

# Use of Shot/Scene Parsing in Generating and Browsing Video Databases

by

Allen Shu

Submitted to the Department of Electrical Engineering and Computer Science  
in Partial Fulfillment of the Requirements for the Degrees of

BACHELOR OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING AND  
MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the  
MASSACHUSETTS INSTITUTE OF TECHNOLOGY  
May 1995

Copyright ©1995 by Allen Shu. All rights reserved.

The author hereby grants to MIT permission to reproduce  
and to distributed copies of this thesis document in whole or in part  
and to grant others the right to do so.

Author .....  
Department of Electrical Engineering and Computer Science  
Sloan School of Management  
May 26, 1995

Certified by .....  
Assistant Professor of Computer Science and Media Technology  
Thesis Advisor  
Michael J. Hawley

Accepted by .....  
Chairman, Department Committee on Graduate Theses  
MASSACHUSETTS INSTITUTE  
OF TECHNOLOGY  
F. R. Morgenthaler

AUG 10 1995

LIBRARIES

Barker Eng



**Use of Shot/Scene Parsing in  
Generating and Browsing Video Databases**

by

Allen Shu

**Thesis Reader**

Reader .....



Glorianna Davenport  
Associate Professor of Media Technology



# Use of Shot/Scene Parsing in Generating and Browsing Video Databases

by

Allen Shu

Submitted to the Department of Electrical Engineering and Computer Science

May 26, 1995

in Partial Fulfillment of the Requirements for the Degrees of  
Bachelor of Science in Computer Science and Engineering  
and Master of Engineering in Electrical Engineering and Computer Science  
at the Massachusetts Institute of Technology

## ABSTRACT

There is a large volume of video which exists in the world today, stored in a multitude of different forms. In order to make this information more easily accessible, we need methods of organizing this data in an intelligible manner. This thesis suggests a method of cataloging and browsing videos by extracting key frames from the individual shots and scenes in the film. The algorithms presented herein were tested on video containing a wide variety of images (interviews, sports, computer animation, etc.) They provide a way to automatically segment an audio/video stream into shots and scenes, which could then be used to generate a database for many applications, from logging a broadcasting station's archives to browsing video in a digital library.

Thesis Supervisor: Michael Hawley

Title: Assistant Professor of Computer Science and Media Technology



## Acknowledgments

I would like to thank Mike Hawley and Glorianna Davenport for their patience, understanding, and helpful advice. Their ideas and thought provoking comments led me to explore new areas — without them, this thesis would never have been started, much less completed. I'd also like to thank the following people: Dave MacCarn, who opened up the resources at WGBH; Martin Szummer, for all his useful suggestions and ideas; and the gang in Interactive Cinema for their help when I needed it.

I would like to give special thanks to my family: mom, dad, little bro — they've been behind me the entire way. And to all my friends, whose kindness and understanding supported me when I thought I was falling. Special thanks to Mike Seidel, the honorable knight, with whom I have celebrated victories and mourned losses; Anthony Patire, whose friendship has lasted through heaven and hell; and Joannah Smith, who was always there to listen, and whose warm hugs make life that much happier.





# Contents

<u>Section</u>	<u>Page</u>
1 INTRODUCTION .....	11
1.1 Video Parsing .....	11
1.2 Overview of this Document .....	13
2 BACKGROUND .....	15
2.1 WGBH Video Logging .....	15
2.2 WGBH Captioning .....	18
2.3 Digital Libraries .....	19
3 SHOT PARSING .....	23
3.1 Algorithms .....	23
3.1.1 Preprocessing .....	24
3.1.2 Statistics .....	24
3.1.3 Hypothesis Testing .....	25
3.2 Comparison .....	26
3.2.1 Scenes of Boston .....	27
3.2.2 NOVA Science Documentary .....	27
3.3 Factors Affecting Shot Detection .....	30
3.3.1 Transition Types .....	30
3.3.2 Frame Rate .....	31
3.3.3 Resolution .....	32
3.3.4 Color .....	32
3.3.5 Video Content .....	33
3.4 Other Systems .....	33
4 SCENE PARSING .....	35
4.1 Captioning .....	35
4.1.1 Artificial Intelligence .....	36
4.2 Audio .....	37
4.2.1 Background .....	37
4.2.2 Audio Processing Tools .....	38
4.3 Algorithms .....	39
4.4 Results .....	40
4.5 Factors Affecting Scene Detection .....	41
4.5.1 Audio/Video relationship .....	41
4.5.2 Audio Content .....	42
4.5.3 Audio Rate .....	42

4.5.4	Bin Size .....	43
5	INTERFACE .....	45
5.1	Users .....	45
5.1.1	Digital Library .....	46
5.1.2	WGBH .....	46
5.1.3	Film Making .....	47
5.2	Browser .....	48
6	FURTHER RESEARCH .....	53
6.1	Shot Parsing .....	53
6.2	Scene Parsing .....	54
6.3	General .....	55
7	CONCLUSIONS .....	57
	APPENDIX A – Shot Parser Code .....	59
	APPENDIX B – Scene Parser Code .....	67
	BIBLIOGRAPHY .....	77

# Chapter 1

## Introduction

There is a large volume of video which exists in the world today. This video is stored in a multitude of forms, from magnetic tapes of varying sizes to laserdisc. In the past, video was manually parsed and logged, and browsing through video clips was a difficult and time consuming process. Video is now transitioning to a digital format. This new medium provides the opportunity to segment video into different levels of granularity by using parsing algorithms. These parsers can aid in the creation of a video database where the user can browse through videos much more easily, searching for a particular shot or scene.

### 1.1 Video Parsing

Videos can be divided into two broad categories, *rushes* and *final programs*. Rushes are the unedited video clips as taken from the camera. An editor puts these rushes together with audio and special effects to create the final program.

These videos can be viewed as linear stories or as granular elements. A *shot* consists of a single contiguous group of frames and has the finest level of descriptive granularity. In final programs we can identify a coarser level of granularity encompassing thoughts and actions within the story — the *scene*. For example, a video such as *Terminator* has several fight scenes, a love scene, a chase scene, etc. In a

documentary, a scene is determined through changes in content, such as transitioning from a talk about lions to a talk about zebras.

In the past, rushes were manually parsed to obtain shots; these shots were then logged into a descriptive database. In creating a final program, an editor made an Edit Decision List (EDL) which contained a listing of all the shots used in the program, indexed by timecode. By using an EDL and the program script, one could browse for shots and scenes within the edit master. However this was still a fairly difficult task; browsing was done by using timecode indexed computer controlled tapes, which required expensive equipment and long search time. Furthermore EDLs and program scripts are not generally available for public use. Laserdiscs have increased “searching ability” by allowing the user to “jump” to a location on the disc, but even this requires some knowledge of the content of the disc (which someone must painstakingly annotate by hand). Thus if one simply wishes to browse through video, there is currently no publically accessible index or automatic indexing system to the contents of videos.

Parsing algorithms can be used to analyze the audio/video stream and automatically index it. There are two aspects of indexing, both of which can be achieved by a parser: segmentation and descriptive content base. The parser can be used to segment the video into different levels of granularity, namely shots and scenes. After segmenting the video, the content of these segments can be described with text and/or pictures. Traditionally, text has been used to describe content. However, the old adage “a picture is worth a thousand words” seems to be true, as people are able to recall a scene better through images than through words. Thus, we can obtain a good overview of a video by taking representative samples from these shots/scenes, which are called *key frames*. These key frames should be representative of the aspects of each specific part of the video; the parsing algorithm can obtain good key frames by extracting either the first, the middle, or the last frame of each shot/scene.

Once we have obtained sequences of key frames for both the shots and the scenes in a video, we can create a video database. This database could be useful for video editors, corporations, and the general public. No longer will a person have to fast forward through video looking for a particular clip. Instead he can browse through the “shots” or the “scenes” of a video. Furthermore, we could cross index these shots with textual information and enhance the capabilities of the video database. Text

queries and possibly video queries (texture, color, etc.) will increase the user's ability to search and browse video. The final result is a rich database system which allows for more efficient browsing and intelligent searching for desired video clips.

## 1.2 Overview of this Document

The work described in this document spanned several groups at the MIT Media Lab, namely the Interactive Cinema group, the Vision and Modeling group, and the Personal Information Architecture group and Library Channel. Furthermore, the research was done in collaboration with WGBH Public Broadcasting.

Chapter two describes some background information about the current state of affairs at WGBH and the research being conducted on Digital Libraries.

Chapter three describes some background of cut detection, and the development of a video shot parser.

Chapter four describes the development of a video scene parser.

Chapter five describes considerations for designing an interface for using the shot/scene parsers described in chapters three and four.

Chapter six describes further research which might be conducted in this area, and some of the open issues of this project.

Chapter seven gives concluding remarks and observations.



# Chapter 2

## Background

The work described in this document was done in collaboration with WGBH Public Broadcasting. WGBH wished to automate the input phase of their video logging, and to create a richer database of information than presently exists. They are currently using an older system for logging and cataloging their videos. Furthermore, automated video logging and the shot/scene browsing mechanisms will be very useful in the ongoing digital library research being conducted at the MIT Media Lab. Having a video database and a fast method of searching and browsing is both desirable and necessary for public use of the Library Channel.

Below we will look at the current state of affairs at WGBH, as well as the digital library research which is being conducted today.

### 2.1 WGBH Video Logging

Currently, WGBH video logging is performed mostly on raw footage, that is the field tape prior to being edited and compiled into a final program. Although Avid software is used to make the logging process somewhat easier, the process is still highly dependent on human input. Below is the process which Alexis Massie, an employee of WGBH, uses in logging rushes [Massie].

First, she takes the beta videos that she receives and watches them carefully, taking copious notes on the content of these videos. These notes focus primarily on who the people are in the video, and most importantly *where* they are. The location is one of the most important comments to be input into the log; this same item is also very difficult to discover in an automated system.

Then the video is placed into a Sony BetacamSP Videocassette Recorder UVW-1800, which is connected to a Sony Trinitron monitor, and a Macintosh LCIII computer running Avid Medialog software. Using the Medialog controls, Massie can rewind and fast-forward through the videotape linearly in time. That is, the speed of the tape is variable, but it does not allow “jumping” through the tape. Scanning the videotape, she looks for scene breaks, which are usually identified by a “snap into a new image.” However, since these are rushes, there may be many takes of the same scene on the videotape; all of these takes are logged as one scene.

Medialog stores the beginning and ending timecodes for each scene, as determined by Massie. By consulting the notes taken during her previous viewing of the videotape, she types in a “name” for the scene, e.g. “Priest walking toward statues.” Furthermore, she records the subject, which in our example is “Religion.” Then she records whether the scene is sync or silent, meaning whether there is audio for the scenic views, or whether it is a silent scene. The “Roll #” is the identification number placed on the videotape when it is stored in the WGBH archives. The “Event Location” is taken from the viewing notes — in this case, “Palos, Spain.” Finally, the “Creation Date” is taken automatically from the computer’s internal clock. In the logging process, Massie often watches only the beginning of each scene to get an impression of what the scene is about. Since speed is an essential concern when logging (and even more important when meeting a program deadline), sometimes specific details about the videos are not that important in the process. A sample of the Medialog user interface is shown in Figure 2.1.

Each videotape is logged in a single bin within Medialog. This bin is then exported as a text document which is imported into Claris Filemaker Pro. Filemaker takes the imported text document and puts each description “name” into its own record; each scene is indexed separately within the flat-file database. A simple search verifies that the data has indeed be imported into the database. These records are then copied over Appleshare to the main server, where it is accessible to anyone.



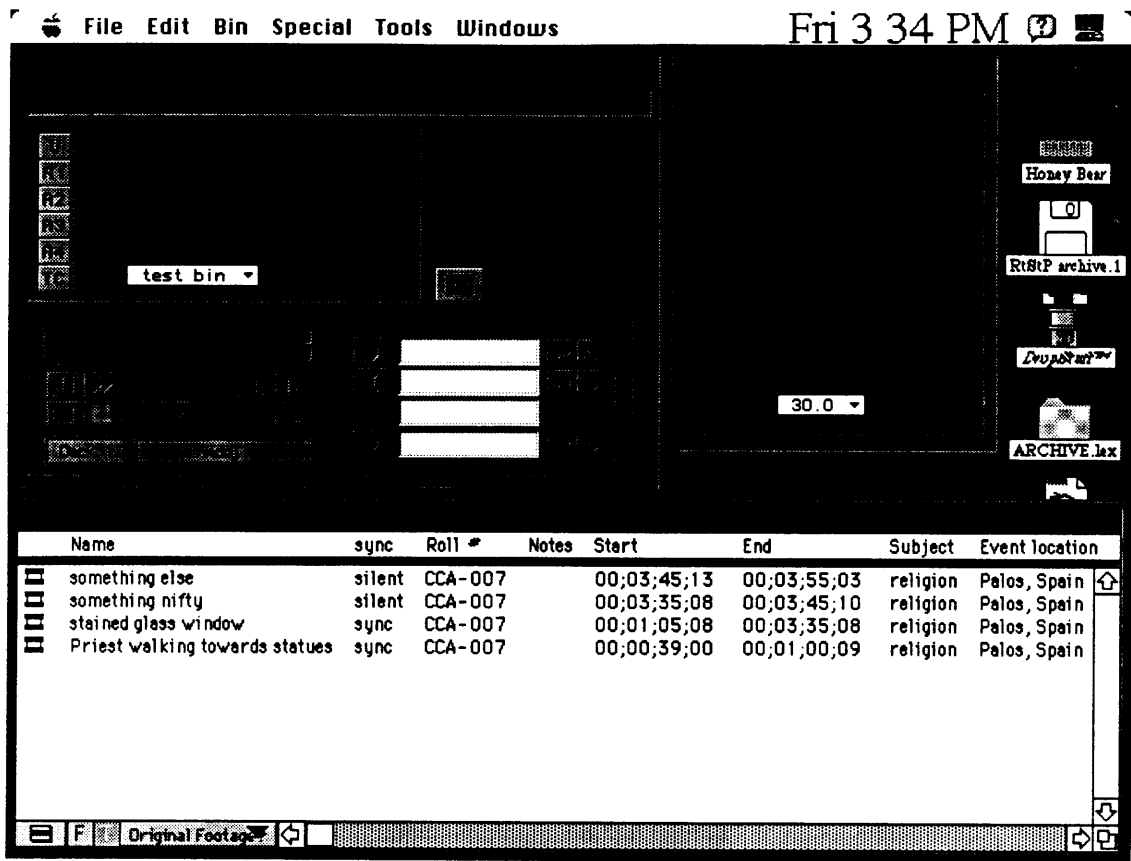


Figure 2.1: Medialog interface

By law, all stock footage must be logged. If the footage is original (and therefore owned by WGBH), this *original footage* is logged, but not the final programs which are made from this footage. Some final footage is logged, and this footage often contains captioning information. This captioning process will be discussed later in section 2.2.

As shown above, human input is essential for video logging at WGBH. Furthermore, the data is being stored in a simple flat-file database, with searches being conducted on the inputted data. The desired videotape must then be retrieved from the archives and viewed to determine if it indeed contained a desired scene. By storing key frames of each scene, we can create a richer database which would be very helpful in locating desired scenes.

## 2.2 WGBH Captioning

The captioning center at WGBH handles both on-line captioning (real-time) as well as off-line captioning. In dealing with on-line captioning, a stenokeyboard is used which is linked to a PC computer running Captivator software from Cheetah Systems. The captions are then translated real-time and are transmitted directly into the videofeed (whether it is from ABC, CBS, etc.)

Off-line, PC computers are used which run CC Writer, a software package developed at WGBH specially for writing captioning information. Basically, it is a word processor package which prepares and transmits the the data with the proper codes. The computer is connected to a Sony VCR system and monitor, as well as a character generator/decoder and a Videomedia controller.

These off-line videos can be received weeks or months in advance of the actual air-date, and include everything from new production videos to rerun programs to industrial presentations. A person sits at the computer console and listens to the audio track of the master video, and writes the appropriate dialogue and sound effects into CC Writer. The software synchronizes the caption text to the audio/video being shown by using the timecode of the program. The caption writer keeps the reading to approximately 150 words per minute, and is very careful about both the placement

of the words on the screen (displayed in 3 lines at the bottom of the screen), as well as the synchronization times of the caption with the audio/video. After the entire caption information has been written for the video, another person will go through the video with the sound on and off and “fine-tune” the product.

When the final product is ready, the CC writer file is “crowded” into encoding form. That is, the caption file data is put through a video caption encoder while the videotape is being played. This video caption encoder takes the caption file, encodes it, and inserts it into data line 21 (NTSC standard). The output is recorded onto a new videotape, which is the CC Master. This tape is then sent to the appropriate agency for final use.

WGBH also has specialized software which reads the captioning information off any broadcast (by converting line 21), and then storing this caption data as a text file.

## 2.3 Digital Libraries

The digital library research being conducted at the MIT Media Lab is about understanding the bits which represent sound, picture, and video. The goal of this research is to build a “Library Channel,” where the vast quantity of information which is currently stored in conventional libraries will be made accessible to the public’s living rooms. The library channel will include material such as:

- photographs, text, and video from the National Geographic Society
- audio and image material from the Library of Congress
- scientific and educational documentaries of WGBH
- selected audio from ABC Radio

The digital library is not limited to these information sources; it is a “rich mix of forms and interactions.” Each of these forms of information will require different

methods of processing in the back-end, yet each should present a similar interface to the user, so that people can access the information easily.

The principal areas of research at MIT are: 1) converting and refining the “old” forms of information into digital forms, 2) resolving intellectual property and economic issues, 3) developing content-oriented storage and transaction mechanisms, 4) developing new interfaces, delivery channels, and useful applications. This is a fairly comprehensive list of research topics, ranging from low-level bit analysis and image indexing systems to copyright and socioeconomic issues [MIT]. Furthermore, digital library research is being conducted in other schools, and similar issues are being addressed. These research teams are all developing a widely accessible information source.

The current research in picture processing at MIT revolves around building tools for content-based searching of videos and image databases, within the context of the already existing Photobook image database system. The Photobook system makes direct use of the image content, instead of relying on annotations. Since images contain vast quantities of information, annotating every item in the image and their interrelationships would be very difficult. Photobook solves this problem by using a semantics-preserving image compression, which is a compact representation to preserve the essential image similarities. Thus the user can browse image databases quickly and efficiently by using both the direct image content and the annotation information stored in an AI database. The interactive browsing uses an X-windows interface, thus allowing the user to pose flexible queries such as “show me images that look like this.” Three applications discussed in [PPS] are the “Face Photobook”, the “Shape Photobook”, and the “Texture Photobook”. To relate this to video processing, we must make use of a combination of all these three applications.

A system which parallels the Photobook research is the Query By Image Content (QBIC) system developed at the IBM Almaden Research Center. This system creates a database automatically by finding color, texture, and 2D shape features, and drawing a “global sketch” of the image. These indices are used to create a multidimensional feature vector for each image. The queries take the form of an  $n$ -dimensional feature vector, and the search/retrieval is done by finding close matches to this vector by using a matrix calculation. For example, if  $V$  and  $W$  are feature vectors, and  $G$  is the identity matrix, the calculation  $(V - W)^T G (V - W)$  would give

us the Euclidean distance between the images. The user could then give more specific queries based upon the images returned. To make this a more effective process, the feature vectors should be made to include very rich descriptions, thus providing more specific characterization of the local image region or object. They should be invariant under irrelevant variation, so that only the pertinent details are considered. Also, the metric being used should be analogous to what the user subjectively uses to determine “what is similar” [Berkeley].

At MIT, one approach to image indexing and retrieval is to collect low-level features such as orientation information and combine them with contextual information. This integration is a complex problem, and a final goal might be to correlate the features discovered with a “descriptive thesaurus.” This link would allow the content of a picture to be automatically described and catalogued, and also enhance retrieval mechanisms.

The image processing techniques described above can also be applied to video processing by reducing the video to a set of still images (e.g. a 2-hour feature film can be reduced to 1200 still images). However, it is more interesting (and useful) to extend the dimensionality of the analysis to include time, and use X, Y, and T coordinates. XYT analysis has already been used to identify the presence of a walking person. XYT analysis can also be used to study camera motion. By using motion analysis based on affine transforms, we can annotate video sequences with regard to their changing camera parameters [MIT].

Another line of research being conducted is that of wavelet decomposition. Wavelet decomposition allows an image to be decomposed into a number of subimages of progressively lower resolutions. By selectively combining these subimages, we can reconstruct images of higher resolution. This procedure has two advantages: 1) it is memory efficient (since storing all the subimages requires the same amount of memory as the original image), and 2) it enhances speed (by storing the lower resolution images in high-speed memory). This reduced resolution data could be used for quick graphical browsing in a user interface [UCSB].

For browsing video, it is necessary to be able to “fast forward” through the video clip in an intelligent manner. That is, we could fast forward through the entire video stream, or we could fast forward by shots and scenes. This type of “smart browsing” can be done by using annotations in the video clip. However, this annotation process

requires a person watching the videoclip and then typing in the appropriate segments. Automating this process by using scene parsing and automatic annotation would make the input phase of the browsing much more efficient. These parsers would select a key frame from the shot/scene. Good key frames often occur at the beginning, middle, and end of clips or scenes, and selection of good key frames will make it possible to browse through video quickly by reducing the search area to the processing of a few individual images [PPS].

By integrating shot/scene parsing with the above digital library research and WGBH's systems, we can develop a very useful system for many people to use and appreciate.

# Chapter 3

## Shot Parsing

As mentioned in the previous sections, a shot is a single continuous group of frames. In raw unedited video, these shots are the result of turning one camera on, shooting the scene, and turning it off again. In edited video, these different shots can be put together with several different types of transitions: cuts, dissolves, fade ins and fade outs, and wipes.

A cut is a transition whereby the camera stops recording one scene and starts shooting another. In unedited video, cuts are the only transitions between shots. Cuts are also the most common transition in edited video. In a dissolve, the intensity of one frame increases from zero to normal, while the intensity of the other frame decreases from normal to zero. Fade ins and outs are similar to dissolves, except that one of the frames is a blank one. Wipes occur when the one frame “slides” into the other frame at a particular angle, taking its place. The algorithms presented here are specifically designed to detect cuts (being the most common transition), and will sometimes also detect the other transitions.

### 3.1 Algorithms

A general cut detection system consists of three stages: preprocessing, statistics, and hypothesis testing, as shown in Figure 3.1 [Szummer]. The raw video is introduced

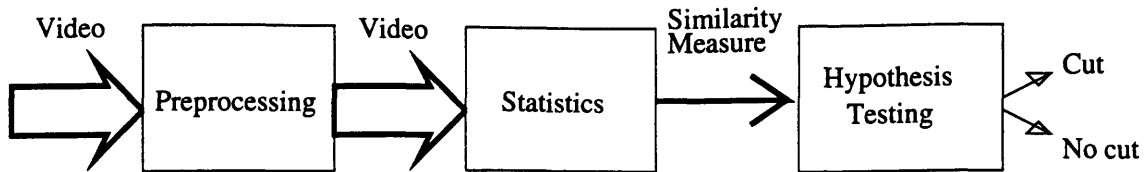


Figure 3.1: Cut detection modules

into the preprocessing module, which applies spatial filtering or spatial subsampling to each frame in the video to obtain a new video. This new video is inputted into the statistics module which then outputs a single number for each pair of successive frames that describes their similarity. Finally, this number is used in a hypothesis test to determine whether a cut has occurred.

### 3.1.1 Preprocessing

Several different types of preprocessing can be used. Low-pass filtering or smoothing of the image may make the cut detector less sensitive to noise and camera motion or object motion. However, smoothing and filtering of an entire image is a computationally expensive operation. Unless a hardware implementation is developed which can quickly perform these operations, filtering may not be feasible for real-time shot detection. Another possibility is the use of subsampling, which reduces the number of pixels to be processed in the statistics module.

### 3.1.2 Statistics

The statistics module calculates a difference measure  $d_i$  for every pair of successive frames. Successive frames within a shot are usually very similar to each other, and cuts introduce discontinuities which will result in a large difference measure. A cut is reported when the difference is large. Listed below is a summary of the statistics used to calculate  $d_i$ , as described by Hampapur et al, Nagasaka et al, Otsuji, and Elliot [HJW1, HJW2, NT, OT, Elliot].



- Difference in gray-level averages over the whole frame
- Number of pixels that change in gray-level by more than  $\Delta$
- Sum of the absolute difference of gray-level or color values corresponding pixels.
- Histograms: Histogram buckets group pixels of similar gray-levels or similar colors. Let  $H_i(b)$  denote the number of pixels in histogram bucket  $b$  at frame  $i$ .
  1. Difference of gray-level or color histograms.

$$d_i = \sum_{b=1}^{64} |H_i(b) - H_{i-1}(b)|$$

2. Chi-squared test of gray-level or color histograms.

$$d_i = \sum_{b=1}^{64} \frac{(H_i(b) - H_{i-1}(b))^2}{H_i(b)}$$

Since the performance of the cut detector depends of the combination of all modules, it is not possible to analyze the statistics in isolation. However, the statistics above have been listed roughly from worst to best performance according to Nagasaka and Tanaka. Clearly the histogram techniques are superior, but the differences between the chi-square and the simple difference methods are small. The statistics described above are first order, since these are the simplest techniques to use. Robust estimation techniques can be added to any of the above techniques, and help eliminate errors due to noise in the image (such as object motion or local light sources) [Szummer].

### 3.1.3 Hypothesis Testing

Below are listed three methods used in determining whether a cut has occurred. In all methods, a cut is reported when the values  $a_i, b_i, c_i$  exceed a threshold  $T$ .

- $a_i = d_i$
- $b_i = d_i - d_{i-1}$
- $c_i = S_{T_1}(d_i) - \max_{T_2} \min_{T_2} S_{T_1}(d_i)$  where  $S_{T_1} = \min_{T_1} \max_{T_1}(d_i)$

The first method [Elliot] detects a scene change whenever the two compared frames are sufficiently different. However, camera pans or zooms can cause the value of  $a_i$  to be high for several frames, thus causing false reports of scene changes. This problem is addressed in the second method, which is commonly used. In this method  $b_i$  takes the difference of successive  $d_i$ 's (a time derivative of  $d$ ). Thus  $d_i$  can be large without reporting scene changes, as long as it does not change rapidly. Furthermore, this method produces sharper peaks than the first method, thus making it easier to find a good threshold.

Problems with the second method are realized when the video contains slow motion, animation, or when the video has been frame-rate converted from film [OT]. In these cases, frames may be repeated causing  $d_i = 0$ , and thus resulting in a very large difference and false reports of scene changes. Otsuji and Tonomura thus introduce the third method to overcome these problems. The parameters  $T1$  and  $T2$  measure the number of frames to be used in the *min* and *max* operations, and are typically set to  $T1 = 4$  and  $T2 = 2$ . Using a simple statistic, namely the number of pixels which have changed in gray level by more than  $\Delta$ , and this hypothesis test, Otsuji and Tonomura report cut detection rates of 98% for nearly three hours of video (taken from a sports show, a news cast, and a movie.) It appears that the different hypothesis testing techniques may be important; however, these different techniques have not been compared here.

## 3.2 Comparison

Szumner implemented four different statistics and used a hypothesis test based on the thresholding  $b_i = d_i - d_{i-1}$ . The statistics are:

- Algorithm 0: Color histogram
- Algorithm 1: Chi-squared color histogram
- Algorithm 2: Chi-squared color histogram with robust statistics
- Algorithm 3: Gray-level histogram

The histograms all had 64 bins. Preprocessing was applied only before the gray-level histogram, where a  $3 \times 3$  Gaussian smoothing filter was applied twice. The robust

Table 3.1: Cut detection on “Scenes of Boston”

Statistic	Resolution	% False positives	% Misses
Color histogram (RGB)	160×1	18%	14%
Color histogram (RGB)	160×120	7%	9%
Gray-level histogram + preproc.	160×120	5%	5%

chi-squared histogram divided the image into  $4 \times 4$  subblocks and discarded half of the most dissimilar blocks. Currently, the algorithms require that the digitized video be in raw non-interlaced RGB format. I modified the algorithms to allow for different searching patterns, which will be discussed further in section 3.2.2.

### 3.2.1 Scenes of Boston

The data for the test performed by Szummer consisted of 10 minutes of video from and unedited documentary with predominately outdoor scenes, which we have called “Scenes of Boston.” It was grabbed at a resolution of  $160 \times 120$  pixels and a frame rate of 3.5 frames per second. There were a total of 2144 frames and 106 scene changes (as labeled by Szummer).

His results are summarized in Table 3.1. They showed that gray-level histogram with preprocessing achieved the best results, better than both the chi-squared histogram and the robust chi-squared histogram. Furthermore, Szummer observed that few pixels are needed to perform a scene cut detection. As shown in the above table, using only 160 pixels from the second scan horizontal scan line resulted in about 18 false positives and 14 misses when using the color histogram.

### 3.2.2 NOVA Science Documentary

To further test the effectiveness of the statistics in detecting cuts, I digitized the first 28 minutes of a 54 minute NOVA episode entitled “Can Science Build a Better Athlete” which was provided by WGBH. The hardware setup was as follows: a VHS video cassette recorder was plugged directly into the SunVideo digitizing card on a Sun Sparcstation 20. By digitizing onto a local drive, I was able to grab data at a

Table 3.2: Cut detection on NOVA documentary

Statistic	% False positives	% Misses
Color histogram (RGB)	15.1%	23.4%
Chi-square color histogram	22.0%	33.7%
Robust Chi-square color histogram	15.5%	19.2%
Gray-level histogram + preproc.	20.3%	18.6%

resolution of  $160 \times 120$  pixels and a frame rate of 30 frames per second. There were a total of 50,400 frames with 291 shots. Furthermore, this proved to be a full test of the capabilities of the shot parser to detect all types of shot transitions. Of the 291 shots, the transitions break down as follows: 248 pure cuts, 27 fades, 3 wipes, and 12 graphic overlays. A graphic overlay is when the principal image intensity is reduced by half, and a computer generated graphic (such as a chart or graph) is placed over the principal image. Sometimes this graphic overlay was steady and sometimes it changed, and the principal image was always in constant motion.

This footage contained shots which were in color as well as black-and-white. There were several slow motion scenes in both color and gray-scale. The footage contained personal interviews with stationary people, action footage of various sports (basketball, swimming, track and field events, gymnastics, etc.) and computer generated images. Combined with the different types of transitions and special effects described above, this NOVA documentary provided a wide range of tests for the shot detection statistics.

The full  $160 \times 120$  resolution was used in the algorithms, and several trial runs were used to determine an appropriate threshold for each statistic. The thresholds used were: algorithm 0,  $th = 1.2$ ; algorithm 1:  $th = 2.5$ ; algorithm 2:  $th = 1.4$ ; algorithm 3:  $th = 0.75$ . Due to disk-space considerations, the 28 minutes of footage was divided into segments of about 8500 frames (about 4 minutes, 40 sec), and the data was compiled together. The results of these tests are summarized in Table 3.2. We can see that the gray-level histogram with preprocessing has the lowest miss rate; however, the robust chi-square has a lower false positive rate. The simple color histogram also performed fairly well.

A unique feature of this film was that between pure cuts, there was an extra frame which was a juxtaposition of the frames before and after the cut. This is likely caused by the video editing system which was used; it is hypothesized that the edit

Table 3.3: Modified cut detection on NOVA documentary

Statistic	% False positives	% Misses
Color histogram (RGB)	8.6%	23.4%
Chi-square color histogram	14.7%	33.7%
Robust Chi-square color histogram	6.9%	19.2%
Gray-level histogram + preproc.	12.3%	18.6%

system cut on field 2, thus creating the juxtaposed frame. This interlacing caused problems with the shot parser when it used the full image in the statistic (160x120), since it would sometimes detect both the interlaced frame and the next frame as a scene cut. I attempted to eliminate this by modifying the algorithms so that they would only use either the odd horizontal scroll lines or the even horizontal scroll lines for the pixel detection. The threshold was half of what was used before, since the difference measure is using half the number of pixels as before. Also, the algorithms were changed so that they could check for a vertical interlacing and checkerboard interlacing (with appropriately modified thresholds). It appears that the interlacing does not follow a simple pattern, since these modified algorithms did not completely eliminate the problem. They reduced the false-positive rate slightly, but they also increased the miss rate slightly – both of which were not statistically significant changes.

Although it appears that these algorithms could not automatically filter out the interlaced frames, it is interesting to see what the results would be if these frames were removed manually. These results are summarized in Table 3.3. An average of about 22 false positives (7.6%) due to interlaced frames was removed. From this data, it appears that the only truly poor statistic was the chi-square color histogram. For general video, the color histogram, the robust chi-square histogram, and the gray-level histogram with preprocessing all performed on approximately the same level.

It is interesting to note that most of the misses were the result of the different types of transitions. That is, the fades, wipes, and graphic overlays often presented difficulty for the shot parsing algorithms. If we subtract out all non-cut transitions from the miss rate and re-calculate the miss-rate, we obtain much lower numbers, as shown in Table 3.4. These results are similar to those obtained by Szummer in his tests on the “Scenes of Boston” data. Please note that Table 3.4 is an estimate of the miss rate by simply subtracting out the non-cut transitions. Actually, some of

Table 3.4: Modified cut detection on NOVA documentary – cuts only

Statistic	% False positives	% Misses
Color histogram (RGB)	8.6%	9.0%
Chi-square color histogram	14.7%	19.3%
Robust Chi-square color histogram	6.9%	4.8%
Gray-level histogram + preproc.	12.3%	4.2%

the statistics had slightly better success in detecting these effects, as will be discussed in the next section. These algorithms may work better with raw unedited footage; production footage (with the different transitions and “special effects” seems to pose more problems.

### 3.3 Factors Affecting Shot Detection

The different types of transitions will clearly affect the performance of the algorithms. Furthermore, the frame rate, the resolution, the use of color, and the content of the video will also affect the performance. These factors are considered next.

#### 3.3.1 Transition Types

Any transitions which take place over several frames are much more difficult to detect. In the NOVA footage, there were 27 fades, of which 3 were four-frame fades. The other 24 fades averaged between twenty and thirty frames. The four different statistics were able to detect the four-frame fades fairly well, either reporting either two or all three of the changes. However, all of these statistics missed the long fades. This was expected, since the gradual fade does not create a large difference value between frames, and thus the shot is not easily detected by the algorithms. Furthermore, fifteen of the above fades were part of a horizontal scrolling scene in the footage. That is, there were several images which were scrolling through the screen, and simultaneously fading into new images. Since the scrolling image was fairly similar from frame to frame and since the fades were 30-frame fades, this scene was not detected by any of the algorithms.

There were 3 wipes which occurred in the footage. All the wipes moved from a

black-and-white image to a color image (or vice versa), and took place over about 30 frames. This lengthy period made these wipes very difficult to detect. Surprisingly, the gray scale histogram with preprocessing managed to detect two of the three wipes which occurred in this footage. The other three statistics missed all the wipes. I am uncertain of whether the gray-level rendering of the color image may have allowed for the detector to work where a color rendering of a gray-scale image had problems. Although further tests need to be conducted, the detection of the wipes by this algorithm makes it very attractive to use.

There were 12 graphic overlays in the footage. The worst performing algorithm (chi-square color histogram) detected only 4 of these transitions. The remaining three algorithms (the gray-level histogram with preprocessing, the robust chi-square color histogram, and the color histogram) detected about half of the shot changes which occurred during the graphic overlays.

Clearly, these algorithms are designed specifically for detecting cuts. However, the “better” algorithms managed to detect some of the other transitions and effects, with the gray-level histogram performing the best overall.

### **3.3.2 Frame Rate**

Having a low frame rate makes it difficult to detect shot changes, since there are more differences between successive frames as the time gap between them increases. Thus, a higher frame rate may reduce the number of false positives. However, since misses are the result of a similar successive frames (as determined by the algorithm), a higher frame rate is not likely to change the miss rate. Szummer’s footage used a frame rate of 3.5 frames per second, while my footage used the full 30 frames per second. I believe that having a higher frame rate is useful in reducing false positives, and that the higher number of false positives in my footage is the result of other factors (as discussed below).

### 3.3.3 Resolution

Szummer performed a shot-detection using only 160 pixels from the second horizontal scan line, resulting in a higher percentage of false positives and misses, yet still performing fairly well. By using less than one percent of the image, the algorithm can run more quickly. This can be very beneficial in real-time applications of shot-detection. On the other hand, using the whole  $160 \times 120$  pixels has much better performance but is much slower. One must determine which is more important, accuracy or speed.

### 3.3.4 Color

Some of the statistics use RGB color in their processing. For example, the implementation of the color histogram statistic discretizes RGB space into 64 bins, and counts how many pixels of a given bin-color there are by using the two-most significant pixels for R, G, and B. According to Nagasaka and Tanaka, color histograms perform better than gray-level histograms. However, Szummer's test suite showed that the gray-level histogram outperformed the color histograms. My tests showed that the gray-level histogram performed at a similar level to the robust chi-square color histogram and color histograms. It is possible that better results may be obtained by using a different color space than RGB.

In the NOVA footage, there were sequences of gray-level shots, color-shots, and transitions from gray-level to color. The statistics were able to detect almost all the transitions between gray-level and color. However, the color algorithms often had difficulty in detecting the shots in a black-and-white sequence. This may be explained by the fact that RGB algorithms are attempting to place pixels in the appropriate bin based on RGB information, and the RGB images of a successive black-and-white frames may appear very similar.

The NOVA footage also contained many sequences of computer graphics animation. These graphics had a limited number of colors (when compared with a real-world image). This limited color palette sometimes resulted in problems with the detectors, since there was not enough color contrast changes in the image.



### 3.3.5 Video Content

Certain types of video can present difficulties for a shot-detector. Szummer reported that uneven lighting conditions (such as large shadows), reflections on water, and flash can increase false positives. Furthermore, sports footage contains fast pans and rapid object motion can also cause false positives. In my tests using the NOVA footage (which contain all of the above) this was proven to be true. The NOVA footage also contained computer animation sequences and old video clips which were sometimes “choppy.” These non-smooth frame transitions fooled the algorithm into detecting a scene change when in fact none occurred, thus increasing the false positive rate.

Elliot found that music videos shown on MTV present a very challenging problem, and that color histograms are not adequate for determining a shot. Also, soap operas and other similar television shows containing many indoor scenes with similar backgrounds may increase misses.

In all these cases, it is often necessary to develop a more precise definition of a shot, since quick motion and computer animation changes the traditional “camera on-off” definition of a shot.

## 3.4 Other Systems

A system was built by Sasnett in 1985 which coded changes in image content (such as luminance and chrominance) for nine points spread through the screen. An algorithm was devised where the different regions of the screen (as determined by the points) voted on whether a scene change had occurred, based upon component differences and average luminance differences. Zooms, pans, and other such transitions were usually not able to be detected. Sasnett’s “Scene Detector” (which was actually a shot detector) captured 90% of valid changes. However, it also had between 15% and 35% false positives, depending on the scene content [Sasnett].

Pincever developed a system in 1991 which detected shots in raw footage by using audio parsing (which will be discussed further in Chapter 4). Pincever’s shot detector has two levels. In the first level, the soundfile was divided into chunks equivalent

to a frame of video, and the root-mean-squared (RMS) amplitude was obtained for each block. The first-level parser then looked for changes in the average RMS value which were larger than a set tolerance level, and reported the three frames preceding and boundary and the three frames after the boundary to the user as a possible shot boundary (and the user then made the final determination). The second-level parser was performed only if the first level seemed ambiguous. In this case, a Short Time Fourier Transform was performed on the soundfile and the average spectrum and average amplitude (RMS spectrum) were calculated. These were used with heuristics based on observations of raw footage to report a shot boundary. Pincever concluded that sound can be used to determine shot boundaries in raw video [Pincever]. However, it is uncertain if his algorithms can be used in edited production grade video.

One hardware cut detector which is sold in the market today is *Scene Stealer*, developed by Dubner International, Inc. It preprocesses video using a low-pass filter, and then subsampling the image by a factor of 7 in the horizontal direction only. It uses the full 30 frames per second of the video, and about 1500 points per field in the gray-level histogram statistic.

# Chapter 4

## Scene Parsing

As mentioned in the introduction, a scene is a unit of action or substance within the video. In feature films, there are often many scenes which are characterized by plot changes. As mentioned previously, a video such as *Terminator* has several fight scenes, a love scene, a chase scene, etc. Within a documentary such as the NOVA science documentary discussed in Chapter 3, there are swimming scenes, basketball scenes, etc.

Since scenes are determined by the *substance* of the video, it is possible that the beginning of a scene is not the beginning of a shot. However, it is often convenient to associate the beginning of a scene with the first shot of the scene. Another consequence of scenes being determined by content is that we cannot simply apply video processing algorithms to determine scene transitions.

### 4.1 Captioning

To parse a video into scenes, it seems at first glance that we must understand the substance of the video. The NOVA video obtained from WGBH is close-captioned, and it was thought that this could be useful. WGBH provided me with a caption decoder with a serial port which could be used to obtain the caption information off the video and dump it directly into the workstation. As with the specialized software

used by WGBH, the captioning information can then be stored as a text file. This text can then be used as part of the indexing scheme for a database containing both video and text indices.

The principal problem with using captioning to determine scene transitions is that we must interpret the text outputted by the decoder. That is, the scene transition detection problem becomes one of artificial intelligence and natural language processing.

#### **4.1.1 Artificial Intelligence**

Two artificial intelligence systems often used are rule-based and knowledge-based systems. A rule-based system uses a set of predefined rules which are programmed into the system to make decisions. A knowledge-based system has pertinent knowledge already stored, and the computer can use this knowledge to generalize or to perform some set of functions. This knowledge can take the form of rules (making knowledge based systems a superset of rule-based systems) or common knowledge. A common representation of such knowledge is a “heuristic” which can be thought of as “information dependent on a particular task”; these heuristics are guesses which are used to make inferences as to what is actually happening.

We can apply these concepts to observations of raw footage as well as the caption information. Heuristics can be formulated that define what a shot, sequence, and scene are in terms of the video image and the textual language. These heuristics can then be programmed into a rule-based module to examine data and make decisions. This is a very difficult task, and I shall not discuss any more of it here. However, natural language processing may reach the point where generalized substance information can be recognized; this breakthrough would be beneficial to many areas of research, including scene parsing.

## 4.2 Audio

Since understanding the substance of a video is very difficult to do using machine cognition, the task remains of finding an alternate method of segmenting video into scenes. The audio track of movies and programs may be useful in determining these scene transitions.

### 4.2.1 Background

When shots are first recorded, a passage of sound is often recorded along with it. This sound is called *synchronous sound*. Synchronous sound is made up of dialog and ambient sound. With most films used today, this synchronous soundtrack is physically bound to the picture. However, the digital world presents us with the possibility of having separate video and audio servers; in this case, an identifying tag must be used to guarantee that a synchronous relationship will be maintained between the audio and video files.

A different category of sound is *wild sound*. As explained by the name, these sounds are not specifically synchronized to the action of the video. For example, in the past when theaters were first being outfitted for projection, presenters would accompany these movies with live music. This music was often performed live and free form, selected to enhance particular reactions of the audience [DPAS].

Currently, the sound is locked to the moving image. Most of the audio tracks for films today (both documentaries and feature films) are created in dubbing studios and foley pits during the post-production process. Sound is often used to blur distinctions at the edges of shots, sequences, and scenes. Also, explanatory commentary or music may be added to make the information conveyed by the image more explicit, to heighten the perception of an action, or to set a mood [Pincever]. The result is a synchronous soundtrack which accompanies and enhances the video experience.

The importance of the soundtrack in a movie is evident if one watches almost any movie with the sound turned off. A person can often identify scene transitions by listening to the audio track and turning off the video. For example, an action scene is often accompanied by music which has a fast-beat, along with the transient

noises of gunshots or police sirens. A love scene often has romantic music in the soundtrack accompanying the image. A person's ability to determine a scene change by the audio track is based both on the transitions between levels of audio (e.g. dialogue → loud music) as well as a conceptual understanding of the emotions evoked by different soundtrack contents. This conceptual understanding is once again an artificial intelligence problem. However, there are many properties of an audio signal which we can exploit.

### 4.2.2 Audio Processing Tools

There are many analytical tools in signal processing which can be used to transform the audio data into a more useful form. Furthermore, statistical properties of the audio signal, such as average power of a block of samples, can be computed. This time-domain analysis of the audio signal can tell us where changes in the power of the signal occurs. This change in the audio signal may be useful in marking a scene change.

Another tool which may be useful is the Fast Fourier Transform, or FFT. The FFT allows for a spectral (frequency-domain) analysis of an audio signal. Although FFT's are very accurate (since the frequency domain provides a lot of information) they are also computationally expensive. Thus, the FFT can sometimes be replaced by simpler transforms.

Once the spectrum of the audio signal has been obtained, the average spectrum and the average amplitude (RMS spectrum) can be calculated. These techniques allow for analysis of the changes in the spectral content of a signal, as well as changes in the power distribution. If there are considerable changes in the power distribution of two successive audio samples, it can be assumed that the spectral content of these samples is different.

As mentioned in Chapter 3, Pincever uses these tools to find shot boundaries. However, his methodology may be extended to find scene boundaries in final production video.

## 4.3 Algorithms

I implemented three different algorithms to detect scenes using digitized audio, using a similar format as the shot parser discussed in the previous chapter. No preprocessing was used. The statistic used for the first two algorithms was the average power (RMS power) of the time-domain signal. Two different methods of hypothesis testing were used, as shown below. The third algorithm used the average amplitude (RMS spectrum) as the statistic. Since the work was done on a Sun Sparc 20, the program reads in audio files in any format which the Sun's internal audio programs can read. The algorithms were as follows:

- Algorithm 0  
Given two bins  $A_i$  and  $A_{i-1}$  of constant size.  
Compute  $RMSPower(A_i)$  and  $RMSPower(A_{i-1})$ .  
Thresholding value  $t = RMSPower(A_i)/RMSPower(A_{i-1})$ .  
That is,  $t$  is the percentage change in average power from one bin to the next.  
A scene transition is reported when  $t > T$  (given threshold).
- Algorithm 1  
Given three bins  $A_i$ ,  $A_{i-1}$ , and  $A_{i-2}$  of constant size.  
Compute  $d_i = |RMSPower(A_i) - RMSPower(A_{i-1})|$   
and  $d_{i-1} = |RMSPower(A_{i-1}) - RMSPower(A_{i-2})|$ .  
Thresholding value  $t = |d_i - d_{i-1}|$ .  
A scene transition is reported when  $t > T$  (given threshold).
- Algorithm 2  
Given three bins  $A_i$ ,  $A_{i-1}$ , and  $A_{i-2}$  of constant size.  
Compute FFT of each bin  
Compute  $d_i = |RMSSpectrum(FFT_i) - RMSSpectrum(FFT_{i-1})|$   
and  $d_{i-1} = |RMSSpectrum(FFT_{i-1}) - RMSSpectrum(FFT_{i-2})|$ .  
Thresholding value  $t = |d_i - d_{i-1}|$ .  
A scene transition is reported when  $t > T$  (given threshold).

In the first method, it was initially considered to use the thresholding

$$t = |RMSPower(A_i) - RMSPower(A_{i-1})|$$

However, this was considered to be impractical since these fluctuations may be rapid and cause a high false positive rate. By using a percentage thresholding value, it was

hoped to reduce this problem. The second and third methods use the same hypothesis testing as the shot parsing algorithms. This time derivative allows for high values of  $d_i$  without reporting scene transitions, as long as it does not change rapidly. Also, as mentioned in the previous chapter, this method produces sharp peaks, thus making it easier to find a good threshold.

## 4.4 Results

The audio being analyzed was that of the NOVA Science Documentary, “Can Science Build a Better Athlete.” The 54 minutes of audio was digitized at a rate of 8 kHz. The audio track of the documentary was composed of a narrator speaking, a music soundtrack, interviews, sports scenes and activities, etc. Choosing where the scene transitions took place was a difficult task. Different people will likely pick scene transitions at different times, as well as a different total number of scene transitions. In viewing the video, I determined that it was possible to perceive a minimum of 20 transitions and a maximum of over 50 transitions. I chose a middle ground at a slightly “higher level”, consisting of 28 scenes. These scenes were defined as “introduction”, “swimming scene”, “rifle shooting scene” etc., each of which had subscenes. For example, the rifle shooting scene contained subscenes of a man shooting a rifle, a computer image, etc. Although it is possible to choose these as individual scenes, I believe that it is better to view them as components of a larger scene.

Currently the program outputs a timecode corresponding to the scene change, which I then compared to my annotated list. I counted a “hit” if the algorithm chose a key frame within each scene. As mentioned in the introduction, a key frame should be representative of each scene, and therefore need not come at the beginning of the scene.

Since Pincever showed that the audio track can be used to perform shot parsing, this was not repeated. The size of the bins used was much larger than a frame. I tested bin sizes of one and two seconds (corresponding to 8000 and 16000 samples), which is the equivalent of 30 and 60 frames of the video. It is conceivable that if the bin size were set to one frame, Pincever’s results could be duplicated. Test trials resulted in setting an appropriate thresholding value for each algorithm. The results



Table 4.1: Scene detection on NOVA documentary

Algorithm	Bin size	T	% False positives	% Misses
0	8000	0.17	57%	29%
0	16000	0.13	46%	29%
1	8000	25	36%	36%
1	16000	19	29%	25%
2	8000	2.15	35%	25%
2	16000	1.44	25%	14%

of these algorithms are summarized in Table 4.1.

Algorithm 0 reported many false positives, although the larger bin size did reduce this rate somewhat. Algorithm 1 performed much better, especially when the bin size was increased to 16000. Algorithm 2 clearly had the best performance. The spectral data did indeed provide the most information about the audio track, and this algorithm found the “beginning” of the scenes better than the other two algorithms. However it was also the slowest algorithm. Although the results above may seem disheartening (since the miss rates and false positive rates are still fairly high), they fell within my expectations. The audio track of a film need not have a corresponding relationship with the scenes, especially in a documentary. In a feature film, this may be a much closer analog, since “action scenes” and “love scenes” are often accompanied by a particular audio track.

## 4.5 Factors Affecting Scene Detection

There are many factors which affect the detection of scenes using audio. These include the audio/video relationship, the audio content, the audio rate, and the size of the buckets being used. These factors are considered next.

### 4.5.1 Audio/Video Relationship

The relationship between the audio and the video of a film is probably the most important factor affecting the above scene detection algorithms. Since we are using

audio to find scenes, it stands to reason that if the audio has a clear link to the scenes, then the algorithms will be more effective. As described above, the soundtracks of many feature films are used to enhance the mood of the film; in these cases, it is likely that each scene will have a different mood, and thus a different background soundtrack.

In documentaries (such as the NOVA documentary being studied), the audio track is more often used to provide information. Also, a narrator is often speaking through scenes. This provides some continuity between the shots of a scene and helps with scene detection. In scenes where there is no “added” soundtrack, a constant ambient background sound level may give clues as to the boundaries of a scene. That is, a scene often contains multiple shots in the same location, and thus have the same background noise levels.

However, it is not always certain that the audio and video of a film’s scenes will be completely interconnected. Although “good film making” often relates these two, there is no guarantee that this will be the case.

## **4.5.2 Audio Content**

Even if the audio track is a 100% match with the video track, it is still possible that loud bangs and large variances in sound or spectrum could confuse the parser. The algorithms would produce higher false positive rate due to these volatile transitions. One might be able to compensate by increasing the bin size, but this can create other problems (as discussed below).

## **4.5.3 Audio Rate**

The rate at which the audio is sampled may make a difference in the detection algorithms. A faster sampling rate will result in greater smoothness of transitions and sound levels within a scene. Thus scene transitions may be more prominent and more easily detected. The NOVA clip was grabbed at 8 kHz, which is fairly low. Higher rates should be tried, up to the compact disc rate of 44.1 kHz.

#### 4.5.4 Bin Size

Bin sizes of one and two seconds were chosen for the interval. These values were based on a personal inspection of the audio track of the NOVA documentary. Using a larger bin size would have a similar effect to increasing the sample rate; it would smooth out the sound levels within a scene. However, one must be careful not to use too large a bin; this could result in too much smoothing across scene transitions and potentially increase the miss rate.

Using too small a bin has the opposite effect; the algorithm will pick up more of the fluctuations in the audio track. This could potentially increase the false positive rate.



# Chapter 5

## Interface

People perceive products on two levels: functionality and features, and ease of use. For this reason, the user interface is one of the most important aspects of a product. Much time is spent in designing a good interface for commercial products, and extensive pilot testing and customer feedback is used to improve the design.

I am not a graphic designer and I do not expect that the interfaces that I propose are exceptional. Here I will give an overview of some design considerations relating to the use of the shot and scene parsers discussed in Chapter 3 and Chapter 4.

### 5.1 Users

The first consideration is who will be using this interface. The shot and scene parsers can be used in many situations, and thus may have many different users. As mentioned in the background, shot/scene parsing could be incorporated into a digital library for use in browsing video. In this case, the users would be the general public. As mentioned by Pincever, shot/scene parsing could also be used to aid in home video creation from raw footage. Thus we would be considering a subset of the public — those people who want to create their own “home movies.” On the other end of this spectrum, we have professional movie editors. These people have specialized software which they use for editing; however, it is possible that they would benefit from an

automatic parser of film. Another specialized use of these parsers is for cataloging video, which is what WGBH mainly needs.

### **5.1.1 Digital Library**

For use in digital libraries, there are two aspects to consider: the final user side, and the input side. For the final user (which is the general public) the interface for the video browser should be simple to use. Ideally, the video would be stored completely in digital form where it could be recalled and played immediately. However since digital video takes up enormous amounts of memory, this is not always practical. Thus we could browse using the key frames pulled from the shots and scenes of the video. If the entire video (or sections of it) were stored digitally, an interface could be designed so that each key frame would be indexed directly into the video, thus allowing for more complex browsing. For example, after searching through the key frames for scenes, one could “choose” a video frame and get an short videoclip of that scene. Thus, there are several levels of browsing: shot, scene, and the entire video.

Inputting video and their respective key frames is an important task, but one which can be mostly automated. Thus, I am not really concerned with the input interface into the digital library. We could conceivably use the commands which are presently being used to create the video key frames. However, a suitable interface must be considered for taking this video and inputting it into the digital library as well as creating the cross indices within the database.

### **5.1.2 WGBH**

In WGBH’s case, digitizing their entire video archives is not really feasible at this point in time. Thus the interface would be used to browse through a multimedia database. A database of these key frames which was also annotated with text would prove very useful to film producers, since they could search for a specific topic and see a sample of the video immediately.

On the input side of this database, the parsers could be used to generate the

key frames automatically. However, one still needs to annotate the text part of the database manually. Currently, object recognition in these images is a difficult task, and much research is being done in this field. For example, the Vision and Modeling Group at the MIT Media Lab has done texture mapping work, whereby the texture of an image can be used to determine identity with a fairly good rate of success. For example, a person could manually annotate a scene describing “grass” and “sky”; the computer would then use these textures as a standard and map all other frames which have similar textures with the same key words [Szummer2]. Using these techniques and a video database system such as Photobook could eventually make this input/annotation process more automated.

Currently, video logging at WGBH uses Avid Medialog (as discussed in Chapter 2), and the user interface for this product is shown in Figure 2.1. It is conceivable that a similar user interface could be developed which utilizes these algorithms, but still allows the user to annotate the text part of the database manually, as well as having the final decision as to whether a shot or scene really took place. If this “over-seer” technique is used, the threshold for the algorithms would be tailored toward fewer misses.

### **5.1.3 Film Making**

For film makers (both professional and home), the interface for the browser must be useful for locating clips and putting them together, as well as logging them in the first place. Home users (who usually have little experience with video editing equipment) will likely desire an interface which is easy to use and understand in their editing process. Professional users have much experience with editing equipment and software, and thus any new browser should have similar features. For example, the most common “browsing mechanism” in a video editing room is the jog/shuttle button. In one mode, this allows the user to fast forward/play/rewind a videoclip by rotating the knob. In another mode, it allows the user to move through the film slowly and with precision by rotating the knob. This jog/shuttle interface has been seen recently on videocassette recorders.

It is possible that a “virtual jog/shuttle” might be used as a user interface on the screen. However, a problem with this is that it will not have the “feel” of a

jog/shuttle. Currently, WGBH (and most New England area filmmakers) use Avid Media Composer in their editing process. In the western U.S., software packages such as Videocube, Lightbox, and a Sony editing system, are used more often. These systems all have a visual interface which allows the user to see all the different tracks of video and audio, and how they overlay with each other. The standard play/stop/forward/rewind buttons are push buttons on the screen. The shot/scene browsing and logging mechanism could be a subsystem of this large video-editing system, and the interface would allow for the video (if in completely digital format) to be dumped directly into the editing software. Currently, the Avid software stores 1 hour 15 minutes of video on 2 gigabytes of disk space, and the audio is sampled at 44-48 kHz. They are stored separately and indexed together for editing, and the software dumps it all out to create the final production tape.

## 5.2 Browser

A proposed browsing interface is suggested in Figure 5.1. Although this interface is specific to the proposed concepts for WGBH, it can apply to other situations. This browser has the standard play, stop, fast forward, and rewind buttons. However, the step search in both directions is determined by whether the “shot” or the “scene” box is highlighted. If the video box is highlighted, the step button will show one frame at a time in the actual video (providing the entire video is digitized). Even if the algorithms discussed in this paper are not used, and the shot and scene key frames are manually extracted, this interface could still be used. Also, clicking on the key frame could bring up the full videoclip of the shot or scene chosen. The frame number could be used to “jump” to a particular frame in the video. The FPS value (frames per second) tells the rate at which the browser will show frames when the “play” button is pressed.

The “database search” key creates an interface between the video browser and database query mechanisms. These database query fields would be determined by the user. For example, suppose that the only query fields were the title of the video, the record date, and the keyword describing the shot/scene. Then pressing the “database search” key could bring up a window such as in Figure 3.2. This search could return a list of files in the “Select Files” window, which the user would click or type in, and



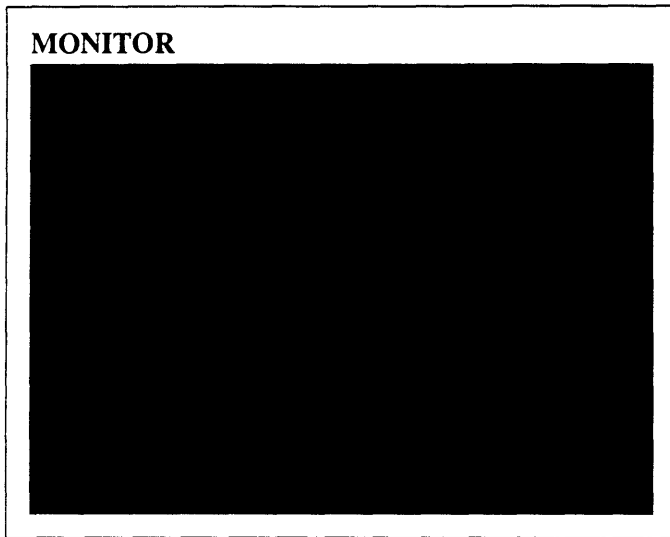
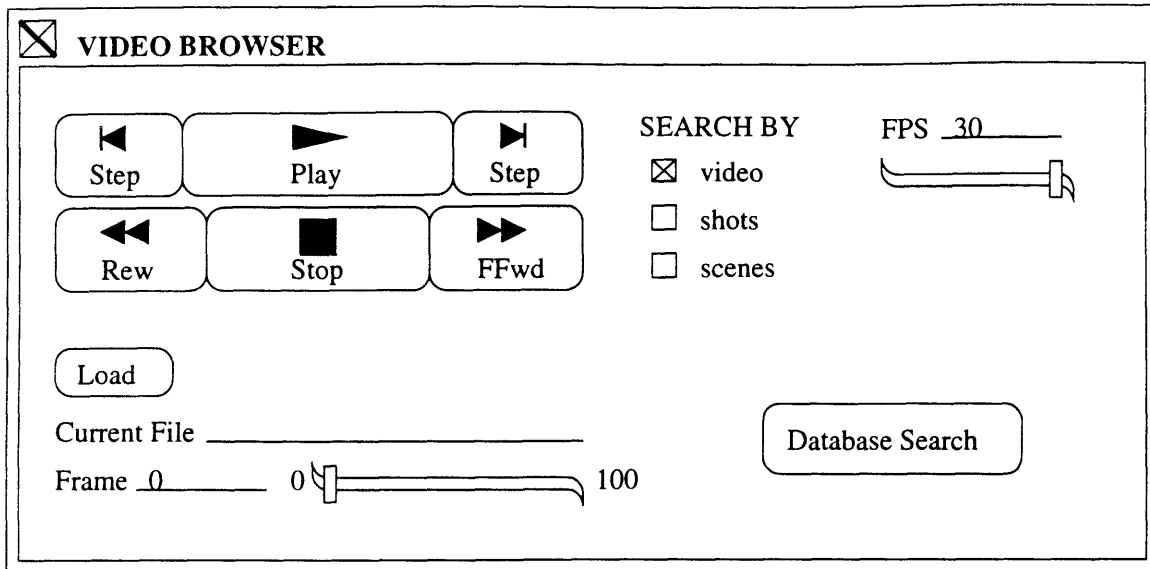


Figure 5.1: Video browser interface

**DATABASE SEARCH**

Title: \_\_\_\_\_

Record Date: From: \_\_\_\_\_ To: \_\_\_\_\_

Keywords: \_\_\_\_\_

**SELECT FILES**

Filename: \_\_\_\_\_

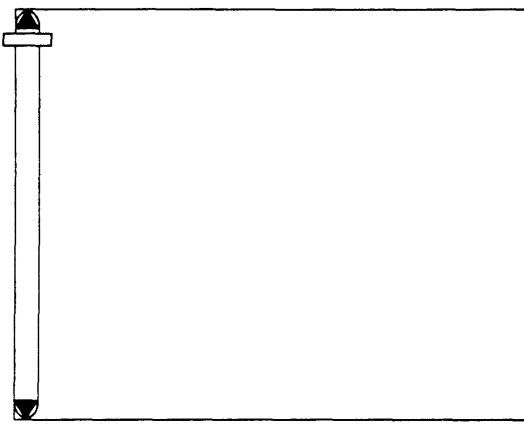
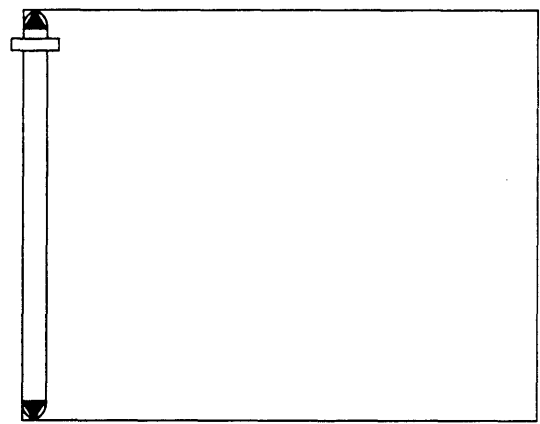
	Selected Files
	

Figure 5.2: Database query interface

the selected files would be copied from the left column to the right column. All the selected files would then be streamed to the video browser for viewing.

This suggested interface can be customized and improved upon for each of the specific user areas above. Here I am simply developing ideas regarding a good interface for video browsing.



# Chapter 6

## Further Research

There are many directions for further research in shot/scene parsing. The following are ideas which will provide greater flexibility and functionality to the systems, as well as improving the effectiveness of the current algorithms. As computing power increases, other methods may be developed using advanced parsing algorithms.

### 6.1 Shot Parsing

The NOVA science documentary provided a very rich footage against which to test the shot parser. The slow-motion sequences, animation, and numerous different interviews and sporting events was very useful to test the parser. However, it would be beneficial to test the shot detector against other sets of data, namely soap operas, news broadcasts, sporting events, and feature films. Although fairly good conclusions can be drawn from the NOVA footage, more data would give a better picture of what factors affect these detection algorithms.

The algorithms developed use a single hypothesis testing technique, and it was deemed more important to find a good statistic of image similarity. Further research would involve using the various different hypothesis testing techniques, such as the one discussed by Otsuji and Tonomura [OT].

Other statistics should be tested. For example, it might be possible to use pre-processing with a motion compensation algorithm, followed by a statistic comparing pixel differences to analyze spatial location. An understanding of optical flow would be useful in this analysis of spatial location and movement. Perhaps the texture of the video can be used to determine similarity. Also, second order statistics (or higher order statistics) should be tested as to their effectiveness in shot parsing. In doing this research, one must always weight the computation power required versus the benefits of the "improved algorithm."

Currently, the algorithms use raw video in a non-interlaced RGB format. Since most movies and videos would be stored in some compressed format, it is necessary to run the algorithms directly on an MPEG, JPEG, or other compression format film without prior decompression. It has been shown that taking the I-frames of a MPEG movie is similar to grabbing video at about 2-3 frames per second, which is similar to Szummer's test footage. A modified algorithm may be able to analyze and parse the compressed format and generate the appropriate key frames.

## 6.2 Scene Parsing

The scene parsing algorithm should be integrated with the video stream, so that the actual frame of each scene would be outputted as a key frame. This is easily accomplished by synchronizing the frame rate of the video with the sampling rate of the audio. Whenever the audio parser finds a cut, the corresponding video frame could be outputted and stored. There are many digitizing packages (such as the SunVideo package which I used) which can digitize video and audio simultaneously. Another possibility is mentioned by Pincever: the audio is analyzed and when a cut is detected, the system will spot digitize and store the necessary frame. This innovation would save us the trouble of digitizing all the footage for the scene parser, and thus speed up the input process.

The algorithms should be tested on more audio/video data (feature films, news broadcasts, etc.) and this data should be digitized at a higher rate. This will give a better picture as to what factors affect scene detection by audio processing. Furthermore, other algorithms should be tested; perhaps an integration of the video

algorithms and the audio algorithms would result in a better scene parser.

## 6.3 General

Real-time parsing would increase the functionality of both the shot parser and the scene parser tremendously. In the case of the shot parser, we would need a way to digitize video quickly and run it through the parser. This could have the advantage of not needing large amounts of disk space to store the digitized video, since we would only need to store the key frames. In the case of the audio parser, the real-time parsing could be implemented by running the audio output directly into a digitizer board. Furthermore, specialized hardware and software could be utilized which would perform time-domain and frequency-domain analysis of the signal while the tape is being played.

Spectral signatures of the sounds could be taken and stored as a template for future reference. The audio could then be digitized and checked against this template. By using pattern matching techniques (such as those used in speech recognition), we could identify various sounds on the audio track. This would allow for the user to not only recognize the different sounds encountered, but some of the content as well. The system could automatically create a text log of the footage which could be used to implement a search function through the database (or through the raw video in real-time) to find a specific item [Pincever].

Similarly, video templates could be created. As mentioned previously, research into texture mapping has shown considerable success in recognizing such things as "grass" and "water." [Szummer2]. These video templates could be used in much the same manner as the audio templates, and provide for an improved searching and indexing system.

A good user interface needs to be developed for browsing through the shots and the scenes. Since the algorithms can also output the frame number of the key frames, these can be used to synchronize the shot frames and the scene frames with the original video. Thus a browser can be developed with a simple interface and high functionality. A graphic designer should be consulted when developing this interface.

A more general issue which should be considered is that of integrating the shot/scene parsing algorithms into larger projects, namely the digital library and the development of a multimedia database for WGBH. Although it seems fairly obvious that the shot/scene parser could provide greater functionality in browsing videos in the library channel and in organizing WGBH's videos, the specific methods of doing this should be examined. For example, how would we design the browsing interface for shots and scenes? What type of storage and access mechanisms are required? These and other technology oriented questions must be answered. But we must also examine the social issues. Will users find it beneficial to search through shots and scenes? If yes, what is the best method of using shot/scene parsing for browsing? It must be determined how to use the shot/scene technology to improve the user's accessing abilities, whether that user is a particular corporation or a teenager playing on the Internet.



# Chapter 7

## Conclusions

In a future dominated by many different sources and types of information, we need to find ways to organize this information and shape it into useful material. Without efficient cataloging mechanisms and an effective browser, all this information will simply exist as groups of bits floating in a digital sea.

Shot and scene parsing provide a way to store and organize video data into forms which can be used by many people. Although the original focus of this project was to develop a cataloging system for WGBH, the algorithms developed can be generally applied. Parsing can be used to create video cross references in a digital library. For digital video programs, it can aid in creating virtual links across existing shots and scenes in a digital library. Perhaps parsing can be used to aid in video swapping across the Internet by identifying common video clips and increasing the accessibility of these clips.

Through the development of new media – digital libraries, interactive television, etc. – there are many opportunities for understanding how the human mind recognizes shots and scenes. Parsing can help further our understanding of how ideas and concepts evolve into storylines, and how we can organize these ideas and concepts for people to use and appreciate. Used in reverse, a parsing algorithm combined with artificial intelligence systems could be used to generate shots and scenes. Such a system could give an editor advice on possible shots to use and different ways to organize scenes and programs.

By integrating speech recognition, object recognition, and other such advanced technologies into the shot/scene parsing system, we can create very advanced “intelligent systems.” It may be possible to have a computer do full transcriptions of any audio/video program, with shot and scene changes. The program could be logged, and the system could automatically create cross references with related programs. The system may be able to provide a text, video, and audio analysis of the program.

There is a wealth of information encoded into the audio track, and even more information encoded in the video itself. New ideas, new analytical techniques, and new computation systems are necessary to use this information fully. I sincerely hope that the ideas and concepts contained herein will be continue to be researched. Some people may believe that research for research’s sake is good enough, but I hope that these ideas will have real benefit in the world, and that someday the general public will see the benefits of audio/video parsing in their daily lives.

# Appendix A – Shot Parsing Code

```

README           Thu May 18 16:56:08 1995           1
=====
SCENE CUT DETECTION PROGRAM
Author   : Martin Szummer, szummer@media.mit.edu, MIT Media Lab, July 1994.
Supervisor: Prof. Rosalind Picard.

Scene cuts occur when a movie camera stops recording, changes viewing angle,
and starts recording again. This program automatically detects scene cuts.

(Modified by Allen Shu, MIT Media Lab, May 1995)
=====
cut-detect -xdim int -ydim int
[-n number]
[-i videoinputfile]
[-a alg (0)]
[-t thresh threshold (1.0)]
[-pos xstart xend ystart yend (0 xdim-1 0 ydim-1)]
[-color R G B (255 255 255)]
[-framesno framesnofile (output)] [-scores score_file (output)]
[-pattern (0)] [-mosaic|single]
[-output_cuts_only]

The cut-detect program detects scene cuts in a sequence of images.
The program reads the image sequence from stdin (use pipes) or from a
file specified with the -i flag.

OUTPUT:
The program writes a stream of images to stdout in which scene cuts
are indicated by inserting blank frames. Use a separate display
program that can read from stdin to view the result. If you don't
want use such a program to view the results, redirect stdout to
/dev/null, as you can still get a file with frame number of scene
cuts.

OPTIONS
-xdim int
Specify the dimensions of the image sequence.
-ychannels
Specify the number of color channels in the image sequence.
-i video_input_file
Specifies the file containing the image sequence to be analyzed.
If none is specified, stdin is used.

File format:
The program expects a raw data file, consisting of unsigned bytes.
Color information is expected to be non-interleaved, i.e. first
the entire channel 1, then channel 2 and 3, for every frame.
-a number
Specifies what algorithms should be used to detect scene cuts.
-a 0 Color histogram
-a 1 Chi-squared color histogram
-a 2 Robust statistic color histogram
-a 3 Smoothing of image, followed by color histogram.
The algorithms 2 and 3 usually perform the best.
-thresh floating-point-number
Multiplies the default threshold by this scalefactor. Default is 1.0.
=====
If the algorithm is reporting too many scene changes (false positives), set
this scalefactor greater than 1.0. If the algorithm is missing scene changes
(misses), set the scalefactor to less than 1.0.
-pos xstart xend ystart yend
Specifies what portion of the input image should be used to detect
scene cuts. Usually the whole image should be used (default).
However, processing is faster if a smaller portion of the image is used.
-color R G B
When a scene cut is detected, a uniform color frame is inserted in the
output image stream. The R G B values specify the color of this
frame.
-output_cuts_only
By default, the program writes the image sequence to stdout with scene
cuts marked by inserting white frames. Alternatively, if the
-output_cuts_only flag is specified, only the first frame after a scene cut will be output.
This is useful when you want to extract keyframes from a movie.
-pattern
Specifies the pattern of pixels used to detect scene cuts.
-pattern 0 use whole image
-pattern 1 use odd horizontal lines only
-pattern 2 use even horizontal lines only
-pattern 3 use odd points in x and y directions
X-X-
----
X-X-
-mosaic
Displays keyframes in separate windows. The windows are displayed using a
mosaic pattern. The windows must exist in the current directory.
The options -mosaic and -mosaic-single are mutually exclusive.
-mosaic-single
Displays keyframes in one window.
The options -mosaic and -mosaic-single are mutually exclusive.
-framesno filename
Name of file to which frame numbers with scene cuts are written.
Output is in ASCII format.
-scores filename
Name of file to which the scores describing the histogram differences
between frames will be written. Output is in ASCII format.
Examples:
# Do only cut detection
cut-detect -i /mas/vision/demos/ros/cut-detect/movie/boston_artery -xdim 160 -ydim 120 -
n 3 -a 3 -thresh 0.8 > /dev/null

# Do cut-detection, and view result using John Wang's xyt-viewer
cut-detect -i /mas/vision/demos/ros/cut-detect/movie/boston_artery -xdim 160 -ydim 120 -
n 3 -a 3 -thresh 0.8 | xyt-viewer -xdim 160 -ydim 120 -n 3 -xcut 158 -ycut 119 -rdim 500
-rydim 500 -space 200 -yspace 200
=====

```

```

cut-detect.c          Thu May 4 22:17:49 1995          1

/* Sheader: /v/users/sumner/program/cut-detect/src/RCS/cut-detect.c,v 1.6 1995/03/05 01:22
   :26 sumner Exp sumner $ */
/* *****
   Copyright 1995 by the Massachusetts Institute of Technology. All
   Rights reserved.
   Developed by Martin Sumner at the Media Laboratory, MIT, Cambridge,
   Massachusetts, with support from AT&T, PLG, Hewlett-Packard, and NEC.
   (modified by Allen Shu, MIT Media Lab, 5/95)

For use by MIT Media Lab students and staff.
This distribution is approved by Nicholas Negroponte, Director of
the Media Laboratory, MIT.

Permission to use, copy, or modify this software and its documentation
for educational and research purposes only and without fee is hereby
granted, provided that this copyright notice and the original
documentation are preserved in all copies of this software. If individual
files are separated from this distribution directory structure, this
copyright notice must be included. For any other uses of this software,
in original or modified form, including but not limited to distribution
in whole or in part, specific prior permission must be obtained from
MIT. These programs shall not be used, rewritten, or adapted as the
basis of a commercial software or hardware product without first
obtaining appropriate licenses from MIT. MIT makes no representations
about the suitability of this software for any purpose. It is provided
"as is" without express or implied warranty.
*****
/* cut-detect.c
   Purpose: Detects scene cuts in a movie.
   Sample usage: (using John Mings video-grab and xyt-viwer)
   source -jyewang/bin/common/DYNAMICrc
   Cut-Detecting from file, with mosaic
   cut-detect -i /mas/vision/demo/roz/cut-detect/movie/boston_artery -xdim 160 -ydim 120
   -n 3 -a 2 -frameno -/prog/cut-detect/results/scene_changes -scores -/prog/cut-detect/res
   ults/scores -mosaic -output_cuts_only | xyt-viewer -xdim 160 -ydim 120 -n 3 -xcut 158 -y
   cut 119 -rdim 500 -rydim 500 -xspace 200 -yspace 200
   Cut-detecting from file:
   cut-detect -i /mas/vision/demo/roz/cut-detect/movie/boston_artery -xdim 160 -ydim 120
   -n 3 -a 2 -frameno -/prog/cut-detect/results/scene_changes -scores -/prog/cut-detect/res
   ults/scores | xyt-viewer -xdim 160 -ydim 120 -n 3 -xcut 158 -ycut 119 -rdim 500 -rydim
   500 -xspace 200 -yspace 200
   Grabbing images (including crop):
   video-grab -xdim 164 -ydim 124 | select-image -xdim 164 -ydim 124 -n 3 -xs 2 -ys 2 -cmd
   im 160 -cyclic 120 -end 100000000 > /tmp/sumner/movie
   Grabbing and detecting:
   video-grab -xdim 164 -ydim 124 | select-image -xdim 164 -ydim 124 -n 3 -xs 2 -ys 2 -cmd
   im 160 -cyclic 120 -end 100000000 | cut-detect -xdim 160 -ydim 120 -n 3 | xyt-viewer -xd
   im 160 -ydim 120 -n 3 -xcut 158 -ycut 119 -rdim 500 -rydim 500 -xspace 200 -yspace 200
   Comparing:
   diff -/prog/cut-detect/results/actual_changes -/prog/cut-detect/results/scene_changes
diff -/prog/cut-detect/results/actual_changes -/prog/cut-detect/results/
scene_changes | grep -c ^$
$ misses
$ false positives
$ displaying subsequences of images
blit -i -sumner/movie/boston_artery/movie -xdim 160 -ydim 120 -start 33 -end 45 -c -fps
3

Color spaces
- Try different color spaces* What pixels? How many pixels?
- Over frames: 1st derivative, zero crossings of 2nd derivative
Efficiency
- Initiate the computation of the histogram
- Command line args cluster
- Better monitoring
- Place standard cluster files in right location
What pixels? How many pixels?
Give row start, row end, column start, column end --> can make arbitrary
cross.
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "image_fixed.h"
#include "image_aa.h"
#include "histogram_aa.h"
#include "util_aa.h"
#include "dat.h"

typedef struct {
int xdim;
int ydim;
int nchannels;
int aig;
double threshold;
int x_start, x_end, y_start, y_end;
int hist_img_area;
char video_infile_name[255];
char frameno_file_name[255];
char scorefile_name[255];
FILE *video_infile; /* video input file */
FILE *frameno_file; /* frame numbers of cuts */
FILE *scorefile; /* scores of all frames */
int mosaic;
int mosaic_single;
int pattern;
int output_cuts_only;
} Command_Options;
Command_Options options;

int frame_no = 1; /* Current frame number */

int int_compare(const void *a, const void *b)
/* effects: returns 1 if a > b, 0 if a == b, -1 if a < b. Used by qsort. */

```



```

--frames vs -scores %aln".
options.xdim, options.ydim, options.nchannels, options.als,
options.threshold,
options.r, options.g, options.b,
options.x_start, options.x_end, options.y_start, options.y_end,
options.video_infile_name,
options.filename_name, options.scorefile_name);
)

int quantise_color(unsigned char r, unsigned char g, unsigned char b)
{
    return (((unsigned char) (r & 0xc0)) >> 2 | ((unsigned char) (g & 0xc0)) >> 4 | ((unsigned
    ned char) (b & 0xc0)) >> 6);
}

int quantise_bv(unsigned char g)
{
    return g >> 2;
}

/* 0
Histogram* histogram_col(const image *image, int dummy1, int dummy2,
/* effects: returns the color histogram of the upper left edge
and lower right portion of the image (exactly like eddie) */
{
    unsigned char *img = image->data;
    int r = 0;
    int g = image->xdim * image->ydim;
    int b = 2 * image->xdim * image->ydim;
    Histogram *h = Histogram_create(64);
    /* scan along first row */
    for (int i = 0; i < image->xdim; i++) {
        Histogram_inc(b, quantise_color(img[i], img[g], img[b]));
    }
    /* scan along first column */
    r = image->xdim;
    g = image->xdim * image->xdim + image->ydim;
    b = image->xdim * 2 * image->xdim + image->ydim;
    while (b < image->xdim * image->ydim) {
        Histogram_inc(b, quantise_color(img[r], img[g], img[b]));
        r += image->xdim, g += image->xdim, b += image->xdim;
    }
    return h;
}
#endif

Histogram* histogram_col(const image *image,
int x_start, int x_end, int y_start, int y_end)
/* requires: x_start <= x_end, y_start <= y_end
effects: returns the histogram of a square portion of the image,
bounded by the box (x_start, y_start) up to including
(x_end, y_end)
*/
{
    int y;
    unsigned char *img = image->data;
    int img_xdim = image->xdim;
    int img_ydim = image->ydim;
    int y_off_end = y_end - image->xdim;
    int y_increment = image->xdim;
    int y_second_line = 0;
    int r_increment = 1;
    Histogram *h = Histogram_create(64);
    assert(x_start <= x_end && y_start <= y_end);
    if (options.pattern != 0)
        y_increment = 2 * image->xdim;
    if (options.pattern == 2)
        y_second_line = image->xdim;
    if (options.pattern == 3)
        r_increment = 2;
    /* scan along x */
    for (y = (y_start - image->xdim) + y_second_line; y <= y_off_end; y += y_increment) {
        int r = x_start + y;
        int x_end = x_end + y;
        int g = r + 1;
        int b = r + 2 * image->xdim;
        for (r = x_end; r <= r_increment, g <= g_increment, b <= b_increment)
            Histogram_inc(b, quantise_color(img[r], img[g], img[b]));
        return h;
    }
}

Histogram* histogram_bv(const image *image,
int x_start, int x_end, int y_start, int y_end)
/* requires: x_start <= x_end, y_start <= y_end
effects: returns the histogram of a square portion of the image,
bounded by the box (x_start, y_start) up to including
(x_end, y_end)
*/
{
    int y;
    unsigned char *img = image->data;
    int y_off_end = y_end - image->xdim;
    int y_increment = image->xdim;
    int y_second_line = 0;
    int r_increment = 1;
    Histogram *h = Histogram_create(64);
    assert(x_start <= x_end && y_start <= y_end);
    if (options.pattern != 0)
        y_increment = 2 * image->xdim;
    if (options.pattern == 2)
        y_second_line = image->xdim;
    if (options.pattern == 3)
        r_increment = 2;
    /* scan along x */
    for (y = (y_start - image->xdim) + y_second_line; y <= y_off_end; y += y_increment) {
        int r = x_start + y;
        int x_end = x_end + y;
        for (r <= x_end; r <= r_increment)
            Histogram_inc(b, quantise_bv(img[r]));
    }
}

```

```

    return h;
}

int scene_detect_robust(Queue *q, int *difference)
{
    const int threshold_robust = (double) options.hist_img_area / 18526 * 5490
        /* try 148 - optimal for our movie
           570 - optimal for I2-158x12-119
           */
        ;
    int hist_diff1, hist_diff2, hist_diff_diff;
    Image *image1 = Queue_get(q, 1);
    Histogram *h1 = histogram_col(image1, options.x_start, options.y_start, options.y_end);
    Image *image2 = Queue_get(q, 2);
    Histogram *h2 = histogram_col(image2, options.x_start, options.y_start, options.y_end);
    Image *image3 = Queue_get(q, 3);
    Histogram *h3 = histogram_col(image3, options.x_start, options.y_start, options.y_end);
    hist_diff1 = Histogram_diff(h1, h2);
    hist_diff2 = Histogram_diff(h2, h3);
    hist_diff_diff = hist_diff1 - hist_diff2;
    Histogram_destroy(h1);
    Histogram_destroy(h2);
    Histogram_destroy(h3);
    *difference = MAX(0, hist_diff_diff);
    if (*difference > threshold_robust) {
        fprintf(stderr, "\nScene change %td Frame no: %td\n",
            *difference, frame_no); /* DEBUG */
        return 1;
    }
    else
        fprintf(stderr, "%td", *difference); /* DEBUG */
    return 0;
}

int scene_detect_chi_robust(Queue *q, int *difference)
{
    /* Effects: performs scene detection using the Chi-squared test comparison
       described in A. Nagasaki & Y. Tanaka in "Automatic Video Indexing
       and Full-Video Search for Object Appearance",
       In Visual Database Systems II, 1992
       Divides image into 4x4 subsections and picks out the 8 largest
       values. Thresholds the sum of the 8 largest values.
       */
    /* threshold 3173 optimal for full 140x120 image */
    const int threshold_chi = (double) options.hist_img_area / 19200.0 * 3173.0
        * options.threshold;
    #define no_subblocks_x 4
    #define no_subblocks_y 4
    int hist_diff1, hist_diff2, hist_diff_diff;
    int i;
    int x_blk, y_blk;
    /* Divide image into 4x4
       subblocks. Loop over
       options.x_start)/4
       options.y_start)/4
       Loop over this doing the comparison and insert into array.
       Then sort the array.
       */
    int sz_block_x = options.xdim / no_subblocks_x;
    int sz_block_y = options.ydim / no_subblocks_y;
    Image *image1 = Queue_get(q, 1);
    Image *image2 = Queue_get(q, 2);
    Image *image3 = Queue_get(q, 3);
}

```

```

for (x_bik = 0; x_bik < no_subblocks_x; x_bik++) {
  for (y_bik = 0; y_bik < no_subblocks_y; y_bik++) {
    int x = x_bik * sz_block_x;
    int y = y_bik * sz_block_y;

    Histogram *h1 = Histogram_col(image1, x, x+sz_block_x, y, y+sz_block_y);
    Histogram *h2 = Histogram_col(image2, x, x+sz_block_x, y, y+sz_block_y);
    Histogram *h3 = Histogram_col(image3, x, x+sz_block_x, y, y+sz_block_y);

    hist_diff1 = Histogram_chi_diff(h1, h2);
    hist_diff2 = Histogram_chi_diff(h2, h3);

    hist_diff = hist_diff1 - hist_diff2;
    Histogram_destroy(h1);
    Histogram_destroy(h2);
    Histogram_destroy(h3);

    diff_subblock[x_bik*no_subblocks_y+y_bik] = MAX(0, hist_diff_diff);
  }
}

qsort(diff_subblock, no_subblocks_x * no_subblocks_y, sizeof(int),
      int_compare);

/* Select only the half of the subblocks, the ones with low priority */
difference = 0;
for (i = 0; i < (no_subblocks_x*no_subblocks_y)/2; i++) {
  *difference += diff_subblock[i];
}

if (*difference > threshold_chi) { /*4d Frame no: %d\n",
  printf(stderr, "\nScene change :%d Frame no: %d\n",
        return 1;
  }
  else
    printf(stderr, "%5d ", *difference); /* DEBUG */
  return 0;
}

int scene_detect_stealer(Queue *q, int *difference)
/* Effects: performs scene detection using the same algorithm as the
  scene stealer hardware. Use every seventh pixel on every line,
  which has previously been severely subsampled.
  Use only gray-level value of pixels to compute a histogram of
  64 gray-levels. No thresholding. Compute the difference of the two frames,
  the difference, no of corresponding histogram buckets.
  If the difference shows a spike between two frames, claim
  there is a histogram, however, if there is a step edge, this
  is most likely due to motion.
*/
const int threshold_stealer = (double) options_hist_img_area / 17433.0 * 6650.0 * opti
one threshold; /* 6650 for [2.158]x[2.118] */
Histogram *h1, *h2, *h3;
int hist_diff1, hist_diff2, hist_diff_diff;
static int first_execution = 1;
static Image *image1bw = NULL;
static Image *image2bw = NULL;
static Image *image3bw = NULL;
static Image *image_tmp = NULL;

```

```

image *image1 = Queue_get(q, 1);
image *image2 = Queue_get(q, 2);
image *image3 = Queue_get(q, 3);

if (first_execution) {
  image1bw = image_create(image1->xdim, image1->ydim, 1);
  image2bw = image_create(image2->xdim, image2->ydim, 1);
  image3bw = image_create(image3->xdim, image3->ydim, 1);
  image_tmp = image_create(image1->xdim, image1->ydim, 1);
  first_execution = 0;
}

image_rgb2bw(image1, image1bw);
image_rgb2bw(image2, image2bw);
image_rgb2bw(image3, image3bw);
image_smooth3x3(image1bw, image_tmp);
image_smooth3x3(image2bw, image_tmp);
image_smooth3x3(image3bw, image_tmp);
image_smooth3x3(image1bw, image2bw);
image_smooth3x3(image2bw, image3bw);
image_smooth3x3(image3bw, image1bw);
image_smooth3x3(image1bw, image2bw);
image_smooth3x3(image2bw, image3bw);
image_smooth3x3(image3bw, image1bw);
hist_diff1 = Histogram_diff(h1, h2);
hist_diff2 = Histogram_diff(h2, h3);
hist_diff_diff = hist_diff1 - hist_diff2;
Histogram_destroy(h1);
Histogram_destroy(h2);
Histogram_destroy(h3);

/* This is uses 2nd derivative instead of scene stealer algorithm */
difference = hist_diff_diff;
if (difference > threshold_stealer) {
  printf(stderr, "\nScene change :%d Frame no: %d\n",
        return 1;
  }
  else
    printf(stderr, "%5d ", *difference); /* DEBUG */
  return 0;
}

int scene_detect(Queue *q, int i, int *value)
{
  switch(i) {
  case 0:
    return scene_detect_ellipse(q, value);
  case 1:
    return scene_detect_chi(q, value);
  case 2:
    return scene_detect_chi_robust(q, value);
  case 3:
    return scene_detect_stealer(q, value);
  }
}

```



```

default:
    printf(stderr, "%s: scene_detect: No such scene detection algorithm\n",
           EXIT_FAILURE);
    return 0;
}

void notify_scene_detect(const Image *image, int cut, int score)
/* effects: Writes score to file options.scorefile.
   If cut!=0 (image represents a cut), outputs monochromatic image.
*/
{
    if (options.scorefile != NULL) {
        printf(options.scorefile, "%d\n", score);
        fflush(options.scorefile);
    }

    if (cut) {
        if (options.output_cuts_only) {
            image_write(image, stdout);
        }
        else {
            static Image *blank = NULL; /*!!! Memory leak; blank is never freed*/
            if (blank != NULL &&
                (image->xdim != blank->xdim || image->ydim != blank->ydim
                 || image->nchannels != blank->nchannels)) {
                Image_Delete(blank);
                blank = NULL;
            }
            if (blank == NULL) {
                blank = image_create(image->xdim, image->ydim, image->nchannels);
                image_fill(blank, options.r, options.g, options.b);
                image_write(blank, stdout);
            }
            if (options.framesofile != NULL) {
                printf(options.framesofile, "%d\n", frame_no);
                fflush(options.framesofile);
            }
        }
        /* Display mosaic of scene cuts */
        if (options.mosaic || options.mosaic_single)
        {
            static FILE *pipe;
            static firftime = 1;

            #define START_X 500
            #define START_Y 100
            #define END_X 1200
            #define END_Y 1000
            static int x = START_X;
            static int y = START_Y;

            #if 0
            /* Allocate pipes */
            #define NWINDOWS 16
            static int npipes = NWINDOWS; /* #keyframe windows to be displayed */
            static cur_window;
            if (firftime) {
                cur_window = 0;
            }
            #endif

            if (pipes[cur_window] == 0 && npipes > cur_window) {
                char rle_path = "/usr/vision/bin/SPACRPPipe";
                char *command = "setenv COLORTERM 'n 3 -h %d -w %d -H | "
                    + "xriedip -w | bcgetall -m -j -s -g 2.0 -t keyframe -s %d";
                char str[1000];
                sprintf(str, rle_path, rle_path, image->ydim, image->xdim,
                    pipes[cur_window] + 1);
                pipes[cur_window] = popen(str, "w");
                if (pipes[cur_window] == NULL) {
                    printf(stderr, "notify_scene_detect: Pipe could not be opened\n");
                    npipes = cur_window; /* Use only npipes, not NWINDOWS pipes */
                    cur_window = 0;
                }
                x += image->xdim + 10;
                if (x >= END_X)
                    y += image->ydim + 10;
                if (y >= END_Y) {
                    y = START_Y;
                }
            }

            fwrite(image->data, image->nchannels, image->xdim + image->ydim,
                pipes[cur_window]);
            fflush(pipes[cur_window]); /* '\c', '\0F' */
            cur_window = (cur_window + 1) % npipes;
        }
        else
        {
            if (firftime) {
                char str[1000];
                sprintf(str, "rleshov-m %d %d %s %d %d", image->ydim, image->xdim, "keyframe", x,
                    y);
                pipe = popen(str, "w");
                if (options.mosaic_single) {
                    firftime = 0;
                }
                else {
                    x += image->xdim + 10;
                    if (x >= END_X) {
                        x = START_X;
                        y += image->ydim + 10;
                        if (y >= END_Y) {
                            y = START_Y;
                        }
                    }
                }
            }
            if (pipe) {
                fwrite(image->data, image->nchannels, image->xdim + image->ydim, pipe);
                fflush(pipe);
                pclose(pipe);
            }
            sendif
        }
    }
}

```

```

    }
}
void clean_up(Queue *images)
/* effects: free allocated memory in images,
   close files.
*/
{
    Queue_destroy(images);
    if (options.video_infile != NULL) fclose(options.video_infile);
    if (options.frameofile != NULL) fclose(options.frameofile);
    if (options.scorefile != NULL) fclose(options.scorefile);
}

int main(int argc, const char *const argv[])
{
    const int buf_length = 3; /* #images stored */
    int i;
    Queue *images;
    Image *cur_image;
    parse_command_line(argc, argv);
    images = Queue_create(buf_length+1);
    /* Fill buffer with buf_length images to enable a start of the analysis.
       Allocate all memory needed for the buffer. This memory will be recycled
    */
    for (i=0; i<buf_length; i++) {
        Image *image = Image_create(options.xdim, options.ydim, options.nchannels);
        Image_read(image, options.video_infile);
        Queue_enqueue(images, (void *)image);
        if (!options.output_cuts_only)
            Image_write(image, stdout);
        frame_no++;
    }
    cur_image = Image_create(options.xdim, options.ydim, options.nchannels);
    while (Image_read(cur_image, options.video_infile)) {
        int cut, score;
        Queue_enqueue(images, (void *)cur_image);
        cut = scene_detect(images, options.sig, score);
        notify_scene_detect(cur_image, cut, score);
        if (!options.output_cuts_only)
            Image_write(cur_image, stdout);
        /* recycle memory */
        cur_image = Queue_dequeue(images);
        frame_no++;
    }
    clean_up(images);
    return EXIT_SUCCESS;
}

```

# Appendix B – Scene Parsing Code

```

README2      Thu May 18 16:46:18 1995      1
*****
SCENE DETECTION PROGRAM
Author: Allen Shu, allen@umit.edu, allen@umedia.mit.edu,
MIT Media Lab, May 1995
Supervisor: Prof. Giordanna Devenport, Prof. Michael Hawley

A scene is a unit of action or substance within a video, often composed
of several shots and sequences. This program uses the audio track of a
film to automatically detect scenes.
*****
scene-detect
-i audio_inputfile
[-o audio_outputfile]
[-t thresh (1)]
[-a (0)]
[-s size (8000)]
[-sps (8000)]
[-start (0)]
[-end (length of file)]
[-frameno frameinfo (output)]
[-scores score_file (output)]

The scene-detect program detects scene transitions using an audio file,
as specified with the -i flag.

OUTPUT:

OPTIONS
-i audio_inputfile      Specify the audio file to be analysed.
-o audio_outputfile    Specify the output audio file. NOT IMPLEMENTED.
-thresh floating-point-number
                        Specify the threshold to be used in the detection. Each algorithm
                        will have a different 'best' threshold value.
-a number
                        Specifies what algorithm should be used in the detection program.
-a 0 Average power, percentage comparison
-a 1 Average power, difference of difference comparison
-a 2 Average amplitude (frequency), difference of difference comparison
                        Algorithm 2 usually works the best. Default is algorithm 0.
-size int
                        Specify the size of the bin to be used for the analysis.
                        Default is 8000 elements.
-sps int
                        Specify the rate at which the audio was captured.
                        Default is 8000 samples per second (8 kHz).
-start int
                        Specify the starting sample for the analysis.
                        Default is 0 (the beginning of the file)

```

```

-end int      Specify the ending sample for the analysis.
              Default is the end of the file.
-frameno filename
              Specify the name of the file to which the timecode of the scene
              transitions are to be written. Format of the ASCII file is:
              Minutes, Seconds
-scores filename
              Specify the name of the file to which the scores describing
              the power or spectral differences are written. Output is in
              ASCII format.
Example:
This is an example of what I used on the Sun Sparc 20 in the IC lab.
scene-detect -i /localdisk1/allenshu/nowaudio.au -thresh 2.15 -a 2 -size 8000 -sps 8000
-frameano /u/allenshu/mywork/audiostuff/results/test2times -scores /u/allenshu/mywork/audi
ostuff/results/test2scores

```

```

/* .....
 * scene-detect.c
 *
 * Scene parsing using audio
 * Allan Shu, 4/95
 * .....
 *
 * #include <stdio.h>
 * #include <stdlib.h>
 * #include <errno.h>
 * #include <string.h>
 * #include <math.h>
 * #include <fcntl.h>
 * #include <ctype.h>
 * #include <sys/types.h>
 * #include <sys/param.h>
 * #include <unistd.h>
 *
 * #ifdef __MOSAI
 * #include <sys/dir.h>
 * #include <sys/audioio.h>
 * #else
 * #include <dirent.h>
 * #include <sys/audioio.h>
 * #endif
 *
 * #include <sys/file.h>
 * #include <sys/filio.h>
 * #include <sys/stat.h>
 * #include <sys/signal.h>
 * #include <sys/ioctl.h>
 *
 * #include <unistd.h>
 *
 * #include <multimedia/libaudio.h>
 * #include <multimedia/audio_device.h>
 * #include <multimedia/audio_encode.h>
 *
 * #include <util_w.h>
 * #include <dat.h>
 *
 * #define PRINTF (void) printf
 * #define SPURRY (void) sprintf
 * #define SPCRT (void) strcpy
 *
 * #define DEMO (void) printf("%s: event %d\n", (P), event_id(E))
 * #define DEMOUP (arg) (void) printf arg
 * #define PRNORM (arg) printf(stderr, "%s(%d): \"%s\" (%s)\n",
 * prog, _LINE_, arg, sys_errlist(errno));
 *
 * #else
 * #define DEMO
 * #define DEMOUP (P) (void) printf("%s: event %d\n", (P), event_id(E))
 * #define DEMOUP (arg) (void) printf arg
 * #define PRNORM (arg) printf(stderr, "%s(%d): \"%s\" (%s)\n",
 * prog, _LINE_, arg, sys_errlist(errno));
 * #endif /* !DEMO */
 *
 * #define INFO_SIZE 256
 * #define NOTICE_FILE "/dev/notice"
 * #define AUDIO_DRV "/dev/audio"
 * #define AUDIO_CTLDEV "/dev/audiocctl"

```

```

Audio_hdr Devices_phdr;
struct sound_buffer {
    Audio_hdr
    char
    char
    char
    unsigned char
    unsigned
    int
    int
    struct {
        unsigned
        unsigned
        unsigned
    }
    struct {
        int cursor_pos;
        int start;
        int end;
        int io_position;
        int play;
    }
    struct {
        int cursor_pos;
        int start;
        int end;
        int io_position;
        int last;
        int display;
    }
} Buffer;

extern char *sys_errlist[];

typedef struct {
    int start;
    int end;
    int size;
    int esp;
    double threshold;
    char audio_infile_name[255];
    char frameofile_name[255];
    char scorefile_name[255];
    char audio_outfile_name[255];
    FILE *audio_infile; /* audio input file */
    FILE *frameofile; /* frame numbers of cuts */
    FILE *scorefile; /* scores of all frames */
    FILE *audio_outfile;
} CommandOptions;

CommandOptions options;
int counter = 0;
double time_val = 0;
double time_inc = 0;
int debugging = 1;

double current_min=0;
double current_avg=0;
double prev_diff=0;

#define pi 3.14159265358979323846
#define SHAP(a,b) temp=(a);(a)=(b);(b)=temp
/* functions */
void printarray(double *sound_array, int size, char *string);
void parse_command_line(int argc, const char *const argv[]);

```

```

void alloc_buffer(unsigned int size);
int soundfile_read(void);
int get_array(int size, double *sound_array);
int power(double x, int n);
void average(double data[], int n, double *pwr);
void fft(double *data, int n, int isgm);
void resfft(double *data, int n, int isgm);
int scene_detect_zero(double *array, double *array2,
                    int size, double *difference);
int scene_detect_one(double *array, double *array2,
                    int size, double *difference);
int scene_detect_two(double *array, double *array2,
                    int size, double *difference);
int scene_detect_dispatch(double *array, double *array2,
                    int i, int size, double *value);
void set_initial_values(double *sound_array, int size);
void notify_scene_detect(int cut, double score);
void cleanup(void);
/*****
 * Test function to print arrays
 */
void printarray(double *sound_array, int size, char *string)
{
    int i=0;
    if (debugging) {
        printf("%s\n", string);
        for (i=0; i<size; i++)
            printf("%f ", sound_array[i]);
    }
    else printf("Debugging off: %s\n", string);
}

void parse_command_line(int argc, const char *const argv[])
/* effects: analyses command line arguments and writes their values
*/
{
    const char usage[] = "%s\n"
        "-i audio_inputfile\n"
        "-o audio_outputfile\n"
        "-t thresh (f)\n"
        "-a (f)\n"
        "-s (size (8000))\n"
        "-sp (8000)\n"
        "-start (f)\n"
        "-end (length of file)\n"
        "-f filename (output)\n"
        "-s score_file (output)\n";

    options.threshold = 1.0;
    options.start = 0;
    options.end = 0;
    options.alg = 0;
    options.size = 8000;
    options.sp = 8000;
    /* FIX SETTING OF START and END */
    strcpy(options.audio_infile_name, "");
    strcpy(options.audio_outfile_name, "");
    strcpy(options.frameofile_name, "");
    options.audio_infile = NULL;
    options.frameofile = NULL;
    options.scorefile = NULL;
    scenargv[argc, argv, usage,
        "-i %s",
        "-o %s",
        "-t thresh %f",
        "-a %d",
        "-s %d",
        "-sp %d",
        "-start %d | -end %d",
        "-f filename %s | -score %s",
        options.audio_infile_name,
        options.audio_outfile_name,
        options.alg,
        options.size,
        options.sp,
        options.start, options.end,
        options.frameofile_name, options.scorefile_name];

    if (strcmp(options.audio_infile_name, "") == 0) {
        printf(stderr, "parse_command_line: Error\n");
        printf(stderr, usage, argv[0]);
        exit(EXIT_FAILURE);
    }
    else
        if ((options.audio_infile = fopen(options.audio_infile_name, "rb")) == NULL) {
            printf(stderr, "Cannot read file '%s'. Exiting...\n",
                options.audio_infile_name);
            exit(EXIT_FAILURE);
        }
        if (strcmp(options.frameofile_name, "") != 0) {
            options.frameofile = fopen(options.frameofile_name, "w");
            if (options.frameofile == NULL) {
                printf(stderr, "parse_command_line: Error, could not open '%s'\n"
                    "for writing. Exit\n");
                exit(EXIT_FAILURE);
            }
            else printf(stderr, "Writing file %s\n", options.frameofile_name);
        }
        if (strcmp(options.scorefile_name, "") != 0) {
            options.scorefile = fopen(options.scorefile_name, "w");
            if (options.scorefile == NULL) {
                printf(stderr, "parse_command_line: Error, could not open '%s'\n"
                    "for writing. Exit\n");
                exit(EXIT_FAILURE);
            }
            else printf(stderr, "Writing file %s\n", options.scorefile_name);
        }
        if (options.size <= 0) {
            printf(stderr, "parse_command_line: Error, size must be greater than zero. Exiting.\n");
            exit(EXIT_FAILURE);
        }
}

```

```

)
time_inc = options.size/options.sps;
}

/* Allocate a buffer to hold data */
void alloc_buffer(unsigned int size)
{
    if (Buffer.data != NULL)
        (void) free((char *)Buffer.data);
    /* allocate a buffer, shrinking if request is too big */
    do {
        Buffer.data = (unsigned char *)malloc(size);
    } while ((Buffer.data == NULL) && (size = size - (size / 8)));
    Buffer.alloc_size = size;
}

/* Initialise the buffer */
void init_buffer(void)
{
    if (Buffer.data != NULL)
        (void) free((char *)Buffer.data);
    Buffer.data = NULL;
    Buffer.alloc_size = 0;
    Buffer.hdr.device_pbrdr;
    Buffer.hdr.data_size = 0;
}

/* Open the named sound file and read it into memory. Return TRUE if error. */
int soundfile_read(void)
{
    int fd;
    unsigned size;
    int valid;
    char msg[256];
    struct stat st;

    if (((fd = open(options.audio_infile_name, O_RDONLY)) < 0) || (fstat(fd, st) < 0) ||
        (ftruncate(fd, (st.st_size * options.audio_infile_name_sps) / options.audio_infile_rate) < 0))
        fprintf(stderr, "Can't read '%s'.\n", options.audio_infile_name, sys_errlist[errno]);
        exit(EXIT_FAILURE);
        return 0;
    }

    /* If the soundfile has a header, read it in and decode it.
    valid = (AUDIO_SUCCESS == audio_read_filehdr(fd, &buffer.hdr,
    Buffer.info, sizeof(Buffer.info)));
    if (valid)
        if (Buffer.hdr.data_size == AUDIO_UNKNOWN_SIZE) {
            /* Calculate the data size, if not already known */
            Buffer.hdr.data_size =
                st.st_size - lseek(fd, 0L, SEEK_CUR);
        }
    else {
        /* If no header, read the file raw and assume compatibility */
        Buffer.hdr = Device_pbrdr; /* use device configuration */
        (void) lseek(fd, 0L, L_SET); /* rewind file */
        Buffer.hdr.data_size = st.st_size - lseek(fd, 0L, SEEK_CUR);
        Buffer.info[0] = '\0';
    }
}

/* If active output, set draining flag so that play_service() won't
* try to read the obsolete buffer. file_update() will turn
* off the draining flag if output is still active.
*/

/* Release the old buffer and allocate a new one to hold the data */
size = Buffer.hdr.data_size;
alloc_buffer(size);

/* Read in as much data as possible and close the file. */
Buffer.hdr.data_size = read(fd,
(char *)Buffer.data, (int)Buffer.alloc_size);
(void) close(fd);

if (options.start != 0 && options.start <= Buffer.hdr.data_size)
    Buffer.play_start = options.start;
else
    Buffer.play_start = 0;

Buffer.play_io_position = Buffer.play_start;

if (options.end != 0 && options.end <= Buffer.hdr.data_size &&
options.end >= options.start)
    Buffer.play_end = options.end;
else
    Buffer.play_end = Buffer.hdr.data_size;

/* If we could not allocate or load the whole file, show msg */
if (size < Buffer.hdr.data_size)
    fprintf(stderr, "%s: %d seconds of data truncated from '%s'.\n",
audio_bytes_to_secs(Buffer.hdr.
(size - Buffer.hdr.data_size), options.audio_infile_name));
}

/* If file is not an audio file, display a warning */
if (!valid) {
    fprintf(stderr, "%s: is not a valid audio file.\n", options.audio_infile_name);
}
else if ((Buffer.hdr.encoding != AUDIO_ENCODING_ULAW) ||
(Buffer.hdr.bytes_per_unit != 1) ||
(Buffer.hdr.samples_per_unit != 1) ||
(Buffer.hdr.channels != 1) ||
(Buffer.hdr.encoding != 's'))
    fprintf(stderr, "%s: audio encoding cannot be played or displayed properly.\n",
options.audio_infile_name);
return 1;
}

/* This is the routine that is called from the main loop that writes
* sound to the device.
* It needs the following state data:
* - current start and end (calculate in proc)
* - current position in the buffer
* - int get_array(int size, double *sound_array)
* - int start;
* - int end;
* - int outcount;

```



```

if (Buffer.hdr.data_size == 0) {
    printf("No data in Buffer.\n");
    return (0);
}

cnt = Buffer.play_end - Buffer.play_start;
if (cnt == 0) {
    printf("No data in selected region.\n");
    return (0);
}

/* XXX - sanity checks for now */
if (Buffer.play_start >= Buffer.hdr.data_size) {
    fprintf(msg, "start position (%d) beyond EOF (%d).\n",
            Buffer.play_start, Buffer.hdr.data_size);
    return (-1);
}

if (cnt < 0) {
    fprintf(msg, "start position (%d) beyond end position (%d).\n",
            Buffer.play_start, Buffer.play_end);
    return (-1);
}

return (cnt);
}

/*
 * Open audio control device (/dev/audiocctl) and read its state.
 * This may be used for state get/set (eg. volume levels) without holding
 * the main audio device (/dev/audio) open.
 */
void audio_control_init(void)
{
    /* open audio control device */
    if ((Audioctl_fd = open(AUDIO_CTLDEV, 0_RDWR)) < 0) {
        perror(AUDIO_CTLDEV);
        Device_phdr.sample_rate = 8000;
        Device_phdr.channels = 1;
        Device_phdr.bytes_per_unit = 1;
        Device_phdr.encoding = AUDIO_ENCODING_ULAW;
        Device_rhdr = Device_phdr;
        Buffer_rhdr = Device_phdr;
    } else {
        /* tell the driver to send SIGPOLL on device state changes */
        if (ioctl(Audioctl_fd, I_SETSIG, S_MSO) < 0)
            perror("Could not issue I_SETSIG ioctl");

        /* Get the device play & record configuration */
        if (Audio_get_play_config(Audioctl_fd, &Device_phdr) !=
            AUDIO_SUCCESS) {
            perror("Could not get play configuration");
            Device_rhdr = Device_phdr;
        }
        if (Audio_get_record_config(Audioctl_fd, &Device_rhdr) !=
            AUDIO_SUCCESS) {
            perror("Could not get record configuration");
            Device_rhdr = Device_phdr;
        }
    }
}

AUDIO_INITINFO(audio_state);
}

endif
/*
 * *****
 * STAT ROUTINES
 */
/* power - function to raise x to the nth power; n > 0
 * from Kernighan & Ritchie
 */
int power(double x, int n)
{
    int i;
    float p;
    p = 1.0; /* force first time execution */
    for(i=1; i<n; i++)
        p = p * x;
    return(p);
}

/* aveper -- From Numerical Recipes in C
 * Given an array data[1...n], this routine returns its average power.
 */
void aveper(data, n, pwr)
{
    int i;
    double data[];
    double pwr;

    for (i=0; i<n; i++)
        pwr += data[i]*data[i];
    pwr /= n;
}

/* fft
 * replaces data by its discrete fourier transform if its sign is 1 or by its
 * idft if its sign is -1. data is a complex array of length nn stored as a
 * real array data[1:2*nn].
 * (from Numerical Recipes in C)
 */
void fft(data, nn, isign)
{
    double *data;
    int nn, isign;
    int n, mmax, m, j, istep, i, k;
    double wtemp, wr, wpr, wpi, wi, theta;
    double temp, tempi;
    n = nn << 1;
    j = 1;
    for (i=1; i<n; i+=2) {
        if (i > 1) {
            SNAP(data[i], data[i+1]);
            SNAP(data[i+1], data[i+1+1]);
        }
        m = n >> i;
}
}

```





scene-detect.c Thu May 18 16:46:16 1995

7

```
int scene_detect_one(double *array1, double *array2,
                    int size, double *difference)
{
    double this_diff, diff_diff;
    double prev1, prev2;
    int ret=0;
    double numsec=0;

    double thisthreshold = options.threshold;
    aveper(array1, size, &prev1);
    aveper(array2, size, &prev2);
    this_diff = fabs(prev1 - prev2);
    diff_diff = fabs(this_diff - prev_diff);
    printf("olddiff %f, newdiff %f\n", prev_diff, this_diff);

    *difference = MAX(0, diff_diff);
    if (*difference > thisthreshold) {
        modf(time_val, &numsec);
        fprintf(stderr, "Scene change: %f Number sec: %d\n",
                *difference, (int)numsec); /* for debugging */
        ret=1;
    }
    else {
        fprintf(stderr, "%5f\n", diff_diff);
        ret=0;
    }
    prev_diff = this_diff;
    return ret;
}

int scene_detect_two(double *array1, double *array2,
                    int size, double *difference)
{
    double this_diff, diff_diff;
    double spec1, spec2;
    double numsec=0;

    double thisthreshold = options.threshold;
    realfft(array1, (size/2), 1);
    realfft(array2, (size/2), 1);
    aveper(array1, size, &spec1);
    aveper(array2, size, &spec2);
    this_diff = fabs(spec1 - spec2);
    diff_diff = fabs(this_diff - prev_diff);
    printf("olddiff %f, newdiff %f\n", prev_diff, this_diff);

    *difference = MAX(0, diff_diff);
    if (*difference > thisthreshold) {
        modf(time_val, &numsec);
        fprintf(stderr, "Scene change: %f Number sec: %d\n",
                *difference, (int)numsec); /* for debugging */
        ret=1;
    }
    else {
        fprintf(stderr, "%5f\n", diff_diff);
        ret=0;
    }
    prev_diff = this_diff;
    return ret;
}

void set_initial_values(double *sound_array1, int size)
{
    double prev1;
    printarray(sound_array1, size, "set initial values");
    printf("calling aveper");
    aveper(sound_array1, size, &prev1);
    current_min = prev1;
    current_avg = prev1;
}

void notify_scene_detect(int cut, double score)
{
    double ip, fp;
    double quotient;
    double secondval;
    if (options.scorefile != NULL) {
        fprintf(options.scorefile, "%f\n", score);
        fflush(options.scorefile);
    }
    if (cut) {
        quotient = time_val/60;
        fp = modf(quotient, &ip);
        secondval = fp*60;
        if (options.frameofile != NULL) {
            fprintf(options.frameofile, "%d, %f\n", (int)ip, secondval);
            fflush(options.frameofile);
        }
    }
    void cleanup(void)
    {
        if (options.audio_infile != NULL) fclose(options.audio_infile);
        if (options.frameofile != NULL) fclose(options.frameofile);
        if (options.scorefile != NULL) fclose(options.scorefile);
        if (options.audio_outfile != NULL) fclose(options.audio_outfile);
    }
    int scene_detect_dispatch(double *array1, double *array2,
                              int i, int size, double *value)
    {
        switch(i) {
            case 0:
                return scene_detect_sero(array1, array2, size, value);
            case 1:
                return scene_detect_one(array1, array2, size, value);
            case 2:
                return scene_detect_two(array1, array2, size, value);
            default:
                ;
        }
    }
}

```

```

    fprintf(stderr, "scene_detect: No such algorithm\n");
    exit(EXIT_FAILURE);
}
return 0;
}

int main(int argc, const char *const argv[])
{
    int size;
    int cut=0;
    double score;
    int retval=1;
    double *sound_array1, *sound_array2;

    parse_command_line(argc, argv);
    size = (int) (double *) malloc(sizeof(double)*1);
    sound_array1 = (double *) malloc(sizeof(double)*1);
    sound_array2 = (double *) malloc(sizeof(double)*1);
    printf("initializing buffer\n");
    init_buffer();

    printf("reading soundfile\n");
    soundfile_read();

    printf("initializing\n");
    get_array(size, sound_array1);
    get_array(size, sound_array2);
    counter=4;
    time_val= (2*time_inc);

    /*
    printarray(sound_array1, size, "array1");
    printarray(sound_array2, size, "array2");
    */

    printf("setting initial values\n");
    set_initial_values(sound_array1, size);
    printf("begin while loop\n");
    while(retval) {
        if ((counter % 2) == 0)
            cut = scene_detect_dispatch(sound_array1, sound_array2,
                                       options.sig, size, score);
        else
            cut = scene_detect_dispatch(sound_array2, sound_array1, options.sig,
                                       size, score);

        notify_scene_detect(cut, score);
        if ((counter % 2) == 0) {
            if (cut)
                set_initial_values(sound_array2, size);
            retval=get_array(size, sound_array1);
        }
        else {
            if (cut)
                set_initial_values(sound_array1, size);
            retval=get_array(size, sound_array2);
        }
        counter++;
        time_val+=time_inc;
    }
}

cleanup();
free(sound_array1);
free(sound_array2);
return EXIT_SUCCESS;
}

```



# Bibliography

[Berkeley] University of California at Berkeley. Robert Wilensky, principal investigator. "The Environmental Electronic Library: A Prototype of a Scalable, Intelligent, Distributed Electronic Library." Also at <http://http.cs.berkeley.edu/~wilensky/proj-html/proj-html.html>

[DPAS] Davenport, G., Pincever, N., Aguierre Smith, T. *Cinematic Primitives for Multimedia*. IEEE Computer Graphics and Applications, June 1991.

[Elliot] Elliot, Edward. *Watch, Grab, Arrange, See - Thinking with Motion Images via Streams and Collages*. M.S. Thesis, Massachusetts Institute of Technology, February 1993.

[HJW1] Hampapapur, A., Jain, R., Weymouth T. *Digital Video Segmentation*. The Proceedings of the ACM conference on Multimedia 1994.

[HJW2] Hampapapur, A., Jain, R., Weymouth T. *Digital Video Indexing in Multimedia Systems*. Proceedings of the workshop on Multimedia Indexing and Reuse, AAAI 94.

[Houbart] Houbart, Gilbert. *Viewpoints on Demand: Tailoring the Presentation of Opinions in Video*. M.S. Thesis, Massachusetts Institute of Technology, September 1994.

[MacCarn] Personal contact with David MacCarn, WGBH.

[Massie] Personal contact with Alexis Massie, WGBH.

[MIT] Massachusetts Institute of Technology Media Lab. Principal investigators: Michael Hawley, Andrew Lippman, Nicholas Negroponte. "The Library Channel," proposal to the NSF.

[NT] Nagasaka, A., Tanaka Y., *Automatic Video Indexing and Full-Video Search for Object Appearances*. Visual Database Systems II, Ed. E. Knuth, L. Wegner, 1992.

[OT] Otsuji, K., Tonomura, K. *Projection-detecting filter for video cut detection*. Multimedia Systems 1:205-210, 1994.

[Pincever] Pincever, Natalio Carlos. *If you could see what I hear: Editing assistance through cinematic parsing*. M.S. Thesis, Massachusetts Institute of Technology, 1991.

[PPS] Pentland, A., Picard, R.W., Sclaroff, S. *Photobook: Tools for Content-Based Manipulation of Image Databases*. MIT Media Lab Perceptual Computing technical report 255, SPIE Conf. Storage and Retrieval of Image and Video Databases II, No. 2185, Feb 6-10, 1994. San Jose, CA.

[Sasnett] Sasnett, Russel. *Reconfigurable Video*. M.S. Thesis, Massachusetts Institute of Technology, February 1986.

[Stanford] Stanford University, "The Stanford Integrated Digital Library Project."  
Also at <http://www-diglib.stanford.edu>

[Sun] Sun Microsystems, *SunVideo User's Guide*, August 1994.

[Szummer] Szummer, Martin and Picard, Rosalind W. *Scene cut detection*. DRAFT – MIT Media Lab, Vision and Modeling. September 1994.

[Szummer2] Personal contact with Martin Szummer, MIT Media Lab.

[UCSB] University of California at Santa Barbara. Co-principal investigators: Jeff Dozier, Michael Goodchild, Oscar Ibarra, Sanjit Mitra, Terence Smith. "The Alexandria Project."

[UIUC] University of Illinois at Urbana-Champaign, "Building the Interspace: Digital Library Infrastructure for a University Engineering Community."  
<http://www.grainger.uiuc.edu/dli>

7103-29