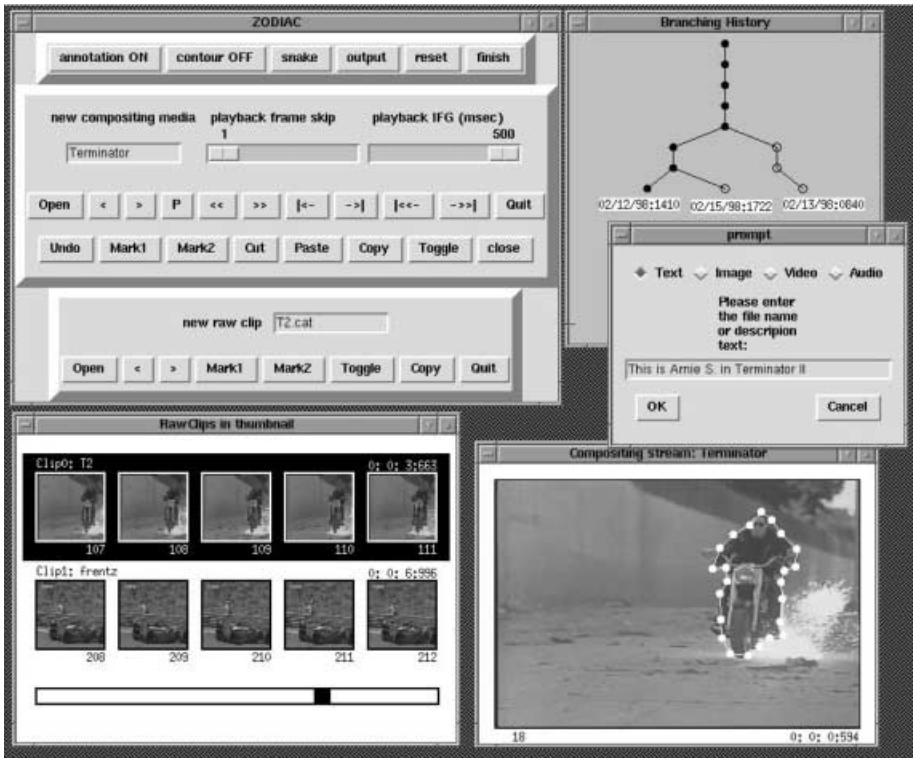

Fig. 6. Zodiac interface for users to specify the initial contour of the annotated object, in this case the person and the motorbike, and the annotation text

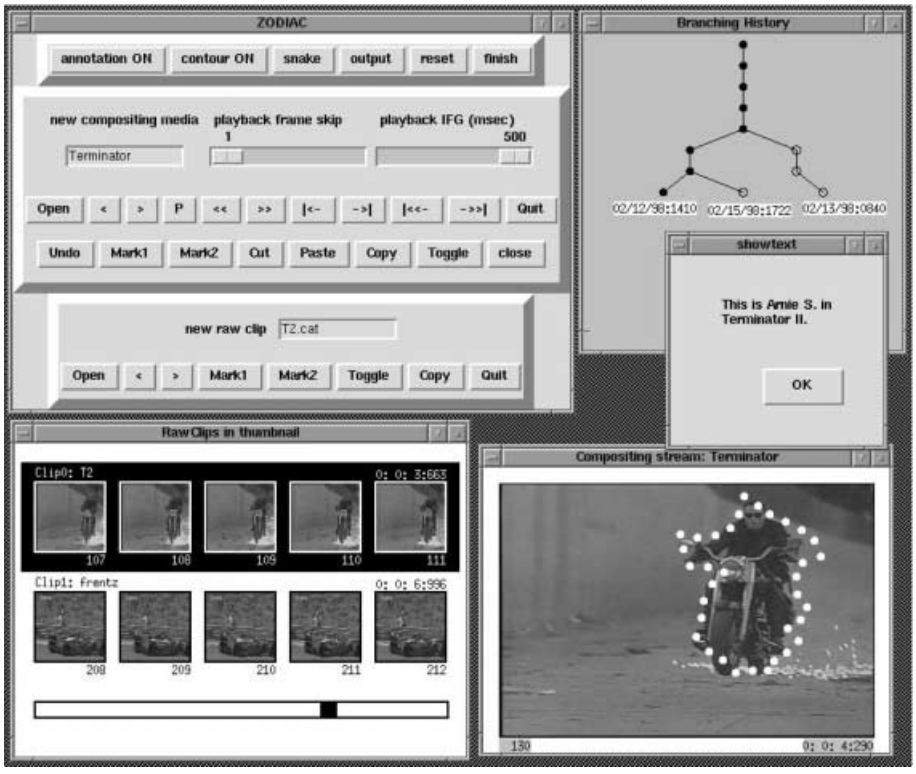

Fig. 7. Invocation of annotation in subsequent frames that depends on accurate object tracking

streams tested with fast-moving objects with pure translational motion, the tracking accuracy is almost 100%. Objects with rotation or camera zoom/pan during motion can be tracked with approximately 80% accuracy, if the user clicks on the object center. In the case of the clip taken from "Terminator II," as shown in Fig. 6, the moving object with rotational and translational motion was tracked with 95% accuracy.

7 Conclusion

This paper describes the design and implementation of *Zodiac*, an interactive video authoring system based on a

unique edit operation history abstraction, which serves as the basis for interactive document version navigation, accurate shot/scene detection and simplified video object annotation authoring. The major contributions of this work are the development of the edit history abstraction as a new user interface metaphor to support interactive exploration of the design space in the context of video editing, and the demonstration of the powerful paradigm of analyzing edit history to infer high-level metadata about digital video streams through the *Zodiac* prototype implementation.

There are several promising research directions to further improve *Zodiac*, and we are currently pursuing some of them. First, the disk representation of the edit operation history of *Zodiac* needs to be improved, so that it could be better integrated with future digital video compression standards such as MPEG-7. Second, *Zodiac*'s buffer management and history management modules need to be re-implemented as separate threads, so that I/O-intensive or compute-intensive tasks could be delegated to the background to minimize the user-perceived delay. Third, *Zodiac* is local disk based, and needs to be extended to a networking environment. Basically the buffer manager should be re-implemented as an independent process running on a storage server that is accessible over the network. Fourth, the object annotation subsystem needs a high-level language to describe more precisely the spatial and temporal arrangements of the annotating and annotated video streams. Such a language could significantly enrich the ways digital video is rendered. Fifth, more sophisticated history-based video analyzers based on well-known film-editing theory should be developed to extract higher level semantic structures of digital video streams. Along the same line, high-level edit operation patterns should be provided as pre-defined templates, so that ordinary video-editing users could effortlessly apply sophisticated editing techniques for complicated effects. Finally, there is very little usage experience and performance data with *Zodiac* because of the lack of audio support. Once we incorporate the audio support into *Zodiac*, we plan to carry out an extensive user trial to collect actual usage patterns and such performance data as scene detection accuracy, and the effectiveness of history-conscious buffer replacement policy.

Acknowledgements. This research is supported by an NSF Career Award MIP-9502067, NSF MIP-9710622, NSF IRI-9711635, NSF EIA-9818342, NSF ANI-9814934, a contract 95F138600000 from Community Management Staff's Massive Digital Data System Program, USENIX student research grants, as well as fundings from Sandia National Laboratory, Reuters Information Technology Inc., and Computer Associates/Cheyenne Inc.

References

1. Adobe Premier. http://www.adobe.com/Apps/Premier.html
2. Avid Technology Inc. web site. http://www.avid.com
3. Baldeschwieler JE, Row LA (1996) Editing extensions to the Berkeley continuous media toolkit. Master's Report, Computer Science Division, EECS, University of California at Berkeley. http://www.bmrc.berkeley.edu/ eric14/masters
4. Chiueh T, Katz R (1993) A history approach of automatic relationships establishment for VLSI design database. IEEE Trans Know Data Eng 5(6):987–990
5. Chiueh T, Katz R (1994) Papyrus: A history-based VLSI design process management system. In: Proceedings of the 10th IEEE International Conference on Data Engineering, pp 385–392, Houston, Texas
6. Ephyx's V-Active Technology. http://www.ephyx.com
7. Hampapur A, Jain R, Weymouth TE (1995) Production model based digital video segmentation. Multimedia Tools Appl 1(1):9–46
8. IBM's HotVideo. http://www4.alphaworks.ibm.com
9. Johnson MB (1995) WAVESworld: A testbed for constructing 3D semi-autonomous animated characters. PhD thesis, Program in Media Arts and Sciences, MIT, Cambridge, Mass.
10. Kass M, Witkin A, Terzopoulos D (1987) Snakes: Active Contour Models. Int J Comput Vision 1(4):321–331
11. Lieberman H (1994) A user interface for knowledge acquisition from video. In: Proceedings of the Conference of the American Association for Artificial Intelligence, pp 527–534, Seattle, Washington
12. Lin Y, Chang Y (1997) Tracking deformable objects with the active contour model. In: Proceedings of the IEEE International Conference on Multimedia Computing and Systems, pp 608–609, Ottawa, Ontario, Canada
13. Matthews J, Gloor P, Makedon F (1993) VideoScheme: A programmable video-editing system for automation and media recognition. In: Proceedings of the First ACM International Conference on Multimedia, pp 419–426, Anaheim, CA
14. Meng HJ, Chang S (1996) CVEPS–A compressed video editing and parsing system. In: Proceedings of the ACM International Multimedia Conference, pp 43–53, Boston, MA
15. Meng HJ, Zhong D, Chang S (1997) A distributed system for editing and browsing compressed video over the network. In: Proceedings of the First Signal Processing Society Workshop on Multimedia Signal Processing, pp 489-494, Princeton, NJ
16. Niranjan T (1996) File system support for multimedia applications. PhD thesis, Computer Science Department, SUNY at Stony Brook, N.Y.
17. Siglar J. Multimedia authoring systems FAQ. http://www.tiac.net/users/jasiglar/MMASFAQ.HTML
18. Wei SX (1998) MediaWeaver – A distributed media-authoring system for networked scholarly workspaces. Multimedia Tools Appl 6(2): 97–111
19. Yeung M, Yeo B, Liu B (1996) Extracting story units from long programs for video browsing and navigation. In: Proceedings of the Third IEEE International Conference on Multimedia Computing and Systems, pp 296–305, Hiroshima, Japan
20. Yoshitaka A, Ishii T, Hirakawa M, Ichikawa T (1997) Content-based retrieval of video data by the grammar of film. In: Proceedings of the IEEE Symposium on Visual Languages, pp 310–317, Capri, Italy
21. Zhang H, Low CY, Smoliar SW (1995) Video parsing and browsing using compressed data. Multimedia Tools App 1(1):89–111
22. Zlib Library. http://www.cdrom.com/pub/infozip/zlib

TZI-CKER CHIUEH is currently an Associate Professor in Computer Science Department of SUNY at Stony Brook. He received his B.S. in EE from National Taiwan University, M.S. in CS from Stanford University, and Ph.D. in CS from University of California at Berkeley in 1984, 1988, and 1992, respectively. He received an NSF CAREER award in 1995. Dr. Chiueh's research interest is on 3D graphics architecture, scalable and secure network routers/gateways, and high-performance memory/storage systems.

TULIKA MITRA is currently a doctorate student in the Department of Computer Science at State University of New York at Stony Brook. She received her B.E. from Jadavpur University in 1995 and M.E. from Indian Institute of Science in 1997, both in Computer Science. Her research interests include computer architecture, distributed/parallel systems, and multimedia systems.

ANINDYA NEOGI is in his third year in the doctoral program of the Dept. of Computer Science, SUNY at Stony Brook. He received his B.E. in Computer Science and Engineering from Jadavpur University, India in 1995 and has an MS from Stony Brook. Before coming to Stony Brook he was in the R&D of Cadence Design Systems, India. His research interests are in QoS-capable scalable router/switch design and fault-tolerant real-time storage systems.

CHUAN-KAI YANG received the B.S. degree in Mathematics from National Taiwan University, Taipei, Taiwan, in 1991, and the M.E. degree in Computer Science and Information Engineering from National Taiwan University, Taipei, Taiwan, in 1993. He is currently a graduate student in the Computer Science Ph.D program at State University of New York at Stony Brook. His research interests include multimedia systems, computer graphics, and efficient algorithms for volume visualization and compression.