LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TR-742

# AN INTERACTIVE APPROACH TO THE IDENTIFICATION AND EXTRACTION OF VISUAL EVENTS

William F. Stasior

February, 1998

This document has been made available free of charge via ftp
from the MIT Laboratory for Computer Science.

545 TECHNOLOGY SQUARE; CAMBRIDGE, MASSACHUSETTS 02139 (617) 253-5851

# An Interactive Approach to the Identification and Extraction of Visual Events

William F. Stasior

*Telemedia Networks and Systems Group*
*Laboratory for Computer Science*
*Massachusetts Institute of Technology*

## Abstract

This report describes an interactive approach to the computerized processing and interpretation of visual information. The objective is to facilitate the development of interactive applications that analyze and interpret video input. The approach is to transform video from raw sensory data into symbolic *media events* which can be incorporated into the design of event-driven programs.

The computational task is modeled as a sequence of representational transformations which are performed in stages on the data stream as it flows through a network of processing modules. The transformations expose the salient information in the image stream which is identified and transformed into *properties*. Properties are then grouped into sequences of *predicates* that are analyzed by *pattern* matching automata. Upon recognition, these automata signal a symbolic event to the event processing layer of the application.

The approach is realized by Sieve, a "System for Identifying and Extracting Visual Events." Sieve is a programming toolkit that provides a computational framework for building multimedia programs that respond to both user and media input. Sieve's functionality may be accessed by embedding the system into a custom application or by employing the higher level *VsGrep* application. VsGrep is a general purpose tool, built using the Sieve toolkit, that may be configured to take specified actions in response to matching specific patterns of video. As such, VsGrep may be used as a high level application builder which takes a concise specification and produces an interactive content analyzing application.

Finally, this report describes five prototype applications built using VsGrep that demonstrate the system's functionality and performance: the Room Monitor, the Whiteboard Recorder, the Computer Librarian, the Television Agent, and the Gesture Detector. These applications show that desktop processing of video information is both feasible and powerful.

©Massachusetts Institute of Technology 1998

This research was supported by the Advanced Research Projects Agency of the Department of Defense, monitored by the United States Air Force (AFSC, Rome Laboratory) under contract No. F30602-92-C-0019, and by a grant from Sun Microsystems.

# Acknowledgments

I'd like to thank my advisor, David Tennenhouse, both for inspiration and for the years of support, guidance, and patience.

I also greatly enjoyed working with both my readers, Eric Grimson and Tomás Lozano-Pérez. In addition to their insights, I am very appreciative of their time and availability.

The research group that I work with has been the best reason for spending the time to earn my degree. Chris Lindblad was a great resource who had a tremendous, positive impact on this work. Vanu Bose, David Wetherall, Mike Ismert, and Henry Houh have all been extremely helpful, and, incredibly fun to work with. John Guttag has been a great help in the final stages. I am also very grateful to a many, many others who have been a part of the Telemedia Networks and Systems Group – which has now become Software Devices and Systems.

Finally, I owe the most to my current and future family. My mom and dad provided me with everything they possibly could to support this endeavor, while my sister, Laura, remains my strongest advocate. At last, I do not know how I ever would have finished if it were not for my fiancé Jennifer providing comfort and support in these last two years.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

This dissertation describes an interactive approach to the computerized processing and interpretation of visual information. The objective is to enable a user armed with a multimedia workstation to build interactive applications that analyze and interpret video input.

The approach is realized by Sieve, a "System for Identifying and Extracting Visual Events." Sieve is a programming toolkit that provides a computational framework for building multimedia programs that respond to both user and media input.

### Processing capability advances multimedia

Multimedia has already extended the domain of computer data from that which was primarily numbers and text to that which includes images and media. Broadband networks coupled with the widespread deployment of video devices, such as cameras, television receivers, and video recorders, are allowing workers to rapidly access and produce visual information. In doing so, it raises the possibility of these media becoming a rich and universal source of information in the work place.

However, while computers are highly capable of examining and transforming text, they have remained largely incapable of penetrating media. Media remains an opaque data type which may be captured, stored, retrieved, and presented but can not be searched, manipulated, or analyzed.

This lack of processing capability, which would be a limitation on any source of data, is particularly severe for audio and video because they have a temporal component. The temporal dependence places demands on the consumer's attention and limits the amount of media which may be consumed. For example, consider the difference between photographs and video tape. Video is easy to produce and rich in content. Many consumers would prefer, however, to browse a set of photographs because they summarize an experience, freeing the viewer from the time extent of the event.

What is needed are tools that can summarize media information and allow the user to browse and consume this vast source of data more efficiently. One would like programs that can watch television for us and tell us when something of interest occurs. One would also like programs that can analyze the video from cameras in our environment to tell us when interesting events occur in the physical world.

### Computer vision provides a starting point for multimedia processing

Unfortunately, the significant advances in computer vision have as of yet done little towards advancing such goals. Building robust, working vision systems has proven to be exceedingly difficult. Most vision algorithms are computationally expensive, difficult to apply, and not robust. Vision systems typically function in a narrow task domain or not at all.

I believe that part of the problem is that vision applications try to be too smart relative to the task at hand. Using an analogy to text, one may argue that while considerable progress has been made in the area of natural language understanding, most users use much simpler search tools, such as the *grep* utility, to find and manipulate text. For example, in searching through one's email to find an old message about a seminar, one is likely to search for words related to the topic of the seminar, the speaker's name, or even the date to find the message for which he or she is looking.

The point is, human beings don't mind applying their own knowledge and intelligence to data manipulation tasks. What humans need are tools which allow them to leverage the computer's ability to perform repetitive tasks such as searching large quantities of data in well defined ways.

### Computer vision + interactivity is the key to unlocking information

Multimedia brings video to the computer desktop. In doing so, the user is brought closer to the data and the processing. This thesis aims to exploit this change by making it possible for the user to be tightly integrated with the extraction of information (figure 1-1) by utilizing traditional computer vision techniques in an interactive environment.

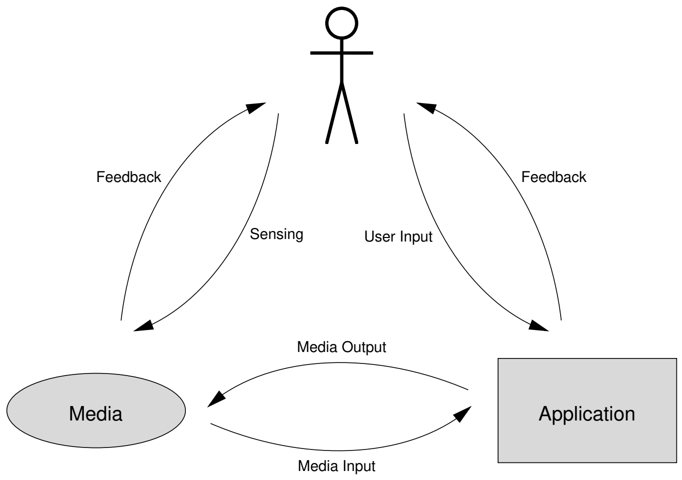



Figure 1-1: User media application loop

Interactive applications inherit the constraint that they must run reasonably fast. Hence, the objective of building *interactive* programs which analyze video may seem overwhelming. To the contrary, this thesis argues that combining interactivity with vision makes the vision problem tractable across a wide range of interesting applications.

The fundamental advantage to interactivity is that it brings a human into the loop. Three principle advantages of a supervised system over a fully autonomous one are:

- A user can supply parameters and feedback.

  Algorithms that process sensory data frequently depend upon a multitude of input parameters. Typically, these parameters are tuned in an ad hoc manner to maximize performance. While computer programs have considerable difficulty in judging their own performance, humans can provide feedback which may be used to adjust the system.

- A user can tolerate mistakes.

  It is considerably easier to prune the space of possibilities than to produce a definitive answer. While an autonomous system may be forced into making an uncertain decision, an interactive system can be useful to its user simply by reducing the amount of information that the user sees, allowing the operator to perform the final arbitration.

- A user can specialize the system.

  A human user is able to apply his or her knowledge about the problem in order to match a specific processing operation to a highly constrained task. For example, in searching a news telecast for a Red Sox baseball highlight, the user may know that Red Sox highlights are typically preceded by an anchor person and followed by a graphic depicting the score of the game. Given the right tools, the user may direct the computer to search the telecast for such occurrences rather than require the system to understand the concept of baseball.

## 1.1 Approach and Contributions

To meet the objectives outlined above, this thesis:

- Develops a framework for building *interactive* applications which *analyze* media content.

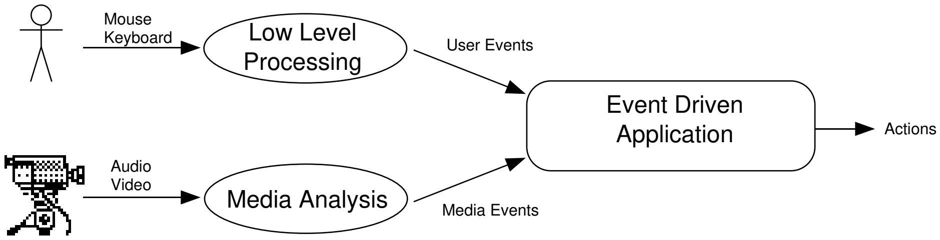



Figure 1-2: User and media events

  Interactive applications are those that respond in a timely manner to a human user. In the case of traditional interfaces, the user communicates with the computer through a keyboard or a mouse. Such input is processed at a low level into symbols and events which are processed at a higher level by the application.

  This thesis treats media input in an analogous manner. Media is transformed from raw sensory data into symbolic *media events* which the application processes along with higher level user events in event handling routines (figure 1-2).

- Develops an approach to analyzing video streams which is suitable for generating symbolic media events.

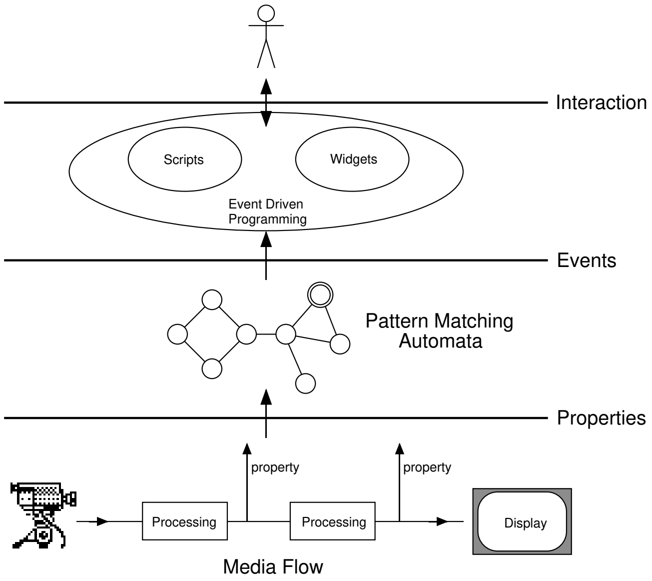



Figure 1-3: Video → properties → events → interaction

  The problem of generating symbolic events from unstructured video is posed as the problem of recognizing patterns in video data. This computational task, depicted in figure 1-3, is modeled as a sequence of representational transformations which are performed in stages on the data stream as it flows through a network of processing modules. The transformations expose the salient information in the image stream which is identified and transformed into *properties*.

Next, *patterns* of properties are recognized by automata which analyze the sequences of property measurements. Upon recognition, these automata signal a symbolic `event` to the event processing layer of the application.

- Adapts algorithms from computer vision so that they may be used in a flow driven media processing network to measure properties in video.

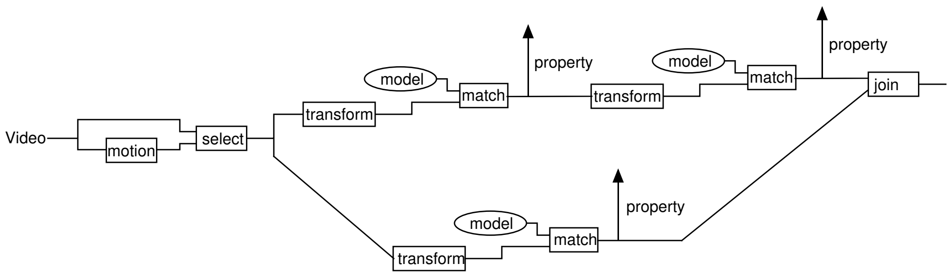



Figure 1-4: Extracting properties from video streams

The video analysis employed in this thesis focuses on two complementary areas from computer vision: matching and motion. Matching is concerned with determining whether two inputs are the same while motion is concerned with detecting and measuring change. Motion is used primarily to focus the attention of the system, telling it *when* and *where* to look. Matching is used to compare input data against models and templates so that a property may represent the degree to which an input matches a particular model. Matching is implemented in a modular fashion whereby representational transformations that expose the relevant information are performed prior to matching (figure 1-4).

- Adapts algorithms from computational automata theory that may be used to recognize patterns of properties.

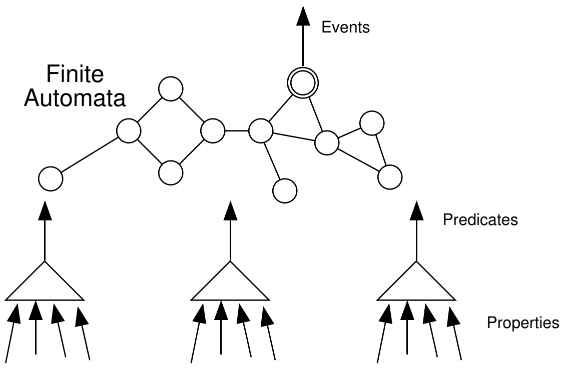



Figure 1-5: Matching patterns of properties

There are many viable approaches to the problem of detecting patterns in streams of property values. This thesis focuses on adapting well developed techniques used to recognize strings of symbols to the problem of recognizing sequences of properties. The concept of a *predicate* is introduced (illustrated in figure 1-5) to map properties to symbols so that regular patterns may be recognized using finite-automata.

- Implements *Sieve*, a programming system that realizes the analytical approach put forth in this thesis.

  Sieve is an extensible system for implementing interactive applications which manipulate and analyze media content.

- Implements *VsGrep*, a generic Sieve application that is the visual equivalent of "grep".


Figure 1-6: The VsGrep application

  VsGrep, shown in figure 1-6, may be used an interactive tool or as a high level application builder which takes as input a concise specification and produces as output an interactive content analyzing application.

- Demonstrates Sieve by using it to build applications which extend the domain of multimedia.

  This thesis presents five prototype applications implemented using Sieve: the Room Monitor, the Whiteboard Recorder, the Computer Librarian, the Television Agent, and the Gesture Detector. In addition to demonstrating Sieve's functionality, these applications demonstrate the possibilities brought about by the desktop processing of media information.

The Television Agent (figure 1-7) demonstrates the possibility of using computers to help us sift through the volumes of video data coming at us from television content providers. In particular, this application captures video clips of special events which the user has programmed the agent to watch for.


Figure 1-7: The Television Agent

The Room Monitor (figure 1-8) demonstrates the potential of using cameras and processing to assist us in managing our physical environment. The Room Monitor summarizes activity in one's office according to user specified visual criteria, such as noting visitors through observations of motion.

Figure 1-8: The Room Monitor

The Whiteboard Recorder (figure 1-9) is a more specialized application which also demonstrates the potential for computers being made aware of the physical world through the use of camera sensors. The recorder summarizes the activity on one's whiteboard over a period of time by constructing a succinct set of still images from the video stream.

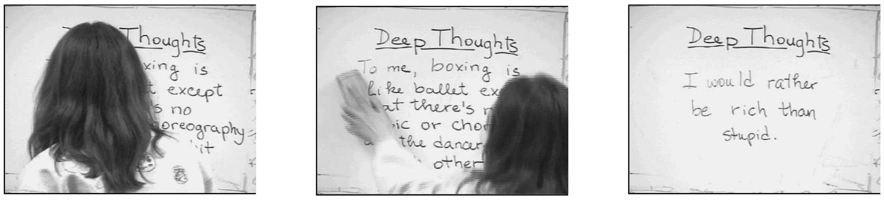



Figure 1-9: The Whiteboard Recorder

The Computer Librarian (figure 1-10) is another specialized application, similar in function to the Whiteboard Recorder. The librarian keeps track of who has borrowed what books from a collection by saving a video clip each time a visitor removes or returns a book.

Figure 1-10: The Computer Librarian

Finally, the Gesture Recognizer (figure 1-11) demonstrates the potential for programs that analyze media to provide for a more natural interface between a human and a computer. The Gesture Monitor is a configurable application designed to permit the user to signal commands to the computer without the assistance of a mouse or keyboard.

Figure 1-11: The Gesture Recognizer

An important element of the overall approach is its focus on interactive and task specific solutions to media interpretation problems. Rather than attempt to solve general vision problems, this thesis acknowledges that while such solutions are desirable, they are difficult to achieve. Instead, this work promotes the aggressive use of application constraints to simplify the computational task. For example, the Whiteboard application exploits the fact that the writing on the board is darker than the board itself. The Room Monitor takes full advantage of the fact that its input comes from a stationary camera while the Television Agent attempts to use the repetition in the format and graphics used in TV broadcasts to find interesting video clips.

The key to exploiting such constraints is to allow the user to specialize the computation. Rather than attempting to build an intelligent system that can specialize itself, this work leverages the intelligence of the user and focuses on the problem of bringing the user closer to the computational task. Sieve provides the linguistic means whereby the user can rapidly combine and configure reusable modules into specialized applications. In addition, Sieve provides high level tools such as the Vex module. Vex enables its user to specify the characteristics of the media that he or she is interested in and to perform actions based upon whether the media meets the specification. Vex frees the user from the mundane programming details of memory management, flow design, and pattern analysis; allowing the user instead to focus on the application specific details of determining what video the he or she is interested in and what they want to do when such a piece of video is encountered.

An important assumption of this approach is that the user knows how to specialize the computation and is willing to do so. Thus, an immediate limitation to this approach is that it expects the user to be a programmer. A subject for future work (described briefly in section 9.3), is to enable the system to learn from examples provided by the user. Such a system could be based on the same computational framework, but would be useful to a wider population.

Sieve provides a platform for exploration of the subset of applications that interpret visual input and produce useful results without the automation of higher level reasoning. This is not meant to imply that human visual proficiency could ever be achieved by such a system. Instead, Sieve is presently targeted at applications where the visual criteria, or properties, are known ahead of time. For instance, a Sieve program might attempt to identify a person by judging the similarity of the input to a model or set of models which represent expected views of the individual.

Sieve currently has no general mechanisms for learning new views or adjusting expectations, though it is of course possible to program Sieve applications to adapt themselves in specific ways on a case by case basis. Whether deeper intelligence could be achieved by overlaying automated reasoning on top of Sieve's symbolic framework remains to be explored.

## 1.2 Road Map

The next chapter relates the work presented in this report to research from the fields of computer vision and interactive multimedia programming systems. Chapter 3 then describes Sieve's approach and system architecture, which is based on low level image processing and higher level pattern matching.

The next three chapters detail the analytical components of the system. Chapter 4 describes those which directly process the "bits" in the media stream. These components transform streams of unstructured video into sequences of properties. Chapter 5 describes ways in which these properties are used to dynamically adjust the system. Finally, chapter 6 describes how patterns of properties are realized and used to generate symbolic media events.

Examples are presented throughout the first 6 chapters that demonstrate Sieve's functionality as a programming toolkit. Chapter 7 moves to a higher level of abstraction by presenting VsGrep, a stand-alone tool based on Sieve for implementing interactive applications that analyze video content. The chapter concludes by presenting several applications built using this tool. Finally, chapter 8 presents quantitative and qualitative performance results for the system and chapter 9 reviews the thesis in light of the experimental work, discusses various insights, and suggests directions for future work.

# Chapter 2

# Related Work

This chapter reviews research in three areas related to this thesis: approaches to computer vision, visual content-based retrieval systems, and multimedia programming systems.

## 2.1 Approaches to Computer Vision

This thesis argues for the aggressive use of application constraints to simplify the vision task. As such, the approach bears similarity to the task oriented approach advocated by Ikeuchi and Herbert [1990] and to the active-vision paradigm summarized by Swain and Stricker [1993]. Horswill [1994] and Woodfill [1992] have both successfully applied this approach to the problem of building special purpose real-time vision systems. However, they are vertically integrated and lack Sieve's layered abstractions, e.g. properties and patterns.

A well developed, general model for task directed visual perception, put for forth by Ullman, is that of *Visual Routines* [Ullman, 1984]. Ullman's model divides the processing of visual information into two stages. The first is a data driven, bottom up stage which creates base representations such as the primal sketch and the 2 1/2D sketch [Marr, 1979]. The second is a goal driven, top down application of visual routines which establishes properties and relations from these representations. The routines are composed from a limited set of elemental operations by a *routine processor* that is driven by high level, cognitive components of the system. Ullman argues that while the elemental operations are selected to meet specific objectives, their results may be shared by subsequent processes.

The fundamental elements of Ullman's model are present in the computational framework developed in this thesis. The first stage of Ullman's analysis may be performed by translation filters which compute edges, motion, and other basic representations. The second stage is performed by model matching and property generating filters which look for specific properties in the image stream. As in Ullman's model, the incremental representations computed by these modules are available in all subsequent processing.

However, this thesis calls upon the user, rather than an AI subsystem, to supply the higher level components needed to specialize the processing task. In addition, the user may assume the role of the routine processor by explicitly configuring the data flow network. Alternatively, higher level facilities, such as VsGrep (see chapter 7) may be used to automate this function. Finally, this thesis also differs in that it proposes a specific model of communication between the levels of computation.

## 2.2 Content Based Retrieval

Driven by the recent explosion in digital media brought about by multimedia capable systems and the world wide web, content-based retrieval systems have become a major focus of the vision research community. The goal of these systems is to provide an interactive interface which enables a user to browse large databases of digital media. Traditionally, these systems have enabled the user to query the database for entries which match hand annotated keywords. Recent work, however, has enabled users to query based on the actual media content (i.e. the pixels) using techniques such as color indexing [Swain and Ballard, 1991], shape matching, and texture matching [Picard, 1996].

A considerable number of new research projects and commercial systems have recently come on-line. While not identical, many of these systems are very similar to one another. Described below are four projects which cover most of the main ideas and have been particularly influential: IBM's QBIC system, the MIT Media Lab's Photobook and FourEyes platforms, Berkeley's Chabot System, and CMU's Informedia Project.

**IBM's 'Query By Image Content" (QBIC) system** [Flickner *et al.*, 1995] [Niblack *et al.*, 1993] has evolved from a research project into a commercial system. QBIC extends IBM's relational database, DB2, to support queries of images and video by visual properties such as shape, texture, and color. The system includes a polished graphical user interface to assist the user with both data entry and retrieval. An important element of the system is that during data entry, the user identifies and optionally adds semantic labels to objects which the system then characterizes using a fixed feature set. Queries may be made by text using both semantic (e.g. "show me fish") and feature (e.g. "show me blue textured objects") based descriptors. Alternatively, similarity based queries may also be made by supplying a sample image or, innovatively, by drawing a sketch.

**The MIT Media Lab's Photobook system** [Pentland *et al.*, 1996] is another content-based retrieval system that allows a user to interactively query a database for image and video entries. Like QBIC, the entries in the database are annotated both by human operators, which may attach semantic labels, and by computers, which may attach feature vectors. The choice of feature vector depends on the population and the application of the database so that a different feature would be used to characterize human faces than would be used to characterize hand tools. An important database element that Photobook lacks is an index. While the reference [Pentland *et al.*, 1996] suggests that the feature vectors constitute an index, it is more accurate to refer to these descriptions as annotations. This is because a query for a matching feature vector requires a linear scan through the entire database. The benefit of pre-computing the feature vector, like the benefit of adding an annotation, is that the processing time per entry may be greatly reduced. Thus, Photobook is more a platform for interactive, content-based searching than a true database. The FourEyes system [Picard and Minka, 1995] extends the Photobook platform with an advanced interactive interface that assists the user in identifying regions of interest and in adding labels. The advanced interface engages the user in a multi-stage interaction where the user provides positive and negative examples while the system provides feedback.

**The Berkeley Chabot System** [Ogle and Stonebraker, 1995] is designed to manage a large collection of images. The Chabot database, which consists of a collection of nature photographs from the California Department of Water Resources (DWR), is in the process of growing from roughly 11,000 images in 1995 towards an eventual goal of 500,000 images (the complete DWR collection). In order to manage this volume of data, Chabot starts with an advanced relational database system that provides multiuser access, large scale storage, and flexible indexing. Images in the database are annotated by human operators who add semantic descriptions of each scene. In addition, images may be annotated by computer generated features such as the color histogram. An important difference between Chabot and Photobook is Chabot's ability to create custom indices based on these features (e.g. an index of images containing mostly yellow). However, the features employed by Chabot are significantly less sophisticated than those of Photobook. Also, unlike Photobook and QBIC, Chabot does not support image segmentation. Thus, the only "objects" in the Chabot system are the complete scenes depicted in the images. Research underway at Berkeley to address the segmentation issue is reported in [Carson *et al.*, 1996].

**The CMU Informedia Digital Video Library** [Wactlar *et al.*, 1996] is a recent project aimed at building a large multi-purpose library of raw and edited video to be distributed over a metropolitan network. The project concerns itself with the problem of searching and browsing the online content. The goal is to simultaneously use speech recognition, natural language understanding, and computer vision to automatically index the data for content-based retrieval. The vision component includes the detection of faces [Rowley *et al.*, 1995], graphical text, and production effects (e.g. shot changes, fades, pans, zooms). A major thrust of the research has been in using automatic analysis to summarize videos into compact "skims" so that a user may judge the relevance of a retrieved clip at a glance [Smith and Kanade, 1995].

**Discussion** There are numerous additional systems similar to those described here. These include several based on color and texture matching: the Virage system [Bach *et al.*, 1996], Los Alamos National Lab's Candid system [Kelly and Cannon, 1995], Columbia University's VisualSeek system [Smith and Chang, 1996], and Boston University's ImageRover [Sclaroff *et al.*, 1997]. There are also several systems based on shape matching, including Singapore Institute of Systems Science's CORE system [Wu *et al.*, 1995] and the University of Northumbria at Newcastle's ARTISAN system [Eakins *et al.*, 1996]. Finally, amongst the systems aimed at motion analysis and video parsing are the University of Palermo's JACOB system [Cascia and Ardizzone, 1996] and the Singapore Institute's system for analyzing news footage [Zhang *et al.*, 1995].

All of these systems share important similarities to Sieve. In particular, these systems:

- **Perform image analysis**. All of the systems listed above include at least some ability to analyze image content. The techniques employed by these systems are similar to the techniques used by Sieve to generate properties.
- **Reduce multimedia information**. These systems are designed for browsing large collections of images and videos. They, thus, share one of Sieve's objectives of reducing vast amounts of media information to a quantity that can be dealt with by a human user.

- **Interact with a human participant**. An integral component of each of these systems is the user interface. The interface permits an interaction between the computer and user whereby the user can experiment with different queries while the system provides feedback. Thus, these systems are philosophically similar to Sieve in that they aim to exploit the intelligence and flexibility of the human user to achieve their results.

Despite these similarities, Sieve differentiates itself in several important ways. In particular, Sieve:

- **Supports live video**. While Sieve is a capable tool for searching through video files, it has been designed primarily as a tool for analyzing live video. The resulting flow-based architecture is designed to continuously process a stream that, conceptually, is infinite in duration.

- **Models time-varying behavior**. Sieve employs a state model to represent time-varying behavior. The model enables Sieve to match sequences in which features evolve over time. The approach is thus similar to recent work involving the use Hidden Markov Models (HMMs) to recognize hand gestures and human dynamics from sequences of observed symbols [Lee and Xu, 1996] [Bregler, 1997] [Starner and Pentland, 1996][1].

- **Performs processing at application time**. Since content-based retrieval queries are based on features, this requires that entries be organized by feature vector. Such organization, however, requires that one know what the interesting features are at the time that the data is entered into the system. Since different features work best for different tasks, this poses a serious problem. However, even if one could develop a limited set of useful features to characterize images, and could then afford to index by every feature vector in this set, one would still need to know what objects in the image to *select* for indexing. Presently, content-based retrieval systems deal with this problem by either having a human operator pre-select the objects of interest (e.g. QBIC) or by simply limiting queries to descriptions of entire scenes (e.g. Chabot).

  In Sieve, the processing is performed at the time that the recognition task has been defined. Thus, a Sieve application typically knows what objects are relevant, what features should be used and where such objects may appear. Moreover, since Sieve restricts itself to processing a limited number of input streams at a time and knows what it is looking for, it is often possible to directly compare inputs to models. Thus, Sieve may employ many matching techniques (for example, the matched filtering and Hausdorff distance techniques described in chapter 4) which are simply not applicable to indexing.

- **Sieve is a programming system**. The content-based retrieval systems provide a fixed user interface designed to meet the requirements of a specific task. The research in this area has focused on developing particular techniques for matching and indexing, and on developing specialized user interfaces for entering and querying data. In contrast, Sieve is a programming system that allows the user/programmer to cus-

[1] Indeed, the symbolic input of a trained HMM analyzer is equivalent to that of the finite state automata. Thus, it should be possible to incorporate such analyzers into Sieve as drop-in replacements for the pattern matchers that are presently implemented.

tomize the processing, user interaction, and function of an application. The focus of the research is not on the specific techniques but rather on the reusable interfaces that define how capabilities are accessed and managed by the programmer.

- The Property/Predicate/Pattern/Event interfaces. These interfaces define how information evolves from raw sensory data to symbolic form.
- The Program/System interface. The programmer first creates modules which define input/output relationships and then connects them into a flow-graph. The system assumes the responsibility for managing the flow of data through the graph. This separation of responsibility allows the system to schedule access to the network, disk, camera device, screen and so on in a manner that minimizes idle wait states and I/O overhead. Potentially, this separation could also be used to schedule parallel execution of data processing.
- The scripting interface. This interface defines how a programmer accesses and extends the capabilities provided by the system so that issues such as abstraction and composition are a major focus. Scripting is of particular importance to Sieve because of Sieve's objective of allowing the programmer to specialize the system to perform a particular task.

## 2.3 Media Systems

Sieve builds on and extends the MIT/LCS **VuSystem** programming environment [Lindblad and Tennenhouse, 1996] [Tennenhouse *et al.*, 1995] [Adam *et al.*, 1993]. The VuSystem was designed to allow for software-based processing of temporally sensitive data. The system is extensible, embeds a scripting language and provides a data flow programming model. Thus, the VuSystem was judged to be an appropriate implementation platform for Sieve[2].

The VuSystem bears some similarity to the **Continuous Media Toolkit** (CMT) developed at Berkeley [Jackson *et al.*, 1996]. Both provide a data flow programming model and both embed a Tcl interpreter to provide a scripting interface. CMT even employs the object-oriented extension, OTcl, written by David Wetherall as part of his work on the VuSystem [Wetherall and Lindblad, 1995]. The main difference is one of emphasis. Whereas the VuSystem was designed for compute intensive application, CMT emphasizes audio and video play back. CMT Research focuses on editing, chaining video segments into continuous streams, and maintaining presentation quality over wide area networks.

The **Khoros** system, developed initially at the University of New Mexico and commercialized by Khoral Research, Inc, is another data flow based image processing system [Young *et al.*, 1995]. Though similar in many ways to CMT, Khoros represents the other end of the spectrum in terms of emphasis – focusing on data processing and visualization rather than on video presentation. Khoros includes an extensive library of image transformation modules. However, it does not provide a framework or mechanism for representing the progression of information into symbolic form.

A different approach is taken by systems such as **VideoScheme** [Matthews *et al.*, 1993]

[2] Background on the VuSystem is provided in chapter 3.

and **Obvius** [Heeger *et al.*, 1992]. These systems implement a more procedural model of computation. Both systems consist of an interactive video editor with an embedded lisp (Scheme for VideoScheme and Common Lisp for Obvius) interpreter. The interpreter allows for the automation of repetitive editing tasks, including those which are based on the content of the media data. VideoScheme, for instance includes built-in functions which transform images to histograms, normalize histograms, and compute differences between histograms. These primitives were used to implement a cut detection algorithm. VideoScheme and Obvius essentially extend the capabilities of video editing systems. The programs are file based and do not model the flow of media data through system. As a result, the systems are not directly applicable to the processing of live video.

Finally, **ActiveMovie** is Microsoft's new[3] API for creating multimedia applications on the Win32 platform. Presently, this API supports the playback of multimedia streams in a variety of formats from local files and network sources.

The ActiveMovie architecture is startlingly similar to that of the VuSystem. Microsoft describes ActiveMovie as an architecture to "control and process streams of time-stamped multimedia data by using modular components called filters connected in a configuration called a filter graph". The configuration of the filter graph and the flow of data through the graph are controlled by a "filter graph manager". Application programs configure the filter graph by communicating with the filter graph manager through a set of Component Object Model (COM) interfaces which may be accessed from different languages such as C++ and Visual Basic. In a manner nearly identical to the VuSystem, filter modules communicate with one another by passing time stamped data through the flow graph and communicate with the application by posting events.

ActiveMovie does not provide tools for media analysis. Presently, the SDK provides filters for reading and writing audio and video data in a variety of formats such as MPEG, QuickTime, AVI, and Wave and a few simple transformation filters, such as a general color space transform filter and a special purpose 16bit-color VGA dither filter are provided. However, modules which examine and interpret the data have not been implemented. Moreover, facilities such as the property and pattern mechanisms developed in this thesis, do not exist. As a result, filter events related to the content of the data are currently limited to those which indicate the detection of errors in the data stream.

The emergence of ActiveMovie suggests a technology transfer path for the research described in this thesis. Since the programming models for ActiveMovie and VuSystem applications are nearly identical, the approach developed for analyzing media content should be directly applicable to both.

[3] Released in December, 1996

# Chapter 3

# The Sieve Architecture

In order to realize an effective approach to processing media, this thesis has developed a system capable of analyzing media information while responding to an interactive user. The approach is to reduce streams of media input into sequences of media events (figure 3-1) and then to handle these events in a manner similar to how programs handle user events.

One may view any program as a filter that takes a stream of data as input and produces a stream of output. Interactive programs are filters which consider input generated by a user, such as that which is produced by keystrokes and mouse clicks. However, unlike many programs which process an input stream that is fixed and produce an output stream which is not time sensitive, interactive programs face fundamental temporal challenges. User input is not naturally synchronized with the data processing. It is difficult to predict what it will consist of and when it will occur. Furthermore, for most applications user input must be handled in a timely fashion.

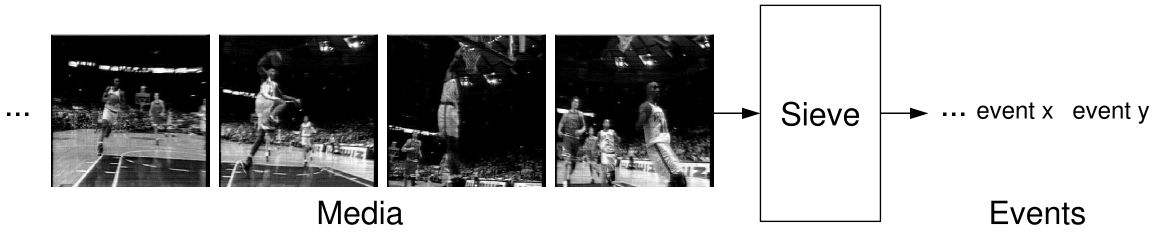



Figure 3-1: Transforming unstructured media into symbolic events

For traditional media, these problems have been addressed by interactive systems such as X [Asente and Swick, 1990], Mac OS [Apple, 1992], and Microsoft Win32 [Microsoft, 1995], using an event driven programming paradigm. In this paradigm user actions are modeled as external events which are queued up by the system and passed to the application. Widget and class libraries process the low level events, such as mouse movements and clicks, into high level events, such as scrolls and drags, and pass these events to the appropriate handler routine. This paradigm has been successful in allowing the implementation of time sensitive, user driven programs.

Media input is in many ways similar to user input. Namely, media input is time sensitive, continuous, and not naturally synchronized with the application. Thus, one may expect

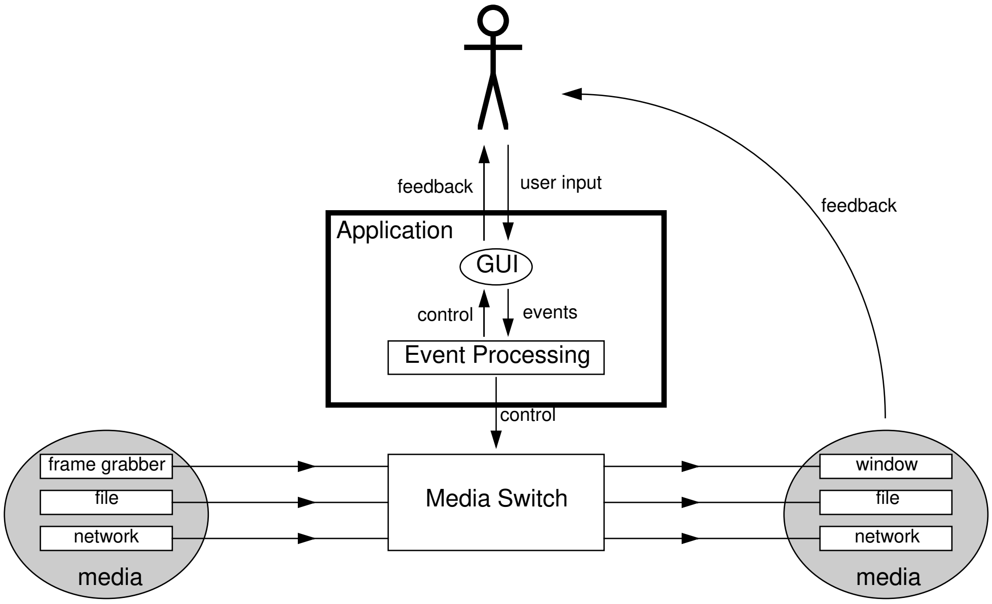



Figure 3-2: Traditional interactive multimedia application

that a paradigm similar to that used for handling user input will prove effective for handling media input. Furthermore, by treating both media and user input similarly, this approach offers the opportunity for the seamless integration of these two sources of input.

Figure 3-2 depicts the structure of a traditional multimedia application built around an event processing core that responds to user events, provides feedback to the user, and outputs control signals that "steer" the media information. The main limitation is that the media is not available to the application, and so, the program is not able to sense or change its content. Instead, the computer acts as a mediating agent through which the user directs and controls the flow of audio and video information.

The introduction of media processing and media events changes the structure of an interactive multimedia application. The new structure enabled by Sieve is depicted in Figure 3-3. The media is brought into the application so that rather than treat the media as an external, opaque entity, the program may sense and manipulate the media's content. The media may even drive the behavior of the program in a manner similar to how a user may drive an interactive program. Finally, with the use of cameras and microphones, the user may in some cases appear in the media itself. For example, media processing modules that recognize a user's hand gesture could provide the application with a way to sense the user in the user's physical environment. Finally, with the use of cameras and microphones, the user may in some cases appear in the media itself, providing the system with a new way to sense the user in the physical environment. Such systems may, for example, recognize a user's hand gesture or track a user's whereabouts.

The rest of this chapter describes the structure of a Sieve application in greater detail. Section 3.1 describes how media data and media processing are modeled in a Sieve program. Section 3.2 goes on to preview how, using this model, applications recognize and signal media events. This topic, which is central to this thesis, is covered in greater detail in Chapters 4 and 5. Section 3.3 provides details concerning the Sieve implementation. Finally, Section 3.4

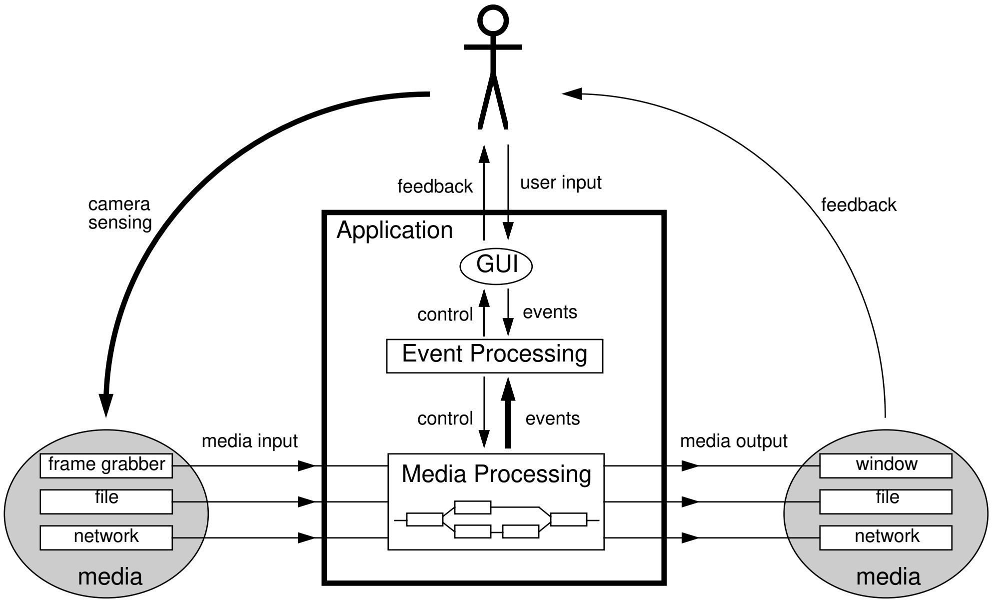



Figure 3-3: New multimedia application architecture

presents an example of a complete Sieve application.

## 3.1 Media Data and Media Processing

The problem of manipulating media is that of handling temporally sensitive information. Media spans time. Interactive programs which process media while involving a human user must meet timing conditions imposed by human perception when presenting media information.

This thesis employs a data flow model of media computation developed in [Lindblad, 1994]. In this model (depicted in figure 3-4), streams of media flow in a pipelined fashion through networks of processing modules. These modules manipulate and transform the media data as it flows through the network.

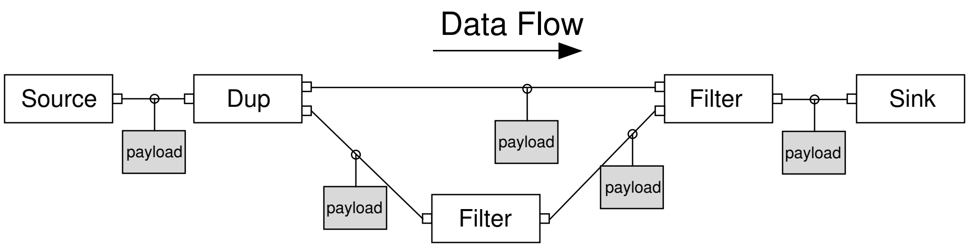



Figure 3-4: Data flow computation

The media data flows through the network in units referred to as payloads. Payloads are lumped data packages which represent finite time-slices of a media stream. For some types of data, the segmentation of the stream into units is natural. For example, a video stream is quite naturally broken up into a sequence of image payloads (i.e. frames). Still, other segmentations are possible. Video data could, instead, be passed from module to module as scan line payloads or as multi-image video clip payloads. The choice is a matter of convention which is determined by considering issues such as communication overhead, latency requirements, and programming convenience.

Other types of media, such as sampled audio, have less obvious conventions. Since packaging individual samples as data units would lead to unacceptable communication overhead, audio samples are grouped into fragments consisting of a number of samples that trades off latency for overhead.

The interaction of the system with the media data occurs in the processing modules. Processing modules may act as data sources, sinks, or filters. A *source* module is one which internalizes a stream of payloads to the system. In practice, such a module may be grabbing payloads from a device, such as a camera, forwarding them from a network, reading them from a file, or generating them from a mathematical formula. However, from the point of view of the program, a source module produces a media stream and therefore has no input ports.

A *sink* module is one which externalizes a stream of payloads. A sink module may display payloads on a console, transfer them to a network, dump them to a file, or simply throw them away. From the point of view of the program a sink module consumes a media stream and therefore has no output ports.

A *filter* module is one which both consumes and produces streams of payloads. One normally thinks of filtering as an operation which takes a single stream of input and produces a single stream of output. This thesis, however, broadly defines a filter module as any module that contains at least one input port and and least one output port. Thus, a filter module may consume and produce a single stream of payloads. In addition, however, a filter module may combine two streams of payloads into a single stream, split a single input stream into multiple output streams, or both input and output multiple streams simultaneously.

In order to support a variety of media types and data formats, payloads are specified to be self identifying data structures. Logically a payload contains two parts: a *descriptor* and *data*. The data is simply a block of memory which contains the media bits. The descriptor specifies the representation of the data. For example, a descriptor might specify that a payload represents an image of a certain height and width, encoded as 8 bit grey scale samples, listed row by row starting from the upper left corner. It is the responsibility of the processing module to examine the descriptor to determine the type and format of the payload.

### Perspective

The most important consequences of the media flow style of programming is that modules are restricted in their ability to look back at the data. Unlike a document based programming model, where the entire document may be internalized and analyzed in any order, processing modules must explicitly limit and manage their state. This means that if a module may need to examine a payload in the future, it must store that payload locally (or make a copy of it) until such a time.

While this restriction may be an inconvenience to the programmer, it reflects the realities associated with processing media information. Media data places tough demands on computational storage so that when processing media it is impractical to adopt an infinite memory storage model. Moreover, while the system may be used for processing stored media, it is designed to be applicable for processing continuous streams of live data coming from a camera or network where future data isn't available and past data is limited to what has been explicitly stored.

### A Flow Graph Example

Figure 3-5 depicts a flow graph for extracting a moving object from a stationary background. A representative output is depicted in figure 3-6.

The boxes in the figure correspond to the primitive filter modules which manipulate the media. An image payload, originating from the source module is passed to a `VsDup` module which makes three copies of the payload[1] and passes them to a `VsColor` module, `VsDiff` module, and `VsStationary` module.

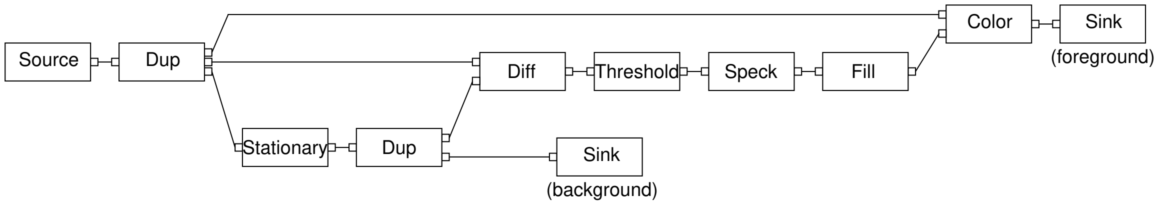



Figure 3-5: Extracting a moving object from a stationary background

The `VsStationary` module, described in Section 4.3.3, executes code which attempts to compute the background of a scene. Each time the `VsStationary` receives an input image it calculates its best estimate of the background and sends the resulting output to another `VsDup` module. The second `VsDup` sends one copy of the background image to a sink module (which displays the image) and the other to the second input of the `VsDiff` module.

The `VsDiff` module waits for an image payload to appear at each of its input ports and then outputs a gray scale image computed by taking the pixel by pixel difference between the two input images. The difference image is passed to `VsThreshold` module which computes a binary map image indicating where the magnitude of the difference is greater than a specified threshold. In this case the two images are the current view and the current estimate of the background, so the map indicates where the scene differs from the background. The map payload is passed down the pipe, first to the `VsSpeck` module which removes spurious data points (see section 4.4.2) and then to the `VsFill` module which fills in missing regions (see section 4.4.2). The `VsColor` module combines the binary image payload with an original copy of the image, replacing true valued pixels in the map with color or gray scale pixels from the scene while setting false valued pixels in the map to a specified background value. The combined payload is passed to a sink module which displays the resulting images and terminates the processing pipeline.

[1] `VsDup` does not copy the actual data, just a reference counted pointer to the data

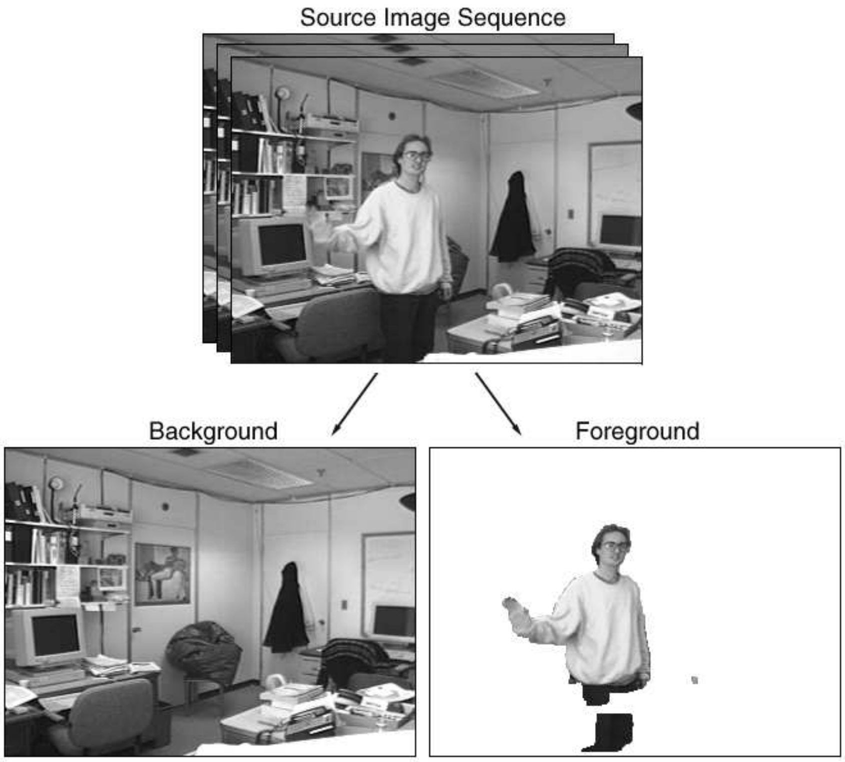



Figure 3-6: Output of the moving object extraction flow graph

A more complete description of the function of this computation, as well as descriptions of the primitive modules, is presented in Chapter 4.

## 3.2 Recognizing Media Events

The flow graph example of the previous section demonstrated the ability to manipulate and transform the video but ultimately the output, like the input, was a stream of images. There was no transition towards a decision or independent action taken on account of the media's content.

The approach taken by this thesis is to treat the problem of understanding media in terms of recognizing *media events*. Programs which use content information from the media stream are modeled as event driven programs which are augmented to respond to media events in addition to standard user events.

An event is a meaningful pattern of data in a temporally sensitive input stream. A "meaningful pattern" is an occurrence about which the program wishes to be notified. Such patterns are summarized to an event driven program as a discrete sequence of symbolic descriptors which describe the identity, time, and relevant details of the events. A media event is a symbolic description of the time and identity of a meaningful pattern of data in a media stream.

For example, a media event may signal the detection of movement in a specified area of a scene, the appearance of a particular color pattern in the imagery, or the detection of a human face in a frame of video. The significance of such a pattern is assumed to be known to the user or programmer. In the case of a security application, for example, the input may come from a fixed camera and the presence of motion may indicate the possibility of an intruder.

For the media event model to be useful there must be a computational means of recognizing patterns in a data stream. In keeping with a modular data flow approach, media analysis is performed in stages by filter modules which transform their input and add properties. Properties, in turn, are used by downstream modules to direct media flow and to recognize patterns. Modules which recognize patterns communicate successful matches by signaling events.

### Transformations

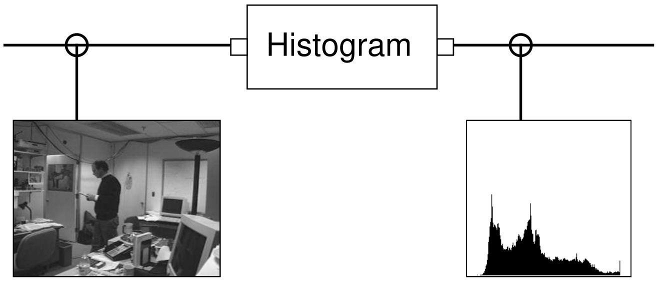



Figure 3-7: Data transformation

Vision processes may be modeled as a series of representational transformations. In Sieve, such transformations are performed by modules which filter streams of payloads. The purpose of these transformations is to expose the salient information from the image data.

For example, the `VsEdge` filter transforms a stream of images from an intensity based representation to a boolean format which explicitly depicts intensity boundaries. Similarly, `VsDiffMotion` uses a binary image format to explicitly represent temporal changes in image intensity. Other modules perform transformations where image data is converted to non-image based representations. For instance, the `VsHistogram` module, illustrated in figure 3-7, transforms images into signal payloads which represent intensity statistics in the data.

### Properties

Properties represent time varying measurements which may be made on media streams. As shown in figure 3-8, these measurements are attached to payloads by processing modules as they pass through the data flow network. The addition of a property to a payload leaves the data otherwise unchanged and there is no fundamental limit to the number of properties which may be accumulated by a given payload.

Each property consists of an identifying symbol and a value. In general, a property value may be an integer, floating point number, boolean number, or a symbol representing a string. The identifying symbol or name of the property that a module uses to describe its

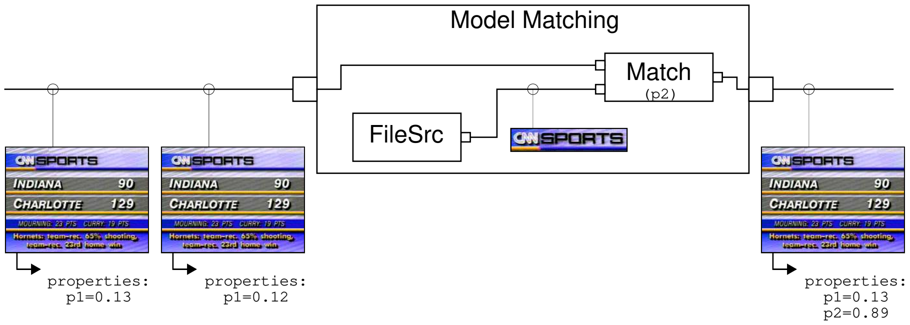



Figure 3-8: Property generation

measurements is typically a parameter to the module. Vector quantities are supported by defining multiple properties with related names[2]. For example, when the user specifies for a `VsFaceDetect` module to use the property symbol `face`, the module generates the properties `face.left`, `face.right`, `face.top`, and `face.bottom` to represent a rectangular region in the image where a face has been located.

Property measurements often summarize information exposed through representational transformations. For instance, a set of properties may characterize the output of a motion detector, indicating the amount of change in a scene and the location of the centroid of the moving pixels. However, the most prevalent use of properties is to represent the degree to which a transformed input matches a model. When used as such, the property name identifies the model to which the input is matched. For example, the `VsPixelMatch` filter, described in Section 4.1.1, takes a model image and an image stream as input and produces as output a stream of properties which it attaches to the image stream payloads as they flow through the filter. The property symbol is a name which identifies the model image while the property value is a floating point number indicating the degree of similarity between each image and the model.

Properties are well suited to the representation of a relatively limited set of values. Complex or dense data representations, such as an image or a histogram, are more efficiently encoded in the data segment of a payload, e.g., through a transform filter. Still, because properties relate to models, they are capable of implicitly representing the complex information captured by the model. For this reason, and because the objective is to transition from unstructured media towards discrete, symbolic events; properties are an effective mechanism for encoding visual information.

### Routing

In addition to transforming payloads and adding properties, processing modules may direct or route the media data. An example of a routing module is `VsClassify`. `VsClassify`, shown in figure 3-9, examines the properties of the payloads in its input stream and routes them to either its `True` or `False` output port depending on a specified property's value.

[2] A more direct method of representing vector quantities can and should be implemented. Such a feature would add elegance to the Sieve implementation but would probably provide no significant insights into the merits of the system.

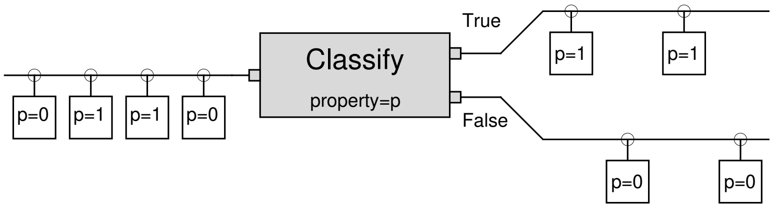



Figure 3-9: Routing

Routing or classifying payloads may be used as a preliminary step towards generating a media event. By routing payloads to one subgraph or another, a classify filter reduces the workload on downstream processing modules. Section 5.2 discusses the role of routing modules.

**Events**

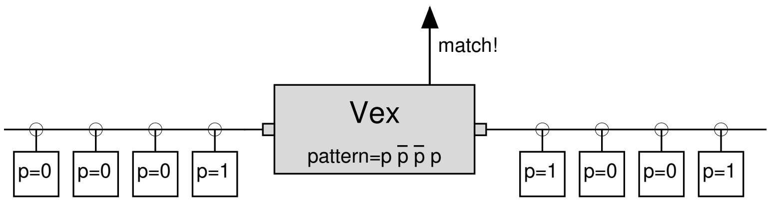



Figure 3-10: Recognition

Ultimately, the objective of the recognition modules is to make a decision. In Sieve, such decisions are expressed as events which are signaled by modules (as shown in figure 3-10) in response to observing specific patterns of input observed in sequences of images. Presently, the event generating modules limit themselves, to examining the sets of properties attached to payloads. Generality is preserved because any pattern matching task that requires direct examination of the payload data may be implemented as a filter module that signals a pattern match by attaching a boolean property that is detected upstream by an event generating module.

There are many different kinds of patterns which could be defined over sequences of properties. A pattern matcher could, for instance, signal an event whenever a specific property reaches a given threshold. Alternatively, a pattern matcher could accumulate the running total of a specific property and signal an event whenever, say, the average of the last `n` values dips below a low water mark.

This work has focused its attention on pattern matchers which may be expressed as non-deterministic finite state automata. Such state machines are well understood and widely used for recognizing patterns in text. An important motivation for choosing finite state automata is that they have a well developed language, regular expressions, for concisely specifying their behavior.

Formally, a finite state automata is specified by a a set of states, a start state, final state(s), a set of input symbols, and a transition function [Hopcroft and Ullman, 1979]. The transition function specifies, for every possible input and state configuration, the new state configuration. Typically, the transition function may be represented by a directed graph where the nodes represent states and the edges represent transitions on particular input symbols.

In order to specify the transitions on a finite state automata where the input is not a sequence of symbols, but rather, a sequence of property sets, this thesis employs *predicates*. A predicate is a boolean function defined over sets of properties. Predicates are used to specify which edges of a finite state automata graph are followed for a given input. The automata signals an event when a sequence of inputs results in the transition of the automata into a final state.

By mapping properties to boolean valued predicate symbols, it becomes possible to use a regular expression syntax to concisely specify a pattern matching automata. The `VsVex` module, described in Section 6.2, uses such syntax to allow a script programmer to conveniently represent a specialized recognition task.

In summary, *properties* represent time varying measurements which may be made on media data. *Predicates* are time varying boolean values which represent simultaneous combinations of properties. *Patterns*, which are recognized by finite state automata, represent sequences of predicates. Finite state automata are specified as either state transition graphs or, equivalently, as regular expressions.

#### Buffering

An important difference between the processing of traditional user input and the processing of media input is that the transformed media data may be an important part of the output. For example, a program that monitors one's office may, in addition to making decisions about what is going on, capture and display video clips corresponding to the recognized event.

To meet this requirement, pattern matchers are typically augmented with buffering capability. For example, the `VsAutomata` module, which implements the finite state automata pattern matchers describe above, may be configured to hold payloads in a buffer queue until a pattern is recognized. The buffer is necessary because for most patterns, it is impossible to tell when a payload arrives whether that payload will be part of a match. The pattern matchers are the natural place to do the buffering because they have the information needed to determine when a payload may be released from the queue or discarded.

The considerable storage requirements for video information make it necessary to carefully manage the buffer queues. Accordingly, Sieve provides the pattern matchers with a mechanism which causes payloads to be written to secondary storage when a buffer grows beyond a specified size.

## 3.3 Additional Background

The following subsections provide information specific to the VuSystem and Sieve which may aid the reader's understanding of the example code fragments that appear in this report. Further details of the VuSystem are described in [Lindblad, 1994].

### 3.3.1 Modules

In the VuSystem, primitive modules correspond to C++ objects which are compiled into the system. The C++ objects implement a work method which contains the high performance code which processes the arrays of bits in the media stream. The VuSystem design specifies the communication and scheduling protocols which define how payloads are transfered between processing modules and when the `Work` method code is run.

Processing modules are combined into flow graphs using the Tcl scripting language, which provides the means of creating and destroying modules, connecting their inputs to outputs, and specifying parameters which affect their behavior. The VuSystem extends the Tcl scripting [Ousterhout, 1994] language for this purpose. Figure 3-11 shows a script that creates the flow graph depicted in figure 3-5.

```
VsVideoSource vs.source \
  -videoSource :vidboard1
VsDup vs.dup1 \
  -numOutputPorts 3 \
  -input "bind vs.source.output"
VsStationary vs.stationary \
  -threshold 20 \
  -input "bind vs.dup1.output0"
VsDup vs.dup2 \
  -numOutputPorts 2 \
  -input "bind vs.stationary.output"
VsSink vs.background \
  -input vs.dup2.output1
VsDiff vs.diff \
  -input1 "bind vs.dup1.output1" \
  -input2 "bind vs.dup2.output0"
VsThreshold vs.thresh \
  -threshold 20 \
  -input "bind vs.diff.output"
VsSpeck vs.speck \
  -input "bind vs.thresh.output"
VsMaskFill vs.fill \
  -input "bind vs.speck.output"
VsColor vs.color \
  -foreground -1 \
  -background 0 \
  -input "bind vs.dup1.output2" \
  -mask "bind vs.fill.output"
VsSink vs.foreground \
  -input vs.color.output
```

Figure 3-11: Script for implementing the moving object extraction flow graph

In addition to providing means of combination, the scripting language provides the ability to create first class abstractions. For example, the filter modules which comprise the flow graph shown in figure 3-5 may be grouped into an abstract, one input, one output filter, shown in figure 3-12, which inputs a stream of image payloads and outputs a stream of images with the background removed[3]. The ability to create first class scripting objects with the same abstract properties as their C++ counterparts is provided by ObjectTcl [Wetherall and Lindblad, 1995], an extension to the Tcl scripting language written by David Wetherall as part of his research studying a graphical programming methodology for building media flow applications [Wetherall, 1994].

[3]The output to the second sink, used for displaying the background, was not included in the abstraction.

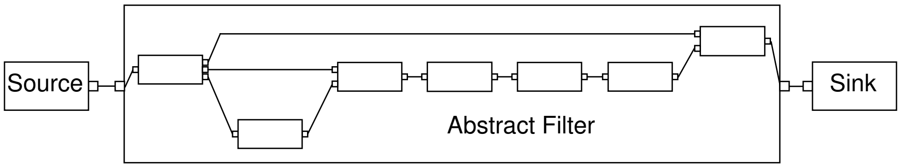



Figure 3-12: Moving object extractor abstraction

### 3.3.2 Payloads

Media data flows through the graph of processing modules in units referred to as payloads. Payloads are time stamped, typed data structures. The time stamp on each payload is used by presentation which need to know the temporal relationship between data units. For instance, the `VsWindowSink` module, which renders video frame payloads on the screen, uses the time stamp field to determine how much delay to insert between renderings and to synchronize video playback with audio playback.

Time stamps may also be used as identifiers for a payload. No two payloads of the same type may have the same time stamp[4]. In addition, the time stamps for a given payload type flowing through a pipeline always advance. Thus, modules which match patterns against sequences of payloads identify the sequence of payloads by their time stamps.

Payloads are passed between modules as reference counted pointers. The actual data resides in shared memory segments. Thus, a module which copies the payload, such as the `VsDup` module, typically performs what is referred to as a *shallow copy*. A shallow copy is one in which the pointer is copied and the reference count is increased, but the data remains in place. Modules which consume payloads decrease the reference count. Memory is freed whenever the count drops to zero. Because a payload's data segment may be shared, filters which alter the media data must either make a copy of the actual data (referred to as a *deep copy*) or check the reference count to be certain that no other module points to it.

A payload may accumulate any number of properties as it flows through a graph of processing modules, each identified by its own symbol. These properties are stored in hash tables which map property symbols to property values. Each set of properties has its own hash table and each payload contains a pointer to one of these tables. As a result, multiple payloads, even payloads of different types, may point to the same table and, therefore, share a set of properties. Thus, a module which transforms a stream of payloads may be implemented so that the transformed payload points to the same property table as the input payload. In such a way, a payload may collect a single set of property measurements even as it is transformed from representation to representation.

The type system allows different kinds of payloads to flow interleaved within a single stream. For example, audio and video payloads can flow along the same path from source to sink. Processing modules are able to modify their actions based on the type and representation of the data arriving at their inputs. Most processing modules pass along payload types that they don't recognize. For example, the `VsJpegC` module, which implements jpeg com-

[4] This is true for data payloads such as those which carry audio and video information. It is not always the case for signaling payloads, such those used to signal the start and end of a flow. Such payloads are described in [Lindblad, 1994]

pression on video frames, simply forwards payloads, such as audio fragments, which aren't video frames. The `VsJpegC` module also forwards video frames which are in unrecognized formats.

Payload types are implemented as C++ classes. The two most common types are `VsVideoFrames` and `VsAudioFragments`. Each type has its own additional descriptor information. For example, a `VsVideoFrame` payload has a height, width, bits per pixel, encoding type (Grey, 8 Bit Color, RGB, or BGR), as well as a number of other fields such as bytes per line, byte order, and others.

### 3.3.3 Event Handling

At the core of any event driven application is an event loop which waits for and responds to external events. For programs driven by a graphical user interface, these events may be user keyboard presses, mouse movements, and button clicks. At a low level, these events can be tedious to interpret. Rather than process these events directly, application programmers interface "widget" libraries which handle the low level details of drawing menus, monitoring the mouse movements, and so on and produce higher level events such as menu selections and window scrolls.

One of the most important functions of a toolkit library is handling event dispatch. Toolkits such as Xt and Tk provide an abstraction for event dispatch whereby the application programmer specializes his/her program by defining callback routines. The routines are called when a given widget is activated by the user. For example, a scrollbar widget will call its callback procedure with the new scroll value whenever the scrollbar position is changed.

In Sieve, event handling is implemented using the Xt callback mechanism. The programmer specializes the event detection apparatus by specifying a callback routine and the media processing module uses the Xt dispatch mechanism to signal events by invoking the procedure. The module supplies as arguments the beginning and ending timestamps over which the pattern was observed. The module may also, optionally, supply additional information about the pattern. Thus, the media processing modules act together as *media widgets* which, together with GUI widgets, the application programmer weaves together and configures to build an event-driven program.


Figure 3-13: The HeadHunter application

## 3.4 Example, The HeadHunter

This section presents an example Sieve program that captures and presents video clips which meet specific visual criteria. In particular, this program captures video clips of "talking heads", such as those shown in figure 3-14, which are video clips in which a human face is prominently depicted in the center of the screen. In the case of television video, these clips are typical of, though not restricted to, an anchor person or announcer speaking directly to the camera.


Figure 3-14: Head shots

In addition to processing the video, the program simultaneously interacts with a human user through a graphical user interface. When invoked, the *HeadHunter* application brings up the window shown in figure 3-13. The central screen is used to display the incoming stream of video (see figure 3-15 for a diagram of the layout). Initially, the two video lists to the left and right of the screen are empty. As the program "recognizes" video clips matching the desired criteria, window entries depicting each clip appear in the "candidate" video list to the left of the screen. The user may then accept or reject candidates by selecting them from the candidate list (by clicking on them) and pressing the "Save" or "Remove" buttons. The remove button deletes the entry while the "Save" button moves it from the candidate video list to the "accepted" video list. Finally, the user may use the "Control Panel" and "Program" buttons to bring up additional windows used to dynamically adjust the processing modules' parameters.

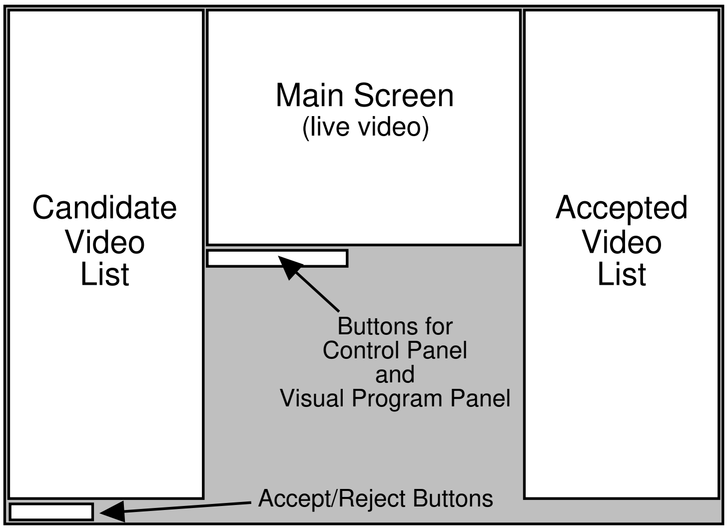



Figure 3-15: HeadHunter layout

While the purpose of this program is more pedagogical than utilitarian, one may imagine such an application being used by an editor or producer putting together a summary graphic depicting the contributions of the various reporters. In addition, a more general version of this program serves as the basis for the VsGrep application describe in section 7.1.

### 3.4.1 Implementation of the user interface

The user interface consists of a top level window which contains seven elements: a central viewing area, two "video lists", and four buttons.

The interface is implemented by the `HeadHunterGui` procedure. `HeadHunterGui` takes as arguments the name to use for the top level widget and the name for the top level data flow module. `HeadHunterGui` proceeds to create a top level form widget and, within that widget, to create and configure the video lists, screen, and commands (i.e. buttons) shown in figure 3-15. Finally, `HeadHunterGui` calls the procedure `HeadHunterFlow` which sets up the media processing flow graph for the application. The code for `HeadHunterGui` is shown in figure 3-16.

The video lists are instances of the Tcl class `VsVideoList`. A `VsVideoList` object is a scrollable viewing areas which contains a dynamic list of entries. Each entry consists of a label and a window for representing a video file. The `VsVideoList` object implements

```
proc HeadHunterGui {w m args} {
  VsEntity $m
  $m set w $w
  apply Form $w $args

  #Create left hand video list with save and remove buttons
  VsVideoList $w.inEntries $m.inEntries
  Command $w.add \
    -label "Save" \
    -callback "SaveSelection $m.inEntries" \
    -fromVert $w.inEntries
  Command $w.sub \
    -label "Remove" \
    -callback "RemoveSelection $m.inEntries" \
    -fromHoriz $w.add \
    -fromVert $w.inEntries

  #Create central screen with control panel and program buttons
  VsScreen $w.screen \
    -resizable true \
    -fromHoriz $w.inEntries
  Command $w.controlPanel \
    -label "Control Panel" \
    -callback "VsPanelShell $w.controlPanel.shell -obj $m" \
    -fromHoriz $w.inEntries \
    -fromVert $w.screen
  Command $w.visualPanel \
    -label "Program" \
    -callback "VsVisualShell $w.visualPanel.shell -obj $m.flow" \
    -fromHoriz $w.controlPanel \
    -fromVert $w.screen

  #Create right hand video list for saved clips
  VsVideoList $w.outEntries $m.outEntries \
    -fromHoriz $w.screen

  #Set up flow graph for video processing
  HeadHunterFlow $w $m.flow
}
```

Figure 3-16: Procedure to set up HeadHunter graphical user interface

the methods `addEntry` and `deleteEntry` which create and destroy entries, causing the chain list to grow and shrink respectively. A user may interactively select an entry in the `VsVideoList` by clicking on its window or label. The program may query the `VsVideoList` object for the identity of the current selection at any time. For example, clicking on the "Remove" button below the candidate video list causes the `RemoveSelection` procedure to be called. `RemoveSelection` queries the candidate video list object for its current selection and, if one exists, deletes the file associated with that entry and removes the entry from the list. The other button located below the candidate video list is labeled "Save". The "Save" button calls the `SaveSelection` procedure which effectively moves the selected entry in the candidate video list to the accepted video list. The code for `SaveSelection` and `RemoveSelection` is shown in figure 3-18.

The `VsScreen` portion of the procedure creates the live display window and two button widgets. The control panel button brings up a compound panel which contains the control panel for each of the data flow modules in the program. These panels allow the user to dynamically adjust a module's parameters. For example, figure 3-17 shows the control panel for the `VsFaceDetect` module. The program button brings up the visual programming environment, called Paves [Wetherall, 1994] which allows the user to dynamically reconfigure

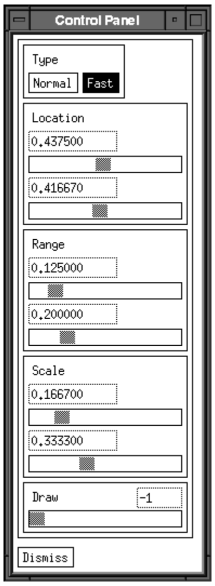



Figure 3-17: VsFaceDetect control panel

```
proc SaveSelection {entryList} {
  set entry [$entryList current]
  if {[info commands $entry] != {}} {
    set newFile "saved.[getNextFileNumber . saved.].uv"
    exec /usr/bin/mv [$entry pathname] $newFile
    $entryList deleteEntry $entry
    [[file rootname $entryList].outEntries addEntry $newFile] expose
  }}
proc RemoveSelection {entryList} {
  set entry [$entryList current]
  if {[info commands $entry] != {}} {
    exec /usr/bin/rm [$entry pathname]
    $entryList deleteEntry $entry
  }}
```

Figure 3-18: Auxiliary procedures to implement `Save` and `Remove`

an application's data flow by graphically creating, destroying, and connecting the processing modules. Figure 3-19 shows the visual programming panel for the HeadHunter application.

### 3.4.2 The flow graph

The data flow for the incoming video stream is graphically depicted in the visual programming panel in figure 3-19. The flow graph is created by the `HeadHunterFlow` procedure shown in figure 3-22.

As shown by the visual panel and the code, the video data flows from a source module to a `VsDup`. The `VsDup` makes two copies of the video stream, sending one to a `VsWindowSink`, which is linked to the live display window, and the other to the analyzing pipeline.

Data analysis is performed by the `VsFaceDetect`, `VsDerive`, and `VsAutomata` modules. `VsFaceDetect`, described in section 4.1.5, attaches the properties `face.top`, `face.bottom`,

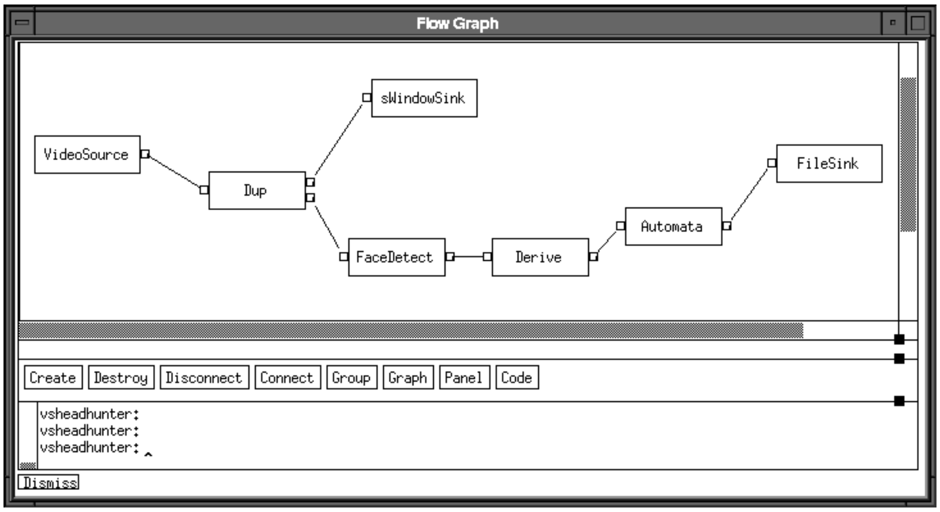



Figure 3-19: HeadHunter data flow

`face.left`, and `face.right` to a payload if a "face" is detected. The `VsDerive` module (see section 5.2) converts these properties to the boolean valued `anchor` property using the predicate function `isAnchor`. Shown in figure 3-21, `isAnchor` tests to see if the position and size of the face is indicative of a news anchor camera shot. Finally, `VsAutomata` implements the five state non-deterministic finite state automata depicted in figure 3-20.

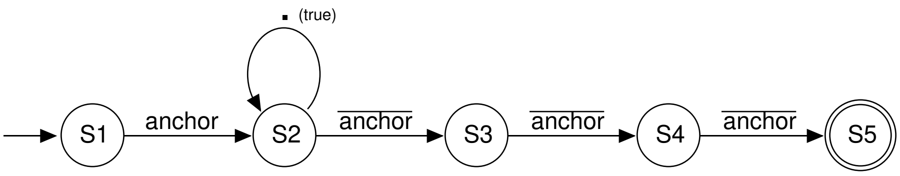



Figure 3-20: HeadHunter automata

```
proc isAnchor {left top right bottom} {
  return [expr {
    $left > .35 && $right  < .65 &&
    $top  > .20 && $bottom < .65 &&
    $right-$left > .15 && $bottom-$top > .2}]
}
```

Figure 3-21: The isAnchor predicate

The pattern recognized by the automata is any sequence of payloads that starts with an image for which the `anchor` property is true and ends with three consecutive images for which the `anchor` property is false. The effect of requiring three non-faces to be detected, rather than one, is to prevent one or two misclassified images from breaking a single video clip into many. Non-deterministic finite state automata are discussed in chapter 6 while the implementation of `VsAutomata` is presented in section 6.2.

The automata module used in this application is buffered. Upon matching an input, the module acts by calling the `FoundOne` procedure and sending the payloads which comprised

```
proc HeadHunterFlow {w m args} {
  global videoSource; VsEntity $m

  #Create a VideoSource, duplicate the output, and display one copy in window
  VsVideoSource $m.source \
    -videoSource $videoSource
  VsDup $m.dup \
    -numOutputPorts 2 \
    -input "bind $m.source.output"
  VsWindowSink $m.sink \
    -input "bind $m.dup.output0" \
    -widget $w.screen

  #Generate face property
  VsFaceDetect $m.face \
    -type fast_har \
    -scale {.16667 .3333} \
    -inputRect {.25 .05 .75 .80} \
    -property face \
    -input "bind $m.dup.output1"

  #Compute anchor and . predicates (. means true)
  VsDerive $m.deriv \
    -input "bind $m.face.output"
  $m.deriv derive {. int 1}
  $m.deriv derive {anchor int {[isAnchor [prop.f face.left] [prop.f face.top]
                                [prop.f face.right] [prop.f face.bottom]]}}
  #Setup automata pattern matcher
  VsAutomata $m.auto \
    -buffer 1 \
    -input "bind $m.deriv.output"
  set state5 [$m.auto makeState {} {}]
  set state4 [$m.auto makeState {} [list [list ^anchor $state5]]]
  set state3 [$m.auto makeState {} [list [list ^anchor $state4]]]
  set state2 [$m.auto makeState {} [list [list ^anchor $state3]]]
  set state1 [$m.auto makeState {} [list [list anchor $state2]]]
  $m.auto addStateTransitions $state2 [list . $state2]
  $m.auto stateAction $state5 "FoundOne $m"
  $m.auto startStates [list $state1]

  #Send matching video clips to a file sink
  VsFileSink $m.filesink \
    -callback "SinkCallback $m" \
    -input "bind $m.auto.output"
}
```

Figure 3-22: Procedure to set up HeadHunter data flow

the pattern to the `VsFileSink` module. The stream of payloads sent by the automata is terminated with a `VsFinish` payload. The arrival of the `VsFinish` at the file sink causes the module to call its Tcl callback procedure, `SinkCallback`. `SinkCallback` closes the file and creates an entry depicting the file in the candidate entry list. The code for `FoundOne` and `SinkCallback` is presented in figure 3-23.

An alternative to hand generating the analytic pipeline and finite state graph is to use `VsVex`. `VsVex` is an higher level facility that allows the programmer to concisely specify the properties, predicates, and patterns that are to be generated and analyzed. Of particular value is the regular expression facility that allows the programmer to specify the finite state graph as a compact expression. Figure 3-24 shows the alternative definition of the analytic portion the HeadHunter application. The `VsVex` code may be used as a replacement for the lines of code between the `VsWindowSink` and the `VsFileSink` in `HeadHunterFlow`.

```
# When we find one send the payload buffer to the file sink
proc FoundOne {self args} {
  $self.auto send $args
  $self.auto clear
  set path "candidate.[getNextFileNumber . candidate.].uv"
  $self.filesink pathname $path
  $self.filesink start}

# When file sink is finished, close the file and add to video list
proc SinkCallback {self args} {
  if [keyarg -sinkFinish $args 0] then {
    set pathname [$self.filesink pathname]
    set entryList [file root $self].inEntries
    $self.filesink pathname ""
    [$entryList addEntry $pathname] expose
  }}
```

Figure 3-23: Callback procedures for automata and file sink

One cosmetic difference between the two versions of the program is that the name of the property that represents the `isAnchor` predicate has been changed from `anchor` to "`A`". This has been done for compatibility with the regular expression parser which treats each character in the regular pattern as a separate property. There is no fundamental reason for this limitation. The restriction, which is limited to expressions parsed by the `RegMatch` procedure, is simply a convention that allows for a cleaner, more compact syntax. The `RegMatch` procedure also implements a second convention that likewise, improves the readability. In particular, lower case characters are reserved to represent the complement of uppercase properties. Thus, the character "`a`" in a regular expression matches inputs where the property "`A`" is false. These conventions allow the expression "A.*aaa" to represent the finite state automata of figure 3-20 [5].

```
VsVex $m.auto \
  -input "bind $m.dup.output1" \
  -buffer 1 \
  -properties {
    {VsFaceDetect -property face -type fast_har -scale {.16667 .3333}
      -inputRect {.25 .05 .75 .80}}} \
  -predicates {
    {. int 1}
    {A int {[isAnchor [prop.f face.left] [prop.f face.top]\
             [prop.f face.right] [prop.f face.bottom]]}}}\
  -patterns [list [list RegMatch "A.*aaa" "FoundOne $m"]]
```

Figure 3-24: Short hand implementation

### 3.4.3 Invocation

For the sake of completeness, the `main` procedure is shown in figure 3-25. `main` parses the command line, initializes the application libraries, creates a top level shell object, and

[5] Actually, the state machine implemented by the code in figure 3-22 is slightly different than that implemented by the code in figure 3-24, though the output is the same. Figure 3-20 depicts the simpler state machine generated by the handwritten code . The state machine automatically generated from the regular expression contains a few redundant states.

catches and reports errors. Thus, code from figures 3-16, 3-18, 3-21, 3-22, 3-23, and 3-25 comprise the complete script for the `HeadHunter` application.

Although some aspects of the Tcl code appear baroque, the complete script[6], including blank lines and comments, is only 130 lines. Of the 99 lines with code[7], 43 implement the user interface and 16 handle the invocation – leaving just 40 lines of code to implement the flow graph and its related procedures (that is `HeadHunterFlow`, `isAnchor`, `FoundOne`, and `SinkCallback`).

```
proc main {} {
  global argv name class errorInfo videoSource

  set name [lindex $argv 0]
  set videoSource [lindex $argv 1]

  xt appInitialize app_context "HeadHunter" argv {}
  vs appInitialize app_context vs

  VsShell $name.top \
    -title HeadHunter \
    -cmd HeadHunterGui \
    -realize "vs start" \
    -dismiss "catch {vs destroy}; exit" \
    -args vs.hh \
    -allowShellResize true

  while {[catch {app_context mainLoop} msg]} {
    VsErrorShell $name.err -summary $msg -detail $errorInfo
  }
}
main
```

Figure 3-25: The main procedure for the HeadHunter application

## 3.5 Summary

This chapter has described the archtectural framework of a Sieve application. The reader now should understand how a Sieve application is organized and have a sense of the scripting language. The next three chapters will detail the analytical components of the Sieve toolkit. Chapter 4 describes the image processing modules that generate properties. Chapter 5 describes ways in which these properties are used to dynamically adjust the system. Chapter 6 then describes how properties are ultimately used to generate symbolic media events.

[6] The VsVex version of the script

[7] Lines containing only brackets are counted as blank. The only purpose for putting a bracket on its own line is cosmetic.

# Chapter 4

# Generating Properties

In order to make decisions based on the content of a video stream, a system requires a computational means of examining the data. Sieve employs a library of low level image and video processing modules which transform the video data and measure its properties. This chapter details those modules.

The property centric approach raises many issues. Should properties be based on two dimensional image characteristics or on three dimensional scene characteristics? How can complex image or scene characteristics be summarized into concise descriptions? What techniques are applicable to building desktop applications? There are no definitive answers to these questions, and so, Sieve is designed to support a wide range of approaches. However, in the course of experimenting with building content analyzing applications, a strategy for generating properties has emerged.

Essentially, this thesis focuses on two complementary areas from computer vision: matching and motion. Matching is concerned with determining whether two inputs are the same while motion is concerned with detecting and measuring change.

Matching is used to tell the system *what* is the identity of the focus of attention. Using the appropriate transformations and representations, inputs may be matched to models in order to generate properties. The property's value represents the degree to which the transformed input matches the model while its name identifies the model to which the input is matched. It is unnecessary, therefore, for the properties themselves to provide rich descriptions of the information in an image. Rather, the complexity of the information is captured in the named models. Typically, the user is aware of the meaning and significance of the models and may instruct the system to interpret a particular match in the appropriate way.

Motion (or more generally change) is used primarily to focus or direct the attention of the system. Properties summarizing detected motion are used to temporally focus the system, telling it *when* interesting events take place. Transformations of images into motion fields spatially focus the system, telling it *where* interesting objects are located. Such information is often used to generate segmentations, represented as binary images, from which geometric and set properties may be derived.

The remainder of this chapter is divided into five sections. Section 4.1 discusses methods for matching and presents modules which implement many of these techniques. Section 4.2

focuses on data transformations. While computationally the transformations are performed prior to matching, they are motivated by the matching operations, and therefore are discussed after matching. Section 4.3 focuses on measuring motion and change. Technically motion detection is a type of data transformation, but its role is of such importance that it is treated separately. Section 4.4 presents several modules which analyze and manipulate the binary images produced by modules, such as the motion detectors, which identify regions of interest in images. Finally, section 4.5 considers alternative modules and filters.

## 4.1 Matching

The problem of matching is to determine whether one input is "the same" as another. Using the notion of sameness, one may generate properties by measuring the degree of sameness of the input to different models (figure 4-1).

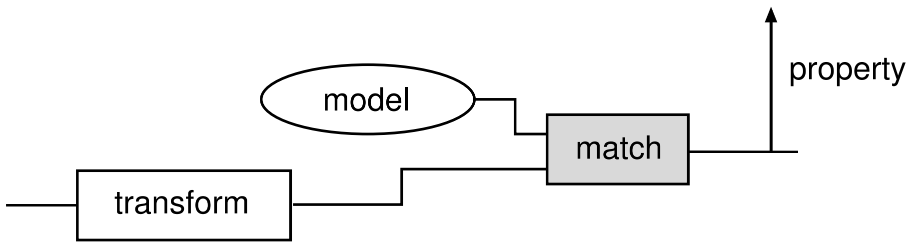



Figure 4-1: Matching models to data

Of course, the notion of sameness differs from application to application. The challenge of matching is to transform the input into a representation which is invariant over the range of inputs which one wishes to match, while discriminating between inputs which are intended to be different.

Another challenge related to matching is determining what to match. Often, a model represents only a subset of the input data. For instance, when processing images, a model may be expected to match only a portion of the image. In such cases, a matching algorithm must not only be able to judge the similarity of two inputs, but also be able to select the portion of the input to match against the model. Sometimes additional information (such as that generated from motion analysis) may be used to focus the system on the portion of the image that should be compared. Frequently, however, the matching algorithm must search for the subset of the input to compare against the model. Such a matcher typically produces as output the measure of similarity between the model and the best matching subset of the input data as well the parameters identifying the subset from which the measure was taken.

This section describes different methods for measuring sameness and the implementation of modules which implement these methods. First, direct pixel matching is presented. Though the technique itself is simple, the discussion clarifies the roll of matching in Sieve. In addition, direct matching serves as a convenient place to discuss how color and gray scale images are represented. Next, histogram matching is discussed as an example of statistical profile matching. The third subsection describes two-dimensional feature matching and presents the Hausdorff match as an example. The fourth subsection outlines the potential roll for geometric and three dimensional matching. Finally, the fifth subsection discusses example-based matching and presents a face detection algorithm based on this method.

### 4.1.1 Direct Matching

Many different techniques may be used to obtain a measure of how closely two sets of pixels match. The most straightforward means is a pixel by pixel comparison between the sets. It is assumed that the two sets of pixels, A and B, comprise identically shaped regions and that pairwise comparison of pixels $a_i$ and $b_i$ are made between pixels in the same relative location. While it may be possible, under special conditions, to use pixel matching on natural objects in real images, the main use of pixel matching presented in this report is for matching graphics found in television video.

A useful way of representing a wide class of pixel by pixel comparisons is by defining a distance metric between pairs of pixels, d = $\|p_1 \ominus p_2\|$, and a function, F, which represents the distance between two sets of pixels, A and B, as follows:

$$A = \{a_1, a_2, ...a_n\} \quad B = \{b_1, b_2, ...b_n\} \tag{4.1}$$

$$D = \{d_i \mid d_i = \|a_i \ominus b_i\|\} \tag{4.2}$$

$$F(D) = \|A \ominus B\| \tag{4.3}$$

Gray scale pixels represent brightness in the image plane as a scalar quantity. Therefore, a gray scale pixel metric is a function defined over scalar values. Some commonly used gray pixel metrics include the square difference, absolute difference, and absolute difference of logs.

Color pixels represent lightness in the image plane as a vector quantity. A discussion of different color representations is presented in section 4.2.1. Regardless of color model that is employed, a distance metric for color pixels is a function that takes two vectors as input and produces a positive scalar value as output. A proper metric must also treat the two inputs symmetrically, return a positive number if they are different, return 0 if they are the same, and obey the triangle inequality [Therrien, 1989]. Candidate metrics include those based on the distance between the two vectors:

$$euclidean = \sqrt{(x_a - x_b)^2 + (y_a - y_b)^2 + (z_a - z_b)^2} \tag{4.4}$$

$$city\_block = |x_a - x_b| + |y_a - y_b| + |z_a - z_b| \tag{4.5}$$

$$maximum\_component = \max(|x_a - x_b|, |y_a - y_b|, |z_a - z_b|) \tag{4.6}$$

and those based on the relative direction or angle, $\theta$, between the two vectors:

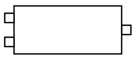
$$1 - \cos(\theta) = 1 - \frac{\vec{a} \cdot \vec{b}}{|\vec{a}||\vec{b}|} \tag{4.7}$$

$$\sin(\theta) = \frac{|\vec{a} \otimes \vec{b}|}{|\vec{a}||\vec{b}|} \tag{4.8}$$

The angle based measures, advocated by [Sung, 1992] and [Subirana and Sung, 1992], offer the advantage they are based on the relative weights of the vector components so that when applied to a tristimulus color space they are insensitive to absolute intensity differences.

Once pixel values have been converted to distance measures, one may treat the accumulation of the distance measures into a single measure in a uniform manner. In this case, we are interested in functions which treat the distance measures as an unordered set of values. Appropriate candidates include the sum of the distances (typically scaled by the number of inputs), the product, the minimum, the maximum, the median, and the k-th rank. Another useful accumulator is simply the number of distance inputs which are greater than a given threshold.

### VsPixelMatch



The `VsPixelMatch` module takes as input two image streams. By convention, one of these streams is labeled the `input` stream and the other the `model` stream. The module searches for the translation of the model with respect to the input which results in the minimum distance as measured by a direct pixel match. The module produces as output a sequence of properties which it attaches to the images in the input stream. These properties indicate the minimum detected measure and the region of points in the input image that yielded that measure.

The `VsPixelMatch` module may be used in one of two ways: as a synchronized comparator of two image streams or as a template matcher. When used as the former, the two input streams are compared frame by frame and the input stream, including the attached properties, is passed through while the model stream is consumed and discarded. When used as the latter, the model stream must be explicitly stepped by the programmer so that once an image is loaded, the module behaves like a model or template matching filter whereby each frame in the input stream is compared against the "held" model template. Figure 4-2 shows the `VsPixelMatch` module being used as a component in a template matching filter.

Presently, `VsPixelMatch` is restricted to matching rectangular regions of points. One may select a subset of the model image to serve as the model point set using the `modelRect` parameter. The value of this parameter is a list of four numbers specifying the location of the left, top, right, and bottom edges of the model. Each edge is represented as a floating point number between 0 and 1 which represents the relative coordinate as a fraction of the model image's width and height. Similarly the `inputRect` parameter specifies the rectangular region of points in the image that is searched for a match. A value of {0.0 0.0 1.0 1.0}, for example, would specify for the full image to be searched whereas a value of {0.5 0.0 1.0 0.5} would specify for only the upper right-hand quadrant to be used.

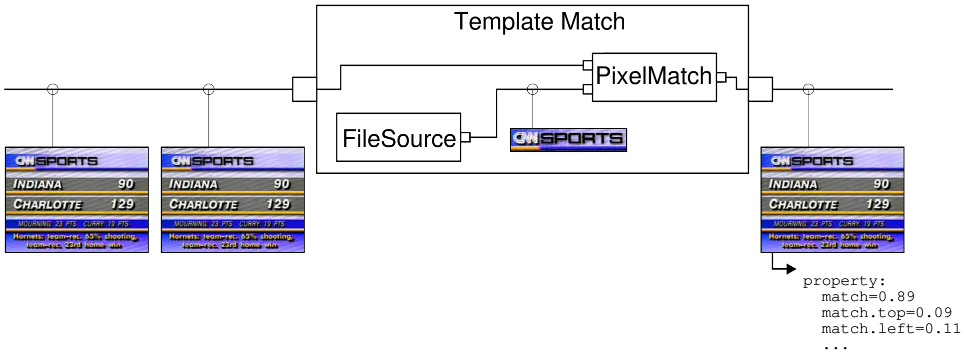



Figure 4-2: VsPixelMatch used in a template matching filter

`VsPixelMatch` implements a variety of comparison methods which may be described by a pixel distance function and an accumulation function. The pixel metric is specified using the `vectorMethod` and the `scalarMethod` parameters, which respectively specify the method used for comparing color pixels and gray pixels. Table 4.1 summarizes the allowable values for these parameters. The only accumulation function currently implemented is the sum of the distances.

| Name | Function |
|---|---|
| scalarMethod squareDiff | $(a-b)^2$ |
| scalarMethod absDiff | $\|a-b\|$ |
| vectorMethod euclidean | Equation 4.4 |
| vectorMethod cityBlock | Equation 4.5 |
| vectorMethod maxDiff | Equation 4.6 |
| vectorMethod dotProduct | Equation 4.7 |

Table 4.1: Distance methods

The output of `VsPixelMatch` is a set of five properties which are attached to the input image. The base name for these properties is specified by the `property` parameter. A parameter value of `match`, for example, causes `VsPixelMatch` to attach the properties `match`, `match.top`, `match.bottom`, `match.left`, and `match.right`. The values of these properties are, respectively, the minimal distance measure and the location of the four edges of the matching region.

The parameters for `VsPixelMatch` are summarized in the following table:

| Parameter | Value | |
|---|---|---|
| model | *input_port* | Model input port |
| input | *input_port* | Input port |
| output | *output_port* | Output port |
| scalarMethod | {absDiff squareDiff} | Default is absDiff |
| vectorMethod | {cityblock euclidean maxDiff dotProduct} | Default is cityblock |
| inputRect | *float float float float* | Search rectangle: left top right bottom |
| modelRect | *float float float float* | Model rectangle: left top right bottom |
| property | *string* | Default is no property |
| hold | {0 1} | Default is 1 |

VsPixelMatch searches for the translation between the model and the input that results in the minimum distance This idea can be extended to account for scale and rotational variations by scaling and/or rotating one image region with respect to the other and searching for the best fit. Indeed, any variation which can be parameterized may be accounted for in such a way. The drawback of this approach is computational cost. Every parameter which is searched adds a multiplicative factor to the amount of computation so that it is important to limit the range and frequency over which a parameter is searched.

One should note that not all distance measures between sets of pixels, not even those which are essentially pixel by pixel comparisons, may be represented as a pixel distance function and a distance set accumulator. An important example is a matched filter [Rosenfeld, 1969]. A matched filter for two image patches is described by the equation:

$$\frac{\sum_{i=1}^{n} a_i b_i}{\sqrt{\sum_{i=1}^{n} a_i^2}\sqrt{\sum_{i=1}^{n} bi^2}} \tag{4.9}$$

The matched filter accumulates the product of the pixels values in the two regions and scales the sum by a factor that is dependent on the value of the pixels in the region. One can define a distance metric as 1 minus the output of the matched filter. Unfortunately, the approach put forth in this section provides no means of calculating the scale factor. It would not be difficult to generalize the approach but there really is no pressing need to do so. The VsPixelMatch filter, described below, implements a variety of matchers supported by the existing scheme. While mathematically elegant, the matched filter does not appear to offer significant practical advantages over metrics such as the sum of the absolute or Euclidean differences. Still, were there a need to provide such functionality, it would not be difficult to add by either extending VsPixelMatch or by implementing it as a new module.

The more significant limitation that is shared by all the direct matching schemes described in this section, including the matched filter, is that they fail to account for important properties between pixels within the regions. Other than the fact that these comparison methods require that the spatial organization of the two regions be the same, there is no use of structural information. As a result, these types of matchers are extremely sensitive to alignment, scale, rotation, deformation, and so on.

### 4.1.2 Statistical Matching

Another means of measuring the similarity between two sets of pixels is to compute a statistical profile which characterizes the sets of pixels and to compare the two profiles.

One of the most straightforward statistics to compute over a set of pixels is the histogram of their values. A histogram represents, for each gray or color range in the image, the number of pixels in the image whose value falls within that range (figure 4-3). Typically, the ranges, referred to as bins, are non-overlapping and collectively exhaustive. In the case of gray scale images, each bin represents a one-dimensional range of intensities while the bins for color histograms are multi-dimensional.

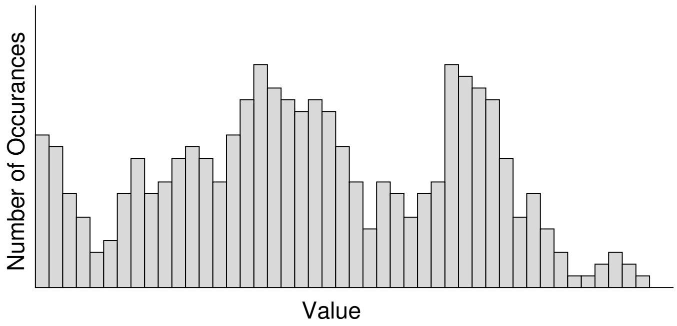



Figure 4-3: One dimensional histogram

Histograms have long found wide spread use in image processing as a basis for normalizing image intensities (histogram modification), and selecting thresholds [Pratt, 1991], [Rosenfeld, 1969]. More recently, histograms have gained popularity in image analysis applications as a tool for matching and indexing images [Swain, 1990] [Swain and Ballard, 1991] [Funt and Finlayson, 1995]. In such applications, a histogram serves as a signature which may be used to identify an object.

While technically, a pixel map of the object is also a signature of the object, a histogram signature is better suited because of its invariant properties. Histograms of deformable objects are frequently invariant to the types of deformations that the object undertakes. For instance, the histogram of the frontal view of a person gesticulating their arms and changing their expression remains relatively constant even though the spatial relationships between pixel locations varies wildly. Furthermore, it is often the case that histogram signatures remain fortuitously stable even when the two dimensional point of view to the object changes because the color components of many three dimensional objects often remains relatively constant across different views.

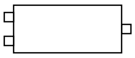
The histogram representation serves as a natural way to organize the information in an image. In particular, the pixels are grouped and sorted by value so that like-valued pixels are placed in the same bin and near-valued pixels are placed in nearby bins. As a result, alignment for matching is not an issue. Furthermore, the placement of near-valued pixels in nearby bins makes histograms robust for matching in the presence of uncorrelated imaging noise. Finally, histograms are easily made scale invariant by simply normalizing the bin counts by the total number of pixels. Thus, each bin value represents the percentage of pixel in the object that fall within its color range.

Another important advantage of histogram matching is computational performance. Histograms are a compact representation that are efficiently generated and efficiently compared. Moreover, when searching for a region that matches a fixed model, the model histogram need only be computed once. More important, however, is the mathematical property that the histogram of a region is the sum of the histograms of the sub-regions. This property allows for an efficient implementation of the search algorithm whereby histograms for overlapping regions need not be computed from scratch. The difference in performance, when compared to direct matching, is dramatically illustrated in the performance results of chapter 8.

Unfortunately, histograms are not invariant to changes in lighting, though their robust nature helps reduce its negative effects. One approach to dealing with this problem is to transform the input to a more lighting invariant form. As was the case with pixel matching, the `VsColorTransform` module has been used for this purpose.

The method used to compare histograms may be described as an accumulation of differences between corresponding bins. The limitation of such bin to bin comparisons is that they do not account for the relationships between non-overlapping bins. For example, in the extreme case of comparing two histograms with absolutely no overlap, such a comparison fails to account for whether the bins are near each other or are far apart. Measures that could be taken to reduce such effects are smoothing the histogram or accumulating the bin values differently. For example, one may model each bin as the accumulation of responses of each pixel to a narrow band filter. In practice, however, such measures tend to be unnecessary for real images, since noise in the image processes serves to distribute pixel values.

### VsHistMatch



The `VsHistMatch` module performs a function similar to that of `VsPixelMatch` in that it searches for the translation of a model region within an input image that results in the minimum distance measurement. Like `VsPixelMatch`, the module takes as input two image streams, one labeled `input` and the other labeled `model`, and produces as output a set of properties which it attaches to the images in the input stream. The modules differ in the way that model and input regions are compared and, therefore, in several of the parameters related to the comparisons.

`VsHistMatch` compares pairs of regions by computing histograms for the two regions and measuring the difference between the histograms. Histograms for a gray scale images are represented as one dimensional arrays of bins. The number of bins is determined by the `grayBits` parameter which specifies the number of higher order bits that are used to determine bin assignment. For example, a `grayBits` value of 6 specifies that each gray scale pixel be assigned to one of $2^6$ bins on the basis of its 6 most significant bits. Histograms for color images are represented as three dimensional arrays of bins. The `colorBits` parameter is a list of three values which independently specify the number of higher order bits used for each of the three color dimensions.

`VsHistMatch` supports two methods for comparing histograms: sum of the absolute differences between the bin counts and sum of the square differences between the bin counts. In both cases, the bin values are first normalized by dividing them by the total number of pixels represented in the histogram. The normalized bins, therefore, represent the percentage of image pixels in each bin. The method employed is specified by the `binMethod` parameter. The value returned by either of these methods is a number between 0 and 2, with 0 being the value returned when comparing two identical histograms. The value 2 is returned by the absolute difference method if the two histograms have absolutely no overlap. In the case of the square difference method, the value returned is 2 if and only if all the pixels for both histograms falls into exactly two distinct bins.

`VsHistMatch` treats the `property`, `imageRect`, `modelRect`, and `hold` parameters identically to `VsPixelMatch`. This common treatment facilitates the implementation of higher level modules such as `VsTemplate` (see section 5.2) which take the identity of a matching module as an input parameter.

The parameters for `VsHistMatch` are summarized as follows:

| Parameter | Value | |
|---|---|---|
| `model` | *input_port* | Model input port |
| `input` | *input_port* | Input port |
| `output` | *output_port* | Output port |
| `inputRect` | *float float float float* | Search rectangle: left top right bottom |
| `modelRect` | *float float float float* | Model rectangle: left top right bottom |
| `grayBits` | *int* | Default is 8 |
| `colorBits` | *int int int* | Default is {3 3 3} |
| `binMethod` | {`absDiff sqDiff`} | Default is `absDiff` |
| `property` | *string* | Default is no property |
| `hold` | {`0 1`} | Default is 1 |

The cost of using a histogram representation is a loss of information which can allow dissimilar regions to match. In particular, the color makeup of an object is preserved but the spatial relationship between the different colors is lost. For example, the three different images in Figure 4-4 share the same histogram.

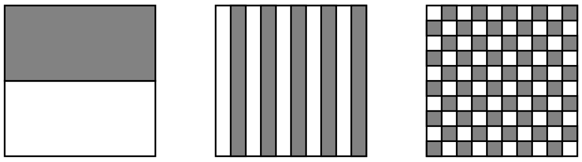


Figure 4-4: Three different images with the same histograms

Some of these problems may be addressed by considering other statistical measures. For example, in the case illustrated, a histogram of a statistic based on local variation, such as the Laplacian (figure 4-5), would distinguish the two images. In general, any local feature which may be quantized and counted may be used to generate a histogram-like statistical profile which may in turn be used to characterize an image.

Many such features, often referred to as textures, have been proposed. [Tamura *et al.*, 1978], for example, presents statistical measures which correspond to visual concepts such as coarseness, contrast, and directionality. These features were employed by Niblack in the

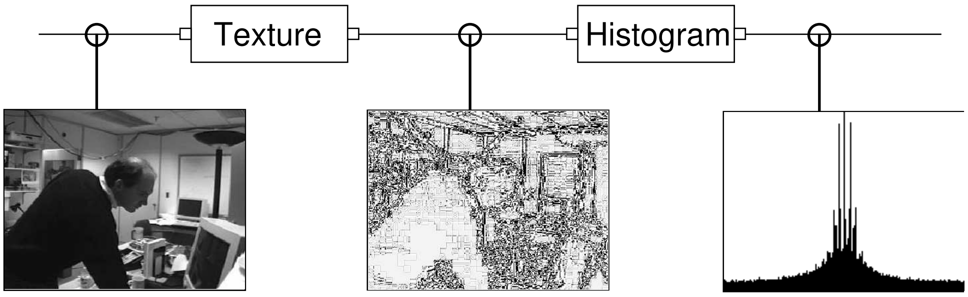



Figure 4-5: Histogram of the Laplacian

QBIC project [Niblack *et al.*, 1993] as a basis for querying an image database by content. More recent work presented by Picard and Minka [Picard, 1996] [Picard and Minka, 1995] characterizes images using multiple texture models based on first and second order pixel statistics.

### 4.1.3 Feature Matching (2D)

A different approach to matching two dimensional regions of pixels is to transform the pixels into a map of features and to perform matching based on the relative locations of such features. For example, a commonly used image feature is edge locations. Edge locations may be loosely defined as abrupt changes in pixel value. The principle advantage of comparing features is that their measurement may, in some cases, be made invariant to different conditions. For example, the existence and location of an edge in an image is relatively insensitive to lighting conditions.

A difficulty in matching feature locations is dealing with perturbations in feature locations. Direct matching, for instance, fails if binary feature locations are skewed by distances of even a pixel. Other methods for comparing point sets are better suited. One such method developed by Huttenlocher and Rucklidge [1992] and implemented in Sieve, is based on the Hausdorff distance. This method calculates the distance between two sets of points on the basis of the Euclidean distances of points in the model set to their nearest counterpart in the image set (and visa versa). The Hausdorff distance measure is the maximum such Euclidean distance.

Formally, given two sets of points, one representing a model, $M = m_1, m_2, ...m_{|M|}$, and the other an image, $I = i_1, i_2, ...i_{|I|}$; the Hausdorff distance, H(M,I), between the two point sets is given by the formula:

$$H(M, I) = max(h(M, I), h(I, M)) \tag{4.10}$$

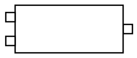
where h(M,I) is defined as the largest distance from any point in M to the closest point in I:

$$h(M, I) = \max_{m \in M} \min_{i \in I} \|m - i\| \tag{4.11}$$

$h(M, I)$ is referred to as the *forward* distance while the converse, $h(I, M)$, is referred to as the *reverse* distance.

Huttenlocher and Rucklidge go on to define the partial Hausdorff distance measure as the k-th ranked Euclidean distance[1] and then calculate the best or minimal such value across different translations and scales.

Huttenlocher, Noh, and Rucklidge [1992] demonstrated the robustness and efficiency of the Hausdorff technique by using it to build a real time tracker on a modest workstation. Their code is the basis for Sieve's `VsHausdorff` module.

### VsHausdorff 

The `VsHausdorff` (figure 4-6) module employs the Hausdorff distance metric to locate a model in an input image. If the model is found, the module tags the image with a set of properties indicating the location where the model appears and an additional property indicating the quality of the match.

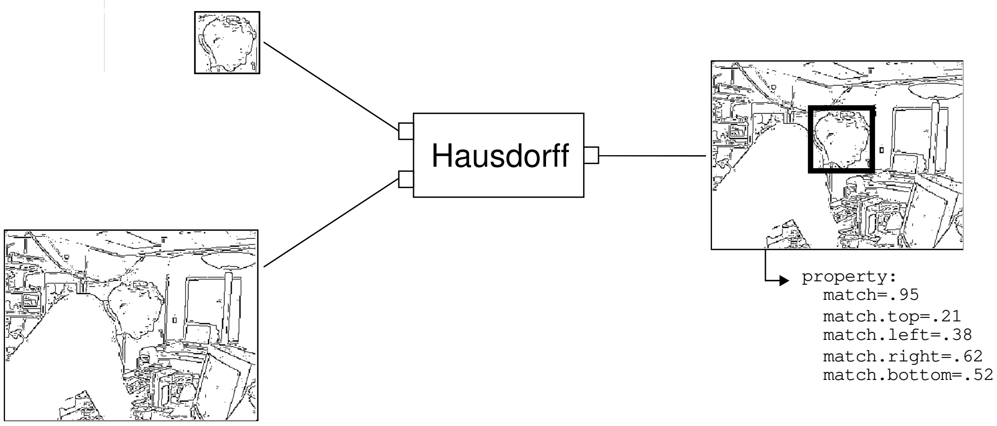



Figure 4-6: VsHausdorff

`VsHausdorff` implements an interface that is very similar to that of `VsHistMatch` and `VsPixelMatch`. One difference is that the `model` and `input` payloads are expected to be binary images. The true pixels in these images represent the model and input point sets.

The module searches for translations of the model with respect to the input image which result in a distance measure that matches the specified criteria. The criteria is defined by the `threshold` and `fraction` parameters. Both parameters are specified as pairs in which the first value specifies the forward value (threshold or fraction) and the second value specifies the reverse value. The forward fraction refers to the fraction of model points which must be within a distance of the forward threshold (measured in pixels) from an input point. Likewise the reverse fraction and reverse threshold specify the same values for the reverse match.

[1] To be precise, the maximum, k-th ranked Euclidean distance where the max is taken between the forward and reverse measures

If one or more translations results in a match, `VsHausdorff` selects the one with the highest *quality* value. The quality value is computed as the product of the fraction of model points falling within forward threshold pixels of an input point and the fraction of input points falling within reverse threshold pixels of a model point. Thus, the quality value is a number between 0 and 1.

As was the case for `VsPixelMatch` and `VsHistMatch`, the `inputRect` parameter specifies the rectangular region of points that is searched for a match to the model. Likewise, the `modelRect` specifies the rectangular region of points used to represent the model. The property output of `VsHausdorff` also resembles that of the other matching modules. The main difference is that whereas the other modules attach a set of properties for every input image, `VsHausdorff` only adds a property if a translation is found that meets the threshold and fraction criteria.

| `Parameter` | `Value` | |
|---|---|---|
| `model` | *input_port* | Model input port |
| `input` | *input_port* | Input port |
| `output` | *output_port* | Output port |
| `inputRect` | *float float float float* | Search rectangle: left top right bottom |
| `modelRect` | *float float float float* | Model rectangle: left top right bottom |
| `threshold` | *float float* | Forward and reverse thresholds |
| `fraction` | *float float* | Forward and reverse fractions |
| `property` | *string* | Base name for properties |
| `hold` | {`0 1`} | Default is 1 |

### 4.1.4 Geometric Matching (3D)

The matching methods discussed thus far are based on the reflectance and texture properties of objects. While such properties are extremely useful, it is worth noting that there are alternatives. For example, the geometry or shape of an object may be employed. The basic strategy, as described in [Grimson, 1990], is to extract geometric features from the data and to search for globally consistent correspondences between the measured features and the model features. A geometric feature may be any identifiable location on the object such as an edge, corner, planar patch, cylinder, or sphere. One advantage of geometric matching is that the technique applies well to three dimensional models.

Sieve does not currently implement a geometric matching filter, though it does provide several transformation modules which may be used to extract features. `VsEdge`, described in section 4.2.2, is such an example. One reason for the omission is that geometric matching algorithms have high computational requirements which currently limit their applicability for interactive applications. Another reason is that while geometric matching systems are complex, free standing libraries implementing these techniques are scarce, so there is a high implementation cost. Were such libraries to become available, however, one means of integrating geometric matching into Sieve would be as a module similar to `VsPixelMatch` or `VsHistMatch` which compares a stream of inputs to a model, attaching a property to each input indicating the degree to which the input and the model correspond.

### 4.1.5 Example-based Matching

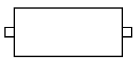
The example-based matching approach involves training a system to classify inputs by providing a number of labeled examples. The examples are used to tune parameters of the decision making algorithm. Once trained, the system acts as a classifier which may be used to label (i.e. attach properties) to new inputs.

Examples and inputs are presented to the system as feature vectors in a multi-dimensional space. Selecting the right features is the most important problem in building an example-based learning system – although the choice of decision making algorithm and training algorithm makes a difference.

One domain where example-based learning has proven to be effective is that of face detection. A face detector attempts to locate the human faces in an image. Sung and Poggio [1994] built a face detector by normalizing two dimensional inputs to a 19x19 pixel window where each pixel location was treated as a dimension in a high dimensional space. A canonical face model was built which attempted to capture the region in the vector space which corresponded to faces. Using this face model, a feature vector based on taking the distance between inputs and the face model was used to train a neural network to classify 19x19 pixel windows as those which did or did not contain a face. Using this classifier, a system was built which searched an image for faces by exhaustively scanning the image for windows at all possible locations and scales.

Sung and Poggio's detector proved to be extremely effective at detecting frontal views of human faces. Part of this success may be attributed the surprising regularity of faces when properly aligned with a low resolution window. Using many of the same ideas, Rowley [1995] built a face detector which again was based on classifying low resolution pixel windows. Rowley's approach differed in that it used the pixels themselves[2] as the feature vector and trained the neural network directly on these vectors. Rowley's detector was also effective and, more importantly, made publicly available as a library (no source code) which could be incorporated into Sieve.

The problem of face recognition is akin to that of face detection, and the use of similar techniques have been explored by many including [Turk, 1991], [Lawrence *et al.*, 1997], and [Beymer, 1996]. However, for several reasons, face recognition is not yet as practical. Fundamentally, recognition is a more difficult classification problem because each input must be classified into one of many categories, whereas detection requires only two categories. In addition, a face detector may be trained on thousands of examples and subsequently used in a wide range of scenarios; whereas a recognizer must be trained for each individual for which it is intended. Indeed, one of the most valuable aspects to Rowley's library is the set of trained parameters which have been compiled into the detector.

**VsFaceDetect** 

`VsFaceDetect` (figure 4-7) implements the face detection algorithm described in [Rowley *et al.*, 1995]. The module is effectively a wrapper around Rowley's library code. It is

[2] Both Rowley and Sung/Poggio preprocess the pixels by performing illumination gradient correction and histogram equalization

implemented as a single input, single output filter which passes a stream of payloads without altering the data. The module acts by attaching properties to any image payload in which it observes one or more faces.

One Face Multiple Faces No Face

Figure 4-7: Faces

The base name for the attached properties is set by `property` parameter. Each detected face results in four properties being attached to the payload which specify the rectangular region of the face. If, for instance, two faces were detected while `property` was set to "head", the eight properties `head.1.left`, `head.1.right`, `head.1.top`, `head.1.bottom`, `head.2.left`, `head.2.right`, `head.2.top`, and `head.2.bottom` would describe the two rectangular regions where the faces were found.

`VsFaceDetect` may be configured to operate in one of two modes. Setting the `type` parameter to "har" [3], selects Rowley's standard detection algorithm which looks for multiple faces at all locations and scales. Setting `type` to "fast_har" instructs the detector to search for faces in a specified sub-region of the image and only at specified scales. In addition, the "fast_har" algorithm terminates once it detects a face so that at most one face location is indicated.

The `inputRect` parameter specifies, in a manner like that of the other matching modules, the rectangular sub-portion of the image to which the detector limits its search. The model employed by `VsFaceDetect` is implicit so there is no `model` input port and no `modelRect` parameter. The `scale` parameter, however, serves a related function by specifying the minimum and maximum fraction of the image that might be covered by a matching face.

The parameters for `VsFaceDetect` are summarized as follows:

| Parameter | Value | |
|---|---|---|
| `input` | *input_port* | Input port |
| `output` | *output_port* | Output port |
| `type` | {`har fast_har`} | Default is `har` |
| `inputRect` | *float float float float* | Search rectangle: left top right bottom |
| `scale` | *float float* | List of two values between 0 and 1 |
| `property` | *string* | Default is no property |

[3] HAR = Henry A. Rowley's initials

## 4.2 Image Transformations

The transformation of sensor data from two dimensional arrays of light intensity measurements into intermediate representations that expose useful information has long been recognized as an important step in visual perception [Marr, 1979] [Ullman, 1984].

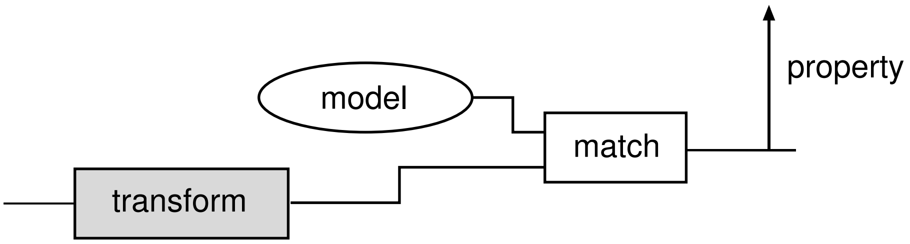



Figure 4-8: Transforming data for matching

This section describes many of the transformation filters which are used by Sieve prior to matching (figure 4-8). For example, Sieve applications frequently use the `VsPixelTransform` module, described in section 4.2.1, to improve the performance of a pixel or histogram matching algorithm. The `VsEdge` module described in section 4.2.2 implements the transformation of a color or gray scale image into a binary representation while `VsHistogram`, described in section 4.2.3, describes the transformation of an image into a histogram. Finally, the `VsConvolve` filter, which convolves an arbitrary stencil with an image is described in section 4.2.4.

### 4.2.1 Color Space Transforms

A color pixel represents lightness in the image plane as a vector quantity. Many vector representations are possible and in practice, many representations are used. Since, the performance of many matching algorithms is highly dependent on the pixel representation, this section concerns itself with transformation from one representation to another. While it is possible to implicitly incorporate a color space transformation into the matching algorithm, it is advantageous from a modularity and re-usability standpoint to separate out the transformation part of the algorithm whenever possible.

The components of a color vector often represent the brightness response of different sensors with varying spectral sensitivities. Because the human visual system itself uses three types of sensors, called cones [Kandel and Schwartz, 1985], it is common to represent colors using a tristimulus model. Examples include C.I.E. standard observer model which represents color as an [r,g,b] vector where the r, g, and b sensitivities have been designed to approximate the spectral response of the three human visual cones, and the more commonly used NTSC/RGB standard which represents color as an [r,g,b] vector where the r, b, and b values are related to the glow of the standard phosphors used in TV sets. Other models include the YIQ, YUV, UVW, XYZ, and rg-by-bw *opponent* color spaces [Pratt, 1991].

The transformations between many of these different models are simply linear transformations which may be represented by a 3x3 coefficient matrix. For example, the following equation describes the transformation from an RGB vector to an *opponent* vector:

$$\begin{bmatrix} RG \\ BY \\ BW \end{bmatrix} = \begin{bmatrix} 1 & -1 & 0 \\ -1 & -1 & 2 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{4.12}$$

Others, such as the transformation from RGB to IHS are non-linear but invertible functions. IHS space, for example, is a representation of the rectangular RGB vector in cylindrical coordinates; where I (intensity) is the distance along the central axis, S (saturation) is the distance from the central axis, and H (hue) is the direction from the central axis to the tip of the vector.

While it is the case that these different pixel representations may be readily converted to and from one another, the behavior of a distance metric is dependent on the color representation. For example, a vector A may appear closer to vector B than vector C in one color space but closer to vector C than vector B in the other. Consider, for example, a city-block metric applied to the RGB color space and the [rg,by,bw] opponent color space. The RGB vector [0, 0, 5] is a city-block distance of 5 from the origin while the vector [3, 2, 1] is a distance of 6 (*further*). However, when transformed to the opponent color space, the first vector becomes [0, 10, 5] which is a distance of 15 from the origin while the second vector becomes [1, -3, 6] which is is a distance of 10 (*closer*).

The transformation to various color models, such as the opponent color space defined in equation 4.12 or a normalized color space defined in equation 4.13 are often used as an ad hoc *color constancy* algorithm [Duda and Hart, 1973], [Ballard and Brown, 1982].

$$\begin{aligned} u &= \tfrac{R}{R+G+B} \\ v &= \tfrac{G}{R+G+B} \\ w &= \tfrac{B}{R+G+B} \end{aligned} \tag{4.13}$$

It is important to realize however that true color constancy, which is defined by [Hurlbert, 1989] as the tendency of objects to stay the same perceived color under changing illumination, requires the measurement of a scene's reflectance properties. Unfortunately, determining these properties from image irradiance on a pixel by pixel basis is an under constrained problem. Since experiments such as those reported in [Land and McCann, 1971] show that humans perceive constant colors with a high proficiency, it appears as though human color perception is based on non-local properties and therefore could never be achieved by a pixel by pixel image transformation.

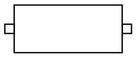
Still, as was shown above, the choice of color space does effect a matching algorithm. The normalized color model, for example, remains constant to changes in illumination which change the three tristimulus irradiance measures by a scalar factor.

$$[R', G', B'] = [cR, cG, cB] \tag{4.14}$$

Conversely, the RG and BY components of the opponent color space are invariant to illumination changes which change the three irradiance measures by an additive factor.

$$[R', G', B'] = [R + c, G + c, B + c] \tag{4.15}$$

A more sophisticated color constancy algorithm would be a useful addition to Sieve. [Luong, 1991] presents a number of candidate approaches. More recent promising approaches are described in [Nagao and Grimson, 1995] and [Freeman and Brainard, 1995].

### VsPixelTransform 

The `VsPixelTransform` filter performs image transformations from one pixel representation to another. The module is a stateless filter that computes an output image, $I'$, for each input image, $I$, such that every output pixel is strictly a function of its corresponding input pixel.

$$I'(x, y) = f(I(x, y)) \tag{4.16}$$

`VsPixelTransform` is used to process both color and gray scale images. The `colorTransform` parameter specifies the transformation that is performed for color image inputs. One possible value for this parameter is `matrix`. When set to this value, the module performs an arbitrary linear transformation on each input pixel. The coefficients of the matrix are specified using the `matrix` parameter, the value of which is a list of twelve coefficients, $\{c_1 ... c_{12}\}$, that are applied to color pixels, $[p_1, p_2, p_3]$, as follows:

$$\begin{bmatrix} p'_1 \\ p'_2 \\ p'_3 \end{bmatrix} = \begin{bmatrix} c_1 & c_2 & c_3 \\ c_5 & c_6 & c_7 \\ c_9 & c_{10} & c_{11} \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ p_3 \end{bmatrix} + \begin{bmatrix} c_4 \\ c_8 \\ c_{12} \end{bmatrix} \tag{4.17}$$

Other possible values for `colorTransform` are `rgb2xyz`, `rgb2yiq`, `rgb2opp`, `xyz2rgb`, `yiq2rgb`, `opp2rgb`, and `null`. Some of these transformations map directly to matrix transformations while others, such as `rgb2yiq` and `yiq2rgb`, are non-linear.

`VsPixelTransform` may be configured to produce gray scale images from color image by extracting a single color component. The `colorSelect` parameter may be set to an integer value between 0 and 3. A value of 0 disables this feature, while a value of 1 to 3 selects the first, second, or third component of the color vector.

Finally, the `grayTransform` parameter specifies the transformation that is performed for gray image inputs. Possible values for this parameter are `sign`, `window2gray`, `gray2window`, and `null`.

The parameters for `VsPixelTransform` are summarized as follows:

| Parameter | Value | |
|---|---|---|
| `input` | *input_port* | Input port |
| `output` | *output_port* | Output port |
| `grayTransform` | {`sign window2gray ...`} | Default is `null` |
| `colorTransform` | {`matrix rgb2yiq ...`} | Default is `null` |
| `colorSelect` | {`0 1 2 3`} | Default is 0 |
| `matrix` | *float float ...* List of 12 floats | |

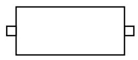
### 4.2.2 Edge Detection

Edge detection is a widely used image transformation in computer vision. Edge detectors typically transform color or gray scale images into binary images where true valued pixels represent *edges* in the input. The precise definition of an edge varies but may be roughly thought of as an abrupt change in color or gray value.

Edge representations are useful because they correspond to boundaries in objects. Thus, they are frequently used as features themselves or as the basis for higher level features such as lines and corners. Edges are relatively invariant to changes in lighting. Lighting invariance and the fact that edge maps remove a great deal of redundant information from images motivated their direct use as the input feature compared by the Hausdorff module described in section 4.1.3.

In principle, an edge detector is any algorithm which attempts to find or label abrupt changes in pixel value. In practice, most edge detectors are based on two dimensional operators which are convolved with the image. A given operator balances conflicting performance criteria: sensitivity (probability of detecting real edges), robustness (probability of detecting false edges), accuracy in determining location, and so on. One of the more successful algorithms was developed by Canny [1983]. Canny's edge detector combines a family of operators based on the directional derivatives of two dimensional Gaussians at different orientations and scales. Canny uses a heuristic decision procedure, based on the local estimate of the signal to noise ratio, to select the best operator at each location in the image.

An implementation of Canny's edge detector, based on code written at Cornell by Greg Klanderman, has been incorporated into Sieve as the `VsEdge` module.

**VsEdge** 

`VsEdge` (figure 4-9) transforms a stream of gray scale images into a stream of binary images where the true valued pixels represent edge locations.

`VsEdge` is designed to support multiple edge detection algorithms. The `type` parameter selects which method is used. Presently, this value may be set to `canny` or `fast`. The default value, `canny`, selects the Canny edge detector described above, while `fast`, which has been used primarily for demonstration, selects the simple transformation described by the following equation:

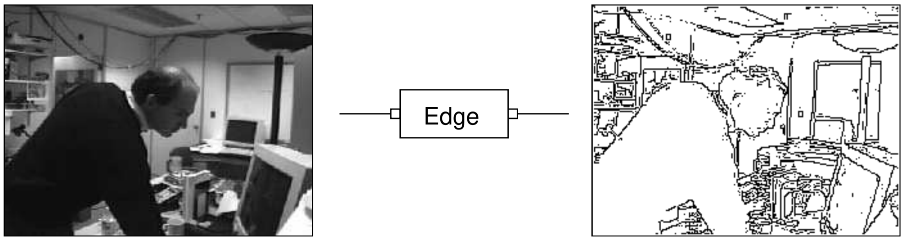



Figure 4-9: VsEdge

$$I'(x,y) = \begin{cases} 1 & \text{if } I(x,y) - I(x-1,y) \geq threshold \\ 1 & \text{if } I(x,y) - I(x,y-1) \geq threshold \\ 0 & \text{otherwise} \end{cases} \tag{4.18}$$

Both algorithms make use of the `threshold` parameter to specify sensitivity (increasing the threshold decreases the number of detected edges). In addition, the Canny algorithm optionally makes use of a second threshold parameter, `threshold2`. The second threshold, if non-zero, acts as the high threshold in the hysteresis scheme described in Canny's report [1983].

Presently, the two algorithms implemented by `VsEdge` are defined for gray scale images only. Thus, any payload which is not a a gray scale image is passed through unaltered. For this reason, `VsEdge` is often preceded by a `VsColorToGray` module.

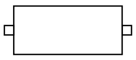
The parameters for `VsEdge` are summarized as follows:

| `Parameter` | `Value` | |
|---|---|---|
| `input` | *input_port* | Input port |
| `output` | *output_port* | Output port |
| `type` | {`fast canny`} | Default is `canny` |
| `threshold` | *int* | Default is 10 |
| `threshold2` | *int* | Default is 0 |

### 4.2.3 Histogram Transforms

Section 4.1.2 presented the `VsHistMatch` module as an example of a statistical matching module based on an image histogram. For performance reasons, histogram generation was built into the `VsHistMatch` module. This section presents the `VsHistogram` filter as a separate, modular component that transforms streams of images into streams of histogram payloads.

#### VsHistogram



The `VsHistogram` filter (figure 4-10) transforms images or portions of images into histograms. The present implementation restricts itself to histograms with uniform sized, non-overlapping bins.

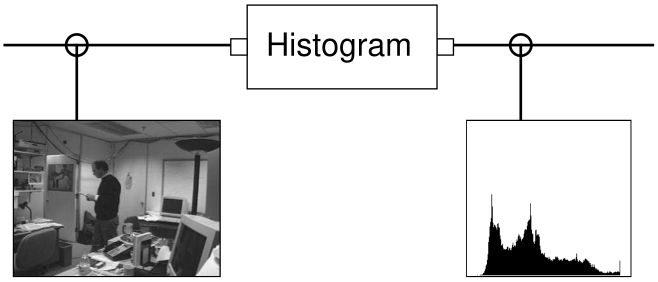



Figure 4-10: Histogram transformation

By default, the histogram is computed from all the pixels in the image. However, by setting the `nullPixel` parameter, the user may select a color or gray value which is not added to any bin. The `nullPixel` may be used to "mask" off regions of the image which are not part of the object of interest. Such histograms may thus be used to characterize segmented objects. Section 4.4 details the mechanisms for masking image regions.

The output of the histogram filter is a payload object called a `VsSignal`. A `VsSignal` is essentially an array of bins. As was the case with `VsHistMatch` the assignment of pixels to bins is determined by the `colorBits` and `grayBits` parameters. For one dimensional histograms (i.e. gray pixels) the assignment is straightforward as the most significant bits of the pixel value are used to index into the bin array.

For multi-dimensional histograms (i.e. color pixels) the assignment is only slightly more complicated. Each component of the color pixel is reduced to its most significant bits according to the `colorBits` parameter (see section 4.1.2), to form a triple: $[p_x, p_y, p_z]$. This three dimensional number is then mapped to an index in a one dimensional array in a manner analogous to how two dimensional image locations are mapped to one dimensional array indices:

$$i = p_x + p_y * dim_x + p_z * dim_x * dim_y \tag{4.19}$$

where $dim_x$ and $dim_y$ are the dimensions of the first two components of the color histogram.

The parameters for `VsHistogram` are summarized as follows:

| Parameter | Value | |
|---|---|---|
| `input` | *input_port* | Input port |
| `output` | *output_port* | Output port |
| `grayBits` | *int* | Default is 8 |
| `colorBits` | *int int int* | Default is {3 3 3} |
| `nullPixel` | *int* | Default is -1 |

### 4.2.4 Spatial Filtering

A large class of useful image processing transformations may be expressed as the convolution of an image with a point spread function. In particular, convolution may be used to implement any linear and shift invariant function where the value of each transformed pixel is the weighted sum of its own value and the value of its neighbors. A *stencil*, which is the discrete representation of the point spread function, represents the relative location and values of the weights.

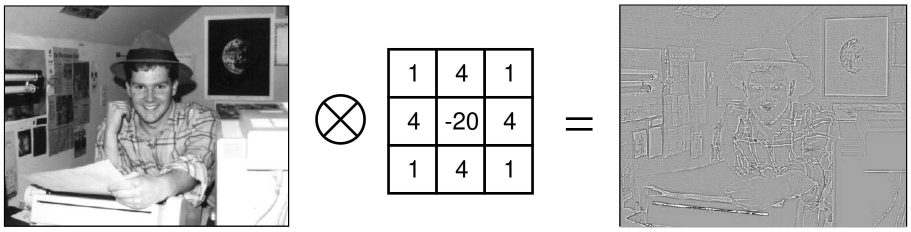



Figure 4-11: Convolution of an image with a stencil approximating the Laplacian

Formally, the convolution of a scalar image, $I(i,j)$ and an $nxm$ stencil $w(i,j)$ yields the transformed image $I'(i,j)$ given by the following equation:

$$I'(i,j) = \sum_{k=0}^{n-1} \sum_{l=0}^{n-1} I(i+k-n/2, j+l-m/2) * w(k,l) \tag{4.20}$$

In order to define convolution for color images, one must either adopt vector replacements for scalar multiplication and summation or simply apply the scalar stencil to each of the vector components independently.

Spatial filters are commonly used to implement edge enhancement operators, such as the Laplacian, or blurring operators, such as a Gaussian or boxed filter. Figure 4-11 shows the effect of convolving a scalar image with a stencil which implements the former, while figure 4-12 depicts the latter.

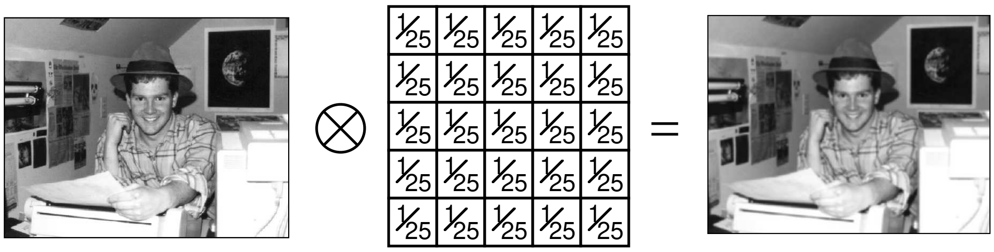
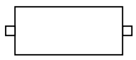



Figure 4-12: Convolution of an image with a box filter

**VsConvolve** 

`VsConvolve` takes a stream of input images and convolves each one with a user specified stencil. Presently, only gray scale images are supported, color images are passed unaltered.

The stencil itself is specified by two parameters. The `dimension` parameter is a pair of integers specifying the width and height of the stencil. These values are required to be odd numbers so that the stencil is centered around the output pixel. The `window` parameter is a list of floating point weights. The position of the weights in the stencil is interpreted as starting in the upper left corner of the stencil and proceeding, row by row, to the bottom right corner.

The parameters for `VsConvolve` are summarized as follows:

| Parameter | Value | |
|---|---|---|
| `input` | *input_port* | Input port |
| `output` | *output_port* | Output port |
| `dimensions` | *int int* | Width and height of the stencil |
| `window` | *float float float ...* | List of weights |

## 4.3 Motion

This section is concerned with the detection and measurement of change in an image stream. Since change in an image stream is often caused by the movement of objects in a scene, change detection is closely related to, and often confused with, the problem of motion detection. Indeed, this thesis itself does not make a strong distinction between the two and uses the term "motion" loosely to refer to changes detected in image streams.

Motion has proven to be be a particularly valuable source of information in Sieve. Not only is motion in an image relatively easy to detect, it indicates what is changing in a scene so that it is frequently the information of greatest interest. In fact, the behavior of many applications are essentially determined by whether or not the program detects change in the scene. For example, the room monitor determines whether there are people in the room essentially by monitoring the amount of motion in the scene. The whiteboard monitor filters its input to remove transitional motion and then makes decisions about whether to save an image based on the longer term changes appearing on the board. The gesture analyzer is concerned with recognizing and characterizing the motion in its view. These programs are effective because the change in the scene is precisely the information in which the user is interested.

Change detection is essentially the converse of matching. Rather than determine whether two things are the same, change detection concerns itself with measuring how they differ. As a result, the methods for measuring change are closely related to those used for matching. The main difference is that rather than compare two inputs which may have been captured at different times and under different viewing conditions, motion detectors typically compare inputs which have been captured from the same source at nearly the same time. As a result, techniques like pixel matching may be applied directly to the image stream with little or no preprocessing.

Sieve implements three motion filters based on three different image processing techniques: `VsDiffMotion`, `VsOptFlowMotion`, and `VsStatMotion`. Each of these modules takes a

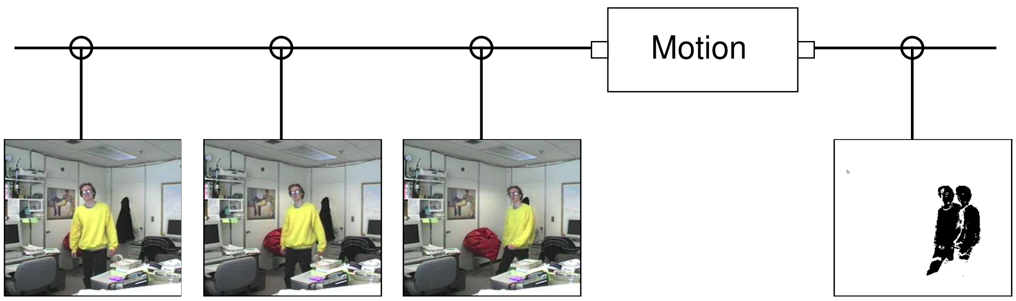



Figure 4-13: Motion detection

stream of images as input and produce a stream of binary images as output (figure 4-13). The binary image represents a foreground/background segmentation whereby the true valued foreground pixels indicate the location on the image plane where change is occurring.

### 4.3.1 Differential Motion

The simplest of the three motion sensing techniques is based on calculating the amount by which a pixel's value changes between successive images. Conceptually, the technique is akin to computing the temporal derivative $dI/dt$ of an image stream and thresholding the result. Sieve implements this concept in the `VsDiffMotion` filter which calculates the amount by a which a pixel changes using the pixel distance metrics described in section 4.1.1.

Formally, `VsDiffMotion` takes a sequence of color or gray scale images: $\{I_t(x,y)|t=1...n\}$ and computes a sequence of binary images $\{D_t(x,y)|t=1...n\}$

$$D_t(x,y) = \begin{cases} 1 & \text{if } \|I_t(x,y) - I_{t-1}(x,y)\| \geq \mathit{threshold} \\ 0 & \text{otherwise} \end{cases} \tag{4.21}$$

where $\|I_t(x,y) - I_{t-1}(x,y)\|$ represents the *distance* between two pixels at the same location at times $t$ and $t-1$.

The output of a differential based method is clearly dependent on the time step between images. For instance, when used to detect a solid colored moving object, a fast frame rate would result in a relatively thin edge of pixels changing around the object whereas a slow frame rate would result in a greater number of pixels changing between frames.

One means of accommodating varying frame rates is to vary the number of frames between images. Thus, rather than restrict comparisons to be between $I_t$ and $I_{t-1}$, `VsDiffMotion` provides the parameter $k$, which may be used to specify that comparisons be made between $I_t$ and $I_{t-k}$ for $k \geq 1$.

Differencing works well for locating moving objects when the image stream has been captured from a stationary camera. The simplicity of the technique results in dependable, high quality output which may be obtained at low computational cost. Still, there are several limitations. `VsDiffMotion` is of little use for processing output captured from a moving camera. Since video produced for TV is packed with pans, zooms, and visual effects, more

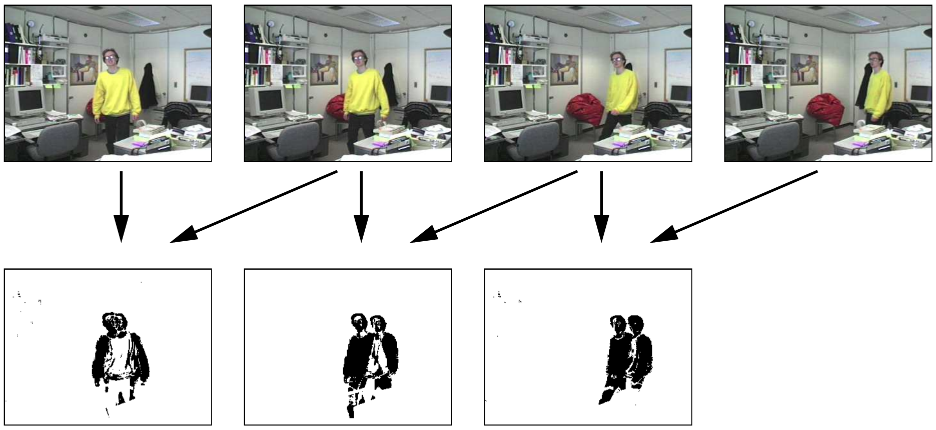
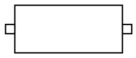


Figure 4-14: The output of `VsDiffMotion` with the shadow artifact

sophisticated techniques must be used for analyzing production video. Another weakness of `VsDiffMotion` for locating moving objects is that it can not distinguish between the foreground and background. Thus, when an object moves across a stationary background, both the location where the object has moved to and the location where the object has moved from appear to have changed. The effect, which is similar to a shadow, is illustrated in Figure 4-14.

### VsDiffMotion



`VsDiffMotion` transforms a stream of color or gray scale images into a stream of binary images which indicate where change is occurring in the image plane. The filter is implemented as a compound module, composed of the four modules shown in Figure 4-15.

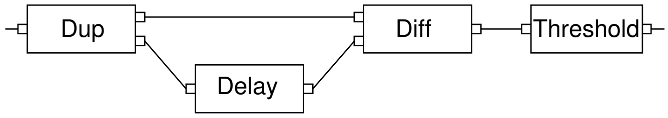


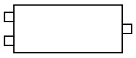

Figure 4-15: VsDiffMotion implementation

`VsDiffMotion` works by duplicating the input stream and passing the two copies to a `VsDiff` module which computes a pixel by pixel difference for every pair of images in the two streams. The `VsDelay` module causes the two input streams to be offset by one or more frames so that the computed differences indicate the amount that a particular pixel has changed during that delay. Finally, a `VsThreshold` module transforms the 8-bit difference measurements produced by `VsDiff` into boolean values by applying a thresholding function to each pixel.

The `VsDiffMotion` module has several input parameters which it passes to its underlying components. For example, the `threshold` and `property` parameters are passed to the `VsThreshold` module, the `delay` parameter is passed to the `VsDelay` module, and both the `vectorMethod` and `scalarMethod` parameters are passed to the `VsDiff` module. Descriptions of these modules are provided below.

The parameters for `VsDiffMotion` are passed to its constituent modules as follows:

| Parameter | Value |
|---|---|
| `input` | `VsDup input` |
| `output` | `VsThreshold output` |
| `threshold` | `VsThreshold threshold` |
| `property` | `VsThreshold property` |
| `delay` | `VsDelay delay` |
| `scalarMethod` | `VsDiff scalarMethod` |
| `vectorMethod` | `VsDiff vectorMethod` |

### VsDiff 

`VsDiff` computes the pixel by pixel distance between pairs of input images. Its implementation is similar to that of `VsPixelMatch`, though in addition to computing a property, `VsDiff` computes an output image as well. The output image is a single band image equal in size to the two input images[4] with eight bit pixel values. The interpretation of the pixel values depends on the specified distance method.

The two input streams are labeled `input1` and `input2`. As was the case with `VsPixelMatch`, one stream may be held constant while the other flows. The user may specify which stream to hold with the `hold` parameter. The value of `hold` may be 1, 2 or 0 which respectively specifies for the filter to hold the first input, second input, or neither input constant.

`VsDiff` implements several methods, shown in Table 4.2, for computing a distance between pixels. As shown in the table, the user may specify the method for gray scale and color images by setting the input parameters `scalarMethod` and `vectorMethod` respectively[5].

| Name | Function |
|---|---|
| scalarMethod absDiff | $\lvert a - b \rvert$ |
| vectorMethod euclidean | Equation 4.4 |
| vectorMethod cityBlock | Equation 4.5 |
| vectorMethod maxDiff | Equation 4.6 |
| vectorMethod dotProduct | Equation 4.7 |
| vectorMethod crossProduct | Equation 4.8 |

Table 4.2: Distance methods for VsDiff

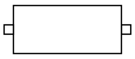
The distance value computed by each of the methods is mapped to a value between 0 and 255, where 0 represents the minimum and 255 represents the maximum possible distance. Thus, in the case of the `absDiff` or `maxDiff` methods, the output value is simply the value computed by the formula shown in Table 4.2 while in the case of the `cityBlock` method, this value is divided by a 3 and in the case of the `crossProduct` or `dotProduct` methods (which produce a value between 0 and 1), the value is multiplied by 255. Finally, the value computed by the `euclidean` method is divided by $\sqrt{3}$.

[4] `VsDiff` assumes its two input images are the same size and type. If they are not, the module throws the two images away, outputs a blank image, and issues a warning.

[5] Currently `absDiff` is the only method provided for gray scale images

The parameters for `VsDiff` are summarized as follows:

| Parameter | Value | |
|---|---|---|
| `input1` | *input_port* | Input port |
| `input2` | *input_port* | Input port |
| `output` | *output_port* | Output port |
| `hold` | {`0 1 2`} | Default is 0 |
| `scalarMethod` | {`absDiff`} | Default is absDiff |
| `vectorMethod` | {`cityblock euclidean maxDiff dotProduct crossProduct`} | Default is `cityblock` |

### VsThreshold 

`VsThreshold` (figure 4-16) converts gray scale images to binary images by applying a thresholding function to the image pixels. Presently, the module is implemented as a simple filter which ignores all payload types other than gray scale images. Gray scale images, however, are transformed according to the formula:

$$I\prime(x,y) = \begin{cases} 1 & \text{if} \quad t_{min} \leq I(x,y) \leq t_{max} \\ 0 & \text{otherwise} \end{cases} \tag{4.22}$$

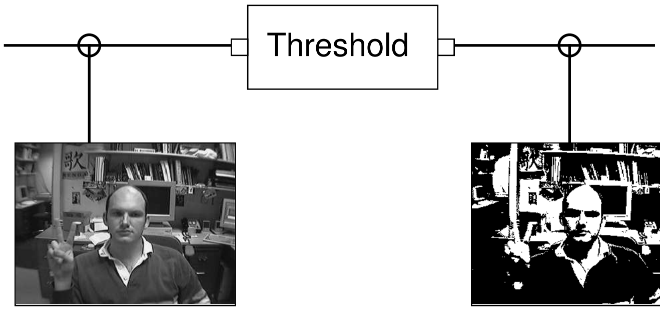



Figure 4-16: VsThreshold

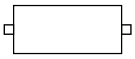
The values of $t_{min}$ and $t_{max}$ are set by the `threshold` parameter. `VsThreshold` may be configured to produce a property which summarizes the percentage of pixels which have been mapped to 1. The `property` parameter specifies the name of this floating point value.

$$property = \frac{1}{hw} \sum_{x=0}^{w-1} \sum_{y=0}^{h-1} I'(x,y) \tag{4.23}$$

The parameters for `VsThreshold` are summarized as follows:

| Parameter | Value | |
|---|---|---|
| `input` | *input_port* | Input port |
| `output` | *output_port* | Output port |
| `threshold` | *int int* | $t_{min}$ $t_{max}$ |
| `property` | *string* | Default is no property |

### VsDelay



The `VsDelay` module is used to buffer payloads. It serves to simplify the implementation of other modules by allowing them to be stateless. The module functions as a one input, one output filter which produces one output payload for each input payload. The output payload it produces is the same as the previous payload it received. The only exception is the first payload, for which there is no previous input. `VsDelay` handles the first payload by duplicating it and sending one copy immediately while holding the other until the next payload arrives. Thus, the output stream produced by a `VsDelay module` is the same as the input stream with the first payload duplicated.

The effect of the duplicated payload is to push back the arrival of later payloads. An alternative would be to have `VsDelay` drop, rather than duplicate, the first payload. Such an implementation would indeed work, but the behavior of `VsDiffMotion` would be slightly different. While in the present implementation, `VsDiffMotion` produces exactly one output payload for each input payload, the alternative implementation would result in a compound module which does not produce an output payload until the arrival of the second input. Thus the output stream would always contain one fewer payload than the input stream. While it is not the case that every filter is required to produce an output stream equal in length to its input stream, those which do are generally easier to use. In particular, such filters may be placed in a pipeline without skewing the arrival of payloads with respect to another stream.

An alternative to duplicating the first payload is to load an initial payload from an alternate input stream. The `initial` parameter implements this alternative. The `initial` parameter is an input port, which by default is disconnected. If, however, the `initial` port is connected to another modules output port, `VsDelay` inserts the first payload which arrives at the initial port into the input stream rather than duplicating the first payload that arrives at its input port. The effect is, once again, to push pack the arrival input payloads.

Multiple payload delays may be specified with the `delay` parameter, which may be any integer value. Negative values cause the first $n$ payloads to be dropped while positive values cause the first payload to be duplicated $n$ times. The default `delay` value is 1.

One caveat with the duplication approach is the result of having different types of payloads flow through the same channels in the processing network. As detailed in chapter 3, modules are designed to ignore payloads which they don't recognize by passing them through. For instance, the `VsDiff` module passes though any payload which is not a video frame. It would be a mistake, therefore, for the `VsDelay` module in the `VsDiffMotion` filter to duplicate the first payload if it were not a video frame. In other applications, one may require the module to delay payloads of some other type. The `payloadType` parameter is used to specify the type of payload which will be delayed. Thus, when the `VsDelay` module receives a payload it passes it through if it is not of type `payloadType`. Otherwise, it delays the payload by passing though the previous payload of that type. The default value for `payloadType` is `VsVideoFrame`.

The sequences below illustrate the behavior of a `VsDelay` module configured to delay video payloads in a stream of interleaved audio payloads, $a_n$ and video payloads $v_n$.

$$\begin{array}{ll} \text{input stream} & a_1\ a_2\ v_1\ a_3\ a_4\ v_2\ a_5\ v_3\ v_4\ \ldots \\ \text{output stream} & a_1\ a_2\ v_1\ a_3\ a_4\ v_1\ a_5\ v_2\ v_3\ \ldots \end{array} \tag{4.24}$$

The parameters for `VsDelay` are summarized as follows:

| Parameter | Value | |
|---|---|---|
| `input` | *input_port* | Input port |
| `initial` | *input_port* | Input port |
| `output` | *output_port* | Output port |
| `delay` | *int* | Default is 1 |
| `payloadType` | *payload_type* | Default is VsVideoFrame |

### 4.3.2 Optical Flow

While the differential method described above provides a fast and robust means of detecting which pixels are changing in an image, it provides little information about how they change. This is unfortunate since there is a great deal of structure in the way an image changes in response to the movement of objects in a scene. In particular, local brightness patterns and textures in an image appear to move or "flow" from one location to another in the two dimensional plane.

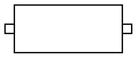
Optical flow is defined as the apparent two dimensional motion of the brightness patterns in a sequence of images [Horn, 1986]. Optical flow may be represented by a vector field which describes the two dimensional velocity of each point in the image plane.

$$OF(x,y) = \begin{bmatrix} u(x,y) \\ v(x,y) \end{bmatrix} \qquad \begin{array}{l} u = dx/dt \\ v = dy/dt \end{array} \tag{4.25}$$

Tracking the brightness of individual pixels or small regions involves considerable ambiguity while the tracking of larger groups of pixels is valid only if all the pixels in the group are undergoing the same transformation. In addition, pixels which appear or disappear due to occlusion as well as those which change in brightness due to changing imaging conditions (such as imaging geometry or lighting) or even spontaneous change in appearance (such as monitor flicker), are problematic.

Despite these problems, many researchers have been successful in measuring and using optical flow [Horn and Schunck, 1981], [Wang and Adelson, 1993], [Woodfill, 1992], [Horswill, 1994]. Sieve has incorporated its own method based on code written by Satyajit Rao.

When compared to the differential technique, optical flow has the advantage that it provides more information and works on input captured from moving cameras. The drawbacks are that it requires considerably more computation and that it can be noisy, especially when computed using some of the faster algorithms.

### VsOptFlowMotion 

VsOptFlowMotion transforms a stream of images into a stream of binary masks which indicate where motion is being detected in the image plane. This filter is a compound module, similar in design to that of VsDiffMotion (figure 4-17). The only differences between the two are the use of a VsOptFlow module, rather than a VsDiff module, and the addition of a VsColorToGray preprocessing module. The function of the VsColorToGray is to convert color images in the input stream to gray scale, since that is the only format currently accepted by VsOptFlow.

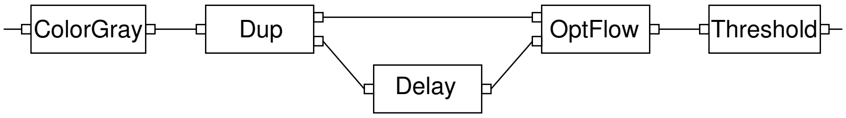
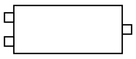



Figure 4-17: VsOptFlowMotion implementation

The two input VsOptFlow module, described below, computes the vector field which describes the change in location of each pixel between the two inputs. Each vector is represented by a magnitude and a direction. For the purpose of this compound filter, VsOptFlow module is configured to output a scalar image representing only the magnitude values. These values are then thresholded to produce the binary image, so that the output of this motion detector is a binary image indicating where the displacement of pixels in the image plane is greater than a specified distance.

The parameters of VsOptFlowMotion are passed through to its component modules. In most cases, the name of the VsOptFlowMotion parameter is the same as that for the underlying module. In the case of the threshold parameter, however, ambiguity exists because two of the underlying modules define a threshold. VsOptFlowMotion distinguishes between the two parameters by adopting the name flowThreshold to refer to the threshold parameter of the underlying VsOptFlow module while retaining threshold as the name of the VsThreshold threshold parameter.

| Parameter | Value |
|---|---|
| input | VsColorToGray input |
| output | VsThreshold output |
| threshold | VsThreshold threshold |
| property | VsThreshold property |
| delay | VsDelay delay |
| sradius | VsOptFlow sradius |
| pradius | VsOptFlow pradius |
| flowThreshold | VsOptFlow threshold |

### VsOptFlow 

`VsOptFlow` (figure 4-18) computes the flow field between two streams of images. The flow field is a two dimensional array of two dimensional vectors. The vectors indicate the displacement for each pixel in `input1` to a corresponding pixel in `input2`.

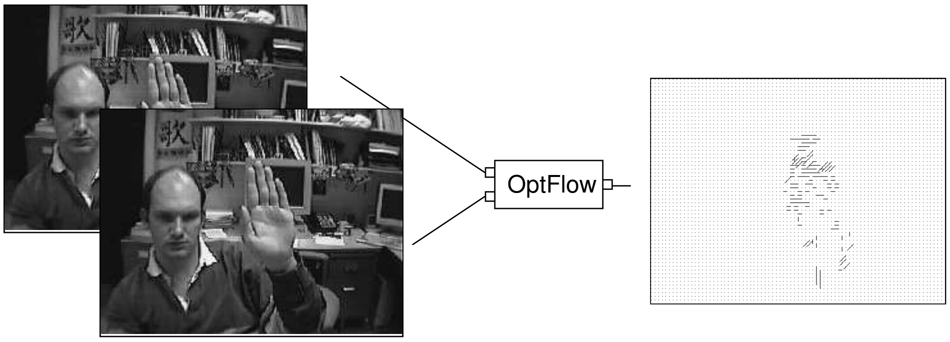



Figure 4-18: VsOptFlow

`VsOptFlow` uses a correlation based algorithm to compute the correspondence between images. In particular, each pixel is matched by taking the rectangular region of its neighboring pixels and searching for the rectangular region in the second image which most closely matches that region. The sum of the absolute difference between the pixel regions is used as the matching metric. The size of the rectangular patch is specified by the `pradius` parameter. A value of two, for instance, specifies that a 5x5 neighborhood of pixels centered around the pixel being matched is to be used.

The area that is searched for a match is limited to the vicinity of the location of the matching pixel. The extent of this area is specified by the `sradius` parameter. Unlike `pradius`, the `sradius` is specified relative to the dimensions of the image as a floating point value between 0 and 1.

Typically, a fair number of pixel regions will fail to closely match any regions in the second input. Such is the case for regions which have moved beyond the search radius and for regions which span object boundaries. Rather than assign a misleading value to such vectors, the `threshold` parameter specifies a cut-off value for the sum of absolute difference, above which the null vector is assigned.

The displacement vectors are represented by a magnitude and a direction. The `VsOptFlow` may be configured with the `select` parameter to output both of these quantities as a vector image or to output a scalar image representing either the magnitude or direction fields. In either case, both magnitude and direction are represented as 8 bit values. For magnitude, this is simply a number between 0 and 255 representing the length of the vector in pixel units. For direction, the value represents the angle measured counter clockwise from the `x` axis in increments scaled so that the 256 values represent a full 360 degrees.

The parameters for `VsOptFlow` are summarized as follows:

| Parameter | Value | |
|---|---|---|
| input1 | *input_port* | Input port |
| input2 | *input_port* | Input port |
| output | *output_port* | Output port |
| sradius | *float* | Default is 0.1 |
| pradius | *int* | Default is 2 |
| threshold | *int* | Default is 20 |
| select | {magnitude direction vector} | Default is magnitude |
| hold | {0 1 2} | Default is 0 |

### 4.3.3 Background Motion

Finally, Sieve implements a third type of motion filter which works by taking the difference between the incoming stream of images and a known, stationary background. A binary image is obtained by applying a threshold to the difference values (figure 4-19).

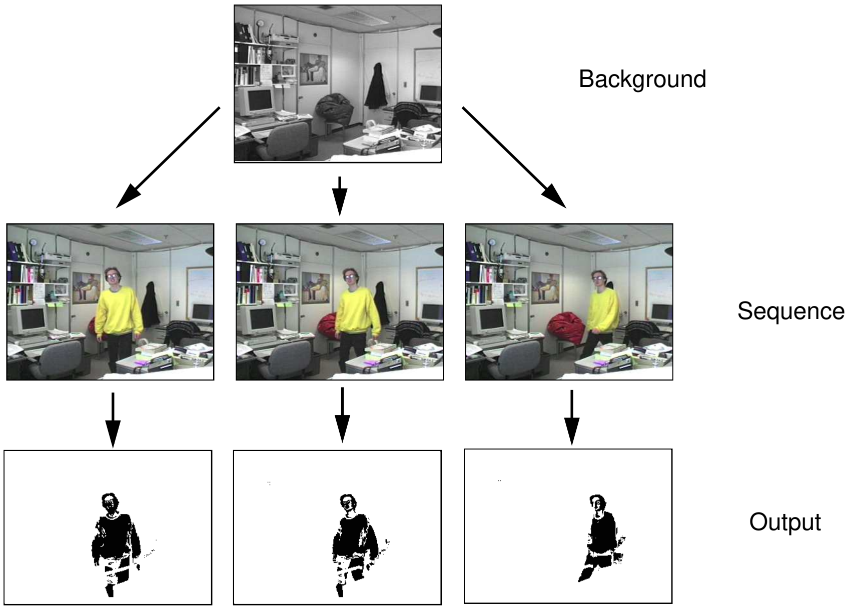
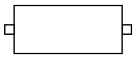



Figure 4-19: Background motion

The background technique is particularly suitable for producing a segmentation which distinguishes a moving object from the background. The reason for this is that images are not compared strictly against recent images, so that moving objects do not "blend" with themselves. Blending occurs when pixels imaged from two different objects which are similar in value, happen to occupy the same location in the image plane. Blending between a moving object and a background object is occasionally unavoidable. However, a more common blending problem occurs when comparing successive images in which different portions of the same moving object blend with each other. The background technique avoids this latter problem by comparing images to an estimate of the background.

The background technique has the additional advantages that its output is less sensitive to variations in frame rate and that it eliminates the *shadow* effect depicted in Figure 4-14. The disadvantage is that it depends strongly on the ability to accurately compute the background, which can at times be problematic. Errors in the background calculation can result in stationary portions of the image appearing to be part of a moving object. In addition, the background computation is based on waiting for regions of the image to stay unchanged for a specified period of time so that delay is introduced into the system. The background computation is performed by the `VsStationary` filter described below.

### VsStatMotion



`VsStatMotion` implements the background subtraction method of detecting motion. As was the case with `VsDiffMotion` and `VsOptFlowMotion`, `VsStatMotion` takes a stream of images and computes a stream of foreground/background segmentations which indicate which portions of the image are changing.

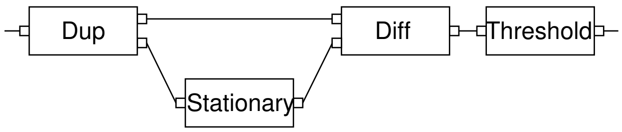



Figure 4-20: VsStatMotion filter

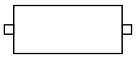
The `VsStatMotion` filter is implemented as a compound module (figure 4-20). The input is duplicated into two identical streams by a `VsDup` module. One of these streams is passed through a `VsStationary` filter and then compared with the unfiltered stream using the `VsDiff` module. The function of the stationary filter, described below, is to filter out the moving objects from a scene so that what remains is a depiction of the background.

As was the case with `VsDiffMotion`, the output of the `VsDiff` module is passed to a `VsThreshold` filter which produces a binary image which indicates which pixels are different in the two streams.

The parameters of `VsStatMotion` are passed through to its component modules. However, several ambiguous parameter names exist because the `VsStationary` module itself contains a `VsDiff` module. The threshold parameter is handled by passing `threshold` to the `VsThreshold` module and using `stationaryThreshold` to refer to the threshold in the `VsStationary` module. In addition `scalarMethod` and `vectorMethod` are passed to both the `VsDiff` module and the `VsStationary` filter so that these parameters are constrained to be the same for both modules.

The following table summarizes how parameters for `VsStatMotion` are passed to the underlying modules:

| Parameter | Value |
|---|---|
| input | VsDup input |
| output | VsThreshold output |
| threshold | VsThreshold threshold |
| property | VsThreshold property |
| blockSize | VsStationary blockSize |
| constantCount | VsStationary constantCount |
| moveBits | VsStationary moveBits |
| stationaryThreshold | VsStationary threshold |
| stationaryProperty | VsStationary property |
| delay | VsStationary delay |
| scalarMethod | VsDiff scalarMethod & VsStationary scalarMethod |
| vectorMethod | VsDiff vectorMethod & VsStationary vectorMethod |

**VsStationary** 

`VsStationary` (figure 4-21) attempts to compute the *background* of a scene. The background is defined to be the image that would have been observed by the camera if all of the moving objects in the scene were removed. `VsStationary` acts as a one input, one output filter which produces an output image for each input image. The output image is the filter's current best estimate of the background, given the latest input.

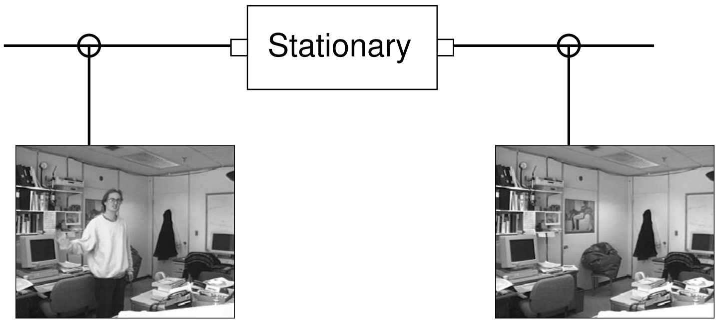



Figure 4-21: VsStationary

The filter starts by storing the first image that arrives as its `memory_image` and outputs a copy of the image as its initial estimate of the background. Upon each additional arrival, the memory image is updated and a copy is produced as output.

There are many possible algorithms which may be used to update the memory image. Zentner [1993] explored several of these, namely frame averaging, random updating, and selective averaging, using the VuSystem programming environment.

Presently, Sieve implements its own scheme based on counting the number of times that each region of the image has remained unchanged. The regions are rectangular blocks which may vary in size from one square pixel up to the full size of the image. A block is considered to have changed if more than a specified number of pixels in the block are sufficiently different in value from that of the previous image. Once a block has remained unchanged for a specified number of frames, its value in the memory image is updated to be the same as that of the current input image.

The use of blocks to detect change helps prevent solid textured regions within moving objects from appearing to be stationary. In addition, blocks of pixels contain more information than singular pixels, so the block technique reduces blending by reducing ambiguity.

The stationary algorithm is capable of composing the background of scenes even when the background is never fully exposed. A problem that arises, however, is that portions of the scene which are composed from different time periods are often captured under varying lighting conditions. For example, even on a short time scale, a person walking through a scene casts shadows, changes the white balance of the imaging system, and even changes the character of the ambient light in the room. As a result, portions of the image which are blocked from being updated (because they are obstructed by the moving object) may be slightly lighter or darker than other portions of the background. The result is a *ghosting* effect whereby the moving object appears as a slightly darkened or lightened region in the background [6].

`VsStationary` is implemented as a compound module (figure 4-22). The compound implementation, consisting of a `VsDup`, `VsDiffMotion`, and `VsStatGate` module, allows Sieve to use `VsDiffMotion` to determine which pixels are changing. `VsStatGate` takes both a stream of images and a stream of binary motion segmentations and produces a stream of images which are updated using the scheme described above.

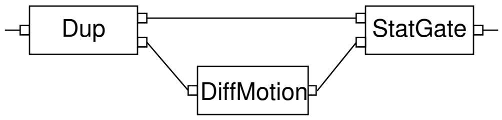
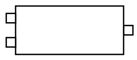



Figure 4-22: VsStationary implementation

The parameters for `VsStationary` are passed to its constituent modules as follows:

| `Parameter` | `Value` |
|---|---|
| `input` | `VsDup input` |
| `output` | `VsStatGate output` |
| `blockSize` | `VsStatGate blockSize` |
| `constantCount` | `VsStatGate constantCount` |
| `moveBits` | `VsStatGate moveBits` |
| `property` | `VsStatGate property` |
| `scalarMethod` | `VsDiffMotion scalarMethod` |
| `vectorMethod` | `VsDiffMotion vectorMethod` |
| `delay` | `VsDiffMotion delay` |
| `threshold` | `VsDiffMotion threshold` |

[6] While visible on a monitor, the ghosting effect is difficult to reproduce in print because the differences are faint and because the effect is more noticeable in video than in stills.

### VsStatGate 

`VsStatGate` is designed to work in conjunction with a motion filter to implement a stationary filter. The module takes two inputs, one labeled `input`, which is a stream of color or gray scale images, and the other labeled `mask`, which is a stream of binary images. The binary images are presumed to have been produced by a motion filter. In the case of the implementation described above, the `VsDiffMotion` filter is used. The module produces a stream of output images, one for each pair of inputs. The output images are the same format (color or gray scale) as the `input` stream.

`VsStatGate` implements the background computation algorithm described above. The size of the blocks is specified by the `blockSize` parameter. The value of `blockSize` is a pair of integers specifying the width and height. If `blockSize` is specified as a single integer, rather than a pair, that value is used for both the width and the height.

The `moveBits` parameter specifies the number of pixels in a block which must be true in the binary input image for the block to be considered to have changed.

The `constantCount` parameter specifies the number of frames for which a block must be considered to be unchanged before its value is updated in the `memory_image`.

Finally, `VsStatGate` may be configured to produce a property which summarizes the percentage of pixels which have been updated for the current image. The `property` parameter specifies the name of this floating point value.

The parameters for `VsStatGate` are summarized as follows:

| `Parameter` | `Value` | |
|---|---|---|
| `input` | *input_port* | Input port |
| `mask` | *input_port* | Binary mask input port |
| `output` | *output_port* | Output port |
| `blockSize` | *int int* | width height |
| `moveBits` | *int* | Default 0 |
| `constantCount` | *int* | Default 10 |
| `property` | *string* | Default is no property |

## 4.4 Binary Image Manipulation

The three motion detectors described in the previous section transform their input from a stream of color and gray scale images into a stream of binary images which represent where change or movement occurs in the image plane. In doing so, these modules expose information that is of potential interest. More generally, any module that segments or identifies regions of interest in the image plane may represent its output in such a way. For example, segmentations may be computed on the basis of color [Subirana and Sung, 1992], texture [Bajcsy, 1973], or stereo disparity [Coombs *et al.*, 1992].

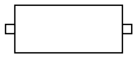
This section describes several modules which analyze and manipulate binary images. Specifically, the module `VsBinaryProps` analyzes binary images, attaching properties which characterize the data. `VsBinaryProps` thus serves an important role in the transformation of image data to symbolic form.

This section also presents `VsMotionExtract`. `VsMotionExtract` is a compound module which incorporates several modules that manipulate and transform binary images. In particular, `VsSpeck`, `VsMaskFill`, and `VsColor` are all used, together with a motion filter, to segment and extract a moving object from a scene.

### 4.4.1 Binary Image Properties

The transformation of binary image data to discrete properties is performed by `VsBinaryProps`. `VsBinaryProps` characterizes binary images by describing qualities about the "foreground" or region of true valued pixels in the image plane.

#### VsBinaryProps



`VsBinaryProps` processes a stream of input payloads by attaching properties to binary images. The properties reflect characteristics of each such image, such as the fraction of pixels which are true, the location of the center of mass, and so on.

The set of properties which `VsBinaryProps` computes is by no means exhaustive. During the development of Sieve, new properties continue to be added as needed. Presently, `VsBinaryProps` supports four measurements, each of which may be independently turned on or off by setting the appropriate property name parameter. The `countProp` parameter, for example, specifies the name of the property that is used to hold a floating point value representing the fraction of true value pixels in the image. Setting this parameter to the null string causes `VsBinaryProps` not to compute its value.

Similarly, the parameter `centerProp` specifies the base name of the pair of properties used to record the location of the center of mass of true pixels. The center of mass is represented by two floating point values between 0 and 1 which indicate the location of the center relative to the width and height of the image. The names of the properties are the base name concatenated with the string ".x" and ".y" respectively.

The `connectProp` parameter specifies the name of the integer property used to indicate the number of connected components in the image. The algorithm used to count the number of connected components is described in [Horn, 1986].

Finally, the `boxProp` parameter specifies the base name of a set of four properties used to define a bounding box around the region covered by the true valued pixels. The full names of the four properties are the base name concatenated with the strings ".top", ".bottom", ".left" and ".right". The properties are floating point numbers between 0 and 1 which represent the rectangle relative to the image dimensions.

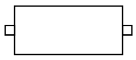
The `boxFrac` parameter is used to remove outliers from the edges of the image prior to computing the bounding box. A `boxFrac` value of .05, for example, specifies for the top-most 5%, bottom-most 5%, left-most 5%, and right-most 5% of true valued pixels be discarded.

| Parameter | Value | |
|---|---|---|
| `input` | *input_port* | Input port |
| `output` | *output_port* | Output port |
| `countProp` | *string* | Default is no property |
| `centerProp` | *string* | Default is no property |
| `boxProp` | *string* | Default is no property |
| `boxFrac` | *float* | Default 0 |
| `connectProp` | *string* | Default is no property |

There are many additional properties which could be added to `VsBinaryProps`. For example, in addition to center of mass, geometric properties such as the orientation may be represented as the axis of least second moment [Horn, 1986]. Topological properties such as the Euler number (the difference between the number of objects and number of holes) may also be used to characterize an image. Rosenfeld [Rosenfeld, 1969] presents methods for computing these and other properties which characterize the shapes of regions in binary images.

### 4.4.2 Segmentation

An important function of a binary image is to describe a region of interest in the image plane. This section describes how such a region may be identified and then used to segment or extract an object from a scene.

**VsMotionExtract** 

The `VsMotionExtract` attempts to extract a moving object from a stationary background (figure 4-23). The filter takes a stream of color or gray scale images as input and produces a stream of output images where the pixels that are not part of the moving object have been masked or "removed". The effect is similar to that obtained using chroma-key segmentation, whereby the boundary of a person or object is obtained by photographing the object in front of a constant colored background. In this case, however, the person may be differentiated from a cluttered background by using motion, rather than color.

The filter is implemented as a compound module consisting of a `VsDup`, `VsStatMotion`, `VsSpeck`, `VsFill`, and `VsColor` filter (figure 4-24). `VsStatMotion` computes the motion segmentation which it represents as a binary image. `VsSpeck` and `VsFill` attempt to improve the segmentation by using spatial properties in the binary image. `VsColor` composes the binary with the original input to produce an image in which the pixels that are not part of the moving object have been removed.

In the case of the `VsStatMotion` filter, the threshold parameter determines the cutoff for determining whether a pixel is considered to be in the foreground or the background. Too low a threshold will cause noise and lighting variations to cause spurious pixels to appear in the foreground while too high a threshold can cause portions of the moving object to blend into the background. To reduce the former, the output is first passed through a speck removal filter, `VsSpeck`. Finally, to alleviate the latter, the output is passed through a filter, `VsMaskFill`, which uses the assumption of spatial coherence in the image plane to fill in

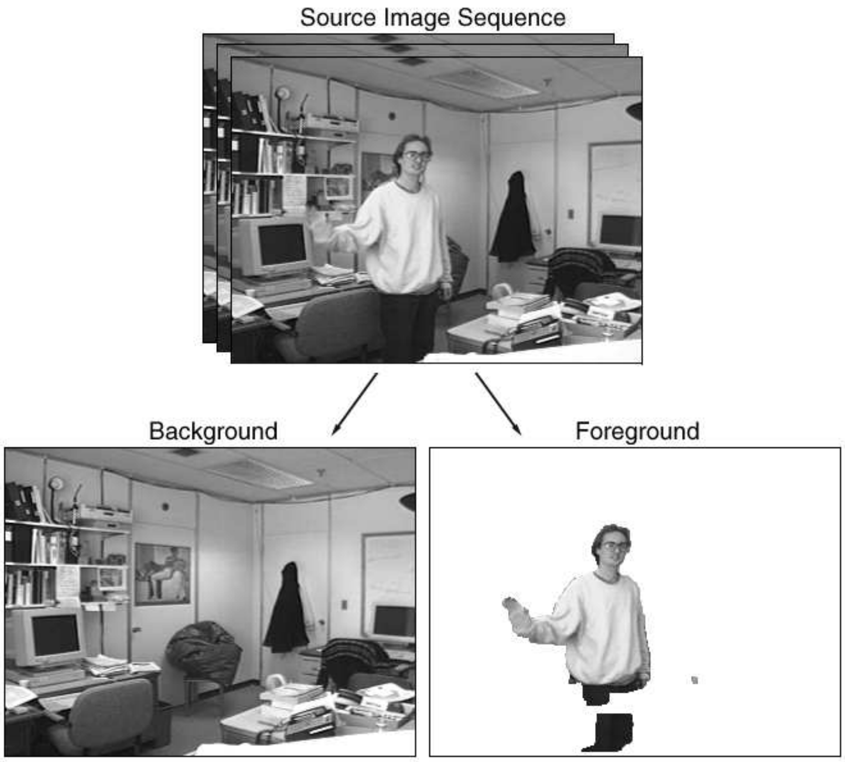



Figure 4-23: VsMotionExtract output

gaps in the mask.

The parameters for `VsMotionExtract` are passed to its constituent modules as follows:

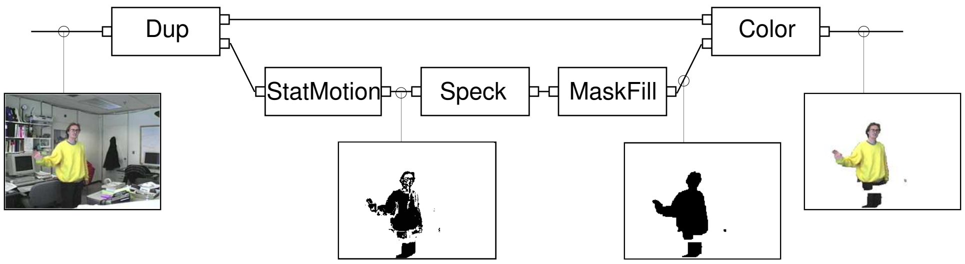
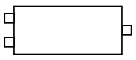



Figure 4-24: VsMotionExtract

| Parameter | Value |
|---|---|
| `input` | `VsDup input` |
| `output` | `VsColor output` |
| `blockSize` | `VsStatMotion blockSize` |
| `constantCount` | `VsStatMotion constantCount` |
| `moveBits` | `VsStatMotion moveBits` |
| `scalarMethod` | `VsStatMotion scalarMethod` |
| `vectorMethod` | `VsStatMotion vectorMethod` |
| `delay` | `VsStatMotion delay` |
| `threshold` | `VsStatMotion threshold` |
| `stationaryThreshold` | `VsStatMotion stationaryThreshold` |
| `speckWindow` | `VsSpeck window` |
| `speckThreshold` | `VsSpeck threshold` |
| `fillWindow` | `VsMaskFill window` |
| `foreground` | `VsColor foreground` |
| `background` | `VsColor background` |

### VsColor



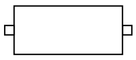
`VsColor` composes binary images with color and gray scale images. The module takes two input streams, labeled `input` and `mask`, and produces an output stream of composed images.

The image composition is governed by the `foreground` and `background` parameters. The `foreground` parameter determines the value of output pixels which are `true` in the `mask` image while the `background` parameter determines the value of pixels which are `false`. As shown in Figure 4-25, a positive value causes each of the foreground or background pixels to be set to the value of the parameter while a negative value causes the foreground or background pixels to be replaced with the corresponding pixel in the `input` image.

The parameters for `VsColor` are summarized as follows:

| Parameter | Value | |
|---|---|---|
| `input` | *input_port* | Input port |
| `mask` | *input_port* | Binary input port |
| `output` | *output_port* | Output port |
| `foreground` | *int* | Default 0 |
| `background` | *int* | Default -1 |

### VsSpeck



The binary masks computed by the motion filter typically consist of a dense cluster of object points surrounded by a sparse region of spurious noise. The difference in character between the dense cluster and the sparse noise permits a simple and effective means for filtering out the latter. `VsSpeck` (figure 4-26) modifies a stream of binary images by eliminating true valued pixels which have little or no local "support".

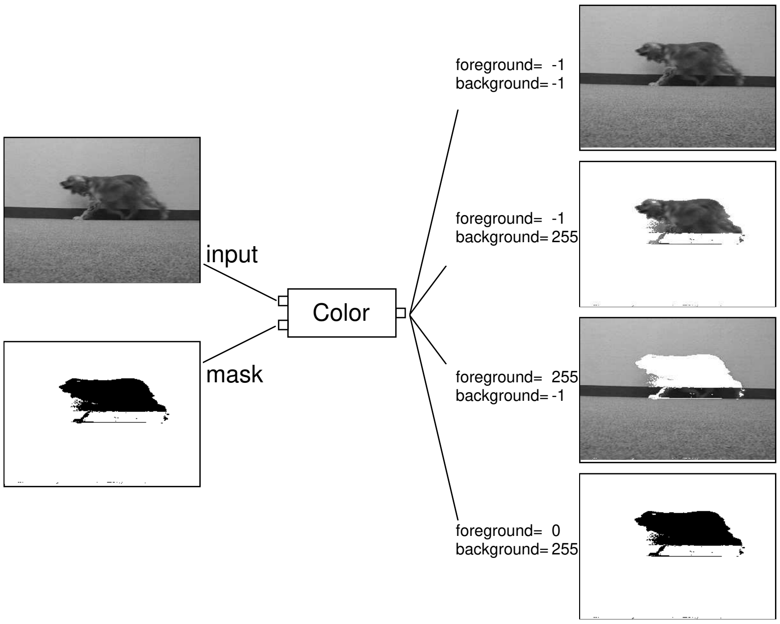



Figure 4-25: VsColor

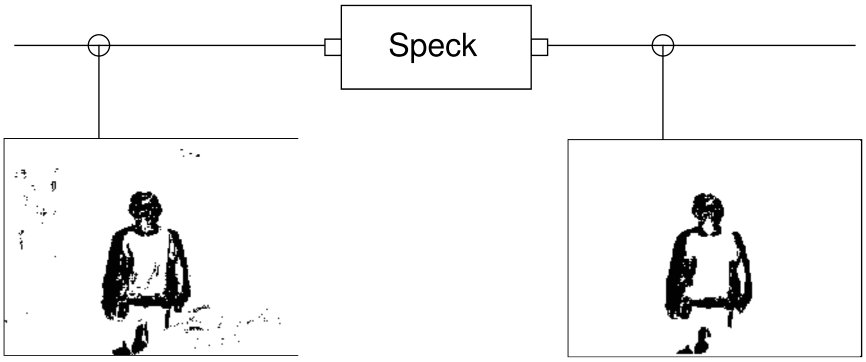
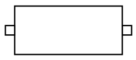



Figure 4-26: VsSpeck

`VsSpeck` employs two parameters, `window` and `threshold`. The `window` parameter specifies the size of a rectangular region defined around each pixel. A value of three, for instance, specifies a 3x3 region of points centered around the current pixel. `VsSpeck` goes to each true valued pixel in the binary image and counts the number of true valued pixels in its local region. If this number is less than the specified `threshold`, the pixel value is set to false. A window of 3 and a threshold of 1, for example, would eliminate all true valued pixels which have no immediate true valued neighbors.

The parameters for `VsSpeck` are summarized as follows:

| Parameter | Value | |
|---|---|---|
| `input` | *input_port* | Input port |
| `output` | *output_port* | Output port |
| `window` | *int* | Default 0 |
| `threshold` | *int* | default 0 |

### VsMaskFill 

`VsMaskFill` (figure 4-27) attempts to improve the segmentation computed by the `VsStatMotion` filter by filling gaps in the binary image or mask caused by the *blending* of the moving object with the background.

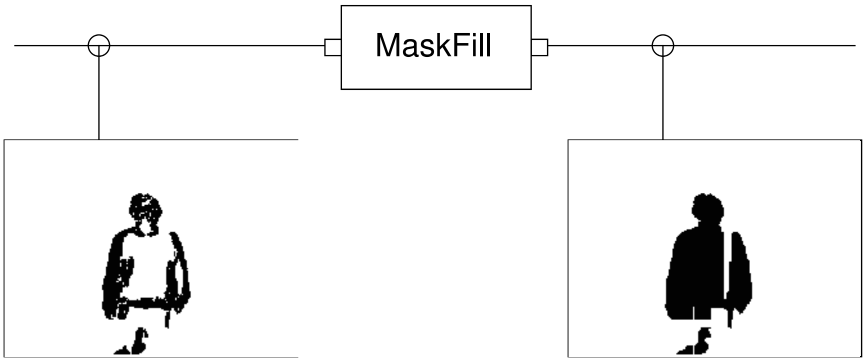



Figure 4-27: VsMaskFill

The algorithm used by `VsMaskFill`, though ad-hoc, does a credible job of filling the gaps and incurs low computational cost. The algorithm begins by computing four contours which define the top, bottom, left, and right boundaries of the region of true pixels in the image. Next, the algorithm optionally smoothes the four contours using a median filtering operation described below. Finally, the set of pixels falling between both the top and bottom contours *and* the left and right contours is calculated and output as the filled mask.

`VsMaskFill` employs a median filter to smooth the four boundary contours. The filter treats each contour as a one dimensional signal. In the case of the top and bottom contours, the signal defines, for each column, a displacement to the top or bottom edge. The median filter transforms this signal by replacing each value by the median value of points in its neighborhood. The number of columns in the neighborhood is specified with the `window` parameter. The left and right contours are treated analogously. A window value of 0 or 1 specifies that median filtering is not performed.

The algorithm for `VsMaskFill` works well on sequences where a single person or object is moving about a scene. While the filter does not correctly handle shapes with particular concavities, and was not designed for multiple objects, it performs reasonably well under such scenarios. Figure 4-28 demonstrates its ability in filling several oddly shaped regions.

The parameters for `VsMaskFill` are summarized as follows:

Figure 4-28: VsMaskFill output

| Parameter | Value | |
|---|---|---|
| input | *input_port* | Input port |
| output | *output_port* | Output port |
| window | *int* | Default is 0 |

## 4.5 Alternative Filters and Properties

This chapter has described how Sieve generates properties from unstructured video. The coverage has focused on a range of filters that have proven useful for the types of interactive applications considered in this report. However, the existing library suggests many ways that other types of filters and associated properties could be incorporated. For example, Sieve does not presently implement any modules based on 3D reconstruction from stereo. For some applications, three dimensional features could be grouped and hypothesis tested to generate properties representing correspondence to 3D models. For others, depth maps may be filtered to obtain binary image segmentations from which geometric and set properties may be derived. Other techniques for summarizing such information are certain to emerge. Thus, the descriptions of the current modules serve not only to document the existing library, but more importantly, as examples for how additional techniques may be adapted to the Sieve framework.

## 4.6 Summary

This concludes discussion of the processing that is used to perform direct manipulation and analysis of the "bits" in the media stream. The next chapter describes ways in which the properties derived from such processing may be used to dynamically adjust the system. Chapter 6 then concludes discussion of the analytical processing, by describing the transformation from properties to symbolic events.

# Chapter 5

# Property Processing

The previous chapter described the transformation of unstructured video into properties. The next step is to transform these properties into events. Before taking this step, however, this chapter describes how properties are used to dynamically adjust the behavior of the system.

In short, properties are used by Sieve in three major ways:

1. Properties may effect the processing performed by a module. In particular, property values may be used to modify a module's configuration parameters.
2. Properties may be used to direct or route payloads through the processing network.
3. Properties and sequences of properties may be used to generate symbolic events.

The first two functions are discussed in this chapter. The third, generating events, represents a higher level of abstraction and is discussed in chapter 6.

## 5.1 Parameter Setting

The behavior of a processing module is a function of its user specified parameters and the incoming payload data. The payload data includes both the media bits and their associated properties, enabling modules to be programmed to use specific property values in their output calculations. While such an approach allows properties to effect the behavior of processing modules, a more flexible mechanism which puts interactive user input on an equal footing with processing output is desirable.

A cleaner approach to integrating processing derived information and user input is to allow properties to change a module's parameters. Modules written to adopt this approach derive all their symbolic input from their parameter settings. In this way, applications which use such modules may be written to allow user input, process derived information (e.g. properties), or both to effect each module's parameter values.

Sieve does not require a special mechanism to implement such an approach. Since properties are used to generate events, and event handlers can adjust parameter values, a mechanism already exists for allowing derived information to mix with user input. However, a serious shortcoming arises from the separation of event processing from media processing. Although this separation models the asynchronous nature by which participants interact with media processing and provides flexibility in implementation, the resulting asynchrony between event processing and media processing makes it impossible to design applications which tightly integrate these tasks.

The following example serves to clarify this limitation and to introduce Sieve's `watch` mechanism as a solution.

### Example: Tracking

This example describes the implementation of a simple tracker. The input to the tracker is a model object and a stream of images. The output is stream of properties describing the object's location. As such, the tracker could be directly implemented using one of the matching modules described in section 4.1. For example, a tracker built using the `VsHausdorff` module is shown in figure 5-1.

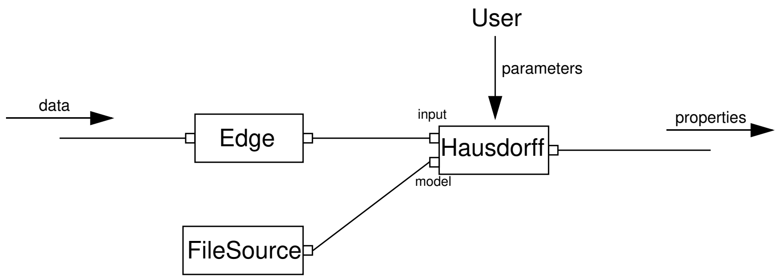



Figure 5-1: Fixed model tracker

Because the model is fixed, this implementation can only be used to track models with features that remain constant as the object moves about the scene. Unfortunately, the features generated by the edge filter used in this example do not meet this criteria. Rather, these features undergo constant change as the objects in the scene change viewing geometry, orientation, and even shape. While it is difficult to limit or parameterize such deformations (particularly those due to changes in the shape of non-rigid objects), in many cases it is reasonable to assume that such changes occur gradually. A tracker which updates the model data used on each successive match may take advantage of these gradual variations by setting the model data to be equal to the output data for the previous match. Figure 5-2 illustrates a simplified example of such an approach.

Two issues which arise are how to initialize the feedback loop and how to arrange for the region identified by the `VsHausdorff` matcher to serve as the model region for the subsequent match. The initialization is handled by placing a `VsDelay` filter (see section 4.3.1) in the feedback loop. The `VsDelay` module must be configured to insert an initial model image into the input stream to be compared to the first data payload.

The more difficult issue is how to transform the property output identifying the region of matched data into the parameter input identifying the region of the model. The approach depicted by the cloud in the figure 5-2 uses properties to trigger events (see chapter 6) which would be handled in turn by procedures that set input parameters. The difficulty previously

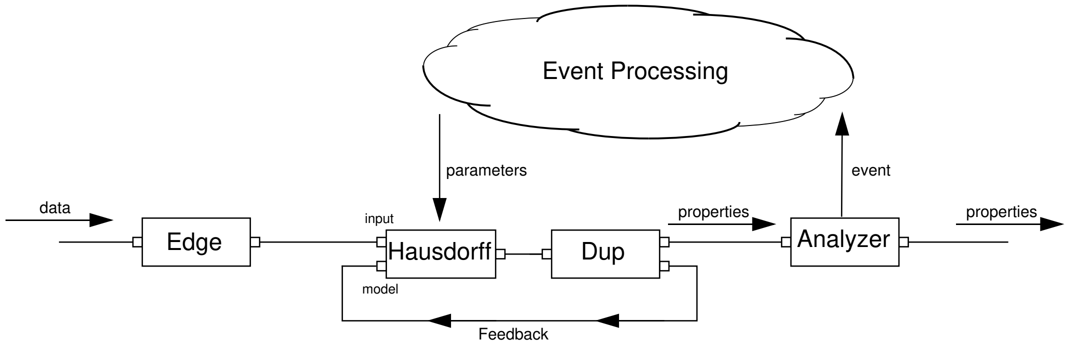



Figure 5-2: Event based tracker for updating model data

alluded to, (event handlers executing asynchronously with the processing of media data), prevents a guarantee that the event handler would be run before the arrival of the next input payload.

The solution adopted by Sieve is to allow payloads themselves to carry the parameter values. The mechanism for doing this is the `watch` method. This method allows a module to *watch* an input stream for a specified property symbol. Whenever a payload bearing that symbol is encountered, the module sets the appropriate input parameter to the value of an expression. The expression may include the values of other properties in the current input payload. Thus, any module may be configured to change its operating parameters on the basis of its input data.

The syntax for `watch` is as follows:

```
<module> watch {<property> <parameter> <expression>}
```

Typically, the expression will refer to the value of properties in the current payload. The values are accessed using the special Tcl function, `prop`. The `prop` function is special because it must executed from within the context of a payload. When executed as part of a `watch` expression, this context is the payload which includes the specified "watch" symbol. The `VsDerive` module, described in section 5.2, also defines a context for the `prop` function. Calling `prop` outside of a watch or derive expression results in an error.

The `prop` function takes as arguments the name of a property, its type and, optionally, a default value. The type is either a `float` or `int`. An alternative syntax is to use `prop.f` to return a floating point property and `prop.i` to return an integer property. The default value is the value that is returned if the property does not exist in the current payload.

For example, if `vs.motion` is the name of a `VsDiffMotion` module, then one may use the following command to cause the module's `threshold` parameter to be reset, whenever the module receives a payload with the property "x" defined, to be twice the value of "x".

```
vs.motion watch {x threshold {2 * [prop.i x]}}
```

Figure 5-3 illustrates the data flow for a working Hausdorff tracker based on the watch mechanism. The figure is similar to that of the previous example (figure 5-2) in that the properties produced by the `VsHausdorff` module are used to derive the new location of the model by setting the module's `modelRect` parameter. The new tracker differs, however, in that it *synchronizes* the parameter settings with the data processing by using the properties attached to the model data itself.

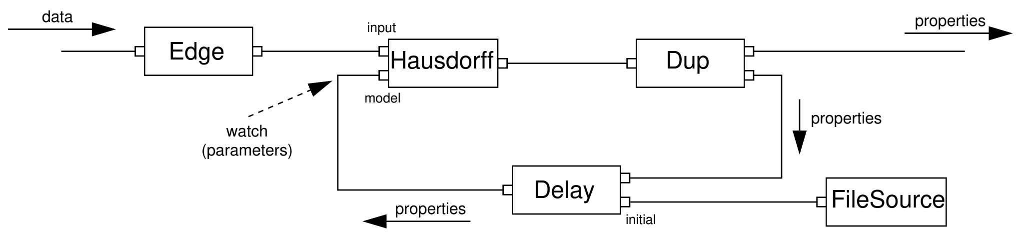



Figure 5-3: Watch based tracker

The figure also includes the `VsDelay` module described earlier that is used to initialize the model input of the `VsHausdorff` module. The script for the working tracker is shown in figure 5-4.

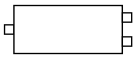
```
VsEdge vs.edge

VsHausdorff vs.haus \
  -input "bind vs.edge.output" \
  -property "haus" \
  -hold 0 \
  -modelRect $initialRect \
  -watch {haus {modelRect [list [prop.f haus.left] [prop.f haus.top] \
                                [prop.f haus.right] [prop.f haus.bottom]]}}
VsDup vs.dup \
  -input "bind vs.haus.output" \
  -numOutputPorts 2

VsFileSource vs.model \
  -pathname "$modelFile"

VsDelay vs.delay \
  -input "bind vs.dup.output1" \
  -initial "bind vs.model.output"
vs.delay.output connect vs.haus.model
```

Figure 5-4: Script for implementing Hausdorff tracking

## 5.2 Routing

Data flow computation depends upon the arrangement of processing modules in a flow graph. Routing may be used to make the arrangement that a payload encounters depend upon "flow time" factors. Such factors may be due to the internal characteristics of the data or due to external inputs such as those from an interactive user. In either case, modules which route payloads may be used to reduce the computational load on downstream modules by splitting flows so that payloads can avoid unnecessary processing. It is even possible, using network sources and sinks, to send different payloads to different machines for parallel computation.

The basis for routing in Sieve is the `VsClassify` module. `VsClassify` is a simple module that routes payloads to one of two output ports on the basis of a specified payload property. One may use this primitive module, often in combination with a `VsDerive` module or a `VsSync` module (see below) to build many types of routing modules and sorting networks.

### VsClassify 

`VsClassify` (figure 5-5) is a one-input, two-output module which directs each input payload to either its true output port or its false output port depending on the binary value of a specified property.

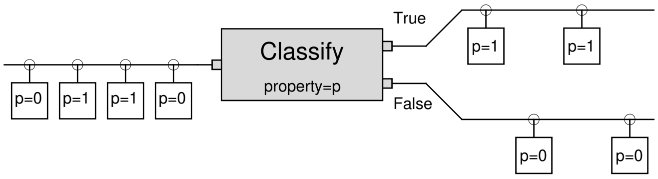


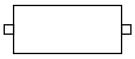

Figure 5-5: VsClassify

The name of the property is specified using the `property` parameter. The module examines each incoming payload for this property and passes it to its `output0` port if the integer value of the property is zero or to its `output1` port if it has any other value. In cases where the specified property is not defined, the module passes the payload to the port which is specified as the default. The `default` parameter is a boolean number which selects the `output0` or `output1` port.

The parameters for `VsClassify` are summarized as follows:

| Parameter | Value | |
|---|---|---|
| `input` | *input_port* | Input port |
| `output0` | *output_port* | False output port |
| `output1` | *output_port* | True output port |
| `property` | *string* | Name of the property to examine |
| `default` | {`0 1`} | Default output port |

The `VsClassify` module is fairly rigid in that it requires input properties to be boolean valued. Since not all properties are of this form, it is often necessary to transform properties from one type to another. The `VsDerive` module performs this function, in effect, acting as an impedance matcher which matches the form of the output produced by one module to the input required by another.

### VsDerive 

`VsDerive` generates properties by evaluating Tcl expressions. The module maintains a list of property names and associated expressions. Each time a payload arrives, the `VsDerive` module evaluates the expression for each property in the list and attaches the result to the payload as the value of the property.

The `derive` method is used to add entries to the expression list. Syntax for the method is as follows:

```
<module> derive {<property> <type> <expression> [<dependencies>]}
```

The "dependencies" is an optional list of property names that must be defined in the current payload for the derive expression to be executed.

For example, to derive a boolean property `fast` (represented by an integer) from a floating point property `speed`, one could use the following statement:

```
<module-name> derive {fast int {[prop.f speed] > 100}}
```

The `VsDerive` module's `clear` method, which takes no arguments, resets the expression list.

Finally, the `payloadType` limits processing to payloads of a specified type. For example, the value `VsVideoFrame` limits processing to video payloads. Setting this parameter to null (the empty string) causes all payloads to be processed.

The parameters and methods for `VsDerive` are summarized as follows:

| `Parameter` | `Value` | |
|---|---|---|
| `input` | *input_port* | Input port |
| `output` | *output_port* | Output port |
| `payloadType` | *payload_type* | Default is VsVideoFrame |
| `derive` | `prop type expr [dep1] [dep2] ...` | Method to add a derive expr |
| `clear` | *n/a* | Method to clear all expressions |

`VsClassify` is meant to be used as a primitive module from which higher level routing modules may be built. For example, one can easily generalize the boolean operation of the module to that which routes payloads on the basis of an integer or floating point property by using a `VsDerive` module to map ranges of integer or floating point values into booleans. Indeed, several `VsClassify` modules together with a `VsDerive` filter may be combined to act as an n-way classifier which routes payloads to any number of distinct output ports depending on the range of values into which an integer or floating point property falls. Figure 5-6 shows one such arrangement.

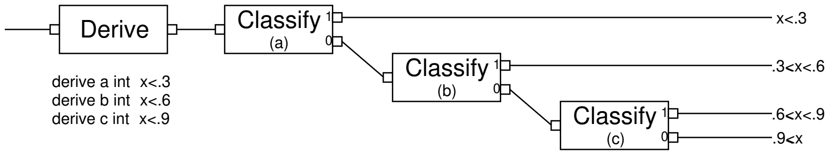
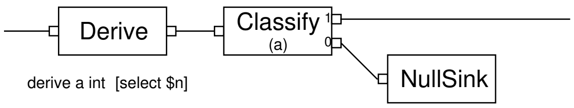



Figure 5-6: A 4-way Classifier built using VsClassify

It is also possible to use a `VsClassify` together with a `VsDerive` and `VsNullSink` module to build a compound module which temporally down-samples a stream of payloads. Figure 5-7 shows an arrangement which uses the `select` procedure (defined in the figure) to produce an output stream which consists of every *nth* payload from its input.

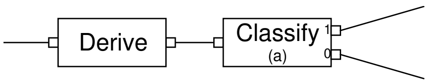



```
proc select {n} {
 global count
 return [expr  [incr count]%$n==0]
}
```

Figure 5-7: A temporal down-sampler built using VsClassify

The `select` procedure in the down-sampler is an example of a derive expression that defines state. In general, any derive expression may do so, thus enabling a property value to depend on previous values of properties.

Finally, figure 5-8 shows how a `VsClassify` and `VsDerive` filter may be used to implement a switch that routes payloads in the direction set by the last call to the `switch` procedure (defined in the figure).




```
$m proc switch {out} {
  $self.derive derive a int  $out
}
```

Figure 5-8: A switch built using VsClassify

One computational task for which the `VsClassify` module has proven to be particularly useful is sifting through an input stream for payloads which meet a specific criteria. The following describes how such computation may be applied to the problem of annotating video.

### Example: Video Annotations

This section describes the Sports Highlight Browser, a special purpose application, implemented using Sieve, that was used to browse CNN's nightly sports broadcast. The browser, shown in figure 5-9, enabled users to retrieve the video highlights for a particular sporting event. The application ran continuously as a demonstration on the World Wide Web for a period of roughly a year beginning in early 1994 [Stasior and Tennenhouse, 1996] [Lindblad *et al.*, 1995].

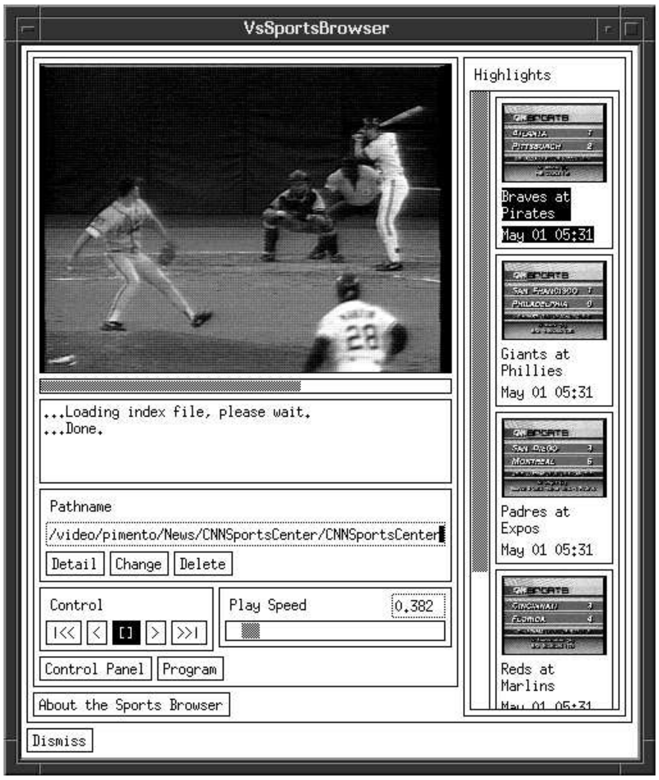



Figure 5-9: Sports Highlight Browser

The browser made use of annotations which labeled specific images in the sports broadcast. The browser took advantage of the broadcasting cliche whereby the viewer would first be shown an announcer introducing the sporting event, then shown highlights from the event, and finally shown a scoreboard graphic summarizing the results (figure 5-10). This cliche would be repeated throughout the broadcast. The task of the annotation generator, therefore, was to identify which frames were scoreboards and to recognize which teams appeared in the scoreboard graphic. The highlight for a particular event, a Boston Celtics basketball game for example, was thus assumed to be the video clip falling between the Celtics scoreboard and the previous scoreboard.

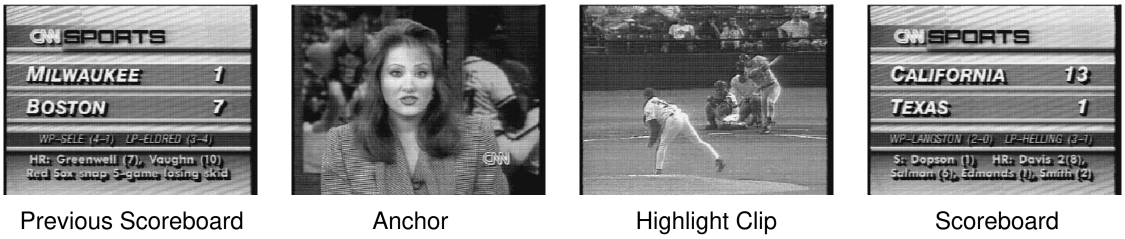
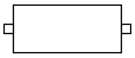



Figure 5-10: Sports highlight cliche

The graphics themselves were identified using a bank of template matching filters. A template matcher compares a stream of input payloads to a fixed model. `VsTemplate` is an abstraction which allows the combination of a two-input matcher with a `VsFileSource` to be treated as a simple, one-input, one-output filter. `VsTemplate` may be used with any two-input matcher that supports the necessary interfaces. The `VsPixelMatch`, `VsHistMatch`, and `VsHausdorff` are all examples of modules which may be used by `VsTemplate`.

### VsTemplate



A common use of a matching module is to compare a stream of payloads to a fixed model. When used in such a way, the two-input matching module effectively becomes a single input, single output filter which attaches properties to the input stream.

`VsTemplate` (figure 5-11) is a compound module which differs from most such modules in that it takes the name of another module as an input parameter. The `matchMod` parameter is a list whose first element is the name of a matching module. The rest of the list are the parameter names and values which are passed to the matching module. For example, the list {`VsHistMatch -binMethod absDiff`} would specify the matching module to be a histogram comparator configured such that its `binMethod` parameter was set to `absDiff` (see section 4.1.2).

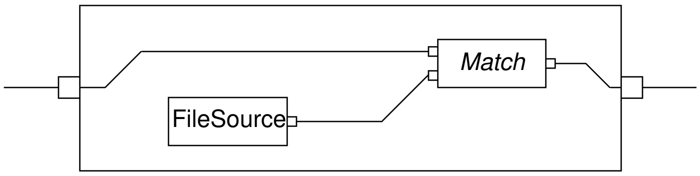



Figure 5-11: VsTemplate

The match module must be a module with two input ports labeled `input` and `model` and an output port labeled `output`. In addition, the match module must define the `hold` and `property` parameters. These parameters are not set in the `matchMod` list. Rather, they are set as follows: `input` and `output` are aliased to be the input and output ports for the compound filter, `model` is connected to the output port of the `VsFileSource`, `hold` is set to 1, and `property` is set to the value of the `VsTemplate` `property` parameter.

The `pathname` parameter is the name of the file containing the model. The value of this parameter is passed directly to the `VsFileSource` component.

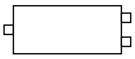
The parameters for `VsTemplate` are summarized as follows:

| Parameter | Value | |
|---|---|---|
| `input` | *input_port* | Aliased to the match module's input port |
| `output` | *output_port* | Aliased to the match module's output port |
| `pathname` | *string* | VsFileSource pathname |
| `property` | *string* | Match module's property |
| `matchMod` | *list* | Match module's name and parameters |

For the purpose of detecting specific sports graphics, `VsTemplate` was used in conjunction with a `VsPixelMatch`. Each pixel matching template filter was configured to search a specific region of the image for a particular graphic in the database.

`VsClassify` modules were used to organize the bank into an efficient network where payloads which match a given template are treated differently from those which do not. Thus, for example, video frames which are identified as being a scoreboard were further examined to see what teams are represented, while frames which did not depict a scoreboard were tested for other properties or simply ignored. The `VsClassify` modules were incorporated by combining them with a template filter and a `VsDerive` filter in an abstract module, referred to as a `VsClassProp`. The `VsClassProp` module served as a unit whereby a property could be computed and immediately used to route a payload. These units acted as the building blocks for the annotation network.

### VsClassProp



`VsClassProp` (figure 5-12) combines a property generating filter with a `VsDerive` and a `VsClassify` module to produce a compound module which routes payloads on the basis of a specific property measurement. The module accepts a stream of input payloads and routes them to either its `output0` or `output1` port on the basis of a boolean expression.

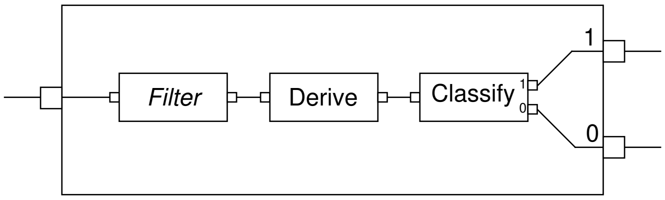



Figure 5-12: VsClassProp

The `propMod` parameter is treated in a manner similar to the `matchMod` parameter of `VsTemplate`. The `propMod` parameter is a list whose first element is the name of a one-input, one-output filter; while the rest of the list are parameter names and values which are passed to the filter itself. The `propMod` module is required to support the parameters `input` and `output`. The `input` parameter is used as the input to the `VsClassProp` module while the `output` is connected to the `VsDerive` filter's input port.

The `VsDerive` filter is used to generate the property which routes the payload. The name of the generated property is specified using the `property` parameter while its value is computed by evaluating the `expression` parameter. Typically, the expression will involve properties generated by the `propMod` module.

The parameters for `VsClassProp` are summarized as follows:

| Parameter | Value | |
|---|---|---|
| `input` | *input_port* | Aliased to the property filter's input port |
| `output0` | *output_port* | Aliased to the classify's output0 port |
| `output1` | *output_port* | Aliased to the classify's output1 port |
| `property` | *string* | Property filter's property |
| `expression` | *tcl_expression* | Tcl expression evaluating to 1 or 0. |
| `propMod` | *list* | Property filter's name and parameters |

One of the most important aspects of the `VsClassProp` module is that it is easily composed. In particular, one may use several `VsClassProp` modules as components in a higher level module that efficiently classifies payloads on the basis of boolean operators such as *and* and *or*. The higher level module supports the same interfaces as the lower level `VsClassProp` module and thus itself may be composed.

The `VsClassProp` modules were used in stages to sort and ultimately identify payloads. The first stage classified video payloads into those which were scoreboards and those which were not. A `VsClassProp` module, which incorporated a `VsTemplate` filter, which itself was built using a `VsPixelMatch`, was used to detect the presence or absence of a standard graphic in the upper left corner of the image. Those which matched were classified as scoreboards while those which did not were passed along. It was soon discovered, however, that the scoreboard graphic changed regularly and that on any given day, any one of three different graphics could be used (figure 5-13).

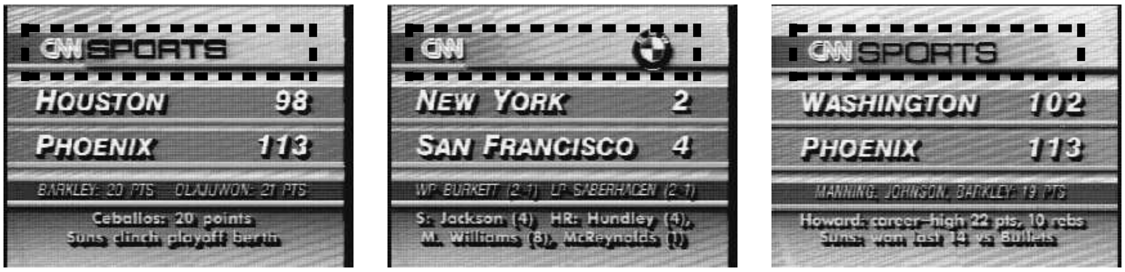



Figure 5-13: Three different graphic markers

In order to handle the different possibilities, payloads which failed to match the original graphic were subsequently tested against the second and then the third template. The resulting arrangement is shown in figure 5-14. In this arrangement, the three `VsClassProp` modules and a `VsMerge` module act as a compound module which classifies a payload as a scoreboard if it matches any of the three templates. The computation is efficient in that only payloads failing to match the first template are matched against the second template and so on.

The mechanism for implementing this arrangement is `VsClassifyOr`, an abstract module which combines a list of `VsClassProp` modules (or any modules that implement the `VsClassProp` input port, output port interface) into a compound module. The

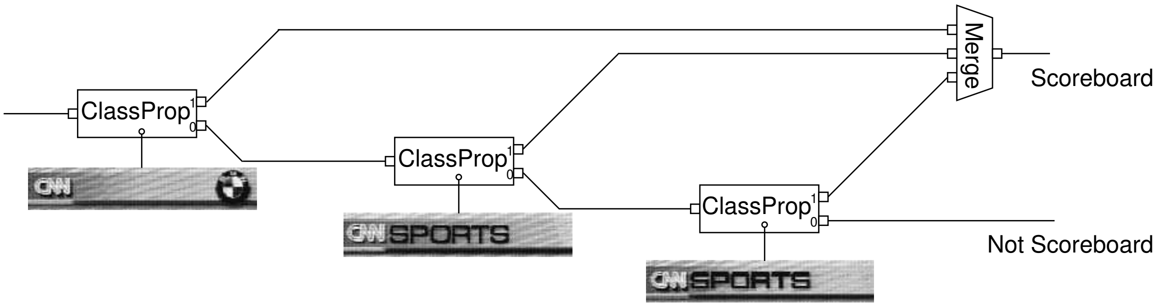
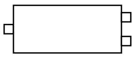



Figure 5-14: Classifying scoreboards

`VsClassifyOr` module itself implements the `VsClassProp` port interface, as does its counterpart `VsClassifyAnd`. It thus possible to build a `VsClassifyOr` module which includes `VsClassifyAnd` modules which include `VsClassifyOr` modules, and so on.

### VsClassifyAnd, VsClassifyOr



`VsClassifyOr` and `VsClassifyAnd` are abstract modules, each of which, like the primitive module `VsClassify` and the compound module `VsClassProp`, processes payloads arriving at its `input` port by routing them to either its `output0` or `output1` port. Both modules take a list of such classifiers as their `mods` parameter and arrange them in the manner shown in figures 5-15 and 5-16.

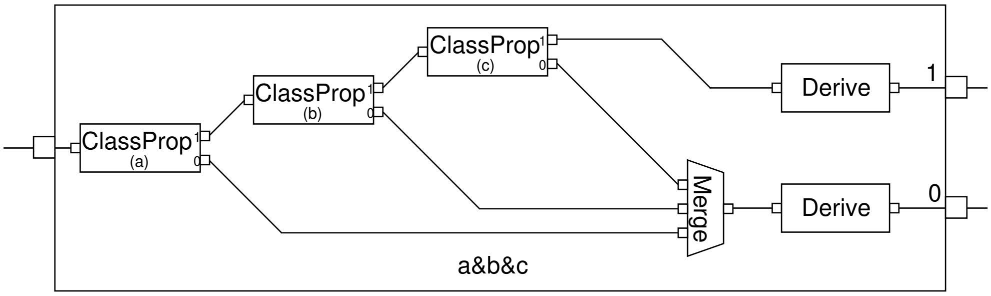



Figure 5-15: Compound module for computing A and B

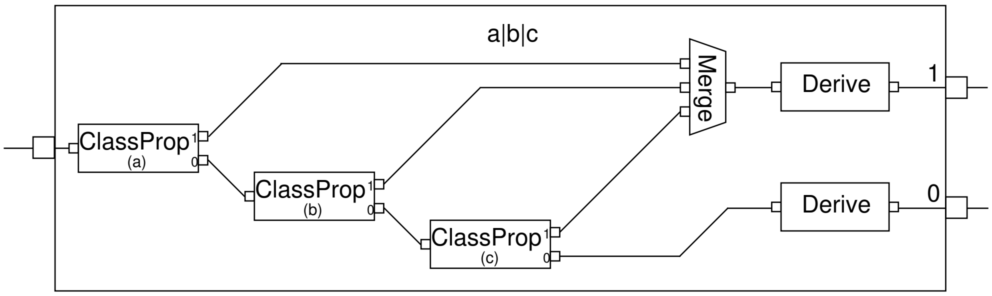



Figure 5-16: Compound module for computing A or B

The result of the arrangement for the `VsClassifyAnd` module is to route payloads to the compound module's `output1` port if and only if all of the component classifiers route the payload to their `output1` port. Similarly, the `VsClassifyOr` module's arrangement routes a payload to the compound modules `output1` port if any of the component classifiers route the payload to their `output1` port.

Though not shown in the figures, one may arrange for payloads passing through the compound module to be tagged with a boolean property that is set to 1 if the payload is routed to the `output1` port and 0 if the payload is routed to the `output0` port. The `property` parameter supplies the name of this property. When this parameter is set to a value other than the empty string, two `VsDerive` modules are placed just before the two output ports to tag each outgoing payload.

The parameters for both `VsClassifyAnd` and `VsClassifyOr` are summarized as follows:

| Parameter | Value | |
|---|---|---|
| `input` | *input_port* | Input port |
| `output0` | *output_port* | False output port |
| `output1` | *output_port* | True output port |
| `property` | *string* | Property name |
| `mods` | *list* | List of classifiers |

In addition to the scoreboard graphic, a library of graphical templates was developed which identified each team[1]. As was the case with the scoreboard graphics, multiple graphics were sometimes used to represent the same team (figure 5-17). Thus, some teams were classified by a singular `VsClassProp` module while others were classified by a `VsClassifyOr` module built from multiple `VsClassProp` modules.

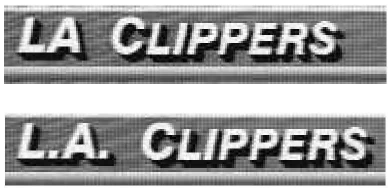



Figure 5-17: Two templates for the same team

Using these classification modules for scoreboards and teams, the annotation network was defined roughly as follows:

```
VsClassifyAnd
  <Scoreboard>
  <Scheduled Game>
```

Thus, each payload that was classified as a scoreboard would subsequently be tested to see whether it represented an athletic contest of interest from the day's schedule. While it was not necessary to first test each payload to determine if it represented a scoreboard, the test

[1]Although optical character recognition could have been employed to identify these particular templates, it was not used for two basic reasons: lack of a freely available, high quality ocr library and an interest in maintaining the flexibility to recognize teams represented as non-textual icons.

was included for efficiency since it allowed most payloads to be skipped by the subsequent test to see if the payload represented a contest of interest. The list of such contests would vary but might consist of the day's pro basketball games or the day's major league baseball games or a subset of such contests.

As stated previously, the classification of a payload as a scoreboard was performed by a `VsClassifyOr` module which sorted input payloads on the basis of comparisons to three templates. Similarly, the classification of a payload as one representing a contest of interest was also performed by a `VsClassifyOr` module which tested to see if payloads matched the expected layout for any individual contest.

```
VsClassifyAnd
  VsClassifyOr
    VsClassProp Scoreboard1
    VsClassProp Scoreboard2
    VsClassProp Scoreboard3
  VsClassifyOr
    <Game1>
    <Game2>
    <Game3>
```

The pseudo-code, "VsClassProp Scoreboard1", is shorthand for a `VsClassProp` module defined using a `VsTemplate` filter which compares a particular scoreboard graphic to each input image. The actual code for such a module would be as follows:

```
VsClassProp \
  -propMod {
    VsTemplate -pathname scoreboard1.uv
      -matchMod {
        VsPixelMatch
          -modelRect {$mLeft $mTop $mRight $mBottom}
          -inputRect {$iLeft $iTop $iRight $iBottom}}}
```

where mLeft, mTop, mRight, mBottom, iLeft, iTop, iRight, and iBottom are parameters which specify the model and input rectangles.

Finally, the test for whether a scoreboard matched a particular contest was performed by a `VsClassifyAnd` module which tested to see whether the scoreboard contained both the home team and away team in their proper locations (traditionally, the home team always appears below the away team). For example, the following expression would be used to search for highlights of basketball games on a night where the Celtics were scheduled to play the Clippers, the Knicks were scheduled to play the Suns, and the Bulls were scheduled to play the Rockets:

```
VsClassifyAnd
  VsClassifyOr
    VsClassProp Scoreboard1
    VsClassProp Scoreboard2
    VsClassProp Scoreboard3
  VsClassifyOr
    VsClassifyAnd -property CelticsAtClippers
      VsClassProp Celtics (away)
      VsClassifyOr
        VsClassProp Clippers1 (home)
        VsClassProp Clippers2 (home)
    VsClassifyAnd -property KnicksAtSuns
      VsClassProp Knicks (away)
```

```
    VsClassProp Suns (home)
  VsClassifyAnd -property BullsAtRockets
    VsClassProp Bulls (away)
    VsClassProp Rockets (home)
```

The `property` parameter used in this example causes the particular module to attach a true valued boolean marker to payloads that are classified as positive examples.

Figure 5-18 depicts the hierarchical arrangement for this classification network. The entire network acts as a classification module which takes a stream of image payloads and sorts them into two streams, one containing frames which depict scoreboards of interest and the other containing all other frames. More importantly, the network acts to annotate the payloads by attaching properties which identify the scoreboard payloads. One may, in fact, merge the two output streams to produce a one-input, one-output filter which annotates payloads.

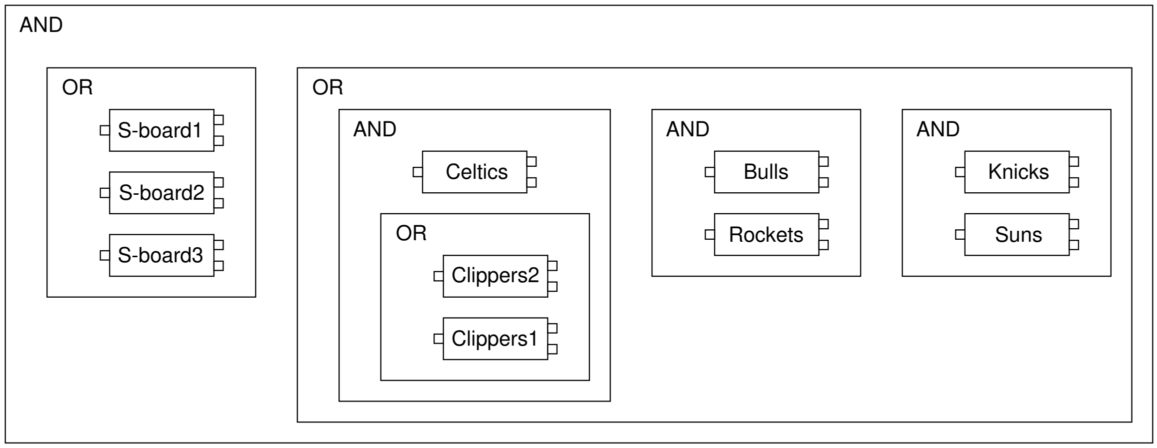
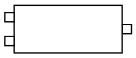



Figure 5-18: Classification network

There is, however, one important caveat to such use of a classification network. Namely, payloads passing through such a network can become reordered. The reason for this is that the different payloads travel along different paths which typically have unequal delays. In such scenarios, the `VsSync` module may be employed.

### VsSync



`VsSync` (figure 5-19) takes two input streams, one of which is assumed to be ordered and one of which can be unordered. By default, `VsSync` simply passes through the ordered stream while discarding payloads in the unordered stream. The utility of `VsSync` comes from the fact that `VsSync` does not pass a payload from the ordered stream until a payload with the same timestamp arrives from the unordered stream. `VsSync` thus insures that the ordered payload does not proceed until its unordered counterpart has passed through the processing network. This can be useful when the payloads in the ordered stream and the unordered stream are in fact shallow copies of one another. In such cases, the properties accumulated by the unordered payloads exist for the ordered payloads by the time the ordered payloads are forwarded through the `VsSync` module.

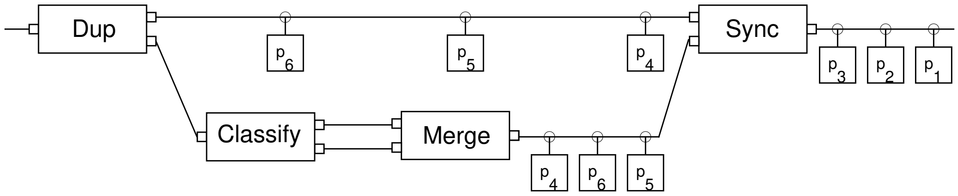



Figure 5-19: VsSync used to "reorder" output from a classification network

By default the ordered payloads are passed while the unordered payloads are deleted. Alternatively, one may use the `ordered` parameter to specify for the payloads arriving at the unordered port to be passed. In either case, the payloads will leave `VsSync` in order. In the latter case, however, any transformations performed on the unordered stream will be preserved. Setting `ordered` to 1 selects the ordered stream while setting the parameter to 0 selects the unordered stream.

The parameters for `VsSync` are summarized as follows:

| `Parameter` | `Value` | |
|---|---|---|
| `inputOrdered` | *input_port* | Ordered input port |
| `inputUnordered` | *input_port* | Unordered input port |
| `output` | *output_port* | Output port |
| `ordered` | {`0 1`} | Select which stream to pass |

## 5.3 Summary

This section has described two ways that Sieve makes use of properties: parameter setting and routing. Both mechanisms were illustrated by example. The Tracker used the parameter setting mechanism to update model parameters as a tracked object moves throughout a scene. The Sports Highlight Browser used a classification network to route images to processing modules depending on their properties.

The next chapter describes a third way that Sieve uses properties: to generate symbolic events.

# Chapter 6

# Generating Events

In this work, the problem of recognizing events in media streams has been addressed in terms of matching patterns of properties of media input. Patterns are defined across sequences of images and thus serve as the key mechanism for explicitly representing inter-frame relationships. Pattern matching is performed by filters monitoring streams of properties for specific patterns and signaling events when those patterns occur (figure 6-1).

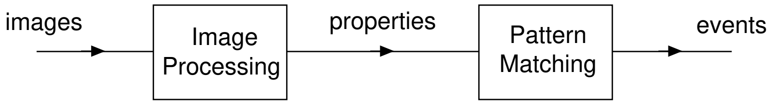



Figure 6-1: Generating events from images

This chapter is divided into three parts. The first part develops a methodology for recognizing media patterns. The approach is based on regular patterns which are recognized by finite state automata [Hopcroft and Ullman, 1979]. The second part describes `VsVex`, a Sieve module which implements this methodology. The third part presents an example applet which uses `VsVex` to recognize a simple gesture.

## 6.1 Methodology

The problem of matching patterns of symbols in an input stream is well studied in computer science. Formally, a string is a sequence of symbols from a specified alphabet and a language is a set of strings. A pattern is a notation for a language. A pattern matcher is an automata which examines an input stream and computes whether the input represents a string in the language.

Traditionally an input string is a sequence of symbols from a limited alphabet whereas the raw input from a stream of video is a sequence of images. In order to apply the tools from computational language theory one must transform the stream of media into something that resembles a stream of symbols.

For this purpose this thesis employs *properties* and *predicates*. Properties are measurements that are made on the input stream. Predicates are boolean functions of properties which map payloads in the input stream to true or false. Predicates act as the symbols in "media strings".

Once media has been transformed into a sequence of predicates, one may apply standard techniques for matching patterns. In particular, one may implement an automata which scans the input for sequences which comprise a formal language.

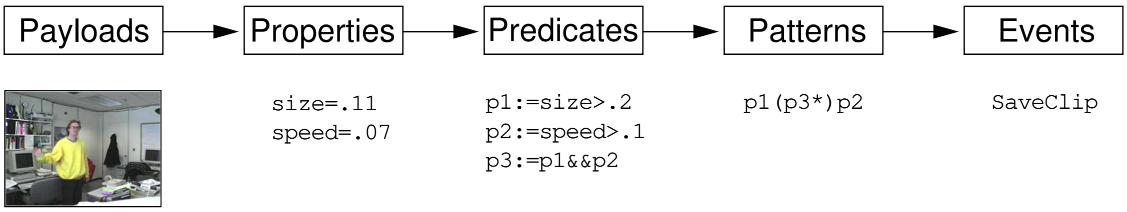



Figure 6-2: Processing methodology

A methodology (illustrated in figure 6-2) for recognizing patterns of media is, therefore, to transform streams of payloads into streams of properties. Properties are in turn used to compute boolean valued predicates. Sequences of predicates are grouped into media strings which are passed as input to pattern matching automata. Finally, the pattern matching automata signal events in response to recognizing sequences of input.

### 6.1.1 Properties

The problem with treating a stream of media like a stream of characters is that media streams contain vast amounts of information. It is impractical, for example, to enumerate all possible images and it is overly restrictive to compare images directly. This thesis addresses this problem by reducing images and media to their properties.

The issue of whether a sparse set of properties may be used to represent the important information in an image is one which is likely to be raised by a methodology which relies on such an approach. This thesis does not argue that such a representation is sufficient to describe all the information in an image. Rather, this thesis uses properties as a means towards reducing media to a sequence of discrete actions. Since these actions are relatively sparse in their occurrences and descriptions, it is reasonable to assume that properties may serve as a sufficient representation at some stage in this reduction.

### 6.1.2 Predicates

A predicate is a boolean function which maps vectors of properties in a payload to true or false. Whereas in a text stream a given character is said to be equal or not equal to a particular symbol, in a media stream a payload is said to be true or false for a given predicate.

Predicates serve two important functions. First, predicates act as thresholding functions which map numerical properties to boolean valued symbols which may be processed directly by simple automata. Second, predicates provide a means of combining different properties, within a payload, into a single value. The following sample predicates demonstrate these two uses:

```
[prop.f motion] > 0.2
[prop.f objSize] * [prop.f objSpeed] < 0.8
[prop.i John] == 1 && [Weekday] == Monday
```

The important difference between payload predicates and text symbols is that payload predicates are non-exclusive. In other words, while an input character is equal to exactly one symbol, an input payload may be true for more than one predicate.

### 6.1.3 Sequences

While individual images are matched against predicates, streams of images are matched against *sequences*. A sequence is a concatenation of predicates which must be true, in series, for a sequence of images. Thus, just as a predicate corresponds to a symbol in a formal language, a sequence corresponds to a formal string.

The main difference between a string and a sequence is that a string is a fixed series of mutually exclusive symbols. In other words, each element in a string is "true" for one symbol and false for all others. Sequences, on the other hand, inherit the non exclusive nature of predicates. An element in a sequence may be true for any number of predicates.

For example, assuming that the predicates $p_1$ and $p_2$ are defined for all possible input images, then the sequence "$p_1\ p_1\ p_1\ p_2\ p_1$" would match any series of five images where the first three images were true for $p_1$, the fourth was true for $p_2$, and the fifth was true for $p_1$. It does not matter whether the first three images (or the fifth) are true for $p_2$ or whether the fourth image is true for $p_1$. If this were important, i.e. if the fourth image should match $p_2$ and not $p_1$, then one would need to define a new predicate, $p_3$ such that:

$$p_3 = p_2 \wedge \neg p_1$$

In general, if a sequence requires a single image to be tested against multiple predicates, one must define a compound predicate against which the image is tested.

In summary, predicates represent combinations of properties within an image while sequences represent combinations of images within a stream.

### 6.1.4 Language

In order to recognize patterns of predicates, one must have a language and a notation for specifying patterns. Formally, a language represents a set of symbolic strings. A useful language must have a concise means of representing rich sets of strings. For our purposes a language must also have an efficient means of computing whether a particular string is a member of the language. Regular languages meet both these criteria.

Regular languages and their notational counterparts, regular expressions, have proven to be an effective tool for parsing text. This thesis uses regular expressions to represent sets of predicate sequences in a manner analogous to that of text parsers which use regular expressions to represent sets of strings.

The three fundamental operators in a regular language are concatenation, closure, and union. As discussed in the previous section, the concatenation of two sequences is the sequence formed by appending the second sequence to the first. Likewise, the concatenation of two languages, $L_1$ and $L_2$, is the language which is formed by taking all the sequences of the second language and appending each one to every sequence in the first language.

$$concat(L_1, L_2) = \{l_1 l_2 | l_1 \in L_1 \ and \ l_2 \in L_2\} \tag{6.1}$$

The notation or pattern for concatenation is simply the juxtaposition of the two patterns.

The closure of a language $L$ is the language that contains all sequences which are formed by zero or more concatenations of sequences in $L$. A recursive definition of a closure is given by the following equation:

$$closure(L) = \{\emptyset, L, concat(L, closure(L))\} \tag{6.2}$$

Closure of a pattern is denoted by appending the * operator. The * operator applies to the element immediately to its left so that $p_1 p_2 *$ denotes $p_1$ concatenated with the closure of $p_2$. To denote the closure of $p_1 p_2$, parentheses must be used.

$$p_1 p_2 * = \{p_1,\ p_1 p_2,\ p_1 p_2 p_2,\ p_1 p_2 p_2 p_2,\ ...\} \tag{6.3}$$

$$(p_1 p_2) * = \{\emptyset,\ p_1 p_2,\ p_1 p_2 p_1 p_2,\ p_1 p_2 p_1 p_2 p_1 p_2,\ ...\} \tag{6.4}$$

Finally, the union of two languages is the language formed by the union of all the sequences in the two languages. The union of two patterns is specified by placing the | operator between the patterns.

$$union(L_1, L_2) = \{l | l \in L_1 \ or \ l \in L_2\} \tag{6.5}$$

Unlike the * operator, the notational convention for the | operator is for the union to apply to as wide a pattern as possible so that $p_1 p_2 | p_2 p_1 *$ would specify the union of the languages $p_1 p_2$ and $p_2 p_1 *$. Parenthesis may be used to limit the scope of the | operator so that $(p_1 p_2 | p_2) p_1 *$ would specify the concatenation of $p_1 p_2 | p_2$ with $p_1 *$.

$$p_1 p_2 | p_2 p_1 * = \{p_1 p_2,\ p_2,\ p_2 p_1,\ p_2 p_1 p_1,\ ...\} \tag{6.6}$$

$$(p_1 p_2 | p_2) p_1 * = \{p_1 p_2,\ p_2,\ p_1 p_2 p_1, p_2 p_1,\ p_1 p_2 p_1 p_1,\ p_2 p_1 p_1,\ ...\} \tag{6.7}$$

In summary, a pattern represents a set of predicate sequences. This thesis implements regular patterns as a notational means of concisely specifying such sets.

### 6.1.5 Automata

A language and a notation for describing sequences of input is of little use unless there is a computational means of differentiating between inputs in the language and inputs not in the language. An automata is a computational engine that scans an input string and determines whether that string is in the language. The class of automata for recognizing regular languages is well understood and widely used.

Regular languages are a subset of all possible languages, meaning that not every set of inputs can be described by a regular language. Indeed, some useful languages, such as the set of all strings with balanced parentheses or the set of all strings which are palindromes are famous examples of languages which are not regular. Still, regular languages are extremely expressive, including for instance, the set of all strings of a given finite length which are a palindrome or the set of all strings of a given finite length which contain matching parentheses, in addition to many infinite sets of strings. Furthermore, while these limitations could be avoided by adopting a different language and automata (such as a push down automata) than the finite state automata described below, there is no formal advantage in doing so because one may define predicates functions with arbitrary state to augment the pattern matching capabilities.

Finite state automata are the computational equivalents to regular languages. That is, the set of languages which are regular is precisely the set of languages which can be computed by a finite state automata. Furthermore, simple, automatic techniques exist for transforming the description of a regular language into a finite state automata which can efficiently compute the language.

A finite state automata is an automata whose behavior can be summarized by a set of states, S, and a state transition function, T. The set of states includes a start state, $s_0 \in S$ and subset of "accept" states, $A \in S$. The state transition function is a mapping from S and the input alphabet, $\Sigma$ to S or the null state, $\perp$:

$$T : S \times \Sigma \rightarrow S \cup \perp \tag{6.8}$$

Finite state automata are either deterministic or nondeterministic. A deterministic finite state automata, DFA, operates as follows: beginning with the start state, $s_0$, the automata reads the first input symbol, $x_0$ in the sequence and applies the transition function to $s_0$, $x_0$ to calculate the new state $s_i$. Continuing from this state, the automata continues to read input symbols and make transitions from state to state. Eventually, one of three things happens: the automata transitions to an accept state, $s_i \in A$, the automata runs out of input, or the automata transitions to a null state. In the first case, the automata is said to accept the input sequence up to and including the most recently read character. If the automata runs out of input without reaching an accept state or transitions to a null state, the automata is said to reject the sequence.

A DFA may be represented by a state transition graph. The nodes in the graph represent states and the directed edges represent transitions. The start node is depicted by an arrow which points into the state but does not originate from any other state. Accept states are depicted by concentric circles. Normally, transitions to the null state are not shown. Figure 6-3 shows an example of a state transition graph for an example DFA.

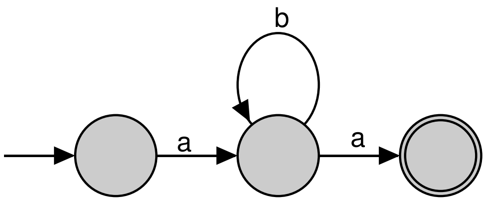



Figure 6-3: A DFA which accepts the strings "aa", "aba", "abba", "abbba", ...

A nondeterministic finite state automata, NFA, is a generalization of a deterministic finite state automata whereby the automata may be in several states simultaneously. The transition function for a non-deterministic automata is a function that maps a state and input symbol to a subset of states in S:

$$T : S \times \Sigma \rightarrow 2^S \quad {}^1 \tag{6.9}$$

The name "nondeterministic" reflects the fact that an NFA can be in many states simultaneously. Theoretically, one may view each transition from one state to multiple states as a non-deterministic branch to multiple parallel DFAs. An NFA is said to accept its input if any one of its nondeterministic branches results in the input being accepted. Like a DFA, an NFA may be represented by a state transition graph with the only difference being that a state may have multiple identically labeled edges leading out of it. Figure 6-4 shows the state transition graph for a sample NFA.

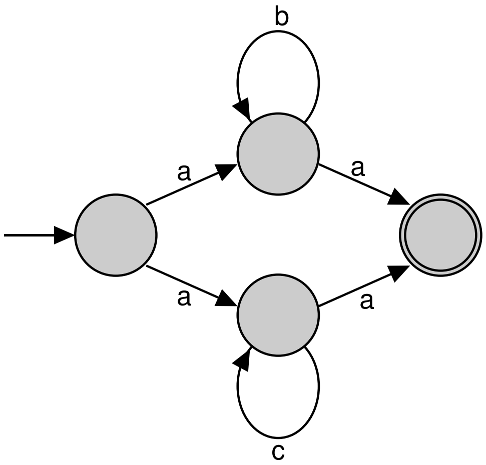



Figure 6-4: An NFA which accepts the strings "aa", "aba", "aca", "abba", "acca", "abbba", "accca", ...

A very important property of a DFA is that the effect of its entire computational history can be summarized by its current state. This allows for an efficient deterministic simulation of an NFA in which parallel branches which arrive at the same state on the same input are collapsed into a single branch. The algorithm for simulating an NFA is, therefore, as

[1] $2^S$ denotes the power set of S

follows: Beginning with a state list of one or more start states, the automata reads the first input symbol in the sequence and applies the transition function to each state/symbol pair. The union of all the transition function results is subsequently used as the new state list for the simulation. The simulation continues to transition from state list to state list as it consumes its input. The automata is said to accept the input sequence up to and including the most recently read symbol whenever the state list includes an accept state. The automata is said to reject the input sequence whenever the state list is reduced to null or when the automata runs out of input.

It is well known that the class of languages which may be computed by NFAs is no different from the class of languages which may be computed DFAs. Nevertheless, the generality of the NFA model often proves to be helpful in specifying a computational task. In particular, this is the case when transforming a regular expression to a finite state automata. While it is possible to transform a regular expression to a DFA, such a transformation may in principle require a number of states which grows exponentially with respect to the length of the expression.

The addition of $\epsilon$ transitions is another generalization of the finite state automata model which does not change the class of languages which may be computed but proves be helpful, particularly for transforming regular expressions. $\epsilon$ transitions are non-deterministic transitions which, unlike the transitions described thus far (which will hereafter be referred to as alpha transitions) can occur in the absence of input. An $\epsilon$ transition acts very much like a branch and forwarding pointer in the sense that an NFA that arrives in a state $s_i$ that has an $\epsilon$ transition to $s_j$ is thereafter in states $s_i$ and $s_j$. There is no limit to the number of $\epsilon$ transitions an NFA may make between inputs. Thus, if $s_j$ has an $\epsilon$ transition to $s_j$, the NFA would be in states $s_i$, $s_j$, and $s_k$. Figure 6-5 depicts an NFA with $\epsilon$ transitions.

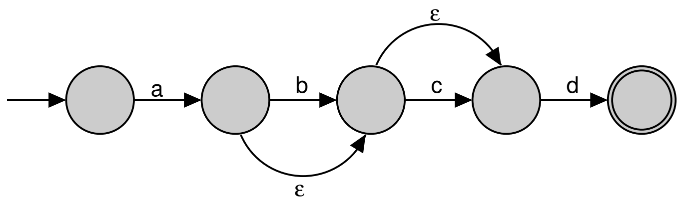



Figure 6-5: An NFA with $\epsilon$ transitions which accepts the strings “ad”, “abd”, and “abcd”

The NFA model is particularly well suited for recognizing languages of predicates. Thus far, the NFA and DFA models presented in this section have taken as input a stream of symbols. As emphasized previously, the fundamental difference between a language of predicates and a language of symbols is that for a given input, more than one predicate may be true. This poses little problem for the NFA model whereby one state may transition to multiple states on a given input. The alpha transition function simply becomes a function which maps a state and a predicate, rather than a symbol, to a set of states. However, the way in which the transition function is used must be changed in a subtle way. Since more than one predicate may be true, the automata must compute, for each state and for each true predicate, the set of states into which the automata transitions from that state. In theory, each predicate represents a non-deterministic branch and each state in the transition set of states is again a non-deterministic branch. However, as before, because the entire computational history can be summarized by the finite state automata's state, parallel branches that arrive at the same state may be collapsed into a single branch.

The algorithm for simulating an NFA with both alpha and $\epsilon$ transitions, which operates on predicate sequences is as follows: Beginning with a list of one or more start states, the automata goes to each state in the list and determines, for each alpha transition corresponding to a true predicate, the set of states into which the automata transitions. The union of these states becomes the new, intermediate state list. Next, the automata determines the set of states which may be reached by following $\epsilon$ transitions from any state in the intermediate set. This set, known as the $\epsilon$ closure is subsequently used as the new state list for the next input to the automata. As was the case with the non-augmented NFA, the automata is said to accept the input sequence up to and including the most recently read symbol whenever the state list comes to include an accept state. The automata is said to reject the input sequence whenever the state list is reduced to null or when the automata runs out of input.

Finally, the transformation of a regular pattern of predicates to these NFAs is straightforward. Figures 6-6, 6-7, and 6-8 show how to construct NFAs which implement the three basic operators: concatenation, union, and closure. By repeatedly applying these transformations to successively more complicated NFAs, a finite state automata may be generated to recognize any regular expression.

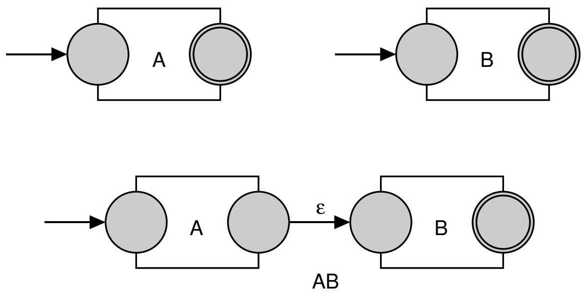



Figure 6-6: Concatenation of A and B

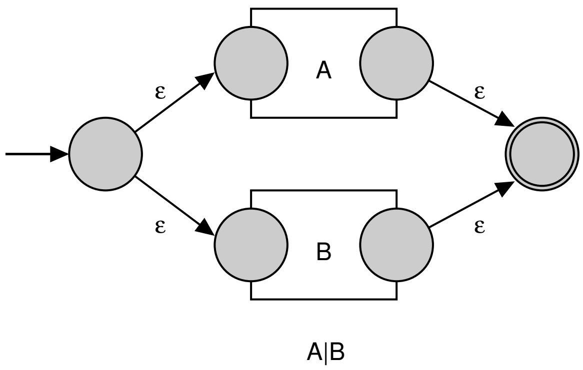



Figure 6-7: Transformation to $A|B$

## 6.2 Implementation

`VsVex` enables a programming style that permits the user to describe patterns of media

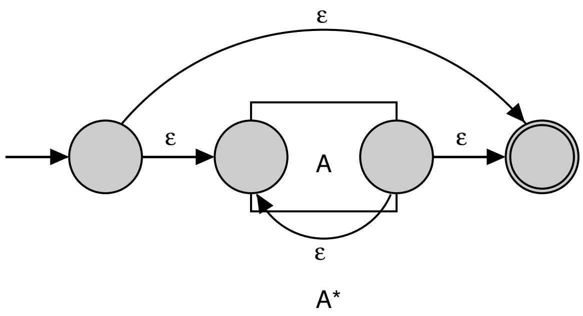



Figure 6-8: Transformation of A to A*

which are of interest and to define what actions should be taken when encountering such patterns. Conceptually, the parameters to `VsVex` [2] are intended to act as a specification of the form:

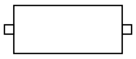
```
definitions
pattern: action
pattern: action
pattern: action
...
```

In practice, `VsVex` is a compound module which implements the pattern matching methodology developed in the previous section. The parameters to `VsVex` allow the programmer to explicitly specify the properties, predicates, and patterns used to generate events. `VsVex` also supports buffering which enables a program to locally store the input data which matches a pattern.

The following code fragment is an example of a simple `VsVex` definition. The module implemented by this code monitors the amount of change in a scene and calls the procedure `Go` when there is sufficient activity and `Stop` when the activity ends. The effect of employing the patterns "MMMM" and "WWWW" (rather than simply "M" and "W") is to add hysteresis to the module's behavior.

```
VsVex vs.vex \
  -properties {
    {VsDiffMotion -property motion -threshold 20}
  } \
  -predicates {
    {M "\[prop.f motion] > .10"}
    {W "\[prop.f motion] < .10"}
  } \
  -patterns {
    {RegMatch "MMMM" "Go"}
    {RegMatch "WWWW" "Stop"}
  }
```

[2] The name Vex is derived from the name "Video Lex" which was chosen because of Vex's resemblance to Lex [Aho *et al.*, 1986], which is used for parsing text.

**VsVex** 

`VsVex` is a compound module consisting of the three stages shown in figure 6-9, each implemented by a different sub-module. The first stage is itself a flow-graph of modules which generate the necessary properties. The second stage transforms these properties into boolean valued predicates. The third stage is a pattern matching automata, that generates events from these sequences of predicates.

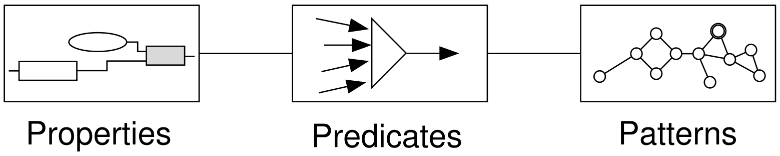



Figure 6-9: Three stages of a VsVex module

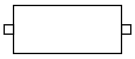
The property generation stage of the computation consists of a pipeline of filters which measure properties in the input stream and tag the payloads with those measurements. The pipeline is specified using the `properties` parameters and is implemented by passing the value of this parameter to a `VsCompose` module (see below) as that module's `pipe` parameter.

The second stage of `VsVex` transforms the properties generated by stage one into the predicates used by stage three. Predicates are implemented simply as boolean valued properties. Thus, the transformations may be performed by a `VsDerive` module. The `predicates` parameter of `VsVex` is a list of predicate descriptions. Each description is a pair consisting of the name of the predicate and the expression that "derives" the predicate. Therefore, each predicate description is processed by passing the name and expression to the derive method of the `VsDerive` module.

The third stage of the computation is the pattern matching automata. The function of the automata is to examine the input stream and to execute action procedures when patterns are matched. `VsAutomata` implements this functionality as a state transition network which analyzes the property lists of payloads in the input stream. The automata is configured according to the `patterns` parameter of `VsVex`. As will be shown below, the value of `patterns` is actually a list of procedure calls which generate the state transition network executed by the automata module.

The parameters for `VsVex` are summarized in the following table. In addition to the parameters shown, `VsVex` inherits the methods and parameters from `VsAutomata` which are described below.

| `Parameter` | `Value` | |
|---|---|---|
| `input` | *input_port* | Input port |
| `output` | *output_port* | Output port |
| `properties` | *list* | List of processing filters |
| `predicates` | *list* | List of predicate expressions |
| `patterns` | *list* | List of patterns |

### VsCompose 

`VsCompose`, which is used to implement the first stage of the `VsVex` module, chains together a sequence of one-input, one-output filters into an abstract module which itself has one `input` port and one `output` port.

The sequence of filters is specified as the `pipe` parameter. The value of the `pipe` parameter is a list of filter descriptions, each of which is treated in a manner similar to the `propMod` module in `VsClassProp`. That is, each filter description is itself a list in which the first element is the name of a filter and the rest is a list of parameter names and values.

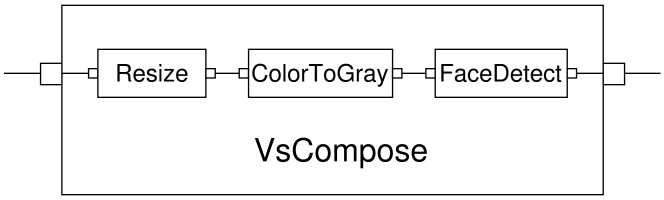


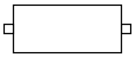

Figure 6-10: Example of a VsCompose pipeline

For example, the following code generates the pipeline shown in figure 6-10:

```
VsCompose vs.compose \
  -pipe {\
    {VsResize -scale .25} \
    {VsColorToGray} \
    {VsFaceDetect -inputRect {.2 .2 .8 .8} -property face}}
```

The parameters for `VsCompose` are summarized as follows:

| Parameter | Value | |
|---|---|---|
| `input` | *input_port* | Input port |
| `output` | *output_port* | Output port |
| `pipe` | *list* | List of processing filters |

### VsAutomata 

`VsAutomata` is a module that implements a state transition network[3]. Once configured, the module acts as a one-input, one-output filter which passes through a stream of payloads without altering the data. Upon each arrival of a payload the automata module examines the payload's properties and updates its internal computational state. In addition, whenever the automata reaches a node with an *action procedure* the automata executes that procedure.

[3] In describing the computation of the automata, there is potential confusion between the states which comprise the automata and the set of states which the automata is in. To avoid confusion, the states which comprise the automata will be referred to as nodes and the set of states which the automata is in will be referred to as the computational state of the automata.

## Representation

VsAutomata represents the state transition network as a list of nodes. Each node contains its own list of outward alpha transitions, $\epsilon$ transitions, and action procedures.

Initially, the node list for an automata module is null. Nodes are added by calling the module's makeState method. Each time makeState is called, a new node is allocated and assigned an identifier. The node identifier, which is simply an integer, is returned as the value of the makeState call.

An alpha transition is a set of links from a source node to one or more destination nodes. The links are followed when the source node is active and the new input payload matches the alpha transition's property name. An alpha transition is represented as a list where the first element is a property name and the rest of the elements are destination node identifiers. Alpha transitions may be positive or negative. Positive alpha transitions are followed when the value of the input property matching the transition's property name is true while negative transitions are followed when that input propery is false. Negative alpha transitions are specified by preceding the property symbol with a "$\wedge$".

The list of alpha transitions for a given node is set using the module's stateTransitions method. The stateTransitions method takes as arguments the identifier of the source node and a list of alpha transitions.

```
<module> stateTransitions 5 [list propA 5 11] [list ^propB 3]
```

An $\epsilon$ transition is a set of links from a source node to one or more destination nodes. When the automata reaches the source node, these links are followed immediately, and thus behave like forwarding pointers. Although they could be optimized out of a finished state transition network, they are a useful building block for composing compound networks out of subnetworks. The list of $\epsilon$ transitions for a given node is is set by the module's stateEtas method. The stateEtas method takes as arguments the identifier of the source node and a list of destination node identifiers.

```
<module> stateEtas 5 [list 1 4 9]
```

An action procedure is simply a Tcl callback which is run whenever an automata computation reaches the node which contains the action procedure. The callback procedure must accept two arguments, a starting time and an ending time which are the time stamps for the payload that started and ended the sequence of inputs resulting in the automata arriving at the current node. Action procedures are the observable output of the automata and are the mechanism by which the module signals a pattern matching event to the event processing core of the application[4]. The action procedure for a node is set by the module's stateAction method which takes as arguments the identifier of the node and the name of the callback procedure.

```
<module> stateAction 5 SaveClip
```

[4] VsAutomata does not explicitly represent final states. Rather, appropriately defined action procedures may be used to reset the automata when the computation reaches a "final" state

Finally, the start nodes for the automata must be specified. The `startStates` method selects a set of nodes which act as start nodes in the automata.

```
<module> startStates 1 3 7
```

### Computation

Once configured the automata maintains a set of "active" nodes. Whenever a new payload arrives, the automata considers each active node and checks the node's alpha transitions. An intermediate set of active nodes is computed as the union of the destination nodes for all the active alpha transitions that are labeled with a true valued predicate.

From the intermediate set of active nodes, the automata computes the set of nodes which may be reached from this set by following $\epsilon$ transitions. This new set, referred to as the $\epsilon$ closure of the intermediate set, is subsequently used as the initial set of active nodes for the next payload arrival. The algorithm for computing the $\epsilon$ closure is the same as that described in Hopcroft and Ullman [1979].

The active set is initialized from the automata's list of start nodes. If the module's `autoStart` parameter is set to 0, the initialization occurs when the module's `init` procedure is explicitly executed. Otherwise the set of active nodes is augmented to include the start nodes as each new payload arrives. In most cases, `autoStart` is set to 1 so that the new start nodes are continually merged into the current state of the automata. This allows an automata with a properly defined network to match a pattern starting with any payload without backtracking.

Finally, after computing the $\epsilon$ closure, the automata executes any action procedures from the new set of active nodes. The action procedures are the only observable output of the automata and therefore act as the "accept nodes" of the formal model. When executing an action procedure, the automata provides as arguments the starting and ending time for the matched sequence. The ending time is always the timestamp of the current payload. The starting time is the timestamp of the oldest payload which contributed to the current match. Computing the starting time requires a certain amount of bookkeeping. For this purpose, the starting time of each active node is maintained. That is, as the automata transitions from node to node, the starting time moves with the thread of computation. An efficient implementation was realized by understanding that when two threads transition to the same node, only the older starting time need be preserved. Thus, the starting times maintained by the automata reflect the longest sequence of inputs which could have resulted in the automata reaching the current node at the current time.

### Buffering

An important difference between the problem of recognizing media events from that of recognizing user events is that often, the media which triggered the event is of great interest.

`VsAutomata` provides the capability of buffering the input data. The buffer is necessary when, upon matching a pattern, the action procedure uses payloads prior to the last arrival. For example, the action procedure `SaveClip` saves the entire matching sequence of payloads to a file. Without the buffer, many of the payloads would have been deallocated by time the analyzer has determined the match, and executed the SaveClip procedure.

The value of the `buffer` parameter affects the `VsAutomata` module's behavior as follows: if `buffer` is 0, buffering is turned off and `VsAutomata` simply passes all incoming payloads to its output port as soon as it has updated its internal state. If `buffer` is 1, the automata module stores each incoming payload in a queue until the payload is either garbage collected or the module's `send` method is executed. The maximum length of this queue may be limited by setting the `bufferLimit` parameter. The arrival of any payload that would cause this limit to be exceeded causes the oldest payload in the queue to be discarded. A value of zero specifies no limit.

The `send` method, which is typically called from an action procedure, takes two arguments: a starting time and ending time. The `send` method causes `VsAutomata` to send all buffered payloads which have time stamps falling between the two time values.

An important component of the buffering facility is the garbage collector. The garbage collector discards payloads from the payload queue which are older than the time stamp of the oldest active node. The rationale for this policy is that no such payload could be part of a matching sequence.

Finally, a buffered automata can rapidly exhaust even large amounts of physical memory. Early implementations of `VsAutomata` relied on virtual memory to provide the memory required for large queues. Unfortunately, such implementations turned out to be impractical on multi-tasked workstation because Sieve's vociferous use of virtual memory caused other tasks (such as the window manager) to be swapped out of main memory. As a result, the workstation (both Suns and Alphas) would slow to the point of being unusable. The current implementation solves this problem by allowing `VsAutomata` to explicitly swap payloads to and from a file. Thus, when the payload queues exceed a specified limit, `VsAutomata` writes the payload data to a private swap file which it explicitly manages.

The parameters and methods for `VsAutomata` are summarized as follows:

| Parameter | Value | |
|---|---|---|
| `input` | *input_port* | Input port |
| `output` | *output_port* | Output port |
| `buffer` | {0 1} | Default 0 |
| `autoStart` | {0 1} | Default 1 |
| `payloadType` | *payload_type* | Default is VsVideoFrame |
| `makeState` | *etaTrans alphaTrans* | Allocate a new node |
| `stateTransitions` | *node alphaTrans ...* | Set the alpha transitions for a node |
| `stateEtas` | *node node node ...* | Set the eta transitions for a node |
| `stateAction` | *procedure* | Set the action procedure for a node |
| `startStates` | *node node ...* | List of start nodes for the automata |
| `init` | *n/a* | Initialize the active nodes |
| `clear` | *n/a* | Clear the active nodes |
| `send` | *time time* | Send payload buffer |
| `bufferLimit` | *int* | Default is no limit (0) |

As alluded to earlier, the value of the `VsVex` `patterns` parameter is a list of procedure calls which generate a state transition network within its `VsAutomata` module. More precisely, the `pattern` parameter is a list of *generators*. Each generator is itself a list in which the first element is the name of a generating procedure and the rest of which are arguments to that procedure. `VsVex` processes its pattern argument by passing the identity of its automata sub-module to the generating procedure along with the list of supplied arguments. The generating procedure then builds a state transition network in the automata module and returns the identity of the start node for the network. The returned value from each generating procedure is subsequently added to the automata module's list of start nodes.

For example, the following is a valid generating procedure which generates the network shown in figure 6-3 for any pair of predicates, $a$ and $b$, and accept procedure $acceptProc$:

```
proc GenExample {mod a b acceptProc} {
  set s1 [$mod makeState]
  set s2 [$mod makeState]
  set s3 [$mod makeState]
  $mod $s1 stateTransitions [list $a [list $s2]]
  $mod $s2 stateTransitions [list $a [list $s3]] [list $b [list $s2]]
  $mod $s2 stateAction $acceptProc
  return $s1
}
```

Thus, using this procedure, one could configure a `VsVex` module to recognize and save patterns of input which matched the sequences "XX", "XYX", "XYYX"... as well as the sequences "UU", "UVU", "UVVU"..., by specifying the `patterns` parameter as follows[5]:

```
<mod> patterns {{GenExample X Y SaveClip} {GenExample U V SaveClip}}
```

An extremely useful generating procedure is `RegMatch`. `RegMatch` takes as arguments the name of the automata module, a regular expression, and an action procedure. Using an algorithm for converting regular expressions to state transition networks modeled after the constructions presented earlier, `RegMatch` generates a state transition network that executes the supplied action procedure whenever a sequence matching the regular expression occurs. Thus, one could use `RegMatch` to generate a state transition network equivalent[6] to the one described above by setting the `patterns` parameter as follows:

```
<mod> patterns {{RegMatch "XY*X" SaveClip} {RegMatch "UV*U" SaveClip}}
```

As pointed out in the HeadHunter example of section 3.4, the `RegMatch` procedure implements two conventions that allow for a more compact syntax. First, `RegMatch` treats each character in the regular pattern as the name of a predicate. Second, lower case characters are reserved to represent the complement of uppercase properties. Thus, the character "`a`" in a regular expression matches inputs where the property "`A`" is false. Should either of these conventions prove to be too restrictive (for instance if one were to run out of single character property names), one could always implement a new generating procedure that uses a different syntax.

[5] The `patterns` method passes the name of the module (self) to the generating function as the first argument

[6] The network would be equivalent in its observable output though it would be implemented differently.

### Example: Gesture Monitor

One of the motivations for this thesis is to demonstrate the potential for using video processing to make computers aware of our physical world and in so doing, to change the way humans and computers interact. One of the most direct ways of doing this is to enable computers to recognize human gestures, such as the hand wave shown in figure 6-11. The following simple example serves both to demonstrate the functionality of `VsVex` and suggest how such capability might be exploited to make the workstation more responsive to the physical world.

Figure 6-11: Hand gesture

The code presented in figure 6-12 describes a simple gesture recognition module. The module is designed to recognize a hand wave. In particular, the module tracks the position of the user's hand and attempts to recognize when the user has waved at the camera. The gesture recognizer is implemented as a `VsVex` module.

The `properties` parameter configures the image processing pipeline to first reduce the resolution of the input to 160x120 and then matches it to a model template using a histogram matching module. The output of the pipeline is a stream of images tagged with the properties `hand`, `hand.left`, `hand.right`, `hand.top`, and `hand.bottom` which indicate estimated location of the hand and the quality of the match.

The `predicates` parameter specifies six predicates to be derived. The first of these, `Q`, is set to true if and only if the quality of the match exceeds a specified threshold. The predicate `L` is set to true if the hand moves to the left while the predicate `R` is set to true if the hand moves to the right. Both `L` and `R` are defined to be false `Q` is false for the current or previous input. In such cases, and in the case that the hand has remained relatively still, the predicate `S` is set to true. Finally, the predicates, “_” and “__” are derived for the sole purpose of setting state variables that record the previous position and match quality.

Finally, `patterns` parameter generates a state transition network capable of recognizing two patterns. The first of these, described by the regular expression "LLr*RRl*LL", is matched by input sequences where the hand moves first to the left, then the right, and finally back to the left. The first movement to the left is recognized by the sub-expression "LLr*". The sub-expression matches any sequence of left and still inputs which begins with at least two movements to the left. Likewise, "RRl*" matches any sequence of right and still inputs which begins with at least two movements to the right.

When an input matching the full expression is encountered, the `Match` procedure is executed. In testing, the `Match` procedure was defined to make the workstation say “Hello”. The second pattern, "SSSSSSSSSS", simply causes the pattern matcher to start over when it encounters 10 consecutive inputs in which the hand does not appear or is judged to be still. The reset is beneficial because the gesture pattern does not itself limit the number of still inputs which may be inserted into the sequence.

```
VsVex vs.vex
  -properties {
    {VsResize -scale .25}
    {VsTemplate -property hand -pathname hand-model.uv
      -matchMod {VsHistMatch
        -modelRect {0.59 0.25 0.68 0.58}
        -inputRect {0 0 1 1}
       }
     }
  }
  -predicate {
    {Q int {[prop.f hand] < 1.1}}
    {L int {[prop.i Q] && [prev Q] &&(([prev pos]-[prop.f hand.left])>.03)}}
    {R int {[prop.i Q] && [prev Q] &&(([prev pos]-[prop.f hand.left])<-.03)}}
    {S int {[prop.i L] || [prop.i R]}}
    {_ int {[setPrev Q [prop.i Q]]}}
    {__ int {[setPrev pos [prop.f hand.left]]}}
  }
  -patterns {
    {RegMatch "LLr*RRl*LL" "Match"}
    {RegMatch "SSSSSSSSSS" "Reset"}
  }
}
```

Figure 6-12: VsVex definition for recognizing a hand gesture

## 6.3 Summary

This chapter has shown how properties are used to generate symbolic events and so concludes the description of the analytical components of the Sieve toolkit. The next chapter describes a higher level of abstraction, *VsGrep*. VsGrep is an application builder that takes as input a high level specification and produces as output an interactive content analyzing application. The chapter also presents several applications built using this tool.

# Chapter 7

# Video Grep

The system described thus far is a toolkit for embedding video content analyzing capability within an event driven application.


Figure 7-1: The VsGrep application

This chapter describes a higher level abstraction, *VsGrep*. VsGrep (shown in figure 7-1) is a general purpose tool, built using the Sieve toolkit, that enables the user to direct the computer to identify sequences of images in video streams that match specific patterns and to take the appropriate actions. VsGrep may be thought of as an application builder, depicted in figure 7-2, which takes as input a high level specification and produces as output an interactive content analyzing application.

By providing a high level means of turning a concise specification into a complete, interactive application, VsGrep achieves one of this thesis's major goals: bringing the user closer to the video data and video processing. Still, VsGrep is not intended as a replacement for the lower level programming system. Indeed, the VsGrep specification language is itself an extension

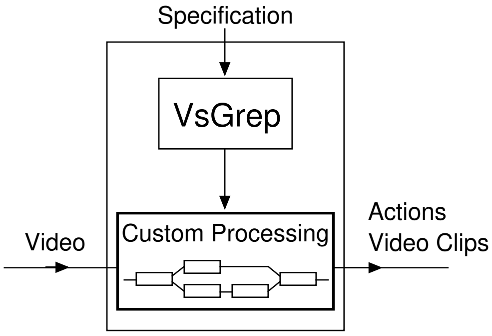



Figure 7-2: The VsGrep application builder

of the scripting language used within the underlying toolkit. Furthermore, though the VsGrep program has been designed for flexibility, it is limited by a fixed graphical interface and a data flow framework that can be inconvenient for implementing certain processing tasks. Thus, VsGrep aspires to be the powerful, though not universal, "grep"[1] utility of video processing tools, while embedded Sieve aspires to be the "regular expression library" of video-capable programming systems.

## 7.1 The VsGrep program

The VsGrep program scans a video stream for user specified patterns of input and produces as output a sequence of video "strings" or clips which match the specified patterns. Typically, though not always, these clips are stored in files and are displayed to the user in a dynamic video list as they are found.

The invocation of VsGrep is similar to that of grep. For example, assuming that the predicates "M", "m", and "." have been defined, and that the modules needed to measure the properties that make up the predicates have been specified, one may execute VsGrep using the command:

```
vsgrep "MMMM.*mmmm"
```

Such an invocation brings up the graphical shell shown in figure 7-1 and causes VsGrep to scan the default input device for sequences of images matching the regular pattern `"MMMM.*mmmm"`. As matching sequences are identified, they are saved to disk and added to the video list in the graphical shell. The user may select videos in the list for either review or removal.

The definitions for the predicates and the arrangement of the processing modules is provided by a specification file. In the example above, a default specification was used. More commonly, the user supplies a specification file to VsGrep using the command line option `-f spec-file`. For example, the following short but complete specification defines the "."

[1] grep is a popular unix utility which allows the user to scan text documents for lines matching a user specified pattern.

predicate to be true for all images, “M” to be true for images which exhibit a certain amount of motion, and “m” (by convention) to be the negation of “M”.

```
vex properties {
  {VsDiffMotion -threshold 40 -property motion}
}
vex predicates {
  {. int 1}
  {M int {[prop.f motion] > 0.001}}
}
vex buffer 1
vex filter 0
```

In the context of this specification, the pattern `"MMMM.*mmmm"` causes VsGrep to act as a motion detector which begins recording when four consecutive images depict motion and stops recording when the motion disappears for four consecutive frames.

The specification language bears a strong resemblance to the parameter syntax of a `VsVex` module. In fact, the VsGrep program is a thin shell which makes the functionality of a `VsVex` module available as an executable program with a graphical user interface (see figure 7-3). A VsGrep specification file is a Tcl script which augments the VsGrep application (itself written in Tcl) with custom procedures and definitions that implement the required properties, predicates, patterns and actions. An important procedure used by nearly all VsGrep specifications is the `vex` command. The `vex` command provides access to the program's underlying `VsVex` module. It simply passes its arguments directly to the `VsVex` module so that one may use the command to set parameters.

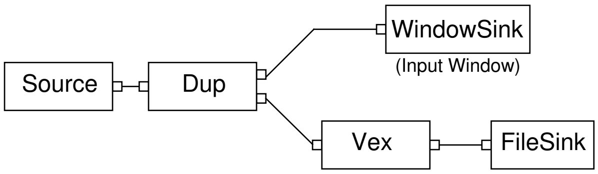



Figure 7-3: The VsGrep data flow

One may specify a pattern either on the command line or in the specification file. When specified on the command line, the pattern is interpreted as a regular expression which, when matched, executes the `SaveClip` procedure. Thus, an equivalent means of invoking VsGrep would be to add the statement:

```
vex patterns {{RegMatch "MMMM.*mmmm" SaveClip}}
```

to the specification file and to run VsGrep with no command line arguments. The `SaveClip` action procedure causes the payloads buffered in the `VsVex` module to be saved to a file and subsequently displayed in the video list. Capture and display are among the actions which VsGrep may take, though in general, any action which can be scripted may be taken in response to matching a video sequence.

The graphical interface to VsGrep (figure 7-4) provides an input window that displays the video input as it is processed. Below this window is a pair of buttons, “Control Panel” and “Program”, which bring up the standard control panel and visual programming panel that are built into most Sieve applications (see section 3.4 for descriptions of these panels). Below these buttons is a text box referred to as the message area. The message area provides textual feedback to the user. Any action procedure may print textual output to this area using the `Message` procedure.


Figure 7-4: The VsGrep graphical shell with a control panel and video player

To the right of the input window and message area is the output video list. The output video list presents a scrollable list of video clips to the user. Video clips that are captured using the `SaveClip` procedure appear automatically in this list. The user may select any file from the video list and either remove it from the list or launch a video player to play it by clicking on the "Remove" or "Play" buttons, respectively, below the list.

Finally, the VsGrep command line syntax is as follows:

```
vsgrep [-f <file>] [-showList 0|1] [-showInput 0|1] [pattern] [video-source]
```

The `-f` option was explained earlier. The `-showList` and `-showInput` options permit the user to change the program's graphical appearance. When set to 0, these options prevent VsGrep from displaying the output video list and the input video window respectively. Eliminating the input window moderately improves the applications throughput performance while eliminating the output video list simplifies the applications appearance in cases where the video list is not used. Additional options to VsGrep (not shown here) are passed directly to the video source. Thus, from a live source, one may select the source's encoding, resolution, port, etc... The available options depend on the particular source and are documented by Lindblad [1994].

The complete code for the program is provided in appendix B.

## 7.2 Gesture Recognizer

This first example of a VsGrep application demonstrates how the `VsVex` gesture code presented in section 6.2 may be implemented using a VsGrep specification. The specification (figure 7-6) defines a special purpose application that monitors a video stream for the appearance of a simple hand wave gesture (figure 7-5). The application responds to the gesture by bringing up a world wide web browser window that shows the day's news stories whenever the gesture is recognized.

Figure 7-5: The Gesture Recognizer

The beginning of the specification defines the action procedures `Match`, which is run whenever the wave gesture is recognized and `Reset`, which is run when the inactivity sequence is detected. The specification then sets the values of the `properties`, `predicates`, and `patterns` parameters for the `VsVex` module. These values are explained in section 6.2 which introduced the gesture example.

The gesture recognizer does a reasonable job of recognizing the hand gesture under a restricted set of conditions. First, the hand in the model image must closely resemble that of the user. This is not generally the case unless the model image was taken of the user's hand under similar imaging conditions. Under such circumstances, the gesture recognizer recognizes the hand wave roughly half the time and rarely mistakes a non-gesture for a gesture (see chapter 8).

```
# Action Procedures & Helper Functions
proc Match {args} {
  Message "...Getting News"
  catch {exec netscape -remote openURL(http://my.yahoo.com/)}
  vex clear
}
proc Reset {args} {vex clear}

vex properties {
  {VsResize -scale .25}
  {VsTemplate -property hand -pathname hand-model.uv
    -matchMod {VsHistMatch
      -modelRect {0.59 0.25 0.68 0.58}
      -inputRect {0 0 1 1}
    }
  }
}
# Predicates: Q = quality of match
#             L = hand has moved to the left
#             R = hand has moved to the right
#             S = hand is stationary
#             _,__ = dummy predicates, defined only to call the setPrev proc
vex predicates {
  {Q int {[prop.f hand] < 1.1}}
  {L int {[prop.i Q] && [prev Q] &&(([prev pos]-[prop.f hand.left])>.03)}}
  {R int {[prop.i Q] && [prev Q] &&(([prev pos]-[prop.f hand.left])<-.03)}}
  {S int {[prop.i L] || [prop.i R]}}
  {_ int {[setPrev Q [prop.i Q]]}}
  {__ int {[setPrev pos [prop.f hand.left]]}}
}
vex patterns {
  {RegMatch "LLr*RRl*LL" "Match"}
  {RegMatch "SSSSSSSSSS" "Reset"}
}
```

Figure 7-6: Gesture Recognizer specification

## 7.3 Room Monitor

The Room Monitor application (figure 7-7) demonstrates VsGrep's ability to summarize information by capturing video that is of interest to the user while discarding that which is not. In particular, this application summarizes activity in a room by capturing video whenever motion is detected. The application is intended to allow its user to use a camera to keep track of visitors that drop by while the user is away.

Figure 7-7: The Room Monitor

The simplest version of the room monitor application was presented in section 7.1 of this chapter. In that example, "M", "m", and "." were defined to represent motion activity, no motion activity, and "true" respectively. The pattern `"MMMM.*mmmm"` matched sequences of images which started with four consecutive frames of motion activity and ended with four consecutive frames of inactivity. The requirement of consecutive frames adds hysteresis to the program's behavior which prevents both transient motions from starting a sequence and short pauses in activity from ending a sequence.

One difficulty with the simple version of the room monitor is that sequences may grow to arbitrary length. A problem could, therefore, occur when a visitor came into the office to work for an extended period of time. In such an instance, the payload queue would likely overflow the storage requirements of the system. One solution is to limit the length of the payload queue. The `VsVex bufferLimit` parameter may be used for this purpose. This parameter causes the payload at the head of the buffer queue (the oldest payload) to be discarded whenever the limit is exceeded. Adding the statement:

```
vex bufferLimit 50
```

to the specification causes the application to record at most the last 50 payloads in a matching sequence.

One may prefer to capture the beginning, rather than the end, of a matching sequence. Unfortunately, it is not possible to implement such a feature in as simple a manner as that provided by the limit mechanism. One can not, for instance, simply employ a buffering system that discards payloads at the tail of the queue since such payloads can move forward in the queue as the match proceeds.

In the general case it is not possible to determine which payloads will comprise the start of a matching sequence until the match is completed. In many specific cases, however, it is possible to make such a determination. For instance, for the `"MMMM.*mmmm"` pattern, once the `"MMMM"` at the start of the pattern is matched to four payloads, it is guaranteed that the next matching sequence identified by the pattern matcher will start with these four payloads and run until the `"mmmm"` sequence is matched.

This observation was used to implement a modified room monitor which captures sequences of video that show the arrival of visitors. The modified specification, shown in figure 7-8, uses the same properties and predicates as the basic room monitor but differs in its use of three simultaneous patterns:

```
vex patterns {
  {RegMatch "MMMM.............................................." "Match"}
  {RegMatch "MMMM.*mmmm" "Match"}
  {RegMatch "mmmm" "Reset"}
}
```

Once four consecutive "M" frames arrive, one of the first two patterns will eventually match. The first pattern effectively limits the length of the matching video clip to 50 frames. The second terminates with the arrival of four consecutive "m" frames. In either case, a successful match causes the `Match` procedure, which captures and displays the video clip, to be executed. In addition, the procedure is implemented so that once it has been executed, further calls have no effect until the `Reset` procedure has been executed. The `Reset` procedure is executed whenever four consecutive "m" frames are encountered. The definitions for both procedures are included in the specification.

```
global occupied; set occupied 0

proc Match {args} {
  global occupied
  if {$occupied == 0} then {
    set occupied 1
    Message "Saved [apply SaveClip $args]"
  }
  vex clear
}
proc Reset {args} {
  global occupied
  if {$occupied == 1} then {
    Message "Reset"
    set occupied 0
  }
  vex clear
}
proc SaveClipCB {pathname} {
  catch {exec vsgrep -f vsgrep.monitor-icon "" $pathname -show 0}
  display $pathname
}
vex properties {
  {VsDraw -fillRect {.14375 .516667 .4406 .691667}}
  {VsDiffMotion -threshold 40 -property motion}
}
vex predicates {
  {. int 1}
  {M int {[prop.f motion] > 0.001}}
}
vex patterns {
  {RegMatch "MMMM................................................" "Match"}
  {RegMatch "MMMM.*mmmm" "Match"}
  {RegMatch "mmmm" "Reset"}
}
vex buffer 1
vex filter 0
```

Figure 7-8: Room monitor specification

The effectiveness of the room monitor application is largely dependent on the success or failure of the `VsDiffMotion` module to measure activity in the room. The module is highly effective at measuring activity in scenes which are not subject to periodic visual changes from objects such as fans, lights, and (most commonly) computer screens. Since the location of such disturbances is usually fixed, one may eliminate their effect by simply removing the appropriate region of points from the image. A `VsDraw` module, which draws constant colored shapes such as retangles and ovals on images, serves this purpose. The specification shown in figure 7-8 includes the removal of a rectangular region of points from the image stream in the property generating portion of the specification.

Finally, the modified specification includes an additional feature which improves the look and feel of the application. Normally the image depicting a video clip in the video list is simply the first image from the clip. However, this default is not always the best choice. For example, in the room monitor, the first image normally shows just the empty room. One may, however, override this default by explicitly providing an icon image. By convention, the icon image for a video clip is stored in a file with the same name as the video clip, plus the extension ".icon". Thus, before the room monitor displays a video clip in the output list, it computes an icon image by selecting the "best" image from the saved clip.

The room monitor implements this feature by redefining the `SaveClipCB` procedure. `SaveClipCB` is a callback procedure that is invoked after a video clip has been saved to a file. Normally, this procedure simply displays the video clip in the output list. The redefined procedure first computes the icon image by running a program that selects an image from the video clip. Appropriately, the program used to select the "best" image from the video clip is itself another VsGrep application.

The icon selection application attempts to find an image in the sequence in which the visitor is prominently displayed in the center of the image. The application uses a `VsStatMotion` filter (section 4.3.3) and `VsBinaryProp` filter (section 4.4.1) to generate a property vector describing the size and shape of the moving object. This property vector is matched against an ideal or "gestalt" vector. The image with the closest match is selected as the icon to represent the clip. The specification for the icon selection application is shown in figure 7-9.

```
# Compute the distance of the current properties to an ideal vector.
proc gestalt {} {
  if {[prop.f count] < $minCount} then { return -1 }
  set bestWidth .25;set bestHeight .575;set bestCount .13;set minCount .013

  set sqMid [sq [expr "(([prop.f box.right]-[prop.f box.left])/2) - .5"]]
  set sqWidth [sq [expr "[prop.f box.right]-[prop.f box.left] - $bestWidth"]]
  set sqHeight [sq [expr "[prop.f box.bottom]-[prop.f box.top]-$bestHeight"]]
  set sqCount [sq [expr "[prop.f count] - $bestCount"]]
  return [expr "$sqMid + $sqWidth + $sqHeight + $sqCount"]
}
proc sq {x} {return [expr "$x * $x"]}

# Returns true if current value is the smallest value passed thus far
proc minSoFarP {val} {
  global minSoFar
  if {$val < 0} then {return 0}
  if {![info exists minSoFar] || $val<$minSoFar} then {
    set minSoFar $val
    return 1
  }
  return 0
}
# After the image has been saved, clean up and exit the application
proc SaveClipCB {pathname} {
  global argv;
  exec mv $pathname [lindex [commandLineArguments $argv] 2].icon
  exit
}
# G is true iff current gestalt value is the smallest value seen thus far
vex predicates {
  {G int {[minSoFarP [gestalt]]}}
}
vex properties {
  {VsStatMotion -maskFill 1 -type count -threshold 40}
  {VsBinaryProps -boxFrac 5 -countProp count -boxProp box -centerProp center}
}
# Start over whenever a new minimum value is detected
vex patterns {
  {RegMatch "G" "setPrev icon"} {RegMatchNoClear "Gg*" ""}
}
vex buffer 1; vex filter 0
# Call SendExit at the end of the video sequence
proc SendExit {args} {apply SaveClip [prev icon]; vex start}
vex callback SendExit
```

Figure 7-9: Specification for icon selection

## 7.4 Computer Librarian

The Computer Librarian application (figure 7-10) allows one to use a workstation to monitor one's book collection. The librarian keeps track of who has borrowed what books by saving a video clip each time a visitor removes or returns a book from the shelf. The captured video clips enable the owner of the book collection to answer the age old question "Who borrowed my Java Reference Manual?" In doing so, this application may actually promote an environment in which workers are willing to share resources more freely.

Figure 7-10: The Computer Librarian

The librarian processes a stream of video from a strategically placed camera pointed at a bookshelf. The application makes use of the stationary filter, `VsStationary`, (see section 4.3.3) to remove the appearance of transitional objects, such as people, from the input stream. The output of the stationary filter is a stream of images in which only changes to the bookshelf appear. The filtered stream is passed through a motion detector, `VsDiffMotion`, which measures the amount of change between frames. When the measured change exceeds a certain threshold, a video clip is captured.

Because of the nature of the stationary filter, change in the scene does not become apparent in the output stream until a significant amount of time after the change has occurred. Thus, the captured video clip must be that which preceded the measured change by an amount greater than the delay built into the stationary filter. In addition, the captured video, like that of the room monitor application, should consist of the unfiltered input rather than the filtered output since the input shows the person borrowing the book.

Figure 7-11 shows the specification for the Computer Librarian. The parameters for the stationary filter configure the module such that the entire image is treated as a single block. This proved to be the most robust technique for filtering out transitional objects in cases in which the entire bookcase was frequently unobscured. However, a smaller block size is more appropriate for scenarios in which one portion or another of the bookcase is frequently blocked. The `bufferLimit` parameter is set so that the application maintains a buffer of the last 50 images. The `patterns` parameter causes this buffer to be saved to a file whenever change is detected.

```
vex properties {
  {VsStationary -blockSize {320 240} -moveBits 200 -constantCount 20
    -threshold 20 -speckThreshold 5 -speckWindow 5}
  {VsDiffMotion -threshold 20 -speckThreshold 5 -speckWindow 5
    -property changed }
}
vex predicates {
  {. int 1}
  {C int {[prop.f changed] > .01}
}
vex patterns {
  {RegMatch ".*C" "SaveClip"}
}
vex buffer 1
vex filter 0
vex bufferLimit 50
```

Figure 7-11: Computer Librarian specification

## 7.5 Whiteboard Recorder

The Whiteboard Recorder (figure 7-12) is another application which analyzes and digests video input in order to automatically summarize information for a human user. Using a strategically placed camera, this application monitors the user's whiteboard, automatically capturing a succinct set of images which summarize the content written on the board over a period of time. The recorder, therefore, permits the user to benefit from the storage and retrieval capabilities of computers while working in a familiar environment.

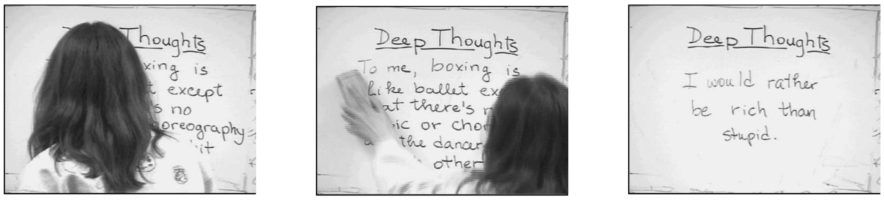



Figure 7-12: The Whiteboard Recorder

Similar to the Computer Librarian, this program uses a stationary filter to remove the appearance of temporary obstructions from the input video stream. It is assumed that the changes that appear in the filtered stream represent changes made to the writing on the board. These changes are subsequently analyzed to determine whether they represent additions, subtractions (erasure), or both to the writing on the board. The application then looks for "peaks" in the amount of writing content on the board. In other words, the program saves images which contain new writing content just before any writing is erased from the board.

The program makes use of a special purpose module, `VsSuperSub`, to determine whether changes are due to writing or erasing. `VsSuperSub` generates two properties, `super` and `sub`, which indicate, respectively, the fraction of pixels in the image that have darkened by a specified threshold or have lightened by a specified threshold, since the previous image. Since the application is specialized for recording information off a whiteboard, it assumes that darkened pixels represent new writing and lightened pixels represent new erasing.

The program accumulates the `super` and `sub` properties measured by the `VsSuperSub` module into two variables: `writeAccum` and `eraseAccum`. The predicates, `W` and `E` are subsequently defined as being true when these accumulated values surpass a given threshold. Finally, the program is configured to grab images when the pattern `WE` (which represents peaks in the writing content) is recognized. Figure 7-13 shows the complete specification.

```
global writeAccum; set writeAccum 1

proc GrabBoard {args} {
  set first [lindex $args 0]
  SaveClip $first $first
}
proc accum {var val} {
  global $var
  if {![info exists $var]} then {set $var 0}
  return [set $var [expr "[set $var]+$val"]]
}
proc resetAccum {} {
  if {[prop.i w] && [prop.i e]} then {
    global writeAccum; set writeAccum [prop.f super]
    global eraseAccum; set eraseAccum 0
  }
  return 1
}
vex filter 1
vex buffer 1

vex properties {
  {VsStationary -blockSize {320 240} -moveBits 200 -constantCount 20
    -threshold 30 -speckThreshold 5 -speckWindow 5}
  {VsSuperSub -threshold 30}
}
# Predicates: W = write
#             E = erase
vex predicates {
  {W int {[accum writeAccum [prop.f super]] > .001}}
  {E int {[accum eraseAccum [prop.f sub]] > .001}}
  {. int {[resetAccum]}}
}
vex patterns {
  {RegMatch "WE" "GrabBoard"}
}
```

Figure 7-13: Whiteboard Recorder specification

## 7.6 Television Agent

The final application, TV Agent (figure 7-14), demonstrates VsGrep's usefulness in building applications that monitor streams of broadcast video for clips which are of specialized interest to the user. The example specification for VsGrep has been adapted from the Sports Highlight Browser program presented in section 5.2. In particular, this specification causes VsGrep to capture video clips which depict Boston sporting events.


Figure 7-14: The Television Agent

The application makes use of several model matching filters: one which matches scoreboard images, one which matches scoreboard images in which Boston is the home team, one which matches scoreboard images in which Boston is the away team, and one which matches news anchor people. These filters are arranged in a classification network which tests first to see if each image is a scoreboard and if so, whether the image matches the Boston at home or the Boston away template and if not, whether the image depicts an anchor person.

Figure 7-15 shows the property generating portion for the television agent's specification. The code makes use of a special purpose module, `VsAnchor`, which is also defined in the specification (see figure 7-16). `VsAnchor` acts as a filter which passes payloads which do not depict scoreboard graphics to a `VsFaceDetect` filter. The `VsFaceDetect` filter looks for faces with a size and position indicative of an anchor person.

The pattern matcher for TV Agent looks for sequences of video which end with a series of Boston scoreboards and saves a video clip whenever such a pattern is encountered. The challenge lies in determining where the video clip begins. The application uses three heuristics to make the determination. The first choice is to begin the clip with a sequence of anchor shots which preceded the Boston scoreboard. This heuristic is designed to capture the cliche of an anchor person introducing a video clip, showing the video clip, and then summarizing the video clip with a scoreboard graphic. This choice is made if and only if an anchor spot has been detected more recently than the scoreboard which preceded the Boston scoreboard. If, on the other hand, the previous scoreboard has been detected more recently than the last anchor shot, the image just after the previous scoreboard is used as the starting point for the clip. In either event, the buffered clip is limited in duration to roughly 2 minutes. Thus, if neither an anchor shot or a scoreboard have been detected in the 2 minutes prior to the Boston scoreboard, the resulting saved clip will consist of the two minutes preceding the Boston scoreboard.

The pattern matcher groups sequences of anchor shots and scoreboard together by using regular expressions of the form "A.*aaaaaa" and "B.*bbbbbb". The effect of such expressions is that once a single image has matched the "A" or "B" predicate, the sequence will

```
# VsClassifyAnd <scoreboard>
#               VsClassifyOr <Boston away>
#                            <Boston home>
# VsAnchor
vex properties {
  {VsResize -scale .25}
  {VsClassifyFilter
    -classMod {VsClassifyAnd
      -modList {
        {VsClassProp -property score -expression {[prop.f p1]<100}
          -propMod {
            VsTemplate -pathname CNNFrames/scoreboard1.uv -property p1
            -fileFilter {VsResize -scale .25}
            -matchMod {VsPixelMatch
              -modelRect {.0781 .1041 .5468 .2082}
              -inputRect {.0625 .0833 .5625 .2291}
            }}
        }
        {VsClassifyOr
          -modList {
            {VsClassProp -property bostonA -expression {[prop.f p2]<100}
              -propMod
              {VsTemplate
                -fileFilter {VsResize -scale .25}
                -pathname CNNFrames/boston-philadelphia.uv -property p2
                -matchMod {VsPixelMatch
                  -modelRect {.0875 .3200 .5875 .4200}
                  -inputRect {.0775 .3100 .5975 .4300}
                }}}
            {VsClassProp -property bostonH -expression {[prop.f p3]<100}
              -propMod
              {VsTemplate
                -fileFilter {VsResize -scale .25}
                -pathname CNNFrames/boston-philadelphia.uv -property p3
                -matchMod {VsPixelMatch
                  -modelRect {.0875 .3200 .5875 .4200}
                  -inputRect {.0775 .4800 .5875 .6000}
                }}}
          }}}}}
  {VsAnchor}
}
```

Figure 7-15: The Television Agent property specification

continue until multiple, consecutive images fail to match. As a result, the expression groups sequences of images even in the presence of short drop-outs caused by noise and glitches.

Figure 7-17 shows the remainder of the television agent specification. As shown, the application simultaneously searches for three types of sequences: Boston scoreboards, non-Boston scoreboards, and anchor shots. When either of the latter two sequences are matched, the application acts by saving away the time-stamps of the matching sequence. When a Boston sequence is matched, the program calls the `Match` procedure which examines the saved time-stamps for the last anchor and scoreboard sequence to make a determination of where to start the saved video clip.

## 7.7 Summary

This chapter has described VsGrep and shown how it may be used to implement customized applications that analyze video content. The next chapter presents quantitative and quali-

```
VsTclClass VsAnchor -findSuperClass VsOpaque
VsAnchor proc create {m args} {
  set input [keyarg -input $args]
  set output [keyarg -output $args]

  $self nextProc $m

  VsClassify $m.class \
    -input "alias $m.input" \
    -property score
  VsFaceDetect $m.face \
    -input "bind $m.class.output0" \
    -scale {.1667 .3333} \
    -type fast_har \
    -inputRect {.25 .05 .75 .80} \
    -property anchor
  VsMerge $m.merge \
    -numInputPorts 2 \
    -output "alias $m.output" \
    -input0 "bind $m.class.output1" \
    -input1 "bind $m.face.output"

  if {$input != ""}  {apply $m.input $input}
  if {$output != ""} {apply $m.output $output}
  return $m
}
```

Figure 7-16: The VsAnchor filter

tative performance results for these programs.

```
proc Match {start end} {
  set aStart [lindex [prev anchor {0 0}] 0]
  set sStart [lindex [prev scoreboard {0 0}] 1]
  if {$aStart >= $sStart} then {
    # Start with Anchor
    SaveClip $aStart $end
  } else {
    # Start with previous scoreboard (up to 720 frame limit)
    SaveClip $sStart $end
  }
}
# B = boston scoreboard
# S = other scoreboard
# A = anchor person
vex predicates {
  {. int 1}
  {B int {[prop.i bostonA 0]||[prop.i bostonH 0]}}
  {S int {[prop.i score 0]&&![prop.i B 0]}}
  {A int {[prop.i anchor.top 0]}}
}
vex patterns {
  {RegMatch ".*B.*bbbbbb" Match}
  {RegMatch "A.*aaaaaa" "setPrev anchor"}
  {RegMatch "SSS.*ssssss" "setPrev scoreboard"}
}
vex buffer 1
vex bufferLimit 720
vex filter 0
```

Figure 7-17: The Television Agent predicates and patterns

# Chapter 8

# Performance

Sieve enables a user armed with a multimedia workstation to build interactive applications that analyze and interpret video input. Having developed the tools and methodology for implementing such applications, this chapter explores the computational requirements and qualitative performance of these programs. These considerations are particularly relevant to this work because they bear direct consequence on two aspects of the objective: interactivity and workstation viability.

Interactivity requires that the programs run in perceptual time. That is, in order to interact with a human user, these programs must achieve throughputs and latencies which are tolerable to the human participant. Furthermore, the nature of these applications, whereby the user specializes the processing to a customized task, is such that they be accessible to the end user. Thus, it is important that these programs are able to achieve the required performance on readily available hardware, such as computer workstations.

In short, this chapter demonstrates that the computational requirements necessary to achieve useful results are affordable. Furthermore, it will be shown that performance is dominated by the image processing and that by comparison, the cost of the powerful pattern recognition and abstraction mechanisms developed in this thesis are negligible.

These points are covered in three sections. Section 8.1 characterizes the computational requirements of several different applications. Section 8.2 describes the application's qualitative performance. Section 8.3 then documents the computational requirements for many of the image processing modules in the Sieve library.

## 8.1 Application Throughput

Table 8.1 gives the performance breakdown for each of the applications presented in chapter 7. Most of the applications were tested on video streams from live camera sources. The input for these applications consisted of either 8 bit gray scale or 24 bit color images[1]. Several applications were tested against both. The TV agent was tested on an 8 bit color video

[1] The input resolution of the images was set to 320x240 for each application, though the Gesture Recognizer and the TV Agent both reduced the internal resolution used for analysis to 160x120

file. The platform used for testing was a 166MHz Sun UltraSparc with a 512K external cache, a 16K/16K internal I/D cache running Solaris 2.5.

| Application | Input | Time/Rate msec (fr/sec) | I/O msec (%) | Properties msec (%) | Patterns msec (%) |
|---|---|---|---|---|---|
| Gesture | color/camera | 519ms (1.9) | 73ms (14%) | 419ms (81%) | 26.6ms (5.1%) |
| Rm Monitor | color/camera | 152ms (6.6) | 73ms (48%) | 79ms (52%) | 0.2ms (0.1%) |
| Rm Monitor | gray/camera | 80ms (12.5) | 64ms (80%) | 15ms (19%) | 0.7ms (0.9%) |
| Whiteboard | color/camera | 242ms (4.1) | 73ms (30%) | 168ms (69%) | 0.8ms (0.3%) |
| Whiteboard | gray/camera | 104ms (9.7) | 64ms (62%) | 38ms (37%) | 1.4ms (1.4%) |
| Librarian | color/camera | 244ms (4.1) | 73ms (30%) | 169ms (69%) | 2.4ms (1.0%) |
| Librarian | gray/camera | 107ms (9.3) | 64ms (60%) | 41ms (38%) | 2.3ms (2.2%) |
| TV Agent | color/file | 168ms (5.9) | 16ms (10%) | 145ms (86%) | 6.4ms (3.8%) |

Table 8.1: Throughput performance by application

- The first column of numbers represents the average total time spent processing each image in the input stream. This value was arrived at by measuring the total running time for the application to process a representative stream of 10000 images on an otherwise *lightly loaded* workstation[2]. The corresponding value in parenthesis is simply the total time inverted to represent frame rate.
- The second column of numbers represents the portion of this time spent on I/O. This value was arrived at by measuring the time spent to process a similar input stream (exactly the same for file inputs) without measuring any properties or deriving any patterns. Thus, this value represents the time taken to capture input from the digitizer (or read the input from a file) and display it in a window.
- The third column of numbers represents the time taken to perform the image processing necessary to convert the image sequence into a stream of properties. This value was arrived at by measuring the total time spent processing the input stream while the pattern matching mechanisms were disabled and subtracting out the estimated I/O overhead.
- Finally, the last column represents the time used to derive the predicates and patterns from the property measurements. This time was computed by measuring the difference between the total time taken by the application and the time taken by the application with pattern matching disabled.

The table reveals that the processing requirements are dominated by image processing (property generation) and I/O. Since the I/O processing time spent on each frame is roughly fixed (for a given platform), the percentage of time spent on I/O for a particular application is a direct reflection of frame rate. If application requirements for time resolution remain fixed, one may assume that increases in processor speed will result in more time being available to the image processing algorithms. In the near term, one may expect that applications

[2]The term lightly loaded is used in this chapter to refer to a workstation that is executing no other major processing tasks. Such a workstation is not, however, disconnected from the network and is not put in single user mode. Thus, performance on a lightly loaded workstation is meant to represent the best that may be achieved by a general purpose, multi-tasking machine operating under realistic conditions. The measurements made on the lightly loaded machine were obtained while using at least 95% of the CPU.

will take advantage of these technological improvements by increasing both frame rate and the sophistication of the processing. Eventually, however, the benefit to increased time resolution will diminish as frame rates approach human perceptual limits. Thus, while I/O remains an important component of computational cost, in the long term that cost should become less significant. The rest of the computational performance results (presented in section 8.3), therefore, focus on the performance of the processing modules.

## 8.2 Qualitative Performance

Throughput is just one measure of performance – and not the most important one. Much more important is how well the modules and the applications perform their task. Unfortunately, these are some of the most difficult measurements to make. Quality is subjective in nature and is difficult to quantify. Furthermore, the most meaningful measures of quality are those which characterize overall application performance – and thus, are dependent upon the scenarios under which the applications are tested.

The following does not attempt to exhaustively characterize the applications. While the number of repetitions are not sufficient to be conclusive, they are indicative and should provide the reader with a sense of how well these techniques work in practice.

The metrics used here to characterize applications, *recall* and *precision*, are those commonly used to describe the performance of retrieval systems. Recall represents the fraction of actual occurrences that were successfully identified. Precision represents the fraction of "hits" that represent actual occurrences.

$$recall = \frac{hits - false\_pos}{hits + false\_neg - false\_pos} \tag{8.1}$$

$$precision = \frac{hits - false\_pos}{hits} \tag{8.2}$$

The results of table 8.2 were obtained by running each application for a period of time under realistic operating conditions. The number of hits represents the number of times the application signaled the recognition of a pattern. The number of false positives represents the number of those hits that were judged to be in error. Likewise, the number of false negatives represents the number of times it was judged that the application failed to recognize a desired occurrence.

| Application | Time | Hits | False Pos | False Neg | Recall | Precision |
|---|---|---|---|---|---|---|
| Librarian | 540 min | 16 | 2 | 2 | 0.9 | 0.9 |
| Whiteboard | 120 min | 10 | 0 | 1 | 0.9 | 1.0 |
| Gesture | 180 min | 9 | 1 | 7 | 0.5 | 0.9 |
| Anchor (TV) | 30 min | 22 | 11 | 3 | 0.8 | 0.5 |

Table 8.2: Recall and Precision

The Computer Librarian functions as a reliable application. During testing, there were two occurrences of false positives and one occurrence of a false negative. One of the false positives was the result of accidently moving the camera during testing, the other occurred when the lights were turned on, causing a sudden lighting change. The false negative was the result of borrowing two different books in close succession. The program misclassified the two events as a single occurrence.

The Whiteboard Recorder is also quite reliable. The only error that was made during testing was a false negative that occurred because the view from the camera to the board never became unobstructed for a sufficient period of time. In general, this application functions well in scenarios in which the users make periodic trips to the board. The application does not record changes that occur while the user remains in front of the board.

The Gesture Recognizer performs adequately only in restricted circumstances. In particular, the gesture must be performed such that the gesturing hand closely resembles the model hand. Experimentation showed that the resemblance was sufficient only in circumstances in which the lighting of the scene and the distance from camera to the hand closely matched the conditions the imaging conditions that were used to record the model. The results of table 8.2 were obtained under such restrictions.

The TV Agent almost always correctly matches a scoreboard graphic unless the graphic itself has been changed by the sports broadcaster. The primary variation in the TV Agent's performance is its ability or inability to identify sports anchor people (see section 7.6). Thus, the results documented in the table indicate the performance of the anchor detector portion of the TV Agent. The anchor detector was tested on a 30 minute sports broadcast. Two of the three false negatives were due to back to back anchor shots that were misclassified as a single shot. The third false negative was the result of the anchor person not being placed in the center of the image[3]. All ten of the false positives were interviews of people. Five were separate shots from the same interview. The inability to distinguish between an interview shot and an anchor person shot is a basic limitation of the simple classification scheme. However, when joined by a human supervisor, the anchor detector is actually quite successful at reducing a video broadcast to a handful of candidate anchor shots for the user.

[3] The location of the head is one of the criteria used to determine whether a face shot is indeed an anchor shot. The inclusion of the off center head location would result in more false positives

## 8.3 Image Processing Throughput

Image processing throughput varies considerably by processing module, parameter settings, and input data. These variations allow the application designer to make performance trade-offs. This section documents the performance that was achieved under varying conditions by various processing modules in Sieve. It also considers performance on different platforms and the performance improvement that can be anticipated through improvements in the underlying hardware.

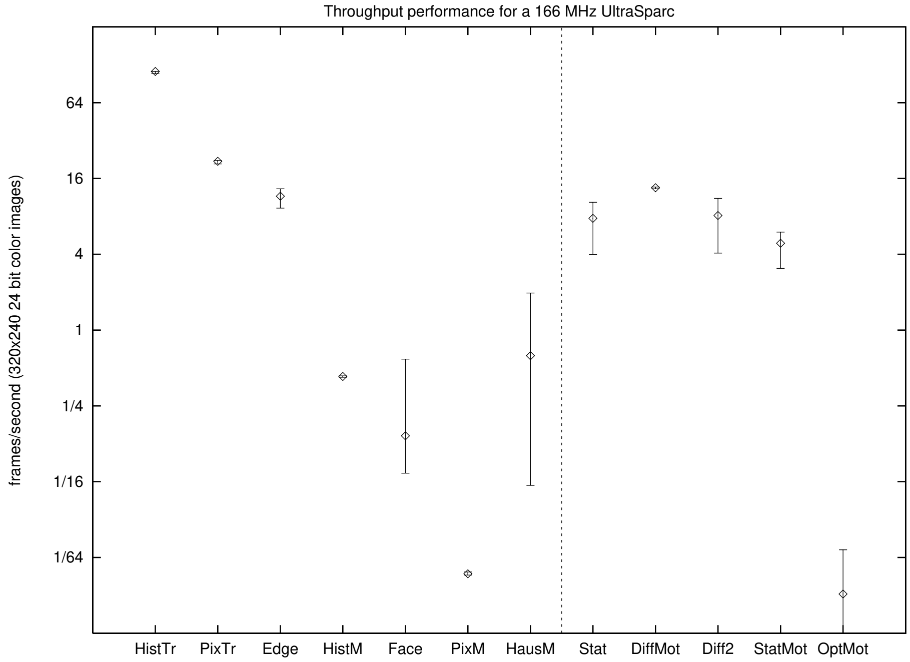



Figure 8-1: Throughput performance by filter

| Trial | Declaration |
|---|---|
| HistTr | `VsHistogram -colorBits {3 3 3}` |
| PixTr | `VsPixelTransform -colorTransform rgb2xyz -grayTransform window2gray` |
| Edge | `VsColorEdge -type canny -threshold 50` |
| HistM | `VsTemplate -matchMod {VsHistMatch -modelRect {.4 .4 .6 .6}}` |
| Face | `VsFaceDetect -type har` |
| HausM | `VsTemplate -matchMod {VsEdgeHausdorff -modelRect {.4 .4 .6 .6} -threshold {1.5 1.5} -fraction {.8 .8}}` |
| PixM | `VsTemplate -matchMod {VsPixelMatch -modelRect {.4 .4 .6 .6}}` |
| Stat | `VsStationary -blockSize {10 10} -moveBits 10 -constantCount 3 -threshold 20 -speckThreshold 5 -speckWindow 5` |
| DiffMot | `VsDiffMotion -threshold 40` |
| Diff2 | `VsDiffMotion -threshold 20 -speckThreshold 5 -speckWindow 5` |
| StatMot | `VsStatMotion -blockSize {10 10} -moveBits 10 -stationaryThreshold 20 -constantCount 3 -speckThreshold 5 -speckWindow 5 -threshold 40` |
| OptMot | `VsOptFlowMotion -threshold {0 -1}` |

Table 8.3: Filter declarations

Figure 8-1 provides a baseline for throughput performance of different processing filters given the filter declarations and input parameters listed in table 8.3. The ranges in value for a given module reflect the variation that was observed across ten different input sequences. Each input sequence consisted of 320x240, 24 bit color images. Thus, large ranges, such as those observed for the `VsFaceDetect` and `VsHausdorff` filters, are indicative of modules whose processing time depends on image content.

The vertical line between "HausM" and "Stat" separates static filters which treat each image input independently (to the left), from those, such as motion filters, which depend on inter-frame relationships (to the right). The two sets of filters were tested on different test sequences. The still image filters were tested on a sequence of ten selected images, chosen to be representative of different application scenarios. The motion filters were tested on ten sequences which were chosen to represent realistic inter-frame relationships. Appendix A shows the image stills and sequences that were used for testing.

The results were obtained using a 166 MHz Sun UltraSparc. Values were obtained by measuring the total elapsed time required to repeatedly pass a short sequence of payloads through the processing module. Care was taken to assure that the entire sequence resided in memory so that system I/O was not a factor.

The following subsections document how processing requirements vary with input characteristics (resolution and encoding type), parameter values, and platform architecture (sparc, alpha, pentium). The data points represent average performances measured against the representative inputs from appendix A.

### 8.3.1 Input Resolution

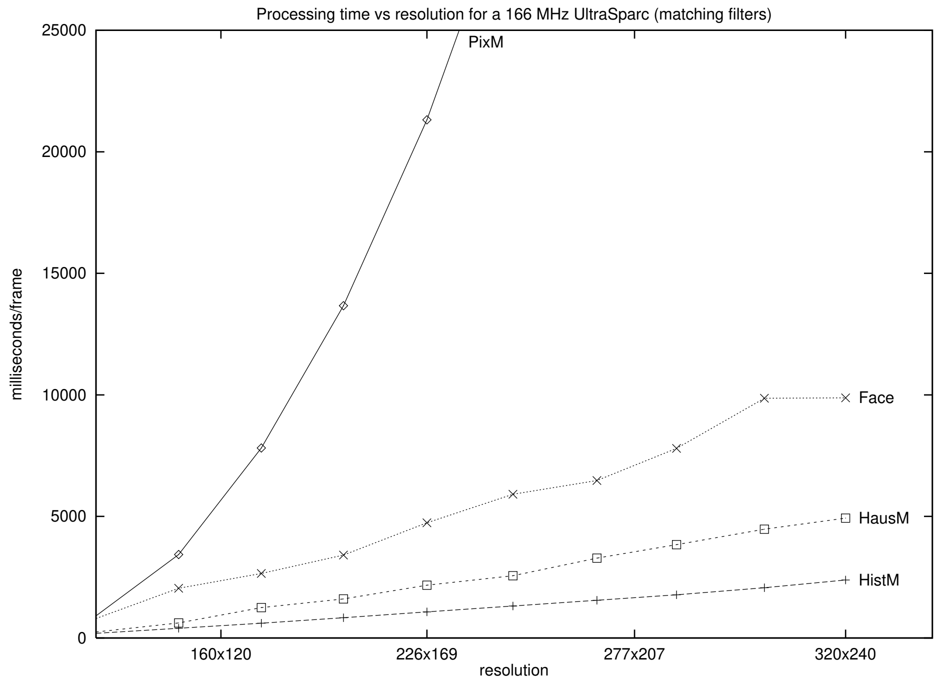



Figure 8-2: Processing time versus resolution for template matching

Figures 8-2, 8-3, and 8-4 document variation in processing time due to changes in resolution. The graphs have been drawn so that the number of pixels grows linearly along the x-axis. As a result, modules which perform a constant amount of work per pixel appear as straight lines. Such is the case for the motion and translation filters shown in figure 8-3.

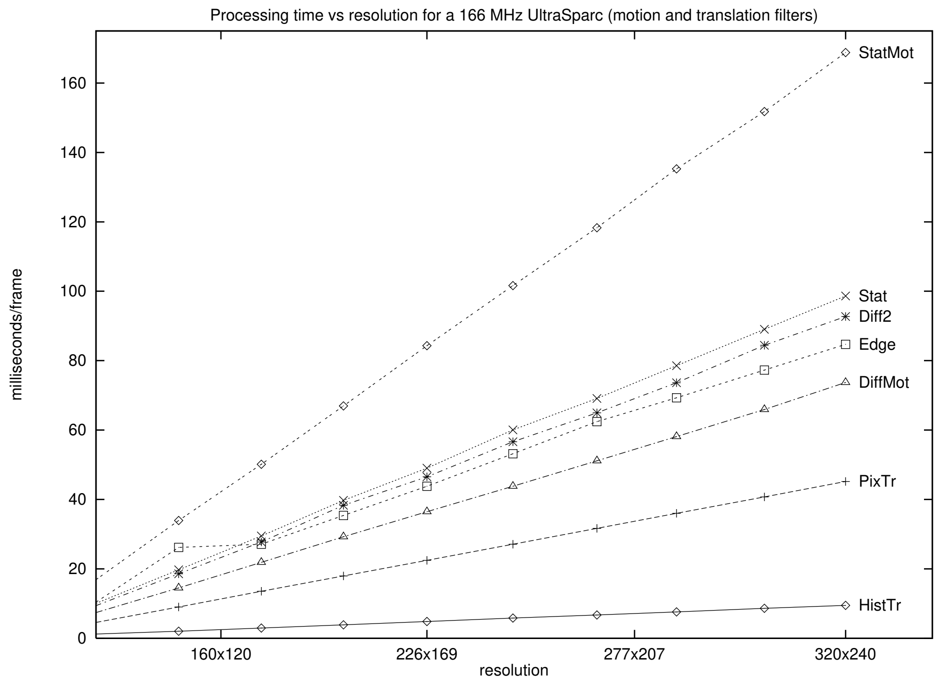



Figure 8-3: Processing time versus resolution for motion and translation filters

Other modules, for which both the number of pixels and the amount of work per pixel grows linearly with the number of pixels in the image, exhibit quadratic growth. This is the case for the `VsPixelMatch` module where the dimensions of the model region are specified as fixed percentages of the dimensions of the image, and the `VsOptFlowMotion` module where the search radius is, likewise, specified relative to the dimensions of the image. Such specifications are appropriate for most applications since the model size and the search radius are fundamentally related to real world objects and their relationship to the camera. Figure 8-4 illustrates the dramatic contrast between the performance curves for `VsPixelMatch` and `VsHistMatch`. The difference is remarkable because the processing time per pixel for `VsHistMatch` is actually a constant plus a factor which grows as the square root of the number of pixels. Thus, the order of growth for `VsHistMatch` is the number of pixels raised to the power of 1.5.

Figure 8-5 illustrates the effect of varying resolution on the axes from the original baseline graph (figure 8-1).

These results confirm that varying resolution is one of the most effective means of controlling processing requirements for a given application. Essentially, one may trade off time resolution (frames/second) for pixel resolution. This is hardly surprising, since resolution has a direct impact on the size of the input. Another means of controlling the size of the input is to change the "resolution" or depth (number of bits) in each pixel. While such a change does indeed affect the size of the input data, it does not, for most algorithms on

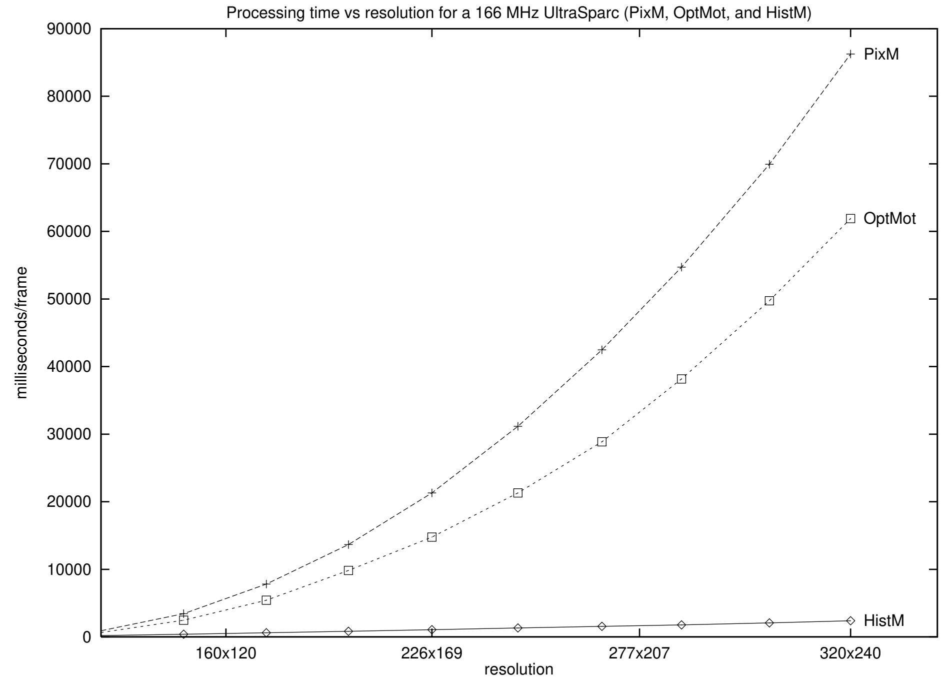



Figure 8-4: Processing time versus resolution for PixM and OptFlow (HistM shown for reference)

most processors, affect the number of operations which must be performed. Still, memory bandwidth is affected so that while bits per pixel does not normally have as great an impact on performance as number of pixels per image, there is still a performance impact.

It would be somewhat difficult to precisely characterize the potential effects of pixel depth on processing performance because fully exploiting such variations requires considerable experimentation with different pixel packing encodings, algorithms, and so on. Informal experiments indicated that while it was possible to obtain a performance benefit, it was common to obtain an unexpected performance loss. These losses appeared to be the result of variations in pixel memory alignments and compiler optimizations, though a thorough analysis was not performed. Tests were performed, however, comparing the differences in performance obtained for 24 bit color images versus 8 bit gray scale images, for several modules. Such comparisons include differences in the algorithms for the handling of color vectors and scalar intensities.

Table 8.3.1 presents the results of comparing the throughput performance for 24 bit color versus 8 bit gray scale. In most cases, the reduction in the size of the data and the algorithmic change from vector operations to scalar operations, results in the performance increasing by a factor of two to four. Two notable exceptions in the table are the case of `VsPixelMatch`, which achieves a considerably higher performance improvement; and `VsFaceDetect`, which is actually slower for gray scale than for color. The explanation for `VsPixelMatch` is simply that the method used to compare scalar pixels is considerably more efficient than the comparable method to compare vector pixels (absolute difference versus city-block difference – see section 4.1.1). The explanation for `VsFaceDetect` is that the filter uses color information, if it is available, to eliminate regions of pixels from consideration.

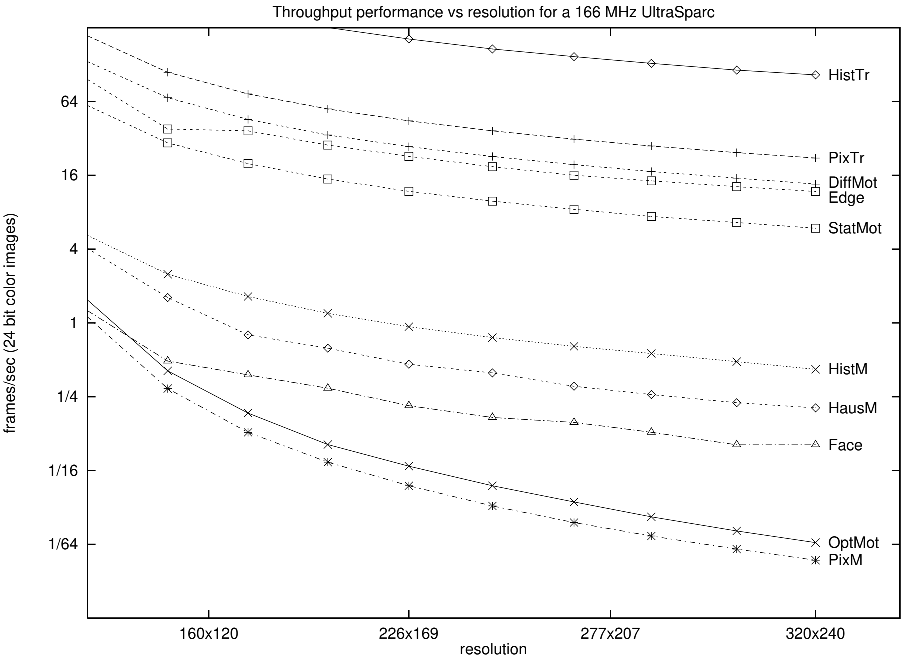



Figure 8-5: Throughput performance versus resolution

| Filter | Ratio | Gray (fr/sec) | Color (fr/sec) |
|---|---|---|---|
| HistTr | 2.32 | 244.499 | 105.374 |
| PixTr | 4.08 | 89.977 | 22.067 |
| HistM | 2.18 | 0.923 | 0.423 |
| Face | 0.95 | 0.096 | 0.102 |
| PixM | 9.65 | 0.110 | 0.011 |
| Stat | 2.58 | 25.806 | 10.010 |
| DiffMot | 4.46 | 60.350 | 13.526 |
| Diff2 | 2.70 | 29.146 | 10.776 |
| StatMot | 3.14 | 18.587 | 5.914 |

Table 8.4: Throughput for 8 bit gray scale versus 24 bit color

As a result, the color algorithm runs slightly faster than the gray scale version.

### 8.3.2 Processing Parameters

One expects a module's throughput performance to depend on its input parameters. Experimental results verify that this is indeed the case. Rather than attempt to exhaustively document the dependency of performance on parameters, this section presents two representative examples.

The first example contrasts the behavior of three region matching modules: `VsHistMatch`, `VsPixelMatch`, and `VsHausdorff`. Each of these modules scan an input region in the image for the set of points that best matches a model region[4]. In each case, the region of model points is specified in relative coordinates using the `modelRect` parameter. The throughput for the matching module is highly dependent on the size of the model rectangle relative to the region of the image to be searched. The shapes of the curves in figure 8-6 reflect the increasing cost per comparison as the model region is enlarged versus the decreasing number of locations that must be compared.

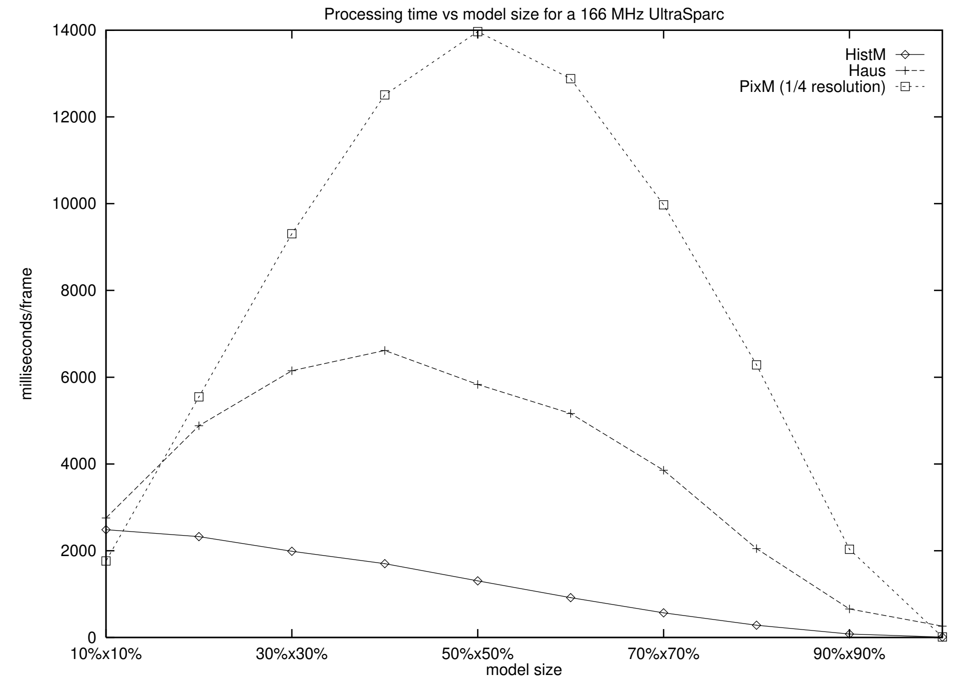



Figure 8-6: Processing time for matching modules as a function of model size. The resolution of the images used to obtain the results shown for VsHausdorff and VsHistMatch (320x240) was four times that used for VsPixelMatch (160x120).

The second example contrasts the varying computational costs associated with different algorithmic methods implemented by a single module. Figure 8-7 documents the variation in processing time due to different settings of the `vectorMethod` parameter of the `VsDiffMotion` filter. In particular, the relative performance of the `maxDiff`, `cityBlock`, `euclidean`, and `dotProduct` metrics (see section 4.3.1) are shown.

[4]In all cases, the input region was specified to cover the complete image. The effect of reducing the input area would be the same as increasing the model size and decreasing the resolution.

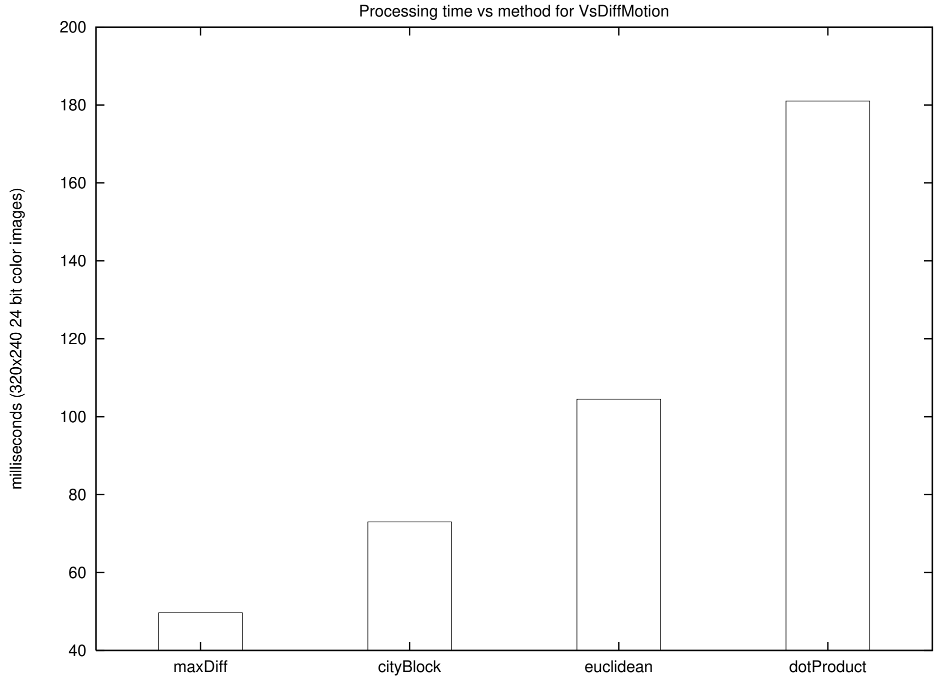



Figure 8-7: Processing time of different methods for VsDiffMotion filter

### 8.3.3 Processing Platform

Finally, the performance measurements presented thus far in this chapter were achieved using a 166 MHz UltraSparc. While this machine is a state of the workstation as of 1997, technological advances will soon render this machine and these numbers obsolete. One would like to be able to test performance on future workstations, but alas, this is not currently possible. Instead, tests were performed on three state of the art machines and their three predecessors. The results are summarized in figure 8-8.

The graph presents six performance numbers for each of twelve trials[5]. All six numbers represent throughput relative to that of the UltraSparc. The left most bars for each trial indicates the relative performance for two Intel x86 machines running Linux. The middle bars represents performance for two Sun Sparcs running Solaris 2, and the right most bars represents performance for two Dec Alphas running OSF/1. Table 8.5 details the specifications for each platform.

There are many interesting observations that one may make from figure 8-8. First, the relative performance of the different architectures was wildly different on the different trials. Each of the three architectures performed best on at least one trial and worst on at least one trial. This suggests that technological advances will have an uneven effect on video processing performance. In contrast to the relationship between different architectures, the relationship between different platforms within the same architectural family was far more consistent. In every experiment the newer platform outperformed its older counterpart,

[5] Exception: the HausM trial was performed only on the Sparc machines since this was the only architecture to which the Hausdorff library was ported

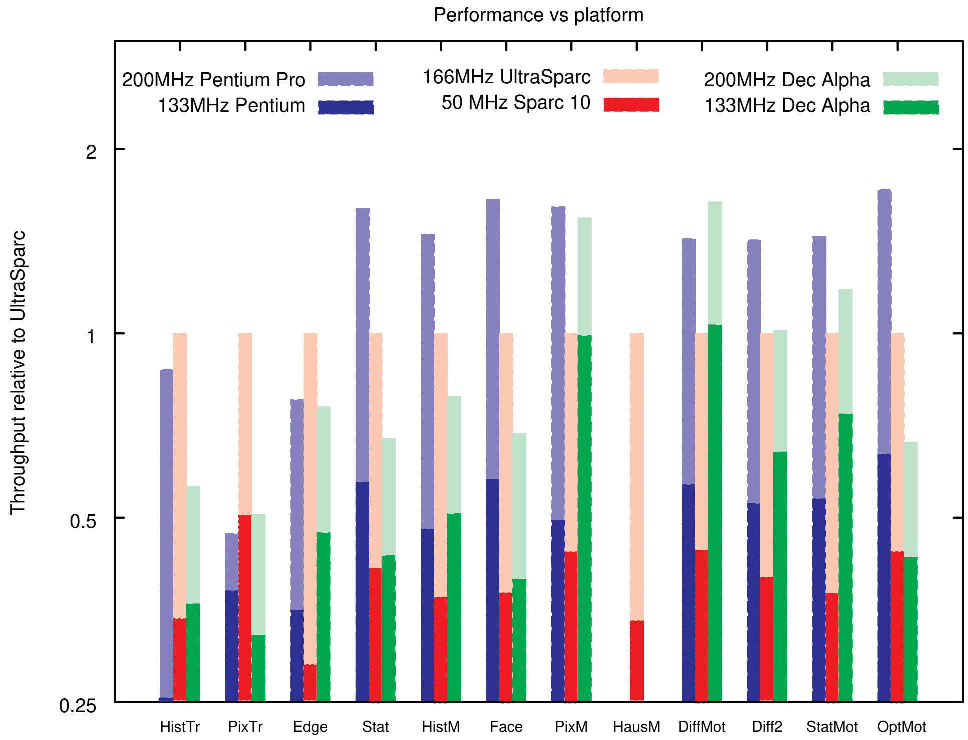



Figure 8-8: Average throughput performance by filter for different platforms measured relative to that of a 166 MHz UltraSparc

usually by a constant ratio.

| Platform | OS | Clock (MHz) ext/int | Cache ext+I/D |
|---|---|---|---|
| Intel Pentium Pro | Linux 2.0 | 66/200 | 256K+8/8 |
| Intel Pentium | Linux 2.0 | 66/133 | 256K+8/8 |
| Sun UltraSparc M170 | Solaris 2.5 | 83/167 | 512K+16/16 |
| Sun Sparc 10 | Solaris 2.5 | 40/50 | 1M+20/16 |
| Dec Alpha 3000/800 | OSF/1 3.2 | 40/200 | 2M+8/8 |
| Dec Alpha 3000/400 | OSF/1 3.2 | 27/133 | 512K+8/8 |

Table 8.5: Platform specifications

# Chapter 9

# Conclusions

This dissertation has shown that an interactive approach to the computerized processing and interpretation of visual information is both feasible and powerful. In support of this thesis, the following has been demonstrated:

- Symbolic representations partition the problem

  Symbolic representations separate the tractable elements of image processing and measurement from the unsolved problems of interpretation and meaning. The scriptable, event-driven Sieve programming environment demonstrates that given a framework through which unstructured media can be transformed to a symbolic representation, the assignment of function and meaning may be made on a customized basis.

- Interactivity leverages the user's intelligence

  An interactive system can leverage a human user's intelligence if it allows the user to customize the system and interpret the results. The Head Hunter application, presented in chapter 3, demonstrated both these elements by showing how a human supervisor could configure the program and evaluate the results, so that an unsophisticated program could be used to identify news anchor shots. The results presented in chapter 8 demonstrate that such a program can achieve a relatively high degree of accuracy.

- Applications that analyze media are powerful

  Applications that react to media content are fundamentally different than those which treat media as an opaque data type. The applications presented in chapter 7 demonstrate the potential for applications that analyze media information.

The following sections review additional contributions, insights, and future directions for this research.

## 9.1 Contributions

### The property → predicate → pattern → event paradigm

This work has developed a computational framework in which information evolves from raw sensory data to symbolic form. In doing so it defines the role that image processing and pattern matching should play in an interactive, content analyzing application.

### Sieve

Building a working system allowed the ideas put forth in this thesis to be tested, forced those ideas to be refined, and contributed to many of the insights of the study. The system also represents progress towards establishing an image analysis library. The complete source code is available[1] to anybody wishing to use Sieve for their own experiments, study the low level details for themselves, or extract code fragments for reuse in other systems.

### Visual Grep

The idea that vision needs a grep utility has been tossed around for some time. However, the complexity of matching unstructured visual input has prevented the development of such a tool. This dissertation puts forth VsGrep as a demonstration of how visual grep can be implemented.

### Novel Applications

The most important reason for developing the applications was to ground the research by forcing the system to solve real problems. However, a second benefit is the applications themselves. In spite of all the research in vision and multimedia systems, there do not appear to be working examples of desktop applications that perform functions like those performed by the Whiteboard Recorder, Computer Librarian, TV Agent, or Room Monitor.

## 9.2 Insights

In the course of this research, many insights have been gained. The following subsections introduce observations that may prove valuable to other researchers.

### 9.2.1 Computer vision libraries

It is surprisingly difficult to obtain libraries of sophisticated vision code. The prolific publishing of experiments and advanced techniques could easily lure one into believing that

[1] http://sds.lcs.mit.edu/sieve/

such libraries must be widely available. Unfortunately, this is not the case. As a result, it is very difficult to build upon other's work and nearly impossible to reproduce other's results.

Vision researchers must be encouraged to share more than just their findings. "Show me the code!" should become motto of the research community. Towards this goal the complete source code to Sieve (see contributions), with all its imperfections, has been made available.

### 9.2.2 Buffered NFAs

The non-deterministic finite state automata implemented by Sieve have been augmented with a buffer that enables the automata to quarantine payloads until it is determined whether they are part of a matching sequence. Since transitions are solely a function of new inputs and the automata's internal state, the class of sequences that may be matched is unchanged by this feature[2].

The ability to quarantine payloads proved to be one of the most beneficial functions of the automata module. In particular, the automatic determination of what frames must be saved and what frames may be released significantly simplified application design.

### 9.2.3 Regular expressions

Regular expressions are conveniently used to match patterns because they are compact and efficient. However, complex regular expressions such as those which attempt to match entire video clips (i.e. the beginning, middle, and end) can yield unexpected results. Experience showed that the expressions were best suited to finding (and labeling) the beginnings and endings of video clips. Procedural mechanisms were then used to group these labels into clips (see the TV News Agent in section 7.6 for an example).

### 9.2.4 Composition of data flow modules

A feature that is missing from many programming systems that arrange modules into flow graphs, is the ability to group modules into compound modules that may then be used as primitives. This capability has proven to be amongst the most important in Sieve.

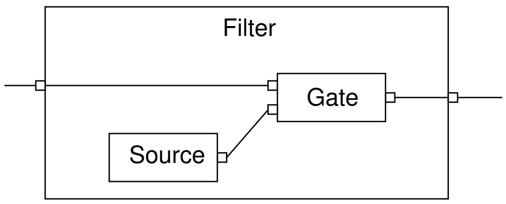



Figure 9-1: Composition

Figure 9-1 demonstrates a useful example, in which a two-input gate becomes a one-input

[2]However, it should be noted that conceptually, the buffer may store an infinite sequence of payloads so, strictly speaking, the augmented NFA violates the restriction of finite state.

filter by grouping the gate with a hidden source. The compound module can then be used wherever a one-input, one-output filter is required.

### 9.2.5 Flow graphs and blocking

Experience with flow graph applications has shown that they can be prone to blocking. In particular, blocking may arise when synchronously merging two or more data streams. Figure 9-2 illustrates a common example.

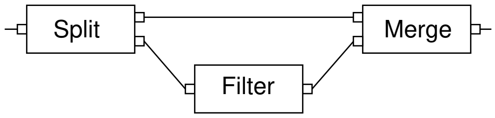



Figure 9-2: Example of potential deadlock

The problem occurs if the filter module does not produce exactly one output each time that it receives an input. For instance, if the filter produces fewer outputs than inputs, or merely delays its output until it receives multiple inputs, then one input port to the merge module is left waiting for the other. The splitter becomes blocked as it tries to forward copies of the payload to each of its outputs before accepting another input[3].

One solution is to implement filter modules so that they produce exactly one output each time an input is consumed. Often, this involves designing modules so that they initially produce blank or default outputs while they build up sufficient state to produce more meaningful results. Another solution is to provide buffering (perhaps in the splitter outputs or the merge inputs) to absorb discrepancies between flows. Still another alternative is to implement modules which merge streams in ways that do not require inputs to arrive in pairs. Unfortunately, none of these solutions are satisfactory for all cases.

### 9.2.6 Latency and data flow

The programming model for Sieve is one where the application programmer defines the input/output relationships of modules, arranges them in a flow-graph, and then allows the system to schedule the flow of data through the graph. The model permits multiple payloads to reside at different stages in the processing pipeline at the same time. Potentially, this ability could be used to schedule parallel execution of data processing. However, when running on a serial processor, the scheduling algorithm can induce unanticipated latency.

Figure 9-3 illustrates the potential problem. The processing time required by the three modules is labeled $T_1$, $T_2$, and $T_3$. Ideally, therefore, the latency, as measured by the time it takes for payload to travel from the source to the sink, is $T_1 + T_2 + T_3$. However, latency

[3]Actually, the program functions if the filter consumes exactly one payload without producing an output. In such a case, the splitter forwards copies to both the merge and filter inputs, and then receives the next payload. The merge input is blocked but the filter is not – so the splitter first passes a copy of the new input to the filter. The filter then produces an output which clears both of the input ports to the merge and the flow continues. The fact that deadlock doesn't occur until a second payload is "dropped" caused the deadlock problem to go unnoticed for some time.

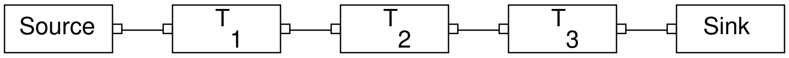



Figure 9-3: Source to sink latency

can grow by a factor of $n$, where n is the number of processing modules, if the processing is scheduled such that each module performs its function in turn, from the last module in the chain back to the front. Such an ordering may seem unnatural, but is indeed the ordering that occurs if the pipeline becomes full (perhaps because the sink temporarily becomes blocked) and module processing is centrally scheduled on a first come first serve basis.

## 9.3 Future Work

The following are suggestions for future research that build upon the work presented in this dissertation.

### 9.3.1 From data push to data pull

Computation in Sieve is modeled as the flow of media data through a network of processing modules that first measure the media's properties and then look for patterns in those properties. An interesting alternative would be to replace this model, in which data is essentially pushed forward from source to sink, with one in which data and processing are pulled, as needed, by the sink.

The pull model could be used to implement lazy evaluation of properties. In other words, a property value for a given payload would only be computed if it was required by a downstream module. Requirements would flow from the pattern matching automata (which depending on their state might require certain properties to make a transition, but not others) back to the property generating filters. For some patterns, the computational savings would be substantial. For example, suppose one was looking for occurrences of an anchor person appearing immediately after a commercial break; by searching for the appearance of a face just after a shot change. One need not perform the computationally expensive face detection algorithm for each image. The finite state automata pattern matcher could determine, from the state it was in, when the face property should be computed.

### 9.3.2 Example Based Programming and Reasoning

This work has focused on enabling a programmer to precisely specify the pattern of input about which he or she is interested. The assumption is that the user knows the features of interest and can determine the appropriate parameters. An alternative approach would be for the user teach the system using examples and have the system reason about the data to automatically derive parameters and possibly even the features.

It is suggested that the partitions between the property generating layer, the pattern recognition layer, and the event handling layer could also be used as the underpinnings for an example or reasoning based system. For example, such a system could attempt to classify inputs by searching for both the properties and patterns which distinguish positive and negative examples.

### 9.3.3 Integration with ActiveMovie

Sieve is a fully functioning, portable, stand-alone system that runs on many platforms. However, it lacks the documentation and general support of a commercial system. Sieve could be used by a wider group of users if it were integrated with an existing commercial environment.

ActiveMovie has emerged as a commercial standard that implements the basic programming model of the VuSystem. Augmenting ActiveMovie with the content analysis tools of Sieve appears, therefore, to be a worthwhile endeavor.

### 9.3.4 Vision components

A major focus of future research will be the development of more sophisticated event recognition through improvements in the underlying components which interpret imagery. Sieve could make immediate use of more advanced techniques for region matching, motion sensing, color constancy, and clustering. Specialized modules, such as the face detector, have already proven themselves to be of great utility.

### 9.3.5 HCI applications

Programs that interpret visual information can change the way humans and computers interact. Human beings have made remarkable strides towards adapting themselves to the world of the computer. For example, graphical user interfaces make it easier for people to work in the computer's environment by providing metaphors which aid the user in remembering and understanding the computer's rules.

To date, a fundamental barrier to computers adapting themselves to us has been their inability to "internalize" our environment, which is physical. People work on desktops, write on blackboards, and handle documents. This thesis has shown that computers can analyze and digest video information that is representative of the physical world. In so doing, they can become more aware of our environment and therefore more responsive to their human users.

# Appendix A

# Test Images

Chapter 8 demonstrated that the computational requirements of many of the image processing modules is highly dependent on the content of the input. For this reason, realistic images and image sequences were used to make the performance measurements.

Figure A-1 shows the ten still images used to test the image processing modules that treat images independently.

Figure A-2 shows an image for each sequence used to test filters which depend on interframe relationships. The last sequence represented in figure A-2 was composed from the ten independent stills. The nine other sequences consist of related images depicting contiguous motion. Figure A-3 shows one such sequence.


Figure A-1: Ten still images that were used for performance testing

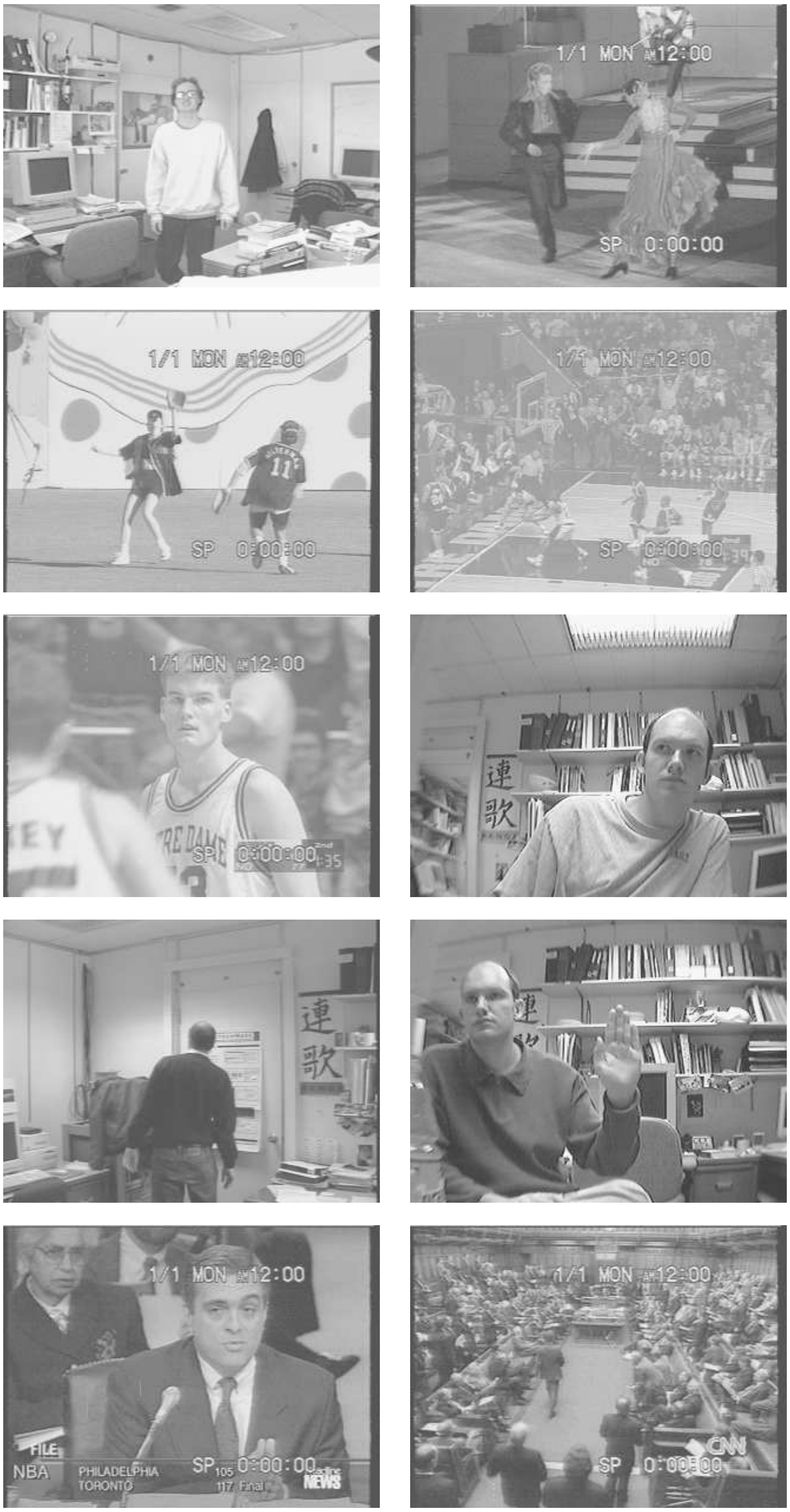



Figure A-2: Representative images from the ten motion sequences used for testing

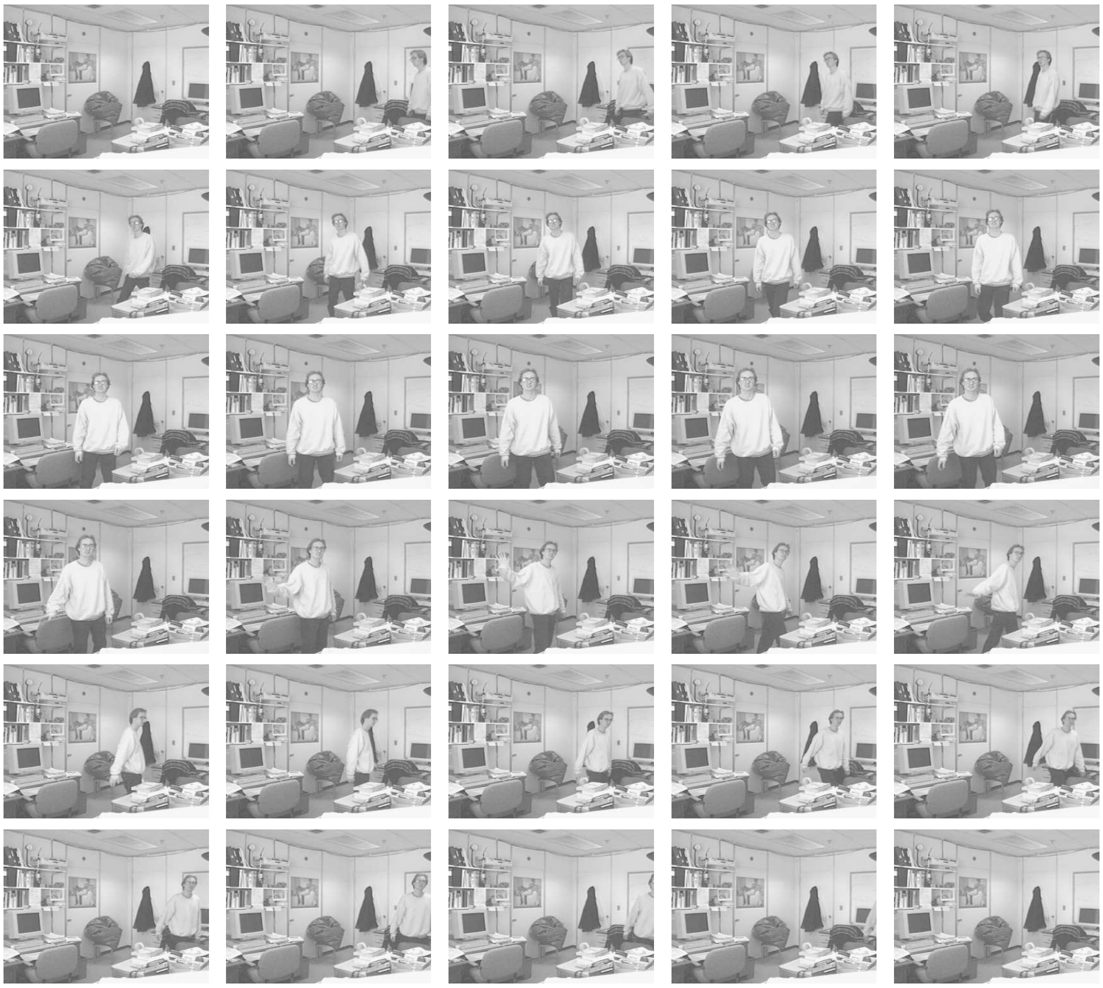


Figure A-3: A complete motion sequence

# Appendix B

# VsGrep Code

```
#
# Procedures to be called from vsgrep scripts
#

# Pre-fabricated action procedures

# Display text message in message box
proc Message {msg} {
  global GrepForm
  set w $GrepForm
  set cend [$w.message getInsertionPoint]
  $w.message replace $cend $cend "$msg\n"
}

# Save the current match as a video clip
proc SaveClip {args} {
  global GrepFlow outFileBase
  $GrepFlow.auto send $args
  $GrepFlow.auto clear
  set dir [file dir $outFileBase]
  set file [file tail $outFileBase]
  set path "$outFileBase.[getNextFileNumber $dir $file.].uv"
  $GrepFlow.filesink pathname $path
  $GrepFlow.filesink start
  return $path
}

# Default callback to be run after a video clip has been saved
# This procedure may be redefined by vsgrep scripts
proc SaveClipCB {pathname} {
  global EXIT
  if {[info exists EXIT] && $EXIT == 1} then {
    exit
  } else {
    display $pathname
  }
}

# Used by scripts to configure the VsVex module

# Old definition
#proc vex {args} {
#  global GrepFlow
#  apply $GrepFlow.auto $args
#}
# New version used for performance testing
#   adds the ability to disable property, predicate, and pattern processing
```

```
proc vex {args} {
  global GrepFlow TestArgs

  set cmd [lindex $args 0]
  set test [keyarg -test $TestArgs]
  set buffer [keyarg -buffer $TestArgs]

  if {$cmd == "buffer"} then {
    if {$test!="none" && $test!="prop"} then {
      if {$buffer!={}} then {
        $GrepFlow.auto buffer $buffer
      } else {
        apply $GrepFlow.auto $args
      }
    }
  } elseif {$cmd== "properties"} then {
    if {$test!="none"} then {
      apply $GrepFlow.auto $args
    }
  } elseif {$cmd=="predicates" || $cmd=="patterns"} then {
    if {$test!="none" && $test!="prop"} then {
      apply $GrepFlow.auto $args
    }
  } else {
    apply $GrepFlow.auto $args
  }
}

# Auxiliary procedures useful for predicates

proc setPrev {var args} {
  global PrevVars
  set PrevVars($var) $args
}
proc prev {var args} {
  global PrevVars
  set default [lindex $args 0]
  if [info exists PrevVars($var)] then {
    return $PrevVars($var)
  } else {
    return $default
  }
}

#
# Set up and configure the flow-graph
#

proc VsGrepFlow {w m args} {
  global GrepFlow; set GrepFlow $m
  set show [keyarg -show $args 1]
  set showList [keyarg -showList $args $show]
  set showInput [keyarg -showInput $args $show]

  set info [keyarg -info $args 0]
  set exit [keyarg -exit $args 0]

  if {$info == 0} then {set info {}}
  if {$info == 1} then {set info PutsPropList}

  VsEntity $m
  $m set showList $showList

  # Create video source
  apply VsVideoSource $m.source \
    -fileSourceEncoding 3 \
    -parentWidget $w \
    -captions [false] \
```

```
    $args
  set nextInput "bind $m.source.output"

  # If -show is not 0 then display input in a window
  if $showInput then {
    VsDup $m.dup \
      -numOutputPorts 2 \
      -input $nextInput
    set nextInput "bind $m.dup.output0"

    # Run as fast as possible
    VsReTime $m.reTime \
      -speed 10000 \
      -input $nextInput
    set nextInput "bind $m.reTime.output"

    # Exit the program at the end of a video file
    $m proc exitCallback {args} {
      set sinkFinish [keyarg -sinkFinish $args 0]
      if $sinkFinish then {exit}
    }

    # Create the window sink
    VsWindowSink $m.sink \
      -widget $w.input.screen \
      -input $nextInput
    set nextInput "bind $m.dup.output1"
    $m.sink callback "$m exitCallback"
  }

  # If -exit is not 0 then this shuts down vsgrep after $exit frames
  if $exit then {
    VsExit $m.exit \
      -input $nextInput \
      -frames $exit
    set nextInput "bind $m.exit.output"
  }

  # Create the VsVex module
  VsVex $m.auto \
    -info $info \
    -input $nextInput
  set nextInput "bind $m.auto.output"

  # Display saved video clips in the VideoList
  $m proc expose {pathname} {
    if $showList then {
      [[file root $self].inEntries addEntry $pathname] expose
    }
  }
  # Reset the file sink at the end of the video clip
  $m proc sinkCallback {args} {
    if [keyarg -sinkFinish $args 0] then {
      set pathname [$self.filesink pathname]
      $self.filesink pathname ""
      SaveClipCB $pathname
    }
  }
  # File sink for saving video clips
  VsFileSink $m.filesink \
    -index {} \
    -callback "$m sinkCallback" \
    -input $nextInput

  return $m
}

#
```

```
# Set up the user interface
#

proc VsGrepForm {w m args} {
  global GrepForm; set GrepForm $w
  set margs [keyarg -margs $args]
  set args [keyargs {-margs} $args exclude]

  set show [keyarg -show $margs 1]
  set showList [keyarg -showList $margs $show]
  set showInput [keyarg -showInput $margs $show]

  VsEntity $m
  $m set w $w

  apply Form $w \
    $args
  Form $w.input \
    -resizable true
  if $showInput then {
    VsScreen $w.input.screen \
      -resizable true
  } else {
    Box $w.input.screen \
      -width [expr "[vsDefault -width]/2"] \
      -borderWidth 0
  }
  Command $w.input.controlPanel \
    -label "Control Panel" \
    -callback "VsPanelShell $w.input.controlPanel.shell -obj $m" \
    -fromVert $w.input.screen
  Command $w.input.visualPanel \
    -label "Program" \
    -callback "VsVisualShell $w.input.visualPanel.shell -obj $m.flow" \
    -fromHoriz $w.input.controlPanel \
    -fromVert $w.input.screen

  if {$showList && !$showInput} then {set height 354} else {set height 114}
  AsciiText $w.message \
    -width [expr "[$w.input.screen getValues -width]+10"] \
    -height $height \
    -displayCaret false \
    -scrollVertical whenNeeded \
    -wrap word \
    -resizable true \
    -fromVert $w.input
  [$w.message getSource] setValues -string "" -editType append

  if $showList {
    apply VsGrepVideoList $w $m $args
  }
  return [apply VsGrepFlow $w $m.flow $margs]
}

# Create the output VideoList
proc VsGrepVideoList {w m args} {

  $m proc playSelection {args} {
    set entry [$self.inEntries current]
    if {[info commands $entry] != {}} {
      catch {exec vsplay [$entry pathname] &}
    }
  }
  $m proc removeSelection {args} {
    set entry [$self.inEntries current]
    if {[info commands $entry] != {}} {
      exec rm [$entry pathname]
```

```
      $self.inEntries deleteEntry $entry
    }
  }
  VsVideoList $w.inEntries $m.inEntries \
    -height 370 \
    -fromHoriz $w.input
  Command $w.play \
    -label "Play" \
    -callback "$m playSelection" \
    -fromVert $w.inEntries \
    -fromHoriz $w.input
  Command $w.sub \
    -label "Remove" \
    -callback "$m removeSelection" \
    -fromHoriz $w.play \
    -fromVert $w.inEntries\
    -fromHoriz $w.play
}

# Display file in the video list
proc display {pathname} {vs.grep.flow expose $pathname}

#
# Main routine and misc procedures
#

# Evaluate the the vsgrep script
proc configure {pattern spec} {
  if {[file exists $spec]} then {
    source $spec
  }
  if {$pattern != {}} then {
    vex patterns [list [list RegMatch $pattern SaveClip]]
  }
}

# Run the vex callback before actually quitting
proc quit {} {
  if {[vex callback] == {}} then {
    catch {vs destroy}; exit
  } else {
    [vex callback]
  }
}

# Main invocation
proc main {} {
  global argv name class errorInfo outFileBase TestArgs

  set name [lindex $argv 0]
  set class VsGrep
  set sources [commandLineArguments $argv]
  set args [commandLineOptions $argv]

  set margs [keyargs {-depth -visual} $args exclude]
  set shArgs [keyargs {-depth -visual} $args]

  set pattern [lindex $sources 1]

  set videoSource [lindex $sources 2]
  if {$videoSource != {}} {
    set margs [concat -videoSource $videoSource $margs]
  }
  set outFileBase [keyarg -out $args grepout]
  set TestArgs [keyargs {-test -buffer} $args]
  set spec [keyarg -f $args]

  xt appInitialize app_context $class argv {}
```

```
  vs appInitialize app_context vs

  apply VsShell $name.top \
    -title VsGrep \
    -cmd VsGrepForm \
    -realize "configure [list $pattern] [list $spec]; vs start" \
    -dismiss "quit" \
    -args [concat vs.grep [list -margs $margs]] \
    -allowShellResize true \
    $shArgs

  while {[catch {app_context mainLoop} msg]} {
    VsErrorShell $name.err -summary $msg -detail $errorInfo
  }
}

main
```

# Bibliography


[Adam *et al.*, 1993] J. F. Adam, H. H. Houh, and D. L. Tennenhouse. Experience with the vunet: A network architecture for a distributed multimedia system. In *Proceedings of the IEEE Conference on Local Computer Networks*, pages 70–76, Minneapolis MN, September 1993.

[Aho *et al.*, 1986] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, Reading, MA, 1986.

[Apple, 1992] Apple. *Inside MacIntosh*. Addison-Wesley, Reading, MA, 1992.

[Asente and Swick, 1990] Paul Asente and Ralph Swick. *X Window System Toolkit*. Digital Press, Bedford, MA, 1990.

[Bach *et al.*, 1996] Jeffrey R. Bach, Charles Fuller, Amarnath Gupta, Arun Hampapur, Bradley Horowitz, Rich Humphrey, Ramesh Jain, and Chiao-Fe Shu. Virage image search engine: an open framework for image management. In *Proc. SPIE*, March 1996.

[Bajcsy, 1973] R. Bajcsy. Computer identification of visual surfaces. *Computer Graphics and Image Processing*, 2(2):118–130, October 1973.

[Ballard and Brown, 1982] Dana Ballard and Christopher Brown. *Computer Vision*. Prentice-Hall, Englewood Cliffs, NJ, 1982.

[Beymer, 1996] David Beymer. *Pose-Invariant Face Recognition Using Real and Virtual Views*. PhD thesis, MIT, 1996. AI-TR 1574.

[Bregler, 1997] Christoph Bregler. Learning and recognizing human dynamics in video sequences. In *IEEE Conf. on Computer Vision and Pattern Recognition*, 1997.

[Canny, 1983] John Canny. Finding edges and lines in images. AI Technical Report 720, MIT, June 1983.

[Carson *et al.*, 1996] Chad Carson, Serge Belongie, Hayit Greenspan, and Jitendra Malik. Region-based image querying. Technical Report 941, UC Berkley, 1996.

[Cascia and Ardizzone, 1996] M. La Cascia and E. Ardizzone. Jacob: Just a content-based query system for video databases. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, 1996.

[Coombs *et al.*, 1992] David Coombs, Ian Horswill, and Peter Von Kaenel. Disparity filtering: proximity detection and segmentation. In *Proc. SPIE Vol. 1825*, pages 195–206. The International Society for Optical Engineering, November 1992.

[Duda and Hart, 1973] Richard Duda and Peter Hart. *Pattern classification and scene analysis*. John Wiley, New York, 1973.

[Eakins *et al.*, 1996] J. P. Eakins, K. Shields, and J. M. Boardman. Artisan - a shape retrieval system based on boundary family indexing. In *Proc. SPIE*, March 1996.

[Flickner *et al.*, 1995] M. Flickner, H. Sawhney, W. Niblack, J. Ashley, Q. Huang, B. Dom, M. Gorkani, J. Hafner, D. Lee, D. Petkovic, D. Steele, and P. Yanker. Query by image and video content: The QBIC system. *IEEE Computer*, 28(9):23–32, 1995.

[Freeman and Brainard, 1995] W. T. Freeman and D. H. Brainard. Bayesian decision theory, the maximum local mass estimate, and color constancy. In *Proceedings IEEE International Conference on Computer Vision*, pages 210–217, 1995.

[Funt and Finlayson, 1995] Brian V. Funt and Graham D. Finlayson. Color constant color indexing. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17(5):522–529, 1995.

[Grimson, 1990] W. Eric L. Grimson. *Object Recognition by Computer*. MIT Press, Cambridge, MA, 1990.

[Heeger *et al.*, 1992] David Heeger, Eero Simoncelli, and EJ Chichilnisky. Obvius: Object-based vision and image understanding system, version 2.2. Vismod 195, MIT Media Lab, 1992.

[Hopcroft and Ullman, 1979] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Publishing Company, Reading, MA, 1979.

[Horn and Schunck, 1981] Berthold K. P. Horn and Brian G. Schunck. Determining optical flow. *Artificial Intelligence*, 17:185–203, 1981.

[Horn, 1986] Berthold K. P. Horn. *Robot Vision*. MIT Press, Cambridge, MA, 1986.

[Horswill, 1994] Ian D. Horswill. *Specialization of Perceptual Processes*. PhD thesis, MIT, 1994. AI-TR-1511.

[Hurlbert, 1989] Anya C. Hurlbert. The computation of color. AI Technical Report 1154, MIT, September 1989.

[Huttenlocher and Rucklidge, 1992] Daniel P. Huttenlocher and William J. Rucklidge. A multi-resolution technique for comparing images using the hausdorff distance. Technical Report TR 92-1321, Cornell University, 1992.

[Huttenlocher *et al.*, 1992] Daniel P. Huttenlocher, Jae J. Noh, and William J. Rucklidge. Tracking non-rigid objects in complex scenes. Technical Report TR 92-1320, Cornell University, 1992.

[Ikeuchi and Hebert, 1990] Katsushi Ikeuchi and Martial Hebert. Task oriented vision. In *Proceedings of Darpa Image Understanding Workship*, September 1990.

[Jackson *et al.*, 1996] MacDonald H. Jackson, J. Eric Baldeschwieler, and Lawrence A. Rowe. Berkeley cmt media toolkit api. Berkeley Multimedia Research Center, September 1996.

[Kandel and Schwartz, 1985] Eric R. Kandel and James H Schwartz. *Principles of Neural*

*Science / 2nd ed.* Elsevier, New York, 1985.

[Kelly and Cannon, 1995] P.M. Kelly and T.M. Cannon. Query by image example: the candid approach. Technical Report LA-UR-95-374, Los Alamos National Laboratory, 1995.

[Land and McCann, 1971] Edwin H. Land and John. J. McCann. Lightness and retinex theory. *Journal of the Optical Society of America*, 61:1–11, 1971.

[Lawrence *et al.*, 1997] Steve Lawrence, C. Lee Giles, A.C. Tsoi, and A.D. Back. Face recognition: A convolutional neural network approach. *IEEE Transactions on Neural Networks*, 8(1):98–113, 1997.

[Lee and Xu, 1996] Christopher Lee and Yangsheng Xu. Online, interactive learning of gestures for human/robot interfaces. In *Proceedings of IEEE International Conference on Robotics and Automation*, volume 4, pages 2982–2987, Minneapolis, MN, 1996.

[Lindblad and Tennenhouse, 1996] Christopher Lindblad and David Tennenhouse. The VuSystem, A Programming System for Compute-Intensive Multimedia. *IEEE JSAC*, 1996.

[Lindblad *et al.*, 1995] Christopher Lindblad, David Wetherall, William Stasior, Joel Adam, Henry Hou, Michael Ismert, David Bacher, Brent Phillips, and David Tennenhouse. Viewstation applications: Implications for network traffic. *IEEE JSAC*, 13(5), 1995.

[Lindblad, 1994] Christopher J. Lindblad. *A Programming System for the Dynamic Manipulation of Temporally Sensitive Data*. PhD thesis, MIT, August 1994. LCS-TR-637.

[Luong, 1991] Q.-T. Luong. Color in computer vision. *Handbook of pattern recognition and computer vision*, pages 311–368, 1991.

[Marr, 1979] David Marr. Early processing of visual information. *Transactions of the Royal Society of London B*, 275:483–524, 1979.

[Matthews *et al.*, 1993] James Matthews, Peter Gloor, and Fillia Makedon. Videoscheme: A programmable video editing system for automation and media recognition. In *Proceedings of ACM Multimedia 93*. ACM, August 1993.

[Microsoft, 1995] Microsoft. *Programming with MFC*, volume 2. Microsoft Press, 4 edition, 1995.

[Nagao and Grimson, 1995] Kenji Nagao and W. Eric. L. Grimson. Recognizing 3d objects using photometric invariant. AI Memo 1523, MIT, February 1995.

[Niblack *et al.*, 1993] Wayne Niblack, Ron Barber, Will Equitz, Myron Flickner, Eduardo Glasman, Dragutin Petkovic, Peter Yanker, Christos Faloutsos, and Gabriel Taubin. The qbic project: Querying images by content using color, texture, and shape. Technical Report RJ 9203, IBM Research, February 1993.

[Ogle and Stonebraker, 1995] Virgina E. Ogle and Michael Stonebraker. Chabot: Retrieval from a relational database of images. *IEEE Computer*, 28(9):40–48, 1995.

[Ousterhout, 1994] John Ousterhout. *An Introduction to Tcl and Tk*. Addison Wesley, Reading, MA, 1994.

[Pentland *et al.*, 1996] A. Pentland, R. W. Picard, and S. Sclaroff. Photobook: Content-based manipulation of image databases. *International Journal of Computer Vision*, 18(03):233–254, 1996. MIT Media Lab Vismod Report 255.

[Picard and Minka, 1995] R. W. Picard and T. P. Minka. Vision texture for annotation. *ACM/Springer-Verlag Journal of Multimedia Systems*, 3:3–14, 1995. MIT Media Lab Vismod Report 302.

[Picard, 1996] R. W. Picard. A society of models for video and image libraries. *IBM Systems Journal*, 35(3&4):292–312, 1996. MIT Media Lab Vismod Report 360.

[Pratt, 1991] William K. Pratt. *Digital Image Processing / 2nd ed.* John Wiley and Sons Inc., New York, 1991.

[Rosenfeld, 1969] Azriel Rosenfeld. *Picture Processing*. Academic Press, New York, 1969.

[Rowley *et al.*, 1995] Henry A. Rowley, Shumeet Baluja, and Takeo Kanade. Human face detection in visual scenes. Technical Report CMU-CS-95-158, Carnegie Mellon University, November 1995.

[Sclaroff *et al.*, 1997] S. Sclaroff, L. Taycher, and M. La Cascia. Imagerover: A content-based image browser for the world wide web. Technical Report TR97-005, Boston University, 1997.

[Smith and Chang, 1996] John R. Smith and Shih-Fu Chang. Visualseek: a fully automated content-based image query system. In *Proceedings of ACM Multimedia 96*. ACM, November 1996.

[Smith and Kanade, 1995] M. Smith and T. Kanade. Video skimming for quick browsing based on audio and image characterization. Technical Report CMU-CS-95-186, Carnegie Mellon University, 1995.

[Starner and Pentland, 1996] Thad Starner and Alex Pentland. Real-time american sign language recognition from video using hidden markov models. Vismod 375, MIT Media Lab, 1996.

[Stasior and Tennenhouse, 1996] William Stasior and David Tennenhouse. The viewstation collected papers ii. LCS TR 696, MIT, 1996.

[Subirana and Sung, 1992] Brian Subirana and Kah-Kay Sung. Ridge-detection for the perceptual organization without edges. AI Memo 1318, MIT, December 1992.

[Sung and Poggio, 1994] Kah-Kay Sung and Tomaso Poggio. Example-based learning for view-based human face detection. AI Memo 1521, MIT, December 1994.

[Sung, 1992] Kah-Kay Sung. A vector signal processing approach to color. AI Technical Report 1349, MIT, January 1992.

[Swain and Ballard, 1991] Michael J. Swain and Dana H. Ballard. Color indexing. *International Journal of Computer Vision*, 7(1):11–32, 1991.

[Swain and Stricker, 1993] Michael J. Swain and Markus A. Stricker. Promising directions in active vision. *International Journal of Computer Vision*, 11(2):109–126, 1993.

[Swain, 1990] Michael J. Swain. *Color Indexing*. PhD thesis, University of Rochester, 1990.

[Tamura *et al.*, 1978] Hideyuki Tamura, Shunji Mori, and Takashi Yamawaki. Textural features corresponding to visual perception. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-8(6):460–473, 1978.

[Tennenhouse *et al.*, 1995] David Tennenhouse, Joel Adam, David Carver, Henry Houh, Michael Ismert, Christopher Lindblad William Stasior, David Wetherall, David Bacher, and Teresa Chang. The viewstation: A software-intensive approach to media processing and distribution. *Multimedia Systems Journal*, 3:104–115, 1995.

[Therrien, 1989] Charles W. Therrien. *Decision Estimation and Classification*. John Wiley and Sons, New York, 1989.

[Turk, 1991] Matthew A. Turk. *Interactive-Time Vision: Face Recognition as a Visual Behavior*. PhD thesis, MIT, 1991.

[Ullman, 1984] Shimon Ullman. Visual routines. *Cognition*, 18:97–159, 1984.

[Wactlar *et al.*, 1996] Howard D. Wactlar, Takeo Kanade, Michael A. Smith, and Scott M. Stevens. Intelligent access to digital video: Informedia project. *IEEE Computer*, 29(05):46–53, 1996.

[Wang and Adelson, 1993] John Wang and Edward Adelson. Layered representation for motion analysis. Vismod 221, MIT Media Lab, April 1993.

[Wetherall and Lindblad, 1995] David Wetherall and Christopher J. Lindblad. Extending Tcl for Dynamic Object-Oriented Programming. In *Tcl/Tk Workshop*, Toronto, Ontario, July 1995.

[Wetherall, 1994] David Wetherall. An Interactive Programming System for Media Computation. Master's thesis, MIT, 1994. LCS-TR-640.

[Woodfill, 1992] John I. Woodfill. *Motion Vision and Tracking for Robots Dynamic Unstructured Environments*. PhD thesis, Stanford, 1992.

[Wu *et al.*, 1995] J. K. Wu, A. Desai Narasimhalu, B.M. Mehtre, C.P. Lam, and Y.J. Gao. Core: a content-based retrieval engine for multimedia information systems. *ACM/Springer-Verlag Journal of Multimedia Systems*, 3:25–41, 1995.

[Young *et al.*, 1995] Mark Young, Danielle Argiro, and Steven Kubica. Cantata: Visual programming environment for the khoros system. *Computer Graphics*, 29(2):22–24, May 1995.

[Zentner, 1993] Daniel Zentner. *Computing Times Square Empty*. Bachelor's Thesis, MIT, 1993.

[Zhang *et al.*, 1995] HongJiang Zhang, Shuang Yeo Tan, Stephen W. Smoliar, and Gong Yihong. Automatic parsing and indexing of news video. *ACM/Springer-Verlag Journal of Multimedia Systems*, 2:256–266, 1995.