LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TR-640

AN INTERACTIVE PROGRAMMING SYSTEM FOR MEDIA COMPUTATION

David J. Wetherall

September 1994

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

An Interactive Programming System for Media Computation

David J. Wetherall

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

Abstract

As digital video is manipulated by increasingly powerful computers, many new applications are becoming viable. This report investigates the programming language aspects of controlling such video applications. It presents the design, implementation, and use of PAVES, a direct manipulation system that combines aspects of visualization and multimedia systems to form an interactive video programming environment.

PAVES is novel in the degree to which it emphasizes liveness and in its approach to extensibility. It extends the VuSystem media processing toolkit through flow graph and textual programming windows. Flow graph windows are used to control the media processing component of applications. Textual programming windows are used to issue interpreted commands and view source code. They work together to allow the user to combine methods as needed.

While developing PAVES, I confronted a number of programming language issues related to the support of multiple program representations. I defined a cooperative model for translating across these representations and implemented the model by leveraging a presentation-style synchronization approach and Object Tcl, an object-oriented extension to the Tcl language.

Keywords: programming languages, digital video, user programming

Acknowledgements

David Tennenhouse, my supervisor, encouraged me when I needed it most and afforded me the latitude to develop ideas into a thesis.

Chris Lindblad was a source of guidance at a formative time, as well as architect of the VuSystem, which I depended upon so much.

All the members of my research group, my friends, provided the jovial atmosphere that saw me through.

The University of Western Australia supported this research with a Hackett Studentship, as did the Advanced Research Projects Agency of the Department of Defense with funding monitored by the United States Air Force (AFSC, Rome Laboratory) under contract No. F30602-92-C-0019.

Contents

1 Introduction **11**
1.1 Digital Video and Computers 12
1.2 The ViewStation 13
1.3 User Programming Systems 15
1.4 Interactive Video Programming with PAVES 16
1.5 Issues in the Development of PAVES 17
1.6 This Report 18

2 Related Work **21**
2.1 Programming in Multimedia Systems 21
2.2 Media Support in User Programming Systems 24
2.3 Perspective 26

3 A Cooperative Approach **27**
3.1 Cooperative Programming Levels 27
3.2 An Embedded Object System 30
3.3 Synchronization of Views 31
3.4 Research Scope 33
3.5 Perspective 35

4 User Programming with PAVES **37**
4.1 A Guided Tour 37
4.2 Broader Tasks 45

5 Design of PAVES **49**
5.1 Flow Graph Windows 49
5.2 Supporting Windows 56
5.3 Structuring Conventions 59

6 Object Tcl **65**
6.1 Tcl as a Starting Point 65
6.2 Introducing Objects 66
6.3 Classes and Inheritance 69

6.4 Class Implementation 74
6.5 Introspection 74
6.6 Perspective 75

7 Results and Conclusions **77**
7.1 Experience with PAVES 78
7.2 Conclusions 80
7.3 Further Work 83
7.4 Summary 84

A Structuring Conventions used by PAVES **85**
A.1 Standard Classes 85
A.2 Reflective Methods on Entities 86
A.3 Reflective Methods on Ports 87
A.4 Reflective Methods on Collections 88
A.5 Methods Supporting Views on Entities 88
A.6 Methods Supporting Views on Ports 91
A.7 Methods Supporting Views on Collections 91

B Object Tcl Reference **93**
B.1 Standard Objects 93
B.2 Object Methods 94
B.3 Class Methods 97
B.4 The Method Environment 99
B.5 The Class Precedence Ordering 99

List of Figures

1.1 A preview of the PAVES user interface . . . 12
1.2 Organization of a VuSystem Program. . . . 13
1.3 The VuNet Environment. . . . 14
1.4 Comparison of Programmable Systems. . . . 15

3.1 Model of Representation Shifts . . . 28
3.2 Presentation-Style Synchronization Model . . . 32
3.3 Three *views* of a video program generated by PAVES . . . 33
3.4 The spread of VuSystem Programming Methods . . . 34

4.1 The BlueScreen Program with its Flow Graph . . . 38
4.2 Before Grouping the BlueScreen Modules . . . 40
4.3 After Forming the BlueScreen Group . . . 40
4.4 Peering Inside the BlueScreen Group . . . 42
4.5 Control Panel for the BlueScreen program . . . 43
4.6 Sample Code Fragment Window . . . 44

5.1 Relationship of Xt Widgets to PAVES . . . 52
5.2 Skeleton of the `draw` method . . . 54
5.3 Model of each View . . . 55
5.4 Skeleton of the `panel` method . . . 57
5.5 Skeleton of the `describe` method . . . 58
5.6 Program Structure . . . 60
5.7 Model of an Entity . . . 61
5.8 Opaque Grouping of Entities . . . 63

6.1 Making a stack object by specializing a generic object. . . . 67
6.2 Superclasses and Method Dispatch . . . 70
6.3 Adding protection to the Stack class with the Safety mixin. . . . 71
6.4 Class Relationships for SafeStack . . . 72
6.5 A Class Relationship . . . 73

B.1 Algorithm for computing the precedence list. . . . 100

List of Tables

5.1 Entity Methods Supporting Views 62
5.2 Reflective Entity Methods 62
5.3 Reflective Port Methods 62
5.4 Collection Methods Supporting Views 63
5.5 Reflective Collection Methods 64

6.1 Object Methods 68
6.2 Method Environment 69
6.3 Class Methods 73

Chapter 1

Introduction

Many new uses of digital video are becoming viable as it is manipulated by increasingly powerful computers. With the right programming tools, video applications can become more intelligent and responsive through greater amounts of media processing [36, 22]. This report investigates the programming language aspects of controlling such video applications. It presents PAVES, a direct manipulation system that allows users to control applications while the applications simultaneously manipulate live video. PAVES, shown in Figure 1.1, meets the needs of interactive video programming by combining the user programming techniques of visualization systems with the temporally-sensitive processing abilities of multimedia systems. PAVES is novel in the extent to which it emphasizes liveness of both program and video data, and in its approach to extensibility through the cooperation of graphical and textual programming methods.

PAVES extends the VuSystem media processing toolkit [22] with several user programming windows that are accessible across a range of video programs. A visual flow graph can be used to observe and control the pattern of media processing implemented by the application. Control panels, code fragments, and interpreters work in conjunction with the flow graph so that the user may employ both textual and graphical methods to solve programming tasks. By exploiting a specially constructed object system and synchronization model, PAVES can automatically generate its programming windows and allow each to be independently and safely used as the programs continue to run. This provides users the flexibility to choose the programming method most suited to the task at hand.

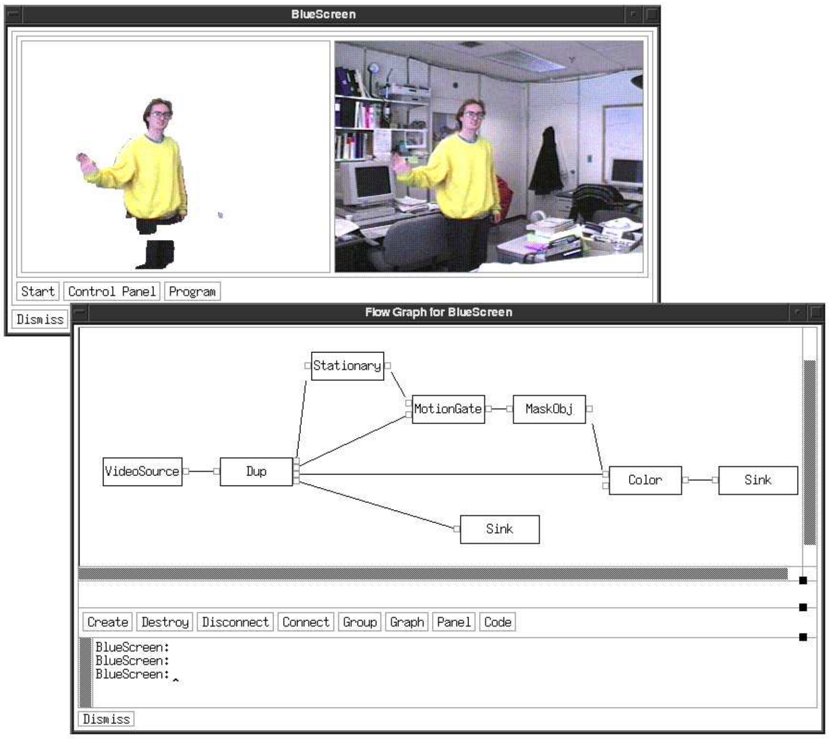

Figure 1.1: A preview of the PAVES user interface

1.1 Digital Video and Computers

In 1991, Apple Computer Inc. introduced its Quicktime [15] software toolkit for manipulating time-based media, such as video. Quicktime-based applications gave many personal computer users a limited capability for recording, replaying, and combining video. Products for the associated technologies of video capture and storage are also commercially available. Digital video grabbers, such as SunVideo and DEC Sound & Motion, allow video sequences to be captured for later computation. Image compression and decompression codecs, such as those based on the JPEG [19] and MPEG [18] standards, facilitate the transfer and storage of video by reducing its sheer size to more manageable proportions.

The falling relative costs of processing video make it possible to encode simple algorithms in software and apply them interactively. This was not practical with previous generation workstations where even copying video frames was an expensive operation. But the memory bandwidth and processing capacity of modern workstations enable them to perform simple processing well above the full-motion video rate [22]. Developments in high-speed networking and video codecs are also encouraging distributed applications where larger computations may be accommodated by splitting them across multiple computers [34].

Using software to apply algorithms to video yields very different applications than those commonly given as examples of multimedia programs. Teleconferencing systems, multimedia encyclopedias, and the like make poor use of the general computation possibilities of computers. The computer is essentially being used to move digital data in a timely fashion.

Intelligent video programs will be capable of making decisions based on video content. For example, a computer scribe may automatically capture and distill a set of notes from a camera pointed at a blackboard during a lecture [36]. A programmable television agent may be capable of reorganizing a video broadcast to meet a user's preferences. Or perhaps a program will be able to distinguish between video-conference participants and their background so that the participants can enhance their images with different background scenes.

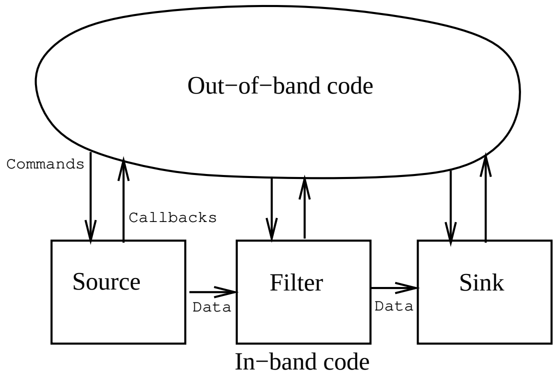

Figure 1.2: Organization of a VuSystem Program.

1.2 The ViewStation

The ViewStation [38] is an environment that is specialized for the study of intelligent and responsive multimedia applications. Its software development component, the VuSystem [22], focuses on *computer-participative* multimedia applications, where the computer not only manipulates media but also digests it and performs independent actions based on media content. This makes it suitable for developing the kind of applications sketched as scenarios in the previous section.

VuSystem applications are modeled as a graph of reusable processing modules through which video and other data flow. Some modules correspond to video

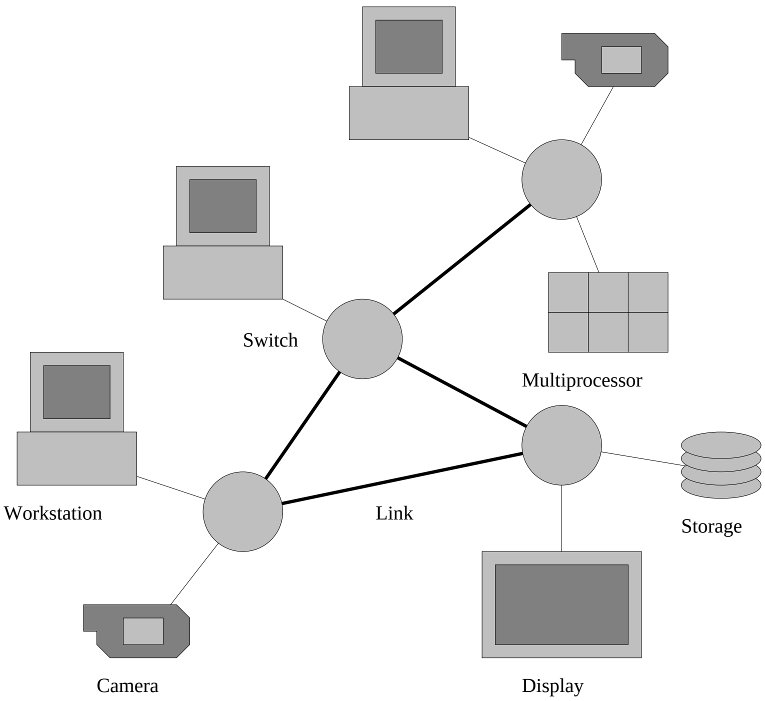

Figure 1.3: The VuNet Environment.

capture from the outside world, some to computations such as compression, some to storage and retrieval, and some to presentation to the user.

Several levels of programming are available to the application developer, each suited to a different task. The basic programming division arises from the model of a program as separated into an in-band data stream and an out-of-band control message queue. This is shown in Figure 1.2.

Programming of the time-critical data stream, where video frames are directly manipulated, is handled by a library of efficient C++ modules. Generic filtering modules may be specialized as a simple but constrained way of implementing new computations. Alternatively, new modules may be written from scratch, with few operational constraints.

Programming of the control message processing, in which user events are decoded and user feedback is displayed, is expressed with an extended version of the Tool Command Language (Tcl) [32]. The system contains an embedded Tcl interpreter, which provides great flexibility in re-configuring and extending the runtime environment. Tcl scripts are used to combine modules into applications, as well as describe standardized user interface panels for configuring each module.

The other component of the ViewStation is the VuNet [1]. It is an ATM-based gigabit per second local area network, shown in Figure 1.3. By complementing the VuSystem, it provides a computationally rich environment in which distributed video applications may be studied. A variety of configurations are possible, linking workstations and custom video capture peripherals [2].

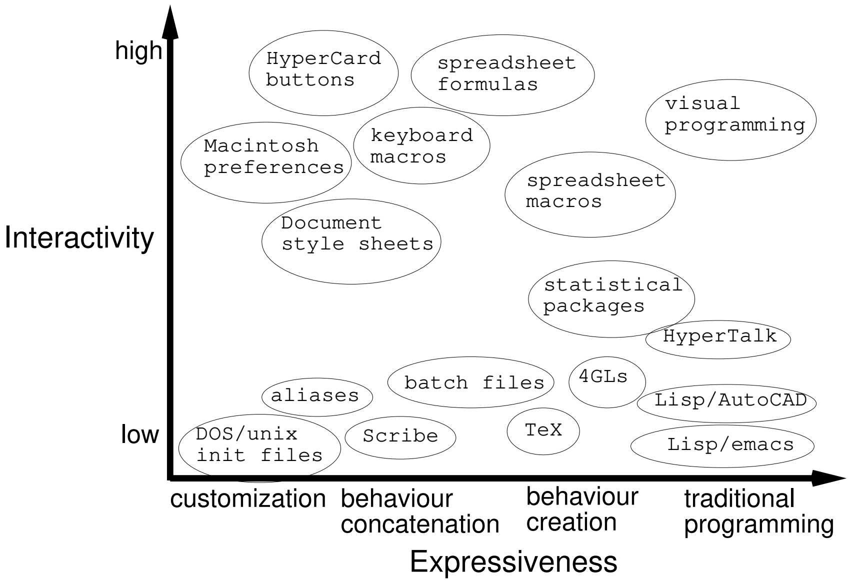

Figure 1.4: Comparison of Programmable Systems.

1.3 User Programming Systems

The VuSystem provides specialized media manipulation abilities to application developers. In contrast, end user programming systems such as PAVES provide specialized interaction facilities to the end users of applications. With little or no training, the end user can configure an application to be better suited to the task at hand.

Many different user programming systems have been devised, with varying degrees of success; work continues in trying to understand the process of user programming [23, 11]. Figure 1.4, reproduced here from Nardi [30], conveys a sense of the diversity of these systems. It roughly characterizes them by their level of programmability and degree of interaction.

Over a decade of experience with spreadsheets, CAD systems, and statistical packages prove that successful user programming systems can be created. Spreadsheets are the most popular user programming system currently available.

They combine the familiar metaphors of a worksheet and calculator to allow users to construct their own solutions to simple numerical problems. CAD systems are designed to let users piece together designs, such as circuit schematics, that may be startlingly complex. They often employ graphical programming-like facilities to achieve this goal. Similarly, statistical packages can employ languages to describe the analysis of sets of data.

Most successful general-purpose user programming systems have a textual programming language as their base. Examples include LOGO, HyperTalk, Visual Basic, and Tk. The LOGO language is for drawing pictures, and uses the metaphor of a turtle moving about the screen to help children learn about mathematics and computational processes. HyperCard [14] uses the metaphor of a stack of cards containing objects to build hypermedia style documents. Programs are written in a verbose textual language called HyperTalk, with fragments of the overall program attached to interface objects to control how they respond to different events. Visual Basic and Tk [33] work in a similar manner, but use an interface metaphor based on grouping window elements and their controls.

Visual programming languages have also received much attention. Graphics has the advantage of a low learning barrier, since much syntax associated with textual languages is eliminated. But visual languages have proven cumbersome at expressing control constructs, such as iteration and conditionals, and few purely graphical systems have proven of general value. To alleviate this difficulty, many systems combine graphics with text or target a restricted application domain or set of users. Some recent visual languages, for example, have targeted VHDL hardware descriptions [28] and the specification of database queries [8]. These problems have a natural hierarchical decomposition that can be shown visually.

Other successful visual languages have based their representation on the flow of data between computational processes. Flow graphs reveal the broad scale structure of a program in terms of the relationship between its input and output data, rather than as a navigable database of related operations. This view has proven valuable in problem domains related to video, such as image processing [40] and scientific visualization [39].

1.4 Interactive Video Programming with PAVES

This report describes the design, implementation, and use of PAVES, an interactive video programming environment. PAVES stands for Programming Active Video with an Embedded System. It brings the advantages of user programming systems to bear on computer-participative multimedia applications.

From the user's point of view, PAVES appears as a set of programming windows.

Because it is embedded in an application toolkit, these windows are available in many video programs, and form a consistent higher level interface for controlling video programs. A direct manipulation flow graph window shows the overall pattern of media processing implemented by the application and allows it to be changed in terms of a visual language. An example graph is shown in Figure 1.1. Its vertices represent processing modules, and its edges represent the flow of media between them from left to right. Other windows can be generated to show finer detail controls and relate the flow graphs to the textual structure of the program.

Video applications in PAVES are implemented with the VuSystem toolkit. In contrast to Quicktime, this is a programming foundation suited to computer-participative media applications. With its companion high-speed network, the VuSystem is tailored for experimentation in the future uses of video in the workplace, and permits flexible implementation choices. It allows PAVES to be used to explore the class of video programs that performs significant computation and so is the most likely to benefit from interactive programming facilities.

Two features distinguish PAVES from other visual language tools. It manipulates active video, where both program and video data are completely live. Video frames continue to flow in real-time as it is used. Video processing continues as the program is changed, and PAVES ensures that the flow graph representation is always correct. PAVES combines graphical and textual programming methods so that they can be used cooperatively. As well as the visual flow graph representation, an interpreter allows textual commands to be entered for immediate evaluation and code fragment panels show the text that corresponds to selected portions of the program. PAVES translates between graphical and textual forms to allow either to be used for a given task and to help the user form an association between the two.

PAVES is targeted at customization and experimentation tasks, as well as prototyping and more general program development. It is intended to be accessible to inexperienced programmers as well as act as an overview level for experts. By providing continuous video feedback as the flow graphs are altered, it allows the user to tune the pattern of media processing. Through its embedding in an application toolkit, it is available for program inspection or reconfiguration as needed.

1.5 Issues in the Development of PAVES

Two issues need to be addressed in the design of PAVES. To integrate graphical and textual programming methods, a means of translating between the different program representations is required. To simultaneously support live video and manipulations on programs as they run, a synchronization mechanism is required.

To use the PAVES facilities on video programs developed textually, as well as to capture visual work done with PAVES for later reuse, requires translating between the visual and textual representations of an application. This is a difficult task because little is specified a priori about the behavior of video programs. The translation mechanisms must be general enough to apply to a range of programs, but flexible enough to accommodate the features of each program. PAVES defines a cooperative model and uses the properties of object-oriented programming to implement it.

A synchronization mechanism preserves the illusion of direct manipulation by ensuring that the flow graph window accurately reflects the program to which it corresponds. Because video programs continue to run as they are manipulated, it is possible for them to alter their pattern of processing. Any such changes must result in the updating of the flow graph. This is difficult to accomplish because all PAVES facilities are intended for a range of video programs. They are automatically generated and cannot be specially coded to accommodate the vagaries of a particular video program. PAVES implements a generalized synchronization scheme with a presentation-style approach [4].

In tackling both these issues, the interpreted programming environment of the VuSystem is valuable. It provides a language midway between the visual and C/C++ program languages, a reference point that can be related to both languages. Because it is interpreted, it can be used to issue commands at any time, including commands that investigate the current structure of the program. Much of the foundation of PAVES relies on organizing video programs into object structures that are expressed in terms of an interpreted VuSystem language.

1.6 This Report

This report presents PAVES, an interactive video programming system. To meet the needs of controlling programs while they are manipulating live video, PAVES combines the aspects of visualization systems with those of multimedia systems. Related work in these fields is presented in Chapter 2.

PAVES employs a programming approach that allows both graphical and textual programming methods to be used simultaneously. I refer to this as a cooperative approach, and describe it in Chapter 3. It appears to the user as a series of programming windows. Both visual windows, including a direct manipulation program representation based on flow graphs, and textual windows, including an interpreter for issuing commands, are available. Their use is demonstrated in Chapter 4. To implement them, a means of translating between graphical and textual forms is needed, as is a means of keeping these forms synchronized with the program they represent. The design and implementation of these mechanisms and the structuring conventions that support them are described in Chapter 5. They build on top of a specially constructed object system that provides the infrastructure necessary to programmatically manipulate video programs. The design of this language, Object Tcl, is presented in Chapter 6. Appendices A and B serve as detailed references for the structuring conventions and Object Tcl, respectively.

Experience with PAVES has helped to evaluate the role of interactive video programming, as well as more general programming concerns such as the value of a cooperative approach and use of abstractions. It has also suggested further work in these areas. These results are discussed in Chapter 7.

Chapter 2

Related Work

A body of research in multimedia systems and user programming systems is relevant to this report. In the introduction I have discussed these areas in general, and this chapter now investigates specific prior work. I concentrate on the intersection of multimedia systems and user programming systems by examining the programming support in multimedia systems and conversely the media support in user programming systems. The contrast helps to place in perspective the interactive programming system presented in this report.

2.1 Programming in Multimedia Systems

Besides the VuSystem [22], there are other multimedia systems that support programming activities. In this section I investigate the programming aspects of multimedia systems that are designed from the ground up to manipulate video, since its timeliness and visual properties drive this research.

Most often, programmable video systems are software libraries with an application programming interface (API) that allows their routines to be combined using a textual programming language. They are described as *toolkits* and are targeted at professional programmers seeking to create new products. Their development environments are typically not interactive, but are worth examining because the programming models they employ are usually well-developed from a programming language standpoint.

Multimedia applications can also provide programming-like facilities for the end user. These may include a fixed set of primitives for manipulating video, along with a means of combining the primitives, and support for stereotyped tasks such as the customization of preferences. Though they do not constitute programming in the traditional sense, these systems are worth examining because they explore frameworks that allow users to specify operations on video and other media.

2.1.1 Quicktime and other toolkits

Apple Computer's Quicktime [15] is a commercial toolkit for manipulating time-based media in the Macintosh environment. It is notable as the most popular system in use today for programming multimedia applications. Many existing applications, such as word processors, have been extended to support video by incorporating its functionality.

The developer programming model used in Quicktime is based on the notion of *components*, each of which implements some processing function. Components form a primitive object system, and serve to encapsulate drivers for special hardware, such as video sequence grabbers. A set of pre-defined components provides all the functionality necessary to add video playback capability to applications. As such, Quicktime may be argued to take a document-centric perspective, where video is treated as simply another type of data managed by the application.

Quicktime provides no direct programmability for the application user, leaving this task to the application developer. It does include several user interface components, and encourages developers to use them. These components were created to reduce development effort and promote interface consistency. They provide standard ways for the user to interact with *movies*. A movie controller element lets the user navigate through video clips during playback; standard dialog boxes let the user configure parameters such as brightness, hue, frame rate, compression type, and audio input level prior to capture.

Other platforms have developer toolkits, but again they provide no direct programming features for the application user. Microsoft's Video For Windows [7] is the equivalent of Quicktime for the Windows environment, and includes a set of tools for editing video sequences. Solaris LIVE! is a set of libraries [17] and an architecture for Sun workstations. It emphasizes media computation more than does Quicktime but supports no interface elements except for media display.

2.1.2 Premiere, VideoShop and Director

Video editing systems such as Adobe Premiere, DiVA VideoShop, and MacroMind Director [12, 6, 16] provide users with facilities for combining video segments in sophisticated ways. VideoShop and Director emphasize the integration of video with other media more than Premiere, but for our purposes they all present a higher level interface to Quicktime and equivalent toolkits — an interface more accessible to users.

Video editors do not present a programming model to the user, but do provide a visual framework for directing the processing of digital video. Sequences are composed in a workspace that shows the temporal dimension of each of many simultaneously active tracks; objects in this video programming model are media rather than processing operations.

Filtering operations can be used to transform the color, spatial, and temporal components of sequences of images, or *tracks*. They include brightening, blurring, sharpening, zooming, panning, enlarging, and freezing frames. A variety of operations such as cuts, fades, wipes, and morphs can be used to combine tracks. There is no form of procedural abstraction for the user, though some extensibility may be provided. Premiere, for example, has standardized an interface for third-party processing modules, or *plug-ins* that augment its capabilities.

Despite their accessible interfaces that let users direct the simple processing of video, editing systems are a weak model for the interactive system presented in this report. Their lack of programming language features could be remedied. But they are modeled on the previous generation of analog editing suites, which treat editing as an off-line activity. They are not suited to live media, nor do they produce their video result interactively. Instead, they transform a given sequence of video into a new one in an unconstrained amount of time, without provision for producing the new sequence as it is needed for play-out.

2.1.3 VideoScheme

VideoScheme [24] is a programmable video editing system developed to research the automation of routine video processing tasks. It addresses the programming weaknesses seen in Premiere and similar systems by using the Scheme language to specify operations. This brings the generality of a real language to bear on the specification of video processing. It has enabled VideoScheme to be used for decision and analysis tasks such as silence and cut detection.

In terms of this work, VideoScheme is limited in two directions. It requires users to be familiar with a textual programming language. Though Scheme is a high level and interactive language, hiding details such as memory management and compilation, it does not compare with the easy-to-use graphical interfaces of Premiere and the like. The situation is analogous to that of the VuSystem and its Tcl programs, on top of which the research presented in this report is built.

VideoScheme also inherits the off-line or batch processing nature of editing systems. Though the Scheme program may be developed interactively, it cannot produce its video result in real-time since there is no resource scheduling based on real-time.

2.1.4 Medusa

The Medusa applications environment [41] is a prototyping system for distributed video and audio applications, based on a peer-to-peer architecture for controlling networked multimedia devices. It uses simple, reliable, and unbuffered channels for connections and capability-based proxies for security. In Medusa, programs are modeled as *active objects*, which are implemented as C++ classes. A Medusa server that is implemented as an extended Tcl/Tk interpreter provides an interactive interface for composing video programs and building graphical browsing and debugging tools.

Like VideoScheme and the VuSystem, Medusa lacks a graphical programming environment for specifying video computations[1]. It is similar to the VuSystem in its ability to handle live video and its use of Tcl as a means of combining video manipulation modules into programs. It differs in its emphasis on distributed programs rather than visual processing, with a correspondingly less well-developed system for constructing program user interfaces.

2.2 Media Support in User Programming Systems

As the use of video becomes widespread, it is natural to expect it to be the subject of specialized user programming systems and be incorporated into others. In this section, I examine existing user programming systems that do not support video but could parallel future systems that do support it. These systems are designed for specialized problem domains that share characteristics with video processing: image processing, computer vision, and scientific visualization.

2.2.1 AVS

The Application Visualization System (AVS) [39] has set a de facto standard as a commercial tool for scientific visualization. It provides a visual programming interface with which users may combine processing elements into a program that represents the visualization. Different colored connections are used to convey various types of data and constrain the connections between modules to those that

[1]Work is in progress to extend the interactive environment of Medusa, but no specifics are available to me at this writing

are legal. Given a program graph, raw experimental data is passed through it, being processed along the way and then presented.

Though it does not handle real-time data, AVS emphasizes liveness of the program and result by allowing changes to the program at all times and re-computing the visualization result as needed. It also constructs a simple user interface for the visualization based on the processing modules it contains. This interface allows module parameters to be examined and adjusted. The system is extended by adding to the library of modules, linking in routines written in languages such as C or FORTRAN. Alternatively, AVS may be linked into existing applications.

AVS offers a well-tried model for a visual programming system, though it offers weak abstraction mechanisms for the user. The programming system presented in this report draws on many of its features, while adapting them to support video.

2.2.2 Cantata, VIVA, and MAVIS

There are a variety of media flow style visual languages for image processing, including Cantata, VIVA, and MAVIS [40, 37, 31].

Cantata provides a graphical front to the KHOROS signal processing system. It lets the user construct a pictorial representation of processing operations, from which a valid KHOROS program is constructed and executed. This has proven a convenient way to prototype applications.

VIVA is similar to Cantata. It is notable for emphasizing "liveness", the extent to which program elements provide feedback to the user as they are manipulated. In a VIVA session, editing the program graph causes the results that are dependent on it to be updated. MAVIS too is fully live, incorporating liveness with continuous image processing as part of its design for computer vision tasks. Elements in MAVIS represent processes that execute continually, firing whenever their input conditions are met. Allowing users to access these conditions and control resource scheduling policies lets them model real-time image processing systems.

These systems are useful precedents for the interactive system, but lack means for combining the visual programming level with other levels. AVS, with its capabilities for visually building new applications as well as linking to existing textual applications, provides greater flexibility. As was the case for AVS, the systems also provide weak languages because they do not support user level abstraction facilities.

2.3 Perspective

The interactive programming system presented in this report is novel in the way it uses visual programming in conjunction with video processing to form a video programming environment accessible to application users.

The systems discussed in this chapter do not adequately support emerging video applications. Visualization and editing systems support user interaction, but not the timely computation, distribution and presentation of video. Multimedia systems support timely manipulations, but provide weak programming facilities for the user.

Beyond the combination of these systems, there are two important differences as compared to prior work. The interactive system emphasizes liveness to an unusual extent because even the video data may be live. Further, it is intended to be used in a different manner than most editing, visualization, and image processing environments.

Media in the interactive system is always flowing because it is constructed on top of a programming system that respects the temporal characteristics of the data. Changes to the system take effect immediately, and the user interface always reflects the instantaneous state of the program.

This differs from digital video editing systems, which combine video segments to create further video segments. Their user interfaces may attempt to be responsive, yet the combination process is inherently off-line because it must run to completion before any output may be replayed. Similarly, visualization systems may provide responsive interfaces, but they are fundamentally simulation style environments. Rather than trying many image processing operations to find the best for a given dataset, video may be digitized from a camera, processed, and then displayed, all without being stored in any persistent form.

The interactive system is intended to be available across many video programs, as well as work in conjunction with the other programming methods of the VuSystem. This differs from visualization and image processing applications, which are often standalone environments in which a program of analysis is prototyped. The focus on live video means that the system will often be used to manipulate running applications, as opposed to constructing fresh ones. Here, reprogramming may be secondary to simply using the application, and should be consistently available on demand without interfering with program operation.

These modes of use place greater weight on the integration of programming methods than is found in systems such as Cantata. They favor abilities such as hiding the visual program editor and linking to existing applications, both features of AVS.

Chapter 3

A Cooperative Approach

In this chapter, I present the approach used to design and implement PAVES as an extension to the VuSystem. The approach stresses the integration of the media flow graphs and other graphical programming tools with the remainder of the toolkit through cooperative methods of programming. I present a model for achieving this that addresses the difficulties of providing programming tools across a range of video programs, as well as synchronizing them with the state of the application and hence each other. It is implemented by embedding an object system and presentation-style model in the VuSystem.

3.1 Cooperative Programming Levels

I developed a *cooperative* model that allows the flow graph and textual programming windows to be used cooperatively. In PAVES, flow graphs can be used to manipulate VuSystem programs, and interpreted VuSystem commands can be used to manipulate program flow graphs. Each is a programming facility accessible from the other.

This strategy addresses a weakness of many visual programming systems. They are often well-suited to their specific tasks, but can lock users to their interface for a larger set of tasks, and become frustrating to use after a promising beginning.

Graphical shells, such as the Macintosh Finder, exhibit the shortcomings of approaches that are not cooperative. They are apt for small file management tasks, but not appropriate for more complicated tasks. Other shells, such as the command line utilities found in UNIX and DOS, can accomplish more complicated tasks, such as archiving files not recently accessed. But they can prove cryptic to the beginner. There is no easy way to combine the strengths of both methods.

Applications such as SchemePaint [9], on the other hand, demonstrate the use of cooperative programming methods. SchemePaint is a graphics application that augments a direct manipulation paint system, similar in spirit to MacPaint, with an interpreter for graphics-enriched Scheme. This allows the simultaneous use of different drawing methods, letting the user decide how to accomplish each task.

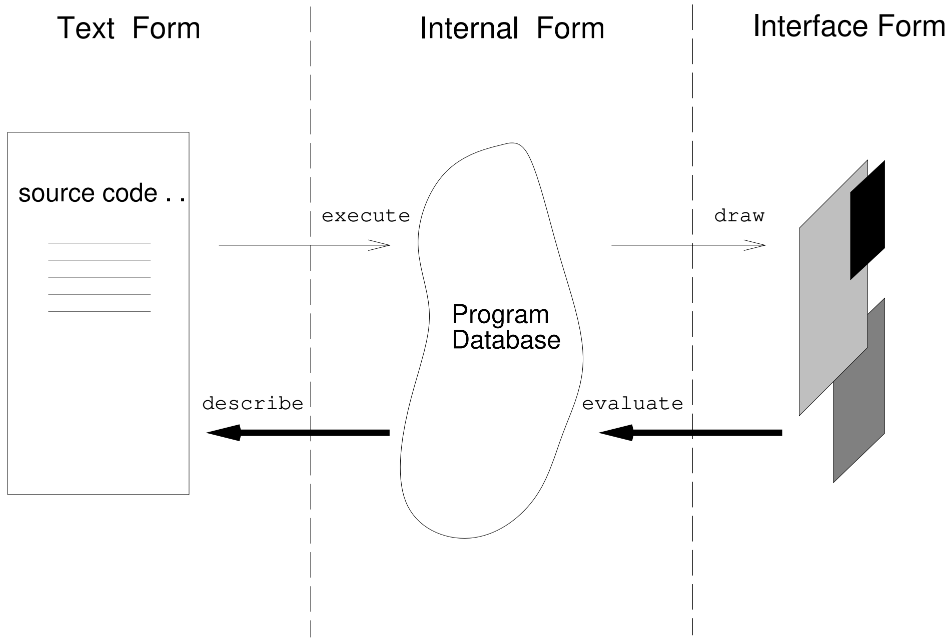

Figure 3.1: Model of Representation Shifts

3.1.1 Defining Cooperation

An abstract model of the cooperating programming levels of PAVES is shown in Figure 3.1. Three *forms* of representation of a single video application can be seen. Each generic form is presented to the user through different types of *views* that capture observable information about the executing program image. Many different views can reveal different aspects of the application.

In the middle, the *internal form* represents the video program as it evolves from moment to moment. Its views are depicted by the program database because they provide momentary direct access to the running VuSystem application. An interpreter is a view of internal form.

The *interface form* models the executing program image across the graphical user interface. Its views are depicted by windows because they provide a direct manipulation representation of the application. The media flow graph, showing the pattern of processing currently implemented by the application and allowing it to be altered, is a view of interface form.

The *text form* models the running video program in terms of the source code that corresponds to it. It is extended Tcl code that is interpreted by the VuSystem shell to run the application, and is edited as the traditional means of programming. Views of text form are depicted by lines of source code because they provide a snapshot of a portion of the program database at the moment they were constructed.

The path moving from left to right represents the course of action for traditional applications: stored code is interpreted to yield a program image, which presents a standard interface to its users. It is the reverse path, from right to left, that enables cooperative programming between different interfaces.

In PAVES, the flow graph may be used as a consistent means of expressing one-time customization tasks (such as tuning the appearance of video and resource usage) across video programs. It may be used for rapid prototyping by starting with an empty application, constructing a program interactively, and then saving it as source code for later refinement.

Conversely, a VuSystem program written directly in the interpreted language may be examined with the flow graph interface, perhaps for a better understanding of its function, or perhaps to be reconfigured and saved. This last type of re-programming is more powerful than the customization facilities found in most applications because any change that may be expressed in the graphical language may be effected, not just a fixed set of changes anticipated by the program developer.

These modes of use require a means of mapping between the interface, text, and internal forms, since any view can be used to drive the others. The mappings are represented in Figure 3.1 with the addition of the *evaluate* and *describe* operations.

The *evaluate* operation represents the interactive re-programming of the application via its graphical interface, such as edit actions on the media flow graph. Since the edit actions are not artificially restricted, they may effect general program changes. For example, a user might remove components of the application, rearrange them, or substitute entirely new components. This generality requires the user interface to be updated to match the altered program, complicating the usually straightforward *draw* operation. For example, controls associated with components that no longer exist must be automatically removed.

A *describe* operation represents the serialization of the program into equivalent source code, code of the same type originally interpreted to launch the program. It allows the application to be saved in Tcl format. This code may be refined as well as reused, in contrast to lower level representations, such as a suspended process image, that are opaque to further development.

Determining the source code that is in some sense equivalent to a running application raises the issue of what state should be observed. Some state information captures the intent of the application, such as the sequence of media processing operations it performs. Other state information pertains only to a particular run of the program, being derived from the content of the video and other environmental bindings. These types of state must be separated during the *describe* operation.

3.2 An Embedded Object System

The natural approach for adding an interactive programming interface to the VuSystem is to embed it in the toolkit. This has the advantage of reinforcing causal programming use, since interactive facilities are then accessible from all video programs if and when they are needed. There is no separate environment for development.

By embedding PAVES in the VuSystem toolkit, a consistent user programming interface is provided for an entire range of applications. This benefits the user. It also minimizes effort on the part of the application programmer, since applications need contain little, if any, code dedicated to generating their own media flow interfaces. Rather, they expect to gain the advantage of automatic user programming tools by adhering to the toolkit structuring conventions.

Embedding also serves to delimit the class of video programs PAVES can manipulate and facilitate the mappings between representations. Programs written with the VuSystem toolkit must adhere to its coding conventions. Greater freedom in program form will translate to fewer and more general manipulations for a given level of implementation complexity. The conventions exist to ensure there is sufficient structure to map between forms. They imply that all toolkit programs have the potential to be manipulated with PAVES.

3.2.1 Object Tcl

The basis I have used for mapping between representations is the Object Tcl language, constructed as part of the research presented in this report. Toolkit structuring conventions alone cannot implement the mapping. A scheme to recover structure is needed, and it must be dynamic since toolkit programs may evolve during their execution. To affect all video programs, the scheme must be specified in a generic manner, yet it must account for the peculiarities of each program.

Object-oriented programming provides inheritance and naming properties that are well-suited to capture the patterns of similarities and differences between programs. Objects naturally model a program of interconnected media processing modules and the operations they undergo.

An example of applying this object model is that of printing objects in human readable form. Even though objects differ in their semantics, each object may be printed by using a consistent syntax. Many objects may have similar printing needs, such as different types of number that share a common format. This may be accomplished by inheriting behavior. Other objects may require special handling, perhaps the number 3.14159... should be printed as π. This may be accomplished by specializing behavior.

The object system is useful for even simple manipulations of programs. But it becomes essential to capture more complicated patterns of behavior. Abstraction operations, where a group of modules is treated as a single module, demonstrate the power of the object system: operations on groups can readily be defined in terms of combinations of the operations on the group members.

3.3 Synchronization of Views

To preserve the illusion of direct manipulation, a strategy for updating the user interface to match the executing program is needed. A video player program, for example, may instantiate a set of objects that depends on the compressed format of the video and whether the video is local or remote. Similar unanticipated behavior can occur in many useful video programs, and the flow graphs and other windows must be able to cope with it. Since the windows are generated by programmatic means, however, it is difficult to special-case operations that may cause inconsistencies, and a more general scheme is needed.

Providing a synchronization scheme creates significant flexibility beyond that of supporting unpredictable programs. It does not matter where the configuration change that triggers an interface update occurs. This means that the simultaneous use of different programming levels is supported by synchronization. When views of interface, internal or text form are capable of accomplishing a given task, any may be used with the same result. Further, a changing and functionally overlapping set of interface displays can be in use, and all views will be maintained correctly.

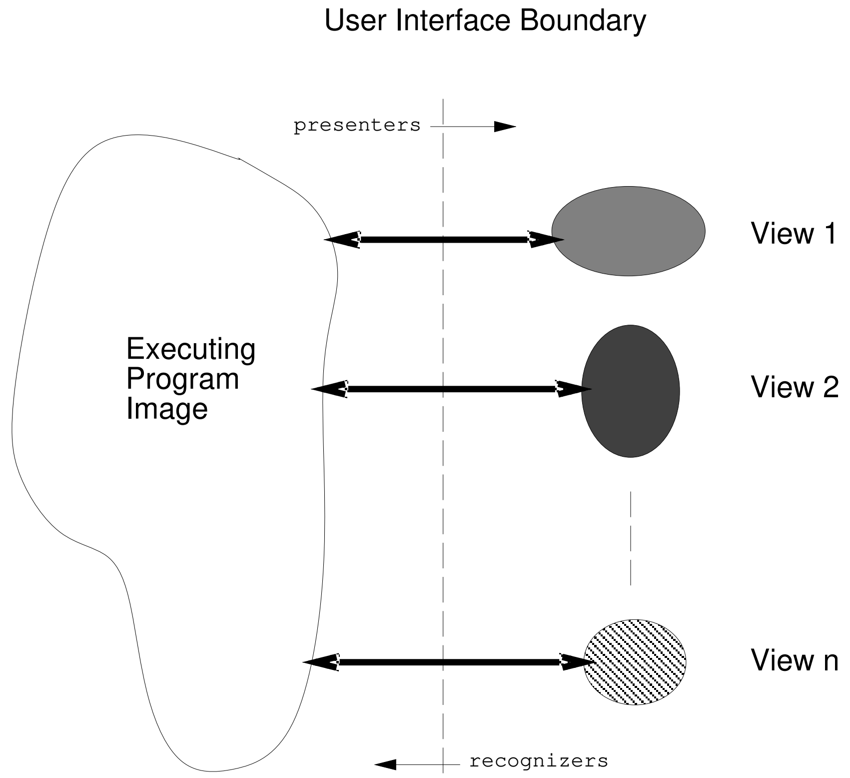

Figure 3.2: Presentation-Style Synchronization Model

3.3.1 A Presentation-Style Model

The synchronization model I chose is a refinement of the *presentation* model of Ciccarelli [4]. In this model, the notion of the state corresponding to the user interface and its objects is collected into *views* that are separated from the underlying program database.

Figure 3.2 shows the synchronization model tying several user interface views to the executing program image they represent. These views represent the flow graphs, control panels, and code fragments (shown with an example in Figure 3.3) that are generated and maintained by PAVES.

Each view is synchronized with the program database by a *presenter* process. Presenters monitor the program database and present to their view any changes that should be reflected across the user interface. To allow a view to affect the program database, a *recognizer* process tracks user manipulations, and converts them into the appropriate program state changes. The act of changing program state then causes the presenter process to update the interface view.

An example of a presentation-style model can be seen when using the Emacs [35] editor. Emacs can display multiple editing buffers at a time, and it is allowable for a single buffer object to be shown more than once. In this situation, if the user types then the result appears in all copies of the single buffer object. Typing causes characters to be appended to the single buffer object. Each change in the buffer object triggers the updating of all screen buffers that reflect it. An indirect model is being used, often with the same observable behavior as direct manipulation, but occasionally revealing more sophisticated behavior.

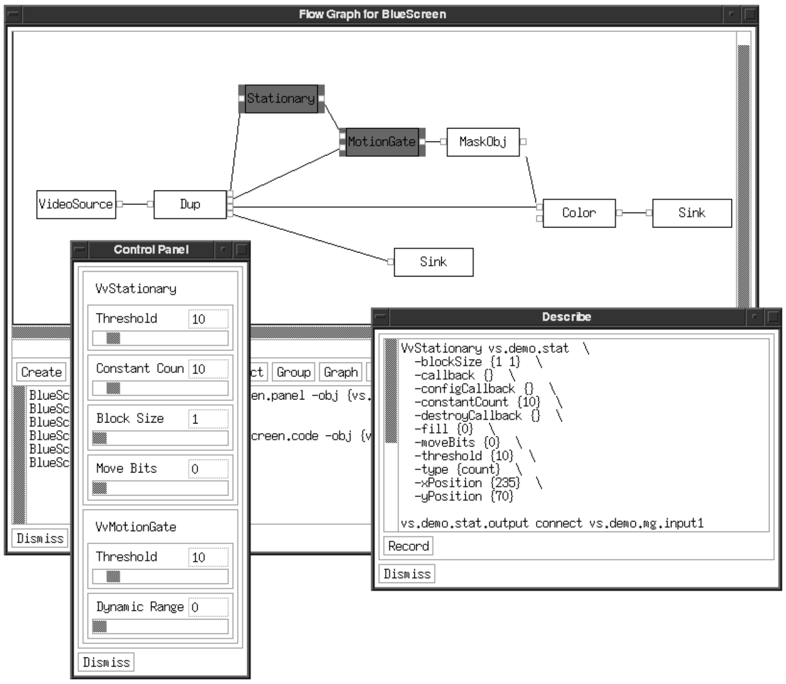

Figure 3.3: Three *views* of a video program generated by PAVES

3.4 Research Scope

This report investigates programming methods. In particular, it investigates the programming language aspects of the visual language and its integration with the underlying interpreted language. To reduce the scope of this effort I have emphasized these areas over other aspects of the system.

I assume that VuSystem video programs can be substantially represented by the media flow graph structure. This kind of representation has been successful in previous efforts, and should capture a useful class of video programs. Other programs, which have a large amount of state not captured by the flow graph representation, cannot be differentiated by the interactive facilities. For them, the value of flow graphs as a means of programming is diminished.

Though the user interface may change in response to changed media flow configurations, I do not study methods for configuring user interfaces in their own right. Modeling video programs by their media flow implies that only the portion of the user interface corresponding to the flow may be controlled with the visual programming interface. With my approach, a system for managing user interface widgets would largely parallel that for managing media processing objects. Much other work, going by the general name of user interface management systems (UIMS) and surveyed in [26, 27], addresses these interface configuration issues.

Finally, when necessary I have traded appearance for functionality. I have adopted an austere interface design, while being careful not to sacrifice ease of use.

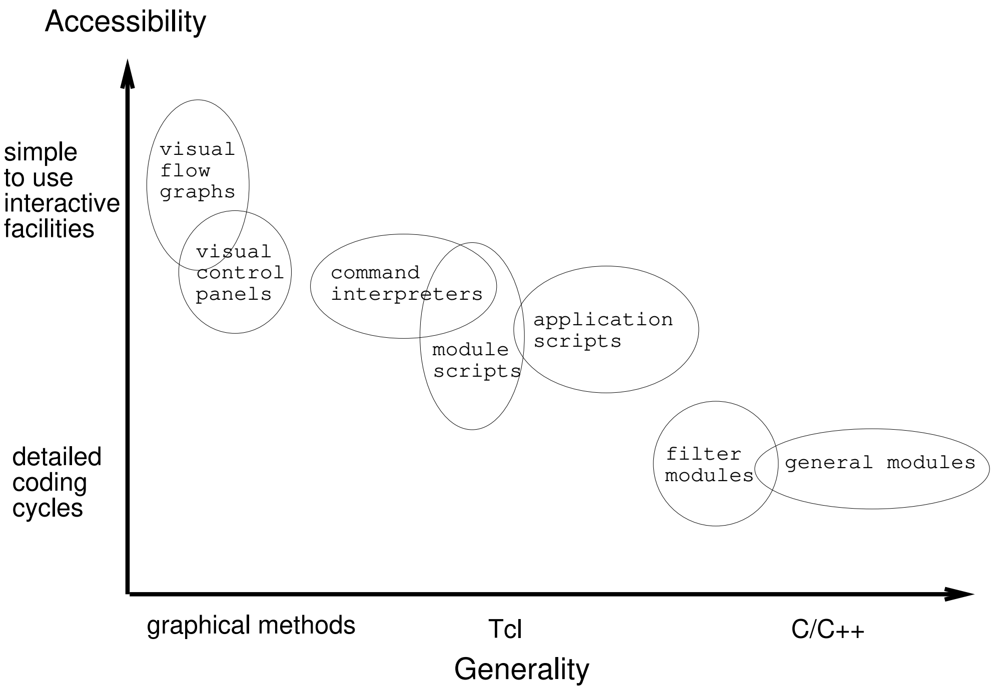

Figure 3.4: The spread of VuSystem Programming Methods

3.5 Perspective

Cooperative programming and the embedding of a user programming facility in an application toolkit are approaches that are not tied to PAVES or media computation. A specialized problem domain lets program structuring conventions that are restrictive yet still useful be defined, and drives the form of the user interface: specialization makes the approach feasible. But the methodology is intended to be general. It should be applicable to other domains whose programs can be well-defined, and for which video programs serve as a testbed.

With this approach, the programmer is presented with a continuum of programming methods instead of a dichotomy. Figure 3.4 shows visual programming as the most accessible though least general method available within the VuSystem. A series of other methods for module and application script tasks lead to programming with the interpreted Tcl language. Below this, primitive modules may be developed in C and C++ when Tcl no longer suffices. At each step a wider range of tasks may be accomplished, but at the cost of learning and using a more detailed programming tool.

Object Tcl allows this continuum to be provided. By exposing it across the user interface, it bridges the gap between the interactive visual levels, and the C/C++ development levels. Object Tcl is tied to the flow graphs and control panels through the Tcl widget set of the VuSystem, which translates between Tcl commands and widget manipulations. Object Tcl is tied to C/C++ through by the tight integration of Tcl and C, which allows routines in either language to be called from the other.

The programming methods are intended to overlap to let more than one method be used for a given task. This encourages experimentation. It provides a gradual learning path from initial user interface programming through the development of new primitive modules. And it means that video programs are extensible, and do not need to be rewritten as features are added.

Chapter 4

User Programming with PAVES

In this chapter, the user interface of PAVES is presented. All programs manipulated with PAVES have several types of windows in common. A sample application is investigated to demonstrate their use to perform a set of user programming tasks. The broader uses of the programming windows are then discussed.

4.1 A Guided Tour

The BlueScreen program is the example video application with which the system is presented. It segments moving objects from a stationary background by assuming a fixed camera position, and is named because of its similarity to the chroma-key technique that is used to combine analog video[1]. The BlueScreen program has several features that make it a suitable example. It is a real application that was developed as part of the COMMA project [36]. The segmentation task it performs is non-trivial, and generalizes from keying by color to keying by motion so that an artificial background is not needed. Yet it is implemented with a small number of more primitive processing modules and is itself used as a subsystem in larger applications.

Figure 4.1 shows the user programmable interface of the BlueScreen program. The upper window, titled “BlueScreen”, displays the input and output video streams, along with overall program control buttons. In this case it shows a moving person

[1]In chroma-keying, a specially colored background is eliminated and substituted by a different background. Saturated blue is traditionally chosen as the special color because it is unlikely to occur in natural scenes.

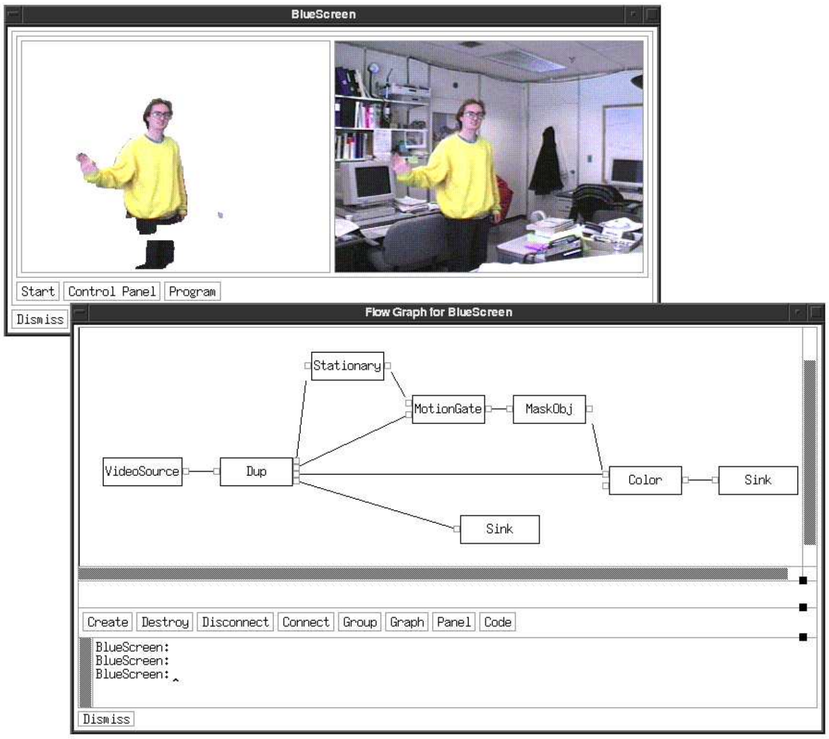

Figure 4.1: The BlueScreen Program with its Flow Graph

segmented from the office scene in the background. The window is named after the program because it presents the main user interface, displaying those widgets that are always present for navigation and media results. This type of window varies in appearance with the application.

As well as a main window, each application presents a varying number of programming windows that are automatically generated by PAVES. These are the flow graphs, interpreters, control panels, code fragments, and other windows that are discussed in the next sections. In terms of the cooperative model, they correspond to different *views* of the interface, text, and internal *forms* of the program.

4.1.1 Media Flow Graphs

The lower window in Figure 4.1, titled "Flow Graph for BlueScreen", is a direct manipulation interface that shows the media flow graph for the program. It was summoned from the main window via the `Program` button. It is an interface form of view, tailored to show the pattern of media processing that the BlueScreen program currently implements.

The flow graph window is the principal means of observing and effecting changes to the overall structure of a program. It shows the program in terms of its VuSystem in-band processing modules. The vertices of the graph are labeled and represent the processing modules themselves, and the edges represent the flow of media between them. Media flows from left to right.

For the BlueScreen application, media originates at a `VideoSource` module, corresponding to live video input. It is then replicated with a `Dup` module. One `Dup` output goes directly to a `Sink` module, to display the unprocessed video on the right-hand side of the main window. The others are processed to achieve the segmentation effect.

Segmentation is computed in four steps. First, a `Stationary` module performs temporal filtering to construct a background image. A `MotionGate` module then computes the raw movement mask by comparing the stationary image with the input video. This estimate is refined with median filtering and region growing techniques, implemented in the `MaskObj` module. It is finally combined with the input image by a `Color` module to pass only the moving foreground for display.

There are several intuitive ways to change the program structure by interacting with the flow graph. Processing modules are objects that can be created, destroyed, combined, and grouped to form new objects. With these operations — a library of primitives, a means of combination, and a means of abstraction — flow graph manipulations form a language and support a method of programming.

All flow graph manipulations work in terms of the current selection. Figure 4.2 shows five selected modules, each of which is indicated by having its background and foreground colors reversed. Ports and modules are selected by clicking the pointer inside their boundary. The selection is extended by shift-clicking, and reset by clicking outside of all object boundaries.

Connections between the ports of modules can be rearranged to alter the combination of media processing. New connections may be made or old ones broken by first selecting input and output ports and then choosing the `Connect` or `Disconnect` button as appropriate.

Existing modules can be destroyed, and new ones created from a library of primitives. To delete a module and all of its ports, it is first selected and then the `Destroy` button is chosen. To create a new module of the same type as an existing module, the existing module is selected and the `Create` button is chosen. If no module is selected, the user is presented with a dialog that lists all the possible types of module that can be constructed.

Modules may be grouped to form a single composite module, adding a new

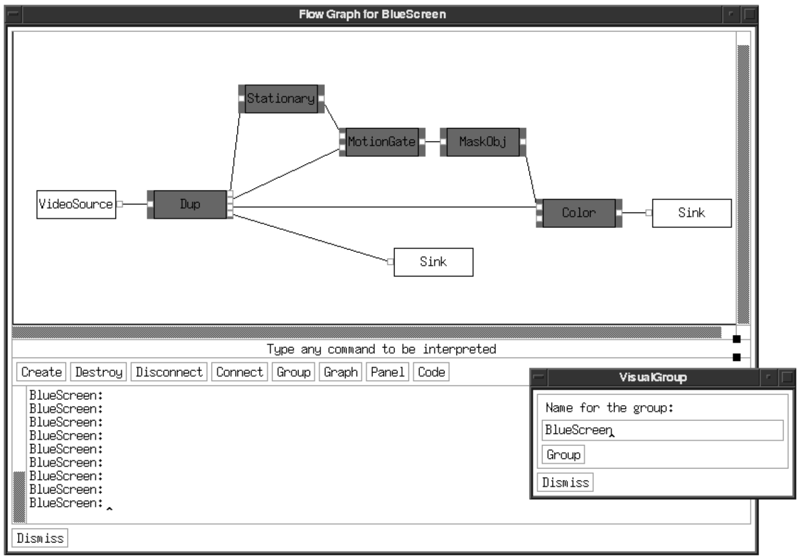

Figure 4.2: Before Grouping the BlueScreen Modules

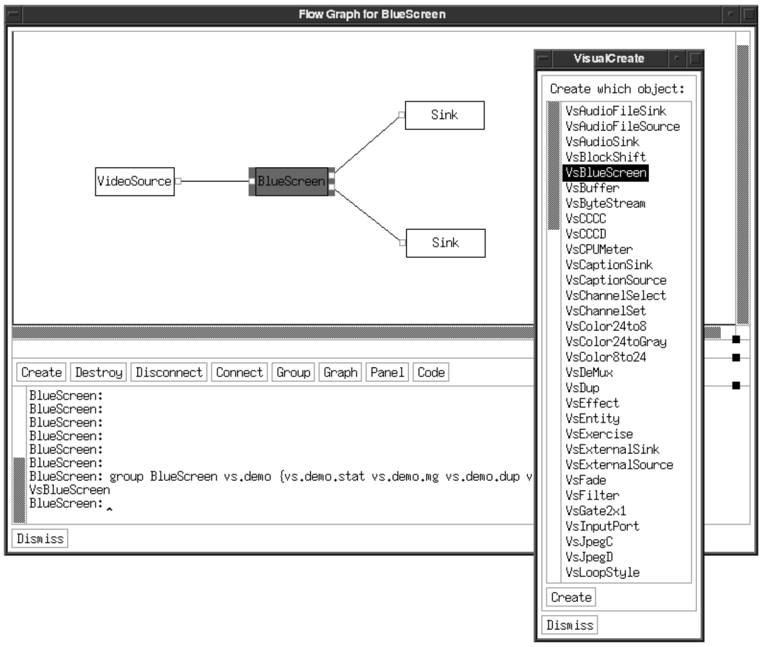

Figure 4.3: After Forming the BlueScreen Group

definition to the module library in the process. This is demonstrated with Figures 4.2 and 4.3, which show the flow graph before and after grouping the modules that comprise the segmentation effect. In Figure 4.2, five modules are selected for grouping and a dialog window is present. The dialog is summoned by choosing the `Group` button, and is used to specify the name for the new grouping prior to its construction. In Figure 4.3, the grouping is complete and the five modules have been replaced by a new module labeled `BlueScreen`. The default creation dialog can also be seen, and now includes the `BlueScreen` module as a candidate since new `BlueScreen` modules can now be manufactured.

All grouped modules are displayed as a single module in the flow graph. Grouping involves creating an encapsulation or abstraction barrier, below which the flow graph allows observation but not editing. To peer inside a grouped module, it is first selected and then the `Graph` button is chosen. This is demonstrated in Figure 4.4, where the internal arrangement of the `BlueScreen` group can be seen. To emphasize the encapsulation barrier, the internal and external flow graphs are drawn in separate windows. The internal window has all of its editing buttons disabled to prevent changes. Observation buttons remain enabled to allow the user to peer inside groups within groups.

Control of the flow graph also flows in the reverse direction, from executing program to user interface. This is because the graph is an interface form of view in the model, requiring it to be synchronized with the program it represents. Programs may create or destroy modules or alter their connections as a normal part of their operation, and these changes will be reflected by the graph. This means that the graph is safe to use over a wide range of applications and in combination with other methods of programming.

Several conventions are silently enforced as the flow graph is manipulated. Modules may be freely repositioned by dragging them with the pointer. This is accompanied by continuous feedback, including the "rubber-banding" of module connections. Repositioning does not alter media processing, but does enable a more aesthetically pleasing layout. To encourage this, positioning information is saved between flow graph sessions. A default layout is also provided for graphs with no positioning information. This makes the flow graph a convenient interface even for completely hand-coded programs (which naturally have no associated layout).

Modules are drawn to show all of their input and output ports, whether connected or not. This provides visual cues, since the shape of the module implies its function. For example, modules with output but no input represent sources of media, such as live video from a camera. Similarly, modules that consume media use the operating system to export it from the VuSystem, perhaps to a screen or storage device. Modules with both inputs and outputs filter and transform media.

Media flows between modules are not visually differentiated and no restrictions are made on the connections between modules. Each flow may contain payloads of any type, and modules process only those types they recognize, passing the rest without error. It is thus allowable, but not useful, to pass video data to an audio processing device.

The flow graph can also be used to summon control panels and code fragments with its `Panel` and `Code` buttons. When invoked, these buttons cause the generation of a control panel or code fragment, respectively, that corresponds to the selected modules. If no module is selected, the windows correspond to the entire program. The operation of these windows is explained in the following sections.

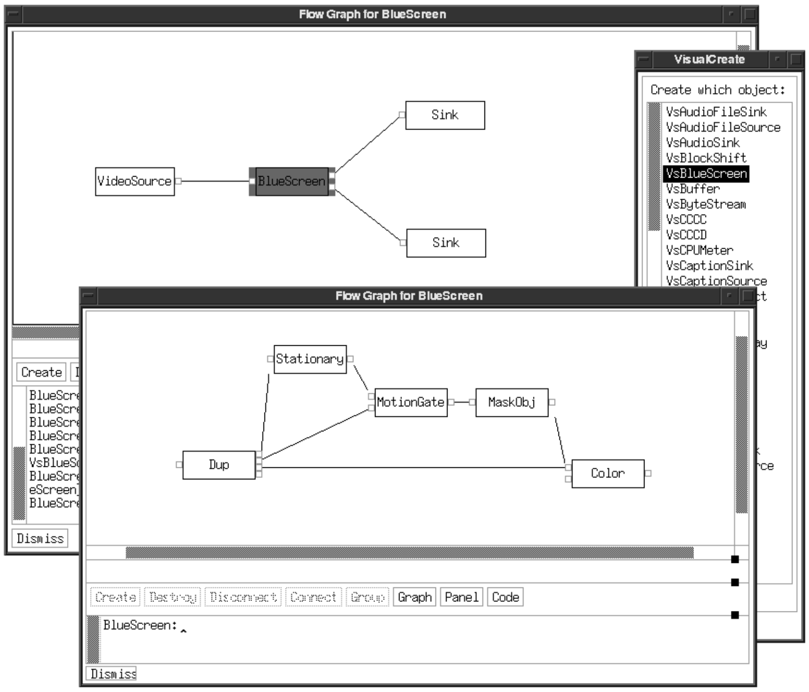

Figure 4.4: Peering Inside the BlueScreen Group

4.1.2 Interpreters

An interpreter window can be seen in the lower section of each flow graph window in Figures 4.1 through 4.4. These provide direct access to the underlying program in its native interpreted language. In terms of the model they correspond to an internal form of view, recording queries and commands at the moment they affect the executing program.

The interpreter functions similarly to a command-line interface, assembling typed input into complete commands and causing them to be evaluated. It is useful for issuing commands that are too complicated or specialized to express with the flow graph, as well as for querying the current state of objects. It also monitors flow graph activity. When a graph operation is completed and executed, its textual equivalent is displayed in the interpreter window as though it had been typed. This mechanism allows either interpreter or flow graph to be used as appropriate, and helps the user form an association between textual and graphical commands.

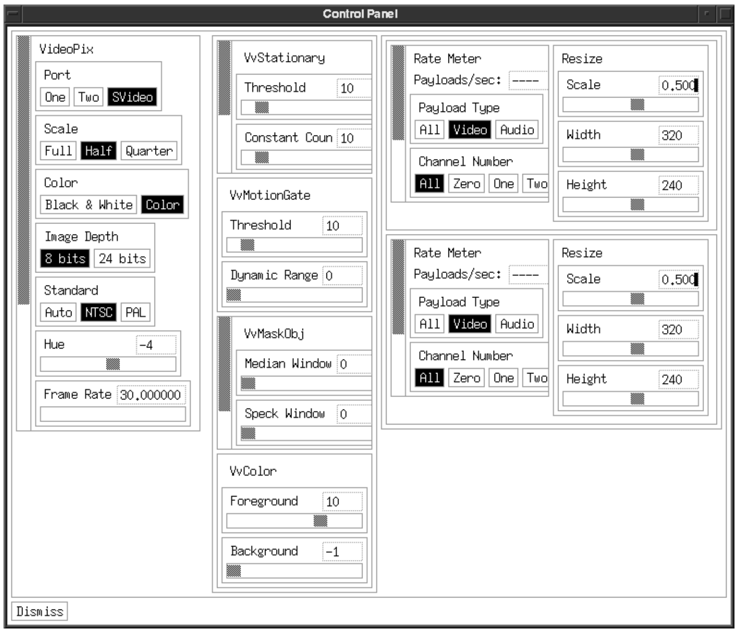

Figure 4.5: Control Panel for the BlueScreen program

4.1.3 Control Panels

The control panel window is another interactive programming view that complements the program window. It is summoned by choosing the `Control Panel` button on the main window, or the `Panel` on the flow graph view. Like the flow graph, it is an interface form of view that dynamically reflects the contents of the application. During the design of PAVES it became apparent that control panels, which already existing in the VuSystem, fit the role of an interface form of view, and they were re-written accordingly[2].

[2]Because they are a VuSystem legacy, control panels are not fully integrated with the model. In particular, they are synchronized with the application only each time they are recreated.

The control panel is the principal means of making small adjustments to a program, tuning or customizing it by using visual feedback. Rather than allowing modules to be rearranged, it displays selected parameters internal to modules and allows them to be altered.

The control panel for the entire Bluescreen program is shown in Figure 4.5. Multiple choice boxes, scrollbars, and other controls are arranged by module for each module of the program. On the left, a column titled "VideoPix" has controls such as port, scale, color, frame rate, etc. for the `VideoSource` module. The right-hand columns have controls for the `Sink` modules that display video on the main window. These include scale and rate information. In the middle column, the `Stationary`, `MotionGate`, `MaskObj`, and `Color` modules can be seen to each have various parameters related to their processing operations. Adjusting these parameters has an immediate effect on the displayed video.

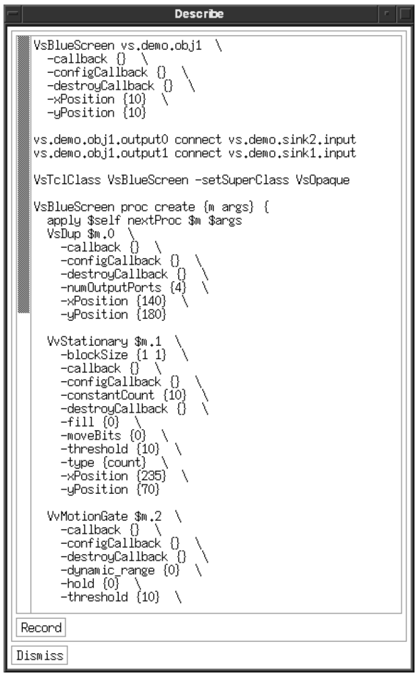

Figure 4.6: Sample Code Fragment Window

4.1.4 Code Fragments

A further type of window is the code fragment window, which reveals the interpreted VuSystem code corresponding to a portion of the video program. It is summoned by choosing the `Code` button on the flow graph view. In terms of the model, it is a text form of view that reflects the state of the program when the view was created.

The code fragment window provides a means of linking the graphical views of the program with the underlying textual representation. It shows a serialization of the application code that is generated from the program as it runs. This code can be used to better understand the function of the program. It can be saved as a snapshot of the program configuration or for later refinement.

Figure 4.6 shows the code fragment window corresponding to the `BlueScreen` module after it has been grouped. Its name and attributes and their values, as well as all connections are pretty-printed. Because it is a user-formed grouping, its module definition was manufactured by PAVES and it is also shown. The fragment can be appended to a working file by choosing the `Record` button. This is valuable for prototyping new abstractions that will later be added to the permanent class library.

4.2 Broader Tasks

The BlueScreen program demonstrates that there are activities the visual facilities support directly that are awkward to accomplish with textual programming alone. Reorganizing the broad structure of an application and prototyping a new construction are two examples. The following paragraphs generalize from the BlueScreen examples to discuss the tasks that PAVES can be used to perform across a range of video programs.

4.2.1 Customization

PAVES is well-suited to customization tasks. Media is always flowing, providing visual feedback to guide the customization process, and the embedded facilities are close at hand for even small and frequent tasks.

Visual results are subject to qualitative tuning, rather than quantitative tuning that may be automated. Some qualities of video that are frequently manipulated by the user include frame size, frame rate, colorspace, contrast, brightness and hue. Audio and other media have a similar range of characteristics. VuSystem modules typically encompass a range of behaviors. For example, the Edge filter module offers a choice of algorithms, each with a range of parameters, and each appropriate in a different context domain.

Preferences can also depend on the available resources. Processing live video is computationally intensive and will remain so for some time. This makes choices, such as compression format, that allow the user to trade the quantity of computation for the quality of result valuable. A VuSystem experiment with collaborative load shedding is investigating the effectiveness of automatic resource customization [5]. Such systems will likely prove useful for improving the default behavior of programs, but they cannot obviate the need for customizations.

4.2.2 Experimentation and Prototyping

With the visual programming windows, an existing program can be readily transformed into a new application. This promotes experimentation because temporary changes in the form of computation can be evaluated quickly. It promotes prototyping because, once found, new and useful combinations of modules can be grouped and saved as code fragments. They can then be refined as needed and added to the module library, making them available to all users and across a range of programs.

Prototyping new abstractions is typically easier at a visual programming level than with textual tools alone. With textual tools, there is much overhead in manufacturing new combinations. It may be automated and hidden at the visual level of manipulation.

Grouping can be used to extend the system as many times as is necessary. Each grouping can be manipulated in the same ways as primitive modules, including being regrouped to construct larger modules. The first-class nature of PAVES groupings is based on the structural equivalent of procedural abstraction, as found in CAD tools, rather than an automatic selection extension mechanism, as found in drawing packages such as MacDraw.

4.2.3 Documentation

Some on-line documentation needs can be well-served by PAVES. By presenting an executive summary of the structure of a program as a diagram, the operation of the program is revealed as a configuration of smaller modules. This is the manner in which the behavior of the BlueScreen program was deduced. By showing the available customizations of each module in a standard graphical form, its range of behaviors is described. These descriptions are always available, always up-to-date, and have a consistent style.

4.2.4 Program Development

Ultimately, the preceding tasks fall under the broader banner of program development. PAVES promotes the manipulation and evolution of programs by their users. As an extensible system, it is not constrained to specific tasks such as minor customizations. It may grow and become more useful as the library of modules and programs expands.

To foster general program development, the visual interface is integrated with the other textual programming levels. This gives the user the flexibility to chose the most appropriate programming level. Work may then be performed at the best level of detail for the task at hand: the visual level for broad structural changes, and the textual levels for specific extensions within modules or the development of new modules. The system encourages learning by translating between these languages.

These capabilities make PAVES attractive to sophisticated users. They can apply their knowledge of a problem domain to specific tasks, such as enhancing the features of an ultrasound scan to assist in diagnosis, without first learning a new programming language. Yet PAVES is an extensible tool with an evolution path for progressively learning and applying programming techniques.

Chapter 5

Design of PAVES

The previous chapter presented PAVES from the perspective of the user, to whom it appeared as a series of windows with different capabilities. This chapter investigates the mechanisms that enable those windows to function. The discussion focuses on the visual programming window because it is the most well-developed of all views, having the greatest synchronization and presentation needs. The supporting control panel, code fragment, and interpreter windows are then discussed. The final section of the chapter describes the underlying structuring conventions, which allow the interactive facilities to work across a range of video programs.

5.1 Flow Graph Windows

The design of the visual programming window is split between a visual language and manipulations on video programs. The language provides a representation of programs, while the manipulations provide a means of linking the flow graphs to the executing program so that one may affect the other.

5.1.1 Visual Language

Recall from the previous chapter that most features of the visual language are evident from the flow graph window because of its austere style and few hidden semantics. Though they require little explanation to use, these features represent design choices. They are discussed in the following paragraphs in terms of language primitives, means of combination, and means of abstraction.

Primitives

The primitives of the visual language are the media processing modules. All primitives are drawn as rectangles and labeled with their type to indicate their function. Icons are an alternative to a text label, but are not implemented in the current version of PAVES because they are essentially syntactic sugar. The ports of modules are always shown, whether connected or not. Input ports are drawn to the left and output ports to the right. This puts a practical limit of approximately five input and five output ports on each module, though there has been no need to exceed this to date.

This scheme means that objects of the same type are visually indistinguishable, implying that they are functionally interchangeable. Drawing the ports visually hints at the role of the object. For example, source modules appear more similar to each other than to sink modules, because sources always have some ports to the right and none to the left, while sinks are reversed.

The primitives are freely moveable within the workspace so that the user may customize the program layout. They are selectable too, with operations specified in an object fashion by first picking the primitives and then indicating the operation.

Combinations

Combinations in the visual language are connections between the ports of modules. They represent the flow of media, and are drawn as lines. Unlike AVS, the flows are visually undifferentiated because VuSystem payloads are self-typing. This implies that all connections between a pair of input and output ports are legal, though not necessarily useful.

The rendering of combination is closely related to the module layout. The shape of primitives and their method of combination is such that media prefers to flow from left to right. This gives the user a direction of reference. PAVES includes an algorithm that organizes flow according to this model. When presented for the first time, a flow graph may have no associated layout information because the video program was constructed textually. Automatic layout is applied in this special case.

As a matter of convenience, more general layout tools would be useful to assist the user in tidying layouts after they have been manipulated. Despite much research on diagram layout (surveyed in [3]) it is difficult to find algorithms that reliably produce good results, are computationally simple, and are stable across user interactions.

Abstractions

An abstraction in the visual language is a set of interconnected modules that function as a single larger module. The visual language admits new abstractions by grouping a set of primitives and/or existing abstractions. New modules of the same type as the new abstraction can then be created. This usage appears straightforward to the user, but involves a series of manipulations reminiscent of programming by example techniques.

When grouped, a set of modules is used to form the definition of a new kind of object. The set is then re-organized into an instance of this new type. Forming the definition requires generalizing from the modules to be grouped to an abstract recipe. It requires specifics to be distinguished from generalities.

The problem of generalizing is solved by the notion of *options*. These are supported by the structuring conventions described later in the chapter. The attributes of modules that are marked as options contain the instance-specific or optional state. It can safely be ignored when forming the general definition. This scheme passes the burden of classifying state to the module developer, rather than the video program user.

Flow graphs respect the encapsulation barrier of abstractions, which are drawn in the same manner as primitives. The two are not visually differentiated because abstractions are first-class: they can be manipulated in the same fashion as primitives. PAVES does provide a means of peering inside abstractions, but places the view in a separate window to reinforce the existence of an encapsulation barrier. Abstractions may be also formed textually, outside the control of the graphical system, and graphically prototyped abstractions may later be refined textually. The visual system cannot distinguish these cases, however, and so it cannot safely break the abstraction barrier to allow editing within a grouping.

Implementation

The screen representation of the visual language is drawn with two Xt constraint widgets developed in support of the research presented in this report. Figure 5.1 shows how the `Graph` and `WorkSpace` widgets, along with other widgets in the Athena widget set (Xaw), relate to views in PAVES. Widgets control the presentation and editing of the flow graph by translating between the X and Object Tcl domains so that PAVES does not need to deal with window events directly. To respond to user manipulations, X *events* are translated into Xt *actions* and their equivalent Object Tcl *callbacks*. To display program feedback, Object Tcl *commands* and their equivalent Xt *resources* alter widget state and trigger a refresh of the display. The Object Tcl commands and callbacks correspond to the

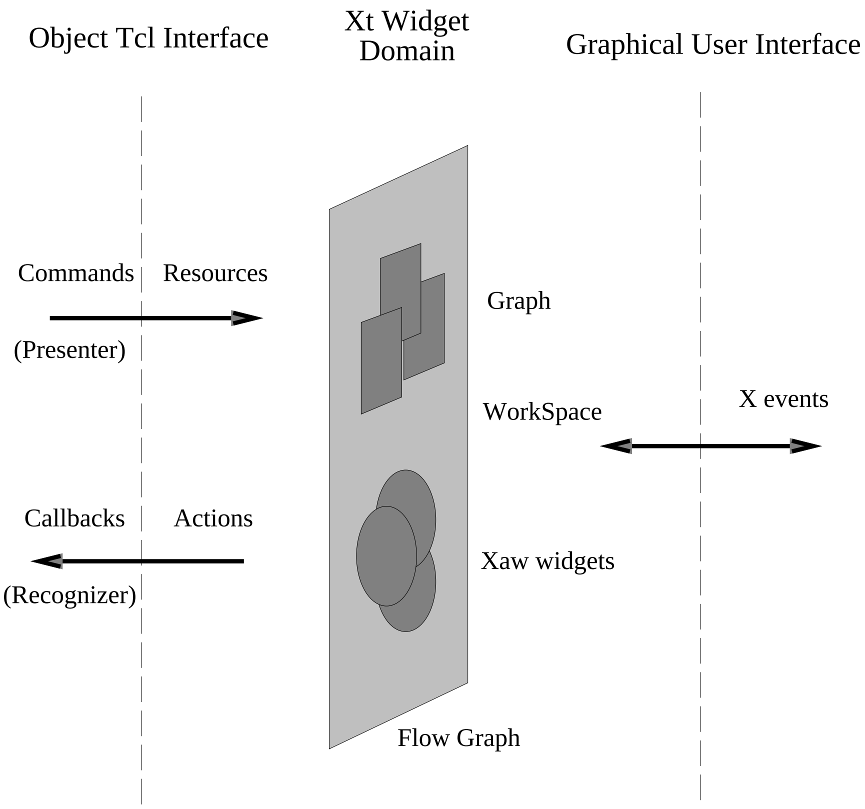

Figure 5.1: Relationship of Xt Widgets to PAVES

presenter and recognizer activities of the synchronization model.

The `WorkSpace` widget acts as a container for the `Graph` widgets. It maintains the current value of the `selection` resource for the graph, along with the translations and actions that affect the selection. The workspace also maintains translations and actions for moving its `Graph` widgets, allowing them to be arranged into a more aesthetically pleasing layout.

Auxiliary buttons are used to combine partial commands with the selection to form a complete command, which is then issued. For example, destroying media objects is accomplished by first selecting them and then pressing the `Destroy` button. The widget corresponding to the button translates the X clicking event into a Tcl callback that retrieves the value of the selection, uses it to form a complete command, and finally issues it to destroy the module.

The `Graph` widgets act as containers for a variable number of widgets that represent ports. Each port has a `gravity` and a `join` resource, which determine where they appear in relation to the graph node and what they are connected to, respectively.

The `Graph` widget redraws itself when moved, creating a "rubber-banding" effect as the connections between ports are stretched. If no position information is set for the overall graph, the `Graph` widget computes a basic layout. A topological sort algorithm applied to the connections between ports makes media flow from left to right. No line crossing elimination is performed.

5.1.2 Manipulating Flow Graphs

The two major aspects of manipulating flow graphs are generating the initial graph of the executing program, and maintaining it in synchrony with program execution.

Generating the Initial Flow Graph

Each module in the video program is a specialized instance of the `VsEntity` object, from which the `draw` method is inherited. `Draw` walks the internal form of the program and generates the set of interface widgets that is the flow graph view. In performing this walk, `draw` relies on the structuring conventions adopted by all PAVES programs. These conventions are described later in the chapter.

A skeleton version of the `draw` method is shown in Figure 5.2. To produce a flow graph, `draw` is called for a portion of the program. The overall flow graph is generated by calling it on the program root, while the flow graph for an abstraction is generated by calling it a grouped module directly. The argument `w` represents the widget in which to construct the graph. It is specified by wrapper code that performs initializations, which include creating the top `WorkSpace` widget.

`Draw` walks the program structure in two passes. In the first pass, `dNodes` is called recursively to create all vertices of the graph. For each media processing module it creates a `Graph` widget, with a `Label` widget for each input and output port. Vertices contain a text label, and their widget names are mapped from the names of the media entities they represent. Input ports are positioned on the west border of the node, and output ports on the east. The vertices are also positioned.

In the second pass, `dEdges` is called recursively to completes the graph with edges. It sets the `join` constraint resource on ports to point to each other. This completes the media flow graph with edges and the `draw` method returns. The wrapper code that called it then *realizes* the widgets, causing them to render themselves on the screen. During this process, the `Graph` widgets are responsible for drawing the media flows and use the flow information to create a default layout if all positioning information is null.

```
VsEntity instanceProc dNodes {w} {
  set lvl $w_[tail $self "."]
  if {[$self children] == {}} then {
    Graph $lvl \
      -label [$self class] \
      -xPosition [$self xPosition] \
      -yPosition [$self yPosition]
    foreach i [$self inputs] {
      Label $lvl.[tail $i "."] -gravity west
    }
    foreach i [$self outputs] {
      Label $lvl.[tail $i "."] -gravity east
    }
  } else {
    foreach i [$self children] {$i dNodes $lvl}
  }
}

VsEntity instanceProc dEdges {} {
  if {[$self children] == {}} then {
    foreach i [$self inputs] {
      if {[$i bind] != {}} {
        [vx $i] setValues join [vx [$i bind]]
        [vx [$i bind]] setValues join [vx $i]
      }
    }
  } else {
    foreach i [$self children] { $i dEdges }
  }
}

VsEntity instanceProc draw {w} {
  $self dNodes $w
  $self dEdges
}
```

Figure 5.2: Skeleton of the `draw` method

Synchronizing Flow Graphs

Once the flow graph has been generated, it must be kept consistent with the program it represents as it is both used and the program continues to run. Recall from Chapter 3 that this is achieved with a presentation-style model. It allows for changing the program by manipulating the graph, as well as for updating the graph in response to program state changes.

Figure 5.3 shows the presentation-model in more detail. A view, representing one flow graph window, is linked to the executing program by recognizer and presenter activities. The recognizer translates edit actions on the flow graph into program commands. The presenter translates in the reverse direction.

Both recognizer and recognizer pass their messages via evaluators. These represent the means for causing the translated commands to take effect. The default

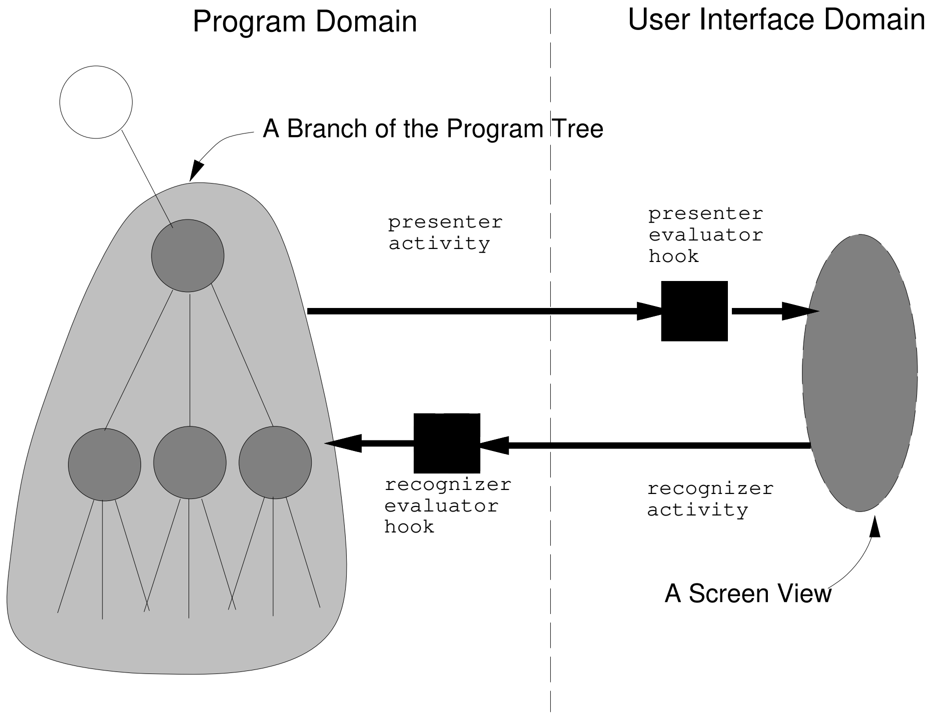

Figure 5.3: Model of each View

evaluator is simply the Tcl `eval` command, corresponding to executing the command. Other evaluator mechanisms are useful for monitoring traffic along these pathways.

Implementing the recognizer activity is accomplished by using the event translation functionality of Xt widgets. Instead of using `eval` directly, translated commands are evaluated by passing them to the interpreter object that sits below the flow graph. This displays the commands in the interpreter window as they are evaluated.

Implementing the presenter activity is more difficult because feedback from each recognizer to all presenters must be generated. Many ad hoc schemes can be invented to cater to particular situations, but they will work at the cost of restricting flexibility.

PAVES implements the presenter activity by specializing the behavior of operations that affect the flow graph. As well as performing their function, the operations check their outcome. If appropriate, they package the change and distribute it amongst the presenters. This is accomplished with the `evalConfigCallback` function, which uses the `configCallback` attribute registered on modules to contact the appropriate presenters.

Presenters are responsible for compressing multiple change messages into a single flow graph edit. For example, destroying a media processing module causes all of its ports to be destroyed too. Multiple messages are generated, yet only one screen widget (corresponding to the entire module) should be destroyed.

5.2 Supporting Windows

The control panel, code fragment, and interpreter windows complement the flow graph window. This section discusses their design.

5.2.1 Control Panels

Control panels are constructed with the `panel` method[1]. It forms a composite panel of controls that reflect the internal structure of the video program.

Panel elements are grouped into rectangles to show the functional hierarchy of the program; boxes within boxes is an alternative form of tree representation that maps well to a windowed display. The different levels of grouping are emphasized with alternating horizontal and vertical tiling of the rectangles. This scheme conveys considerable information without being visually intrusive.

A skeleton version of the `panel` method is shown in Figure 5.4. It is called recursively to combine the contributions of all modules. The alternate horizontal and vertical tiling is indicated by the orientation variables and supported by the layout functionality of the Xaw toolkit.

`Panel` relies on the individual media processing modules to define their selected controls. By convention, the controls correspond to options that may take on a set of values but not otherwise affect the structure of the program. They are represented as pushbuttons, scrollbars, and other combinations of standard user interface widgets.

Two special cases are handled when composing the panel. First, not all media processing modules need a user interface for controls. This is accommodated by pruning empty panels as they are found. Second, default controls for abstractions should be computed without requiring user input. They are automatically generated as the sum of their internal controls, labeled with the name of the abstraction.

[1] Chris Lindblad developed and implemented the control panel organization before it was incorporated into the visual programming system.

```
VsEntity instanceProc panel {w orient args} {
  set ch [keyarg -children $args \
          [$self children]]
  set args [keyargs -children $args exclude]
  apply Form $w $args
  set args [keyargs {-fromVert -fromHoriz} \
            $args exclude]
  if {$orient == "-fromVert"} then {
    set o "-fromHoriz"
  } else {
    set o "-fromVert"
  }
  set last {}
  foreach i $ch {
    set j [tail $i "."]
    if {[apply $i panel \
         $w.panel$j $o $args] != {}} then {
      if {$last != {}} then {
        $w.panel$j setValues $orient $last
      }
      set last $w.panel$i
    }
  }
  if {$last == {}} then {
    $w setValues -width 10 -height 10
    $w destroy
    return {}
  }
  return $w
}
```

Figure 5.4: Skeleton of the `panel` method

5.2.2 Code Fragments

Code fragment windows are produced with the `describe` method. It attempts to serialize a portion of the program into code that is useful across application sessions. The most expedient format for this purpose is Tcl code itself. It may be saved to be later evaluated by a VuSystem program, and in a proper environment, to re-create the portion of the program it describes. It may be cut from the code fragment window and pasted into another window, such as the interpreter.

A skeleton version of `describe` is shown in Figure 5.5. It walks the program tree recursively, in a similar manner as the `draw` and `panel` methods. It accumulates two sets of results: the media objects themselves with their current settings, and the connections between objects.

As was the case for abstractions, the notion of *options* is used to classify state. Options determine which module attributes should be retrieved and which are irrelevant outside of the current application session. The attribute values are retrieved by calling the option methods with no arguments. This uses the

```
VsEntity instanceProc describe {args} {
  foreach i [lsort [$self info options]] {
    append olst "  -$i "
    append olst "[list [$self $i]]  \\\n"
  }
  set olst "[$self class] $self  \\\n$olst"
  set olst "[string trim $olst "\\ \n"]\n"

  foreach i [lsort [$self outputs]] {
    if {[$i connect] != {}} then {
      append ops "$i connect [$i connect]\n"
    }
  }

  foreach i [lsort [$self children]] {
    set dCh [$i describe]
    append olst "\n[lindex $dCh 0]"
    append ops "\n[lindex $dCh 1]"
  }
  set ops [string trimleft $ops "\n"]
  return [list $olst $ops]
}
```

Figure 5.5: Skeleton of the `describe` method

"set-and-get" convention honored by all media processing objects.

When producing the code, several steps are taken to ensure execution safety and to enhance readability. The values of options are protected with the `list` command. This escapes special character sequences so that the output will be correctly interpreted when it is re-read as input. Modules definitions always precede their connections so that the code may be linearly executed. Lists of options and their values are pretty-printed, one option per line in alphabetical order. Modules, inputs, and outputs are also listed in alphabetical order.

Special modules use their own `describe` method. For example, the generic `VideoSource` object is realized as a different set of objects (a hardware peripheral, a file on disk, or a network connection) depending on the run-time environment. Its version of `describe` accounts for this variation. In this manner, objects that are more than the sum of their parts are wrapped so that they can be safely used with the interactive programming facilities.

5.2.3 Interpreters

Interpreter objects provide direct access to the VuSystem language while the video program executes. Their design addresses three issues.

They need to simulate a command-line interface in the midst of an X windows application. This is accomplished by configuring and combining standard user interface widgets. They need to assemble fragments of text into complete commands. This is achieved with the Tcl library routine `Tcl_CommandComplete`. They need to select an execution context and operate within it. For simplicity, the top-level environment of the current interpreter is used for evaluation. Other, more flexible, execution contexts are useful for debugging.

The interpreter window serves two purposes. As well as accepting typed commands, it monitors the flow graph. By acting as the recognizer evaluator hook for the flow graph view (see Figure 5.3), it displays the textual equivalent of flow graph operations, treating them as though they were typed. This provides a record of activity that may be cut and pasted into another X application.

5.3 Structuring Conventions

The interactive facilities of the previous sections are dependent on program structuring conventions. These govern how video programs may be expressed. They reuse and extend the original VuSystem model [22], contributing to flow graphs, control panels, and code fragment in two important ways.

VuSystem conventions support a model of the video program in terms of objects and their relationships. Manipulations defined in terms of this a model are safe for all conforming video programs. This means the problem of producing many flow graphs for many applications is reduced to that of producing flow graphs for the model program.

The conventions provide video programs with functionality through the inheritance mechanisms of object-oriented programming. Base objects, upon which all programs build, provide default implementations of the flow graph and other functionality. This ensures that some flow graph representation can be provided for all programs. More sophisticated programs can improve on this default to cater for special needs. In this manner, the similarities and differences between programs are accommodated. Chapter 6 describes Object Tcl, the language created for this purpose.

5.3.1 Programs as Entity Trees

The main convention for structuring a video program is the arrangement of its media objects into a tree with an embedded directed graph. Media objects comprise the essential portion of a video program by describing the pattern of processing that it implements. General information about the types of processing is nominally considered to belong to the system libraries.

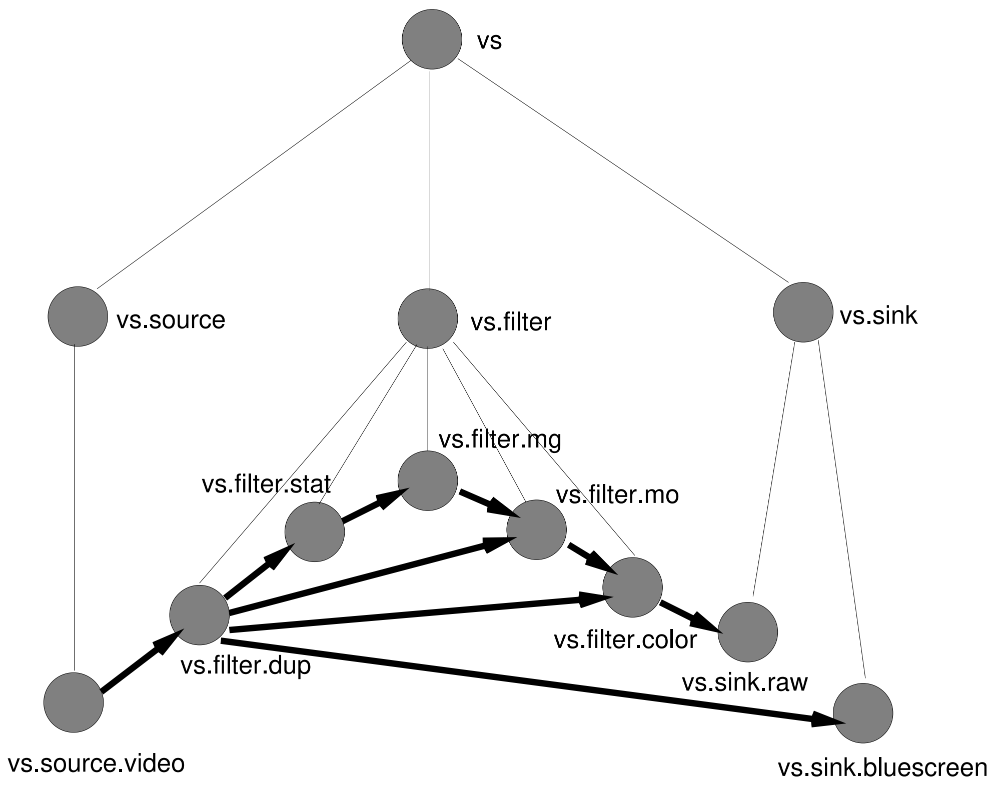

Figure 5.6: Program Structure

The tree for the BlueScreen program, the example used to tour the user interface in Chapter 4, is shown in Figure 5.6.

The branches of the tree convey a functional decomposition. The root of the tree corresponds to the entire program. For the BlueScreen program, it is split into three branches: a source, a filter, and a sink. The source branch has a single leaf, corresponding to the module that generates video. The sink has two leaves, one for each of the video display windows. The filter branch contains all the remaining processing modules, which achieve the segmentation effect. Each branch of this example has the same depth, but this is not required.

The leaves of the tree represent the media processing operations. They are connected in a directed graph that conveys the flow of media as it is processed by the program. In this case, video originates in the source branch and travels to the sink branch for display. The filter branch splits its input into two output flows. One is unprocessed, the other is passed through four leaves. These four leaves combine the unprocessed video with processed video, finally producing a flow that contains only the moving foreground.

When used with a sensible naming scheme, much of the meaning of the program can be inferred from these two structures.

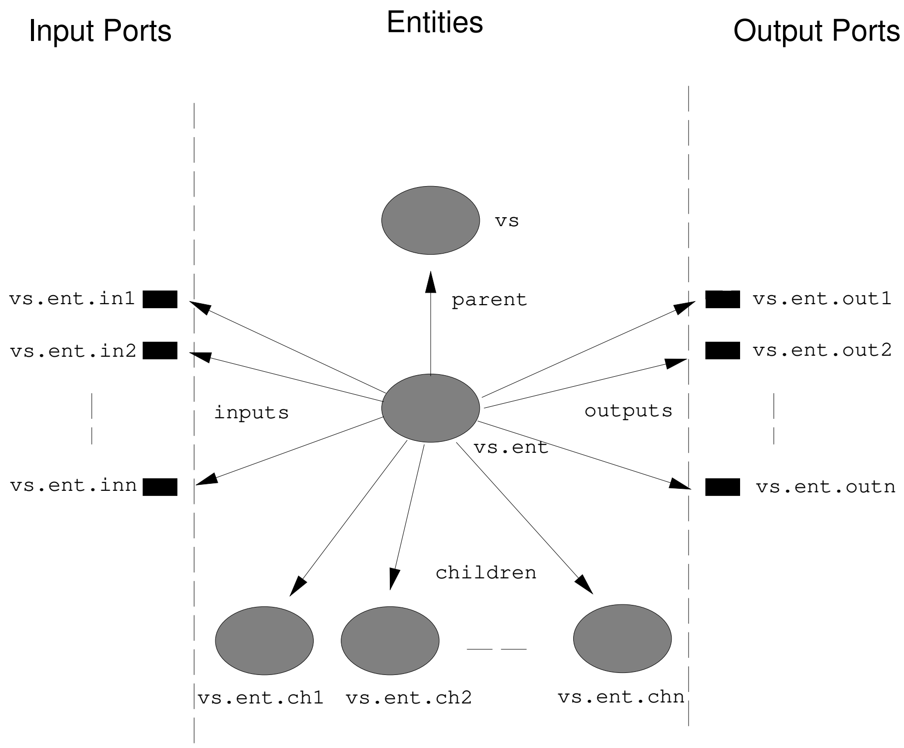

Figure 5.7: Model of an Entity

5.3.2 Media Processing Entities

All media objects are based on a foundation object that is specialized to implement different processing functions. The `VsEntity` foundation class is provided by the toolkit to define the default functionality of all program objects.

The model of an entity is shown in Figure 5.7. Its input and output media ports are modeled by separate objects. General entities have both ports and children, but specific entities need only ports or children. The root of the program tree is an instance of `VsEntity`. When used as branch nodes, entities signify transparent containment in the program tree. When used as leaves, entities typically have a fixed number of input and output ports and implement a specific media processing operation. The objects of classes `MotionGate`, `Color` and `MaskObj` were used as leaves in the BlueScreen program.

name	description
`option`	define option method
`xPosition`	horizontal position
`yPosition`	vertical position
`reparent`	change parent
`configCallback`	presenter notification callback
`evalConfigCallback`	notify presenter of change
`draw`	make flow graph
`panel`	make control panel
`describe`	generate source code

Table 5.1: Entity Methods Supporting Views

Entities include infrastructure for generating flow graphs, control panels, and code fragments. These methods have been discussed in the previous sections and are summarized in Table 5.1.

name	description
`inputs`	list input ports
`outputs`	list output ports
`children`	list child entities
`parent`	get parent entity
`info options`	list option methods

Table 5.2: Reflective Entity Methods

name	description	type
`bind`	get connected port	*input port*
`connect`	get bound port	*output port*
`entity`	get owning entity	*input/output port*

Table 5.3: Reflective Port Methods

Entities also include reflective or introspective methods that provide access to the structure of the program as it runs. These are necessary because the program may be extended or reconfigured while it is used, and the flow graphs and other windows should change accordingly. The reflective methods for entities are summarized in Table 5.2. Those for input and output ports are given in Table 5.3.

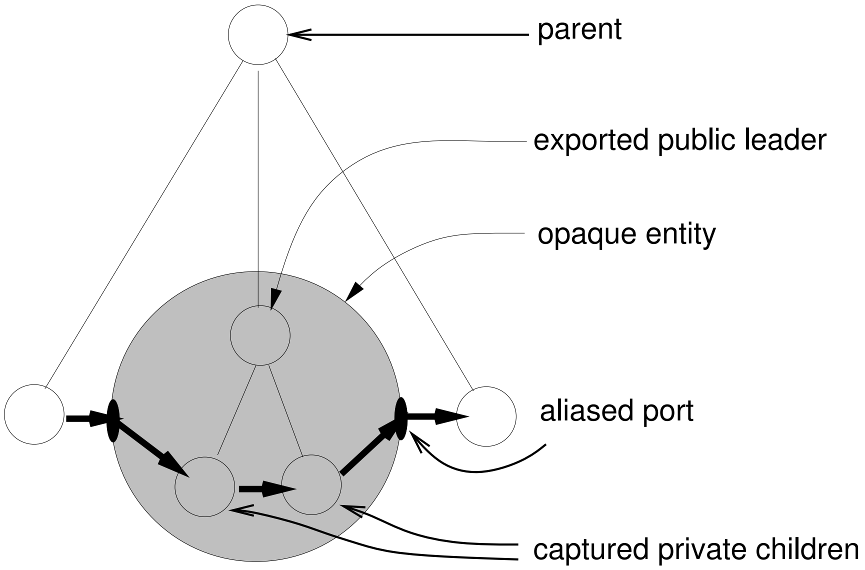

Figure 5.8: Opaque Grouping of Entities

5.3.3 Media Processing Collections

A second foundation class extends the model of entities to handle opaque collections of media processing objects. This allows patterns of media processing to be captured in a reusable way. It is the basis for forming abstractions with the visual language.

`VsOpaque` builds on `VsEntity` to enable a collection of entities to be grouped and treated as a single larger entity. The model for this is shown in Figure 5.8. One entity acts as a public leader and represents its children, which remain hidden from the view of the interface windows. Media ports on the children that are available for use are aliased to appear as though they belong to the collection.

The `BlueScreen` module, grouped in Chapter 4, is an opaque collection. It acts as a single module, but is internally composed of all the modules that implement the BlueScreen effect.

name	description	type
`capture`	hide private entities	*addition*
`advertise`	make public interface	*addition*
`panel`	make control panel	*override*

Table 5.4: Collection Methods Supporting Views

To correctly represent flow graphs and other windows, `VsOpaque` defines two new methods and redefines an existing one. A `capture` method is used to specify the children to be hidden by the abstraction. Two children are hidden in Figure 5.8. They are removed from the list of publically available children and placed on the list of private children. An `advertise` method is used to manufacture a public interface on the leader. For each method on a hidden child that is to be publically available, `advertise` places a code stub on the public leader. When invoked, the stub redirects the call to the hidden child. The `panel` method is changed so that the hidden children may contribute to a default interface. Table 5.4 summarizes these methods.

name	description	type
`inputs`	list input ports	*override*
`outputs`	list output ports	*override*
`children`	list child entities	*override*
`info options`	list option methods	*override*

Table 5.5: Reflective Collection Methods

Collections support reflective methods as entities so that programs may be examined dynamically. This requires several of the methods to be redefined so that they do not report hidden children. Table 5.5 summarizes these methods.

Chapter 6

Object Tcl

This chapter describes Object Tcl, the object system constructed in support of the research presented in this report. Previous chapters have described the architecture and user interface of the interactive video programming system, assuming the existence of an appropriate object foundation on which to build. Object Tcl is that foundation.

The chapter presents Object Tcl as a general-purpose object system, concentrating on motivations and the ensuing language design and implementation issues. Its development is traced, from the starting point of Tcl through the introduction of objects and their inheritance, class, and introspection mechanisms. Object syntax and semantics are exposed along the way. The final section discusses the role of Object Tcl in PAVES. Appendix B contains a reference guide to the language.

6.1 Tcl as a Starting Point

The Tool Command Language (Tcl) [32] is a simple yet extensible interpreted language. Its easy syntax and use of strings plus its interpreted and dynamic nature make for rapid prototyping. Its compatibility with C and lightweight implementation mean it can be embedded in extensible applications.

Object Tcl is an extension to the Tool Command Language (Tcl) for the management of complicated data types and dynamic object-oriented programming in general. It arose out of the prior *object command* mechanism employed to organize VuSystem modules. Its design was driven by the needs of the visual programming system to create a foundation with powerful abstraction and introspection capabilities. It retains both the spirit and benefits of Tcl, and at 2000 lines of C/C++ it forms a small extension to the 25000 lines of Tcl.

An introduction of objects to Tcl is effective both as a means of managing complicated data and for providing a powerful abstraction mechanism. Objects provide a natural way to group related operations and data, such as video processing modules and their parameters. They complement and extend the built-in Tcl structures of lists, arrays, procedures, and commands. And with a C interface they are especially useful for data that is not readily converted to and from strings.

The result is a compact, dynamic, and introspective language, more in the spirit of CLOS [20] (the object system that extends common lisp) than static models such as C++ [10]. This distinguishes it from [incr Tcl] [25], an alternative Tcl-based object system that aims to supplement Tcl with a structured programming environment. The Object Tcl approach shares goals with languages such as Dylan [13], which attempt to be practical on small machines while providing many of the language features found in CLOS and other advanced object systems.

6.2 Introducing Objects

In Tcl, all commands and values are represented as strings. It is easy to pass data between the Tcl interpreter and commands written in C code in an application: the C code need only be able to convert the internal representations to and from strings.

When data too complex to be readily converted to and from strings must be managed, the implementation may be pushed further into the command written in C. With an *object command* approach, each object of complicated data is registered with the interpreter as a single command. Operations on an object are performed with *subcommands* by using the first argument to the command to specify the operation. The Tk [33] graphical user interface toolkit uses this approach to manage widgets.

6.2.1 Objects

Object Tcl formalizes the notion of grouping related data so that object models may be realized without replicating the infrastructure that would otherwise be necessary. For example, objects include a dispatch mechanism for finding and invoking the method implementation that matches the method name. Without using this infrastructure, each object would need to implement its own dispatch mechanism. Objects support the same flexibility in combining Tcl and C as the Tcl language itself. Methods may be implemented in C or Tcl as appropriate, leading to objects with a mixture of implementations.

```
Object create astack

astack set things {}

astack proc put {thing} {
  set things [apply list $thing $things]
  return $thing
}

astack proc get {} {
  set top [lindex $things 0]
  set things [lreplace $things 0 0]
  return $top
}

astack put toast    ==> toast
astack get          ==> toast
astack destroy
```

Figure 6.1: Making a stack object by specializing a generic object.

Each object in Object Tcl is a single command registered with the interpreter. This is a straightforward way of combining objects with the existing Tcl interpreter, though like C++ it restricts operation dispatch to depend on the type of a single object; a more general dispatch on the types of multiple objects is found in the CLOS multi-methods model.

Objects are created by executing the command that corresponds to their type or class and passing the name of the new object as an argument. An example is shown in Figure 6.1, where the object `astack` is created as an instance of the generic class `Object`. With one instance variable and two methods it implements a stack.

Objects are deallocated and release their storage by invoking a consistently named method, the `destroy` method. The `astack` object destroys itself at the end of the example. `Destroy` uses the polymorphism provided by the naming of the object system. This is in contrast to the C++ style `delete` operator and the Tk `destroy` command that take the object as an argument and then reverse the situation, eventually calling the destructor for the specific object.

6.2.2 Methods

Methods are the operations that are executed on behalf of a particular object. Object Tcl methods are added with the `proc` method that mirrors the behavior of the Tcl `proc` command. This is shown in Figure 6.1, where the `put` and `get` methods are added to the stack object. The methods that Object Tcl defines for all objects are summarized in Table 6.1. As well as Tcl methods, C methods can

name	description
`class`	set class
`name`	get the primary name
`names`	get all names
`alias`	make secondary name
`destroy`	destroy object
`proc`	define Tcl method
`set`	define Tcl variable
`auto`	demand load Tcl method
`info procs`	list Tcl methods
`info commands`	list Tcl/C methods
`info vars`	list Tcl variables
`info class`	list classes

Table 6.1: Object Methods

be added with an analogous `CreateCommand` function.

In Tcl, procedures are digested with the `Tcl_ProcCmd` to form closures of a function of type `Tcl_CmdProc` plus a single item of type `ClientData`. This is the internal representation for all Tcl commands. They are executed uniformly by calling the function with its clientdata and arguments, arranged in an argc/argv style array. The function that interprets procedures does so within a new innermost stack call frame. The closures are stored by command name in a hash table in the interpreter for efficient access. They return a single string result and an integer error code.

Object Tcl reuses the Tcl scheme and its implementation for finding and executing methods. This ensures a light footprint and a high degree of integration with the Tcl interpreter. In effect, objects are packages or namespaces within which the Tcl implementation is reapplied.

Methods are stored as closures in an object hash table by their name. `Tcl_ProcCmd` is used to digest Tcl methods into the common closure format of a single function and token of `ClientData`. Argument values and return results use the Tcl calling conventions. Methods look like procedures during their execution. This means that Tcl commands that access information further up the call chain (like `uplevel`, `global`, `upvar`, and `info`) function correctly. By convention, the object can be reached via the `Clientdata`.

name	description	type
name	variable defined with set	*var*
`self`	name of the object	*var*
`proc`	name of the method	*var*
`class`	class method defined	*var*
`next`	next shadowed method	*proc*

Table 6.2: Method Environment

6.2.3 Instance Variables

Object Tcl *instance variables* are introduced with the `set` method, which mirrors the Tcl `set` command. Figure 6.1 provides an example, using `things` to store the items currently on the stack. Again, the Tcl implementation of variables is reused.

Instance variables are made accessible during method execution with the same mechanism as used by `upvar` and `global`. They do not need to be declared (as do globals and locals in upper call frames) before they are used; all are made available automatically. This approach was chosen because most object-oriented languages treat instance variables as locals within methods, and local variables in Tcl do not need to be declared before they are used.

Some special variables are also defined to describe the context during method execution. For example, `self` holds the name of the object for which the method is executing. It is the equivalent of `this` in C++. Table 6.2 summarizes the environment in effect during method execution.

6.3 Classes and Inheritance

By themselves, objects allow related data and procedures to be grouped. Inheritance mechanisms strengthen this model by allowing the sharing of functionality between different types of objects. This encourages code reuse.

Inheritance is based on *classes* and their *superclasses*. In Object Tcl, each object is an instance of a class, and inherits its functionality from that class and its superclasses. Figure 6.5 shows the class relationships for the `astack` object used in the example of Figure 6.1. It is an instance of the class `Object`, and so inherits its behavior from the root object of the system. Object Tcl supports multiple inheritance — each class may have many superclasses. The class of an object is discovered with the `info class` method and changed with the `class` method.

A method is dispatched by searching first the object table of methods, then

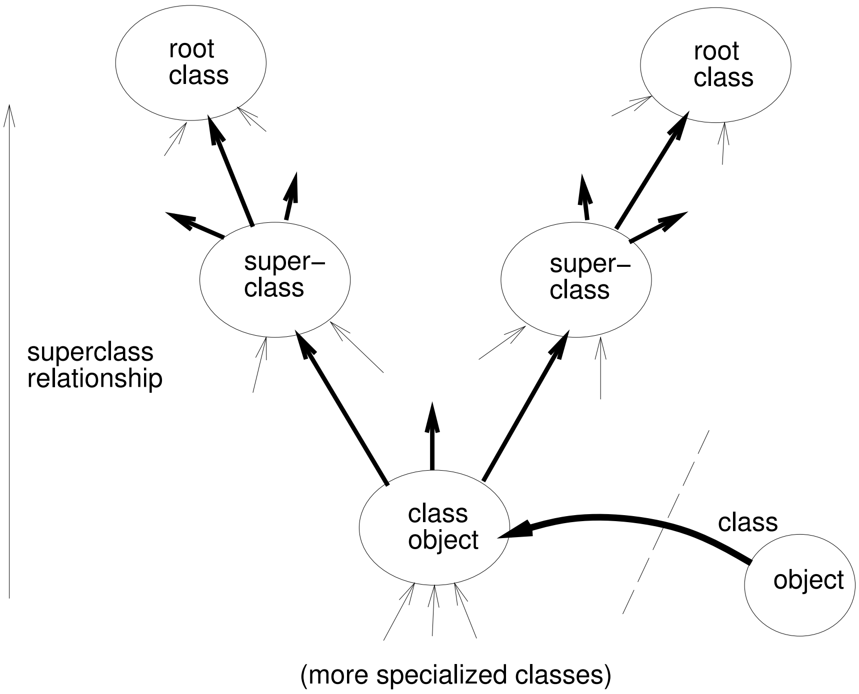

Figure 6.2: Superclasses and Method Dispatch

methods associated with the class, and then through the superclasses (Figure 6.2). The standard method for all objects are given in Table 6.1. The first method located is invoked, so methods closer to the object shadow more distant ones. This scheme requires a linear ordering of superclasses. Object Tcl uses a CLOS-like ordering, where the direct superclasses of a class take precedence from indirect superclasses.

This ordering is intuitive as well as simple to implement. It is intuitive because the local superclass orderings are respected globally: the programmer may think in terms of many small local orderings rather than a single large global ordering. It is simple because it may be determined with a topological sort algorithm, a variant of a breadth-first search of a graph that may be implemented in a handful of lines. To ensure efficiency, the ordering is cached since the running time of the algorithm is proportional to the size of the superclass graph. The code that specifies the method combination algorithm is given in Appendix B.

Figure 6.3 gives an example of multiple inheritance. Two classes that adhere to a protocol for managing collections are defined. The `Safety` class adds safety checking to collections, catching attempts to withdraw from empty collections. The `Stack` class implements a last-in-first-out queue, or stack. These classes are

```
Class create Safety -superclass Object

Safety proc create {acollection} {
  $self next $acollection
  $acollection set count 0
}

Safety instproc put {thing} {
  incr count
  $self next $thing
}

Safety instproc get {} {
  if {$count == 0} then { return {} }
  incr count -1
  $self next
}

Class create Stack -superclass Object

Stack proc create {astack} {
  $self next $astack
  $astack set things {}
}

Stack instproc put {thing} {
  set things [apply list $thing $things]
  return $thing
}

Stack instproc get {} {
  set top [lindex $things 0]
  set things [lreplace $things 0 0]
  return $top
}

Class create SafeStack -superclass {Safety Stack}

SafeStack create astack
astack put toast    ==> toast
astack get          ==> toast
astack get          ==> {}
```

Figure 6.3: Adding protection to the Stack class with the Safety mixin.

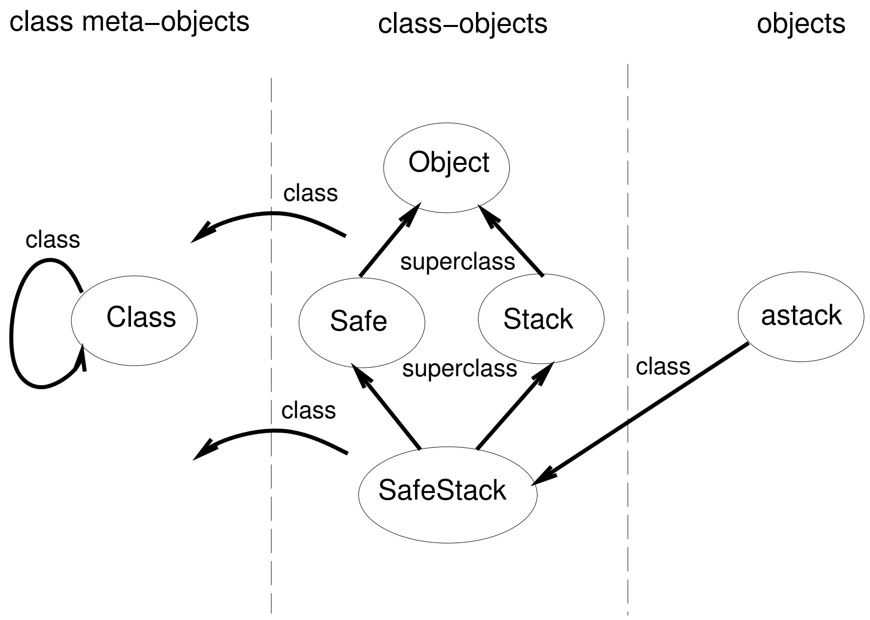

Figure 6.4: Class Relationships for SafeStack

combined to form a `SafeStack` class, which is then demonstrated. Figure 6.4 shows these class relationships. The `Safety` class can be combined with any type of collection (for example, stack, queue, and set) but is not required. It is known as a *mixin* in Flavors [29], and demonstrates an intuitive use of inheritance that is difficult to achieve with only single inheritance.

The inheritance mechanisms cause a minor complication to demand loading. Methods written in Tcl should be auto-loaded in a similar fashion as procedures. This distributes the startup time taken to parse method definitions and reduces unneeded parsing. However, because the dispatch follows a search path, shadowing methods may not be loaded in favor of their more primitive methods already loaded by other paths. To overcome this complication, auto-loading is supported with the `auto` method. This inserts a code stub that forces demand loading if the method is invoked. The stubs are typically inserted at application startup, but are considerably faster than loading the methods themselves.

6.3.1 Method Combination

Multiple inheritance needs method combination to be effective. Method combination allows the functionality of a method to be split across classes, rather than simply being shadowed or not. It increases the abstraction capabilities of inheritance.

As an example, a constructor in C++ implicitly invokes the constructors of all its base classes — it combines their construction methods. Similarly, destructors in C++ combine the destructors of their base classes, but in a reverse order. Making the combination mechanism accessible to the programmer allows derived classes to refine the behavior of their subclasses by wrapping functionality around them. CLOS terms these combinations before, after, and around methods.

In Object Tcl, method combination is achieved with the `next` method. When called within a method, it causes the next most shadowed implementation of the method to be invoked. It determines this automatically according to the precedence ordering and is not told explicitly.

The example in Figure 6.3 uses method combination. The `get` and `put` methods of the `Safety` *mixin* class first contribute their functionality and then call the next most shadowed method. For the `SafeStack` class, method combination ensures that the `get` and `put` methods of the `Stack` class are called, even though the `Stack` class has no direct relationship to the `Safety` class.

name	description
`create`	make new instance (default)
`superclass`	set superclasses
`instproc`	define Tcl method for instances
`instauto`	demand load Tcl method for instances
`info instances`	list instances
`info superclass`	list superclasses
`info instprocs`	list Tcl methods for instances
`info instcommands`	list Tcl/C methods for instances

Table 6.3: Class Methods

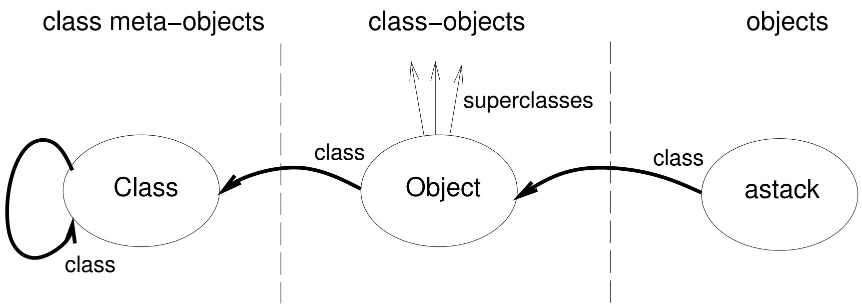

Figure 6.5: A Class Relationship

6.4 Class Implementation

In Object Tcl, classes are implemented as objects that have the added ability to create and manage other objects. They support the superclass relation and serve as a repository for methods on behalf of their objects. This approach means the power of the object system can be applied to influence class behavior.

Table 6.3 lists the methods for class objects. The superclasses of a class are discovered with `info superclass` and changed with `superclass`. `Instproc` is used to define methods for use by instances. `Get` and `put` in Figure 6.3 are inherited methods. `Instauto` is used to set demand loading of methods used by instances. The `create` method is used to manufacture new instances. As the default method, it allows the name of an instance in the method position to cause its creation.

Because they are implemented as objects, classes can have their own methods and instance variables. This provides a convenient location for information that is shared by the instances of a class, known as static members in C++. Making classes be objects helps Object Tcl provide increased functionality without increasing the size of its implementation.

6.4.1 Class Meta-Objects

Since classes are themselves objects, they are managed in turn by other classes. These managing objects form a third category of objects called *class meta-objects*. They are closed under the class relationship, in effect responsible for managing themselves. Figure 6.5 illustrates this by showing the class relationships for the `astack` object of Figure 6.1.

The standard class meta-object is called `Class`. Its `create` method is used to make new class objects. New class meta-objects may be manufactured too, as derived classes of `Class`. This allows the behavior of classes to be widely changed, and is especially useful in conjunction with the introspective features of the language.

6.5 Introspection

Introspection allows the running program to discover information about itself. Object Tcl makes information about the state of the object system available in the same manner that Tcl does for its procedures and variables. Each object has an `info` method that matches the Tcl `info` command by performing an analogous function for objects. It is used to list the methods and instance variables for objects, as well as the class of an object. These methods are summarized in Table

6.1. For class objects, the `info` method has additional options for listing information about superclasses and instances, and methods managed on behalf of instanced. These methods are summarized in Table 6.3. Finally, during method execution, special instance variables are bound to describe the execution context, such as the name of the object and the location of the method implementation. These bindings are summarized in Table 6.2.

Introspection is most useful when the program structure is dynamic and may be altered at any time. Tcl itself is dynamic in that variables and commands may be removed as well as created, and Object Tcl commands match this model.

6.6 Perspective

PAVES uses Object Tcl as the foundation on top of which video programs are built. Objects naturally model the VuSystem media processing modules and the operations they undergo. By using Object Tcl to organize programs instead of simply following an object-style, PAVES gains the advantages of inheritance, introspection, specialization, and extensibility.

The structuring conventions described in Chapter 5 use these mechanisms to embed the functionality of PAVES in a range of video programs. An object naming scheme is well-suited for defining manipulations on programs since it can express the patterns of similarities and differences between programs in a modular fashion. Inheritance allows the behavior of opaque collections of modules to be defined in terms of the behavior of its members.

Chapter 7

Results and Conclusions

In this report, I have presented an interactive programming system for media computation. I have discussed its motivation, architecture, implementation and use. PAVES combines the programming aspects of multimedia and visualization systems to allow users to control applications while they are manipulating live video.

PAVES is available throughout a range of video applications as a set of automatically generated programming windows. A direct manipulation interface based on flow graphs is used to observe and control the broad structure of media processing implemented by the application. Control panel, code fragment, and interpreter windows complement the flow graph. Together, these windows permit the user to employ both textual and graphical methods to solve programming tasks.

In designing PAVES, I developed a cooperative model for integrating the graphical and textual representations of an application. This allowed PAVES to be used to control video programs created outside of the visual programming environment. I solved the problem of keeping the windows in synchrony with the program to which they correspond by adopting a presentation-style approach. This supports the cooperative model by allowing the different programming windows to be used simultaneously. The implementation of the model is built on Object Tcl, an object-oriented extension to Tcl that I developed for the purpose. An interesting aspect of PAVES is its support for grouping in the flow graph window. This not only permits specific modules to be encapsulated, but results in the manufacture of an abstraction that is a first-class object.

The following sections describe experience with using PAVES, and the results it has yielded. Future work, based on the results, is then suggested.

7.1 Experience with PAVES

Half a dozen users have experimented with PAVES through a variety of video programs[1]. It has successfully been used as a consistent programming means for controlling video applications while they manipulate live video.

7.1.1 Suitability

A strength of the system is that its programming windows support both textual and visual operations, and are suited to tasks that are largely complementary. This equips the user with a set of tools that comprise a programming interface across a variety of tasks, with the role of each tool becoming well-defined.

Flow graphs show the broad structure of a program. They are apt for reconfiguration tasks, but not for fine-tuning. Conversely, control panels emphasize the detailed options of the program rather than its structure. They are apt for customization tasks, but not the large-scale re-organization of the program.

Interpreters cooperate with the flow graphs. Both tools overlap in terms of the tasks they can accomplish, and this gives the user flexibility in deciding how to accomplish a given task. The visual methods are suited to rapid or special case changes. The textual methods allow a recourse for complicated or repetitive changes. Textual monitoring of graphical actions helps the user form an association between the two and encourages the use of both methods.

Code fragment windows are useful for prototyping. They take the association between visual and textual representations further by showing the interpreted VuSystem code that corresponds to the selected modules of the flow graph. They provide access to a representation midway between the flow graph view of modules and their C++ implementation, to which there is no run-time access.

7.1.2 Modes of Use

Over time, several modes of use have become dominant. Programming is often a secondary concern to simply using the application. The embedding of the programming tools is valuable because it allows the visual programming windows and other tools to remain hidden until they are required.

[1]Actually, many more have used PAVES. The TNS World Wide Web technology demonstrations are available throughout the Internet and include several video programs from which PAVES is accessible. The feedback gained from this exposure was positive, but not detailed enough to include in this section.

The flow graphs are used by novices and experts alike to study program structure. Because they show the modules created during a particular session, the flow graphs are often used by developers to check the program contents for debugging purposes. The flow graph is a natural and accurate representation that explains the operation of the program at a high level. It is quickly grasped because it is a diagram, and has the added advantage that it is active: it may be tweaked to test hypotheses.

Customization and experimentation tasks are frequent. When a program is being examined, changes are effected quickly and consistently. And they are accompanied by continuous visual feedback. Most customizations are temporary, lasting only for the duration of a session. This is because default behaviors plus command line overriding tend to capture the parameterization needed between runs.

PAVES has also proved feasible for one-step prototyping by saving and refining the code of user-grouped constructions. But it is weak when used for the iterative development of applications by alternate graphical and textual refinement. The major difficulty is that the quality of synthesized code does not match that of code written by developers, especially in terms of naming and procedural abstraction. In effect, multiple passes through the graphical layer tend to undo the programmer's organization, if not end result.

7.1.3 Applicability

At the outset, it was unclear what range of programs PAVES would be able to control. There is an inevitable tension between the design of an expressive visual language and that of a clear user interface. Beyond this, the success of the cooperative approach impacts the range of programs.

PAVES has been applied to video programs written directly in Object Tcl, and has faithfully represented them. Such programs now represent the majority of the applications used in conjunction with the system. The flow graph language has proved adequate for managing these programs, which are typified by the BlueScreen application of Chapter 4, that is, programs that process video rather than simply browse it. Some aspects of programs fell outside the system, but this often did not adversely affect the use of flow graphs. Many visual processing experiments, for example, calculate figures of merit for their result decisions, but this portion of the program is independent of the flow graphs.

A practical difficulty lies in the failure modes of the system. Since general programs may not comply with the structuring conventions, the system can fail in ways that are not easily visible or are difficult to understand. For example, the code synthesized from such programs may appear correct but not be useful. It may omit needed functionality or bind too tightly to the environment in effect when it is created. This problem has tended to restrict development with the system to one-step prototyping.

7.2 Conclusions

Experience with PAVES has shown computations on video to be a problem domain in which both interactivity and programmability are useful. Video is a temporal and visual medium that is naturally manipulated when live and in a graphical environment that provides continuous feedback. A programming system based on a visual representation readily lets many interesting computations be applied to video in a systematic manner. Beyond this observation, several more general conclusions can be drawn.

7.2.1 Cooperative Programming Methods

As a whole, the system shows the feasibility and usefulness of the cooperative approach: many programs of different origin were used with both graphical and textual programming facilities. Few commercial applications combine methods in this manner, and I speculate this is because of the necessary infrastructure, rather than because the technique is of little value.

Implementing cooperation required a synchronization mechanism to ensure that the graphical windows accurately reflected the program to which they corresponded. This is a general useful mechanism that is not limited to PAVES. Implementing cooperation also required a means of translating between program representations. This was supported by the introspective and specialization properties of Object Tcl.

Much of the implementation success was the result of focusing on a specialized problem domain with its consequent reductions in the scope of the design task. Specializing allowed the structuring conventions to balance restrictions for the programmer against the regularities needed to programmatically manipulate programs.

The conventions cost the programmer little, but benefited the user greatly. The philosophy used was to adopt conventions that did not restrict the type of programs that could be formed but did simplify the extraction of useful information. For example, the root of the program tree is conventionally called `vs`, though any name would be valid. This does not reduce the number of useful programs, but does make it simple to find the root of the program tree. On the whole, few conventions were needed beyond those of regular VuSystem applications, with the notable exception of rewriting the object library to better support abstractions.

7.2.2 Visual Program Representations

The flow graph form of representation is apt for describing many computer-participative video programs. As for the domains of scientific visualization and image processing, flow graphs provide a natural visual representation of the computation that occurs as media is processed by a video program.

An important aspect of implementing the flow graph, and especially manipulations on it, was Object Tcl. This acted as an intermediate form between the compiled C++ code and the graph itself. As a scripting language, Object Tcl hid many of the C++ implementation details that were not relevant to higher level representations. As an interpreted and introspective language, Object Tcl allowed the running program to be used as the basis for its own representation, obviating the many consistency problems caused by using more than one database.

The approach of using a visual representation as a higher level interface to running scripts is not limited to PAVES. It may be applied to Tk [33], for example. This is because Tk arranges its user interface widgets in terms of a containment tree, which has a natural visual representation.

7.2.3 Abstractions in User Programming

As well as extensibility for the user, abstraction mechanisms are useful for importing and exporting objects between the user programming system and the environment on which it rests.

An ability to abstract over patterns of usage is the basis for extensibility, which is often the difference between toy systems and real-world systems. Yet few user programming systems support it. This may be because it is not necessary to begin using a system or because it complicates the interface design. It can require the user interface to view the program at different levels and to convey distinctions between classes and instances of a class. Neither of these may be necessary otherwise.

In PAVES, abstraction was presented visually as groupings, and textually as the opaque structuring of a collection of modules. Both representations were useful.

Abstraction was useful for importing textual modules to the graphical environment, encapsulating their less constrained functionality so that it could be safely used. This is essentially the third-party extensibility of XCMDs in HyperCard and plug-ins in Adobe Premiere and PhotoShop, though used with a different emphasis. Hypercard and the others use the import mechanism in place of user extensibility. It is a poor substitute because it presents too steep a learning curve to be useful to the majority of users.

Conversely, abstraction was useful for exporting visually grouped constructions to the development library. An encapsulation barrier sanitized groupings for inclusion in hand-crafted programs. It was used to separate synthesized code from hand-crafted code. The grouping mechanism itself was interesting because it allowed general group definitions to be inferred after-the-fact from specific modules. This is counter to the sequence of operations in traditional textual programming languages, such as C++, but is well-suited to interactive programming because it provides an evolution path that minimizes outstanding operations.

7.2.4 Object-Oriented Programming

Implementing the system led to two insights about object-oriented programming.

Multiple inheritance proved simple and efficient to implement as well as intuitive to use. Mechanisms for combining methods to not need to be complicated, as demonstrated by the topological sort algorithm. At the same time they may be natural to use, as exemplified by the mixin model of Flavors [29].

The lack of automatic method combination in C++ appears to be the major reason why multiple inheritance is perceived to be complicated and of little value there. In C++, the next most shadowed method must be specified explicitly using the scope resolution operator. This is suited to static superclass relations and unnecessarily propagates information about the class hierarchy to methods. It becomes more difficult for multiple inheritance because it forces the programmer to deal with the large global ordering rather than thinking in terms of small local orderings. It is also less powerful. The example programs of Chapter 6, for example, would fail if organized in the same way in C++. They rely on calls across the hierarchy rather than to direct superclasses, and so cannot be specified explicitly.

Object naming is effective at capturing approximate models. Manipulating programs, for instance, was a task in which one model was effective in the majority of cases. Inheritance allowed it to be applied as a default. Specialization then accommodated particular cases by letting them substitute a specific model for the general one. Together, inheritance and specialization allowed an approximate operation to be specified in a modular way. Object Tcl provided strong support for this style by allowing both individual instances and classes to be specialized.

7.3 Further Work

There are several interesting directions for future work that are motivated by the results. The cooperative approach can be explored further, and the role of abstractions in user programming can be studied.

7.3.1 Cooperative Development

The cooperative use of graphical and textual programming may be extended to cooperative development as well as applied in other problem domains. Much of the power of the system derives from its ability to combine programming methods during a session. Combining them across iterative development cycles would allow for different modes of use than the current system can support.

The problem with using both programming methods repeatedly lies with the quality of the synthesized code. Automatic code generation is not trivial and is unlikely to reach human standards in the near future. A pragmatic approach is to extend the system to incorporate the input of human programmers.

A better model of default values would obviate the need for specifying the value of much program state. It could also retain knowledge of session-specific parameters, such as command-line options, which are effectively a higher level default.

Conventions for tuning the generation process would allow the programmer to improve on general behavior as a special case. By using introspective facilities, the system may be able to retain hand-crafted idiosyncrasies such as the use of variable names instead of literals.

Schemes that allow the system to remain open in terms of the programs it can control yet ensure that manipulations are safe are also needed for robustness. The ability to manipulate legacy video programs created textually, not visually, is a strength of PAVES. But its implementation suffered from the tendency to silently fail while appearing to work successfully. This is because it lacked mechanisms to verify the legality of programs. A better implementation would fail gracefully, perhaps by using redundant structuring conventions for checking the integrity of the program against its model or gauging its level of conformance.

7.3.2 Manipulating Abstractions

A systematic means of visually manipulating abstractions would strengthen the system. PAVES uses abstractions as a barrier to import objects into the graphical system as well as to export user groupings. It uses the same internal representation for both, making user groupings first-class. This avoids artificial restrictions on the groupings, but makes them subject to the same limitations as imported abstractions — their internals cannot be safely manipulated by the visual system.

A system for manipulating abstractions must convey several semantics across the user interface. For user groupings, the internal composition is essentially a definition, which is open to editing. For imported objects, the internal composition reveals the instantaneous state. Both are instructive to the user. The system must further distinguish editing a single grouping from editing the definition of the grouping. For imported abstractions, it must resolve the difficulties of what to present and how to signal the implications of editing. Imported abstractions may have state that cannot be represented visually, and if they are manipulated then future operations, such as synthesizing code, may fail.

7.3.3 Extending Object Tcl

Improvements and additions to the Object Tcl language can make it more generally applicable to user programming systems.

A meta-object protocol (MOP) [21] would let the language user incrementally modify and extend the Object Tcl definition. It would then occupy a region of the design space, rather than a point, and consequently be more widely applicable. Basic object traits such as creation, destruction, method dispatch, and precedence orderings for inheritance should be accessible via a MOP in both Tcl and C.

Standard tools, such as program browsers, could provide a lead-in for user programming interfaces. By using introspective capabilities, each object can be described in detail and the entire class inheritance graph can be displayed. In conjunction with a MOP, these tools can monitor events such as object creation and destruction and aid in debugging. They should also be customizable by using inheritance.

7.4 Summary

In conclusion, interactive programming using visual representations is a valuable technique for controlling video applications. The report has presented PAVES, a system that combines a specially constructed programming system with a graphical language by mapping between visual and textual representations as the video program runs. A cooperative model, supported by a synchronization mechanism, allows the visual and textual programming methods to be used simultaneously.

Appendix A

Structuring Conventions used by PAVES

This appendix summarizes the key objects and methods used to structure video programs. The standard classes are described first. Then their reflective methods and methods supporting the flow graph, control panel, and code fragment views are catalogued.

A.1 Standard Classes

Video programs are arrangements of objects derived from three base classes.

A.1.1 Modules and Transparent Collections

The `VsEntity` class implements the functionality common to media processing objects, including collecting them into hierarchies. Its instances are generic modules or entities. It is derived from `Object`, the root class of Object Tcl. It is of class `Class`, since it is able to manufacture modules.

A.1.2 Ports

The `VsInputPort` and `VsOutputPort` classes implement the functionality of input and output media ports, respectively. Ports are associated with entities and do not exist in isolation of them. These classes are derived from `Object` and are of class `Class`, since they are able to manufacture ports.

A.1.3 Opaque Collections

The `VsOpaque` class implements the functionality common to collections of media processing objects that act as a single larger media processing object. It is the basis for abstraction over media objects. Its instances are opaque collections that are functionally indistinguishable from generic modules or entities. It is derived from `VsEntity`, supplementing it with the ability to encapsulate media objects. It is of class `Class`, since it is able to manufacture opaque collections.

A.2 Reflective Methods on Entities

Each entity has access to the following introspective methods over and above those of all objects.

A.2.1 Inputs

```
<entity> inputs
  ==> <inputs>
```

The `inputs` method gets the names of the input ports of the entity. It returns:

inputs *(Command List)* The names of the input ports.

A.2.2 Outputs

```
<entity> outputs
  ==> <outputs>
```

The `outputs` method gets the names of the output ports of the entity. It returns:

outputs *(Command List)* The names of the output ports.

A.2.3 Children

```
<entity> children
  ==> <children>
```

The `children` method gets the names of the child entities of the entity. It returns:

children *(Command List)* The names of the child entities.

A.2.4 Parent

```
<entity> parent
  ==> <parent>
```

The `parent` method gets the name of the parent entity of the entity. It returns:

parent *(Command)* The name of the parent entity.

A.2.5 Info Options

```
<entity> info options [<pattern>]
  ==> <options>
```

The `info options` method provides the names of option methods for the entity. The names of both primitive option methods, implemented in C/C++, and option procs, implemented in Tcl, are accessible. It takes:

pattern *(String)* A regular expression.

It returns:

options *(List)* The list of option method names that match the pattern. If no pattern is supplied, all option method names are returned.

A.3 Reflective Methods on Ports

Input and output ports have access to the following introspective methods over and above those of all objects.

A.3.1 Bind

```
<input port> bind
  ==> <output port>
```

The `bind` method gets the output port bound to the input port. It returns:

output port *(Command)* The name of the output port. If the input port is not bound then an empty command is returned.

A.3.2 Connect

```
<output port> connect
  ==> <input port>
```

The `connect` method gets the input port connected to the output port. It returns:

input port *(Command)* The name of the input port. If the output port is not connected then an empty command is returned.

A.3.3 Entity

```
<port> entity
  ==> <entity>
```

The `entity` method gets the name of the entity that owns the port. It returns:

entity *(Command)* The name of the owning entity.

A.4 Reflective Methods on Collections

Opaque collections support the same introspective methods as entities and use the same syntax. To accomplish this, the implementation of the following methods is overridden:

inputs The names of input ports are aliased to appear as if they correspond to a single object.

outputs The names of output ports are aliases to appear as if they correspond to a single object.

children Captured children are not reported.

info options Option methods are redirected to appear as if they correspond to a single object. Their names are qualified as necessary to avoid clashes.

A.5 Methods Supporting Views on Entities

Each entity has a default implementation of the following methods that support the construction of flow graphs, control panels, and code fragments. It may override them to cater to special cases.

A.5.1 Draw

```
<entity> draw <w>
```

The `draw` method translates the entity and its children into a set of widgets that represents their flow graph. It takes:

w *(Command)* A name of a widget within which to construct the flow graph.

A.5.2 Panel

```
<entity> panel <w> <orient>
```

The `panel` method translates the entity and its children into a set of widgets that represents their control panel. It takes:

w *(Command)* A name of a widget within which to construct the control panel.

orient *(-fromHoriz | -fromVert)* An initial orientation (horizontal or vertical) for tiling the panels.

A.5.3 Describe

```
<entity> describe
  ==> <code>
```

The `describe` method translates the entity and its children into a code fragment. It returns:

code *(String)* The executable Tcl code that represents the entity and its children.

A.5.4 Option

```
<entity> option <name> <args> <body>
```

The `option` method defines an option method for the entity. It is similar to the `proc` method on all objects in syntax and use, except that the method is marked as an option for future reference. It takes:

name *(String)* A name for the option method.
args *(List)* A list of formal parameters to the option method.
body *(List)* A body for the option method.

A.5.5 XPosition

```
<entity> xPosition [<pos>]
  ==> <pos>
```

The `xPosition` method sets and gets the x position of the entity. It takes:

pos *(String)* An x position of the entity.

It returns:

pos *(String)* The x position of the entity.

A.5.6 YPosition

```
<entity> yPosition [<pos>]
  ==> <pos>
```

The `yPosition` method sets and gets the y position of the entity. It takes:

pos *(String)* A y position of the entity.

It returns:

pos *(String)* The y position of the entity.

A.5.7 ConfigCallback

```
<entity> configCallback [<callback>]
  ==> <callback>
```

The `configCallback` method sets and gets the configuration callback code of the entity. It takes:

callback *(String)* A callback string for the entity to execute when its configuration changes.

It returns:

callback *(String)* The callback string the entity will execute when its configuration changes.

A.5.8 EvalConfigCallback

```
<entity> evalConfigCallback <args>
```

The `evalConfigCallback` method causes the configuration callback code of the entity to be invoked. It takes:

args *(List)* A list of arguments to pass to the configuration callback.

A.5.9 Reparent

```
<entity> reparent <parent>
```

The `reparent` method moves the entity to a new parent. It takes:

parent *(Command)* A new parent for the entity.

A.6 Methods Supporting Views on Ports

Input and output ports provide no methods for supporting views directly; they are manipulated through the entity that owns them instead.

A.7 Methods Supporting Views on Collections

Opaque collections provide the same methods for supporting views as entities and use the same syntax. To accomplish this, the implementation of the `panel` method is overridden to create a default panel that is the sum of the captured children. In addition, opaque collections provide the following two methods that support viewing abstractions.

A.7.1 Capture

```
<collection> capture <names>
```

The `capture` method adds selected children of the entity to the opaque collection. It takes:

names *(Command List)* A list of names of the children to be captured.

A.7.2 Advertise

```
<collection> advertise <name> [ <methods> ]
```

The `advertise` method creates a public interface to the given methods of the entity on the named entity. It takes:

name *(Command)* A name of an opaque collection on which to create the public interface.

methods *(List)* A list of the methods to advertise on the entity. Elements of the list are one or two element lists. If two elements, they give the public and private name of a method, respectively. If one element, the public and private name is identical.

Appendix B

Object Tcl Reference

This appendix summarizes the behavior of the Object Tcl language. First, the two standard objects are listed, followed by their methods. Then the environment in which methods execute and the class precedence ordering are described.

B.1 Standard Objects

Object Tcl appears to the programmer as two standard class objects which may be specialized as needed.

B.1.1 Object

The `Object` class implements the functionality common to all objects. Its instances are generic objects. It is a root class, having no superclass. It is of class `Class`, since it is able to manufacture objects.

B.1.2 Class

The `Class` class implements the functionality common to all classes. Its instances are generic classes. Its superclass is `Object`, since classes are objects too and inherit object behavior from the standard object. Its class is `Class`, since it is able to manufacture objects. `Class` is the meta-class object.

B.2 Object Methods

Each object has access to the following methods.

B.2.1 Name

```
<object> name
  ==> <name>
```

The `name` method gets the primary name of the object. It returns:

name *(Command)* The name of the object.

B.2.2 Names

```
<object> names [<pat>]
  ==> <name>
```

The `names` method gets all names of the object. It takes:

pat *(String)* A regular expression.

It returns:

name *(Command)* The name of the object.

B.2.3 Alias

```
<object> alias <name>
```

The `alias` method makes a secondary name for the object. It takes:

name *(String)* A name of for the object.

B.2.4 Class

```
<object> class [<name>]
```

The `class` method sets the class of the object. It takes:

name *(String)* A name of a class object.

B.2.5 Destroy

```
<object> destroy
```

The `destroy` method destroys the object, deleting all of its methods and instance variables before removing the command from the interpreter.

B.2.6 Proc

```
<object> proc <name> <args> <body>
```

The `proc` method defines a method for the object. It is similar to the `proc` top-level command, except that the procedure is defined on the object. It takes:

name *(String)* A name for the proc.
args *(List)* A list of formal parameters to the proc.
body *(List)* A body for the proc.

B.2.7 Next

```
<object> next [<args>]
  ==> <value>
```

The `next` method provides access to methods defined further up the class graph. It is called within the implementation of a Tcl proc on an object to reach the next most method in the class precedence ordering with the same name. It takes:

args *(List)* A list of arguments to pass to the next method.

It returns:

value *(String)* The value returned by the next method.

B.2.8 Auto

```
<object> auto <name> <command>
```

The `auto` method installs a stub that forces a method implementation to be demand loaded if and when the method is invoked. It takes:

name *(String)* A method name that is to be demand loaded.
command *(String)* A command that is evaluated to load the named method.

B.2.9 Info Procs

```
<object> info procs [<pattern>]
  ==> <procs>
```

The `info procs` method provides the names of methods for the object. Only the names of procs, implemented in Tcl, are accessible. It takes:

pattern *(String)* A regular expression.

It returns:

procs *(List)* The list of method names that match the pattern. If no pattern is supplied, all method names are returned.

B.2.10 Info Commands

```
<object> info commands [<pattern>]
  ==> <commands>
```

The `info commands` method provides the names of methods for the object. The names of both primitive methods, implemented in C/C++, and procs, implemented in Tcl, are accessible. It takes:

pattern *(String)* A regular expression.

It returns:

commands *(List)* The list of method names that match the pattern. If no pattern is supplied, all method names are returned.

B.2.11 Info Vars

```
<object> info vars [<pattern>]
  ==> <vars>
```

The `info vars` method provides the names of instance variables for the object. It takes:

pattern *(String)* A regular expression.

It returns:

vars *(List)* The list of instance variable names that match the pattern. If no pattern is supplied, all instance variable names are returned.

B.2.12 Info Class

```
<object> info class [<class>]
  ==> <0 | 1 | class>
```

The `info class` method provides the access to the class of the object and tests class membership. It takes:

pattern *(String)* A class name.

It returns:

0 | 1 | class *(String | Command)* If no pattern is supplied, the class of the object is returned. Otherwise, a true or false result is returned depending on whether the object is a member of the class specified by the pattern.

B.3 Class Methods

Each class has access to the following methods over and above the methods accessible to all objects.

B.3.1 Create

```
<class> create <name> [<keyword> <value> ... ]
  ==> <object>
```

The `create` method creates and initializes a new object that is an instance of the class. It is the default method for classes, and the `create` method argument specifier may be omitted if the request remains unambiguous. It takes:

name *(String)* A name of the object to be created.
keyword *(String)* A name of a method the instance provides.
value *(String)* A value used as the argument to the method.

It returns:

object *(Object)* The newly created object.

B.3.2 Superclass

```
<class> superclass [<classes>]
```

The `superclass` method sets the superclasses of the class. It takes:

classes *(List)* A list of classes.

B.3.3 InstProc

```
<class> instproc <name> <args> <body>
```

The `instproc` method defines an instance method for the class. It is similar to the `proc` top-level command, except that the procedure is defined on the class for use by its instances. It takes:

name *(String)* A name for the proc.
args *(List)* A list of formal parameters to the proc.
body *(List)* A body for the proc.

B.3.4 InstAuto

```
<class> instauto <name> <command>
```

The `instauto` method installs a stub that forces an instance method implementation to be demand loaded if and when the method is invoked. It takes:

name *(String)* An instance method name that is to be demand loaded.

command *(String)* A command that is evaluated to load the named instance method.

B.3.5 Info InstProcs

```
<class> info instprocs [<pattern>]
  ==> <instprocs>
```

The `info instprocs` method provides the names of instance methods for the class. Only the names of instance procs, implemented in Tcl, are accessible. It takes:

pattern *(String)* A regular expression.

It returns:

instprocs *(List)* The list of instance method names that match the pattern. If no pattern is supplied, all instance method names are returned.

B.3.6 Info InstCommands

```
<class> info instcommands [<pattern>]
  ==> <instcommands>
```

The `info instcommands` method provides the names of instance methods for the class. The names of both primitive instance methods, implemented in C/C++, and instance procs, implemented in Tcl, are accessible. It takes:

pattern *(String)* A regular expression.

It returns:

instcommands *(List)* The list of instance method names that match the pattern. If no pattern is supplied, all instance method names are returned.

B.3.7 Info Instances

```
<class> info instances [<pattern>]
  ==> <objects>
```

The `info instances` method provides the names of instances of the class. It takes:

pattern *(String)* A regular expression.

It returns:

objects *(Command List)* The list of objects instances that match the pattern. If no pattern is supplied, all instances are returned.

B.3.8 Info Superclass

```
<class> info superclass [<class>]
  ==> <0 | 1 | classes>
```

The `info superclass` method provides the access to the superclasses of the class and tests superclass membership. It takes:

pattern *(String)* A class name.

It returns:

0 | **1** | **classes** *(String* | *Command List)* If no pattern is supplied, the superclasses of the class are returned. Otherwise, a true or false result is returned depending on whether the class is a subclass of the class specified by the pattern.

B.4 The Method Environment

During method execution, all instance variables (that were previously declared with the `set` method) are accessible. They appear as local variables, but their state is saved across method invocations.

In addition, the following special variables are always defined:

self The object on behalf of which the method is executing.

proc The name of the method which is executing.

class The class on behalf of which the method is defined, or null is the method is defined on a non-class object.

B.5 The Class Precedence Ordering

When a method is invoked, it is searched for on the object itself and then along the class precedence ordering. The first method found is executed. The `next` method may be used to continue the search and combine method implementations.

The precedence ordering always begins with the class of the object. It orders superclasses with a CLOS-like scheme. The following rules always hold locally and are useful for determining precedence:

1. A class always has precedence over its superclasses

```
void
VsTclClass::TopologicalSort(VsTclClass* base) {
  color = True;
  for (ClassList* l = SuperClass(); l != 0; l = l->next) {
    VsTclClass* next = l->cl;
    if (next->color == False)
      next->TopologicalSort(base);
  }
  ClassList* chain = base->precedence;
  base->precedence = new ClassList;
  base->precedence->cl = this;
  base->precedence->next = chain;
}

void
VsTclClass::ComputePrecedence() {
  while (precedence != 0) {
    ClassList* n = precedence->next;
    delete precedence; precedence = n;
  }
  TopologicalSort(this);
  for (ClassList* l = precedence; l != 0; l = l->next)
    l->cl->color = False;
  cache = True;
}
```

Figure B.1: Algorithm for computing the precedence list.

2. Each class sets the order of its direct superclasses by the order they are specified with the `superclass` method

The complete precedence ordering is computed for a given class by the topological sort algorithm given in Figure B.1.

Bibliography

[1] J. F. Adam, H. H. Houh, M. Ismert, and D. L. Tennenhouse. A Network Architecture for Distributed Multimedia System. In *Proceedings of the International Conference on Multimedia Systems and Computing*, pages 76–86, Boston, MA, May 1994. IEEE.

[2] Joel. F. Adam and David. L. Tennenhouse. The vidboard: A video capture and processing peripheral for a distributed multimedia system. In *Proceedings of ACM Multimedia 93*, pages 113–120, Anaheim, CA, August 1993. ACM.

[3] G. D. Battista, P. Eades, R. Tamassia, and I. G. Tollis. Algorithms for drawing graphs: an annotated bibliography. ftp from wilma.cs.brown.edu (128.148.33.66), June 1993. file /pub/gdbiblio.ps.Z.

[4] Eugene C. Ciccarelli. Presentation based user interfaces (revised Ph.D.). Technical Report MIT/AI/TR 794, Artificial Intelligence Lab., Massachusetts Institute of Technology, August 1984.

[5] Charles L. Compton and David L. Tennenhouse. Collaborative load shedding for media-based applications. In *Workshop on the Role of Real-Time in Multimedia/Interactive Computer Systems*, Raleigh, NC, November 1993.

[6] Digital Video Applications Corp. *DiVA VideoShop: Users Guide*. Digital Video Applications Corp., November 1991.

[7] Microsoft Corporation. *Microsoft Video For Windows Users Guide*. Microsoft Corporation, 1992.

[8] Isabel. F. Cruz. DOODLE: A Visual Language for Object-Oriented Databases. In *Proceedings of SIGMOD*. ACM, 1992.

[9] Michael Eisenberg. Programmable applications: Interpreter meets interface. Technical Report AI Memo 1325, MIT, October 1991.

[10] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[11] M. Gantt and B. Nardi. Gardeners and Gurus: Patterns of cooperation among CAD users. In *Proceedings of CHI*, pages 107–117, May 1992.

[12] Adobe Systems Inc. *Premiere*. Adobe Systems Inc., 1991.

[13] Apple Computer Inc. *Dylan: an object-oriented dynamic language*. Apple Computer Inc., April 1992.

[14] Apple Computer Inc. *HyperCard (version 2.2)*. Apple Computer Inc., 1993.

[15] Apple Computer Inc. *Inside MacIntosh: Quicktime, Inside MacIntosh: Quicktime Components*. Addison Wesley, 1993.

[16] MacroMind Inc. *MacroMind Director (Version 3.0): Overview Manual*. MacroMind Inc., June 1991.

[17] Sun Microsystems Inc. *Solaris XIL 1.1 Imaging Library: Programmers' Guide*. Sun Microsystems Inc., November 1993.

[18] ISO/IEC JTC1/SC29. *Coded Representation of Picture, Audio, and Multimedia/Hypermedia Information*. Committee Draft of Standard ISO/IEC 11172, 1991.

[19] ISO/IEC JTC1/SC2/W10. *Digital Compression and Coding of Continuous-Tone Still Images*. IEC Draft International Standard 10918-1, 1992.

[20] Sonya E. Keene. *Object-Oriented Programming in COMMON LISP: A Programmer's Guide to CLOS*. Addison-Wesley, December 1988.

[21] G. Kiczales, J. des Rivieres, and D. Bobrow. *The Art of the Metaobject Protocol*. The MIT Press, 1992.

[22] C. Lindblad, D. Wetherall, and D. Tennenhouse. The VuSystem: A Programming System for Visual Processing of Digital Video. In *Proceedings of ACM Multimedia 94*. ACM, October 1994.

[23] Wendy E. Mackay. Triggers and barriers to customizing software. In *CHI'91 Conference Proceedings*, pages 153–160. ACM, April 1991.

[24] James Matthews, Peter Gloor, and Fillia Makedon. Videoscheme: A programmable video editing system for automation and media recognition. In *Proceedings of ACM Multimedia 93*, pages 419–426, Anaheim, CA, August 1993. ACM.

[25] Michael J. McLennan. [incr Tcl] - Object-Oriented Programming in Tcl. AT&T Bell Laboratories, 1994. ftp from harbor.ecn.purdue.edu.

[26] Brad A. Meyers. State of the art in user interface software tools. *Advances in Human-Computer Interaction*, 4, 1992.

[27] Brad A. Meyers and Mary Beth Rosson. Survey on User Interface Programming. In *Proceedings of SIGCHI*. ACM, 1992.

[28] D. L. Miller-Karlow and E. J. Golin. vVHDL: A Visual Hardware Description Language. In *Proceedings of Workshop on Visual Languages*, pages 133–139, Seattle, WA, September 1992. IEEE.

[29] David A. Moon. Object-Oriented Programming with Flavors. In *Proceedings of ACM Conference on Object-Oriented Systems, Languages, and Applications (OOPSLA) 1986*. ACM, September 1986.

[30] Bonnie A. Nardi. *A Small Matter of Programming: Perspectives on End User Computing*. The MIT Press, 1993.

[31] Thomas J. Olsen et al. MAVIS: A Visual Environment for Active Computer Vision. In *Proceedings of Workshop on Visual Languages*, pages 170–176, Seattle, WA, September 1992. IEEE.

[32] John Ousterhout. Tcl: An Embeddable Command Language. *USENIX*, 1990.

[33] John Ousterhout. An X11 Toolkit Based on the Tcl Language. *USENIX*, 1991.

[34] Brent Phillips. A Distributed Programming System for Media Applications. SM Thesis Proposal, Department of Electrical Engineering and Computer Science, MIT, July 1994.

[35] Richard Stallman. *GNU Emacs Manual*. Free Software Foundation, March 1987. Sixth Edition, Version 18.

[36] William Stasior. Visual Processing for Seamless Interactive Computing. in MIT/LCS/TR-590, November 1993.

[37] Steven L. Tantimoto. Viva: A visual language for image processing. *Journal of Visual Languages and Computing*, 1(2), 1990.

[38] D. L. Tennenhouse, J. F. Adam, D. Carver, H. Houh, M. Ismert, C. Lindblad, W. Stasior, D. Wetherall, D. Bacher, and T. Chang. A Software-Oriented Approach to the Design of Media-Processing Environments. In *Proceedings of the International Conference on Multimedia Systems and Computing*, Boston, MA, May 1994. IEEE.

[39] C. Upson et al. The application visualization system: A computational environment for scientific visualization. *IEEE Computer Graphics and Applications*, pages 30–42, July 1989.

[40] C. Williams and J. Rasure. A visual language for image processing. In *IEEE Computer Society Workshop on Visual Languages*, Skokie, IL, 1990. IEEE Computer Society.

[41] S. Wray, T. Glauert, and A. Hopper. The Medusa Applications Environment. In *Proceedings of the International Conference on Multimedia Systems and Computing*, pages 265–273, Boston, MA, May 1994. IEEE.